Broadview
www.broadview.com.cn

Orange'S

一个操作系统的实现

于渊 著

《自己动手写操作系统》第2版

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Orange'S 一个操作系统的实现

- 翔实的文字，丰富的图表，清晰的代码，作者亲自用LaTeX排版，内容与形式并重。
- 人性化的代码组织，帮读者关注每节重点；完备的行号标识，便于读者对照实际代码。
- 专属网站和邮件列表（请访问http://www.osfromscratch.org/），方便读者交流。
- 立足实践层面，关注动手操作过程中的细节，一步一步教读者完成自己的操作系统：

最简单的Boot Sector
↓
由Boot Sector和Loader加载的内核
↓
实现一个和多个进程
↓
多控制台
↓
进程间通信
↓
轻巧的FS
↓
简单的MM
↓
自己的C运行时库
↓
运行自己的应用程序
↓
在真机运行自己的操作系统

网上订购：www.dearbook.com.cn
第二书店·第一服务

责任编辑：江 立
责任美编：李 玲

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

上架建议：操作系统

ISBN 978-7-121-08442-3

9 787121 084425 >

定价：69.00元（含光盘1张）

于渊自述——

性禀，好奇，涉猎甚广，然所精者少。遇所好之事，譬如着迷，必沉迷其中，恍恍然如痴如醉。

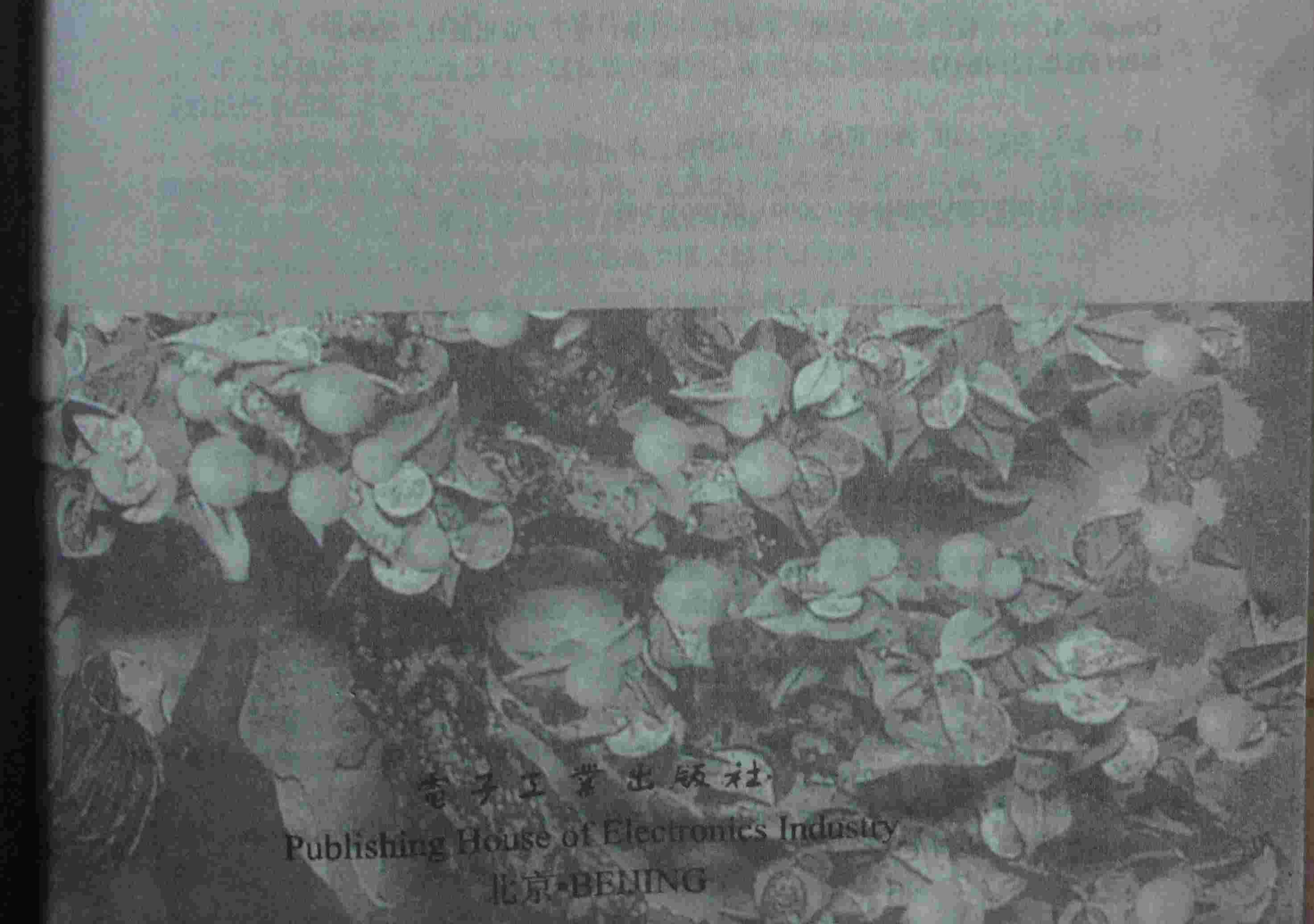
读书非多，然每读必思，偶有心得，自写其乐。遇知其所云者，欣然以为知音，必邀之共饮，所饮不必多，喜闻靴筒里铮铮之声，与谈笑交错，快意淋漓。

本性电执，喜钻研，求本质，不满于旁光掠影，凡可能之事，必躬亲而后快，以求甚解之心，究操作系统之原委，所亲为之，老叫数用，成等形，回顾此历程，自有一番甘苦，乃以此记录，与同道分享。虽有贻笑方家之虑，然凡此种种，皆切身之感受，所感所思，点滴皆为领悟，故心下坦然。若恰能为后来者借鉴一二，心当甚慰。

Orange'S

一个操作系统的实现

于渊 著



电子工业出版社
Publishing House of Electronics Industry
北京·BEIJING

内 容 简 介

本书从只有二十行的引导扇区代码出发，一步一步地向读者呈现一个操作系统框架的完成过程。书中不仅关注代码本身，同时关注完成这些代码的思路和过程。本书不同于其他的理论型书籍，而是提供给读者一个动手实践的路线图。读者可以根据路线图逐步完成各部分的功能，从而避免了一开始就面对整个操作系统数万行代码时的迷茫和挫败感。书中讲解了大量在开发操作系统中需注意的细节问题，这些细节不仅能使读者更深刻地认识操作系统的核心原理，而且使整个开发过程少走弯路。本书分上下两篇，共 11 章。其中每一章都以前一章的工作成果为基础，实现一项新的功能。而在章的内部，一项大的功能被分解成许多小的步骤，通过完成每个小的步骤，读者可以不断获得阶段性的成果，从而让整个开发过程变得轻松并且有趣。

本书适合各类程序员、程序开发爱好者阅读，也可作为高等院校操作系统课程的实践参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Orange’S：一个操作系统的实现 / 于渊著. —北京：电子工业出版社，2009.6
ISBN 978-7-121-08442-3

Ⅰ. O… Ⅱ. 于… Ⅲ. 操作系统 Ⅳ. TP316

中国版本图书馆 CIP 数据核字（2009）第 030117 号

责任编辑：江 立
印 刷：北京天宇星印刷厂
装 订：三河市皇庄路通装订厂
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×1092 1/16 印张：30.75 字数：700 千字
印 次：2009 年 6 月第 1 次印刷
印 数：5000 册 定价：69.00 元（含光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，
联系及邮购电话：（010）88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：（010）88258888。

做真正 Hacker 的乐趣——自己动手去实践

2004 年我听编辑说有个年轻人写了本《自己动手写操作系统》，第一反应是不可能，恐怕是翻译稿，写这种书籍是要考作者硬功夫的，不但需要深入掌握操作系统的原理，还需要实际动手写出原型。

历史上的 Linux 就是这么产生的，Linus Torvalds 当时是一名赫尔辛基大学计算机科学系的二年级学生，经常要用自己的电脑去访问大学主机上的新闻组和邮件，为了方便读写和下载文件，他自己编写了磁盘驱动程序和文件系统，这成为了 Linux 第一个内核的雏形。

我想中国有能力写出内核原型的程序员应该也有，但把这个题目写成一本书，感觉上不会有人愿意做这件事情，作者要花很多时间，加上主题比较硬，销售量不会太高，经济上回报有限。

但拿来文稿一看，整个编辑部大为惊艳，内容文笔俱佳，而且绝对原创，马上决定在《程序员》连载。2005 年博文视点出版的第一版也广受好评。

不过有很多读者还是质疑：现在软件编程主要领域是框架和应用，还需要了解操作系统底层吗？

经过四年的磨练成长，于渊又拿出第二版的书稿《Orange'S：一个操作系统的实现》，这本书是属于真正 Hacker 的。我虽然已经有多年不写代码了，但看这本书的时候，让我又重新感受到做程序员的乐趣：用代码建设属于自己的系统，让电脑听从自己的指令，对系统的每个部分都了如指掌。

黑客（hacker）实际是褒义词，维基百科的解释是喜欢用智力通过创造性方法来挑战脑力极限的人，特别是他们所感兴趣的领域，例如软件编程或电气工程。个人电脑、软件和互联网等划时代的产品都是黑客创造出来的，如苹果的 Apple 电脑、微软的 Basic 解释器、互联网的 Mosaic 浏览器。

回答前面读者的质疑，学软件编程并不需要看这本书，想成为优秀程序员和黑客的朋友，我强烈建议你花时间来阅读这本书，并亲自动手实践。正如于渊在本书结尾中所说“我们写自己的操作系统是出于一种好奇，或者说一种求知欲。我希望这样不停地‘过把瘾’能让这种好奇不停地延续”。

好奇心是动力的源泉，追究问题的本质是优秀黑客的必备素质，只有充分掌握了系统原理，才能在技术上游刃有余，才能有真正的创新和发展。中国需要更多真正的黑客，也希望更多的程序员能享受属于黑客的创造乐趣。

蒋涛

2009 年 4 月

作者自序

本书是《自己动手写操作系统》的第二版，通过一个具体的实例向读者呈现一个操作系统雏形的实现过程。有关操作系统的书籍资料可以找到很多，但是关注如何帮助读者实现一个试验性操作系统的书籍却不多见，本书便是从一个简单的引导扇区开始，讲述一个操作系统成长的故事，以作读者参考之用。

本书面向实践，通过具体实例教读者开发自己的操作系统。书中的步骤遵循由小到大、由浅入深的顺序，跟随这些步骤，读者可以由一个最简单的引导扇区开始，逐渐完善代码，扩充功能，最后形成一个小的操作系统。

本书不仅介绍操作系统的各要素，同时涉及开发操作系统需要的各个方面，比如如何建立开发环境、如何调试以及如何在虚拟机中运行等。书中的实例操作系统采用 IA32 作为默认平台，所以保护模式也作为必备知识储备收入书中，而这是传统的操作系统实践书籍经常忽略的。总之，只要是开发自己的操作系统中需要的知识，书中都尽量涉及，以便于读者参考。

众所周知，一个成型的操作系统往往非常复杂。如果考虑到操作系统作为软硬件桥梁的特殊地位，那么它可能看上去比一般的软件系统更难理解，因为其核心部分往往包含许多直接针对 CPU、内存和 I/O 端口的操作，它们夹杂在一片代码汪洋之中，显得更加晦涩。

我们有许多源代码公开的操作系统，可供随时下载和阅读，看上去好像让实现一个供自己把玩的微型操作系统变得容易很多，但事实往往不尽人意，因为这些代码动辄上万甚至几十几百万行，而且细节之间经常互相关联，要理解它们着实不易。我们有许多容易得到的操作系统教程，但读来好像总觉得跟我们有隔膜，不亲近。造成这些的根本原因，在于学习者一开始就面对一个完整的操作系统，或者面对前辈们积累了几十年的一系列理论成果。而无论作者多么擅长写作，读者多么聪明，或者代码多么优秀，要一个初学者理清其中的头绪都将是非常困难的。

我并非在此危言耸听，因为这曾经是我的亲身体会。当然，如果只是为了考试，几本操作系统理论书籍就足够了，你不需要对细节那么清楚。但如果是出于兴趣呢？如果你是想编写自己的操作系统呢？你会发现理论书籍好像一下子变得无用武之地，你会发现任何一个细节上的理解错误都可能导致自己辛辛苦苦编写的代码运行异常甚至崩溃。

我经历过这一切！我曾经翻遍了一本《操作系统：设计与实现》，也没有找到实现一个操作系统应该从何处着手。并不是这些书不好，也不是前人的代码不优秀，而是作为一无所知的初学者，我们所不了解的不仅是高居庙堂的理论

知识，还有让我们举步维艰的实践细节。

可能在这些教科书作者的眼里，操作的细节不属于课程的一部分，或者这些细节看上去太容易，根本不值一提，甚至作者认为这些属于所谓“经验”的一部分，约定俗成是由读者本人去摸索的。但是实际情况往往是，这些书中忽略掉的内容恰恰占去了一个初学者大部分的时间，甚至影响了学习的热情。

我至今仍记得当我开始编写自己的操作系统时所遭受的挫败感，那是一种不知道如何着手的无助的感觉。还好我坚持了下来，克服了各种困难，并完成了自己的操作系统雏形。

进而我想到，一定不只是我一个人对编写自己的操作系统怀有兴趣，也一定不只是我一个人在实践时遇到困难。或许我应该把自己的经历写下来，从而可以帮助跟我相似的后来者，就这样，我编写了本书的第一版，也就是《自己动手写操作系统》。我相信，如果你也对神奇的计算机世界充满好奇，并且希望通过自己编写操作系统的方式来了解背后发生的故事，那么你一定可以在这本书中得到一些帮助。而假如你真的因为我的书而重新燃起实践的热情，从而开始一段操作系统旅程，我将会感到非常高兴。

不过我得坦白，在写作《自己动手写操作系统》的时候，我并不敢期待它能引起多少反响，一方面因为操作系统并不是时尚的话题，另一方面我也是走在学习的路上，或许只是比读者早走了一小步而已。然而出乎我的意料，它面世后重印多次，甚至一度登上销量排行榜的榜首，这让我觉得它的确有一定的参考价值，我要借此机会感谢所有支持我的读者。

在我写作《自己动手写操作系统》的时候，并没有想过今天会有一个第二版。原因在于，我希望这本书是用来填补空白的，而不是重复去做别人已经做得很好的事情。所谓填补空白，具体说就是让像我一样的操作系统爱好者在读完本书之后，能够有信心去读其他比较流行的开源的操作系统代码，有能力从零开始自己动手写操作系统，而这个任务第一版已经完成了。

那么为什么我又写作了第二版呢？原因有几个方面。第一，虽然第一版未曾涉及的进程间通信、文件系统等内容在许多书中都有讲解，但阅读的时候还是感觉有语焉不详的通病，作者本人可能很清楚原委，但写得太简略，以至于读者看来未必清晰。第二，我自己想把这个圈画圆。第一版的书虽然完成了它的使命，但毕竟到书的结尾，读者看到的不是一个真正的操作系统，它没有文件系统，没有内存管理，什么也干不了。在第二版中，你将会看到，你已经可以通过交叉编译的方式为我们的实验性 OS 编写应用程序了，也就是说，它已经具备操作系统的基本功能，虽然仍然极其简陋，但第一个圈，毕竟是已经圆起来了。第三，实践类的操作系统书籍还是太少了，以至于你要想看看别人是怎么做的，除了读以《操作系统：设计与实现》为代表的极少数书籍之外，就是一头扎进源代码中，而结果有时相当令人气馁。我自己也气馁过，所以我在第二版中，仍然试图把话说细一点，把自己的经验拿出来分享。而且我选择我能想到的最精简的设计，以便让读者不至于陷入太多细节而无法看到全貌。我想这是本书可能具有的价值所在——简化的易懂的设计，还有尽量详细的文字。

在这一版中，内容被划分成上下两篇。上篇基本上是第一版的修订，只是做了一个调整，那便是在兼顾 Windows 和 Linux 两方面用户的基础上，默认在 Linux 下建立开发环境来编写我们的操作系统。至于这样做的原因，在本书第 2 章有比较详细的说明。当然，开发环境毕竟是第二位的，书中讲述的内容以及

涉及的代码跟第一版都是一致的。本书的下篇全部都是新鲜内容，主要是增加了进程间通信、文件系统和内存管理。跟第一版的做法相同，下篇仍然不仅关注结果，更加致力于将形成一个结果的过程呈现出来。与此同时，由于本书旨在分享和引路，所以尽可能地简化了设计，以便将最重要的部分凸显出来。读者将看到，一个操作系统的文件系统和内存管理可以简陋到什么程度。简陋不是缺点，对于我们初学者而言，正是需要从简陋入手。换言之，如果你已经对实现一个操作系统有了一定的经验，那么这本书可能不适合你。这本书适合从来没有编写过操作系统的初学者。

本书的排版是我用LaTeX自己完成的。在排版中我花了一些工夫，因为我希望读者购买的首先是一本易于阅读且赏心悦目的书，其次才是编写操作系统的方法。另外，书中列出的代码均由我自己编写的程序自动嵌入LaTeX源文件，从而严格保证书和光盘的一致性，读者可以根据文件名和行号方便地找到光盘中代码的准确位置。

此外，在第二版中还有一些小的变化。首先是操作系统的名字改变了，原因在于虽然我们的试验性 OS 从前辈们那里借鉴了很多东西，但其各个部分的设计（比如文件系统和内存管理）往往有其独特之处，所以我将原先的 Tinix（本意为 Try Minix）改成了新名字Orange'S（这个名字来自于我的妻子☺），以表示它们的不同。另外，书中的代码风格，有些地方也做了调整。

我想，虽然第二版有着这样那样的变化，但有一点没有变，那就是本书试图将我在编写自己操作系统的过程中的经验尽可能地告诉读者，同时尽可能将我当初的思路和编码过程呈现出来。很可能读者比我更聪明，有更好的解决问题的方法，但无论如何，我认为我自己的经验可以为读者所借鉴。如果真是如此，我将会非常欣慰。

在第二版的编写过程中，我同样要感谢许多人。感谢我的父母和爷爷对我的爱，并希望爷爷不要为我担心，写书是件辛苦的事，但同时也使我收获良多。爸爸在第二版的最后阶段帮我订正文字，这本书里有你的功劳。我要感谢博文视点的各位朋友，感谢郭老师的理解和支持，感谢李玲的辛勤工作，感谢江立和李冰，你们的高效让我非常钦佩。我还要感谢孟岩老师，你给我的鼓励我一直记在心里。我要感谢我的挚友郭洪桥，不仅仅因为你在技术上给我的帮助，更加因为你在精神上给我的支持。感谢我的同事和朋友张会昌，你在技术上的广度和深度总令我钦佩。另外，在第一版中帮助我的人，我要再次谢谢你们，因为没有第一版，也就没有第二版。

在所有人中我最应该感谢和最想感谢的，是我的妻子黄丹红，感谢你给我的所有建议，还有你帮我画的图。尤其是，当这本书在我预想的时间内没有完成的时候，当我遇到困难迟迟不能解决的时候，你总在一旁给我鼓励，在你那里，我从来都能感觉到一种温暖，我深知，如果没有你的支持，我无法坚持下来将书写完。谢谢你，这本书同样属于你。

跟第一版相比，这本书涉及的内容触及操作系统设计的更多方面，而由于笔者的水平实在有限，难免有纰漏甚至错误。如果读者有任何的问题、意见或建议，请登录 http://www.osfromscratch.org，让我们共同探讨，共同进步。

本书导读

这本书适合谁

本书是一本操作系统实践的技术书籍。对于操作系统技术感兴趣，想要亲身体验编写操作系统过程的实践主义者，以及 Minix、Linux 源代码爱好者，都可以在本书中得到实践中所需的知识和思路。

本书以“动手写”为指导思想，只要是跟“动手写”操作系统有关的知识，都作为介绍对象加以讨论，所以，从开发环境的搭建，到保护模式，再到 IBMPC 中有关芯片的知识，最后到操作系统本身的设计实现，都能在本文中找到相应介绍。所以如果你也想亲身实践的话，本书可以省去你在书店和互联网寻找相应资料的过程，使你的学习过程事半功倍。在读完本书后，你不但可以获得对于操作系统初步的感性认识，并且对 IBMPC 的接口、IA 架构之保护模式，以及操作系统整体上的框架都将会有一定程度的了解。

笔者相信，当你读完本书之后，如果再读那些纯理论性的操作系统书籍，所获得的体验将会完全不同，因为那些对你而言不再是海市蜃楼。

对于想阅读 Linux 源代码的操作系统爱好者，本书可以提供阅读前所必要的知识储备，而这些知识储备不但在本书中有完整的涉及，而且在很多 Linux 书籍中是没有提到的。

特别要提到的是，对于想通过阅读 Andrew S. Tanenbaum 和 Albert S. Woodhull 的《操作系统：设计与实现》来学习操作系统的读者，本书尤其适合作为你的引路书籍，因为它翔实地介绍了初学者入门时所必需的知识积累，而这些知识在《操作系统：设计与实现》一书中是没有涉及的，笔者本人是把这本书作为写操作系统的主要参考书籍之一，所以在本书中对它多有借鉴。

你需要什么技术基础

在本书中所用到的计算机语言只有两种：汇编和 C 语言。所以只要你具备汇编和 C 语言的经验，就可以阅读本书。除对操作系统常识性的了解（比如知道中断、进程等概念）之外，本书不假定读者具备其他任何经验。

如果你学习过操作系统的理论课程，你会发现本书是对于理论的吻合和补充。它是从实践的角度为你展现一幅操作系统画面。

书中涉及了 Intel CPU 保护模式、Linux 命令等内容，到时候会有尽可能清晰的讲解，如果笔者认为某些内容可以通过其他教材系统学习，会在书中加以说明。

另外，本书只涉及 Intel x86 平台。

统一思想——让我们在这些方面达成共识

道篇

让我们有效而愉快地学习

你大概依然记得在你亲自敲出第一个“Hello world”程序并运行成功时的喜悦，那样的成就感助燃了你对编写程序浓厚的兴趣。随后你不断地学习，每学到新的语法都迫不及待地在计算机上调试运行，在调试的过程中克服困难，学到新知，并获得新的成就感。

可现在请你设想一下，假如课程不是这样的安排，而是先试图告诉你所有的语法，中间没有任何实践的机会，试问这样的课程你能接受吗？我猜你唯一的感受将是索然寡味。

原因何在？只是因为你不再有因为不断实践而获得的源源不断的成就感。而成就感是学习过程中快乐的源泉，没有了成就感，学习的愉快程度将大打折扣，效果于是也将变得不容乐观。

每个人都希望有效而且愉快的学习过程，可不幸的是，我们见到的操作系统课程十之八九令我们失望，作者喋喋不休地讲述着进程管理存储管理I/O控制调度算法，可我们到头来也没有一点的感性认识。我们好像已经理解却又好像一无所知。很明显，没有成就感，一点也没有。笔者痛恨这样的学习过程，也决不会重蹈这样的覆辙，让读者获得成就感将是本书的灵魂。

其实这本书完全可以称作一本回忆录，记载了笔者从开始不知道保护模式为何物到最终形成一个小小 OS 的过程，这样的回忆录性质保证了章节的安排完全遵从操作的时间顺序，于是也就保证了每一步的可操作性，毫无疑问，顺着这样的思路走下来，每一章的成果都需要努力但又尽在眼前，步步为营是我们的战术，成就感是我们的宗旨。

我们将从二十行代码开始，让我们最简单的操作系统婴儿慢慢长大，变成一个翩翩少年，而其中的每一步，你都可以在书中的指导下自己完成，不仅仅是看到，而是自己做到！你将在不断的实践中获得不断的成就感，笔者真心希望在阅读本书的过程中，你的学习过程可以变得愉快而有效。

学习的过程应该是从感性到理性

在你没有登过泰山之前，无论书中怎样描写它的样子你都无法想象出它的真实面目，即便配有插图，你对它的了解仍会只是支离破碎。毫无疑问，一千本对泰山描述的书都比不上你一次登山的经历。文学家的描述可能是华丽而优美的，可这样的描述最终产生的效果可能是你非去亲自登泰山不可。反过来想

呢，假如你已经登过泰山，这样的经历产生的效果会是你想读尽天下描述泰山的书而后快吗？可能事实恰恰相反，你可能再也不想去看那些文字描述。

是啊，再好的讲述，又哪比得上亲身的体验？人们的认知规律本来如此，有了感性的认识，才能上升为理性的理论。反其道而行之只能是事倍功半。

如果操作系统是一座这样的大山，本书愿做你的导游，引领你进入它的门径。传统的操作系统书籍仅仅是给你讲述这座大山的故事，你只是在听讲，并没有身临其境，而随着这本书亲身体验，则好像置身于山门之内，你不但可以看见眼前的每一个细节，更是具有了走完整座大山的信心。

值得说明的是，本书旨在引路，不会带领你走完整座大山，但是有兴趣的读者完全可以在本书最终形成的框架的基础上容易地实现其他操作系统书籍中讲到的各种原理和算法，从而对操作系统有个从感性到理性的清醒认识。

暂时的错误并不可怕

当我们对一件事情的全貌没有很好理解的时候，很可能会对某一部分产生理解上的误差，这就是所谓的断章取义。很多时候断章取义是难免的，但是，在不断学习的过程中，我们会逐渐看到更多，了解更多，对原先事物的认识也会变得深刻甚至不同。

对于操作系统这样复杂的东西来说，要想了解所有的细节无疑是非常困难的，所以在实践的过程中，可能在很多地方，会有一些误解发生。这都没有关系，随着了解的深入，这些误解总会得到澄清，到时你会发现，自己对某一方面已经非常熟悉了，这时的成就感，一定会让你感到非常愉悦。

本书内容的安排遵从的是代码编写的时间顺序，它更像是一本开发日记，所以在书中一些中间过程不完美的产物被有意保留了下来，并会在以后的章节中对它们进行修改和完善，因为笔者认为，一些精妙的东西背后，一定隐藏着很多中间的产物，一个伟大的发现在很多情况下可能不是天才们刹那间的灵光一闪，背后也一定有着我们没有看到的不伟大甚至是谬误。笔者很想追寻前辈们的脚步，重寻他们当日的足迹。做到这一点无疑很难，但即便无法做到，只要能引起读者的一点思索，也是本书莫大的幸事。

挡住了去路的，往往不是大树，而是小藤

如果不是亲身去做，你可能永远都不知道，困难是什么。

就好像你买了一台功能超全的微波炉回家，研究完了整本说明书，踌躇满志想要烹饪的时候，却突然发现家里的油盐已经用完。而当时已经是晚上十一点，所有的商店都已经关门，你气急败坏，简直想摸起铁勺砸向无辜的微波炉。

研究说明书是没有错的，但是在没开始之前，你永远都想不到让你无法烹饪的原因居然是十块钱一瓶的油和一块钱一袋的更加微不足道的盐。你还以为困难是微波炉面板上密密麻麻的控制键盘。

其实做其他事情也是一样的，比如写一个操作系统，即便一个很小的可能受理论家们讥笑的操作系统雏形，仍然可能遇到一大堆你没有想过的问题，而

这些问题在传统的操作系统书籍中根本没有提到。所以唯一的办法，便是亲自去做，只有实践了，才知道是怎么回事。

术篇

用到什么再学什么

我们不是在考试，我们只是在为了自己的志趣而努力，所以就让我们忠于自己的喜好吧，不必为了考试而看完所有的章节，无论那是多么的乏味。让我们马上投入实践，遇到问题再图解决的办法。笔者非常推崇这样的学习方法：

实践 → 遇到问题 → 解决问题 → 再实践

因为我们知道我们为什么学习，所以我们才会非常投入；由于我们知道我们的目标是解决什么问题，所以我们才会非常专注；由于我们在实践中学习，所以我们才会非常高效。而最有趣的是，最终你会发现你并没有因为选择这样的学习方法而少学到什么，相反，你会发现你用更少的时间学到更多的东西，并且格外的扎实。

只要用心，就没有学不会的东西

笔者还清楚地记得刚刚下载完 Intel Architecture Software Developer Manual 那三个可怕的 PDF 文件时的心情，那时心里暗暗嘀咕，什么时候才能把这些东西读懂啊！可是突然有一天，当这些东西真的已经被基本读完的时候，我想起当初的畏惧，时间其实并没有过去多少。

所有的道理都是相通的，没有什么真正可怕，尤其是，我们所做的并非创造性的工作，所有的问题前人都曾经解决，所以我们更是无所畏惧，更何况我们不仅有书店，而且有互联网，动动手脚就能找到需要的资料，我们只要认真研究就够了。

所以当遇到困难时，请静下心来，慢慢研究，因为只要用心，就没有学不会的东西。

适当地囫囵吞枣

如果囫囵吞枣仅仅是学习的一个过程而非终点，那么它并不一定就是坏事。大家都应该听说过鲁迅先生学习英语的故事，他建议在阅读的过程中遇到不懂的内容可以忽略，等到过一段时间之后，这些问题会自然解决。

在本书中，有时候可能先列出一段代码，告诉你它能完成什么，这时你也可以大致读过，因为下面会有对它详细的解释。第一遍读它的时候，你只要了解大概就够了。

本书的原则

1. 宁可啰嗦一点，也不肯漏掉细节

在书中的有些地方，你可能觉得有些很“简单”的问题都被列了出来，甚至显得有些啰嗦，但笔者宁可让内容写得啰嗦点，因为笔者自己在读书的时候有一个体验，就是有时候一个问题怎么也想不通，经过很长时间终于弄明白的时候才发现原来是那么“简单”。可能作者认为它足够简单以至于可以跳过不提，但读者未必那么幸运一下子就弄清楚。

不过本书到后面的章节，如果涉及的细节是前面章节提到过的，就有意地略过了。举个非常简单的例子，开始时本书会提醒读者增加一个源文件之后不要忘记修改 Makefile，到后来就假定读者已经熟悉了这个步骤，可能就不再提及了。

2. 努力做到平易近人

笔者更喜欢把本书称作一本笔记或者学习日志，不仅仅是因为它基本是真实的学习过程的再现，而且笔者不想让它有任何居高临下甚至是晦涩神秘的感觉。如果有一个地方你觉得书中没有说清楚以至于你没有弄明白，请你告诉我，我会在以后做出改进。

3. 代码注重可读性但不注重效率

本书的代码力求简单易懂，在此过程中很少考虑运行的效率。一方面因为书中的代码仅仅供学习之用，暂时并不考虑实际用途；另一方面笔者认为当我们对操作系统足够了解之后再考虑效率的问题也不迟。

本书附带光盘说明

本书附带光盘中有本书用到的所有源代码。值得一提的是，其中不止包含完整的操作系统代码，还包含各个步骤的中间产物。换句话说，开发中每一步骤的代码，都可在光盘中单独文件夹中找到。举例说明，书的开篇介绍引导扇区，读者在相应文件夹中就只看到引导扇区的代码；第 9 章介绍文件系统，在相应文件夹中就不会包含第 10 章内存管理的代码。在任何一个步骤对应的文件夹中，都包含一个完整可编译运行的代码树，以方便读者试验之用。这样在学习的任何一个阶段，读者都可彻底了解阶段性成果，且不必担心受到自己还未学习的内容的影响，从而使学习不留死角。

在书的正文中引用的代码会标注出出自哪个文件。以“chapter5/b/bar.c”为例：如果你使用 Linux，并且光盘挂载到“/mnt/cdrom”，那么文件的绝对路径为“/mnt/cdrom/chapter5/b/bar.c”；如果你使用 Windows，并且光盘是 X: 盘，那么文件的绝对路径为“X:\chapter5\b\bar.c”。

目　录

上　篇

第 1 章　马上动手写一个最小的“操作系统”……2

1.1　准备工作……2
1.2　十分钟完成的操作系统……3
1.3　引导扇区……4
1.4　代码解释……4
1.5　水面下的冰山……6
1.6　回顾……7

第 2 章　搭建你的工作环境……8

2.1　虚拟计算机 Bochs……8
2.1.1　Bochs 初体验……8
2.1.2　Bochs 的安装……9
2.1.3　Bochs 的使用……10
2.1.4　用 Bochs 调试操作系统……12
2.2　QEMU……15
2.3　平台之争：Windows 还是*nix……16
2.4　GNU/Linux 下的开发环境……20
2.5　Windows 下的开发环境……22
2.6　总结……23

第 3 章　保护模式（Protect Mode）……25

3.1　认识保护模式……25
3.1.1　保护模式的运行环境……25
3.1.2　GDT（Global Descriptor Table）……29
……31

3.1.3 实模式到保护模式，不一般的 jmp ····· 33
3.1.4 描述符属性 ····· 35
3.2 保护模式进阶 ····· 38
3.2.1 海阔凭鱼跃 ····· 38
3.2.2 LDT（Local Descriptor Table） ····· 44
3.2.3 特权级概述 ····· 48
3.2.4 特权级转移 ····· 51
3.2.5 关于“保护”二字的一点思考 ····· 65
3.3 页式存储 ····· 65
3.3.1 分页机制概述 ····· 66
3.3.2 编写代码启动分页机制 ····· 67
3.3.3 PDE 和 PTE ····· 68
3.3.4 cr3 ····· 71
3.3.5 回头看代码 ····· 72
3.3.6 克勤克俭用内存 ····· 73
3.3.7 进一步体会分页机制 ····· 81
3.4 中断和异常 ····· 87
3.4.1 中断和异常机制 ····· 87
3.4.2 外部中断 ····· 90
3.4.3 编程操作 8259A ····· 91
3.4.4 建立 IDT ····· 94
3.4.5 实现一个中断 ····· 95
3.4.6 时钟中断试验 ····· 96
3.4.7 几点额外说明 ····· 98
3.5 保护模式下的 I/O ····· 100
3.5.1 IOPL ····· 100
3.5.2 I/O 许可位图（I/O Permission Bitmap） ····· 100
3.6 保护模式小结 ····· 101
第 4 章 让操作系统走进保护模式 ····· 102
4.1 突破 512 字节的限制 ····· 102
4.1.1 FAT12 ····· 103
4.1.2 DOS 可以识别的引导盘 ····· 108
4.1.3 一个最简单的 Loader ····· 108

4.1.4 加载 Loader 入内存 109
4.1.5 向 Loader 交出控制权 116
4.1.6 整理 boot.asm 116
4.2 保护模式下的“操作系统” 117
第 5 章 内核雏形 119
5.1 在 Linux 下用汇编写 Hello World 119
5.2 再进一步，汇编和 C 同步使用 120
5.3 ELF（Executable and Linkable Format） 123
5.4 从 Loader 到内核 127
5.4.1 用 Loader 加载 ELF 127
5.4.2 跳入保护模式 131
5.4.3 重新放置内核 137
5.4.4 向内核交出控制权 142
5.5 扩充内核 143
5.5.1 切换堆栈和 GDT 144
5.5.2 整理我们的文件夹 148
5.5.3 Makefile 149
5.5.4 添加中断处理 155
5.5.5 两点说明 168
5.6 小结 169
第 6 章 进程 171
6.1 迟到的进程 171
6.2 概述 171
6.2.1 进程介绍 171
6.2.2 未雨绸缪——形成进程的必要考虑 172
6.2.3 参考的代码 172
6.3 最简单的进程 173
6.3.1 简单进程的关键技术预测 174
6.3.2 第一步——ring0→ring1 175
6.3.3 第二步——丰富中断处理程序 178
6.4 多进程 189
6.4.1 添加一个进程体 200
..... 200

6.4.2 相关的变量和宏 ……200
6.4.3 进程表初始化代码扩充 ……202
6.4.4 LDT ……203
6.4.5 修改中断处理程序 ……203
6.4.6 添加一个任务的步骤总结 ……206
6.4.7 号外：Minix 的中断处理 ……207
6.4.8 代码回顾与整理 ……212
6.5 系统调用 ……220
6.5.1 实现一个简单的系统调用 ……222
6.5.2 get_ticks 的应用 ……227
6.6 进程调度 ……232
6.6.1 避免对称——进程的节奏感 ……232
6.6.2 优先级调度总结 ……240
第7章 输入/输出系统 ……242
7.1 键盘 ……242
7.1.1 从中断开始——键盘初体验 ……242
7.1.2 AT、PS/2 键盘 ……243
7.1.3 键盘敲击的过程 ……244
7.1.4 用数组表示扫描码 ……248
7.1.5 键盘输入缓冲区 ……251
7.1.6 用新加的任务处理键盘操作 ……253
7.1.7 解析扫描码 ……254
7.2 显示器 ……263
7.2.1 初识 TTY ……264
7.2.2 基本概念 ……264
7.2.3 寄存器 ……267
7.3 TTY 任务 ……270
7.3.1 TTY 任务框架的搭建 ……272
7.3.2 多控制台 ……277
7.3.3 完善键盘处理 ……281
7.3.4 TTY 任务总结 ……288
7.4 区分任务和用户进程 ……289
7.5 printf ……291

7.5.1 为进程指定 TTY 292
7.5.2 printf()的实现 292
7.5.3 系统调用 write() 294
7.5.4 使用 printf() 296

下　篇

第 8 章　进程间通信 300
8.1 微内核还是宏内核 302
8.1.1 Linux 的系统调用 303
8.1.2 Minix 的系统调用 305
8.1.3 我们的选择 306
8.2 IPC 306
8.3 实现 IPC 309
8.3.1 assert()和 panic() 313
8.3.2 msg_send()和 msg_receive() 321
8.3.3 增加消息机制之后的进程调度 322
8.4 使用 IPC 来替换系统调用 get_ticks 324
8.5 总结 325
第 9 章　文件系统 325
9.1 硬盘简介 325
9.2 硬盘操作的 I/O 端口 326
9.3 硬盘驱动程序 327
9.4 文件系统 337
9.5 硬盘分区表 338
9.6 设备号 344
9.7 用代码遍历所有分区 347
9.8 完善硬盘驱动程序 352
9.9 在硬盘上制作一个文件系统 355
9.9.1 文件系统涉及的数据结构 356
9.9.2 编码建立文件系统 358
9.10 创建文件 366
9.10.1 Linux 下的文件操作 366

9.10.2 文件描述符（file descriptor）……367
9.10.3 open()……369
9.11 创建文件所涉及的其他函数……377
9.11.1 strip_path()……377
9.11.2 search_file()……378
9.11.3 get_inode()和 sync_inode()……379
9.11.4 init_fs()……381
9.11.5 read_super_block()和 get_super_block()……382
9.12 关闭文件……383
9.13 查看已创建的文件……384
9.14 打开文件……386
9.15 读写文件……387
9.16 测试文件读写……390
9.17 文件系统调试……393
9.18 删除文件……395
9.19 插曲：奇怪的异常……401
9.20 为文件系统添加系统调用的步骤……403
9.21 将 TTY 纳入文件系统……404
9.22 改造 printf……411
9.23 总结……413
第 10 章 内存管理……414
10.1 fork……414
10.1.1 认识 fork……414
10.1.2 fork 前要做的工作（为 fork 所做的准备）……417
10.1.3 fork()库函数……421
10.1.4 MM……421
10.1.5 运行……427
10.2 exit 和 wait……427
10.3 exec……432
10.3.1 认识 exec……433
10.3.2 为自己的操作系统编写应用程序……434
10.3.3 “安装”应用程序……436
10.3.4 实现 exec……442

10.4 简单的 shell 447
10.5 总结 449
第 11 章 尾声 451
11.1 让 mkfs()只执行一次 451
11.2 从硬盘引导 455
11.2.1 编写硬盘引导扇区和硬盘版 loader 455
11.2.2 “安装” hdboot.bin 和 hdldr.bin 461
11.2.3 grub 461
11.2.4 小结 463
11.3 将 OS 安装到真实的计算机 465
11.3.1 准备工作 465
11.3.2 安装 Linux 466
11.3.3 编译源代码 466
11.3.4 开始安装 467
11.4 总结 467
参考文献 470

上篇

本书分上下两篇。上篇的内容与本书第一版（也即《自己动手写操作系统》）基本一致，共分七章。

第1章是个简单的开头，是我们操作系统之旅的第一步。虽然简单，但它的意义却非同小可，希望读者读完之后能够明白，一项看似繁重的工作，完全可以分解成相对容易的若干小步，而一旦你开始了，那么剩下的只不过是一点耐心而已。

第2章和第3章是准备工作，这是理论型的书籍最容易忽视的部分，然而它的重要性却不容忽视。再高明的厨师也需要有张案板，有个锅灶。编写自己操作系统的热情，决不应该被准备阶段的困难所浇灭。所以笔者用了不少的篇幅来介绍如何搭建自己的工作环境，以及讲述 Intel CPU 保护模式的基本概念和原理。当然，如果读者已经熟悉其中的内容，可以有选择性地跳过。

第4章和第5章介绍了如何写一个可用的引导扇区和用以加载内核的 Loader。这些内容也是传统操作系统书籍容易忽视的，因为引导扇区和内核加载器（Loader）严格来讲并不能算是操作系统的一部分。然而一个火箭不能没有发射架，要完成自己的操作系统，这两章内容必不可少。希望在读完这两章之后，读者能够彻底弄明白一个静态的内核是如何转变成一个运转中的操作系统的。

第6章介绍进程，这算得上是操作系统中最重要的概念。在这一章中我们将共同实现一个进程，接着是多个进程，并且让它们同时运行。在这一章中还引入了系统调用的概念，并实现了简单的进程调度。

第7章介绍输入/输出系统，引入了控制台的概念，主要涉及的是键盘和显示器的读写。通过这一章，读者可以了解操作系统与外部设备的通信方法。

总体来说，上篇的主要侧重点在于帮助读者开始一段旅程。如果读者跟随本书一起实践的话，那么到本篇结尾处，你应该已经初窥门径，并且具备进一步探索的信心了。

Verum ipsum factum. —— *Giambattista Vico*

1 马上动手写一个最小的“操作系统”

虽说万事开头难，但有时也未必。比如说，写一个有实用价值的操作系统是一项艰巨的工作，但一个最小的操作系统或许很容易就实现了。现在我们就来实现一个小得无法再小的“操作系统”，建议你跟随书中的介绍一起动手来做，你会发现不但很容易，而且很有趣。

1.1 准备工作

对于写程序，准备工作无非就是硬件和软件两方面，我们来看一下：

1. 硬件
   - 一台计算机（Linux 操作系统[1]或 Windows 操作系统均可）
   - 一张空白软盘
2. 软件
   - 汇编编译器 NASM

     NASM 最新版本可以从其官方网站获得[2]。此刻你可能会有疑问：这么多汇编编译器中，为什么选择 NASM？对于这一点本书后面会有解释。
   - 软盘绝对扇区读写工具

     在 Linux 下可使用 dd 命令，在 Windows 下则需要额外下载一个工具比如 rawrite[3]或者图形界面的 rawwritewin[4]。当然如果你愿意，也可以自己动手写一个“能用就好”的工具，并不是很复杂[5]。

---

[1]实际上 Linux 并非一种操作系统，而仅仅是操作系统的内核。在这里比较准确的说法是 GNU/Linux，本书提到的 Linux 泛指所有以 Linux 为内核的 GNU/Linux 操作系统。GNU 经常被人们遗忘，但它的贡献无论怎样夸大都不过分，这不仅仅是因为在你所使用的 GNU/Linux 中，GNU 软件的代码量是 Linux 内核的十倍，更加因为，如果没有以 Richard Stallman 为首的 GNU 项目倡导自由软件文化并为之付出艰辛努力，如今你我可能根本没有自由软件可用。本书将 GNU/Linux 简化为 Linux，仅仅是为表达方便，绝不是因为 GNU 这一字眼可有可无。

[2]NASM 的官方网站位于 http://sourceforge.net/projects/nasm。

[3]rawrite 可以在许多地方找到，比如http://ftp.debian.org/debian/tools/。

[4]下载地址为http://www.chrysocome.net/rawwrite。

[5]我们不需要太强大的软盘读写工具，只要能将 512 字节的数据写入软盘的第一个扇区就足够了。

1.2 十分钟完成的操作系统

你相不相信，一个“操作系统”的代码可以只有不到20行？请看代码1.1。

代码 1.1 chapter1/a/boot.asm

```
        org     07c00h                  ; 告诉编译器程序加载到7c00处
        mov     ax, cs
        mov     ds, ax
        mov     es, ax
        call    DispStr                 ; 调用显示字符串例程
        jmp     $                       ; 无限循环
DispStr:
        mov     ax, BootMessage
        mov     bp, ax                  ; ES:BP = 串地址
        mov     cx, 16                  ; CX = 串长度
        mov     ax, 01301h              ; AH = 13,  AL = 01h
        mov     bx, 000ch               ; 页号为0(BH = 0) 黑底红字(BL = 0Ch,高亮)
        mov     dl, 0
        int     10h                     ; 10h 号中断
        ret
BootMessage:            db      "Hello, OS world!"
times   510-($-$$)      db      0       ; 填充剩下的空间，使生成的二进制代码恰好为512字节
dw      0xaa55                          ; 结束标志
```

把这段代码用 NASM 编译一下：

```
▷ nasm boot.asm -o boot.bin
```

我们就得到了一个 512 字节的 boot.bin，让我们使用软盘绝对扇区读写工具将这个文件写到一张空白软盘的第一个扇区。在 Linux 下可以这样做[6]：

```
▷ dd if=boot.bin of=/dev/fd0 bs=512 count=1
```

在 Windows 下可以这样做[7]：

```
▷ rawrite2.exe -f boot.bin -d A
```

好了，你的第一个“操作系统”就已经完成了。这张软盘已经是一张引导盘了。

把它放到你的软驱中重新启动计算机，从软盘引导，你看到了什么？

计算机显示出你的字符串了！红色的“Hello, OS world!”，多么奇妙啊，你的“操作系统”在运行了！

如果使用虚拟机比如 Bochs 的话（下文中将会有关于 Bochs 的详细介绍），你应该能看到如图1.1所示的画面[8]。

这真的是太棒了，虽然你知道它有多么简陋，但是，毕竟你已经制作了一个可以引导的软盘了，而且所有工作都是你亲手独立完成的！

---

[6]取决于硬件环境和具体的 Linux 发行版，此命令可能稍有不同。
[7]rawrite 有多个版本，此处选用的是 2.0 版。
[8]画面看上去有点乱，因为我们打印字符前并未进行任何的清屏操作。

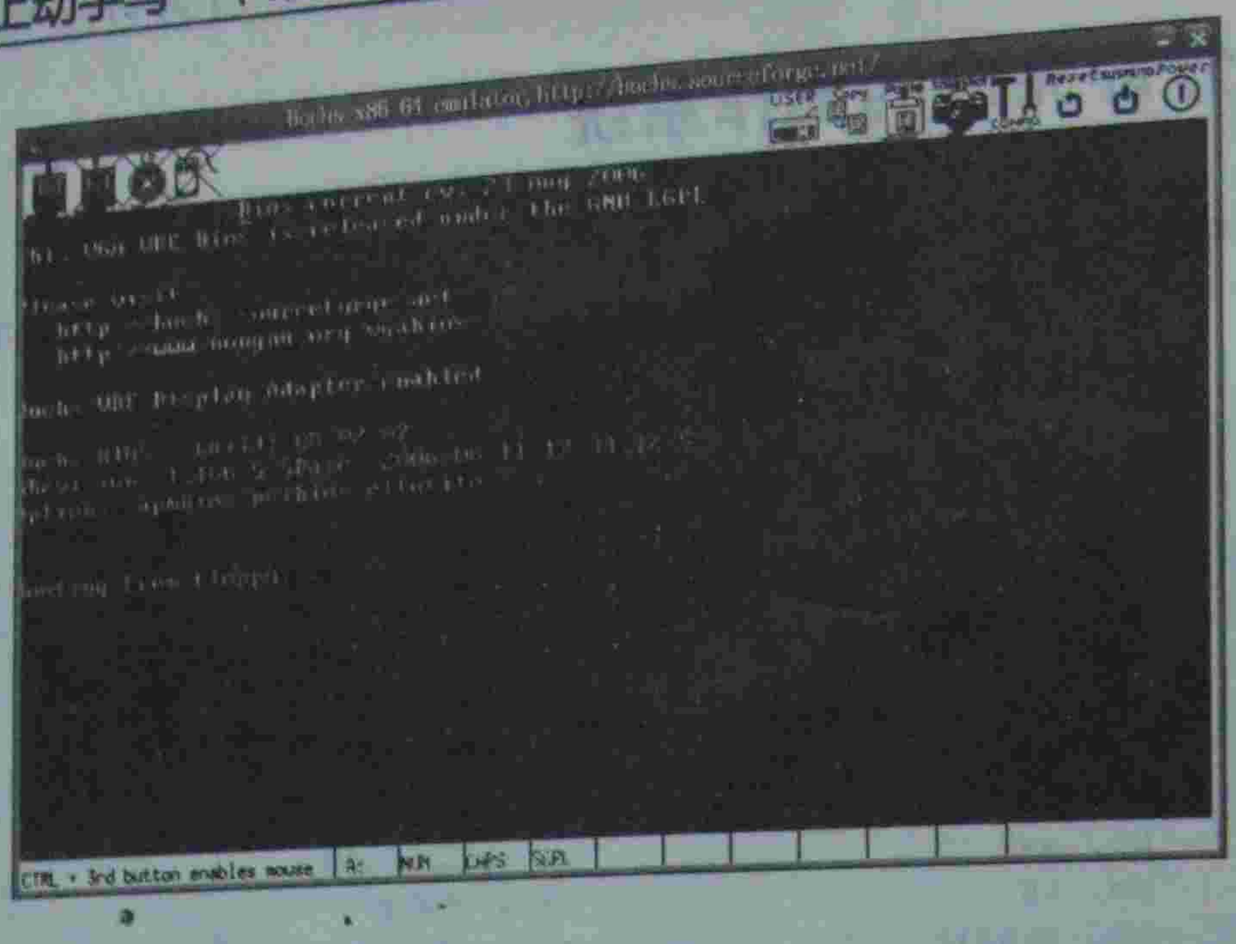


图 1.1 最小的“操作系统”

1.3 引导扇区

你可能还没有从刚刚的兴奋中走出来，可是我不得不告诉你，实际上，你刚刚所完成的并不是一个完整的操作系统，而仅仅是一个最最简单的引导扇区（Boot Sector）。然而不管我们完成的是什么，至少，它是直接在裸机上运行的，不依赖于任何其他软件，所以，这和我们平时所编写的应用软件有本质的区别。它不是操作系统，但已经具备了操作系统的一个特性。

我们知道，当计算机电源被打开时，它会先进行加电自检（POST），然后寻找启动盘，如果是选择从软盘启动，计算机就会检查软盘的 0 面 0 磁道 1 扇区，如果发现它以 0xAA55[9]结束，则 BIOS 认为它是一个引导扇区。当然，一个正确的引导扇区除了以 0xAA55 结束之外，还应该包含一段少于 512 字节的执行码。

好了，一旦 BIOS 发现了引导扇区，就会将这 512 字节的内容装载到内存地址 0000:7c00 处，然后跳转到 0000:7c00 处将控制权彻底交给这段引导代码。到此为止，计算机不再由 BIOS 中固有的程序来控制，而变成由操作系统的一部分来控制。

现在，你可能明白了为什么在那段代码的第一行会出现“org 07c00”这样的代码。没错，这行代码就是告诉编译器，将来我们的这段程序要被加载到内存偏移地址 0x7c00 处。好了，下面将对代码的其他部分进行详细解释。

1.4 代码解释

其实程序的主体框架只是第2行到第6行这么一点点而已，其中调用了一个显示字符串的子程序。程序的第2、3、4行是 3 个 mov 指令，使 ds 和 es 两个段

[9]假如把此扇区看做一个字符数组sector[]，那么此结束标志相当于sector[510]==0x55，且sector[511]==0xAA。

寄存器指向与 cs 相同的段，以便在以后进行数据操作的时候能定位到正确的位置。第5行调用子程序显示字符串，然后jmp $让程序无限循环下去。

可能有很多人开始学汇编时用的都是 MASM，其实 NASM 的格式跟 MASM 总体上是差不多的，在这段程序中，值得说明的地方有以下几点：

1. 方括号 [] 的使用

   在 NASM 中，任何不被方括号 [] 括起来的标签或变量名都被认为是地址，访问标签中的内容必须使用 []。所以，

```
mov ax, BootMessage
```

   将会把“Hello, OS world!”这个字符串的首地址传给寄存器 ax。又比如，如果有：

```
foo dw 1
```

   则mov ax, foo将把foo的地址传给ax，而 mov bx, [foo]将把bx的值赋为 1。

   实际上，在 NASM 中，变量和标签是一样的，也就是说：

```
foo dw 1 等价于 foo: dw 1
```

   而且你会发现，Offset 这个关键字在 NASM 也是不需要的。因为不加方括号时表示的就是 Offset。

   笔者认为这是 NASM 的一大优点，要地址就不加方括号，也不必额外地用什么 Offset，想要访问地址中的内容就必须加上方括号。代码规则非常鲜明，一目了然。

2. 关于 $ 和 $$

   $ 表示当前行被汇编后的地址。这好像不太容易理解，不要紧，我们把刚刚生成的二进制代码文件反汇编来看看：

```
▷ ndisasmw -o 0x7c00 boot.bin >> disboot.asm
```

   打开 disboot.asm，你会发现这样一行：

```
00007C09  EBFE                jmp short 0x7c09
```

   明白了吧，$ 在这里的意思原来就是 0x7c09。

   那么 $$ 表示什么呢？它表示一个节（section[10]）的开始处被汇编后的地址。在这里，我们的程序只有 1 个节，所以，$$ 实际上就表示程序被编译后的开始地址，也就是 0x7c00。

在写程序的过程中，$-$$可能会被经常用到，它表示本行距离程序开始处的相对距离。现在，你应该明白510-($-$$)表示什么意思了吧？

times 510-($-$$) db 0表示将0这个字节重复510-($-$$)遍，也就是在剩下的空间中不停地填充0，直到程序有 510 字节为止。这样，加上结束标志 0xAA55 占用的 2 字节，恰好是 512 字节。

[10]注意：这里的 section 属于 NASM 规范的一部分，表示一段代码，关于它和 $$ 更详细的注解请参考 NASM 联机技术文档。

1.5 水面下的冰山

即便是非常袖珍的程序，也有可能遇到不能正确运行的情况，对此你一定并不惊讶，谁都可能少写一个标点，或者在一个小小的逻辑问题上犯迷糊。好在我们可以调试，通过调试，可以发现错误，让程序日臻完美。但是对于操作系统这样的特殊程序，我们没有办法用普通的调试工具来调试。可是，哪怕一个小小的引导扇区，我们也没有十足的把握一次就写好，那么，遇到不能正确运行的时候该怎么办呢？在屏幕上没有看到我们所要的东西，甚至于机器一下子重启了，你该如何是好呢？

每一个问题都是一把锁，你要相信，世界上一定存在一把钥匙可以打开这把锁。你也一定能找到这把钥匙。

一个引导扇区代码可能只有20行，如果Copy&Paste的话，10秒钟就搞定了，即便自己敲键盘抄一遍下来，也用不了10分钟。可是，在遇到一个问题时，如果不小心犯了小错，自己到运行时才发现，你可能不得不花费10个10分钟甚至更长时间来解决它。笔者把这20行的程序称做水面以上的冰山，而把你花了数小时的时间做的工作称做水面下的冰山。

古人云：“授之以鱼，不如授之以渔。”本书将努力将冰山下的部分展示给读者。这些都是笔者经历了痛苦的摸索后的一些心得，这些方法可能不是最好的，但至少可以给你提供一个参考。

好了，就以我们刚刚完成的引导扇区为例，你可以想像得到，将来我们一定会对这20行进行扩充，最后得到200行甚至更多的代码，我们总得想一个办法，让它调试起来容易一些。

其实很容易，因为有Bochs，我们前面提到的虚拟机，它本身就可以作为调试器使用[11]。请允许我再次卖一个关子，留待下文分解。但是请相信，调试一个引导扇区——甚至是一个操作系统，在工具的帮助下都不是很困难的事情。只不过跟调试一个普通的应用程序相比，细节上难免有一些不同。

如果你使用的是Windows，还有个可选的方法能够帮助调试，做法也很简单，就是把“org 07c00h”这一行改成“org 0100h”就可以编译成一个.COM文件让它在DOS下运行了。我们来试一试，首先把07c00h改成0100h，编译：

```
▷ nasm boot.asm -o boot.com
```

好了，一个易于执行和调试的引导扇区就制作完毕了。如果你是以DOS或者Windows为平台学习的编程，那么调试.COM文件一定是你拿手的工作，比如使用Turbo Debugger。

调试完之后要放到软盘上进行试验，我们需要再把0100h改成07c00h，这样改来改去比较麻烦，好在强大的NASM给我们提供了预编译宏，把原来的“org 07c00h”一行变成许多行即可：

代码1.2 chapter1/b/boot.asm

```
1 ;%define _BOOT_DEBUG_            ; 制作 Boot Sector 时一定将此行注释掉!
2                                  ; 去掉此行注释后可做成.COM文件易于调试:
```

[11]如果你实在性急，请翻到第12页第2.1.4节看一下具体怎么调试。

```
3                                       ;    nasm Boot.asm -o Boot.com
4
5   %ifdef  _BOOT_DEBUG_
6           org  0100h                  ; 调试状态，做成 .COM 文件，可调试
7   %else
8           org  07c00h                 ; BIOS 将把 Boot Sector 加载到 0:7C00 处
9   %endif
```

这样一来，如果我们想要调试，就让第一行有效，想要做引导扇区时，将它注释掉就可以了。

这里的预编译命令跟C语言差不多，就不用多解释了。

至此，你不但已经学会了如何写一个简单的引导扇区，更知道了如何进行调试。这就好比从石器时代走到了铁器时代，宽阔的道路展现在眼前，运用工具，我们有信心将引导扇区不断扩充，让它变成一个真正的操作系统的一部分。

1.6 回顾

让我们再回过头看看刚才那段代码吧，大部分代码你一定已经读懂了。如果你还是一个NASM新手，可能并不是对所有的细节都那么清晰。但是，毕竟你已经发现，原来可以如此容易地迈出写操作系统的第一步。

是啊，这是个并不十分困难的开头，如果你也这样认为，就请带上百倍的信心，以及一直以来想要探索OS奥秘的热情，随我一起出发吧！

They who can give up essential liberty to obtain a little temporary safety, deserve neither liberty nor safety.

—— Benjamin Franklin

2 搭建你的工作环境

我知道，现在你已经开始摩拳擦掌准备大干一场了，因为你发现，开头并不是那么难的。你可能想到了 Linus，或许他在写出第一个引导扇区并调试成功时也是同样的激动不已；你可能在想，有一天，我也要写出一个 Linux 那样伟大的操作系统！是的，这一切都有可能，因为一切伟大必定是从平凡开始的。我知道此刻你踌躇满志，已经迫不及待要进入操作系统的殿堂。

可是先不要着急，古人云：“工欲善其事，必先利其器”，你可能已经发现，如果每次我们编译好的东西都要写到软盘上，再重启计算机，不但费时费力，对自己的爱机简直是一种蹂躏。你一定不会满足于这样的现状，还好，我们有如此多的工具，比如前面提到过的 Bochs。

在介绍 Bochs 及其他工具之前，需要说明一点，这些工具并不是不可或缺的，介绍它们仅仅是为读者提供一些可供选择的方法，用以搭建自己的工作环境。但是，这并不代表这一章就不重要，因为得心应手的工具不但可以愉悦身心，并且可以起到让工作事半功倍的功效。

下面就从 Bochs 开始介绍。

2.1 虚拟计算机 Bochs

即便没有听说过虚拟计算机，你至少应该听说过磁盘映像。如果经历过 DOS 时代，你可能就曾经用 HD-COPY 把一张软盘做成一个 .IMG 文件，或者把一个 .IMG 文件恢复成一张软盘。虚拟计算机相当于此概念的外延，它与映像文件的关系就相当于计算机与磁盘。简单来讲，它相当于运行在计算机内的小计算机。

2.1.1 Bochs 初体验

我们先来看看 Bochs 是什么样子的，请看图2.1和图2.2这两个屏幕截图。

要看清楚哦，你看到的不是显示器，仅仅是窗口而已。如果你是第一次接触“虚拟机”这个东西的话，一定会感到很惊讶，你会惊叹：“啊，像真的一样！”没错，像真的一样，不过窗口的标题栏一行“Bochs x86-64 emulator”明白

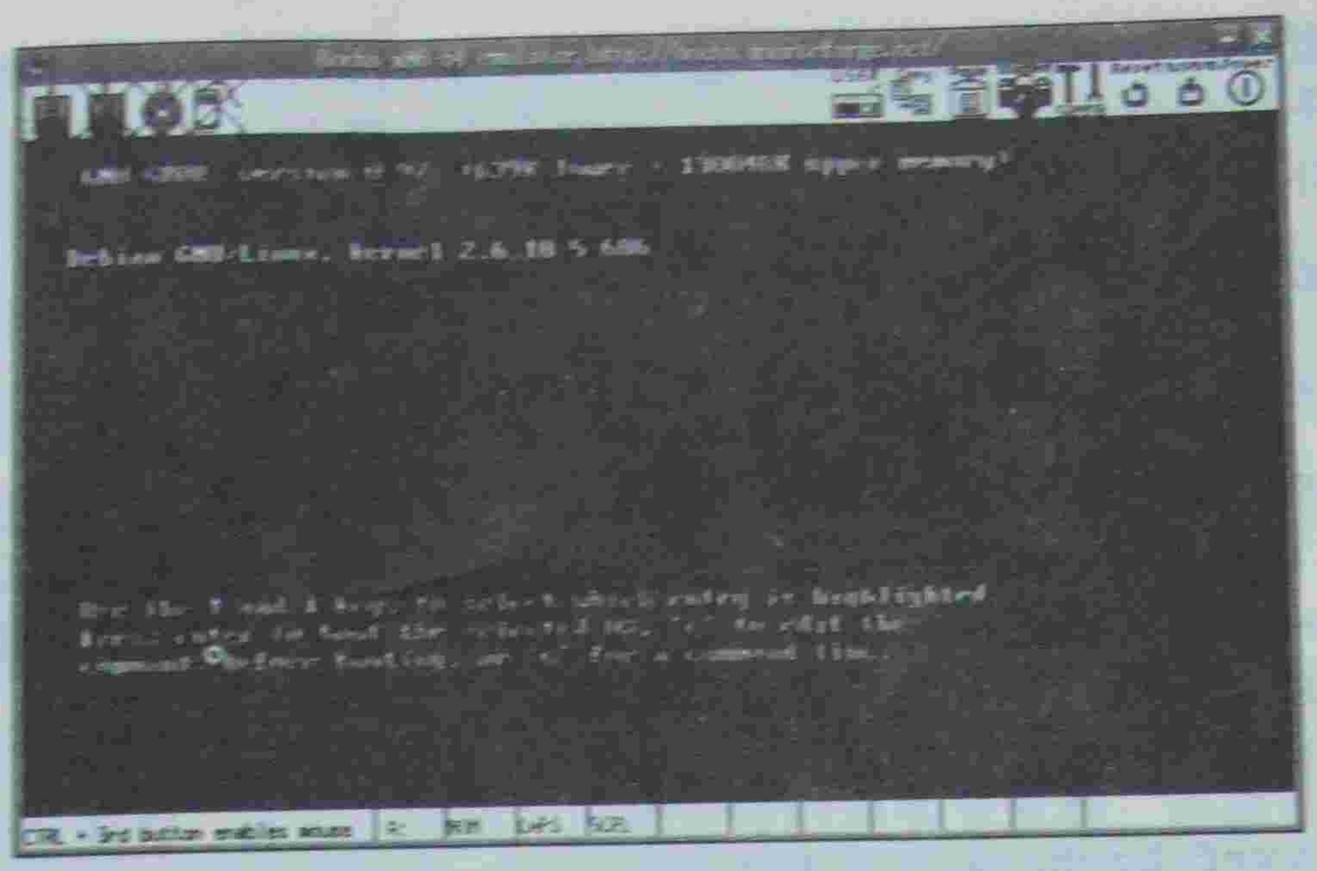


图 2.1 Bochs 中的 grub

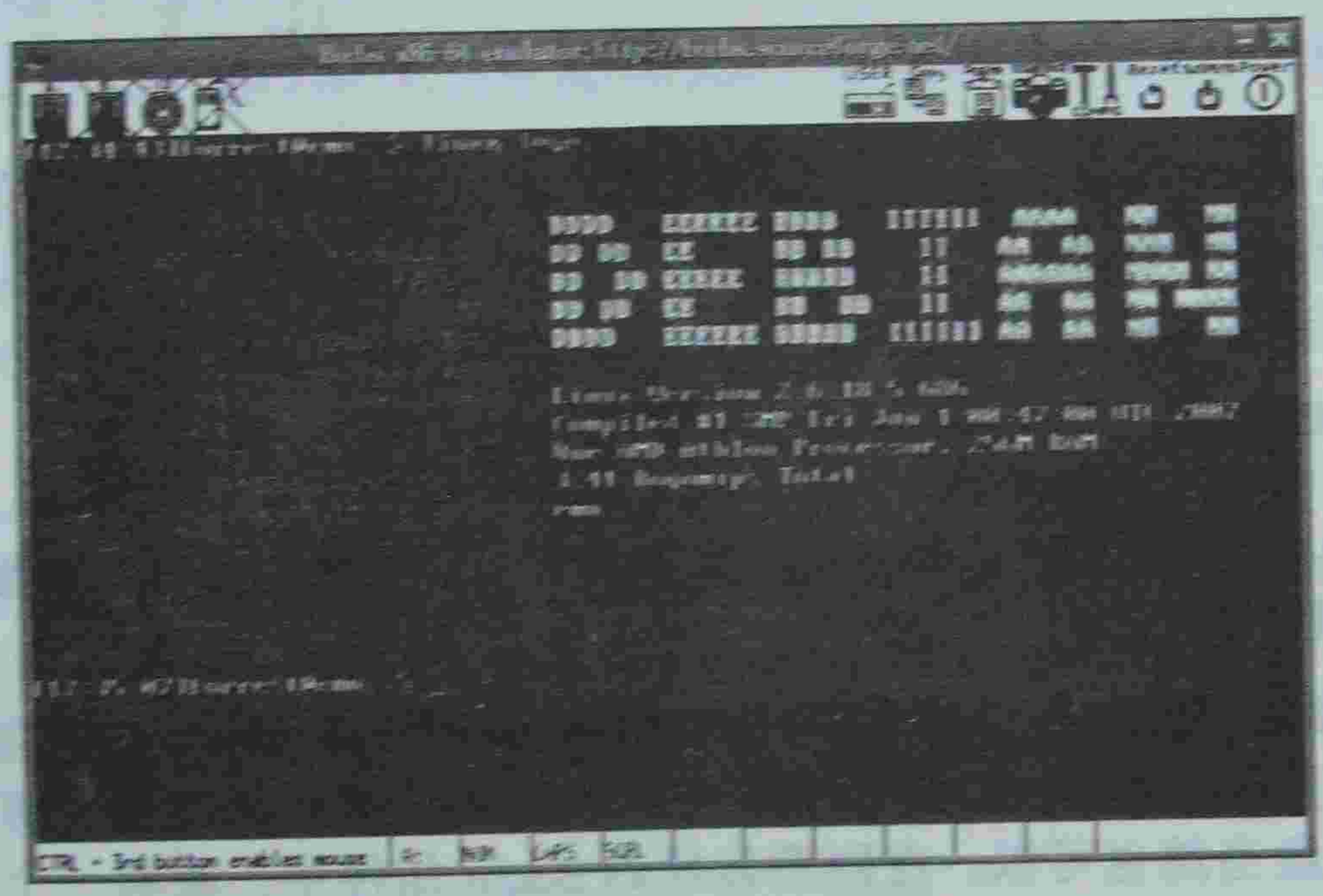


图 2.2 Bochs 中的 Linux

无误地告诉我们，这仅仅是个“emulator”——模拟器而已。在本书中我们把这种模拟器称为虚拟机，因为这个词使用得更广泛一些。不管是模拟还是虚拟，我们要的就是它，有了它，我们不再需要频繁地重启计算机，即便程序有严重的问题，也丝毫伤害不到你的爱机。更加方便的是，你可以用这个虚拟机来进行操作系统的调试，在它面前，你就好像是上帝，你可以随时让时间停住，然后钻进这台计算机的内部，CPU 的寄存器、内存、硬盘，一切的一切都尽收眼底。这正是进行操作系统的开发实验所需要的。

好了，既然 Bochs 这么好，我们就来看看如何安装，以及如何使用。

2.1.2 Bochs 的安装

就像大部分软件一样，在不同的操作系统里面安装 Bochs 的过程是不同的，在 Windows 中，最方便的方法就是从 Bochs 的官方网站获取安装程序来安装

(安装时不妨将“DLX Linux Demo”选中，这样你可以参考它的配置文件)。在Linux中，不同的发行版（distribution）处理方法可能不同。比如，如果你用的是Debian GNU/Linux或其近亲（比如Ubuntu），可以使用这样的命令：

```
▷ sudo apt-get install vgabios bochs bochs-x bximage
```

敲入这样一行命令，不一会儿就装好了。

很多Linux发行版都有自己的包管理机制，比如上面这行命令就是使用了Debian的包管理命令，不过这样安装虽然省事，但有个缺点不得不说，就是默认安装的Bochs很可能是没有调试功能的，这显然不能满足我们的需要，所以最好的方法还是从源代码安装，源代码同样位于Bochs的官方网站[1]，假设你下载的版本是2.3.5，那么安装过程差不多是这样的：

```
▷ tar vxzf bochs-2.3.5.tar.gz
▷ cd bochs-2.3.5
▷ ./configure --enable-debugger --enable-disasm
▷ make
▷ sudo make install
```

注意“./configure”之后的参数便是打开调试功能的开关。在安装过程中，如果遇到任何困难，不要惊慌，其官方网站上有详细的安装说明。

2.1.3 Bochs的使用

好了，Bochs已经安装完毕，是时候来揭晓第1章的谜底了，下面我们就一步步来说明图1.1的画面是怎样来的。

在第1章我们提到过，硬件方面需要的是一台计算机和一张空白软盘，现在计算机有了——就是刚刚安装好的Bochs，那么软盘呢？既然计算机都可以“虚拟”，软盘当然也可以。在刚刚装好的Bochs组件中，就有一个工具叫做bximage，它不但可以生成虚拟软盘，还能生成虚拟硬盘，我们也称它们为磁盘映像。创建一个软盘映像的过程如下所示：

```
▷ bximage
========================================================================
                                bximage
                  Disk Image Creation Tool for Bochs
        $Id: bximage.c,v 1.32 2006/06/16 07:29:33 vruppert Exp $
========================================================================

Do you want to create a floppy disk image or a hard disk image?
Please type hd or fd. [hd] fd ↩

Choose the size of floppy disk image to create, in megabytes.
Please type 0.16, 0.18, 0.32, 0.36, 0.72, 1.2, 1.44, 1.68, 1.72, or 2.88.
[1.44] ↩
I will create a floppy image with
  cyl=80
  heads=2
  sectors per track=18
  total sectors=2880
  total bytes=1474560
```

[1]实际上通过命令行也可以获取源代码，只不过通常不是最新的，在此不做介绍。

```
What should I name the image?
[a.img] ↩

Writing: [] Done.

I wrote 1474560 bytes to a.img.

The following line should appear in your bochsrc:
  floppya: image="a.img", status=inserted
```

凡是有↩记号的地方，都是 bximage 提示输入的地方，如果你想使用默认值，直接按回车键就可以。在这里我们只有一个地方没有使用默认值，就是被问到创建硬盘还是软盘映像的时候，我们输入了“fd”。

完成这一步骤之后，当前目录下就多了一个 a.img，这便是我们的软盘映像了。所谓映像者，你可以理解为原始设备的逐字节复制，也就是说，软盘的第M个字节对应映像文件的第M个字节。

现在我们已经有了“计算机”，也有了“软盘”，是时候将引导扇区写进软盘了。我们使用 dd 命令[2]：

```
▷ dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
```

注意这里多用了一个参数“conv=notrunc”，如果不用它的话软盘映像文件 a.img 会被截断（truncated），因为 boot.bin 比 a.img 要小。第 1 章中我们使用这个命令时不需要此参数，因为真实的软盘不可能被“截断”——真的和假的总是会有一点区别。

现在一切准备就绪，该打开电源启动了。可电源在哪儿呢？不要慌，我们还剩一样重要的东西没有介绍，那就是 Bochs 的配置文件。为什么要有配置文件呢？因为你需要告诉 Bochs，你希望你的虚拟机是什么样子的。比如，内存多大啊、硬盘映像和软盘映像都是哪些文件啊等内容。不用怕，这配置文件也没什么难的，代码2.1就是一个 Linux 下的典型例子。

代码 2.1 bochsrc 示例

```
1  ##############################################################
2  # Configuration file for Bochs
3  ##############################################################
4
5  # how much memory the emulated machine will have
6  megs: 32
7
8  # filename of ROM images
9  romimage: file=/usr/share/bochs/BIOS-bochs-latest
10 vgaromimage: file=/usr/share/vgabios/vgabios.bin
11
12 # what disk images will be used
13 floppya: 1_44=a.img, status=inserted
14
15 # choose the boot disk.
16 boot: floppy
17
18 # where do we send log messages?
19 log: bochsout.txt
20
```

[2]如果你用 Windows，那么使用 Linux 常用命令需要额外一些劳动，比如安装一个 Cygwin，或者下载某个工具的 Windows 版本。在这里你可以简单下载一个“dd for Windows”，其下载地址为http://www.chrysocome.net/dd。

```
# disable the mouse
mouse: enabled=0

# enable key mapping, using US layout as default.
keyboard_mapping: enabled=1, map=/usr/share/bochs/keymaps/x11-pc-us.map
```

可以看到，这个配置文件本来就不长，除去注释之后内容就更少了，而且很容易理解，字面上稍微不容易理解的只有 romimage 和 vgaromimage[3]，它们指定的文件对应的其实就是真实机器的 BIOS 和 VGA BIOS，读者自己操作的时候要确保它们的路径是正确的，不然过一会儿虚拟机启动时可能会被提示“couldn’t open ROM image file”。读者还要注意 floppya 一项，它指定我们使用哪个文件作为软盘映像。

如果你在 Windows 下的话，romimage 和 vgaromimage 两项指定的文件应该是安装目录下的BIOS-bochs-latest和VGABIOS-lgpl-latest。当然，最保险的方法是参考安装程序自带的 DLX linux 的配置文件，将其稍作修改即可。

好了，现在一切准备就绪，是时候启动了，输入命令：

```
▷ bochs -f bochsrc
```

一个回车[4]，你想要的画面就呈现在眼前了。是不是很有趣呢？

顺便告诉你个窍门，如果你输入一个不带任何参数的 Bochs 并执行之，那么 Bochs 将在当前目录顺序寻找以下文件作为默认配置文件：

- .bochsrc
- bochsrc
- bochsrc.txt
- bochsrc.bxrc（仅对 Windows 有效）

所以刚才我们的“`-f bochsrc`”参数其实是可以省略的。读者在给配置文件命名时不妨从这些文件里选一个，这样可以省去许多输入命令的时间。

此外，Bochs 的配置文件还有许多其他选项，读者如果想详细了解的话，可以到其主页上看一看。由于本书中所用到的选项有限，在此不一一介绍。

2.1.4 用 Bochs 调试操作系统

如果单是需要一个虚拟机的话，你有许许多多的选择，本书下文也会对其他虚拟机有所介绍，之所以 Bochs 称为我们的首选，最重要的还在于它的调试功能。

假设你正在运行一个有调试功能的 Bochs，那么启动后，你会看到控制台出现若干选项，默认选项为“6. Begin simulation”，所以直接按回车键，Bochs 就

[3]Bochs 使用的 vgaromimage 来自于 vgabios 项目，如果读者感兴趣，可以去它的主页看看：http://www.nongnu.org/vgabios/。

[4]如果你正在使用的是自己编译的有调试功能的 Bochs，回车后还需要再一次回车，并在出现 Bochs 提示符之后输入“c”，再次回车。不要被这些输入吓怕了，下文有妙计可以让你不必总是这么辛苦。

启动了，不过既然是可调试的，Bochs 并没有急于让虚拟机进入运转状态，而是继续出现一个提示符，等待你的输入，这时，你就可以尽情操纵你的虚拟机了。

还是以我们那个最轻巧的引导扇区为例，假如你想让它一步步地执行，可以先在07c00h处设一个断点——引导扇区就是从这里开始执行的，所以这里就是我们的入口地址——然后单步执行，就好像所有其他调试工具一样。在任何时刻，你都可以查看 CPU 寄存器，或者查看某个内存地址处的内容。下面我就来模拟一下这个过程：

```
... ... ...
Next at t=0
(0) [0xfffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b          ; ea5be000f0
<bochs:1> b 0x7c00↩
<bochs:2> c↩
(0) Breakpoint 1, 0x00007c00 in ?? ()
Next at t=886152
(0) [0x00007c00] 0000:7c00 (unk. ctxt): mov ax, cs                 ; 8cc8
<bochs:3> dump_cpu↩
eax:0x0fffaa55, ebx:0x00000000, ecx:0x00120001, edx:0x00000000
ebp:0x00000000, esp:0x0000fffe, esi:0x000088d2, edi:0x0000ffde
eip:0x00007c00, eflags:0x00000282, inhibit_mask:0
cs:s=0x0000, dl=0x0000ffff, dh=0x00009b00, valid=1
ss:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=7
ds:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
es:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
fs:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
gs:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
ldtr:s=0x0000, dl=0x0000ffff, dh=0x00008200, valid=1
tr:s=0x0000, dl=0x0000ffff, dh=0x00008300, valid=1
gdtr:base=0x00000000, limit=0xffff
idtr:base=0x00000000, limit=0xffff
dr0:0x00000000, dr1:0x00000000, dr2:0x00000000
dr3:0x00000000, dr6:0xffff0ff0, dr7:0x00000400
cr0:0x00000010, cr1:0x00000000, cr2:0x00000000
cr3:0x00000000, cr4:0x00000000
done
<bochs:4> x /64xb 0x7c00↩
[bochs]:
0x00007c00 <bogus+   0>:  0x8c  0xc8  0x8e  0xd8  0x8e  0xc0  0xe8  0x02
0x00007c08 <bogus+   8>:  0x00  0xeb  0xfe  0xb8  0x1e  0x7c  0x89  0xc5
0x00007c10 <bogus+  16>:  0xb9  0x10  0x00  0xb8  0x01  0x13  0xbb  0x0c
0x00007c18 <bogus+  24>:  0x00  0xb2  0x00  0xcd  0x10  0xc3  0x48  0x65
0x00007c20 <bogus+  32>:  0x6c  0x6c  0x6f  0x2c  0x20  0x4f  0x53  0x20
0x00007c28 <bogus+  40>:  0x77  0x6f  0x72  0x6c  0x64  0x21  0x00  0x00
0x00007c30 <bogus+  48>:  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x00007c38 <bogus+  56>:  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
<bochs:5> n↩
Next at t=886153
(0) [0x00007c02] 0000:7c02 (unk. ctxt): mov ds, ax                 ; 8ed8
<bochs:6> trace-reg on↩
Register-Tracing enabled for CPU 0
<bochs:7> n↩
Next at t=886154
eax: 0x0fff0000 268369920
ecx: 0x00120001 1179649
edx: 0x00000000 0
ebx: 0x00000000 0
esp: 0x0000fffe 65534
ebp: 0x00000000 0
esi: 0x000088d2 35026
edi: 0x0000ffde 65502
eip: 0x00007c04
eflags 0x00000282
IOPL=0 id vip vif ac vm rf nt of df IF tf SF zf af pf cf
```

```
(0) [0x00007c04] 0000:7c04 (unk. ctxt): mov es, ax                    ; 8ec0
<bochs:8> c↩
.........
```

以上带有↩符号并以加粗字体显示的是输入，其他均为Bochs的输出。如果你用过GDB，你会觉得这个过程很亲切。没错，它跟用GDB调试程序的感觉是很相似的，最大的区别可能就在于在Bochs的调试模式下我们需要跟CPU、内存、机器指令等内容打更多交道。

在上面的演示过程中，最开始的“b 0x7c00”在0x7c00处设置了断点，随后的命令“c”让代码继续执行，一直到我们设置的断点处停止，然后演示的是用“dump_cpu”指令查看CPU寄存器以及用“x”指令查看内存。随后用一个“n”指令让代码向下走了一步，“trace-reg on”的功能是让Bochs每走一步都显示主要寄存器的值。之所以选择演示这些命令，因为它们基本是调试过程中最常用到的。

如果你在调试过程中忘记了指令的用法，或者根本就忘记了该使用什么指令，可以随时使用help命令，所有命令的列表就呈现在眼前了。你将会发现Bochs的调试命令并不多，不需要多久就可以悉数掌握。表2.1列出了常用的指令以及其典型用法。

表2.1 部分Bochs调试指令

行为	指令	举例
在某物理地址设置断点	b addr	b 0x30400
显示当前所有断点信息	info break	info break
继续执行，直到遇上断点	c	c
单步执行	s	s
单步执行（遇到函数则跳过）	n	n
查看寄存器信息	info cpu r fp sreg creg	info cpu r fp sreg creg
查看堆栈	print-stack	print-stack
查看内存物理地址内容	xp /nuf addr	xp /40bx 0x9013e
查看线性地址内容	x /nuf addr	x /40bx 0x13e
反汇编一段内存	u start end	u 0x30400 0x3040D
反汇编执行的每一条指令	trace-on	trace-on
每执行一条指令就打印CPU信息	trace-reg	trace-reg on

其中“xp /40bx 0x9013e”这样的格式可能显得有点复杂，读者可以用“help x”这一指令在Bochs中亲自看一下它代表的意义。

好了，虽然你可能还无法熟练运用Bochs进行调试，但至少你应该知道，即便你的操作系统出现了问题也并不可怕，有强大的工具可以帮助你进行调试。由于Bochs是开放源代码的，如果你愿意，你甚至可以通过读Bochs的源代码来间接了解计算机的运行过程——因为Bochs就是一台计算机。

2.2 QEMU

如果你选择在 Linux 下开发，其实 Bochs 自己就完全够用了。在本书中，今后的大部分例子均使用 Bochs 作为虚拟机来运行。如果你在 Windows 下开发的话，或许你还需要一个运行稍微快一点的虚拟机，它不是用来运行我们自己的操作系统的，而是用来运行 Linux 的，因为我们的代码是用 Linux 来编译的，并且生成的内核代码是 ELF 格式的。

之所以不用 Bochs 来装一个 Linux，是因为 Bochs 速度比较慢，这是由它的运行机制决定的，它完全模拟硬件及一些外围设备，而很多其他虚拟机大都采用一定程度的虚拟化（Virtualization）技术[5]，使得速度大大提高。

我以 QEMU 为例介绍速度较快的虚拟机，但它绝不是唯一的选择，除它之外，Virtual Box、Virtual PC、VM Ware 等都是很有名气的虚拟机。它们各有自己的优缺点，QEMU 的显著优势是它可以模拟较多的硬件平台，这对于一个操作系统爱好者而言是很具有吸引力的。

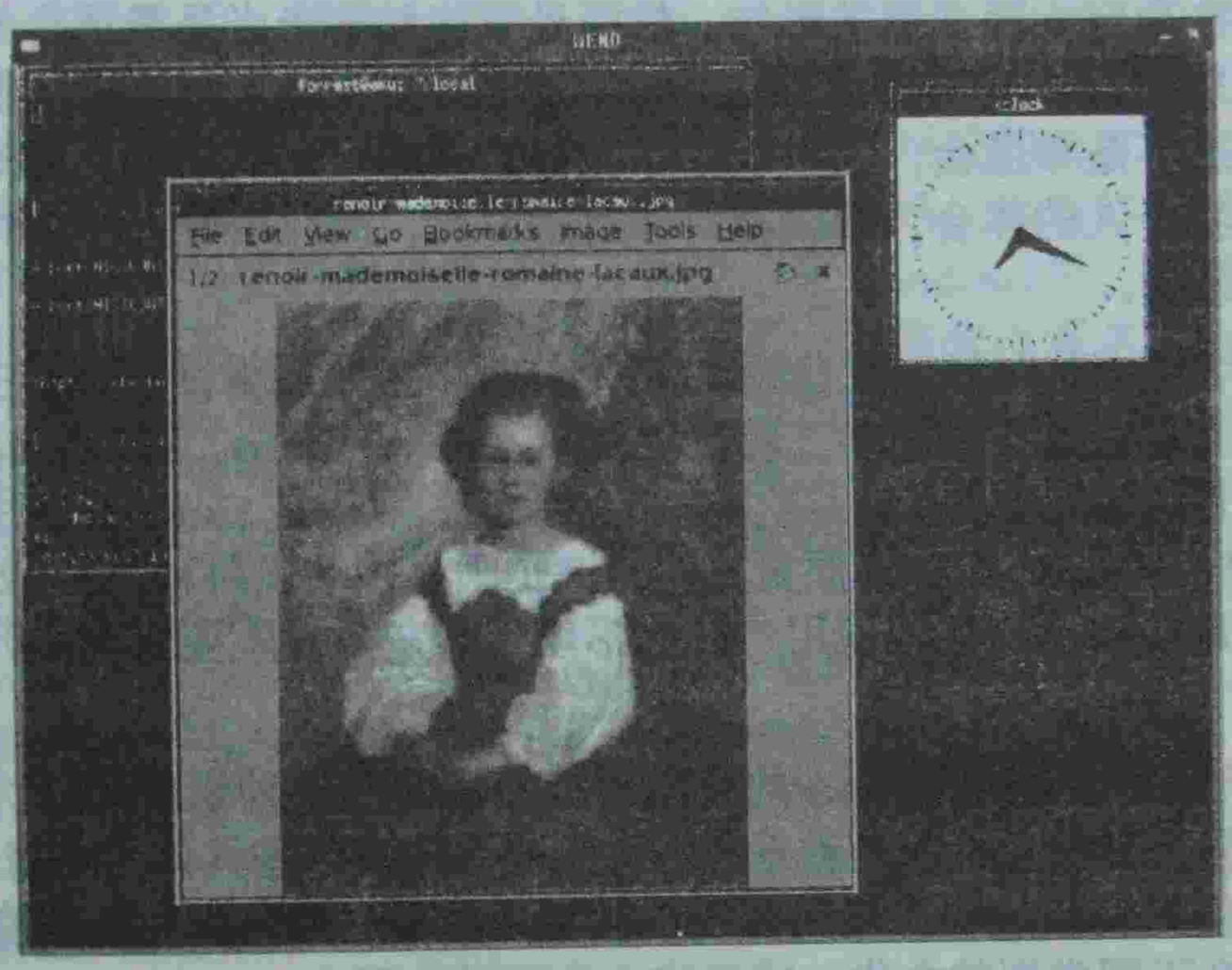


图 2.3 QEMU 中的 Linux

好了，百闻不如一见，图2.3就是 QEMU 上运行 Linux 的抓图。这里我安装的是 Debian 4.0，窗口管理器用的是 Fvwm。在我并不强大的个人电脑上，安装整个系统也没用很久，运行起来也相当顺畅。

与 Bochs 使用配置文件不同，QEMU 运行时需要的参数是通过命令行指定的[6]。比如，如果要用 QEMU 来引导我们之前已经做好的虚拟软盘，可以这样做：

```
▷ qemu -fda a.img
```

[5]读者如果对这一技术感兴趣，可在网上搜索相应资料，比如维基百科上就有个大致的介绍：http://en.wikipedia.org/wiki/Full_virtualization。

[6]实际上 Bochs 也可以用命令行指定参数，详见 Bochs 联机手册。

看起来比 Bochs 还要清爽，不是吗？其实如果你不需要调试的话，在 Linux 下也可以用 QEMU 来运行你的操作系统，这样每次都可以更快地看到结果——这便是我之前说过的“可以让你不必总是这么辛苦”的“妙计”了[7]。

2.3 平台之争：Windows 还是 *nix

读到这里，读者可能发现书中经常出现“如果你用的是 Windows”或者“如果你用的是 Linux”这样的字眼。有时这样的字眼甚至可能影响到你的阅读，如果真的这样请你原谅。我试图照顾尽量多的读者，但是对每一个人来讲，却必须面临一个选择——在什么平台下开发。本书第一版使用的是 Windows 平台，而在第二版中，我投诚了。接下来你会发现，虽然以后的行文会最大限度地兼顾 Windows，但总体是以 Linux 为默认平台的。

其实在什么平台下开发，有时纯粹是口味问题，或者是环境问题——你开始接触计算机时使用什么，很大程度上取决于你周围的人使用什么，而这往往对你的口味产生巨大而深远的影响。然而最早接触的未必是最适合的，在我亲身体会和比较之后，我决定从 Windows 彻底换到 Linux，我想在这里说说为什么。请注意这不是布道会，更不是你开发自己的操作系统必须阅读的章节，我仅仅是谈谈我个人的体会，希望能对你有所启发，同时解释一下为什么第二版会有这样的改动。

在第一版成书的时候，我已经在使用 Linux，但是用得并不多，主要是觉得用不习惯，而现在过了两三年，我已经基本不用 Windows，在 Windows 下我会觉得很不习惯。我的这一经历至少有两点启示：第一是 Linux 不好用是个误解（有一种说法是 Windows 的桌面更好用，这是个复杂的误解），好不好用是习惯问题；第二是如果你有兴趣使用一样你不熟悉的东西，不要因为刚开始的不习惯而放弃。

其实对于 Linux 和 Windows 的误解有很多，我把这种误解归结为操作系统文化上的差异。其实在提起两种系统时，人们往往拿一些具体的事情来做比较。比如比较它们的安装过程、使用方法，甚至是界面。但实际上隐藏在表面背后的是两种完全不同类型的文化，或者称之为不同的理念。

对于 Windows 而言，它的文化植根于微软公司的愿景，“让每个家庭的每个桌面上都有一台电脑”，当然他们希望此电脑内运行的是 Windows 操作系统。这个理想加上 Windows 作为商业软件的性质，决定了 Windows 具有相当程度的亲和力，用户界面显得相当友好。岂止友好，它简直友好到每个人——无论儿童还是老人，受过高等教育还是只念过小学——都能比较容易地开始使用电脑，这无疑是微软对这个社会的巨大贡献。但是界面友好并不一定就完美了，这一点暂且按下不表，我们先来说说 Linux。

Linux 的文化很大程度上来源于 UNIX，UNIX 所倡导和遵循的文化也被称为 UNIX 哲学[8]，其中很重要的一条原则叫做“做一件事并做好”[9]，这听上去

[7]其实妙计不止一条，你也可以在系统内安装两种 Bochs，一种是打开调试功能的，一种是没有打开的，你可以自由选择运行哪一种。

[8]简单的介绍可参见http://en.wikipedia.org/wiki/Unix_philosophy；若想较全面地了解，建议读者阅读 Eric S. Raymond 所著的《UNIX 编程艺术》。

[9]原文作“Do one thing, do it well”。理解这一原则的内涵及外延是理解 UNIX 世界的基本条件。

跟 Windows 的界面友好说的不是一码事，但其实仔细分析起来大有关联。做一件事并做好意味着两件事情，第一件事就是工具之间可以协同作战，不然各人做各人的，无法完成复杂应用；第二件事就是接口要统一，不然无法做到协同。这个统一的接口就是文本流（text stream），这也就意味着，命令行是 UNIX 文化的核心。而 Windows 的做法大有不同，因为要界面友好，于是不能指望用户开始就知道怎么把工具串接在一起，所以 Windows 选择任何应用都自己完成所有功能——至少让用户看起来如此，这使得每个工具都各自为战，从而增加了每个程序的复杂性和开发成本。不仅如此，由于功能都是软件开发者定好的，所以你基本上不能指望大部分的程序具备可扩展性，而在 UNIX 下，大部分的程序都可以跟其他程序协同起来完成程序不曾“设计”的功能。这也是上文我说“界面友好并不一定完美”的原因，友好是有代价的。

那么 UNIX 是一个“不友好”的系统吗？这个问题其实没有看起来那么简单。首先是 UNIX 下流行的桌面环境正在越来越“友好”，你甚至可以将其配置得看上去跟 Windows 别无二致，不过关键点不在于此，而在于长期来看，UNIX 的学习成本并不比 Windows 要高，但收益却要高得多。我们刚刚提到，友好是有代价的，而且代价比想像中要高。对于一个初学者，开始的简单容易使他产生错觉，认为电脑是个简单器械，但实际情况并非如此，一旦遇到麻烦，用户很容易陷入束手无策的境地，一旦有一件事情没有现成的软件可以解决，你马上一筹莫展。而 UNIX 不同，它的学习曲线比较陡峭，但是你一旦入门，就会发现自己的工作可以变得如此轻松而且有趣。在 Windows 中，虽然使用一个工具第一步往往很容易，但很快你就容易迷失在一堆嵌套很深名字晦涩的菜单里面，学习这些菜单可不是一件容易的事情，而且在一个工具里学会的东西到了另一个工具里可能就变了样。如果你想看看程序的帮助，有时也是件困难的事情，因为为了达到“友好”的效果，帮助经常也是一层一层的，很难找到自己需要的内容。而在 UNIX 中，所有的工具都有个手册（Manual），可以通过统一的命令“man”来查看，而且这些手册都是平坦的，你可以一口气从头看到尾，可以随时查看你所要的关键字。此外，除了少数极其复杂的工具，手册基本上是够用的。简而言之，在 UNIX 中，软件使用看起来复杂了，实际上如果你想真正掌握一个东西，用的时间不会比 Windows 中更多。况且，在 Windows 中你很难真正掌握一个东西。

我并非故意贬低 Windows，我说过它对社会的贡献巨大。对于一个平常只用电脑来收收邮件看看电影的用户，它的易用性绝对是巨大的优点，但你我不是这样的用户。我相信阅读本书的人都是程序员，而且都是像我一样喜欢探索的程序员——不喜欢探索的程序员很难有心思写自己的操作系统做消遣。一个程序员的要求和普通用户是不同的，程序员需要了解他的电脑，掌握它，并且可以熟练地让它帮助自己完成工作，从这个角度上讲，UNIX 无疑具有巨大的优势。它里面的每个工具都很锋利，你可以组合着使用，持久地使用，而且许多年都不会过时。

在这里我可以举一个我自己遇到过的例子。在我编写操作系统的文件系统时，需要多次查看某几个扇区的内容，并对其中的数据进行分析。在 Linux 中，我可以很容易地将 od、grep、sed 和 awk 等工具[10]串在一起完成这项工作，我也可以编写一个简单的脚本，将命令放在脚本中方便取用。而在 Windows 中，我

[10]这些都是 UNIX 下的常用工具，读者可以通过联机手册查看它们的用法。更多 UNIX 下的工具介绍可参考http://en.wikipedia.org/wiki/List_of_Unix_utilities。

通常只能在窗口间反复地单击鼠标，费时费力而且效率低下。类似的例子不胜枚举，你一旦熟悉了这些工具，就会发现通过组合它们，你能得到比任何图形界面工具都多的功能。而在 Windows 下就不得不看具体菜单的眼色了。不仅如此，UNIX 下的工具往往学习一次就能长久使用，很少过时。比如刚才提到的几个工具大部分有 20 年以上的历史，到现在它们依然被广泛使用，即便它们学起来会稍微难一点，平摊在 20 年里面，成本也是极低的。这就是 UNIX 哲学，你不需要重复学习，每个工具都好用，而且可以因为跟其他工具结合而发挥多种作用。

大多数 Windows 下的软件都有个毛病，它经常试图隐藏一些东西。它的本意是好的，就是让界面更“友好”，但这对程序员有时是件坏事，因为它让人难以透彻地理解软件的所作所为。或许你会说，如果你想理解，你总能理解的。没错，这跟“在 UNIX 下能做的事情在 Windows 下都能做”是相似的命题，甚至于，只要安装一些额外的软件（比如 Cygwin[11]），你可以在 Windows 下使用 UNIX 的命令。问题是即便能做，你也未必去做，这就是所谓文化的力量。理论上你在任何地方都能读书学习，但效率最高的地方还是教室和书房，在客厅舒服的沙发上，你不自主地就拿起了电视遥控器。

所以以我自己的体会而言，一个程序员最好还是使用类 UNIX 的操作系统。它能在日常生活中帮你提高自己的水平和工作效率。这一点与摄影有点类似，市面上数量最多的是傻瓜相机，但一个专业的摄影师总是会选择功能复杂的专业级设备，不是傻瓜相机不好，而是适应的人群不同，如果你想成为好的摄影师，那么上手容易的傻瓜相机一定不是你的最终选择。不是好不好的问题，是适不适合的问题。

上面论及的是两类操作系统文化上的差异，其实即便是纯粹应用层的，也有诸多误解，比如以下几条：

误解一. Linux 难安装。如果你曾被 Linux 的安装难倒过，我建议你下次试试 Ubuntu[12]。在本书第一版开始写作之时，Ubuntu 的第一个版本还没有发布[13]，但短短几年时间，它已经变成全世界最流行的发行版[14]，这与它的易装和易用性是分不开的。笔者本人大规模地使用 Linux 也是从 Ubuntu 开始的，它的安装过程一点也不比 Windows 的难，而且中文资料也相当丰富，很容易找到志同道合的人。Ubuntu 的另一特点是它的驱动程序很丰富，支持很多的硬件，大部分情况下驱动程序都能自动安装好，甚至不需要用户参与，在这一点上它甚至比 Windows 更“友好”。

误解二. Linux 难学。希望你永远记住，电脑不是个简单器械，无论是硬件、操作系统还是应用软件，都经常比看上去复杂得多。所以易用未必是好事，它肯定向你隐瞒了些什么，花一点时间绝对是值得的，尤其是当你想做一个程序员的时候。况且 Linux 也没那么难学，且不说它的图形界面越来越好用，就是完全用命令行的话，入门也相当容易。而且 Linux 世界的文档齐全且易于检索，更有高度发达的社区文化，在这里你学到的往往比预期的还要多。

误解三. Linux 难用。再强调一下，Linux 的学习曲线是陡峭的，然而一旦你度过了开始的适应期，就会发现原来命令行可以这么好用，原来有这么丰富的

[11] Cygwin 官方网站为 http://www.cygwin.com/。
[12] Ubuntu 官方网站为http://www.ubuntu.com/。
[13] Ubuntu 的第一个版本（代号 Warty Warthog）发行于 2004 年 10 月。
[14] 根据 2008 年 8 月的数据。及时情况可参考http://distrowatch.com/。

工具来提高效率，而且这些好用的工具居然都是自由的[15]！你不需要向作者付费，甚至他们鼓励你使用和传播，你甚至可以随便修改这些工具的源代码，遇到问题可以发一封邮件反馈给作者。这在 Windows 下都是很难做到的，在那片土地上你很难体会什么叫“自由”。不仅如此，当你熟悉之后会发现，同样一件事情，其实 Linux 下的解决方案往往比 Windows 下要简单。就比如我们提到过的安装 Bochs 一例吧，在 Windows 下你通常需要先到 Bochs 网站，在数次单击之后找到下载链接，然后下载，再然后是双击安装程序来安装。在 Linux 下呢，你看到了，只需要一个命令行就可以了，即便你打字的速度再慢，也比那些鼠标单击操作要快。

我在此并非贬低鼠标的好处，我每天都使用鼠标，它绝对是个伟大的发明。但是我们应该只在需要它的时候使用它，而不是试图用它来解决所有事情。这就好比图形界面是个好东西，如果你的工作是图形图像处理，很难想像没有图形界面该怎么做，但并不是图形界面在任何时候都是好的。我们应该分辨每一类工作最适合的工具是什么，而不是用一种思维解决所有问题。这也是我说“Windows 桌面好用是个复杂的误解”的原因，图形界面是个好东西，Windows 把它用得极端了。

误解四. Linux 下软件少。这是最大的一个误区，事实上很少有事情你在 Linux 下是做不到的。而且 Linux 的发行版通常都有发达的包管理工具，无论是关键字查找、安装、更新还是卸载都可以用一组统一的命令来完成。这使得你在需要某种软件时，使用简单的命令就可以找到它，很多时候你能找到不止一种。如果你想看看除了 od 之外还有哪些二进制查看器，在 Ubuntu 或者 Debian 下通过一个 `apt-cache search 'hex.*(view|edit)'`[16]命令就能找到十几种，这些都是自由软件，有命令行的也有图形界面的，而在 Windows 下，怕是又要在浩瀚的互联网中搜索了。

在 Linux 中找一些 Windows 下软件的替代品是很容易的，虽然这种对应有时并非必要。比如字处理软件就有 OpenOffice.org、KOffice、AbiWord 等选择；图像处理软件有 GIMP；多媒体播放软件有 MPlayer、Totem 等。如果你喜欢玩游戏，Linux 下的游戏数量也会让你大吃一惊，不信你可以来这里看一看：http://en.wikipedia.org/wiki/Category:Linux_games。

其实 Linux 的好处还远不止这些，众所周知的一个优点是它基本没有病毒的烦恼。不是 Linux 中开发不出病毒来[17]，而是因为 Linux 系统有自身的权限机制作保障，加上软件来源都可信赖，且大部分都是源代码开放的（其中相当一部分都是自由软件），所以说 Linux 下没有病毒烦恼并非夸张。想想你在与病毒做斗争的过程中浪费了多少时间吧，我已经很久没有这种烦恼了。Windows 或许正变得越来越稳定，但 Linux 一直都很稳定，而且你不需要整天重启你的电脑，笔者的电脑就有时几十天不重启。除非你要升级内核，否则没有很多关机和重启的理由（不管是安装还是卸载，或是对系统内的包进行升级，都不需要重启电脑）。

作为一个操作系统爱好者，使用 Linux 的理由还有一条，那就是 Linux 的内核是“自由”的，注意它不仅仅是“开放源代码”的，你不仅可以获取其源代

[15]注意这里没用“免费”这个词。Free Software 的 Free 是“自由”之意，它比“免费”一词包含了更多意义。欲获得更详细的内容请访问http://www.fsf.org/。

[16]apt-cache 是个 Debian 家族中常用的包管理命令，可以使用正则表达式来搜索软件包。

[17]关于 Linux 系统下的病毒，读者可以参考：http://en.wikipedia.org/wiki/Linux_malware。

码，而且可以自由地复制、修改、传播它，当然也包括学习它。如果你也想加入到内核黑客[18]的队伍，那么就先从使用它开始吧。Linux 不是完美的，它的问题有很多，但每个问题都是你参与的机会，而这种参与可能是你成为顶尖高手的开始。

笔者本人完全使用 Linux 来工作的时间其实很短，几年而已，但我已经深深体会到它给我带来的好处。我并非想说服你，人不能被说服，除非他自己愿意相信。我只是希望你能尝试着去用一用 Linux，或者 UNIX 的其他变种，然后用自己的判断去选择。

如果你坚持使用 Windows，没问题，两者之中都可以很容易地搭建起开发环境，本章后面的部分将会就 Linux 和 Windows 分别来做介绍。

2.4 GNU/Linux 下的开发环境

在工作环境中，虚拟机是个重头戏，所以在本章的前面单独做了介绍。除了虚拟机之外，还有几样重要的东西，分别是编辑器、编译器和自动化工具 GNU Make。

许多在 Linux 下工作的人会使用 Vi 或者 Emacs 作为编辑器。如果你有兴趣尝试，那么还是那句建议，“不要因为刚开始的不习惯而放弃”，因为它们的确是编辑器中的经典，而且和 Linux 一样，具有陡峭的学习曲线。许多人一旦学会使用就爱上它们，这其中也包括笔者自己。当然，学习它们并不是必需的，而且你的选择范围比操作系统要大多了，相信会有一款能让你满意。

对于编译器，我们选择 GCC 和 NASM 分别来编译 C 代码和汇编代码。选择 GCC 的原因很简单，它是 Linux 世界编译器的事实标准。GCC 的全称是 GNU Compiler Collection，在这里我们只用到其中的 C 编译器，所以对我们而言它的全部意义仅为 GNU C Compiler——这也正是它原先的名字。之前提到过，我们使用的 Linux 其实应该叫做 GNU/Linux，所以使用 GCC 是比较顺理成章的，那么为什么不能使用 GCC 来编译我们的汇编代码呢？何苦再用个 NASM 呢？原因在于 GCC 要求汇编代码是 AT&T 格式的，它的语法对于习惯了 IBMPC 汇编的读者而言会显得很奇怪，我猜大部分读者可能都跟我一样，学习汇编语言时使用的教材里介绍的是 IBMPC 汇编。NASM 的官方网站位于http://nasm.sourceforge.net/，你还可以在上面找到详细的文档资料。

关于 GNU Make 的介绍见本书第 5 章。

还是以 Debian 作为示例，安装 GCC 和 NASM 可以通过以下命令来完成：

```
▷ sudo apt-get install build-essential nasm
```

注意这里的 build-essential 软件包中包含 GCC 和 GNU Make。

好了，现在可以总结一下了，如果你想要搭建一个基于 Linux 的开发环境，那么你需要做的工作有以下这些：

- 安装一个 Linux 发行版，如果你对 Linux 不甚熟悉，推荐使用 Ubuntu。

[18]如果你对成为黑客感兴趣，或许可以读一读 Eric S. Raymond 的“How To Become A Hacker”。

- 通过 Linux 发行版的包管理工具或者通过下载源代码手工操作的方式来安装以下内容：
  - 一个你喜欢的编辑器，比如 Emacs。
  - 用于编译 C 语言代码的 GCC。
  - 用于编译汇编代码的 NASM。
  - 用于自动化编译和链接的 GNU Make。
  - 一个用于运行我们的操作系统的虚拟机，推荐使用 Bochs。

再次强调，如果你在安装或使用它们时遇到困难，不要着急，也不要气馁，因为一帆风顺的情形在现实生活中着实很少见。你或许可以试试以下的解决方案（这些方法也适用于其他在自由软件的安装配置及使用等方面的问题）：

- 向身边的朋友求助。
- 使用搜索引擎看看是不是有人遇到类似的问题，那里或许已经给出解决方案。
- 仔细阅读相应资料（不要怕英文），比如安装说明，或是 FAQ。
- 订阅相应的邮件列表（Mailing List），只要能将问题描述清楚[19]，通常你能在几小时内得到答复。
- 到论坛提问。
- 如果实在是疑难杂症，你可以试着联系软件的开发者，通常也是通过邮件列表的方式（同一个项目可能有多个邮件列表，开发者邮件列表通常与其他分离）。
- 自己阅读源代码并独立解决——这或许是个挑战，然而一旦解决了问题，你将获得知识、经验以及成功的喜悦。

将来，如果一切顺利的话，你编写操作系统时的步骤很可能是这样的：

1. 用编辑器编写代码。
2. 用 Make 调用 GCC、NASM 及其他 Linux 下的工具来生成内核并写入磁盘映像。
3. 用 Bochs 来运行你的操作系统。
4. 如果有问题的话
   (a) 用各种方法来调试，比如用 Bochs；
   (b) 返回第 1 步。

[19]关于提问的技巧，请参考 Eric S. Raymond 的“How To Ask Questions The Smart Way”。

2.5 Windows 下的开发环境

我们在介绍 QEMU 时提到过，在 Windows 下你需要一个虚拟的 Linux 来帮你编译操作系统的源代码。将操作系统内核编译链接成 ELF 格式有诸多好处，我们不但可以用 Linux 下现成的工具[20]来分析编译好的内核，还可以在必要时参考 Linux 内核的源代码来帮助我们自己的开发，总之这拉近了我们与 Linux 之间的距离。所以不要因为在 Windows 下也离不开 Linux 这件事而沮丧，况且装一个 Linux 是件很容易的事情。

不过装一个虚拟的 Linux 跟装一个真实的 Linux 还是有所不同，主要在于两点。一是我们仅仅想用这个 Linux 来做编译链接的工作，所以在选择组件的时候尽量去除不必要的内容，这样可以节省时间和空间；二是要确保你选择的虚拟机容易跟宿主机进行网络通信，因为你需要将宿主机上的源代码拿给虚拟机来编译。

安装方法可以有多种选择，比较简单的方法是通过光盘安装，当然这个光盘也可以是“虚拟”的，也就是一个光盘映像。首先到你所中意的 Linux 发行版的官方网站下载一个安装光盘的映像，有些发行版还提供免费或付费的邮寄服务，读者可以根据自己的喜欢自行选择。这里假设你得到的是光盘映像，文件名为 inst.iso。

有了光盘映像，我们还缺少一个硬盘映像，读者可以用前文提到过的 bximage 来生成它，也可以使用下面的命令：

```
▷ qemu-img create hd.img 1500M
```

这样就能生成一个大小约为 1.5GB 的硬盘映像了。

接下来就可以进行安装了：

```
▷ qemu -cdrom inst.iso -hda hd.img -boot d
```

安装过程从略，注意尽量精简你的组件，不要安装太多无用的东西。这些组件对我们是必需的：GCC、GNU Make、NASM、Samba。如果它们在安装时默认没有装上，那么你需要在系统安装结束后将它们安装上。由于目前大多数虚拟机都具有好用的网络功能，所以安装它们并非难事。

装完之后，我们还需要解决让宿主机和虚拟机通信的问题。其实你可以把它们看成是局域网中的两台机器，局域网中适用的方法这里同样适用，所以 Samba 就很适合。

首先在 Windows 中以可读写方式共享一个文件夹，假设叫做 OrangeS，然后在虚拟的 Linux 上运行下面这条命令：

```
▷ sudo mount -t smbfs -o username=user,password=blah \
//10.0.2.2/OrangeS /mnt
```

其中假设你的宿主机 IP 地址为 10.0.2.2。这样在 Linux 的 /mnt 目录下就能看到 Windows 共享文件夹下的内容了，你可以在虚拟机中随意读写，就像对待本地文件一样。

[20] 比如 readelf。

这样一来，你的编译环境就安装完成了，接下来，如同在Linux下一样，你还需要一个编辑器。据说始终有一部分人使用记事本（notepad）来编写代码，不管基于何种理由，希望你不要这样做，因为你可以找到许多比notepad更适合编写代码的编辑器，有收费的，也有免费的，它们通常都具备关键字颜色，自动缩进等方便开发者的功能，可以大大提高工作效率。

总结一下的话，搭建一个Windows下的开发环境，你需要做以下工作：

- 安装Windows。
- 安装Bochs（安装程序可到其官方网站获取）。
- 安装一个你喜欢的编辑器用来编写代码。
- 安装一个速度较快的虚拟机，如QEMU（安装程序可到其官方网站获取[21]）。
- 在速度较快的虚拟机上安装一个Linux。
- 在虚拟的Linux中安装GCC、GNU Make、NASM、Samba——如果它们没有默认被安装上的话。
- 在虚拟的Linux和宿主机之间共享一个可读写的文件夹。

将来你的开发过程看起来很可能是这样的：

1. 在Windows中用编辑器编写代码。
2. 在虚拟Linux中用Make调用GCC、NASM及其他工具来生成内核并写入磁盘映像。
3. 在Windows中用Bochs来运行你的操作系统。
4. 如果有问题的话。
   (a) 用各种方法来调试，比如用Bochs；
   (b) 返回第1步。

2.6 总结

好了，到这里相信读者已经知道如何搭建自己的开发环境了，说白了它跟开发一个普通的软件区别基本就在一个虚拟机上。它既是我们的“硬件”，又是我们的调试器，有了它我们安心多了。那是不是马上就可以开始我们的操作系统开发之旅了呢？很遗憾，还不能那么着急，因为你知道，操作系统是跟硬件紧密相连的，如果想实现一个运行在使用IA32架构的IBM PC上的操作系统，免不了要具备相关的知识。其中的重头戏就是32位Intel CPU的运行机制，毕竟CPU是一台计算机的大脑，也是整个计算机体系的核心。

[21] QEMU的官方网站位于http://bellard.org/qemu/。

所以紧接着我们要学习的，就是要了解 IA32 保护模式。掌握了保护模式，我们才知道 Intel 的 CPU 如何运行在 32 位模式之下，从而才有可能写出一个 32 位的操作系统。

如果读者已经掌握了保护模式的内容，可以直接跳到第 4 章。

Everyone can be taught to sculpt: Michelangelo would have had to be taught how not to. So it is with the great programmers.

——*Alan Perlis*

3

保护模式（Protect Mode）

或许你从来没有接触过保护模式这个概念，如果真的是这样，请不要害怕，这个概念虽不容易用定义解释清楚，但如果读者怀着对它的好奇和疑问阅读下面的章节，在获得了一定的感性认识之后，会看到眼前的迷雾渐渐散去，并且最终很好地理解它。而且，本书也会不断跟读者一起来体会这个概念中“保护”一词的内在含义，跟读者一起边学习边思考。

3.1 认识保护模式

很多时候，我觉得自己是懒惰和急功近利的，在学习新东西的时候不肯花时间阅读大段概念性的叙述。我认为那是给已经明白了的人看的，当我也懂了的时候，我可能能弄清楚作者在说些什么，可当我初学的时候，我真的不太喜欢泛泛的介绍，我想尽快看到些吸引眼球的东西。

说不定，你也有这样的想法。

那好，在你不知道什么叫保护模式之前，让我们先来看一段代码，如果你没有接触过这些内容，可能会觉得一头雾水，不知所云，不要紧，这正是我们想要的效果——在好奇心的驱使下，人总是很勤劳。

请看代码3.1。

代码 3.1 chapter3/a/pmtest1.asm

```
; ==========================================
; pmtest1.asm
; 编译方法：nasm pmtest1.asm -o pmtest1.bin
; ==========================================

%include        "pm.inc"        ; 常量，宏，以及一些说明

org     07c00h
        jmp     LABEL_BEGIN

[SECTION .gdt]
; GDT
;                                段基址,       段界限     , 属性
LABEL_GDT:          Descriptor       0,                0, 0                    ; 空描述符
LABEL_DESC_CODE32:  Descriptor       0, SegCode32Len - 1, DA_C + DA_32; 非一致代码段
LABEL_DESC_VIDEO:   Descriptor 0B8000h,           0ffffh, DA_DRW             ; 显存首地址
```

```
; GDT 结束

GdtLen          equ     $ - LABEL_GDT   ; GDT长度
GdtPtr          dw      GdtLen - 1      ; GDT界限
                dd      0               ; GDT基地址


; GDT 选择子
SelectorCode32          equ     LABEL_DESC_CODE32       - LABEL_GDT
SelectorVideo           equ     LABEL_DESC_VIDEO        - LABEL_GDT
; END of [SECTION .gdt]

[SECTION .s16]
[BITS   16]
LABEL_BEGIN:
        mov     ax, cs
        mov     ds, ax
        mov     es, ax
        mov     ss, ax
        mov     sp, 0100h

        ; 初始化 32 位代码段描述符
        xor     eax, eax
        mov     ax, cs
        shl     eax, 4
        add     eax, LABEL_SEG_CODE32
        mov     word [LABEL_DESC_CODE32 + 2], ax
        shr     eax, 16
        mov     byte [LABEL_DESC_CODE32 + 4], al
        mov     byte [LABEL_DESC_CODE32 + 7], ah

        ; 为加载 GDTR 作准备
        xor     eax, eax
        mov     ax, ds
        shl     eax, 4
        add     eax, LABEL_GDT          ; eax <- gdt 基地址
        mov     dword [GdtPtr + 2], eax ; [GdtPtr + 2] <- gdt 基地址

        ; 加载 GDTR
        lgdt    [GdtPtr]

        ; 关中断
        cli

        ; 打开地址线A20
        in      al, 92h
        or      al, 00000010b
        out     92h, al

        ; 准备切换到保护模式
        mov     eax, cr0
        or      eax, 1
        mov     cr0, eax

        ; 真正进入保护模式
        jmp     dword SelectorCode32:0  ; 执行这一句会把 SelectorCode32 装入 cs,
; END of [SECTION .s16]                 ; 并跳转到 Code32Selector:0  处

[SECTION .s32]; 32 位代码段. 由实模式跳入.
[BITS   32]

LABEL_SEG_CODE32:
        mov     ax, SelectorVideo
        mov     gs, ax                  ; 视频段选择子(目的)

        mov     edi, (80 * 11 + 79) * 2 ; 屏幕第 11 行, 第 79 列。
```

```
84          mov     ah, 0Ch                 ; 0000: 黑底    1100: 红字
85          mov     al, 'P'
86          mov     [gs:edi], ax
87
88          ; 到此停止
89          jmp     $
90
91  SegCode32Len    equ     $ - LABEL_SEG_CODE32
92  ; END of [SECTION .s32]
```

如果你是第一次读到这里，不知道你对这段代码有怎样的感觉，可能对开头几行的耐心到了第二页就消失得差不多了，然后很快地扫视，很短的时间之后，你已经到达代码的末尾。是啊，可能它显得的确有点晦涩。首先要说明的是，这段代码将实现由实模式到保护模式的转换。至于实模式和保护模式的概念及区别，我们之后再做详细解释，现在，先让我们把这段过一遍，分别看一下它的三个部分。

我们将用下面的命令来编译它：

```
▷ nasm pmtest1.asm -o pmtest1.bin
```

和之前一样，除了指定了生成的文件名，并没有用到NASM的其他参数。于是在默认情况下[1]，NASM将生成一个纯二进制文件（也称为Raw Binary File或Flat-form Binary File）。也就是说，生成的二进制中除了你写的源代码之外不包含其他任何东西。这也意味着，程序执行时的内存映像和二进制文件映像是一样的。所以，在上面的程序中定义的section并没有什么实质上的作用，即便不定义它们，从执行结果来看也是一样的（编译出来的二进制会有微小差别）。定义它们只是使代码结构上比较清晰，而且，后面我们对这个程序渐渐扩展的时候，它还有一点小小的妙用。

好了，首先看[SECTION .gdt]这个段，其中的Descriptor是在pm.inc中定义的宏（见代码3.2）。先不要管具体的意义是什么，看字面我们可以知道，这个宏表示的不是一段代码，而是一个数据结构，它的大小是8字节。

代码3.2 用来生成描述符的宏（节自chapter3/a/pm.inc）

```
251 ; 描述符
252 ; usage: Descriptor Base, Limit, Attr
253 ;        Base:  dd
254 ;        Limit: dd (low 20 bits available)
255 ;        Attr:  dw (lower 4 bits of higher byte are always 0)
256 %macro Descriptor 3
257         dw      %2 & 0FFFFh                             ; 段界限1
258         dw      %1 & 0FFFFh                             ; 段基址1
259         db      (%1 >> 16) & 0FFh                       ; 段基址2
260         dw      ((%2 >> 8) & 0F00h) | (%3 & 0F0FFh)     ; 属性1 + 段界限2 + 属性2
261         db      (%1 >> 24) & 0FFh                       ; 段基址3
262 %endmacro ; 共 8 字节
```

在段[SECTION .gdt]中并列有3个Descriptor，看上去是个结构数组，你一定猜到了，这个数组的名字叫做GDT。

GdtLen是GDT的长度，GdtPtr也是个小的数据结构，它有6字节，前2字节是GDT的界限，后4字节是GDT的基地址。

[1]在默认情况下NASM生成什么格式的文件可通过命令 nasm -hf 来查看。

另外还定义了两个形如SelectorXXXX的常量，至于是做什么用的，我们暂且不管它。

再往下到了一个代码段，[BITS 16]明确地指明了它是一个 16 位代码段。你会发现，这段程序修改了一些 GDT 中的值，然后执行了一些不常见的指令，最后通过jmp指令实现一个跳转（第71行）。正如代码注释中所说的，这一句将“真正进入保护模式”。实际上，它将跳转到第三个 section，即[SECTION .s32]中，这个段是 32 位的，执行最后一小段代码。这段代码看上去是往某个地址处写入了 2 字节，然后就进入了无限循环。

虽然我们还未能了解更多细节，但我猜，你一定在想这段程序的执行结果会是怎样的，我们先来看一下。

首先按照刚才所说的办法编译，然后将第 2 章中我们用过的软盘映像a.img和 Bochs 的配置文件bochsrc复制过来，并将生成的二进制写入软盘映像：

```
▷ dd if=pmtest1.bin of=a.img bs=512 count=1 conv=notrunc
```

好了，运行 Bochs 来看一下，你会得到如图3.1所示的效果。

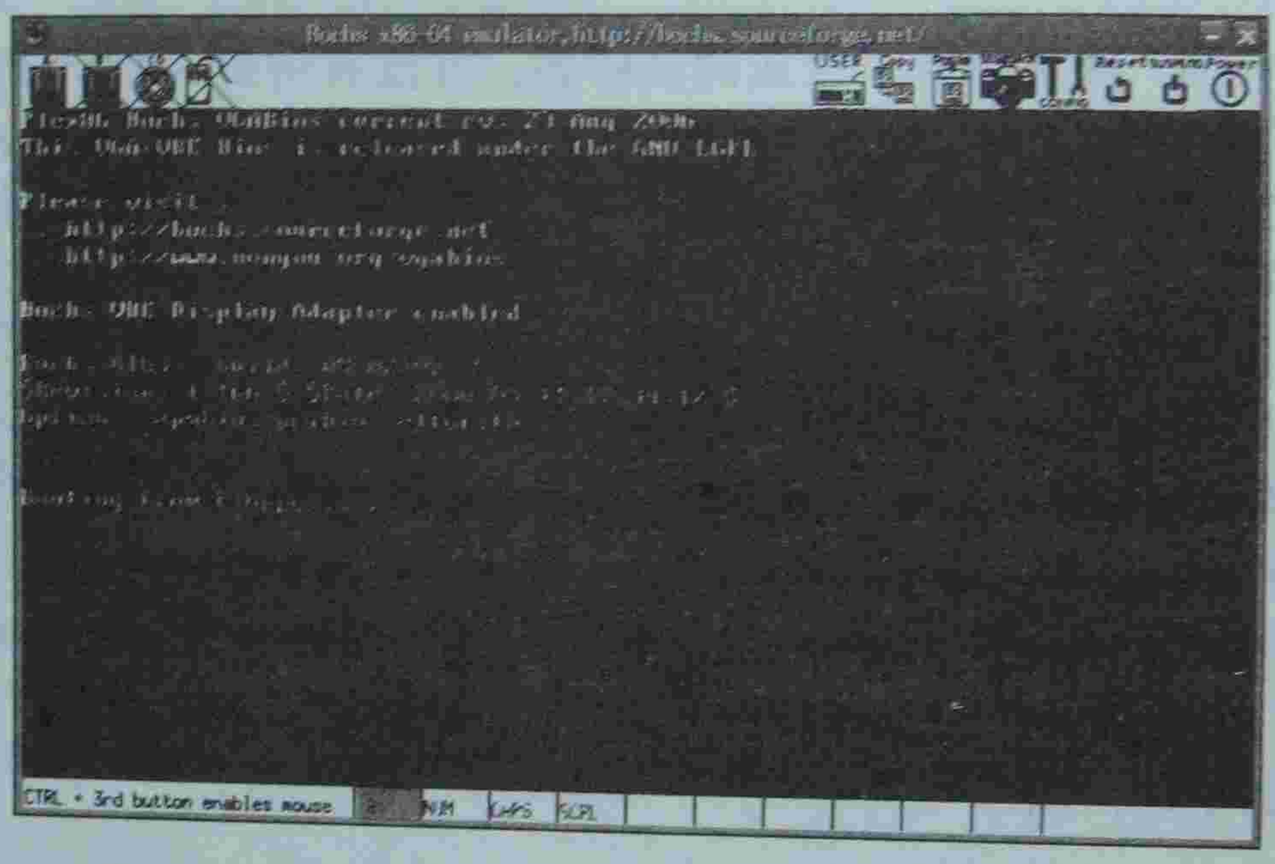


图 3.1 pmtest1.bin 的执行结果

可以看到，在屏幕中部右侧，出现了一个红色的字母“P”，然后再也不动了。不难猜到，程序的最后一部分代码中写入的两个字节是写进了显存中。

现在，大致的感性认识已经有了，但你一定有一些疑惑，什么是 GDT？那些看上去怪怪的指令到底在做什么？现在我们先来总结一下，在这个程序中，我们了解到什么，有哪些疑问。

我们了解到的内容如下：

- 程序定义了一个叫做 GDT 的数据结构。
- 后面的 16 位代码进行了一些与 GDT 有关的操作。
- 程序最后跳到 32 位代码中做了一点操作显存的工作。

我们不明就里的内容如下：

- GDT是什么？它是干什么用的？
- 程序对GDT做了什么？
- 那个jmp SelectorCode32:0跟我们从前用过的jmp有什么不同？

好了，下面我们将一一解答这些问题，并以此为突破口引领你走进保护模式的大门。在解答的过程中，我想说明的是，本章不打算成为全面介绍保护模式的课程，本着够用原则，我们不会涉及保护模式的所有内容，只要能自由地编写操作系统代码就足够了。一个典型的例子是V86，它是保护模式的一部分，但是如果你不想在自己的操作系统中支持16位程序，你可能永远都不需要知道它的实现方法，学习它简直是对自己宝贵时间的浪费。还有一点就是，通过由一个简单程序辐射开来的方式学习一个并不简单的体系，开始对某些地方的认识可能是片面的，这没有关系，随着我们对这个程序的不断扩充，你终究会有比较全面的了解。

好的，现在就让我们出发，跟我一起走近保护模式。

3.1.1 保护模式的运行环境

说好了现在出发，我又打断你，真抱歉。不过有件事不得不说，你刚才看到了，我们是把pmtest1.bin写到了引导扇区运行的，这样做固然方便，但是有个隐患，就是引导扇区空间有限，只有512个字节，如果我们的程序越来越大超过了512个字节这方法就不灵了，所以我们得想个更好的方法。

事实上有两个方法可解决这个问题，一是写一个引导扇区，可以读取我们的程序并运行它，就好像这程序是个操作系统内核似的，这个方法对目前的我们来说难度大了点，不过倒是可以先把别人的引导扇区借用一下，也是可行的。第二个方法就是借助别的东西，比如DOS，我们把程序编译成COM文件，然后让DOS来执行它。

虽然第一个方法更酷，但我们还是选择使用第二种方法，因为很多保护模式的教程都是基于DOS来讲的，如果读者在本书中有东西没弄明白，可以同时参考其他教程，如果本书另择一种新方法，或许会变成一种潜在的障碍。

其实使用一个DOS是很容易的事情，只需要按照以下步骤操作即可：

1. 到Bochs官方网站下载一个FreeDos。解压后将其中的a.img复制到我们的工作目录中，并改名为freedos.img。
2. 用bximage生成一个软盘映像，起名为pm.img。
3. 修改我们的bochsrc，确保其中有以下三行：

```
floppya: 1_44=freedos.img, status=inserted
floppyb: 1_44=pm.img, status=inserted
boot: a
```

4. 启动Bochs，待FreeDos启动完毕后格式化B:盘，如图3.2所示。
5. 将代码3.1的第8行中的07c00h改为0100h，并重新编译：

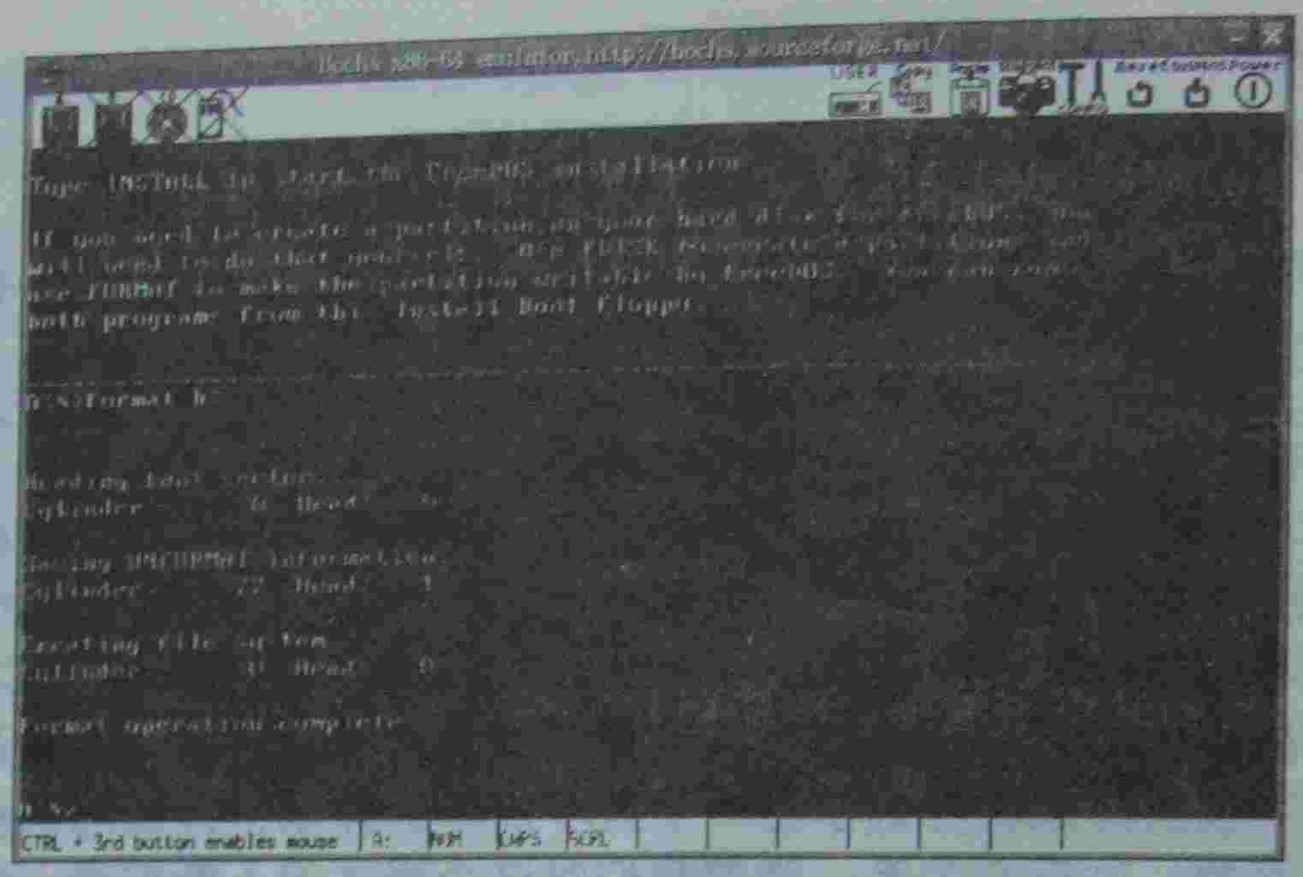


图 3.2 在 FreeDos 中格式化（虚拟）软盘

```
▷ nasm pmtest1.asm -o pmtest1.com
```

6. 将pmtest1.com复制到虚拟软盘pm.img上：

```
▷ sudo mount -o loop pm.img /mnt/floppy
▷ sudo cp pmtest1.com /mnt/floppy/
▷ sudo umount /mnt/floppy
```

7. 到 FreeDos 中执行如下命令：

```
▷ B:\pmtest1.com
```

这样pmtest1.com就运行起来了，请看图3.3，一个红色的字母“P”出现了。

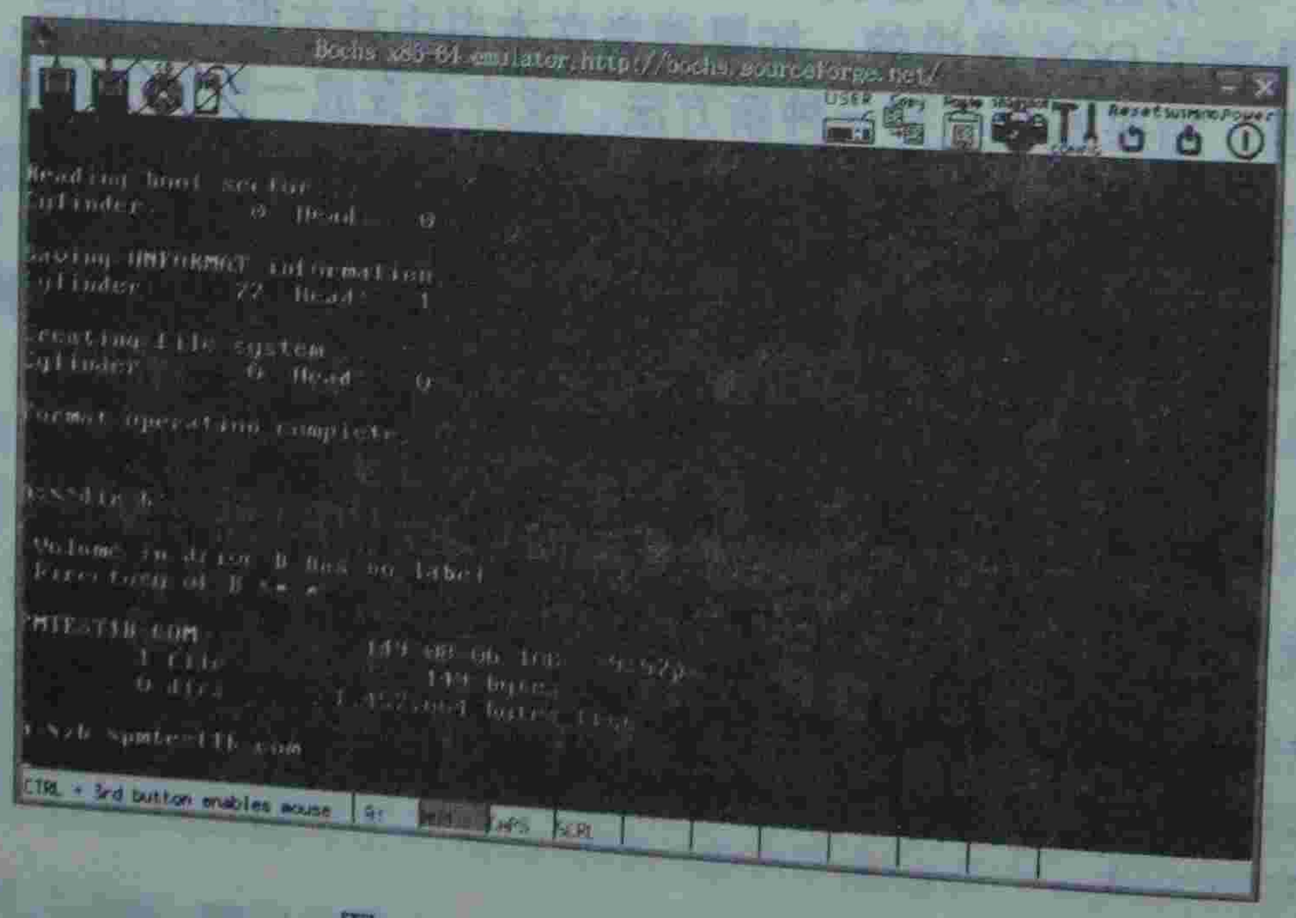


图 3.3 在 FreeDos 中运行 pmtest1

上述步骤在 Linux 和 Windows 下没有太大区别，唯一例外的是第 6 步，如果你用 Windows 的话这一步骤需要在虚拟的 Linux 中完成。

好了，希望这些步骤没有打扰你太久，现在进入正题。

3.1.2 GDT（Global Descriptor Table）

在 IA32 下，CPU 有两种工作模式：实模式和保护模式。直观地看，当我们打开自己的 PC，开始时 CPU 是工作在实模式下的，经过某种机制之后，才进入保护模式。在保护模式下，CPU 有着巨大的寻址能力，并为强大的 32 位操作系统提供了更好的硬件保障。如果你还是不明白，我们不妨这样类比，实模式到保护模式的转换类似于政权的更替，开机时是在实模式下，就好像皇帝 A 在执政，他有他的一套政策，你需要遵从他订立的规则，否则就可能被杀头。后来通过一种转换，类似于革命，新皇帝 B 登基，登基的那一刻类似于程序中那个历史性的jmp（我们后面会有专门的介绍）。之后，B 又有他的一套完全不同的新政策。当然，新政策比老政策好得多，对人民来说有更广阔的自由度，虽然它要复杂得多，这套新政策就是保护模式。我们要学习的，就是新政策是什么，我们在新政策下应该怎样做事。

我们先来回忆一下旧政策。Intel 8086 是 16 位的 CPU，它有着 16 位的寄存器（Register）、16 位的数据总线（Data Bus）以及 20 位的地址总线（Address Bus）和 1MB 的寻址能力。一个地址是由段和偏移两部分组成的，物理地址遵循这样的计算公式：

物理地址（Physical Address）＝段值（Segment）×16＋偏移（Offset）

其中，段值和偏移都是 16 位的。

从 80386 开始，Intel 家族的 CPU 进入 32 位时代。80386 有 32 位地址线，所以寻址空间可以达到 4GB。所以，单从寻址这方面说，使用 16 位寄存器的方法已经不够用了。这时候，我们需要新的方法来提供更大的寻址能力。当然，慢慢地你能看到，保护模式的优点不仅仅在这一个方面。

在实模式下，16 位的寄存器需要用“段:偏移”这种方法才能达到 1MB 的寻址能力，如今我们有了 32 位寄存器，一个寄存器就可以寻址 4GB 的空间，是不是从此段值就被抛弃了呢？实际上并没有，新政策下的地址仍然用“段:偏移”这样的形式来表示，只不过保护模式下“段”的概念发生了根本性的变化。实模式下，段值还是可以看做是地址的一部分的，段值为xxxxh表示以xxxx0h开始的一段内存。而保护模式下，虽然段值仍然由原来 16 位的cs、ds等寄存器表示，但此时它仅仅变成了一个索引，这个索引指向一个数据结构的一个表项，表项中详细定义了段的起始地址、界限、属性等内容。这个数据结构，就是 GDT（实际上还可能是 LDT，这个以后再介绍）。GDT 中的表项也有一个专门的名字，叫做描述符（Descriptor）。

也就是说，GDT 的作用是用来提供段式存储机制，这种机制是通过段寄存器和 GDT 中的描述符共同提供的。为了全面地了解它，我们来看一下图3.4所示的描述符的结构。

这个示意图表示的是代码段和数据段描述符，此外，描述符的种类还有系统段描述符和门描述符，下文会有介绍。除了 BYTE5 和 BTYE6 中的一堆属性看

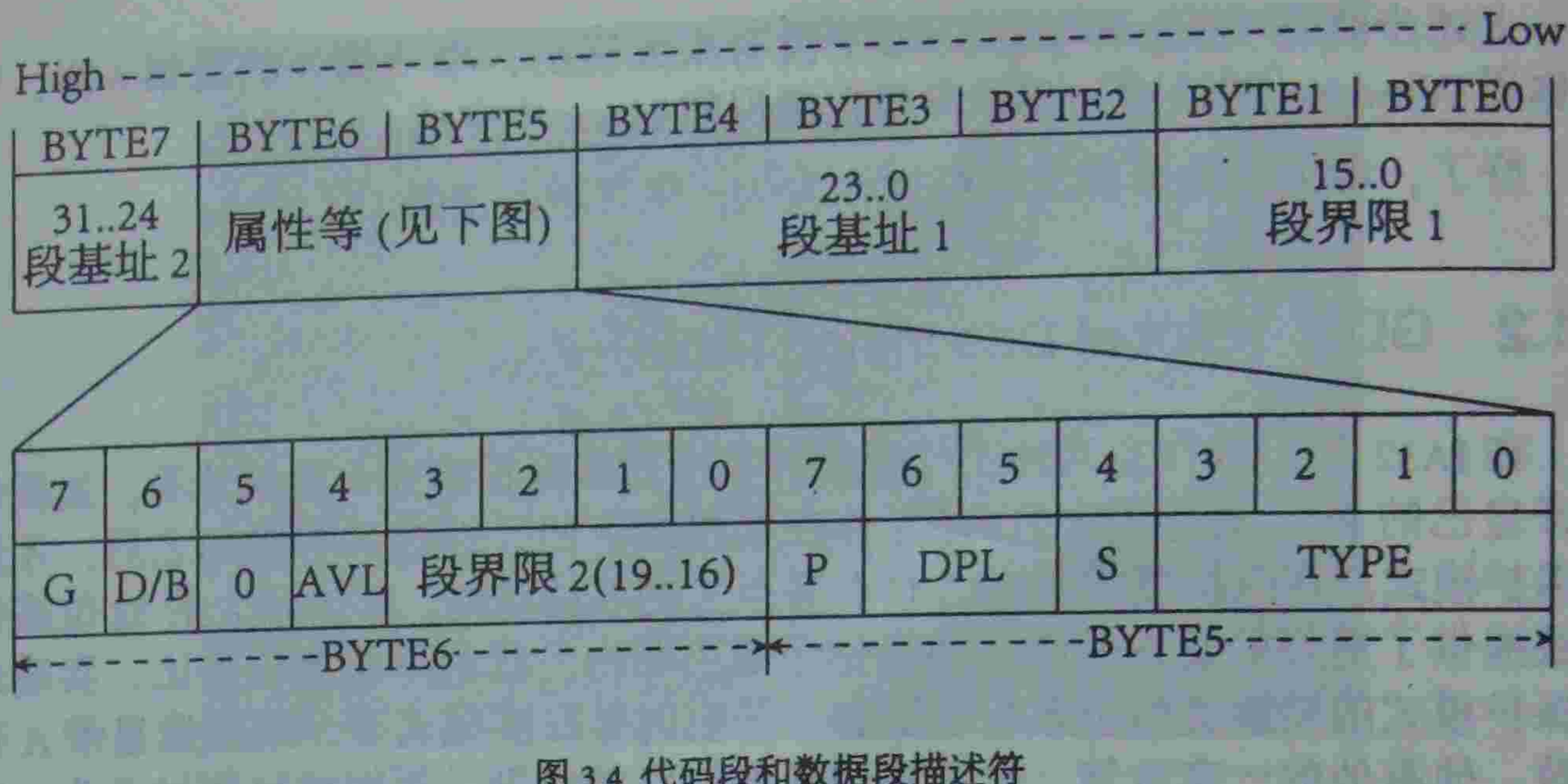


图 3.4 代码段和数据段描述符

上去有点复杂以外，其他三个部分倒还容易理解，它们分别定义了一个段的基址和界限。不过，由于历史问题，它们都被拆开存放。至于那些属性，我们暂时先不管它。

好了，我们回头再来看看代码3.1，Descriptor这个宏用比较自动化的方法把段基址、段界限和段属性安排在一个描述符中合适的位置，有兴趣的读者可以研究这个宏的具体内容。

本例的 GDT 中共有 3 个描述符，为方便起见，在这里我们分别称它们为DESC_DUMMY、DESC_CODE32和DESC_VIDEO。其中DESC_VIDEO的段基址是0B8000h，顾名思义，这个描述符指向的正是显存。

现在我们已经知道，GDT 中的每一个描述符定义一个段，那么cs、ds等段寄存器是如何和这些段对应起来的呢？你可能注意到了，在[SECTION.s32]这个段中有两句代码是这样的（第80行和第81行）：

```
mov     ax, SelectorVideo
mov     gs, ax
```

看上去，段寄存器gs的值变成了SelectorVideo，我们在上文中可以看到，SelectorVideo是这样定义的（第25行）：

```
SelectorVideo    equ     LABEL_DESC_VIDEO - LABEL_GDT
```

直观地看，它好像是DESC_VIDEO这个描述符相对于 GDT 基址的偏移。实际上，它有一个专门的名称，叫做选择子（Selector），它也不是一个偏移，而是稍稍复杂一些，它的结构如图3.5所示。

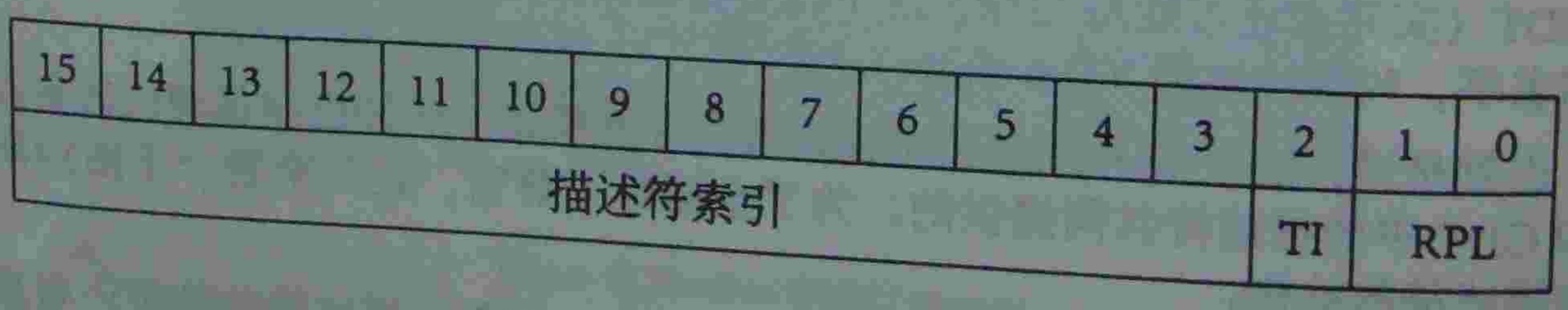


图 3.5 选择子（Selector）的结构

不难理解，当TI和RPL都为零时，选择子就变成了对应描述符相对于GDT基址的偏移，就好像我们程序中那样。

看到这里，读者肯定已经明白了第86行的意思，gs值为SelectorVideo，它指示对应显存的描述符DESC_VIDEO，这条指令将把ax的值写入显存中偏移位edi的位置。

总之，整个的寻址方式如图3.6所示。

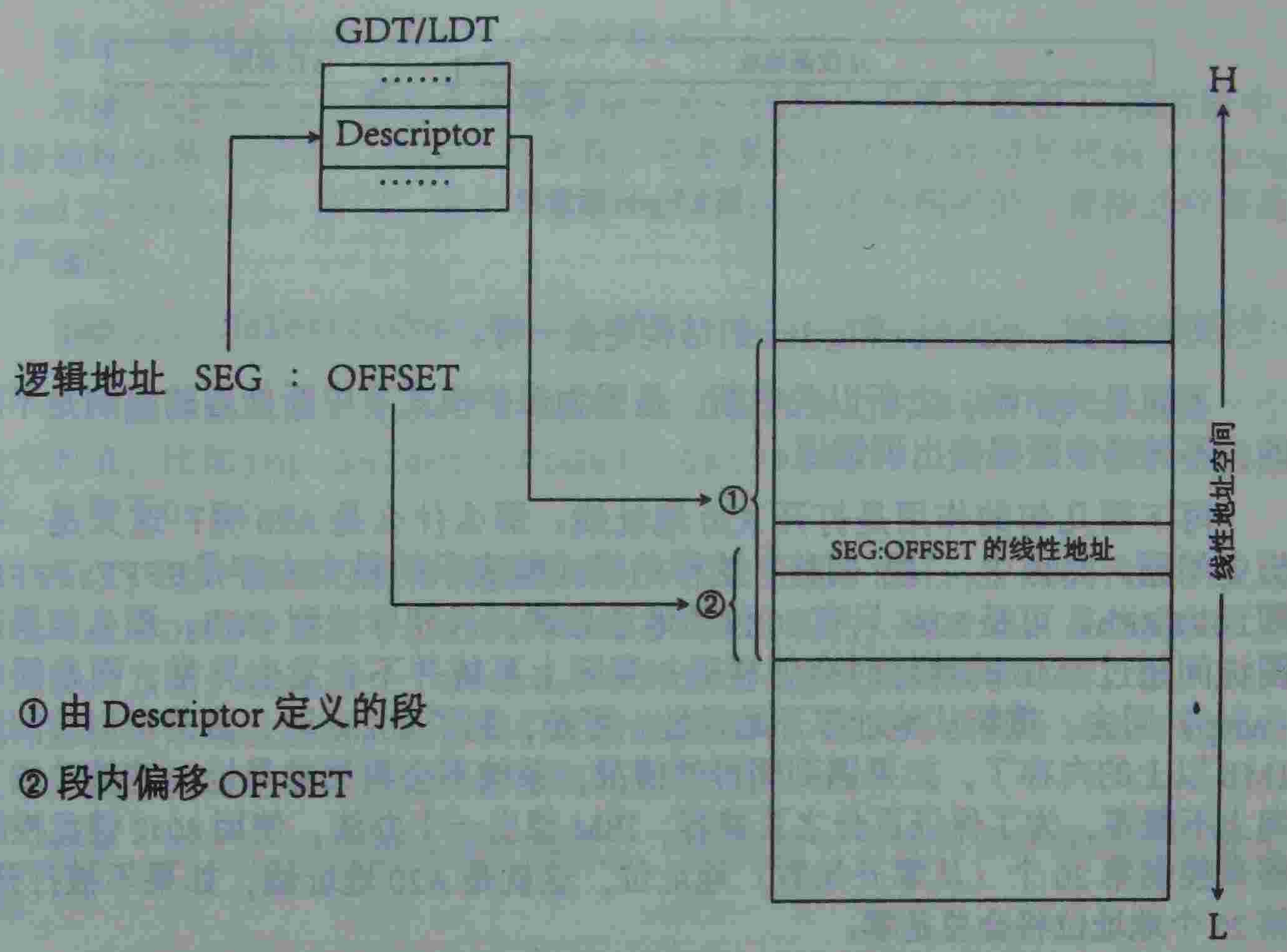


图 3.6 段式寻址示意图

注意图3.6中“段:偏移”形式的逻辑地址（Logical Address）经过段机制转化成“线性地址”（Linear Address），而不是“物理地址”（Physical Address），其中的原因我们以后会提到。在上面的程序中，线性地址就是物理地址。另外，包含描述符的，不仅可以是GDT，也可以是LDT。

明白了这些，离明白整个程序的距离已经只剩一层窗纸了。因为只剩下[SECTION.s16]这一段还没有分析。不过，既然[SECTION.s32]是32位的程序，并且在保护模式下执行，那么[SECTION.s16]的任务一定是从实模式向保护模式跳转了。下面我们就来看一下实模式是如何转换到保护模式的。

3.1.3 实模式到保护模式，不一般的jmp

让我们到[SECTION .s16]这段，先看一下初始化32位代码段描述符的这一段，代码首先将LABEL_SEG_CODE32的物理地址（即[SECTION .s32]这个段的物理地址）赋给eax，然后把它分成三部分赋给描述符DESC_CODE32中的相应位置。由于DESC_CODE32的段界限和属性已经指定，所以至此，

DESC_CODE32的初始化全部完成。

接下来的动作把 GDT 的物理地址填充到了GdtPtr这个 6 字节的数据结构中，然后执行了一条指令（第55行）：

```
lgdt [GdtPtr]
```

这一句的作用是将GdtPtr指示的 6 字节加载到寄存器gdtr，gdtr的结构如图3.7所示。

图 3.7 gdtr 示意图

可以看到，GdtPtr和gdtr的结构完全一样。

下面是关中断，之所以关中断，是因为保护模式下中断处理的机制是不同的，不关掉中断将会出现错误。

再下面几句的作用是打开 A20 地址线。那么什么是 A20 呢？这又是一个历史问题。8086 中，“段:偏移”这样的模式能表示的最大内存是FFFF:FFFF，即10FFEFh。可是 8086 只有 20 位的地址总线，只能寻址到 1MB，那么如果试图访问超过 1MB 的地址时会怎样呢？实际上系统并不会发生异常，而是回卷（wrap）回去，重新从地址零开始寻址。可是，到了 80286 时，真的可以访问到 1MB 以上的内存了，如果遇到同样的情况，系统不会再回卷寻址，这就造成了向上不兼容，为了保证百分之百兼容，IBM 想出一个办法，使用 8042 键盘控制器来控制第 20 个（从零开始数）地址位，这就是 A20 地址线，如果不被打开，第 20 个地址位将会总是零。

显然，为了访问所有的内存，我们需要把 A20 打开，开机时它默认是关闭的。

如何打开呢？说起来有点复杂，在这里我们只使用通过操作端口 92h 来实现这一种方式（第61行到第63行），其他方式略过不提。你需要知道的是，这不是唯一的方法，而且在某些个别情况下，这种方法可能会出现问题。但是，在绝大多数情况下，它是适用的。

在代码中你一定看到了，我们离那个历史性的jmp越来越近，只剩下最后一段代码，这段代码从字面上看也是很简单的，只是把寄存器cr0的第 0 位置为 1。实际上，这一位正是决定实模式和保护模式的关键，cr0的结构如图3.8所示。

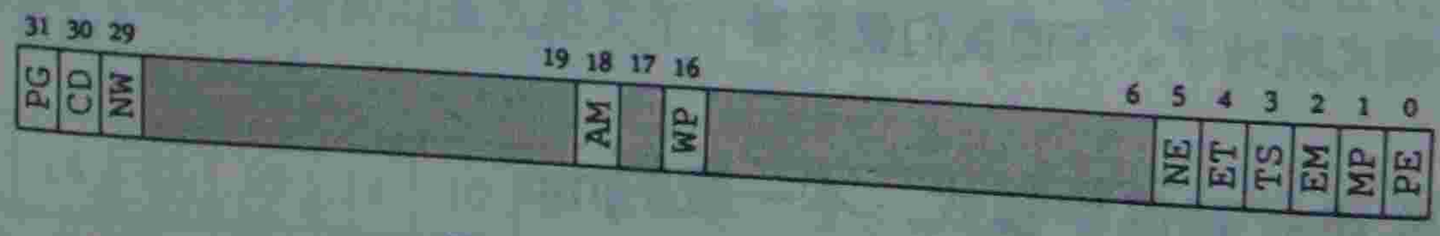


图 3.8 cr0（灰色表示保留位）

寄存器cr0的第 0 位是 PE 位，此位为 0 时，CPU 运行于实模式，为 1 时，CPU 运行于保护模式。原来我们已经闭合了进入保护模式的开关，也就是

说，“mov cr0, eax”这一句之后，系统就运行于保护模式之下了。但是，此时cs的值仍然是实模式下的值，我们需要把代码段的选择子装入cs。所以，我们需要第71行的jmp指令：

```
jmp     dword SelectorCode32:0
```

根据寻址机制我们知道，这个跳转的目标将是描述符DESC_CODE32对应的段的首地址，即标号LABEL_SEG_CODE32处。

至此，新皇帝登基，我们进入保护模式。

不过，这个jmp比看起来还要复杂一点，因为它不得不放在16位的段中，目标地址却是32位的。从这一点来看，它是混和16位和32位的代码（Mixing 16 and 32 Bit Code）。所以，这个jmp跟一般的jmp是很不相同的，直接这样写是不严谨的：

```
jmp     SelectorCode32:0 ; 错!
```

这样编译出来的只是16位的代码。如果目标地址的偏移不是0，而是一个较大的值，比如jmp SelectorCode32:0x12345678，则编译后偏移会被截断，只剩下0x5678。

所以，这个特殊的跳转需要特殊的方法来处理。在Linux内核代码中，这个跳转是用DB指令直接写二进制代码的方式来实现的，而NASM显然提供了更好的解决方法，就是加一个dword，本来dword应该加在偏移之前，但NASM允许加在整个地址之前，就像我们之前做的那样。这又可以说是NASM的优点之一，也让我们充分看到开发者在细节上的用心。

至此，我们已成功进入保护模式，下面总结一下进入保护模式的主要步骤：

1. 准备GDT。
2. 用lgdt加载gdtr。
3. 打开A20。
4. 置cr0的PE位。
5. 跳转，进入保护模式。

现在，回过头看看代码3.1，是不是觉得非常简单呢？

3.1.4 描述符属性

在刚才的程序中，涉及描述符属性的地方被忽略了过去，下面我们就详细了解一下描述符的属性。其中，可能有的细节暂时还无法很好理解，没有关系，在以后遇到相关问题时可以回过头来，把这里当成一个参考。

描述符各属性详情如下（可同时参考图3.4）：

- **P位** 存在（Present）位。P=1表示段在内存中存在；P=0表示段在内存中不存在。

- **DPL** 描述符特权级（Descriptor Privilege Level）。特权级可以是 0、1、2 或者 3。数字越小特权级越大。

- **S 位** 指明描述符是数据段 / 代码段描述符（S=1）还是系统段 / 门描述符（S=0）。

- **TYPE** 描述符类型。详情见表3.1。

表 3.1 描述符类型

TYPE 值	数据段和代码段描述符	系统段和门描述符
0	只读	< 未定义 >
1	只读，已访问	可用 286TSS
2	读 / 写	LDT
3	读 / 写，已访问	忙的 286TSS
4	只读，向下扩展	286 调用门
5	只读，向下扩展，已访问	任务门
6	读 / 写，向下扩展	286 中断门
7	读 / 写，向下扩展，已访问	286 陷阱门
8	只执行	< 未定义 >
9	只执行、已访问	可用 386TSS
A	执行 / 读	< 未定义 >
B	执行 / 读、已访问	忙的 386TSS
C	只执行、一致码段	386 调用门
D	只执行、一致码段、已访问	< 未定义 >
E	执行 / 读、一致码段	386 中断门
F	执行 / 读、一致码段、已访问	386 陷阱门

- **G 位** 段界限粒度（Granularity）位。G=0 时段界限粒度为字节；G=1 时段界限粒度为 4KB。

- **D/B 位** 这一位比较复杂，分三种情况。

  - 在可执行代码段描述符中，这一位叫做 D 位。D=1 时，在默认情况下指令使用 32 位地址及 32 位或 8 位操作数；D=0 时，在默认情况下使用 16 位地址及 16 位或 8 位操作数。
  - 在向下扩展数据段的描述符中，这一位叫做 B 位。B=1 时，段的上部界限为 4GB；B=0 时，段的上部界限为 64KB。
  - 在描述堆栈段（由ss寄存器指向的段）的描述符中，这一位叫做 B 位。B=1 时，隐式的堆栈访问指令（如push、pop和call）使用 32 位堆栈指针寄存器esp；D=0 时，隐式的堆栈访问指令（如push、pop和call）使用 16 位堆栈指针寄存器sp。

- **AVL 位** 保留位，可以被系统软件使用。

表 3.2 部分术语中英文对照

英文	中文
Descriptor	描述符
Privilege	特权级
Accessed	已访问
Conforming Code Segment	一致码段
Expand-down	向下扩展
Granularity	粒度
Call Gate	调用门
Interrupt Gate	中断门
Trap Gate	陷阱门
Implicit	隐式的

为避免由语言产生的歧义，表3.2给出了本文中相关的主要术语的中英文对照。

到目前为止我们遇到的术语的字面意思大部分都比较容易理解，只是“一致码段”中“一致”这个词比较令人费解。“一致”的意思是这样的，当转移的目标是一个特权级更高的一致代码段，当前的特权级会被延续下去，而向特权级更高的非一致代码段的转移会引起常规保护错误（general-protection exception，#GP），除非使用调用门或者任务门。如果系统代码不访问受保护的资源和某些类型的异常处理（比如，除法错误或溢出错误），它可以被放在一致代码段中。为避免低特权级的程序访问而被保护起来的系统代码则应放到非一致代码段中。

要注意的是，如果目标代码的特权级低的话，无论它是不是一致代码段，都不能通过call或者jmp转移进去，尝试这样的转移将会导致常规保护错误。

所有的数据段都是非一致的，这意味着不可能被低特权级的代码访问到。然而，与代码段不同的是，数据段可以被更高特权级的代码访问到，而不需要使用特定的门。

总之，通过call和jmp的转移遵从表3.3所示的规则。

表 3.3 一致与非一致

	特权级 低 → 高	特权级 高 → 低	相同特权 级之间	适用于何种代码
一致代码段	Yes	No	Yes	不访问受保护的资源和某些类型的异常处理的系统代码
非一致代码段	No	No	Yes	避免低特权级的程序访问而被保护起来的系统代码
数据段 （总是非一致）	No	Yes	Yes	

好了，关于描述符属性以及相关的内容先介绍到这里，现在让我们回去看

一下，我们的例子中到底设置了怎样的段属性。

代码段的属性是“DA_C+DA_32”，根据pm.inc中的定义（见代码3.3），DA_C是98h，对应的二进制是10011000b。也就是说，P 位是 1 表明这个段在内存中存在，S 位是 1 表明这个段是代码段或者数据段，TYPE=8 表明这个段是个代码段，且是只执行的代码段。DA_32是4000h，由于这个段是代码段，D 位是 1 表明这个段是 32 位的代码段。所以，这个段是存在的只执行的 32 位代码段，DPL 为 0。

代码 3.3 描述符类型（节自 chapter3/a/pm.inc）

```
193 ; 描述符类型
194 DA_32           EQU     4000h   ; 32 位段
195
196 DA_DPL0         EQU       00h   ; DPL = 0
197 DA_DPL1         EQU       20h   ; DPL = 1
198 DA_DPL2         EQU       40h   ; DPL = 2
199 DA_DPL3         EQU       60h   ; DPL = 3
200
201 ; 存储段描述符类型
202 DA_DR           EQU     90h     ; 存在的只读数据段类型值
203 DA_DRW          EQU     92h     ; 存在的可读写数据段属性值
204 DA_DRWA         EQU     93h     ; 存在的已访问可读写数据段类型值
205 DA_C            EQU     98h     ; 存在的只执行代码段属性值
206 DA_CR           EQU     9Ah     ; 存在的可执行可读代码段属性值
207 DA_CCO          EQU     9Ch     ; 存在的只执行一致代码段属性值
208 DA_CCOR         EQU     9Eh     ; 存在的可执行可读一致代码段属性值
209
210 ; 系统段描述符类型
211 DA_LDT          EQU       82h   ; 局部描述符表段类型值
212 DA_TaskGate     EQU       85h   ; 任务门类型值
213 DA_386TSS       EQU       89h   ; 可用 386 任务状态段类型值
214 DA_386CGate     EQU       8Ch   ; 386 调用门类型值
215 DA_386IGate     EQU       8Eh   ; 386 中断门类型值
216 DA_386TGate     EQU       8Fh   ; 386 陷阱门类型值
```

类似地，VIDEO 段是存在的可读写数据段。

3.2 保护模式进阶

我们虽然成功进入了保护模式，但是并没有体验到保护模式带给我们的便利，上一个例子中打印了一个红色的 P，这在实模式中也可以容易地做到。可是，显然，保护模式能做的远不止如此，下面，我们就来慢慢体验一下保护模式带来的无限便利。

3.2.1 海阔凭鱼跃

上文中我们提到，在保护模式下寻址空间可以达到 4GB，这无疑让人感到激动，实模式下 1MB 的寻址能力差得太远了。那么下面，我们就把程序稍做修改，体验一下它对超过 1MB 内存的访问能力。

另外，上面的程序为了让代码最短，进入保护模式之后开始死循环，除了重启系统外没有其他办法，之所以这样做是因为笔者想在开始的时候用最简短的代码把重点突出出来，不至于让读者陷入与不重要和次重要的内容的纠缠之

中，从而理解起来更加快速、容易。但实际上，在 DOS 下完全可以在程序结束时跳回实模式，下面就让程序优雅些，在程序的末尾跳回实模式，使之有头有尾。

首先试验一下读写大地址内存。在前面程序的基础上，新建一个段，这个段以 5MB 为基址，远远超出实模式下 1MB 的界限。我们先读出开始处 8 字节的内容，然后写入一个字符串，再从中读出 8 字节。如果读写成功的话，两次读出的内容应该是不同的，而且第二次读出的内容应该是我们写进的字符串。字符串是保存在数据段中的，也是新增加的。增加的这两个段见代码3.4第18行和第20行，相应的选择子见第32行和第34行。

代码 3.4 GDT、数据段和堆栈段（节自 chapter3/b/pmtest2.asm）

```
11  [SECTION .gdt]
12  ; GDT
13  ;                                段基址,         段界限 , 属性
14  LABEL_GDT:          Descriptor       0,                0, 0           ; 空描述符
15  LABEL_DESC_NORMAL:  Descriptor       0,            0ffffh, DA_DRW     ; Normal 描述符
16  LABEL_DESC_CODE32:  Descriptor       0, SegCode32Len-1, DA_C+DA_32; 非一致代码段, 32
17  LABEL_DESC_CODE16:  Descriptor       0,            0ffffh, DA_C       ; 非一致代码段, 16
18  LABEL_DESC_DATA:    Descriptor       0,       DataLen-1, DA_DRW     ; Data
19  LABEL_DESC_STACK:   Descriptor       0,      TopOfStack, DA_DRWA+DA_32; Stack, 32 位
20  LABEL_DESC_TEST:    Descriptor 0500000h,          0ffffh, DA_DRW
21  LABEL_DESC_VIDEO:   Descriptor  0B8000h,          0ffffh, DA_DRW     ; 显存首地址
22  ; GDT 结束
23
24  GdtLen          equ     $ - LABEL_GDT   ; GDT长度
25  GdtPtr          dw      GdtLen - 1      ; GDT界限
26                  dd      0               ; GDT基地址
27
28  ; GDT 选择子
29  SelectorNormal          equ     LABEL_DESC_NORMAL       - LABEL_GDT
30  SelectorCode32          equ     LABEL_DESC_CODE32       - LABEL_GDT
31  SelectorCode16          equ     LABEL_DESC_CODE16       - LABEL_GDT
32  SelectorData            equ     LABEL_DESC_DATA         - LABEL_GDT
33  SelectorStack           equ     LABEL_DESC_STACK        - LABEL_GDT
34  SelectorTest            equ     LABEL_DESC_TEST         - LABEL_GDT
35  SelectorVideo           equ     LABEL_DESC_VIDEO        - LABEL_GDT
36  ; END of [SECTION .gdt]
37
38  [SECTION .data1]         ; 数据段
39  ALIGN   32
40  [BITS   32]
41  LABEL_DATA:
42  SPValueInRealMode       dw      0
43  ; 字符串
44  PMMessage:              db      "In␣Protect␣Mode␣now.␣^-^", 0    ; 在保护模式中显示
45  OffsetPMMessage         equ     PMMessage - $$
46  StrTest:                db      "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 0
47  OffsetStrTest           equ     StrTest - $$
48  DataLen                 equ     $ - LABEL_DATA
49  ; END of [SECTION .data1]
50
51
52  ; 全局堆栈段
53  [SECTION .gs]
54  ALIGN   32
55  [BITS   32]
56  LABEL_STACK:
57          times 512 db 0
58
59  TopOfStack      equ     $ - LABEL_STACK - 1
60
61  ; END of [SECTION .gs]
```

接着来看段[SECTION .s32]（代码3.5），这个段的开头初始化了ds、es和gs，让ds指向新增的数据段，es指向新增的 5MB 内存的段，gs指向显存（第167行到第172行）。接着显示一行字符串，之后就开始读写大地址内存了（第198行到第200行）。由于要读两次相同的内存，我们把读的过程写进一个函数TestRead，写内存的内容也写进函数TestWrite，这两个函数的入口分别在第206行和第222行。可以看到，在TestRead中还调用了DispAL和DispReturn这两个函数（第253行和第286行），DispAL将al中的字节用十六进制数形式显示出来，字的前景色仍然是红色；DispReturn模拟一个回车的显示，实际上是让下一个字符显示在下一行的开头处。要注意的一个细节是，在程序的整个执行过程中，edi始终指向要显示的下一个字符的位置。所以，如果程序中除显示字符外还用到edi，需要事先保存它的值，以免在显示时产生混乱。

代码 3.5 chapter3/b/pmtest2.asm 中的 32 位代码段

```
166 LABEL_SEG_CODE32:
167         mov     ax, SelectorData
168         mov     ds, ax                  ; 数据段选择子
169         mov     ax, SelectorTest
170         mov     es, ax                  ; 测试段选择子
171         mov     ax, SelectorVideo
172         mov     gs, ax                  ; 视频段选择子
173
174         mov     ax, SelectorStack
175         mov     ss, ax                  ; 堆栈段选择子
176
177         mov     esp, TopOfStack
178
179
180         ; 下面显示一个字符串
181         mov     ah, 0Ch                 ; 0000: 黑底    1100: 红字
182         xor     esi, esi
183         xor     edi, edi
184         mov     esi, OffsetPMMessage    ; 源数据偏移
185         mov     edi, (80 * 10 + 0) * 2  ; 目的数据偏移。屏幕第 10 行, 第 0 列。
186         cld
187 .1:
188         lodsb
189         test    al, al
190         jz      .2
191         mov     [gs:edi], ax
192         add     edi, 2
193         jmp     .1
194 .2:     ; 显示完毕
195
196         call    DispReturn
197
198         call    TestRead
199         call    TestWrite
200         call    TestRead
201
202         ; 到此停止
203         jmp     SelectorCode16:0
204 ; ------------------------------------------------------------------------
205 TestRead:
206         xor     esi, esi
207         mov     ecx, 8
208 .loop:
209         mov     al, [es:esi]
210         call    DispAL
211         inc     esi
212
```

```
        loop    .loop

        call    DispReturn

        ret
; TestRead 结束------------------------------------------------------------

; ------------------------------------------------------------------------
TestWrite:
        push    esi
        push    edi
        xor     esi, esi
        xor     edi, edi
        mov     esi, OffsetStrTest      ; 源数据偏移
        cld
.1:
        lodsb
        test    al, al
        jz      .2
        mov     [es:edi], al
        inc     edi
        jmp     .1
.2:

        pop     edi
        pop     esi

        ret
; TestWrite 结束-----------------------------------------------------------


; ------------------------------------------------------------------------
; 显示 AL 中的数字
; 默认地:
;       数字已经存在 AL 中
;       edi 始终指向要显示的下一个字符的位置
; 被改变的寄存器:
;       ax, edi
; ------------------------------------------------------------------------
DispAL:
        push    ecx
        push    edx

        mov     ah, 0Ch                 ; 0000: 黑底    1100: 红字
        mov     dl, al
        shr     al, 4
        mov     ecx, 2
.begin:
        and     al, 01111b
        cmp     al, 9
        ja      .1
        add     al, '0'
        jmp     .2
.1:
        sub     al, 0Ah
        add     al, 'A'
.2:
        mov     [gs:edi], ax
        add     edi, 2

        mov     al, dl
        loop    .begin
        add     edi, 2

        pop     edx
        pop     ecx
```

```
280
281             ret
282     ; DispAL 结束----------------------------------------------------------
283
284
285     ; ---------------------------------------------------------------------
286     DispReturn:
287             push    eax
288             push    ebx
289             mov     eax, edi
290             mov     bl, 160
291             div     bl
292             and     eax, 0FFh
293             inc     eax
294             mov     bl, 160
295             mul     bl
296             mov     edi, eax
297             pop     ebx
298             pop     eax
299
300             ret
301     ; DispReturn 结束------------------------------------------------------
```

在TestWrite中用到一个常量OffsetStrTest，它的定义在代码3.4第47行。注意，我们用到这个字符串的时候并没有用直接标号StrTest，而是又定义了一个符号OffsetStrTest，它等于StrTest-$$。还记得我们在第 1 章中提到过的$$的含义吗？它代表当前节（section）开始处的地址。所以StrTest-$$表示字符串StrTest相对于本节的开始处（即LABEL_DATA处）的偏移。我们来看一下代码3.6，容易发现数据段的基址便是LABEL_DATA的物理地址。于是OffsetStrTest既是字符串相对于LABEL_DATA的偏移，也是其在数据段中的偏移。我们在保护模式下需要用到的正是这个偏移，而不再是实模式下的地址。前文中提到过的 section 的一点妙用指的便是这里的$$，它不是没有替代品，而是这样做思路会比较清晰。OffsetPMMessage的情形与此类似。

代码 3.6 初始化数据段描述符（节自 chapter3/b/pmtest2.asm）

```
96              ; 初始化数据段描述符
97              xor     eax, eax
98              mov     ax, ds
99              shl     eax, 4
100             add     eax, LABEL_DATA
101             mov     word [LABEL_DESC_DATA + 2], ax
102             shr     eax, 16
103             mov     byte [LABEL_DESC_DATA + 4], al
104             mov     byte [LABEL_DESC_DATA + 7], ah
```

另外，由于在保护模式下用到了堆栈，我们建立了一个堆栈段，见代码3.4第53行到第59行。它对应的描述符在第19行，注意属性里包含DA_32，表明它是 32 位的堆栈段。

在[SECTION .s32]中我们还改变了ss和esp（代码3.5第174行到177行），这样，在 32 位代码段中所有的堆栈操作将会在新增的堆栈段中进行。

至此，我们的程序已经可以运行了。但正如前面所提到的，如果不能有始有终地从保护模式返回实模式显然还不够，现在我们就来增加一点返回实模式的代码。

我们从实模式进入保护模式时直接用一个跳转就可以了，但是返回的时候

却稍稍复杂一些。因为在准备结束保护模式回到实模式之前，需要加载一个合适的描述符选择子到有关段寄存器，以使对应段描述符高速缓冲寄存器中含有合适的段界限和属性。而且，我们不能从32位代码段返回实模式，只能从16位代码段中返回。这是因为无法实现从32位代码段返回时cs高速缓冲寄存器中的属性符合实模式的要求（实模式不能改变段属性）。

所以，在这里，我们新增一个Normal描述符（代码3.4第15行）。在返回实模式之前把对应选择子SelectorNormal加载到ds、es和ss，就是上面所说的这个原因。

好了，现在看一下这个返回实模式前用到的16位的段，见代码3.7。

代码3.7 保护模式到实模式（节自chapter3/b/pmtest2.asm）

```
308 [SECTION .s16code]
309 ALIGN   32
310 [BITS   16]
311 LABEL_SEG_CODE16:
312         ; 跳回实模式:
313         mov     ax, SelectorNormal
314         mov     ds, ax
315         mov     es, ax
316         mov     fs, ax
317         mov     gs, ax
318         mov     ss, ax
319
320         mov     eax, cr0
321         and     al, 11111110b
322         mov     cr0, eax
323
324 LABEL_GO_BACK_TO_REAL:
325         jmp     0:LABEL_REAL_ENTRY      ; 段地址会在程序开始处被设置成正确的值
326
327 Code16Len       equ     $ - LABEL_SEG_CODE16
328
329 ; END of [SECTION .s16code]
```

这个段是由[SECTION .s32]中的jmp SelectorCode16:0跳进来的，这句跳转不必多讲。我们来看一下这个段，开头的语句把SelectorNormal赋给ds、es、fs、gs和ss，完成我们刚刚提到的使命。然后就清cr0的PE位，接下来的跳转看上去好像不太对，因为段地址是0。其实这里只是暂时这样写罢了，在程序的一开始处可以看到代码3.8中的这几句。

代码3.8 保护模式到实模式的准备工作（节自chapter3/b/pmtest2.asm）

```
67          mov     ax, cs
68          mov     ds, ax
69          mov     es, ax
70          mov     ss, ax
71          mov     sp, 0100h
72
73          mov     [LABEL_GO_BACK_TO_REAL+3], ax
```

mov [LABEL_GO_BACK_TO_REAL+3], ax的作用就是为回到实模式的这个跳转指令指定正确的段地址，这条指令的机器码如图3.9所示。

图3.9告诉我们，LABEL_GO_BACK_TO_REAL+3恰好就是Segment的地址，而第73行执行之前ax的值已经是实模式下的cs（我们记做cs_real_mode）了，所以它将把cs保存到Segment的位置，等到jmp指令执行时，它已经不再是：

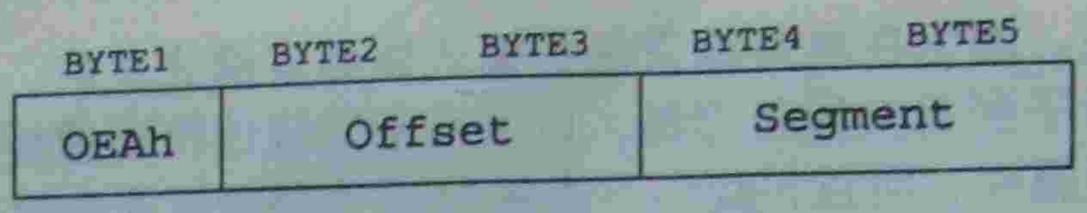


图 3.9 实模式下长跳转指令图示

```
jmp     0:LABEL_REAL_ENTRY
```

而变成了：

```
jmp     cs_real_mode:LABEL_REAL_ENTRY
```

它将跳转到标号LABEL_REAL_ENTRY处。

在跳回实模式之后，程序重新设置各个段寄存器的值，恢复sp的值，然后关闭 A20，打开中断，重新回到原来的样子（见代码3.9）。

代码 3.9 回到实模式（节自 chapter3/b/pmtest2.asm）

```
144 LABEL_REAL_ENTRY:                   ; 从保护模式跳回到实模式就到了这里
145         mov     ax, cs
146         mov     ds, ax
147         mov     es, ax
148         mov     ss, ax
149
150         mov     sp, [SPValueInRealMode]
151
152         in      al, 92h             ; `.
153         and     al, 11111101b       ;  | 关闭 A20 地址线
154         out     92h, al             ; /
155
156         sti                         ; 开中断
157
158         mov     ax, 4c00h           ; `.
159         int     21h                 ; /  回到 DOS
```

好了，整个程序现在大功告成了（中间诸如初始化 16 位段描述符的代码等容易实现的内容略过不提），编译：

```
▷ nasm pmtest2.asm -o pmtest2.com
```

运行，结果如图3.10所示。

我们清晰地看到，程序打印出两行数字，第一行全部是零，说明开始内存地址 5MB 处都是 0，而下一行已经变成了 41 42 43…，说明写操作成功。十六进制的 41、42、43、…、48 正是 A、B、C、…、H。

同时看到，程序执行结束后不再像上一个程序那样进入死循环，而是重新出现了 DOS 提示符。这说明我们重新回到了实模式下的 DOS。

3.2.2 LDT（Local Descriptor Table）

LDT 这个名字你一定已经不再陌生，尽管我们已经提到它好几次，却到现在也没仔细研究过。其实，看名字我们就知道，它跟 GDT 差不多，都是描述

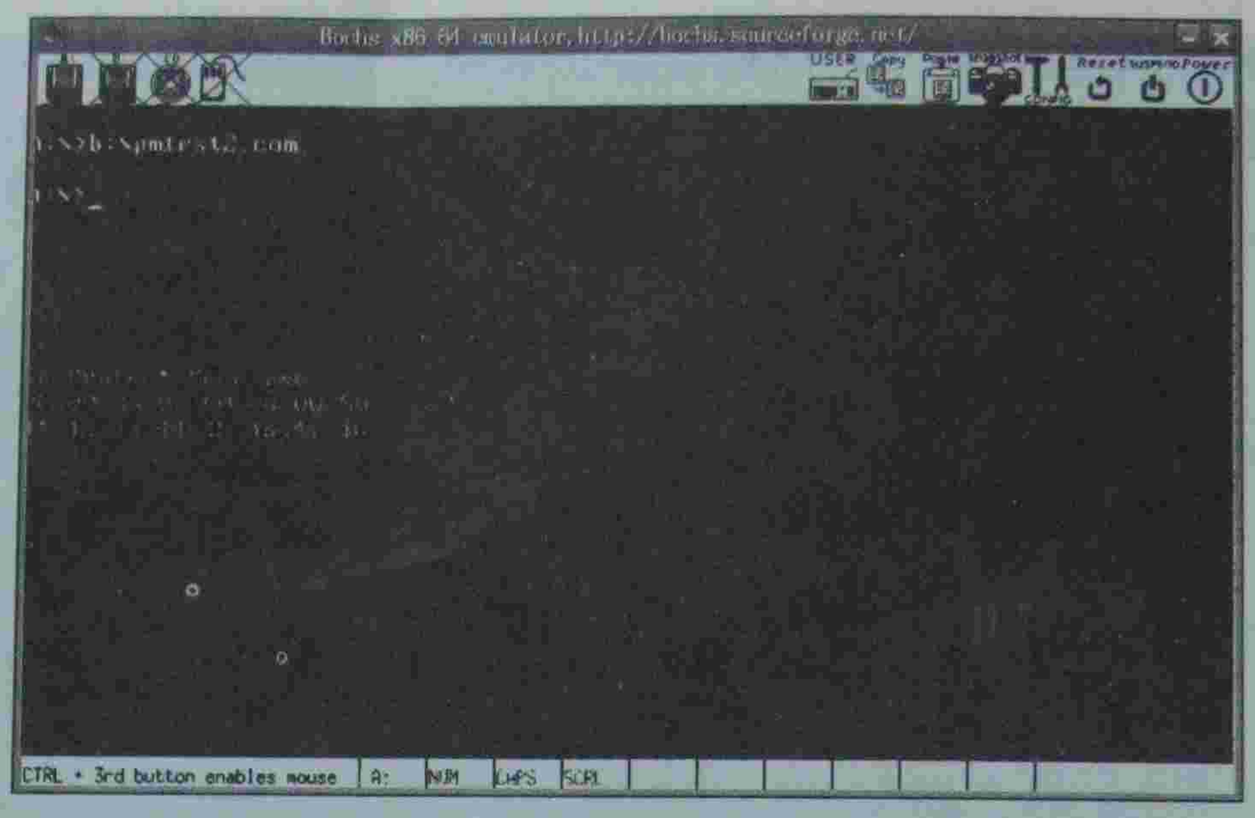


图 3.10 pmtest2 的执行结果

符表（Descriptor Table），区别仅仅在于全局（Global）和局部（Local）的不同。我们还是先有一些感性认识，在原先程序的基础上增加一点代码。

需要说明的一点是，为了节省篇幅，这里尽量只列出新增加的代码，但是，列出的代码太少也不利于阅读和理解。所以，如果读者觉得代码列得太多了或者太少了，请予以谅解。读者最好边阅读本书边参考本书附带的光盘中的代码。

这一小节对应的代码是pmtest3.asm，我们来看看这个程序中增加了什么，请看代码3.10。

代码 3.10 ldt（节自 chapter3/c/pmtest3.asm）

```
11  [SECTION .gdt]
     ...
20  LABEL_DESC_LDT:    Descriptor      0,        LDTLen - 1, DA_LDT        ; LDT
     ...
34  SelectorLDT             equ     LABEL_DESC_LDT          - LABEL_GDT
     ...
64  [SECTION .s16]
     ...
116         ; 初始化 LDT 在 GDT 中的描述符
117         xor     eax, eax
118         mov     ax, ds
119         shl     eax, 4
120         add     eax, LABEL_LDT
121         mov     word [LABEL_DESC_LDT + 2], ax
122         shr     eax, 16
123         mov     byte [LABEL_DESC_LDT + 4], al
124         mov     byte [LABEL_DESC_LDT + 7], ah
125
126         ; 初始化 LDT 中的描述符
127         xor     eax, eax
128         mov     ax, ds
129         shl     eax, 4
130         add     eax, LABEL_CODE_A
131         mov     word [LABEL_LDT_DESC_CODEA + 2], ax
132         shr     eax, 16
133         mov     byte [LABEL_LDT_DESC_CODEA + 4], al
134         mov     byte [LABEL_LDT_DESC_CODEA + 7], ah
     ...
```

```
183 [SECTION .s32]; 32 位代码段. 由实模式跳入.
    ...
216         ; Load LDT
217         mov     ax, SelectorLDT
218         lldt    ax
219
220         jmp     SelectorLDTCodeA:0      ; 跳入局部任务
    ...
269 ; LDT
270 [SECTION .ldt]
271 ALIGN   32
272 LABEL_LDT:
273 ;                                   段基址          段界限      属性
274 LABEL_LDT_DESC_CODEA: Descriptor 0, CodeALen - 1, DA_C + DA_32 ; Code, 32 位
275
276 LDTLen          equ     $ - LABEL_LDT
277
278 ; LDT 选择子
279 SelectorLDTCodeA        equ     LABEL_LDT_DESC_CODEA    - LABEL_LDT + SA_TIL
280 ; END of [SECTION .ldt]
281
282
283 ; CodeA (LDT, 32 位代码段)
284 [SECTION .la]
285 ALIGN   32
286 [BITS   32]
287 LABEL_CODE_A:
288         mov     ax, SelectorVideo
289         mov     gs, ax                  ; 视频段选择子(目的)
290
291         mov     edi, (80 * 12 + 0) * 2  ; 屏幕第 10 行, 第 0 列。
292         mov     ah, 0Ch                 ; 0000: 黑底    1100: 红字
293         mov     al, 'L'
294         mov     [gs:edi], ax
295
296         ; 准备经由16位代码段跳回实模式
297         jmp     SelectorCode16:0
298 CodeALen        equ     $ - LABEL_CODE_A
299 ; END of [SECTION .la]
```

我们看到，代码增加得并不多，而且，结构还是很清晰的。先是在 GDT 中增加了一个描述符，当然还有与描述符对应的选择子，以及对这个描述符的初始化代码。另外，还增加了两个节，其中一个是新的描述符表，也就是 LDT，另一个是代码段，对应新增的 LDT 中的一个描述符。

我们来看一下第217行到第220行。你可能发现一个看起来有点面熟的指令lldt，它不但长得有点像lgdt，而且功能也差不多，负责加载ldtr，它的操作数是一个选择子，这个选择子对应的就是用来描述 LDT 的那个描述符（标号LABEL_DESC_LDT）。

你也看到了，LDT 跟 GDT 差不多，本例用到的 LDT 中只有一个描述符（标号LABEL_LDT_DESC_CODEA处），这个描述符跟 GDT 中的描述符没什么分别，可是选择子却不一样，多出了一个属性SA_TIL。可以在pm.inc中找到它的定义：

```
SA_TIL EQU 4
```

由图3.5可知，SA_TIL将选择子SelectorLDTCodeA的 TI 位置为 1。在前面几节的代码中我们从未用过这一位，因为我们之前并未涉及过 LDT，实际上，这一位便是区别 GDT 的选择子和 LDT 的选择子的关键所在。如果 TI 被

置位，那么系统将从当前 LDT 中寻找相应描述符。也就是说，当代码3.10中用到SelectorLDTCodeA时，系统会从 LDT 中找到LABEL_LDT_DESC_CODEA描述符，并跳转到相应的段中。

本例的 LDT 中的代码段也很简单，只是打印一个字符“L”。不难想像，在[SECTION .s32]中打印完“In Protect Mode now.”这个字符串后，一个红色的字符 L 将会出现。程序的运行结果如图3.11所示。

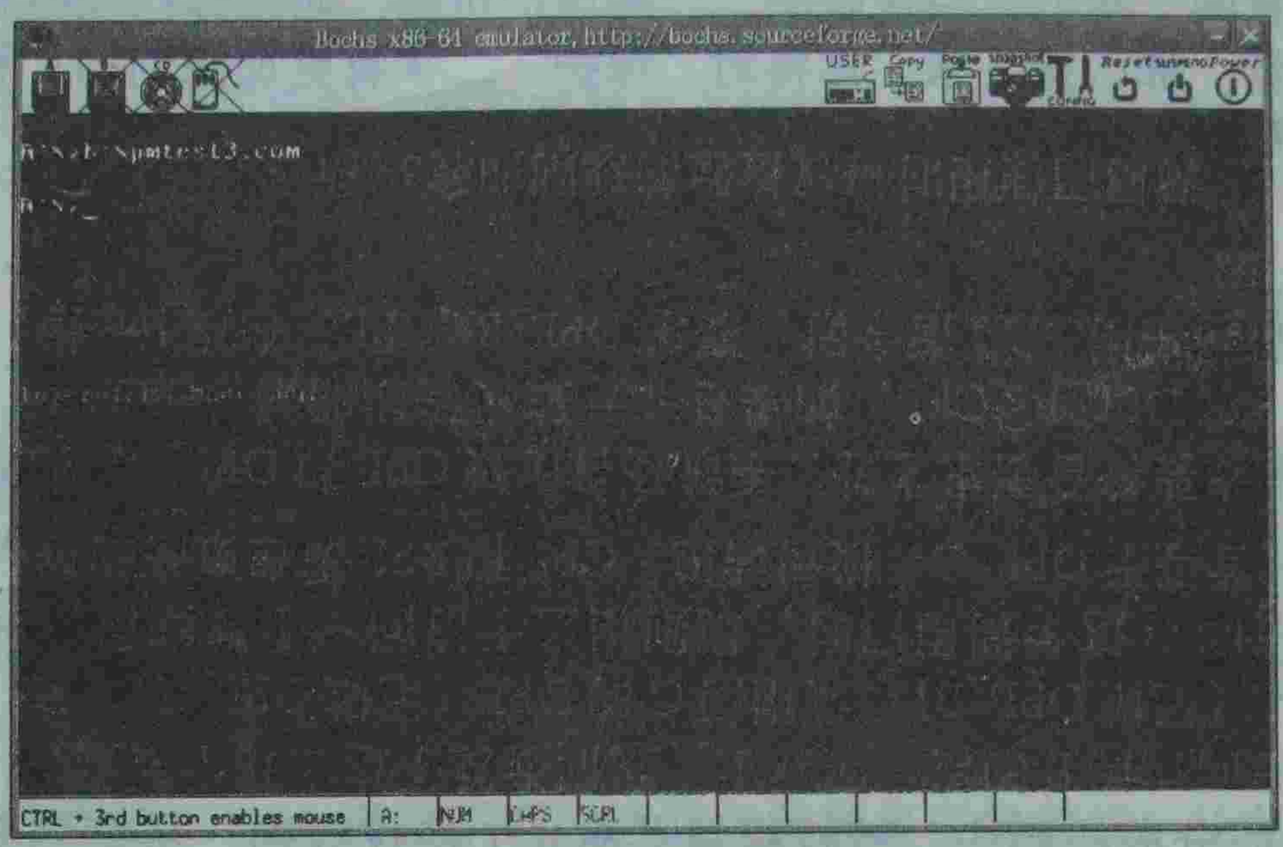


图 3.11 pmtest3 的执行结果

现在，你对于 LDT 是不是已经有了大致的了解了呢？简单来说，它是一种描述符表，与 GDT 差不多，只不过它的选择子的 TI 位必须置为 1。在运用它时，需要先用lldt指令加载ldtr，lldt的操作数是 GDT 中用来描述 LDT 的描述符。

上例的 LDT 很简单，只有一个代码段。不难想像，我们还可以在其中增加更多的段，比如数据段、堆栈段等，这样一来，我们可以把一个单独的任务所用到的所有东西封装在一个 LDT 中，这种思想是我们在后面章节中的多任务处理的一个雏形。如果读者有兴趣，可以按照下面的步骤增加一个用 LDT 描述的任务：

1. 增加一个 32 位的代码段，内容不妨尽量简单。
2. 增加一个段，内容是一个描述符表（LDT），可以只有一个代码段描述符，也可以添加更多的描述符描述更多的段。注意，涉及的选择子的 TI 位是 1。
3. 在 GDT 中增加一个描述符，用以描述这个 LDT，同时要定义其描述符。
4. 增加新添的描述符的初始化代码，主要是针对段基址。
5. 用新加的 LDT 描述的局部任务准备完毕。
6. 使用前用lldt指令加载ldtr，用jmp指令跳转等方式运行。

通过几个简单的例子，我们对 IA32 的分段机制大致已经有所了解了。现在，我们来提出一个问题，“保护模式”中“保护”二字到底是什么含义？

我们已经看到，在描述符中段基址和段界限定义了一个段的范围，对超越段界限之外的地址的访问是被禁止的，这无疑是对段的一种保护。另外，有点复杂的段属性作为对一个段各个方面的定义规定和限制了段的行为和性质，从功能上来讲，这仍然是一种保护。

不知不觉间，我们已经接触到了一些保护机制，如果你已经有初窥保护模式门径的感觉，那么接下来，你将对“保护”二字有更加深刻的了解，因为下面我们即将介绍的是特权级。

3.2.3 特权级概述

对于“特权级”这个词，你可能仍然感到有些陌生，但如果提起 DPL 和 RPL，你一定仍有印象。实际上，DPL 所代表的 Descriptor Privilege Level 以及 RPL 所代表的 Requested Privilege Level 都是用来表示特权级别的。只不过，前面所有的例子都是运行在最高特权级下，所以涉及到的 DPL 和 RPL 都是 0（最高特权级）。

在 IA32 的分段机制中，特权级总共有 4 个特权级别，从高到低分别是 0、1、2、3。数字越小表示的特权级越大。

如图3.12所示，较为核心的代码和数据，将被放在特权级较高的层级中。处理器将用这样的机制来避免低特权级的任务在不被允许的情况下访问位于高特权级的段。如果处理器检测到一个访问请求是不合法的，将会产生常规保护错误（#GP）。

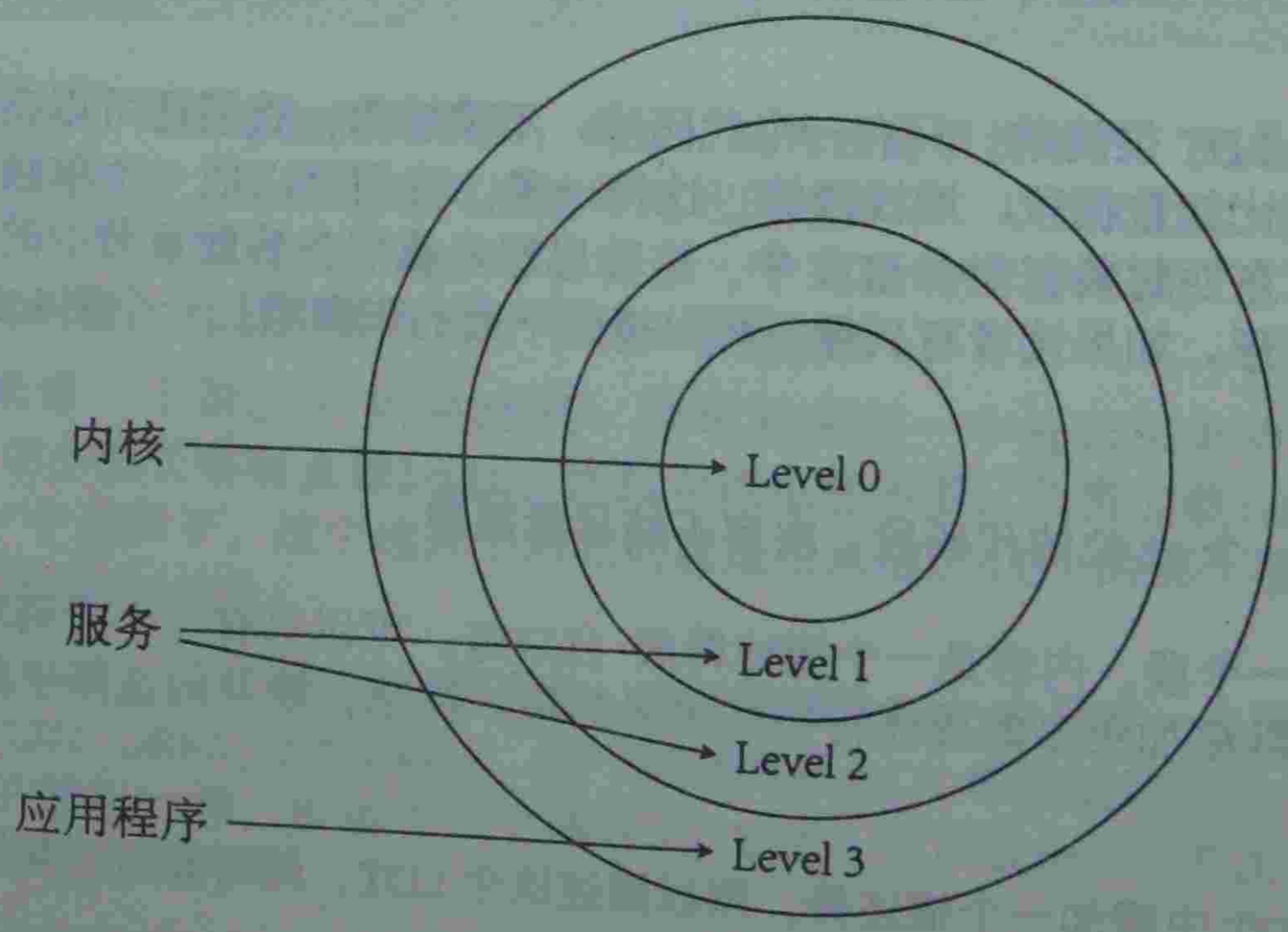


图 3.12 特权级

注意，由于数字越大表示的特权级越小，所以有时为避免混淆，也将高特权级称做内层，而把低特权级称做外层。

3.2.3.1 CPL、DPL、RPL

处理器通过识别 CPL、DPL、RPL 这 3 种特权级进行特权级检验。

1. **CPL（Current Privilege Level）**

CPL 是当前执行的程序或任务的特权级。它被存储在cs和ss的第 0 位和第 1 位上。在通常情况下，CPL 等于代码所在的段的特权级。当程序转移到不同特权级的代码段时，处理器将改变 CPL。

在遇到一致代码段时，情况稍稍有点特殊，一致代码段可以被相同或者更低特权级的代码访问。当处理器访问一个与 CPL 特权级不同的一致代码段时，CPL 不会被改变。

2. **DPL（Descriptor Privilege Level）**

DPL 表示段或者门的特权级。它被存储在段描述符或者门描述符的 DPL 字段中，正如我们先前所看到的那样。当当前代码段试图访问一个段或者门时，DPL 将会和 CPL 以及段或门选择子的 RPL 相比较，根据段或者门类型的不同，DPL 将会被区别对待，下面介绍一下各种类型的段或者门的情况。

- **数据段：** DPL 规定了可以访问此段的最低特权级。比如，一个数据段的 DPL 是 1，那么只有运行在 CPL 为 0 或者 1 的程序才有权访问它。
- **非一致代码段（不使用调用门的情况下）：** DPL 规定访问此段的特权级。比如，一个非一致代码段的特权级为 0，那么只有 CPL 为 0 的程序才可以访问它。
- **调用门：** DPL 规定了当前执行的程序或任务可以访问此调用门的最低特权级（这与数据段的规则是一致的）。
- **一致代码段和通过调用门访问的非一致代码段：** DPL 规定了访问此段的最高特权级。比如，一个一致代码段的 DPL 是 2，那么 CPL 为 0 和 1 的程序将无法访问此段。
- **TSS：** DPL 规定了可以访问此 TSS 的最低特权级（这与数据段的规则是一致的）。

3. **RPL（Requested Privilege Level）**

RPL 是通过段选择子的第 0 位和第 1 位表现出来的。处理器通过检查 RPL 和 CPL 来确认一个访问请求是否合法。即便提出访问请求的段有足够的特权级，如果 RPL 不够也是不行的。也就是说，如果 RPL 的数字比 CPL 大（数字越大特权级越低），那么 RPL 将会起决定性作用，反之亦然。

操作系统过程往往用 RPL 来避免低特权级应用程序访问高特权级段内的数据。当操作系统过程（被调用过程）从一个应用程序（调用过程）接收到一个选择子时，将会把选择子的 RPL 设成调用者的特权级。于是，当操作系统用这个选择子去访问相应的段时，处理器将会用调用过程的特权级（已经被存到 RPL 中），而不是更高的操作系统过程的特权级（CPL）进行特权检验。这样，RPL 就保证了操作系统不会越俎代庖地代表一个程序去访问一个段，除非这个程序本身是有权限的。

3.2.3.2 一个小试验

通过这样的介绍我们知道，对于数据的访问，特权级检验还是比较简单的，只要 CPL 和 RPL 都小于被访问的数据段的 DPL 就可以了。那么，我们就先小试牛刀，把先前例子中的数据段描述符的 DPL 修改一下，看会发生什么现象。

先将LABEL_DESC_DATA对应的段描述符的 DPL 修改为 1：

```
LABEL_DESC_DATA: Descriptor 0, DataLen-1, DA_DRW+DA_DPL1
```

编译链接并运行，怎么样？跟原来一样，不是吗？这说明我们对这个段的数据访问是合法的。

继续修改，把对刚才修改过的数据段的选择子的 RPL 改为 3：

```
SelectorData equ LABEL_DESC_DATA-LABEL_GDT+SA_RPL3
```

再运行一下，发生了什么？

Bochs 重启了，系统崩溃了，在控制台你能看到这样的字样：

```
load_seg_reg(DS): RPL & CPL must be <= DPL
```

容易理解，崩溃的原因在于我们违反了特权级的规则，用 RPL=3 的选择子去访问 DPL=1 的段，于是引起异常。而我们又没有相应的异常处理模块，于是最为严重的情况就发生了。

虽然是只针对数据段的很小的试验，但我们已经对特权级这个事物有了初步的直观认识。那么，就让我们再接再厉，看一下不同特权级代码段之间的转移情况是怎样的。

3.2.3.3 不同特权级代码段之间的转移

程序从一个代码段转移到另一个代码段之前，目标代码段的选择子会被加载到cs中。作为加载过程的一部分，处理器将会检查描述符的界限、类型、特权级等内容。如果检验成功，cs将被加载，程序控制将转移到新的代码段中，从eip指示的位置开始执行。

程序控制转移的发生，可以是由指令 jmp、 call、 ret、 sysenter、 sysexit、 int n 或 iret 引起的，也可以由中断和异常机制引起。

使用jmp或call指令可以实现下列 4 种转移：

1. 目标操作数包含目标代码段的段选择子。
2. 目标操作数指向一个包含目标代码段选择子的调用门描述符。
3. 目标操作数指向一个包含目标代码段选择子的 TSS。
4. 目标操作数指向一个任务门，这个任务门指向一个包含目标代码段选择子的 TSS。

这 4 种方式可以看做是两大类，一类是通过jmp和call的直接转移（上述第1种），另一类是通过某个描述符的间接转移（上述第2、3、4种）。下面就来分别看一下。

3.2.4 特权级转移

3.2.4.1 通过jmp或call进行直接转移

我们在第3.1.4节中对通过jmp或call进行直接转移已经有过一些讨论，如果目标是非一致代码段，要求 CPL 必须等于目标段的 DPL，同时要求 RPL 小于等于 DPL；如果目标是一致代码段，则要求 CPL 大于或者等于目标段的 DPL，RPL 此时不做检查。当转移到一致代码段中后，CPL 会被延续下来，而不会变成目标代码段的 DPL。也就是说，通过jmp和call所能进行的代码段间转移是非常有限的，对于非一致代码段，只能在相同特权级代码段之间转移。遇到一致代码段也最多能从低到高，而且 CPL 不会改变。如果想自由地进行不同特权级之间的转移，显然需要其他几种方式，即运用门描述符或者 TSS。

3.2.4.2 调用门初体验

“门”这个字眼我们已看了不少，我们还没介绍它是种什么东西呢。其实，门也是一种描述符，门描述符的结构如图3.13所示。

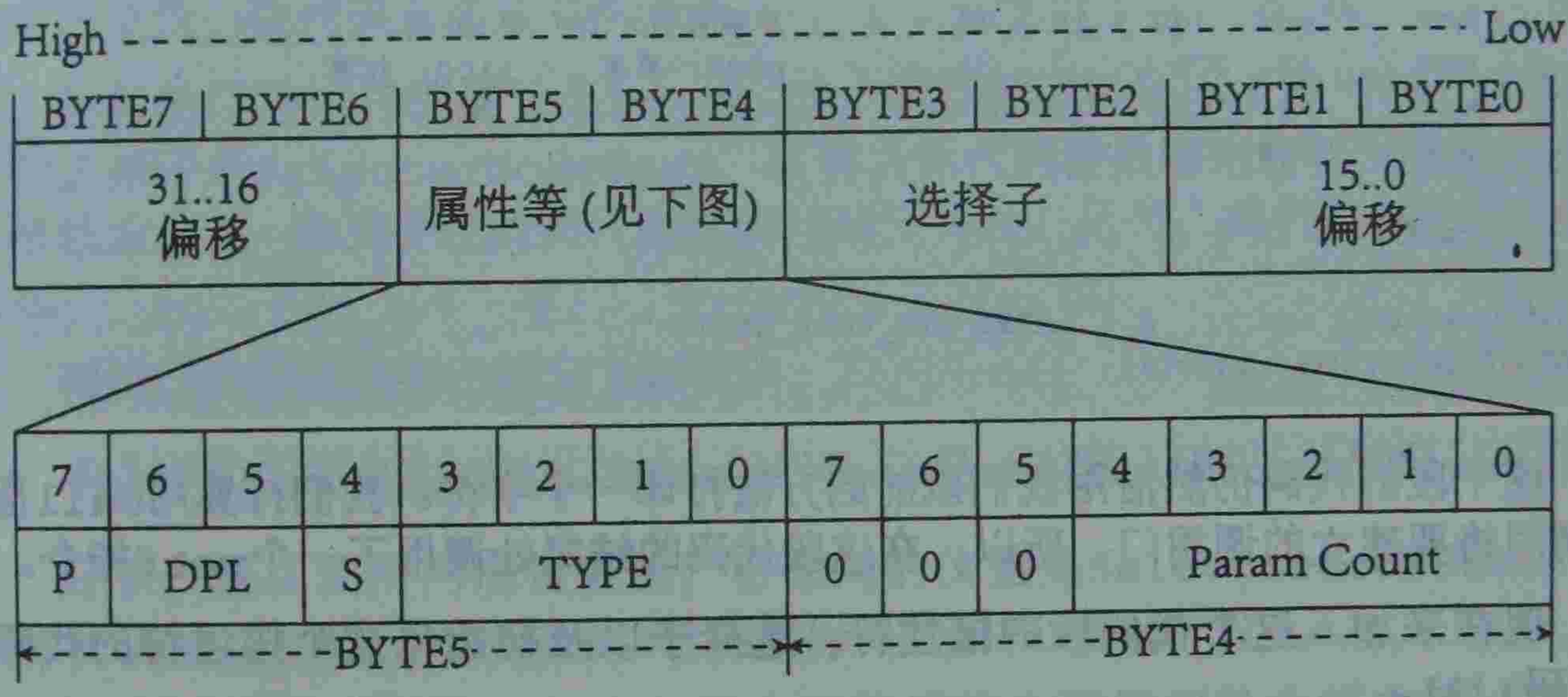


图 3.13 门描述符（Gate Descriptor）

可以看到，门描述符和我们前面提到的描述符有很大不同，它主要是定义了目标代码对应段的选择子、入口地址的偏移和一些属性等。可是，虽然这样的结构跟代码段以及数据段描述符大不相同，我们仍然看到，第 5 个字节（BYTE5）却是完全一致的，都表示属性。在这个字节内，各项内容的含义与前面提到的描述符也别无二致，这显然是必要的，以便识别描述符的类型。在这里，S 位将是 0，这跟我们在第3.1.4节里讲到的是一致的。

第 4 个字节（BYTE4）无法从直观上了解到用途，不要紧，我们先放一下，一会儿再来讨论。

门描述符的结构就是这样的，直观来看，一个门描述了由一个选择子和一个偏移所指定的线性地址，程序正是通过这个地址进行转移的。门描述符分为 4 种：

- 调用门（Call gates）

- 中断门（Interrupt gates）
- 陷阱门（Trap gates）
- 任务门（Task gates）

其中，中断门和陷阱门是特殊的调用门，将会在后面的章节中提到，我们先来介绍调用门。来看一个例子，在这个例子中，我们用到调用门。为简单起见，先不涉及任何特权级变换，而是先来关注它的工作方法。

在pmtest3.asm的基础上增加一个代码段作为通过调用门转移的目标段（代码3.11）：

代码 3.11 通过调用门转移的目标段（节自 chapter3/d/pmtest4.asm）

```
265 [SECTION .sdest]; 调用门目标段
266 [BITS   32]
267
268 LABEL_SEG_CODE_DEST:
269         ;jmp    $
270         mov     ax, SelectorVideo
271         mov     gs, ax                  ; 视频段选择子(目的)
272
273         mov     edi, (80 * 12 + 0) * 2  ; 屏幕第 12 行, 第 0 列。
274         mov     ah, 0Ch                 ; 0000: 黑底    1100: 红字
275         mov     al, 'C'
276         mov     [gs:edi], ax
277
278         retf
279
280 SegCodeDestLen  equ     $ - LABEL_SEG_CODE_DEST
281 ; END of [SECTION .sdest]
```

这个段的代码仍然沿用我们以前的方法打印一个字符。我们打算用call指令调用将要建立的调用门，所以，在这段代码的结尾处调用了一个retf指令。

现在来加入这个代码段的描述符，选择子以及初始化这个描述符的代码（代码3.12）。

代码 3.12 节自 chapter3/d/pmtest4.asm

```
18  LABEL_DESC_CODE_DEST: Descriptor 0,SegCodeDestLen-1, DA_C+DA_32; 非一致代码段,32
    ...
36  SelectorCodeDest        equ     LABEL_DESC_CODE_DEST    - LABEL_GDT
    ...
103         ; 初始化测试调用门的代码段描述符
104         xor     eax, eax
105         mov     ax, cs
106         shl     eax, 4
107         add     eax, LABEL_SEG_CODE_DEST
108         mov     word [LABEL_DESC_CODE_DEST + 2], ax
109         shr     eax, 16
110         mov     byte [LABEL_DESC_CODE_DEST + 4], al
111         mov     byte [LABEL_DESC_CODE_DEST + 7], ah
```

初始化描述符的代码想必你已经非常熟悉了，我们每初始化一个描述符都会进行这项操作，以后再添加一个描述符时也是这样，到时为了节省篇幅，类似的代码将略过不提。

好了，现在添加调用门（见代码3.13）。

代码 3.13 调用门（节自 chapter3/d/pmtest4.asm）

```
                                    目标选择子,偏移,DCount, 属性
24  ; 门
25  LABEL_CALL_GATE_TEST: Gate SelectorCodeDest,    0,      0, DA_386CGate+DA_DPL0
```

这里，我们用了一个宏Gate来初始化这个门描述符，Gate的定义在pm.inc中可以找到（代码3.14）。

代码 3.14 门描述符（节自 chapter3/d/pm.inc）

```
264 ; 门
265 ; usage: Gate Selector, Offset, DCount, Attr
266 ;        Selector:  dw
267 ;        Offset:    dd
268 ;        DCount:    db
269 ;        Attr:      db
270 %macro Gate 4
271         dw      (%2 & 0FFFFh)                           ; 偏移1
272         dw      %1                                      ; 选择子
273         dw      (%3 & 1Fh) | ((%4 << 8) & 0FF00h)       ; 属性
274         dw      ((%2 >> 16) & 0FFFFh)                   ; 偏移2
275 %endmacro ; 共 8 字节
```

这个宏和Descriptor宏有点类似，也是将描述符的构成要素分别安置在相应的位置，使代码看起来非常清晰。

我们的门描述符的属性是DA_386CGate，表明它是一个调用门。里面指定的选择子是SelectorCodeDest，表明目标代码段是刚刚新添加的代码段。偏移地址是0，表示将跳转到目标代码段的开头处。另外，我们把其DPL指定为0。

调用门对应的选择子的定义如下：

```
SelectorCallGateTest equ LABEL_CALL_GATE_TEST - LABEL_GDT
```

好了，现在调用门准备就绪，它指向的位置是SelectorCodeDest:0，即标号LABEL_SEG_CODE_DEST处的代码（代码3.11第268行）。

我们刚刚说过，用一个call指令来使用这个调用门是个好主意（见代码3.15）。

代码 3.15 使用调用门（节自 chapter3/d/pmtest4.asm）

```
233         ; 测试调用门（无特权级变换），将打印字母 'C'
⇒ 234       call    SelectorCallGateTest:0
...
241         jmp     SelectorLDTCodeA:0          ; 跳入局部任务，将打印字母 'L'.
```

这个call指令被放在进入局部任务之前，由于我们新加的代码以指令retf结尾（代码3.11第278行），所以最终代码将会跳回到call指令的下面继续执行。所以，我们最终看到的结果应该是在pmtest3.exe执行结果的基础上多出一个红色的字母C，如图3.14所示。

其实调用门这种听起来很可怕的东西本质上只不过是个入口地址，只是增加了若干的属性而已。在我们的例子中所用到的调用门完全等同于一个地址，我们甚至可以把使用调用门进行跳转的指令修改为跳转到调用门内指定的地址的指令：

```
call  SelectorCodeDest:0
```

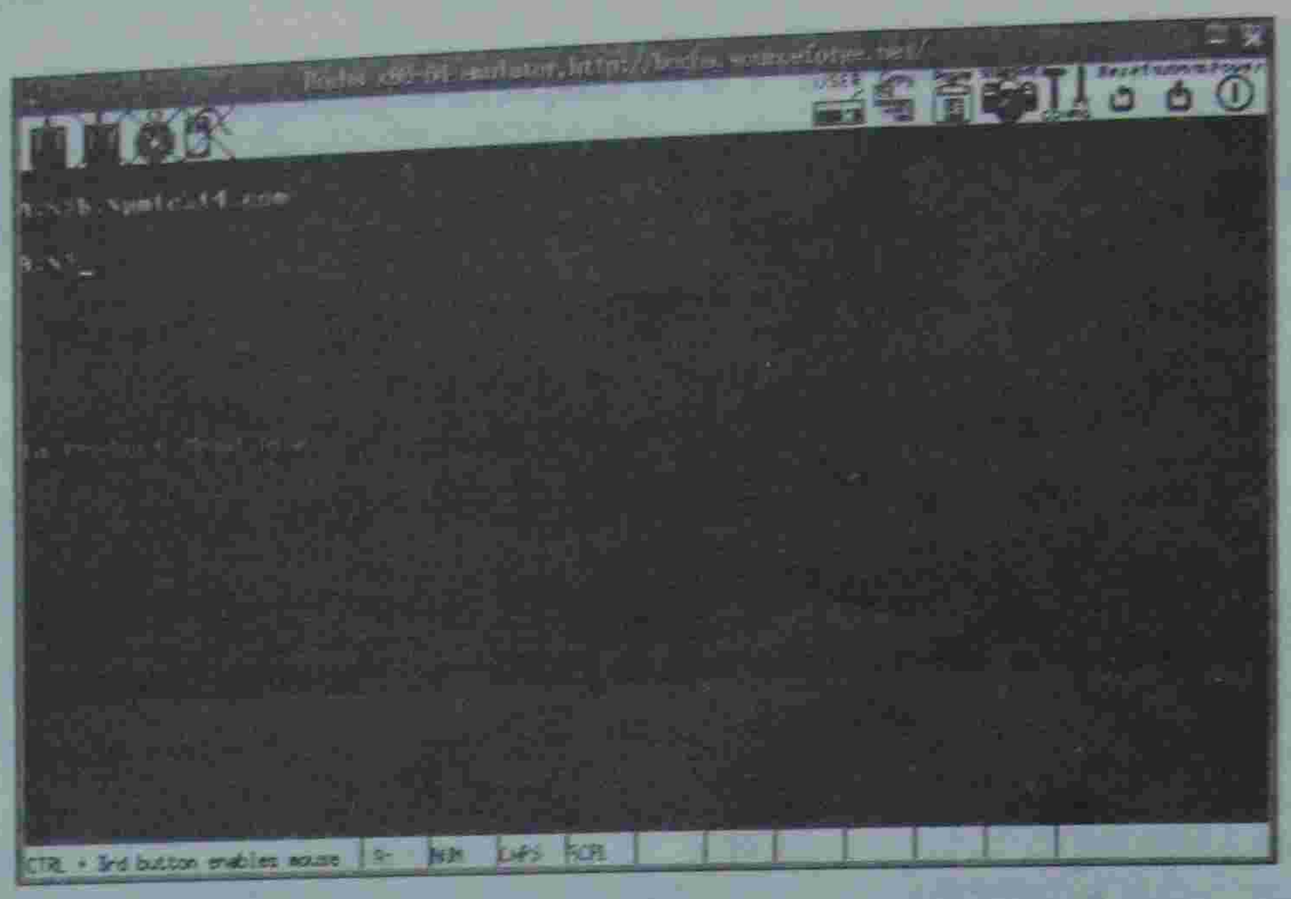


图 3.14 pmtest4 的执行结果

运行一下，效果是完全相同的。

可是，调用门显然不是多此一举的东西，因为我们将要用它来实现不同特权级的代码之间的转移。下面我们就来介绍一下使用调用门进行转移时特权级检验的规则。

假设我们想由代码 A 转移到代码 B，运用一个调用门 G，即调用门 G 中的目标选择子指向代码 B 的段。实际上，我们涉及了这么几个要素：CPL、RPL、代码 B 的 DPL（记做 DPL_B）、调用门 G 的 DPL（记做 DPL_G）。根据 3.2.3.1 中提到的，A 访问 G 这个调用门时，规则相当于访问一个数据段，要求 CPL 和 RPL 都小于或者等于 DPL_G。换句话说，CPL 和 RPL 需在更高的特权级上。

除了这一步要符合要求之外，系统还将比较 CPL 和 DPL_B。如果是一致代码段的话，要求 DPL_B≤CPL；如果是非一致代码段的话，call指令和jmp指令又有所不同。在用call指令时，要求 DPL_B≤CPL；在用jmp指令时，只能是 DPL_B=CPL。

综上所述，调用门使用时特权检验的规则如表3.4所示。

表 3.4 调用门特权级规则

	call	jmp
目标是一致代码段	CPL≤DPL_G， RPL≤DPL_G， DPL_B≤CPL	
目标是非一致代码段	CPL≤DPL_G， RPL≤DPL_G， DPL_B≤CPL	CPL≤DPL_G， RPL≤DPL_G， DPL_B=CPL

也就是说，通过调用门和call指令，可以实现从低特权级到高特权级的转移，无论目标代码段是一致的还是非一致的。

说到这里，你一定又跃跃欲试了，写一个程序实现一个特权级变换应该是件有趣的事情。可是你可能突然发现，调用门只能实现特权级由低到高的转移，而我们的程序一直是在最高的特权级下的。也就是说，我们需要先到相对低一点的特权级下，才可能有机会对调用门亲自实践一番。那么，如何才能到低一点的特权级下呢？先不要慌，调用门的故事还没有讲完。

有特权级变换的转移的复杂之处，不但在于严格的特权级检验，还在于特权级变化的时候，堆栈也要发生变化。处理器的这种机制避免了高特权级的过程由于栈空间不足而崩溃。而且，如果不同特权级共享同一个堆栈的话，高特权级的程序可能因此受到有意或无意的干扰。

为了更好地理解堆栈切换时发生的情况，让我们先来一段回忆。

3.2.4.3 回忆——关于堆栈

如果此时你心情浮躁，想要急切地知道如何进行从高特权级到低特权级转移的话，我建议你先放平静，我知道这有点为难，因为有时我也性急，但是没办法，在继续之前，一点回忆在所难免。

如果你的汇编语言是从 8086/8088 开始的，那么你一定对长跳转、短跳转等字眼并不陌生。不过，如果你刚开始就接触 Windows 下的汇编而没有涉及保护模式，那么，可能对长和短没有多少概念。如果你在调试 Windows 应用程序的时候打开汇编窗口，你可能发现jmp就是jmp，call就是call，不同位置的jmp之间以及call之间没什么区别。而到这里，如果回头想一想的话，你一定有些明白了，在我们的程序中，指令call DispReturn和call SelectorCodeDest:0显然不同。与在实模式下类似，如果一个调用或跳转指令是在段间而不是段内进行的，那么我们称之为“长”的（Far jmp/call），反之，如果在段内则是“短”的（Near jmp/call）。

那么长的和短的jmp或call有什么分别呢？对于jmp而言，仅仅是结果不同罢了，短跳转对应段内，而长跳转对应段间；而call则稍微复杂一些，因为call指令是会影响堆栈的，长调用和短调用对堆栈的影响是不同的。我们下面的讨论只考虑 32 位的情况，对于短调用来说，call指令执行时下一条指令的eip压栈，到ret指令执行时，这个eip会被从堆栈中弹出，如图3.15所示。

图3.15显示了call指令执行前后堆栈的变化，可以看出，调用者的eip被压栈，而在此之前参数已经入栈。图中的“调用者eip”对应nop指令地址。而在函数foo调用最后一条指令ret（带有参数）返回之前和之后，堆栈的变化如图3.16所示。

这是短调用的情况，长调用的情况与此类似，容易想到，返回的时候跟调用的时候一样也是“长”转移，所以返回的时候也需要调用者的cs，于是call指令执行时被压栈的就不仅有eip，还应该有cs，如图3.17所示。

相应地，带参数的ret指令执行前后的情形如图3.18所示。

3.2.4.4 通过调用门进行有特权级变换的转移——理论篇

上面是我们对前面内容的一点再讨论，联系当前通过调用门的转移，我们想到，call一个调用门也是长调用，情况应该跟上面所说的长调用差不多才

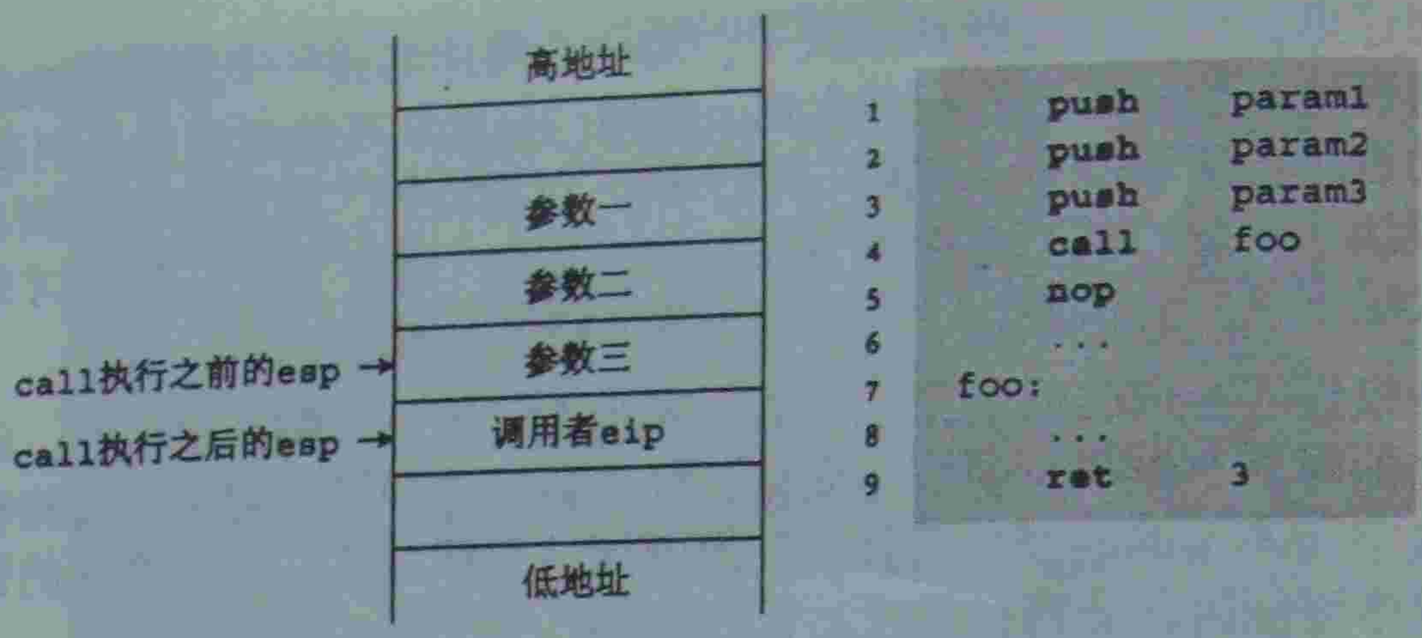


图 3.15 短调用时堆栈示意

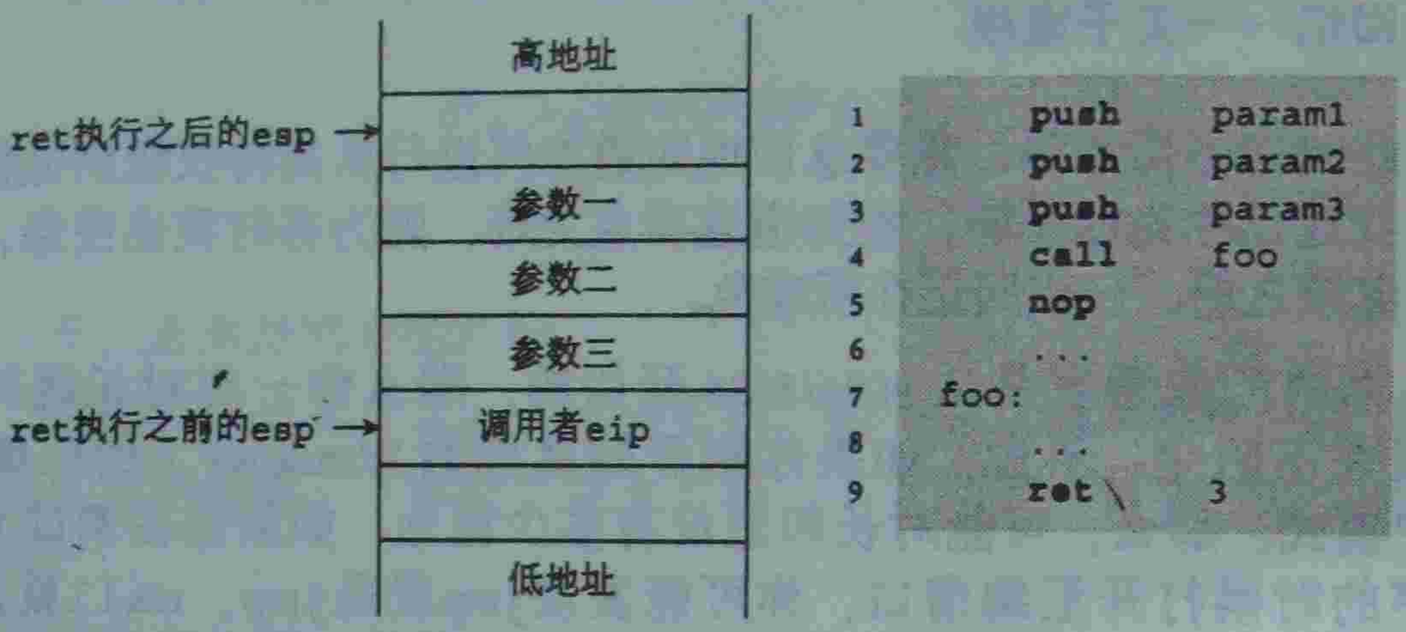


图 3.16 短调用返回时堆栈示意

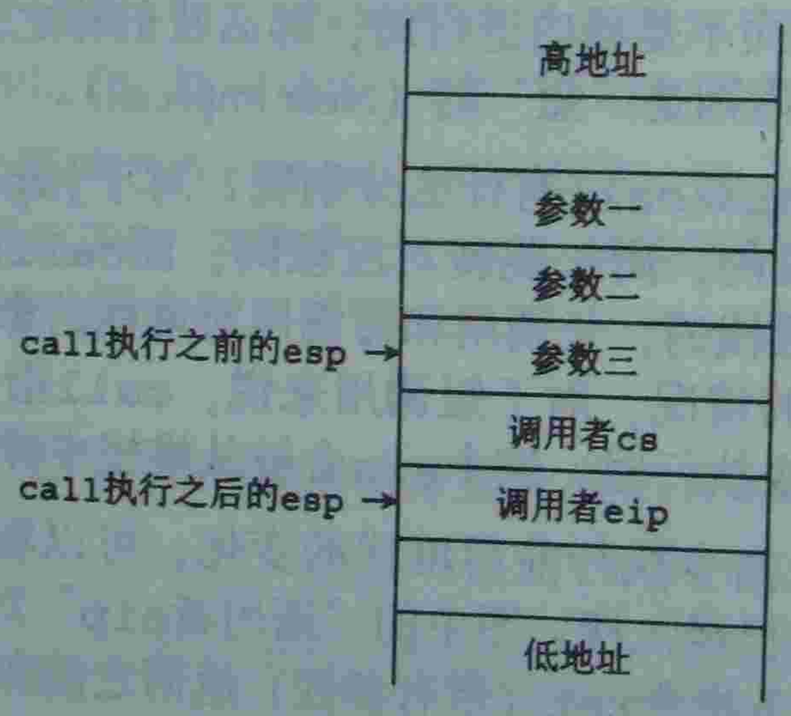


图 3.17 长调用时堆栈示意

对。可是，正如我们已经提到的，由于一些原因堆栈发生了切换，也就是说，call指令执行前后的堆栈已经不再是同一个。这样一来问题出现了，我们在堆栈 A 中压入参数和返回时地址，等到需要使用它们的时候堆栈已经变成 B 了，这该怎么办呢？Intel 提供了这样一种机制，将堆栈 A 的诸多内容复制到堆栈 B 中，如图3.19所示。

这里，我们涉及两个堆栈。事实上，由于每一个任务最多都可能在 4 个特权级间转移，所以，每个任务实际上需要 4 个堆栈。可是，我们只有一个ss和

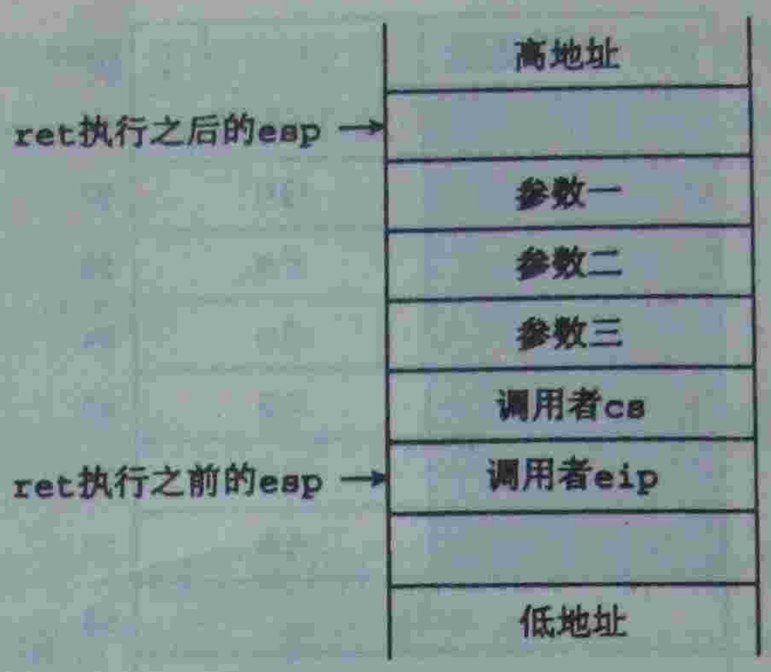


图 3.18 长调用返回时堆栈示意（通过带参数的 ret 指令）

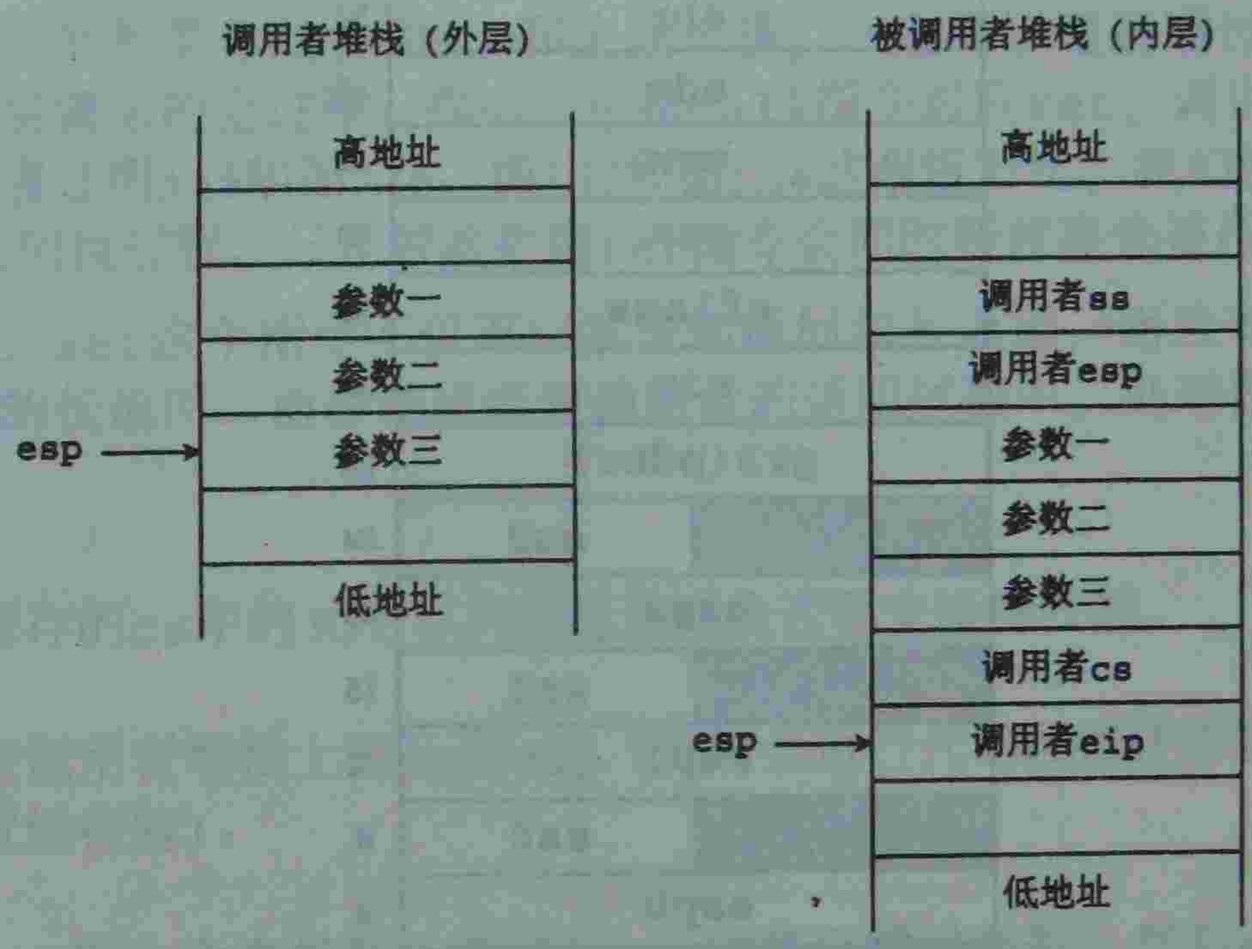


图 3.19 有特权级变换的转移时堆栈变化

一个esp，那么当发生堆栈切换，我们该从哪里获得其余堆栈的ss和esp呢？实际上，这里涉及一样新事物 TSS（Task-State Stack），它是一个数据结构，里面包含多个字段，32 位 TSS 如图3.20所示。

可以看出，TSS 包含很多个字段，但是在这里，我们只关注偏移 4 到偏移 27 的 3 个ss和 3 个esp。当发生堆栈切换时，内层的ss和esp就是从这里取得的。

比如，我们当前所在的是 ring3，当转移至 ring1 时，堆栈将被自动切换到由ss1和esp1指定的位置。由于只是在由外层到内层（低特权级到高特权级）切换时新堆栈才会从 TSS 中取得，所以 TSS 中没有位于最外层的 ring3 的堆栈信息。

好了，新堆栈的问题已经解决，就让我们看一下整个的转移过程是怎样的。下面就是 CPU 在整个过程中所做的工作：

1. 根据目标代码段的 DPL（新的 CPL）从 TSS 中选择应该切换至哪个ss

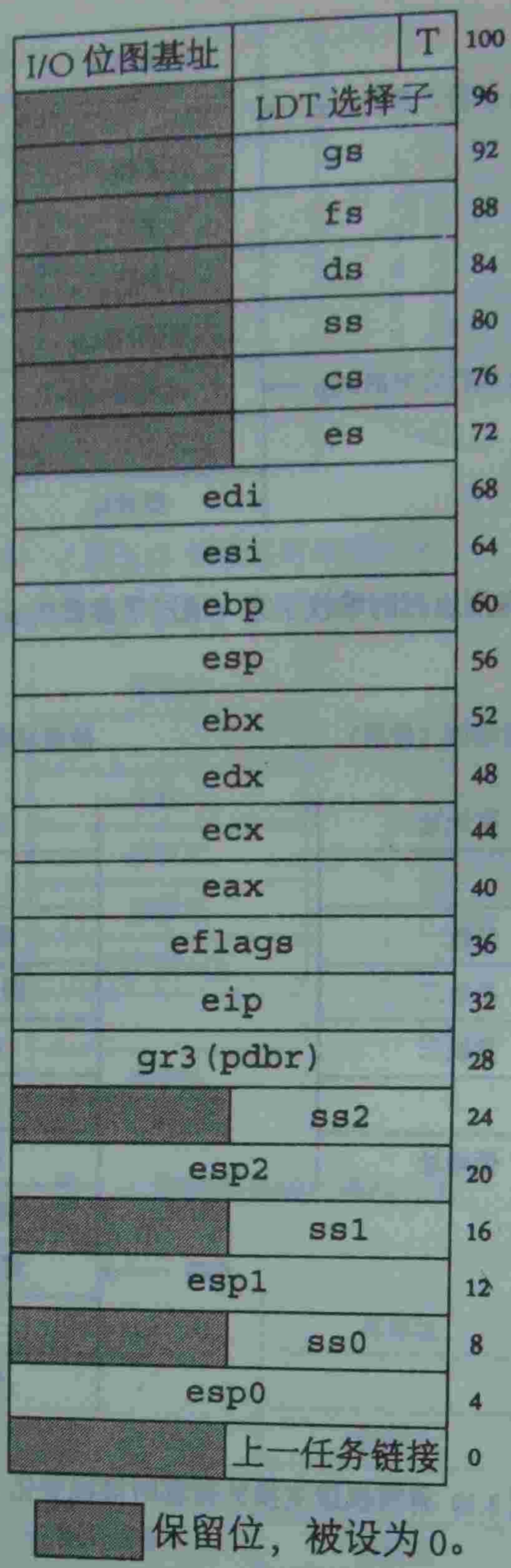


图 3.20 32 位 TSS（Task-State Segment）

和esp。

2. 从 TSS 中读取新的ss和esp。在这过程中如果发现ss、esp或者 TSS 界限错误都会导致无效 TSS 异常（#TS）。

3. 对ss描述符进行检验，如果发生错误，同样产生 #TS 异常。

4. 暂时性地保存当前ss和esp的值。

5. 加载新的ss和esp。

6. 将刚刚保存起来的ss和esp的值压入新栈。

7. 从调用者堆栈中将参数复制到被调用者堆栈（新堆栈）中，复制参数的数目由调用门中 Param Count 一项来决定。如果 Param Count 是零的话，将不会复制参数。

8. 将当前的cs和eip压栈。

9. 加载调用门中指定的新的cs和eip，开始执行被调用者过程。

在第7步中，我们终于明白了调用门中 Param Count 的作用，至此，调用门中各个部分的作用不再留有疑问。要说明的是，Param Count 只有 5 位，也就是说，最多只能复制 31 个参数。如果参数多于 31 个该怎么办呢？这时可以让其中的某个参数变成指向一个数据结构的指针，或者通过保存在新堆栈里的ss和esp来访问旧堆栈中的参数。

好了，此刻如果你结合图3.20和上述步骤，一定可以理解通过调用门进行由外层到内层调用的全过程。那么，正如call指令对应ret，调用门也面临返回的问题。通过图3.15和图3.16、图3.17和图3.18这两组对比，我们发现，ret基本上是call的反过程，只是带参数的ret指令会同时释放事先被压栈的参数。

实际上，ret这个指令不仅可以实现短返回和长返回，而且可以实现带有特权级变换的长返回。由被调用者到调用者的返回过程中，处理器的工作包含以下步骤：

1. 检查保存的cs中的 RPL 以判断返回时是否要变换特权级。

2. 加载被调用者堆栈上的cs和eip（此时会进行代码段描述符和选择子类型和特权级检验）。

3. 如果ret指令含有参数，则增加esp的值以跳过参数，然后esp将指向被保存过的调用者ss和esp。注意，ret的参数必须对应调用门中的 Param Count 的值。

4. 加载ss和esp，切换到调用者堆栈，被调用者的ss和esp被丢弃。在这里将会进行ss描述符、esp以及ss段描述符的检验。

5. 如果ret指令含有参数，增加esp的值以跳过参数（此时已经在调用者堆栈中）。

6. 检查ds、es、fs、gs的值，如果其中哪一个寄存器指向的段的 DPL 小于 CPL（此规则不适用于一致代码段），那么一个空描述符会被加载到该寄存器。

图3.21可以比较形象地表示出这个过程。

综上所述，使用调用门的过程实际上分为两个部分，一部分是从低特权级到高特权级，通过调用门和call指令来实现；另一部分则是从高特权级到低特权级，通过ret指令来实现。说到这里，我想你一定明白了，通过ret指令可以实现由高特权级到低特权级的转移。好的，事不宜迟，我们马上行动。

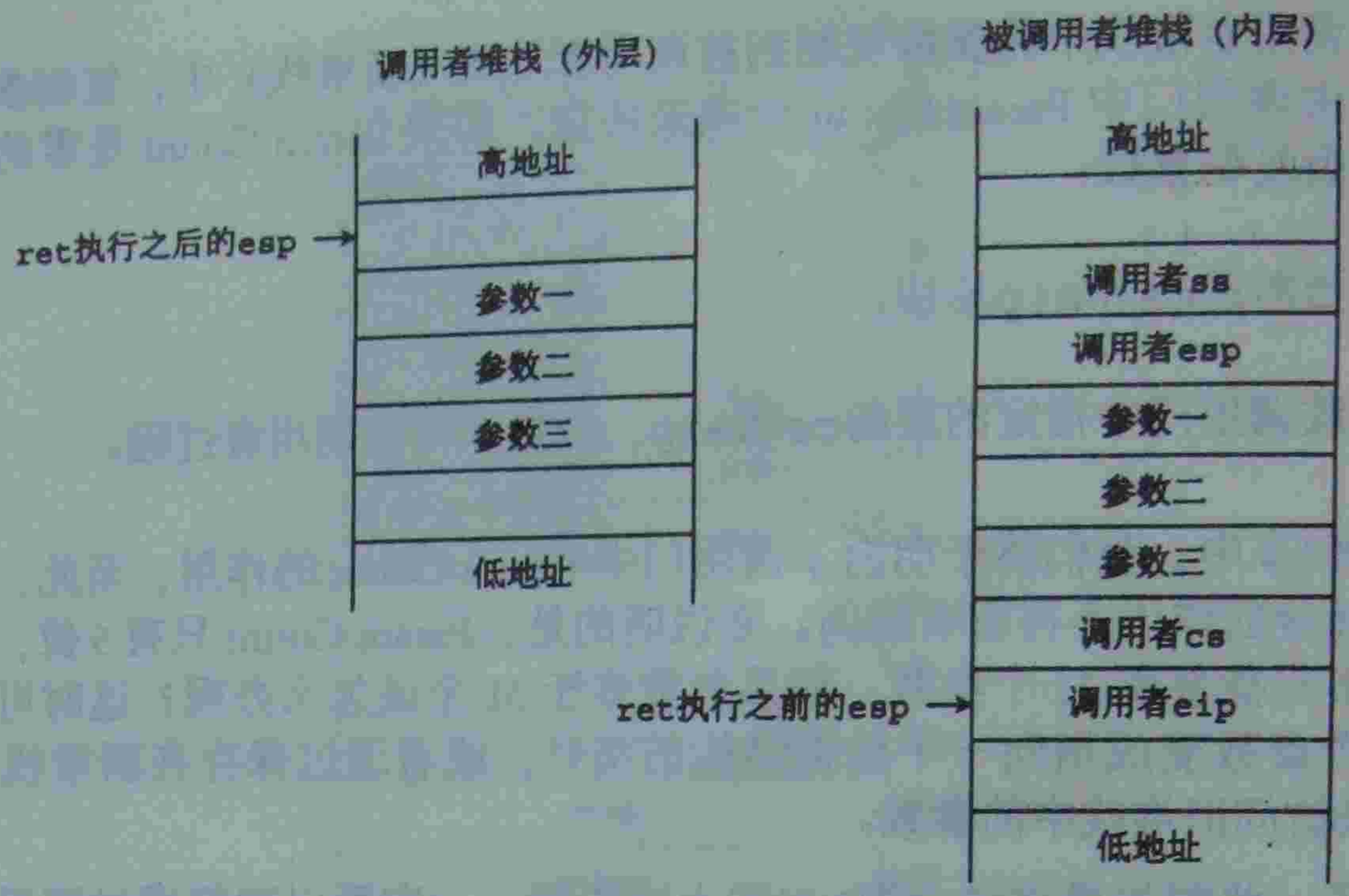


图 3.21 有特权级变换的长调用返回时堆栈示意（通过带参数的 ret 指令）

3.2.4.5 进入 ring3

我们已经知道，在ret指令执行前，堆栈中应该已经准备好了目标代码段的cs、eip，以及ss和esp，另外，还可能有参数。这些可以是处理器压入栈的，当然，也可以由我们自己压栈。在我们的例子中，在ret前的堆栈如图3.22所示。

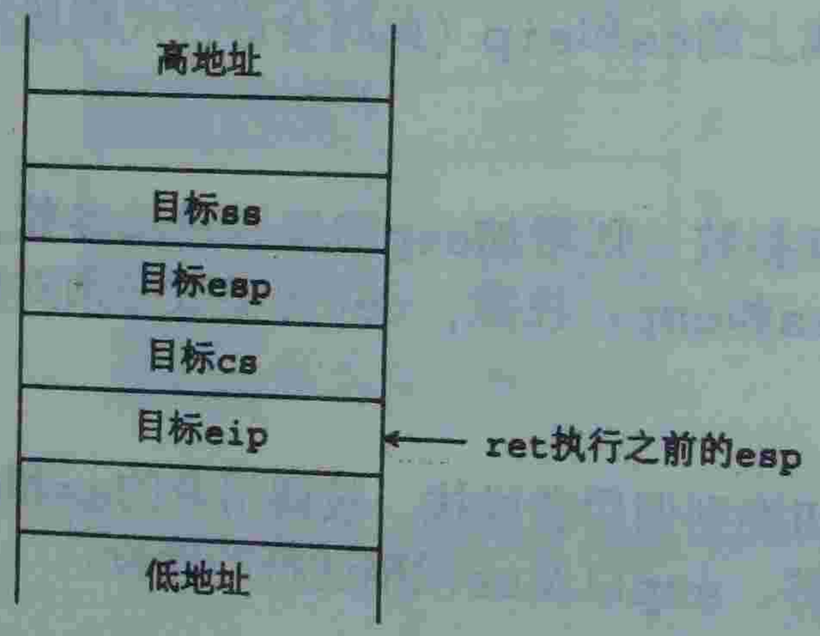


图 3.22 ret 指令执行前的堆栈

这样，执行ret之后就可以转移到低特权级代码中了。我们还是在前文所写的程序（pmtest4.asm）基础上做一下修改（形成pmtest5a.asm）。如图3.22所示，我们至少要添加一个 ring3 的代码段和一个 ring3 的堆栈段。我们马上来添加它们（代码3.16）。

代码 3.16 ring3 的代码段（节自 chapter3/e/pmtest5a.asm）

```
19  LABEL_DESC_CODE_RING3: Descriptor 0,SegCodeRing3Len-1, DA_C+DA_32+DA_DPL3
    ...
22  LABEL_DESC_STACK3:     Descriptor 0,      TopOfStack3, DA_DRWA+DA_32+DA_DPL3
    ...
25  LABEL_DESC_VIDEO:      Descriptor 0B8000h,     0ffffh, DA_DRW+DA_DPL3
```

```
 ...
40. SelectorCodeRing3       equ     LABEL_DESC_CODE_RING3   - LABEL_GDT + SA_RPL3
 ...
43  SelectorStack3          equ     LABEL_DESC_STACK3       - LABEL_GDT + SA_RPL3
 ...
75  ; 堆栈段ring3
76  [SECTION .s3]
77  ALIGN   32
78  [BITS   32]
79  LABEL_STACK3:
80          times 512 db 0
81  TopOfStack3     equ     $ - LABEL_STACK3 - 1
82  ; END of [SECTION .s3]
 ...
379 ; CodeRing3
380 [SECTION .ring3]
381 ALIGN   32
382 [BITS   32]
383 LABEL_CODE_RING3:
384         mov     ax, SelectorVideo
385         mov     gs, ax
386
387         mov     edi, (80 * 14 + 0) * 2
388         mov     ah, 0Ch
389         mov     al, '3'
390         mov     [gs:edi], ax
391
392         jmp     $
393 SegCodeRing3Len equ     $ - LABEL_CODE_RING3
394 ; END of [SECTION .ring3]
```

这个 ring3 代码段非常简单，跟[SECTION .la]和[SECTION .sdest]的内容差不多，同样是打印一个字符。只是需要注意，由于这段代码运行在 ring3，而在其中由于要写显存而访问到了 VIDEO 段，为了不会产生错误，我们把 VIDEO 段的 DPL 修改为 3（第25行）。

可以看到，第392行让程序不再继续执行。之所以这样做，是为了先验证一下由 ring0 到 ring3 的转移是否成功。如果屏幕上出现红色的 3，并且停住不动，不再返回 DOS，则说明转移成功。

新段对应的描述符的属性加上了DA_DPL3，让它的 DPL 变成了 3（第19行），相应选择子的SA_RPL3将 RPL 也设成了 3（第40行）。

初始化描述符的代码与初始化其他描述符的代码类似，在此略去。

这样，代码段和堆栈段都已经准备好了。让我们将ss、esp、cs、eip依次压栈，并且执行retf指令（代码3.17）。

代码 3.17 节自 chapter3/e/pmtest5a.asm

```
266         push    SelectorStack3
267         push    TopOfStack3
268         push    SelectorCodeRing3
269         push    0
270         retf
```

此段代码放在显示完字符串“In Protect Mode now.”后立即执行。

编译，运行，结果如图3.23所示。

我们看到了红色的 3！这表明我们由 ring0 到 ring3 的历史性转移成功完成！这是我们第一次进入不同的特权级别！

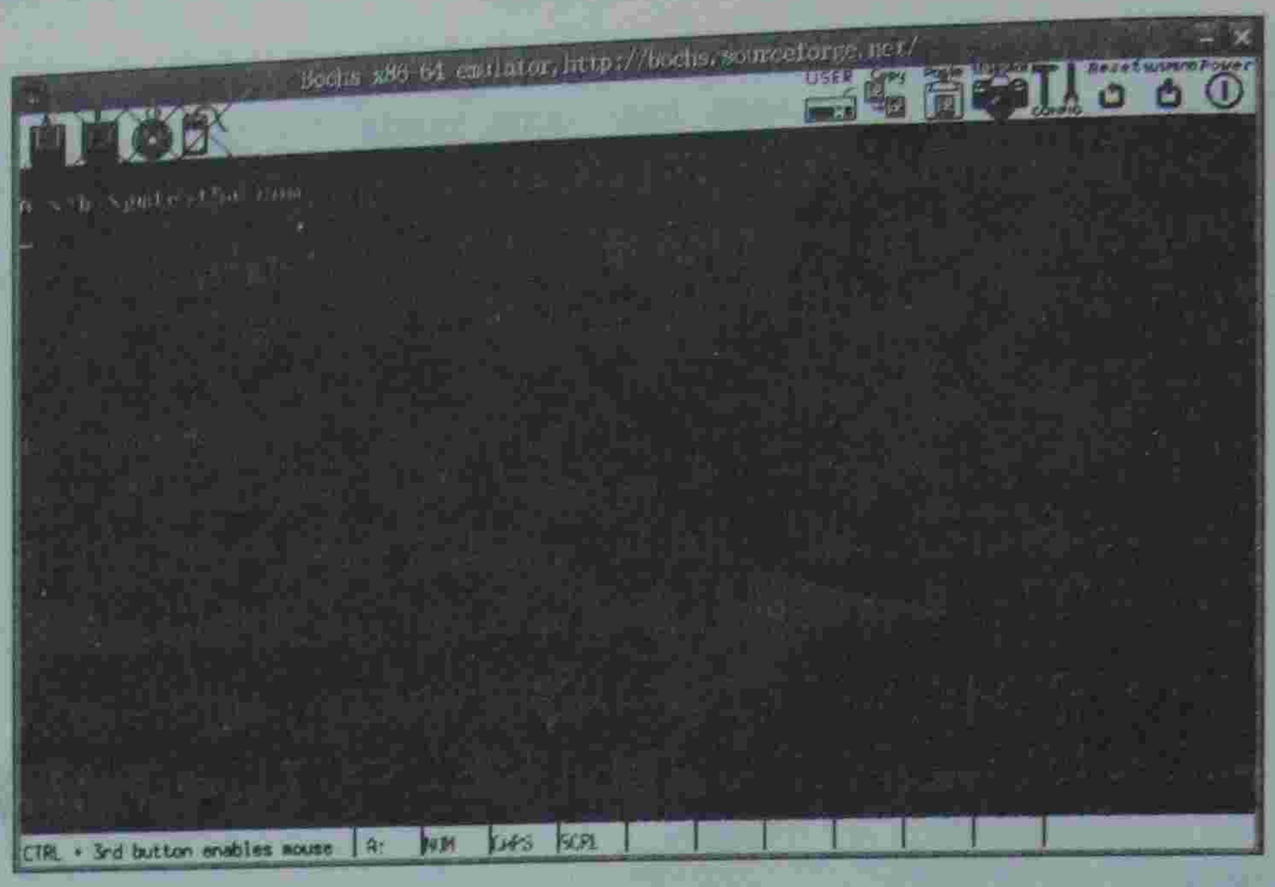


图 3.23 pmtest5a 的执行结果

3.2.4.6 通过调用门进行有特权级变换的转移——实践篇

既然已经位于 ring3 中了，就让我们试验一下调用门的使用。将调用门的描述符和选择子以及 [SECTION .ring3] 的代码稍做修改（代码3.18）。

代码 3.18 在 ring3 中使用调用门（节自 chapter3/e/pmtest5b.asm）

```
28  LABEL_CALL_GATE_TEST: Gate SelectorCodeDest,    0,      0, DA_386CGate+DA_DPL3
    ...
47  SelectorCallGateTest    equ     LABEL_CALL_GATE_TEST    - LABEL_GDT + SA_RPL3
    ...
380 [SECTION .ring3]
381 ALIGN   32
382 [BITS   32]
383 LABEL_CODE_RING3:
384         mov     ax, SelectorVideo
385         mov     gs, ax
386         mov     edi, (80 * 14 + 0) * 2
387         mov     ah, 0Ch
388         mov     al, '3'
389         mov     [gs:edi], ax
390
⇒ 391       call    SelectorCallGateTest:0
392
393         jmp     $
394 SegCodeRing3Len equ     $ - LABEL_CODE_RING3
```

在进入死循环之前，我们增加了使用调用门的指令，这个调用门是我们之前定义的。修改描述符和选择子是为了满足 CPL 和 RPL 都小于等于调用门 DPL 的条件。

现在是不是可以编译并运行了呢？试一下？什么？出现错误？你可能想起来了，从低特权级到高特权级转移的时候，需要用到 TSS，我们就来准备一个 TSS（代码3.19）。

代码 3.19 TSS（节自 chapter3/e/pmtest5c.asm）

```
24  LABEL_DESC_TSS:     Descriptor 0,           TSSLen-1, DA_386TSS
    ...
```

```
45  SelectorTSS         equ     LABEL_DESC_TSS          - LABEL_GDT
    ...
85  ; TSS
86  [SECTION .tss]
87  ALIGN   32
88  [BITS   32]
89  LABEL_TSS:
90                  DD      0                       ; Back
91                  DD      TopOfStack              ; 0 级堆栈
92                  DD      SelectorStack           ;
93                  DD      0                       ; 1 级堆栈
94                  DD      0                       ;
95                  DD      0                       ; 2 级堆栈
96                  DD      0                       ;
97                  DD      0                       ; CR3
98                  DD      0                       ; EIP
99                  DD      0                       ; EFLAGS
100                 DD      0                       ; EAX
101                 DD      0                       ; ECX
102                 DD      0                       ; EDX
103                 DD      0                       ; EBX
104                 DD      0                       ; ESP
105                 DD      0                       ; EBP
106                 DD      0                       ; ESI
107                 DD      0                       ; EDI
108                 DD      0                       ; ES
109                 DD      0                       ; CS
110                 DD      0                       ; SS
111                 DD      0                       ; DS
112                 DD      0                       ; FS
113                 DD      0                       ; GS
114                 DD      0                       ; LDT
115                 DW      0                       ; 调试陷阱标志
116                 DW      $ - LABEL_TSS + 2       ; I/O位图基址
117                 DB      0ffh                    ; I/O位图结束标志
118 TSSLen          equ     $ - LABEL_TSS
```

可以看出，除了 0 级堆栈之外，其他各个字段我们都没做任何初始化。因为在本例中，我们只用到这一部分。

添加初始化 TSS 描述符的代码（从略）之后，TSS 就准备好了，我们需要在特权级变换之前加载它（代码3.20）。

代码 3.20 加载 TSS（节自 chapter3/e/pmtest5c.asm）

```
311         call    DispReturn
312
⇒ 313       mov     ax, SelectorTSS
⇒ 314       ltr     ax
315
316         push    SelectorStack3
317         push    TopOfStack3
318         push    SelectorCodeRing3
319         push    0
320         retf
```

再运行，好，成功了！运行结果如图3.24所示。

我们不但看到了数字 3，而且看到了字母 C，这表明我们在 ring3 下对调用门的使用也是成功的！

到目前为止，我们已经成功实现了两次从高特权级到低特权级以及一次从低特权级到高特权级的转移，最终在低特权级的代码中让程序停住。我们已经具备了在各种特权级下进行转移的能力，并且熟悉了调用门这种典型门描述符

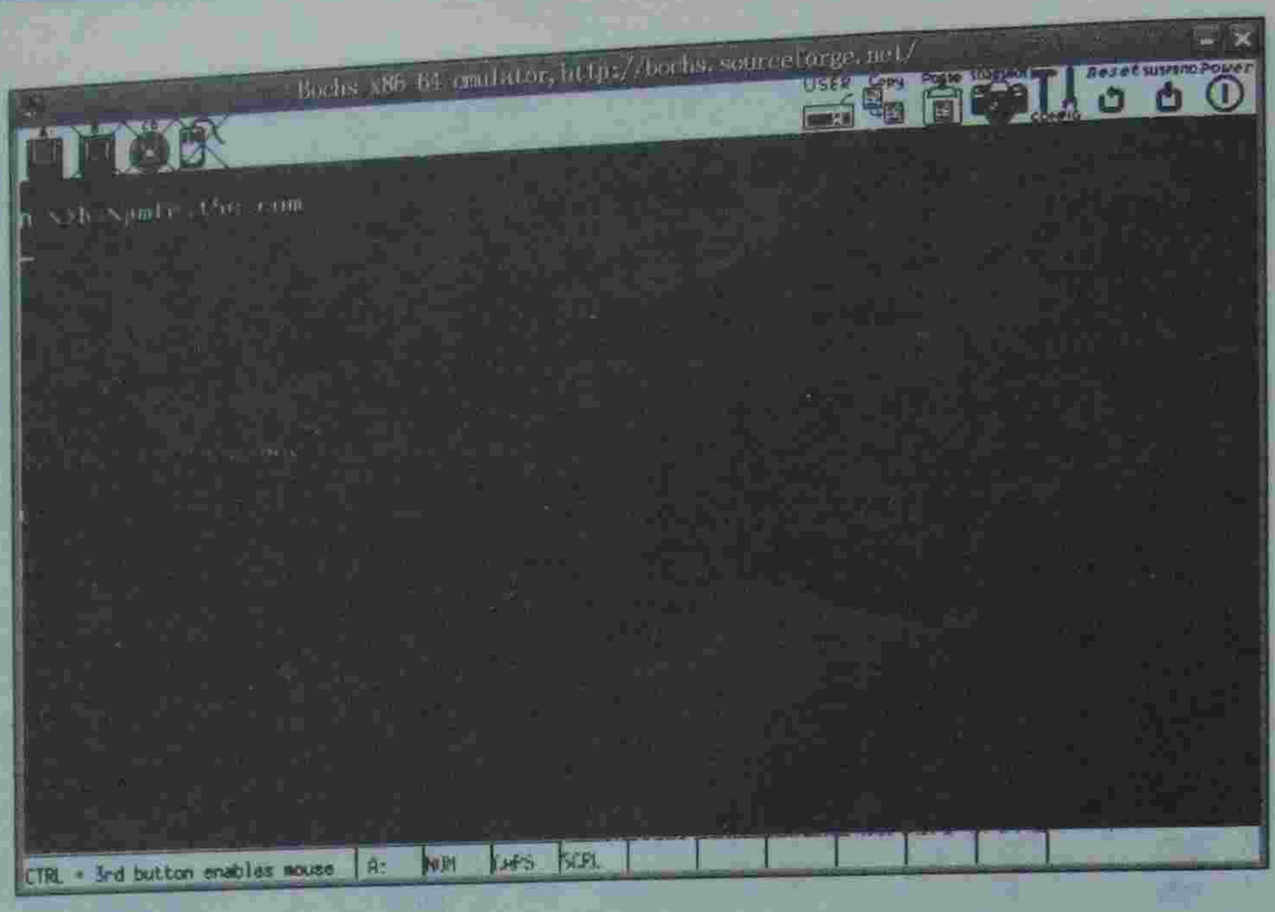


图 3.24 pmtest5c 的执行结果

的用法。值得祝贺的是，如果到这里你都能够弄得清楚的话，今后的内容也会比较容易弄懂，对保护模式这项内容而言，你已经登堂入室了。

好了，为了让我们的程序能够顺利地返回实模式，我们将调用局部任务的代码加入到调用门的目标代码（[SECTION .sdest]）。最后，程序将由这里进入局部任务，然后经由原路返回实模式（代码3.21）。

代码 3.21 经由局部任务返回实模式（节自 chapter3/e/pmtest5.asm）

```
346 [SECTION .sdest]; 调用门目标段
347 [BITS   32]
348
349 LABEL_SEG_CODE_DEST:
350         mov     ax, SelectorVideo
351         mov     gs, ax                          ; 视频段选择子(目的)
352
353         mov     edi, (80 * 12 + 0) * 2  ; 屏幕第 12 行, 第 0 列。
354         mov     ah, 0Ch                         ; 0000: 黑底    1100: 红字
355         mov     al, 'C'
356         mov     [gs:edi], ax
357
358         ; Load LDT
⇒ 359       mov     ax, SelectorLDT
⇒ 360       lldt    ax
361
⇒ 362       jmp     SelectorLDTCodeA:0      ; 跳入局部任务，将打印字母 'L'。
363
364         ;retf
365
366 SegCodeDestLen  equ     $ - LABEL_SEG_CODE_DEST
367 ; END of [SECTION .sdest]
```

编译，运行，结果如图3.25所示。

屏幕输出同时出现 C、L 和 3，这是程序各个部分的输出，正是我们所期望的结果。

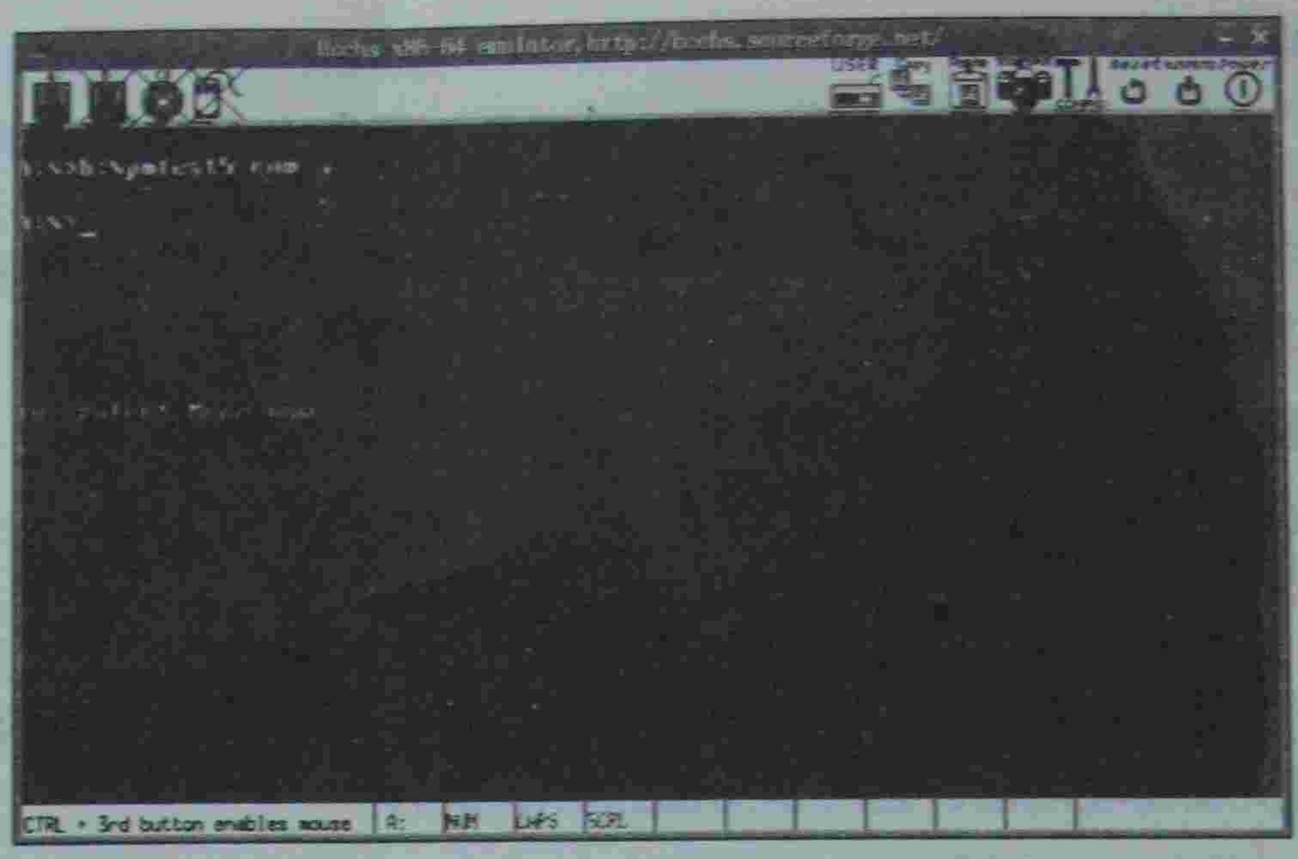


图 3.25 pmtest5 的执行结果

3.2.5 关于“保护”二字的一点思考

笔者一直试图总结保护模式的中心思想，用简单的概括性语句对它进行描述，可最终发现这有点困难，因为“保护”二字其实体现在多个方面。

关于这个问题，我们在接触特权级之前就有所思考。我们提到，描述符中段基址、段界限和段属性都是对段的一种保护。如今，我们已经有了对其有更加深刻理解的资本，因为我们不但已经了解了 IA32 的段式存储机制，而且成功实现了特权级之间的变换。

在涉及特权级的每一步中，处理器都会对 CPL、RPL、DPL 等内容进行比较，这种比较无疑是动态的，是在运行过程中进行的，是发生在多个因素之间的行为。相对而言，段描述符中的界限、属性等内容则是静态的，是对某一项内容的界定和约束。

这样一来，我们对于前面所接触的内容就有了系统的了解：保护模式其实是通过这样动静相宜的方式去见证“保护”二字的含义。

当然，我们并未接触到保护模式所有的内容，但这样的思考不但具有管窥的意义，而且在思考中进一步学习无疑将会更有效率，也将更加深刻。

3.3 页式存储

只要你读过任何一本介绍操作系统的书籍，对于段页式存储这个术语就一定不会陌生，但是它不像是算法或者数据结构，可以很容易通过代码实践，所以很有可能你对它并没有感性认识。

如今不同了，我们已经亲自用代码体验了段式存储，对于分段的概念和细节已经了然于胸。那么现在就让我们一起来体验页式存储。

在开始之前，先对初学者常见的几个问题做一下简单说明。

1. **什么叫做“页”**

所谓“页”，就是一块内存，在 80386 中，页的大小是固定的 4096 字节(4KB)。在 Pentium 中，页的大小还可以是 2MB 或者 4MB，并且可以访问到多于 4GB 的内存，在此我们不予讨论。下文中，我们只讨论页大小为 4KB 的情况。

2. **辑地址、线性地址、物理地址**

前面我们介绍段机制的时候已经提到“线性地址”这个概念（参见图3.6及其说明）。在未打开分页机制时，线性地址等同于物理地址，于是可以认为，逻辑地址通过分段机制直接转换成物理地址。但当分页开启时，情况发生变化，分段机制将逻辑地址转换成线性地址，线性地址再通过分页机制转换成物理地址。这可以用图3.26来说明。

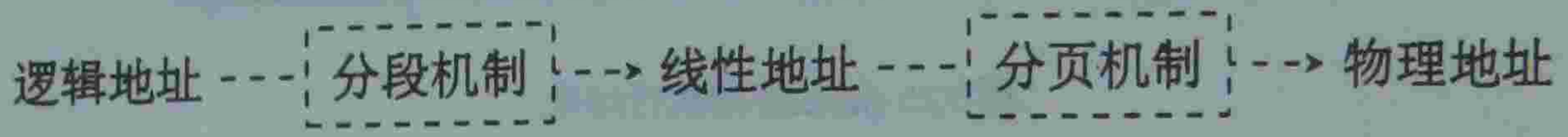


图 3.26 分页开启时的地址转换

这个图的后半部分你可能还不怎么明白，这不要紧，我们随后会越来越清楚。

3. **为什么分页**

我们看到，分段管理机制已经提供了很好的保护机制，那为什么还要加上分页管理机制呢？其实它的主要目的在于实现虚拟存储器。稍后你可以看到，线性地址中任意一个页都能映射到物理地址中的任何一个页，这无疑使得内存管理变得相当灵活。

3.3.1 分页机制概述

从图3.26中我们知道，分页机制就像一个函数：

$$\text{物理地址} = f(\text{线性地址})$$

我们通过图3.27来看一下这个f是怎样的。

如图3.27所示，转换使用两级页表，第一级叫做页目录，大小为 4KB，存储在一个物理页中，每个表项 4 字节长，共有 1024 个表项。每个表项对应第二级的一个页表，第二级的每一个页表也有 1024 个表项，每一个表项对应一个物理页。页目录表的表项简称 PDE（Page Directory Entry），页表的表项简称 PTE（Page Table Entry）。

进行转换时，先是从由寄存器cr3指定的页目录中根据线性地址的高 10 位得到页表地址，然后在页表中根据线性地址的第 12 到 21 位得到物理页首地址，将这个首地址加上线性地址低 12 位便得到了物理地址。

分页机制是否生效的开关位于cr0的最高位 PG 位（参见图3.8）。如果 PG=1，则分页机制生效。所以，当我们准备好了页目录表和页表，并将cr3指向页目录表之后，只需要置 PG 位，分页机制就开始工作了。下面我们就来写一段代码试验一下。

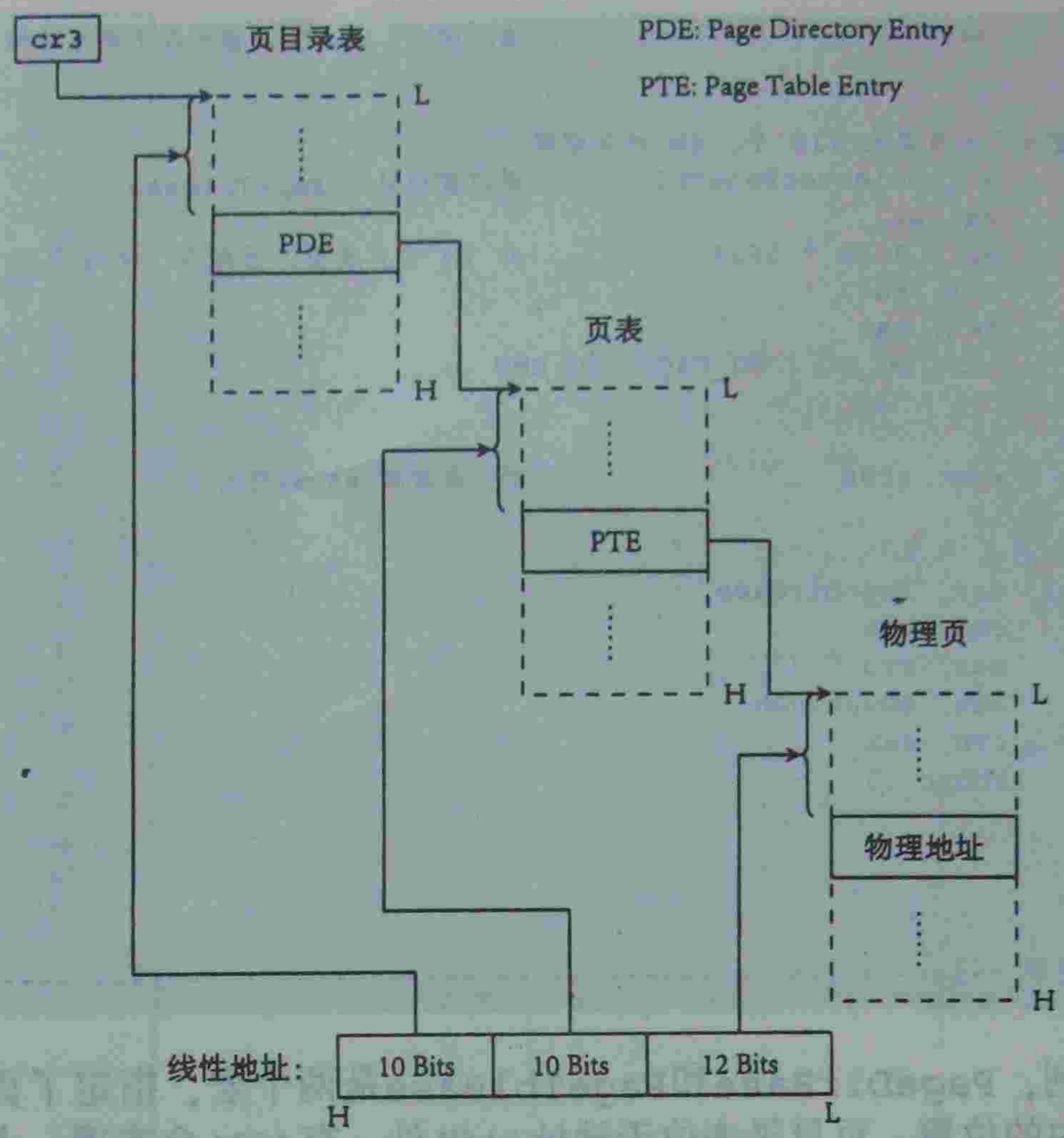


图 3.27 分页机制示意

3.3.2 编写代码启动分页机制

为简单起见，我们在pmtest2.asm的基础进行修改，将试验内存写入和读取的描述符、代码以及数据统统去掉，并添加这样一个函数SetupPaging（代码3.22）。

代码 3.22 初始化分页机制（节自 chapter3/f/pmtest6.asm）

```
 8  PageDirBase             equ     200000h ; 页目录开始地址: 2M
 9  PageTblBase             equ     201000h ; 页表开始地址: 2M+4K
     ...
19  LABEL_DESC_PAGE_DIR: Descriptor PageDirBase, 4095, DA_DRW;Page Directory
20  LABEL_DESC_PAGE_TBL: Descriptor PageTblBase, 1023, DA_DRW|DA_LIMIT_4K;Page Tables
     ...
34  SelectorPageDir         equ     LABEL_DESC_PAGE_DIR     - LABEL_GDT
35  SelectorPageTbl         equ     LABEL_DESC_PAGE_TBL     - LABEL_GDT
     ...
202 ; 启动分页机制 --------------------------------------------------------------
203 SetupPaging:
204         ; 为简化处理，所有线性地址对应相等的物理地址。
205
206         ; 首先初始化页目录
207         mov     ax, SelectorPageDir     ; 此段首地址为 PageDirBase
208         mov     es, ax
209         mov     ecx, 1024               ; 共 1K 个表项
210         xor     edi, edi
211         xor     eax, eax
212         mov     eax, PageTblBase | PG_P  | PG_USU | PG_RWW
213 .1:
214         stosd
```

```
215             add     eax, 4096                       ; 为了简化，所有页表在内存中是连续的.
216             loop    .1
217
218             ; 再初始化所有页表 (1K 个，4M 内存空间)
219             mov     ax, SelectorPageTbl     ; 此段首地址为 PageTblBase
220             mov     es, ax
221             mov     ecx, 1024 * 1024        ; 共 1M 个页表项，也即有 1M 个页
222             xor     edi, edi
223             xor     eax, eax
224             mov     eax, PG_P  | PG_USU | PG_RWW
225     .2:
226             stosd
227             add     eax, 4096                       ; 每一页指向 4K 的空间
228             loop    .2
229
230             mov     eax, PageDirBase
231             mov     cr3, eax
232             mov     eax, cr0
233             or      eax, 80000000h
234             mov     cr0, eax
235             jmp     short .3
236     .3:
237             nop
238
239             ret
240     ; 分页机制启动完毕 ----------------------------------------------------------------
```

可以看到，PageDirBase和PageTblBase是两个宏，指定了页目录表和页表在内存中的位置。页目录表位于地址 2MB 处，有 1024 个表项，占用 4KB 空间，紧接着页目录表便是页表，位于地址 2MB+4KB 处。在这里，我们假定最大的可能，共有 1024 个页表。由于每个页表占用 4096 字节，所以这些页表共占用 4MB 空间。也就是说，本程序所需的内存需至少大于 6MB。

为了逻辑清晰和代码编写简便，我们分别定义两个段，用来存放页目录表和页表，大小分别是 4KB 和 4MB。

为了简单起见，我们的程序将所有的线性地址映射到相同的物理地址，于是线性地址和物理地址的关系符合下面的公式：

$$物理地址 = f(线性地址) = 线性地址$$

所以程序的大部分代码都是写入页目录表和页表，以便让上面的公式成立。为了完全理解写入的内容，我们需要先来看一下 PDE 和 PTE 的结构。

3.3.3 PDE 和 PTE

图3.28和图3.29是 PDE（4KB 页表）和 PTE（4KB 页）的结构和各位详细解释。为避免因翻译不当而造成歧义，图中各位的名称使用的是英文原文。

PDE 和 PTE 中各位的解释如下：

- P 存在位，表示当前条目所指向的页或页表是否在物理内存中。P=0 表示页不在内存中，如果处理器试图访问此页，将会产生页异常（page-fault exception，#PF）；P=1 表示页在内存中。
- R/W 指定一个页或者一组页（比如，此条目是指向页表的页目录条目）的

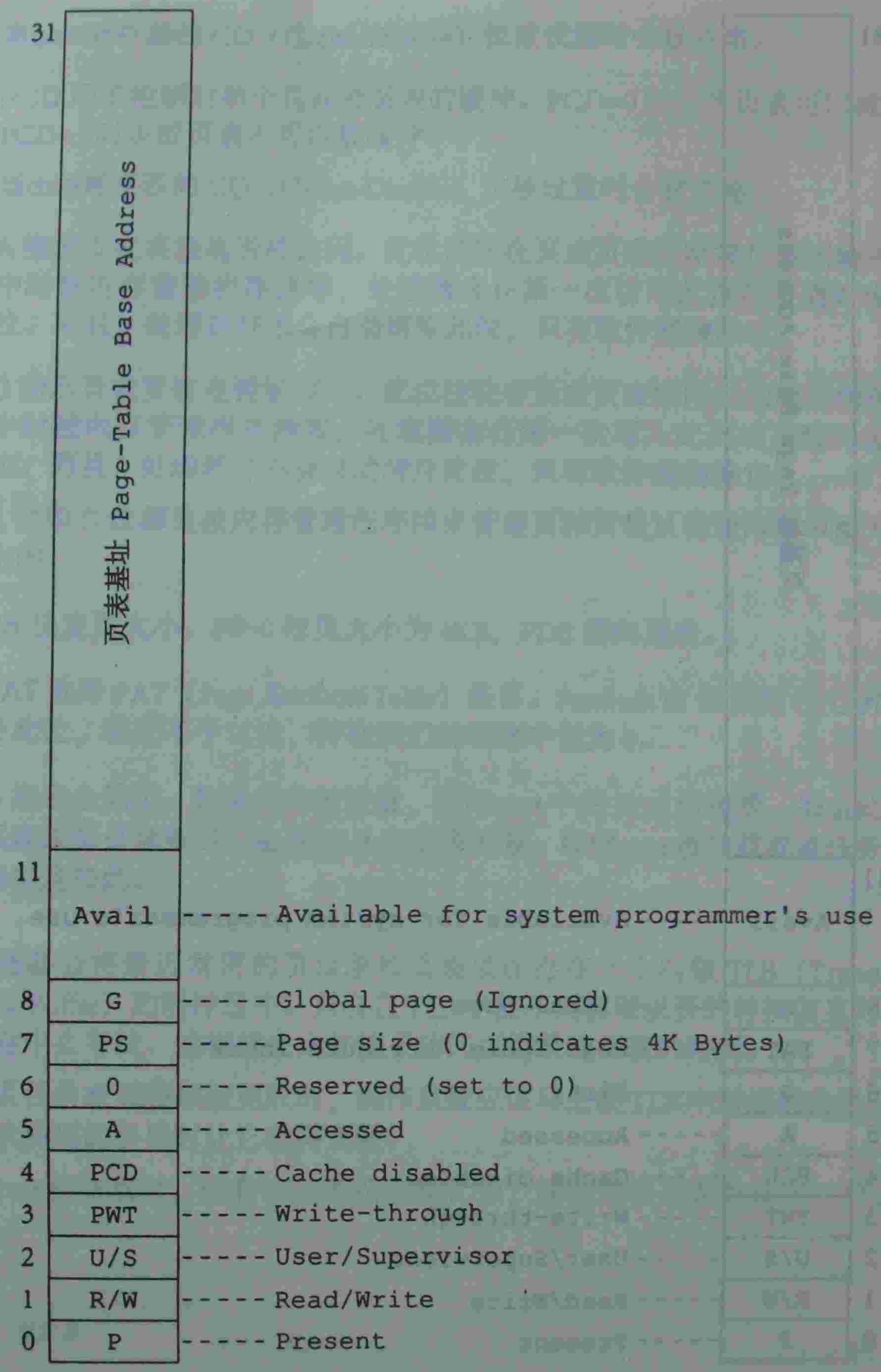


图 3.28 PDE（Page-Directory Entry）

读写权限。此位与 U/S 位和寄存器cr0中的 WP 位相互作用。R/W=0 表示只读；R/W=1 表示可读并可写。

- U/S 指定一个页或者一组页（比如，此条目是指向页表的页目录条目）的特权级。此位与 R/W 位和寄存器cr0中的 WP 位相互作用。U/S=0 表示系统级别（Supervisor Privilege Level），如果 CPL 为 0、1 或 2，那么它便是在此级别；U/S=1 表示用户级别（User Privilege Level），如果 CPL 为 3，那么

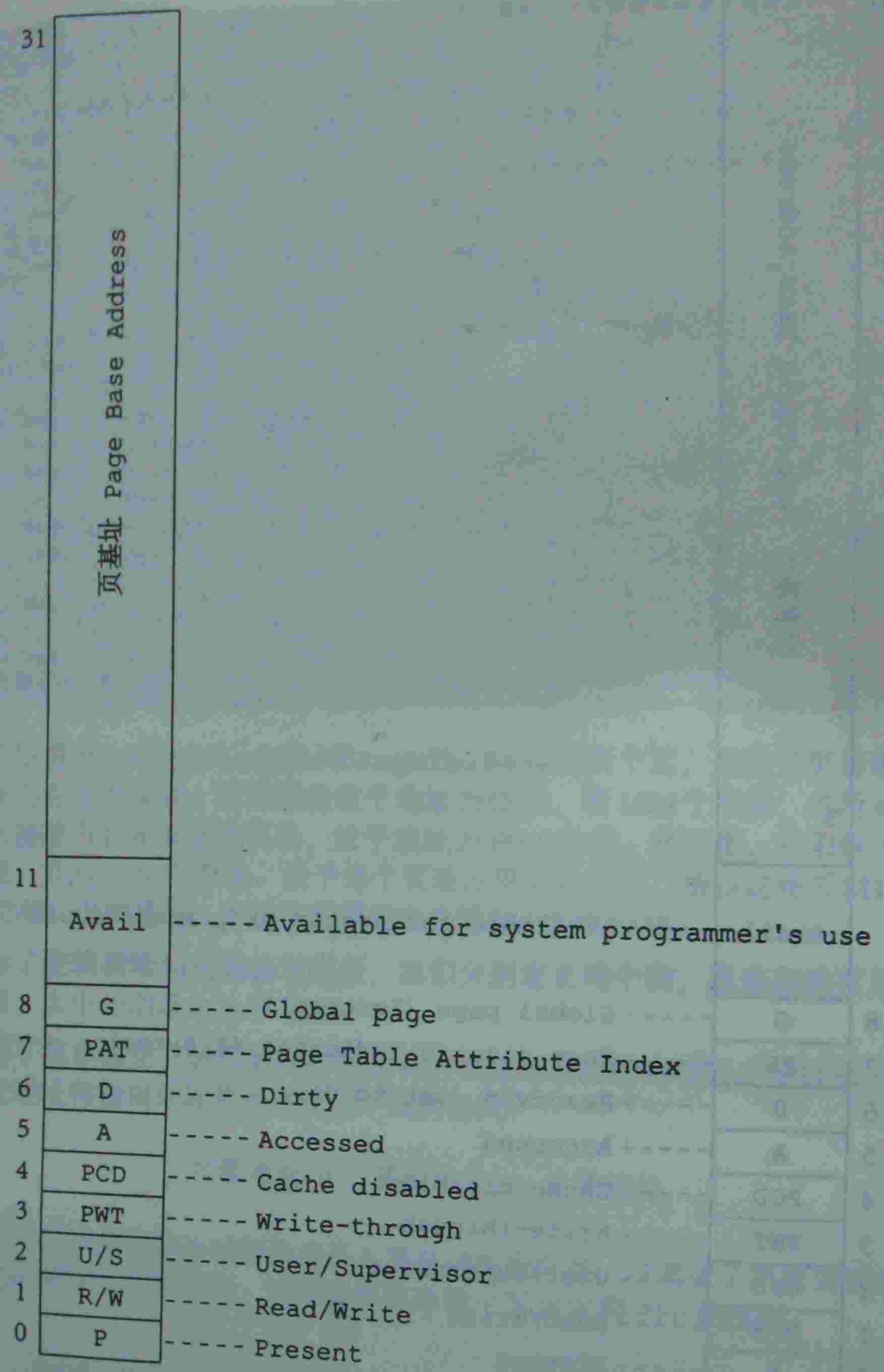


图 3.29 PTE (Page-Table Entry)

它便是在此级别。

如果cr0中的 WP 位为 0，那么即便用户级（User P.L.）页面的 R/W=0，系统级（Supervisor P.L.）程序仍然具备写权限；如果 WP 位为 1，那么系统级（Supervisor P.L.）程序也不能写入用户级（User P.L.）只读页。

- PWT 用于控制对单个页或者页表的缓冲策略。PWT=0 时使用 Write-back 缓冲策略；PWT=1 时使用 Write-through 缓冲策略。

当cr0寄存器的 CD（Cache-Disable）位被设置时会被忽略。

- **PCD** 用于控制对单个页或者页表的缓冲。PCD=0 时页或页表可以被缓冲；PCD=1 时页或页表不可以被缓冲。

  当cr0寄存器的 CD（Cache-Disable）位被设置时会被忽略。

- **A** 指示页或页表是否被访问。此位往往在页或页表刚刚被加载到物理内存中时被内存管理程序清零，处理器会在第一次访问此页或页面时设置此位。而且，处理器并不会自动清除此位，只有软件能清除它。

- **D** 指示页或页表是否被写入。此位往往在页或页表刚刚被加载到物理内存中时被内存管理程序清零，处理器会在第一次写入此页或页面时设置此位。而且，处理器并不会自动清除此位，只有软件能清除它。

  A 位和 D 位都是被内存管理程序用来管理页和页表从物理内存中换入和换出的。

- **PS** 决定页大小。PS=0 时页大小为 4KB，PDE 指向页表。

- **PAT** 选择 PAT（Page Attribute Table）条目。Pentium III 以后的 CPU 开始支持此位，在此不予讨论，并在我们的程序中设为 0。

- **G** 指示全局页。如果此位被设置，同时cr4中的 PGE 位被置，那么此页的页表或页目录条目不会在 TLB 中变得无效，即便cr3被加载或者任务切换时也是如此。

处理器会将最近常用的页目录和页表项保存在一个叫做 TLB（Translation Lookaside Buffer）的缓冲区中。只有在 TLB 中找不到被请求页的转换信息时，才会到内存中去寻找。这样就大大加快了访问页目录和页表的时间。

当页目录或页表项被更改时，操作系统应该马上使 TLB 中对应的条目无效，以便下次用到此条目时让它获得更新。

当cr3被加载时，所有 TLB 都会自动无效，除非页或页表条目的 G 位被设置。

3.3.4 cr3

说起cr3，我们虽然提到它指向页目录表，但并未谈起过它的结构，cr3的结构如图3.30所示。

图 3.30 cr3

cr3又叫做 PDBR（Page Directory Base Register）。它的高 20 位将是页目录表首地址的高 20 位，页目录表首地址的低 12 位会是零，也就是说，页目录表会

是 4KB 对齐的。类似地，PDE 中的页表基址（Page-Table Base Address）以及 PTE 中的页基址（Page Base Address）也是用高 20 位来表示 4KB 对齐的页表和页。

至于第 3 位和第 4 位的两个标志，我们暂时可以忽略它们。

3.3.5 回头看代码

现在再来看代码3.22，我想你差不多应该明白了（请同时参考图3.31）。

开头的第207行和第208行将段寄存器es对应页目录表段，下面让edi等于0，于是es:edi就指向了页目录表的开始。第214行的指令stosd第一次执行时就把eax中的PageTblBase|PG_P|PG_USU|PG_RWW存入了页目录表的第一个 PDE。

那么来看看这个 PDE 是什么值。PageTblBase|PG_P|PG_USU|PG_RWW（第212行）让当前（第一个）PDE 对应的页表首地址变成PageTblBase，而且属性显示其指向的是存在的可读可写的用户级别页表。

实际上，当为页目录表中的第一个 PDE 赋值时，一个循环就已经开始了。循环的每一次执行中，es:edi会自动指向下一个 PDE，而第215行也将下一个页表的首地址增加 4096 字节，以便与上一个页表首尾相接。这样，经过 1024 次循环（第209行由ecx指定）之后，页目录表中的所有 PDE 都被赋值完毕，它们的属性相同，都为指向可读可写的用户级别页表，并且所有的页表连续排列在以PageTblBase为首地址的 4MB（4096×1024）的空间中。

接下来的工作是初始化所有页表中的 PTE（第218行到第228行）。由于总共有 1024^2个 PTE，于是将ecx赋值为 1024×1024，以便让循环进行 1024^2次。开始对es和edi的处理让es:edi指向了页表段的首地址，即地址PageTblBase处，也是第一个页表的首地址。

第一个页表中的第一个 PTE 被赋值为PG_P|PG_USU|PG_RWW，不难理解，它表示此 PTE 指示的页首地址为 0，并且是个可读可写的用户级别页。这同时意味着第 0 个页表中第 0 个 PTE 指示的页的首地址是 0，于是线性地址 0~0FFFh 将被映射到物理地址 0~0FFFh，即$f(x)=x$，其中 0≤ x ≤0FFFh。接下来进行的循环初始化了剩下的所有页表中的 PTE，将 4GB 空间的线性地址映射到相同的物理地址。如图3.31所示即为pmtest6.asm中的分页图示。

这样，页目录表和所有的页表被初始化完毕。接下来到了正式启动分页机制的时候了。首先让cr3指向页目录表（第230行和第231行），然后设置cr0的 PG（第232行到第234行），这样，分页机制就启动完成了。

执行结果如图3.32所示。

除了不见了pmtest2.com中因试验内存写入和读取而打印的字符，并没有其他改变，甚至于我们从这里看不出一点分页机制的影子。这也难怪，我们把所有线性地址映射到完全相同的物理地址。不过，细心的你可能发现了两个问题：一是页表显然浪费得太多了，我们可能根本没有那么大的内存；二是我们除了“实现了”分页，并没有“得益于”分页，也就是说，我们还没有体会到分页的妙处。那么下面就继续修改我们的程序。

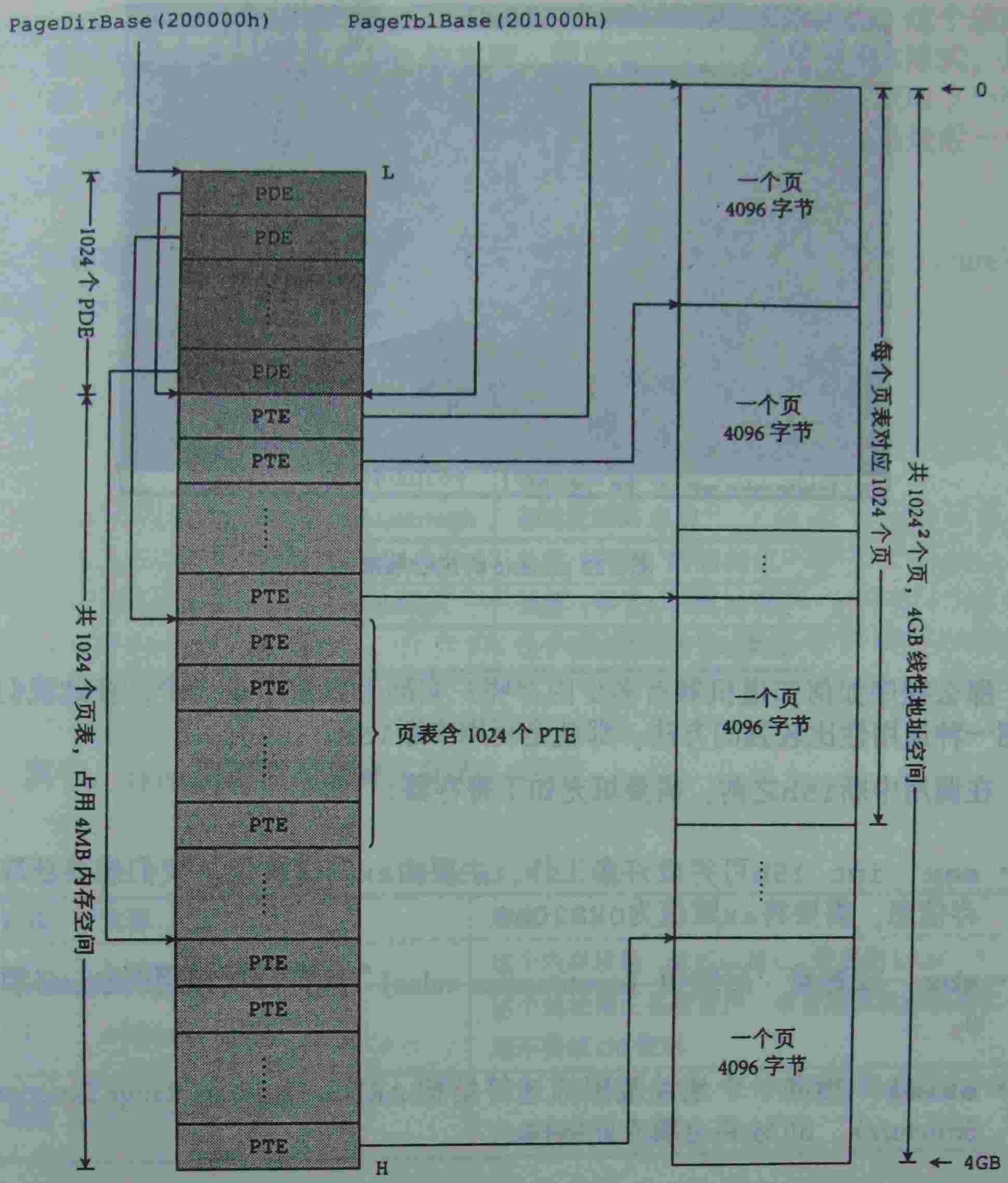


图 3.31 pmtest6.asm 中的分页图示

3.3.6 克勤克俭用内存

在前面的程序中，我们用了 4MB 的空间来存放页表，并用它映射了 4GB 的内存空间，而我们的物理内存不见得有这么大，这显然是太浪费了。如果我们的内存总数只有 16MB 的话，只是页表就占用了 25% 的内存空间。而实际上，如果仅仅是对等映射的话，16MB 的内存只要 4 个页表就够了。所以，我们有必要知道内存有多大，然后根据内存大小确定多少页表是够用的。而且，一个操作系统也必须知道内存的容量，以便进行内存管理。

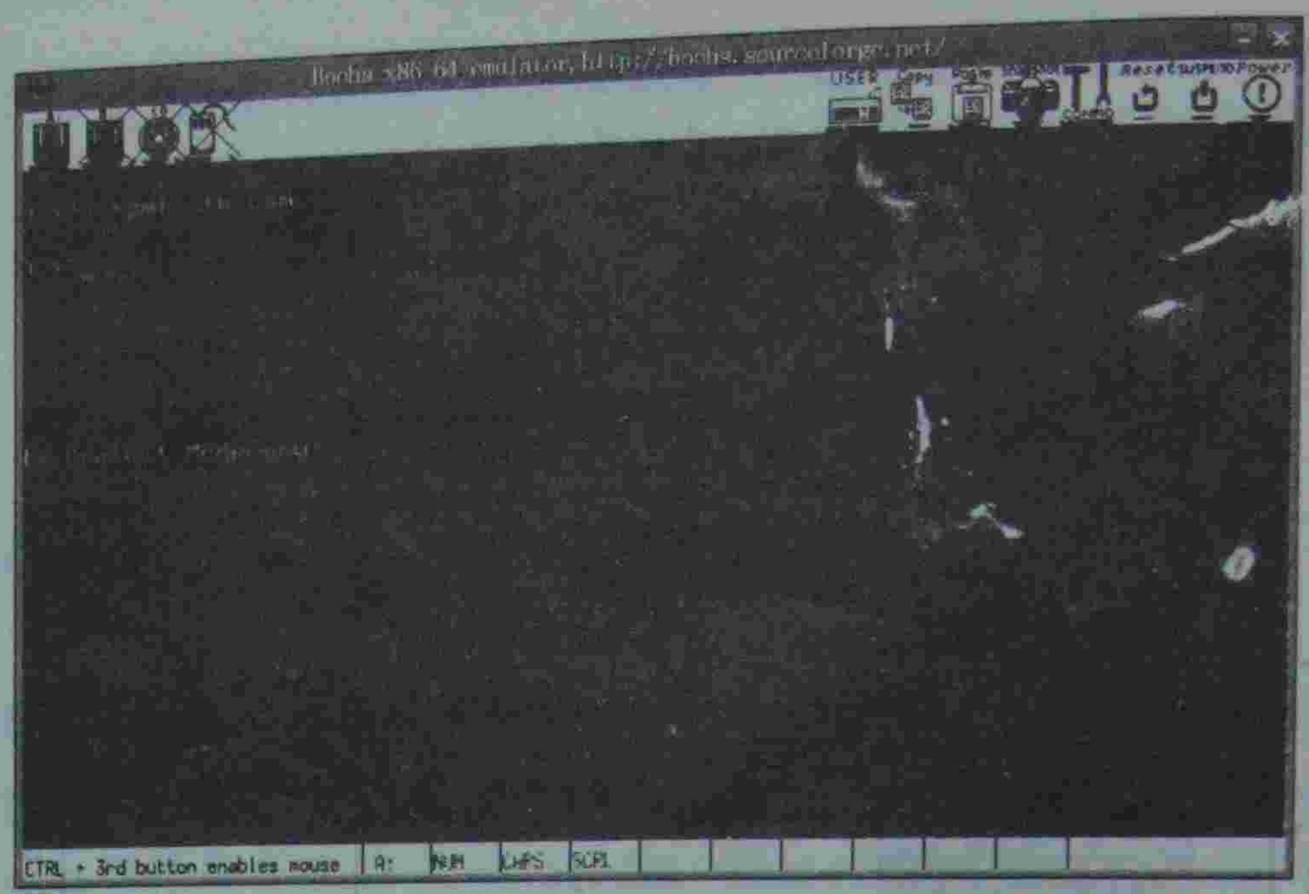


图 3.32 pmtest6 的执行结果

那么程序如何知道机器有多少内存呢？实际上方法不止一个，在此我们仅介绍一种通用性比较强的方法，那就是利用中断15h。

在调用中断15h之前，需要填充如下寄存器：

- **eax** int 15h可完成许多工作，主要由ax的值决定，我们想要获取内存信息，需要将ax赋值为0E820h。
- **ebx** 放置着“后续值（continuation value）”，第一次调用时ebx必须为0。
- **es:di** 指向一个地址范围描述符结构 ARDS（Address Range Descriptor Structure），BIOS 将会填充此结构。
- **ecx** es:di所指向的地址范围描述符结构的大小，以字节为单位。无论es:di所指向的结构如何设置，BIOS 最多将会填充ecx个字节。不过，通常情况下无论ecx为多大，BIOS 只填充 20 字节，有些 BIOS 忽略ecx的值，总是填充 20 字节。
- **edx** 0534D4150h（'SMAP'）——BIOS 将会使用此标志，对调用者将要请求的系统映像信息进行校验，这些信息会被 BIOS 放置到es:di所指向的结构中。

中断调用之后，结果存放于下列寄存器之中。

- **CF** CF=0表示没有错误，否则存在错误。
- **eax** 0534D4150h（'SMAP'）。
- **es:di** 返回的地址范围描述符结构指针，和输入值相同。
- **ecx** BIOS 填充在地址范围描述符中的字节数量，被 BIOS 所返回的最小值是 20 字节。

- **ebx** 这里放置着为等到下一个地址描述符所需要的后续值，这个值的实际形势依赖于具体的 BIOS 的实现，调用者不必关心它的具体形式，只需在下次迭代时将其原封不动地放置到ebx中，就可以通过它获取下一个地址范围描述符。如果它的值为 0，并且CF没有进位，表示它是最后一个地址范围描述符。

上面提到的地址范围描述符结构（Address Range Descriptor Structure）如表3.5所示。

表 3.5 ARDS

偏移	名称	意义
0	BaseAddrLow	基地址的低 32 位
4	BaseAddrHigh	基地址的高 32 位
8	LengthLow	长度（字节）的低 32 位
12	LengthHigh	长度（字节）的高 32 位
16	Type	这个地址范围的地址类型

其中，Type 的取值及其意义如表3.6所示。

表 3.6 ARDS 之 Type

取值	名称	意义
1	AddressRangeMemory	这个内存段是一段可以被 OS 使用的 RAM
2	AddressRangeReserved	这个地址段正在被使用，或者被系统保留所以一定不要被 OS 使用
其他	未定义	保留，为未来使用，任何其他值都必须被 OS 认为是 AddressRangeReserved

由上面的说明我们看出，ax=0E820h时调用int 15h得到的不仅仅是内存的大小，还包括对不同内存段的一些描述。而且，这些描述都被保存在一个缓冲区中。所以，在我们调用int 15h之前，必须先有缓冲区。我们可以在每得到一次内存描述时都使用同一个缓冲区，然后对缓冲区里的数据进行处理，也可以将每次得到的数据放进不同的位置，比如一块连续的内存，然后在想要处理它们时再读取。后一种方式可能更方便一些，所以在这里定义了一块 256 字节的缓冲区（代码3.23第65行），它最多可以存放 12 个 20 字节大小的结构体。我们现在还不知道它到底够不够用，这个大小仅仅是凭猜测设定。我们将把每次得到的内存信息连续写入这块缓冲区，形成一个结构体数组。然后在保护模式下把它们读出来，显示在屏幕上，并且凭借它们得到内存的容量。

让我们看代码3.23。

代码 3.23 得到内存信息（节自 chapter3/g/pmtest7.asm）

```
65  _MemChkBuf:     times   256     db      0
    ...
111             ; 得到内存数
```

```
112             mov     ebx, 0
113             mov     di, _MemChkBuf
114     .loop:
115             mov     eax, 0E820h
116             mov     ecx, 20
117             mov     edx, 0534D4150h
118             int     15h
119             jc      LABEL_MEM_CHK_FAIL
120             add     di, 20
121             inc     dword [_dwMCRNumber]
122             cmp     ebx, 0
123             jne     .loop
124             jmp     LABEL_MEM_CHK_OK
125     LABEL_MEM_CHK_FAIL:
126             mov     dword [_dwMCRNumber], 0
127     LABEL_MEM_CHK_OK:
```

可以看到，代码使用了一个循环，一旦CF被置位或者ebx为零，循环将结束。在第一次循环开始之前，eax为0000E820h，ebx为0，ecx为20，edx为0534D4150h，es:di指向_MemChkBuf的开始处。在每一次循环进行时，寄存器di的值将会递增，每次的增量为20字节。另外，eax、ecx和edx的值都不会变，ebx的值我们置之不理。同时，每次循环我们让_dwMCRNumber的值加1，这样到循环结束时它的值会是循环的次数，同时也是地址范围描述符结构的个数。

好了，下面我们来到保护模式下的32位代码，添加显示内存信息的过程（代码3.24）。

代码 3.24 显示内存信息（节自 chapter3/g/pmtest7.asm）

```
305     DispMemSize:
306             push    esi
307             push    edi
308             push    ecx
309
310             mov     esi, MemChkBuf
311             mov     ecx, [dwMCRNumber];for(int i=0;i<[MCRNumber];i++)//每次得到一个ARDS
312     .loop:                            ;{
313             mov     edx, 5            ;  for(int j=0;j<5;j++) //每次得到一个ARDS中的成员
314             mov     edi, ARDStruct    ;  {//依次显示BaseAddrLow,BaseAddrHigh,LengthLow,
315     .1:                               ;                       LengthHigh,Type
316             push    dword [esi]       ;
317             call    DispInt           ;    DispInt(MemChkBuf[j*4]); //显示一个成员
318             pop     eax               ;
319             stosd                     ;    ARDStruct[j*4] = MemChkBuf[j*4];
320             add     esi, 4            ;
321             dec     edx               ;
322             cmp     edx, 0            ;
323             jnz     .1                ;  }
324             call    DispReturn        ;  printf("\n");
325             cmp     dword [dwType], 1 ;  if(Type == AddressRangeMemory)
326             jne     .2                ;  {
327             mov     eax, [dwBaseAddrLow];
328             add     eax, [dwLengthLow];
329             cmp     eax, [dwMemSize]  ;    if(BaseAddrLow + LengthLow > MemSize)
330             jb      .2                ;
331             mov     [dwMemSize], eax  ;    MemSize = BaseAddrLow + LengthLow;
332     .2:                               ;  }
333             loop    .loop             ;}
334
335             call    DispReturn        ;
336             push    szRAMSize         ;printf("\n");
337             call    DispStr           ;
                                          ;printf("RAM size:");
```

```
338             add     esp, 4                  ;
339                                             ;
340             push    dword [dwMemSize]       ;
341             call    DispInt                 ;DispInt(MemSize);
342             add     esp, 4                  ;
343
344             pop     ecx
345             pop     edi
346             pop     esi
347             ret
```

这段代码的主体框架的注释被写成了C代码，不过这些C代码只是用来帮助读者理解汇编代码而已。可以看出，C代码的可读性要强得多，不过，我们目前暂且忍受一下晦涩的汇编吧，况且，直接操作寄存器和内存还是蛮有成就感的。

由于注释中的C代码仅仅是帮助理解汇编，所以语法和变量名未必全对。不过看起来真的是方便多了，我们一下子就知道，程序的主题是一个循环，循环的次数为地址范围描述符结构（下文用ARDStruct代替）的个数，每次循环将会读取一个ARDStruct。首先打印其中每一个成员的各项，然后根据当前结构的类型，得到可以被操作系统使用的内存的上限。结果会被存放在变量dwMemSize中，并在此模块的最后打印到屏幕。

如果对照注释中的C代码的话，这段程序还是比较好理解的，只是其中新添加了DispInt和DispStr等函数。它们用来方便地显示整形数字和字符串。而且，为了读起来方便，它们连同函数DispAL、DispReturn被放在了lib.inc中，并且通过如下语句包含进pmtest7.asm中：

```
%include "lib.inc"
```

实际上，这跟直接把代码写进这个位置的效果是一样的，但是，把它们单独放进一个文件阅读起来要方便得多。

文件 lib.inc 如代码3.25所示。

代码 3.25 chapter3/g/lib.inc

```
1   ;; lib.inc
    ...
42  ;; 显示一个整形数
43  DispInt:
44          mov     eax, [esp + 4]
45          shr     eax, 24
46          call    DispAL
47
48          mov     eax, [esp + 4]
49          shr     eax, 16
50          call    DispAL
51
52          mov     eax, [esp + 4]
53          shr     eax, 8
54          call    DispAL
55
56          mov     eax, [esp + 4]
57          call    DispAL
58
59          mov     ah, 07h                 ; 0000b: 黑底    0111b: 灰字
60          mov     al, 'h'
61          push    edi
62          mov     edi, [dwDispPos]
```

```
        mov     [gs:edi], ax
        add     edi, 4
        mov     [dwDispPos], edi
        pop     edi

        ret
;; DispInt 结束

;; 显示一个字符串
DispStr:
        push    ebp
        mov     ebp, esp
        push    ebx
        push    esi
        push    edi

        mov     esi, [ebp + 8]   ; pszInfo
        mov     edi, [dwDispPos]
        mov     ah, 0Fh
.1:
        lodsb
        test    al, al
        jz      .2
        cmp     al, 0Ah ; 是回车吗?
        jnz     .3
        push    eax
        mov     eax, edi
        mov     bl, 160
        div     bl
        and     eax, 0FFh
        inc     eax
        mov     bl, 160
        mul     bl
        mov     edi, eax
        pop     eax
        jmp     .1
.3:
        mov     [gs:edi], ax
        add     edi, 2
        jmp     .1

.2:
        mov     [dwDispPos], edi

        pop     edi
        pop     esi
        pop     ebx
        pop     ebp
        ret
;; DispStr 结束

;; 换行
DispReturn:
        push    szReturn
        call    DispStr                 ;printf("\n");
        add     esp, 4

        ret
;; DispReturn 结束
```

在DispInt中，[esp+4]即为已经入栈的参数，函数通过4次对DispAL的调用显示了一个整数，并且最后显示一个灰色的字母“h”。函数DispStr通过一个循环来显示字符串，每一次复制一个字符入显存，遇到\0则结束循环。同时，DispStr加入了对回车的处理，遇到0Ah就会从下一行的开始处继续显示。由于这一点，DispReturn也做了简化，通过DispStr来处理回车。

在以前的程序中，我们用edi保存当前的显示位置，从这个程序开始，我们改为用变量dwDispPos来保存。这样我们就可以放心地使用edi这个寄存器。

至此，我们新增的内容已经准备得差不多了，另外还需要提到的一点是，在数据段中，几乎每个变量都有类似的两个符号，比如：

```
_dwMemSize:     dd      0
```

和

```
dwMemSize       equ     _dwMemSize - $$
```

在实模式下应使用_dwMemSize，而在保护模式下应使用dwMemSize。因为程序是在实模式下编译的，地址只适用于实模式，在保护模式下，数据的地址应该是其相对于段基址的偏移。这个原因我们在前文中曾经提到过，读者可以参考第42页对于OffsetStrTest的解释。

代码3.26中调用了函数 DispMemSize。

代码 3.26 调用 DispMemSize（节自 chapter3/g/pmtest7.asm）

```
238         push    szMemChkTitle
239         call    DispStr
240         add     esp, 4
241
242         call    DispMemSize                     ; 显示内存信息
```

在调用它之前，我们还显示了一个字符串作为将要打印的内存信息的表格头。

好了，程序已经可以运行了，运行结果如图3.33所示。

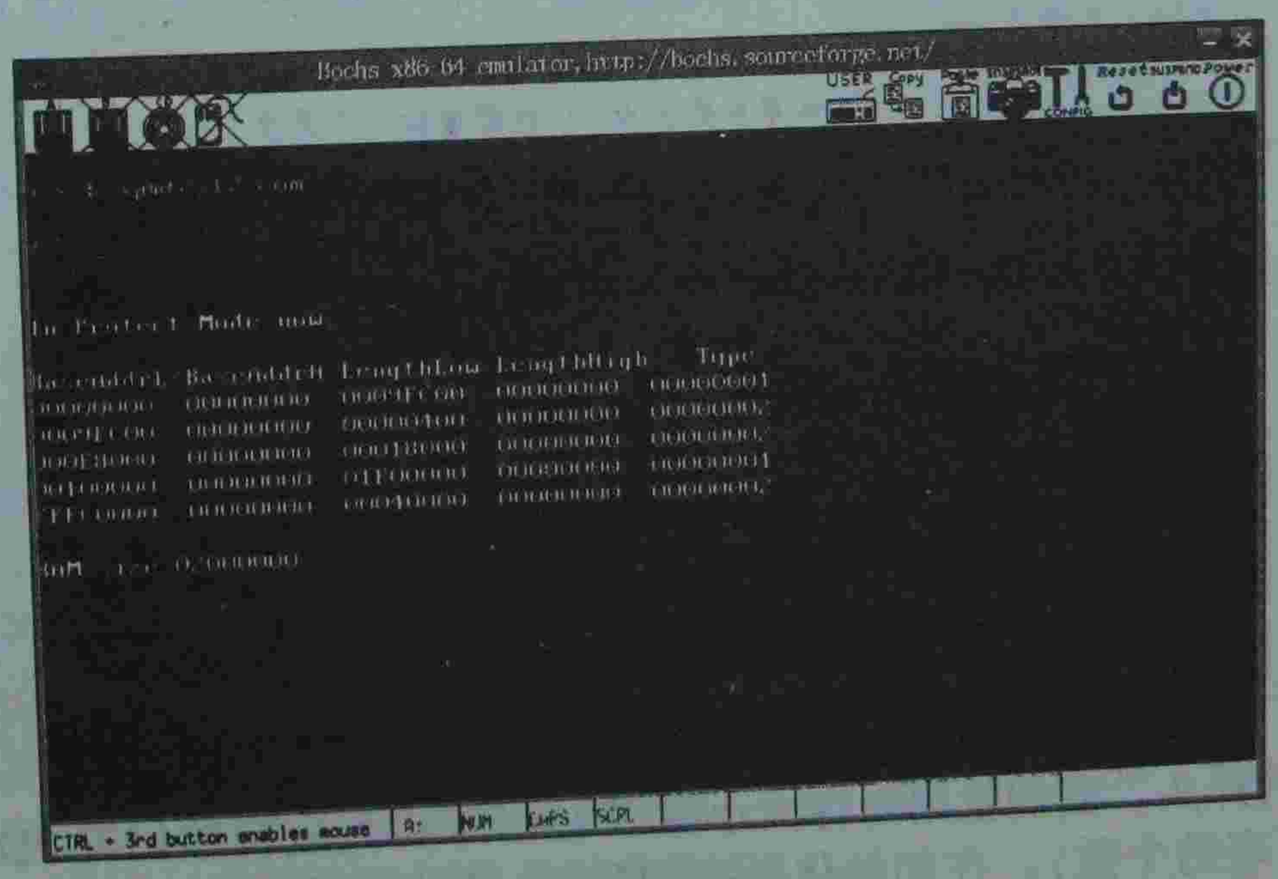


图 3.33 pmtest7 的执行结果

从图3.33中我们看出，总共有 5 段内存被列了出来，对列出的内存情况的解释如表3.7所示。

表 3.7 内存情况解释

内存段	属性	是否可被 OS 使用
00000000h～0009FBFFh	AddressRangeMemory	可
0009FC00h～0009FFFFh	AddressRangeReserved	不可
000E8000h～000FFFFFh	AddressRangeReserved	不可
00100000h～01FFFFFFh	AddressRangeMemory	可
FFFC0000h～FFFFFFFFh	AddressRangeReserved	不可

从这里我们可以看到，操作系统所能使用的最大内存地址为01FFFFFFh，所以此机器拥有 32MB 的内存。而且，幸运地，我们指定的 256 字节的内存MemChkBuf是够用的。

你可能没有想到，得到内存容量还要这么多代码，不过，实际上我们除了得到了内存的大小，还得到了可用内存的分布信息。由于历史原因，系统可用内存分布得并不连续，所以在使用的时候，我们要根据得到的信息小心行事。

内存容量得到了，你是否还记得我们为什么要得到内存？我们是为了节约使用，不再初始化所有 PDE 和所有页表。现在，我们已经可以根据内存大小计算应初始化多少 PDE 以及多少页表，下面来修改一下函数SetupPaging（代码3.27）。

代码 3.27 修改 SetupPaging（节自 chapter3/g/pmtest7.asm）

```
250 SetupPaging:
251         ; 根据内存大小计算应初始化多少PDE以及多少页表
252         xor     edx, edx
253         mov     eax, [dwMemSize]
254         mov     ebx, 400000h    ; 400000h = 4M = 4096 * 1024, 一个页表对应的内存大小
255         div     ebx
256         mov     ecx, eax        ; 此时 ecx 为页表的个数，也即 PDE 应该的个数
257         test    edx, edx
258         jz      .no_remainder
259         inc     ecx             ; 如果余数不为 0 就需增加一个页表
260 .no_remainder:
261         push    ecx             ; 暂存页表个数
    ...
276         ; 再初始化所有页表
277         mov     ax, SelectorPageTbl     ; 此段首地址为 PageTblBase
278         mov     es, ax
279         pop     eax                     ; 页表个数
280         mov     ebx, 1024               ; 每个页表 1024 个 PTE
281         mul     ebx
282         mov     ecx, eax                ; PTE个数 = 页表个数 * 1024
283         xor     edi, edi
    ...
300         ret
```

在函数的开头，我们就用内存大小除以 4MB 来得到应初始化的 PDE 的个数（同时也是页表的个数）。在初始化页表的时候，通过刚刚计算出的页表个数乘以 1024（每个页表含 1024 个 PTE）得出要填充的 PTE 个数，然后通过循环完成对它的初始化。

这样一来，页表所占的空间就小得多，在本例中，32MB 的内存实际上只要 32KB 的页表就够了，所以在 GDT 中，这样初始化页表段：

```
LABEL_DESC_PAGE_TBL: Descriptor PageTblBase,4096*8-1,DA_DRW
```

这样，程序所需的内存空间就小了许多。

3.3.7 进一步体会分页机制

上文中我们提到，我们还没有得益于分页。分页的益处其实体现在多个方面，这里，我们先举一个例子，以便先有初步的认识。

在此之前不知道你有没有注意过一个细节，如果你写一个程序（在 Linux 或 Windows 下均可），并改个名复制一份，然后同时调试，你会发现，从变量地址到寄存器的值，几乎全部都是一样的！而这些“一样的”地址之间完全不会混淆起来，而是各自完成着自己的职责。这就是分页机制的功劳，下面我们就来模拟一下这个效果。

先执行某个线性地址处的模块，然后通过改变cr3来转换地址映射关系，再执行同一个线性地址处的模块，由于地址映射已经改变，所以两次得到的应该是不同的输出（本例对应的代码是pmtest8.asm）。

映射关系转换前的情形如图3.34所示。

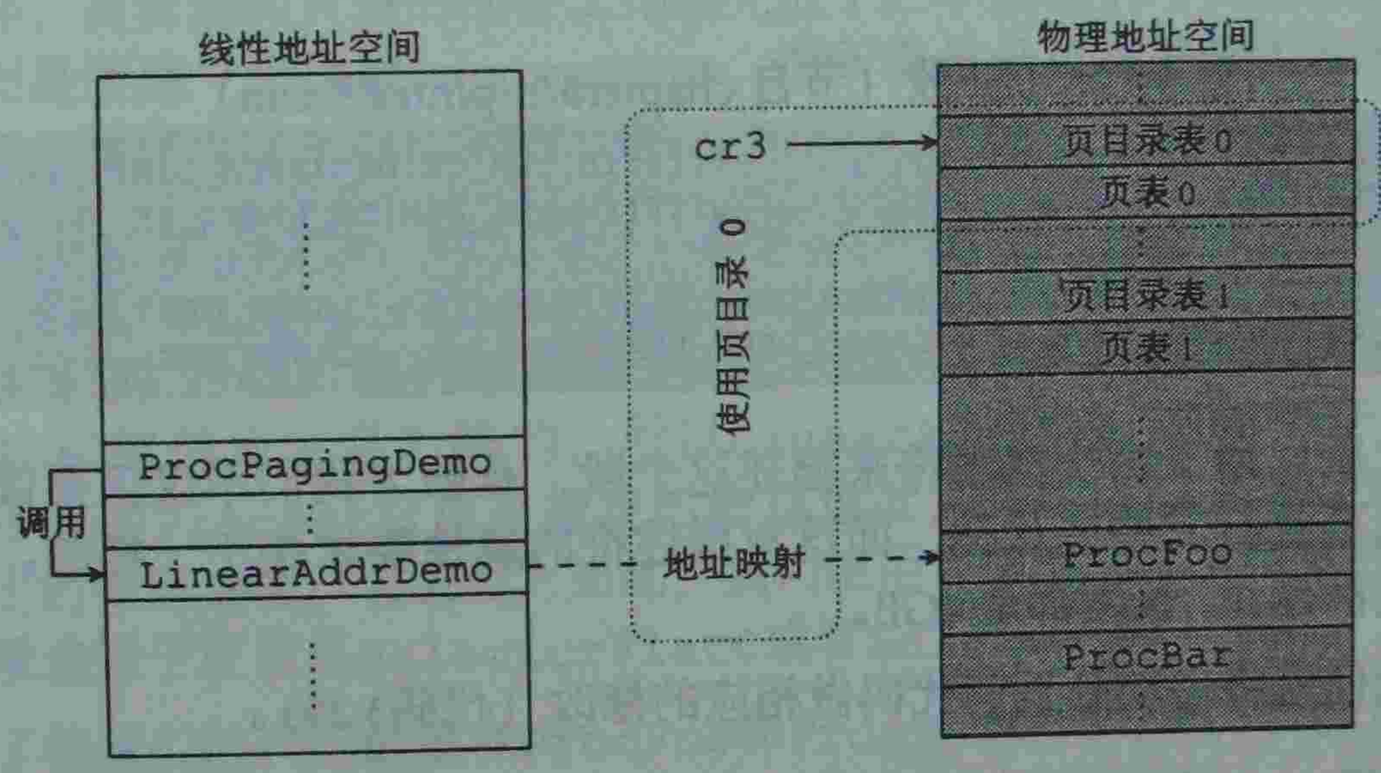


图 3.34 开始时的内存映射关系

开始，我们让ProcPagingDemo中的代码实现向LinearAddrDemo这个线性地址的转移，而LinearAddrDemo映射到物理地址空间中的ProcFoo处。我们让ProcFoo打印出红色的字符串Foo，所以执行时我们应该可以看到红色的Foo。随后我们改变地址映射关系，变化成如图3.35所示的情形。

页目录表和页表的切换让LinearAddrDemo映射到ProcBar（物理地址空间）处，所以当我们再一次调用过程ProcPagingDemo时，程序将转移到ProcBar处执行，我们将看到红色的字符串Bar。

下面我们就来看一下实现这些需要对pmtest7.asm做哪些修改。

首先，我们用到了另外一套页目录表和页表，所以原先的页目录段和页表段已经不再够用了。事实上，前面的程序中我们用两个段分别存放页目录表和页表，是为了让读者阅读时更加直观和形象。在pmtest8.asm中，我们把它们放

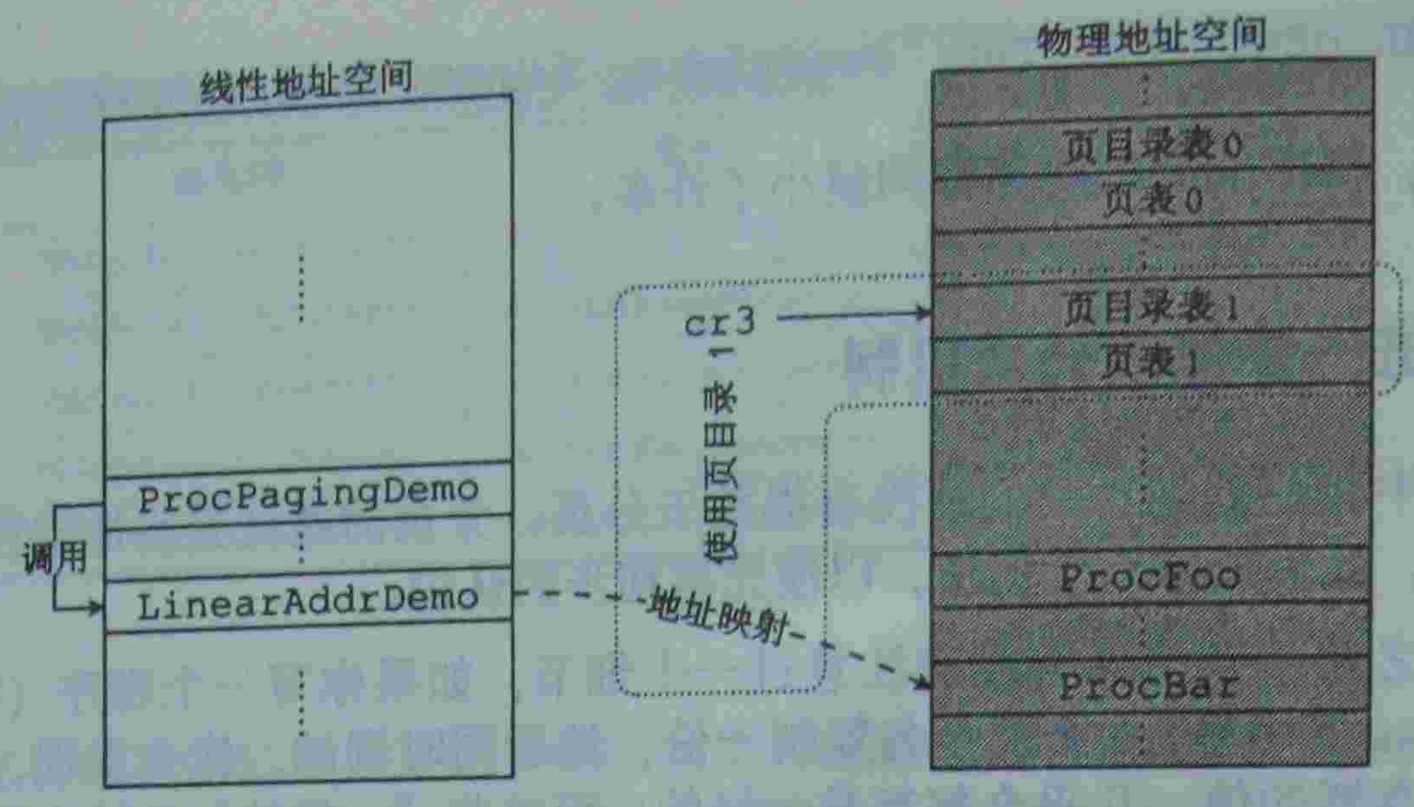


图 3.35 后来的内存映射关系

到同一个段中，同时把增加的一套页目录和页表也放到这个段中。

为了操作方便，我们新增加一个段，其线性地址空间为 0～4GB。由于分页机制启动之前线性地址等同于物理地址，所以通过这个段可以方便地存取特定的物理地址。此段的定义如代码3.28所示。

代码 3.28 Flat 段（节自 chapter3/h/pmtest8.asm）

```
26  LABEL_DESC_FLAT_C:  Descriptor 0,        0fffffh, DA_CR|DA_32|DA_LIMIT_4K; 0-4G
27  LABEL_DESC_FLAT_RW: Descriptor 0,        0fffffh, DA_DRW|DA_LIMIT_4K     ; 0-4G
...
41  SelectorFlatC       equ     LABEL_DESC_FLAT_C       - LABEL_GDT
42  SelectorFlatRW      equ     LABEL_DESC_FLAT_RW      - LABEL_GDT
```

我们之所以用了两个描述符来描述这个段，是因为我们不仅仅要读写这段内存，而且要执行其中的代码，而这对描述符的属性要求是不一样的。这两个段的段基址都是 0，长度都是 4GB。

下面我们就将启动分页的代码做相应的修改（代码3.29）。

代码 3.29 修改后的 SetupPaging（节自 chapter3/h/pmtest8.asm）

```
258 SetupPaging:
259     ; 根据内存大小计算应初始化多少PDE以及多少页表
...
268 .no_remainder:
⇒269    mov     [PageTableNumber], ecx  ; 暂存页表个数
270
271     ; 为简化处理，所有线性地址对应相等的物理地址。并且不考虑内存空洞。
272
273     ; 首先初始化页目录
⇒274    mov     ax, SelectorFlatRW
275     mov     es, ax
⇒276    mov     edi, PageDirBase0
277     xor     eax, eax                ; 此段首地址为 PageDirBase0
⇒278    mov     eax, PageTblBase0 | PG_P  | PG_USU | PG_RWW
279 .1:
280     stosd
281     add     eax, 4096
282     loop    .1                      ; 为了简化，所有页表在内存中是连续的。
283
284     ; 再初始化所有页表
⇒285    mov     eax, [PageTableNumber]  ; 页表个数
```

```
286             mov     ebx, 1024                       ; 每个页表 1024 个 PTE
287             mul     ebx
288             mov     ecx, eax                        ; PTE个数 = 页表个数 * 1024
⇒ 289           mov     edi, PageTblBase0               ; 此段首地址为 PageTblBase0
290             xor     eax, eax
291             mov     eax, PG_P  | PG_USU | PG_RWW
292     .2:
293             stosd
294             add     eax, 4096                       ; 每一页指向 4K 的空间
295             loop    .2
296
⇒ 297           mov     eax, PageDirBase0
298             mov     cr3, eax
```

我们原来并没有把页表个数保存起来，而现在情况发生了变化，我们不只有一个页目录和页表，为了初始化另外的页表时方便起见，在这里增加了一个变量PageTableNumber，页表的个数就存在里面。

在整个初始化页目录和页表的过程中，es始终为SelectorFlatRW。这样，想存取物理地址的时候，只需将地址赋值给edi，那么es:edi指向的就是相应物理地址。比如页目录物理地址为PageDirBase0，第276行将edi赋值为PageDirBase0，es:edi于是指向地址PageDirBase0处，赋值通过指令stosd来实现。初始化页表也是同样的道理。

这样，页目录和页表的准备工作就完成了。不过我们不再在原来的位置调用它，而是新建一个函数PagingDemo，把所有与分页有关的内容全都放进里面，这样，程序看起来结构清晰一些。

根据图3.34和图3.35，我们可以认为在这个程序的实现中有 4 个要关注的要素，分别是ProcPagingDemo、LinearAddrDemo、ProcFoo和ProcBar，我们把它们称为 F4。因为程序开始时LinearAddrDemo指向ProcFoo并且线性地址和物理地址是对等的，所以LinearAddrDemo应该等于ProcFoo。而ProcFoo和ProcBar应该是指定的物理地址，所以LinearAddrDemo也应该是指定的物理地址。也正因为如此，我们使用它们时应该确保使用的是 FLAT 段，即段选择子应该是SelectorFlatC或者SelectorFlatRW。

为了将我们的代码放置在ProcFoo和ProcBar这两处地方，我们先写两个函数，在程序运行时将这两个函数的执行码复制过去就可以了。

ProcPagingDemo要调用 FLAT 段中的LinearAddrDemo，所以如果不想使用段间转移，我们需要把ProcPagingDemo也放进 FLAT 段中。我们需要写一个函数，然后把代码复制到ProcPagingDemo处。

这样看来，F4 虽然都是当做函数来使用，但实际上却都是内存中指定的地址。我们把它们定义为常量（代码3.30）。

代码 3.30 作为函数使用的常量（节自 chapter3/h/pmtest8.asm）

```
13  LinearAddrDemo  equ     00401000h
14  ProcFoo         equ     00401000h
15  ProcBar         equ     00501000h
16  ProcPagingDemo  equ     00301000h
```

将代码填充进这些内存地址的代码就在上文我们提到的PagingDemo中（代码3.31）。

代码 3.31 PagingDemo（节自 chapter3/h/pmtest8.asm）

```
311 PagingDemo:
312         mov     ax, cs
313         mov     ds, ax
314         mov     ax, SelectorFlatRW
315         mov     es, ax
316
317         push    LenFoo
318         push    OffsetFoo
319         push    ProcFoo
320         call    MemCpy
321         add     esp, 12
322
323         push    LenBar
324         push    OffsetBar
325         push    ProcBar
326         call    MemCpy
327         add     esp, 12
328
329         push    LenPagingDemoAll
330         push    OffsetPagingDemoProc
331         push    ProcPagingDemo
332         call    MemCpy
333         add     esp, 12
334
335         mov     ax, SelectorData
336         mov     ds, ax                  ; 数据段选择子
337         mov     es, ax
338
339         call    SetupPaging             ; 启动分页
340
341         call    SelectorFlatC:ProcPagingDemo
342         call    PSwitch                 ; 切换页目录，改变地址映射关系
343         call    SelectorFlatC:ProcPagingDemo
344
345         ret
```

其中用到了名为MemCpy的函数，它复制三个过程到指定的内存地址，类似于 C 语言中的memcpy。但有一点不同，它假设源数据放在ds段中，而目的在es段中。所以在函数的开头，你可以找到分别为ds和es赋值的语句。函数MemCpy也放进文件lib.inc，读者可以自己阅读其代码，在此不再赘述。

被复制的三个过程的代码如代码3.32所示。

代码 3.32 三个过程（节自 chapter3/h/pmtest8.asm）

```
402 PagingDemoProc:
403 OffsetPagingDemoProc    equ     PagingDemoProc - $$
404         mov     eax, LinearAddrDemo
405         call    eax
406         retf
407 LenPagingDemoAll         equ     $ - PagingDemoProc
408
409 foo:
410 OffsetFoo                equ     foo - $$
411         mov     ah, 0Ch                 ; 0000: 黑底    1100: 红字
412         mov     al, 'P'
413         mov     [gs:((80 * 17 + 0) * 2)], ax    ; 屏幕第 17 行, 第 0 列。
414         mov     al, 'o'
415         mov     [gs:((80 * 17 + 1) * 2)], ax    ; 屏幕第 17 行, 第 1 列。
416         mov     [gs:((80 * 17 + 2) * 2)], ax    ; 屏幕第 17 行, 第 2 列。
417         ret
418 LenFoo                   equ     $ - foo
419
420 bar:
```

```
421 OffsetBar                   equ     bar - $$
422         mov     ah, 0Ch                     ; 0000: 黑底    1100: 红字
423         mov     al, 'B'
424         mov     [gs:((80 * 18 + 0) * 2)], ax    ; 屏幕第 18 行, 第 0 列。
425         mov     al, 'a'
426         mov     [gs:((80 * 18 + 1) * 2)], ax    ; 屏幕第 18 行, 第 1 列。
427         mov     al, 'r'
428         mov     [gs:((80 * 18 + 2) * 2)], ax    ; 屏幕第 18 行, 第 2 列。
429         ret
430 LenBar                  equ     $ - bar
```

其实，代码3.32第405行只是一个短调用。foo和bar两个函数中为了简化对段寄存器的使用，仍然使用直接将单个字符写入显存的方法。

我们回过头看代码3.31，其中大部分语句是内存复制工作，但实际上真正激动人心的语句却是代码最后的4个call指令。它们首先启动分页机制，然后调用ProcPagingDemo，再切换页目录，最后又调用一遍ProcPagingDemo。

现在ProcPagingDemo、ProcFoo以及ProcBar的内容我们都已经知道了，由于LinearAddrDemo和ProcFoo相等，并且函数SetupPaging建立起来的是对等的映射关系，所以第一次对ProcPagingDemo的调用反映的就是图3.34的情况。

接下来调用的是PSwitch，我们来看一下这个切换页目录的函数是怎样的(代码3.33)。

代码 3.33 改变地址映射关系（节自 chapter3/h/pmtest8.asm）

```
350 PSwitch:
351         ; 初始化页目录
352         mov     ax, SelectorFlatRW
353         mov     es, ax
354         mov     edi, PageDirBase1           ; 此段首地址为 PageDirBase1
355         xor     eax, eax
356         mov     eax, PageTblBase1 | PG_P  | PG_USU | PG_RWW
357         mov     ecx, [PageTableNumber]
358 .1:
359         stosd
360         add     eax, 4096                   ; 为了简化, 所有页表在内存中是连续的.
361         loop    .1
362
363         ; 再初始化所有页表
364         mov     eax, [PageTableNumber]      ; 页表个数
365         mov     ebx, 1024                   ; 每个页表 1024 个 PTE
366         mul     ebx
367         mov     ecx, eax                    ; PTE个数 = 页表个数 * 1024
368         mov     edi, PageTblBase1           ; 此段首地址为 PageTblBase1
369         xor     eax, eax
370         mov     eax, PG_P  | PG_USU | PG_RWW
371 .2:
372         stosd
373         add     eax, 4096                   ; 每一页指向 4K 的空间
374         loop    .2
375
376         ; 在此假设内存是大于 8M 的
377         mov     eax, LinearAddrDemo
378         shr     eax, 22
379         mov     ebx, 4096
380         mul     ebx
381         mov     ecx, eax
382         mov     eax, LinearAddrDemo
383         shr     eax, 12
384         and     eax, 03FFh          ; 1111111111b (10 bits)
```

```
385         mov     ebx, 4
386         mul     ebx
387         add     eax, ecx
388         add     eax, PageTblBase1
389         mov     dword [es:eax], ProcBar | PG_P | PG_USU | PG_RWW
390
391         mov     eax, PageDirBase1
392         mov     cr3, eax
393         jmp     short .3
394 .3:
395         nop
396
397         ret
```

这个函数前面初始化页目录表和页表的过程与SetupPaging是差不多的，只是紧接着程序增加了改变线性地址LinearAddrDemo对应的物理地址的语句。改变后，LinearAddrDemo将不再对应ProcFoo，而是对应ProcBar。

所以，此函数调用完成之后，对ProcPagingDemo的调用就变成了图3.35所示的情况。

在代码3.33的后半部分，我们把cr3的值改成了PageDirBase1，这个切换过程宣告完成。

我们来看程序的运行情况，结果如图3.36所示。

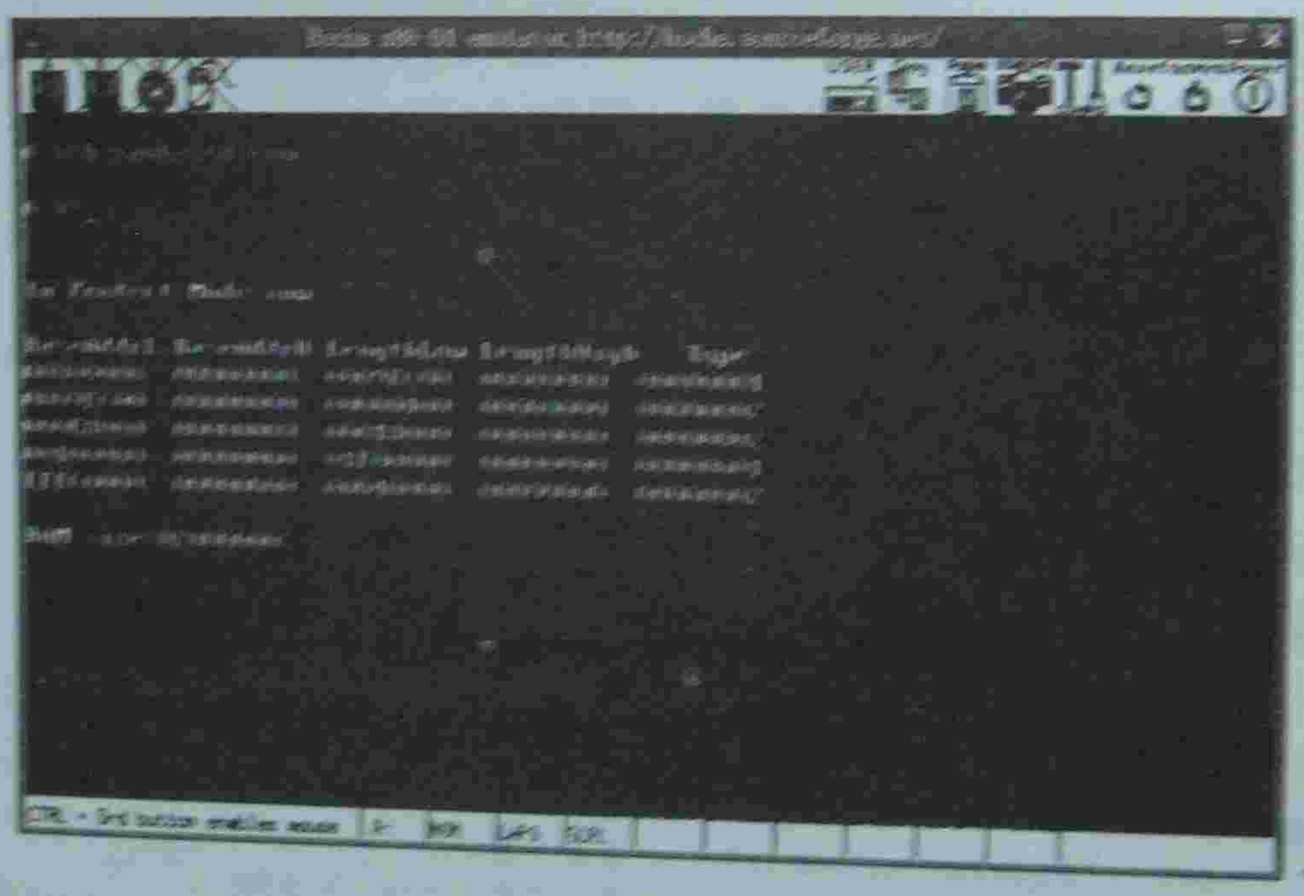


图3.36 pmtest8的执行结果

我们看到红色的Foo和Bar，这说明我们的页表切换起作用了。其实，我们先前提到的不同进程有相同的地址，原理跟本例是类似的，也是在任务切换时通过改变cr3的值来切换页目录，从而改变地址映射关系。

这就是分页的妙处。其实，妙处还不仅仅如此。由于分页机制的存在，程序使用的都是线性地址空间，而不再直接是物理地址。这好像操作系统为应用程序提供了一个不依赖于硬件（物理内存）的平台，应用程序不必关心实际上有多少物理内存，也不必关心正在使用的是哪一段内存，甚至不必关心某一个地址是在物理内存里面还是在硬盘中。总之，操作系统全权负责了这其中的转换工作，我们如今已经了解了这其中所有的原委，它可能有点复杂，但我们已经不再感到惧怕。

3.4 中断和异常

说起中断，好像我们一直在用。最近的一次是我们通过int 15h得到了计算机内存信息。但是不知道你发现没有，我们所有中断的操作都是在实模式下进行的。我们在实模式下用int 15h得到内存信息，然后在保护模式下把它们显示出来。

并不是我们故意把问题搞复杂，而是在保护模式下，中断机制发生了很大变化，原来的中断向量表已经被IDT所代替，实模式下能用的BIOS中断在保护模式下已经不能用了。你可能没有听说过IDT，但看名字可以猜到，它跟GDT、LDT应该有相似的地方。没错，其实它也是个描述符表，叫做中断描述符表(Interrupt Descriptor Table)。IDT中的描述符可以是下面三种之一：

- 中断门描述符
- 陷阱门描述符
- 任务门描述符

IDT的作用是将每一个中断向量和一个描述符对应起来。从这个意义上说，IDT也是一种向量表，虽然它形式上跟实模式下的向量表非常不同。而我们在“调用门初体验”中也曾经提到，中断门和陷阱门是特殊的调用门，所以，虽然本节中我们接触的是新的概念，其核心却只是在原有内容的基础上的一点改变。

让我们看看中断向量到中断处理程序的对应过程（如图3.37所示）。

联系调用门我们知道，其实中断门和陷阱门的作用机理几乎是一样的，只不过使用调用门时使用call指令，而这里我们使用int指令。

刚才我们提到，IDT中可以有中断门、陷阱门或者任务门。但任务门在有些操作系统中根本就没有用到（比如Linux），这里，我们也不做太多关注。中断门和陷阱门的结构如图3.38所示。

对比调用门的结构我们知道，在中断门和陷阱门中BYTE4的低5位变成了保留位，而不再是Param Count。而且，表示TYPE的4位也将变为0xE（中断门）或0xF（陷阱门）。当然，S位仍将是0。

那么，是不是知道了这些我们就可以写一段程序来试验一下中断机制了呢？先不要着急，因为中断不但涉及到处理器以及指令，还涉及处理器与硬件的联系等内容。所以，我们还是要对中断进行全面了解。

3.4.1 中断和异常机制

我们在说到中断时通常将它与异常相提并论，实际上，它们都是程序执行过程中的强制性转移，转移到相应的处理程序。中断通常在程序执行时因为硬件而随机发生，它们通常用来处理处理器外部的事件，比如外围设备的请求。软件通过执行int n指令也可以产生中断。异常则通常在处理器执行指令过程中检测到错误时发生，比如遇到零除的情况。处理器检测的错误条件有很多，比如保护违例、页错误等。

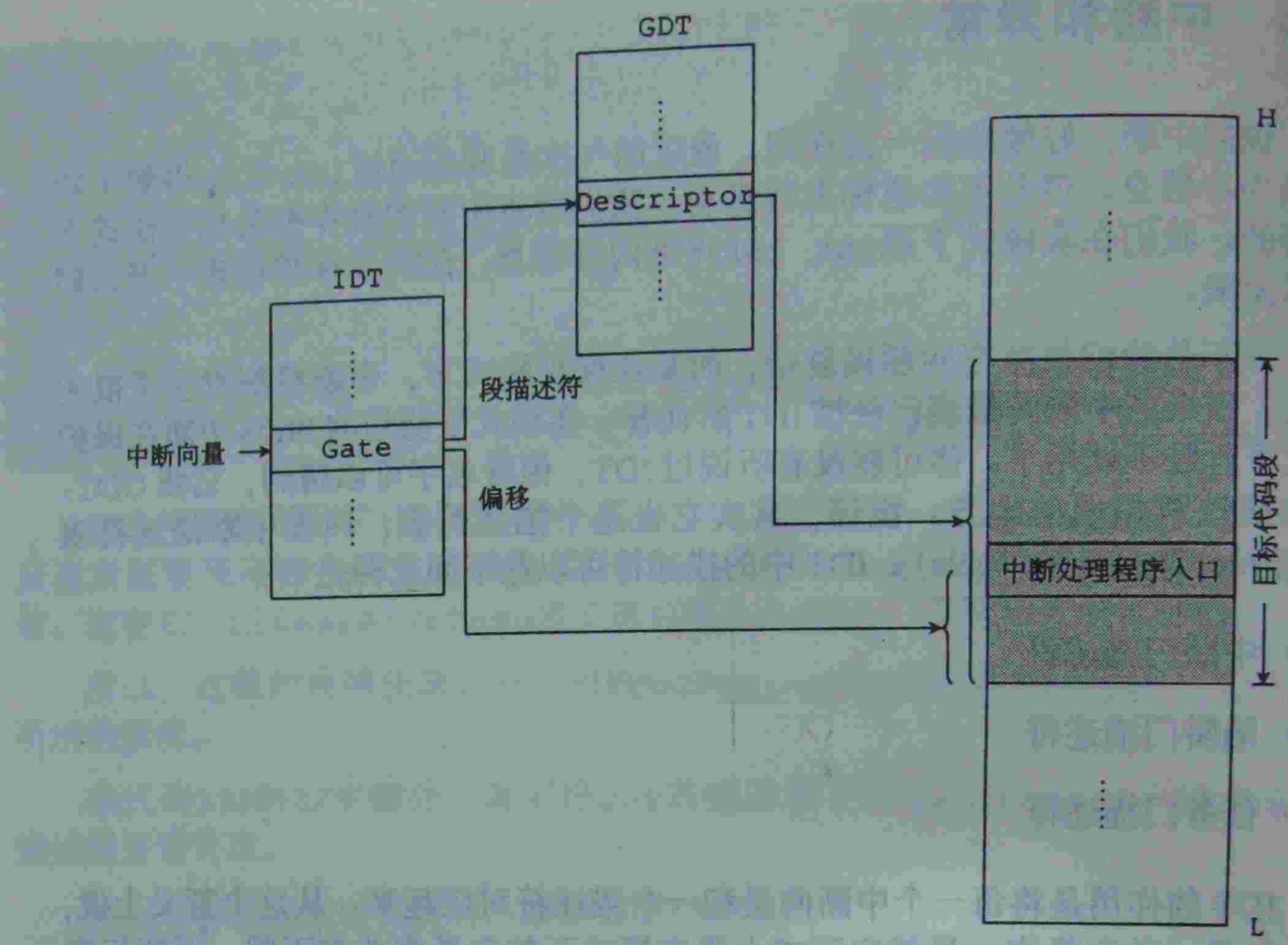


图 3.37 中断向量到中断处理程序的对应过程

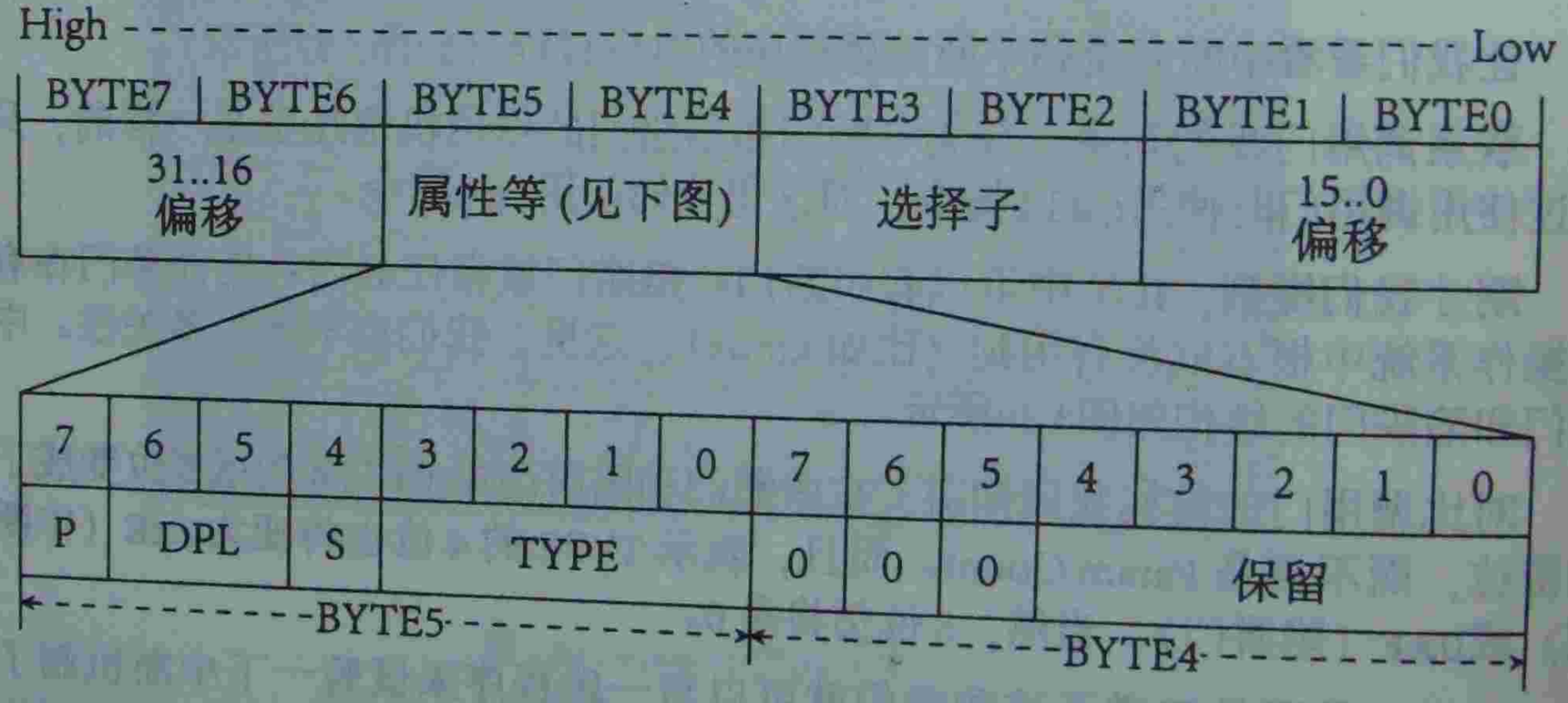


图 3.38 中断门和陷阱门

不管中断还是异常，通俗来讲，都是软件或者硬件发生了某种情形而通知处理器的行为。于是，由此引出两个问题：一是处理器可以对何种类型的通知做出反应；二是当接到某种通知时做出何种处理。

其实，仔细再想一下的话，就发现这里又引出一个问题。假设处理器可以处理 A、B、C 三种中断（异常），分别进行 α、β、γ 三种处理，我们得有一种方法把 A、B、C 和 α、β、γ 对应起来。实际上，解决这个问题的方法就是我们前文中提到的中断向量。每一种中断（异常）都会对应一个中断向量号，而这个向量号通过 IDT 就与相应的中断处理程序对应起来（见图3.37）。

那么，处理器到底能处理哪些中断或异常呢？表3.8不但给出了处理器可以处理的中断和异常列表，而且给出了它们对应的向量号以及其他一些描述。

表 3.8 保护模式中的中断和异常

向量号	助记符	描述	类型	出错码	源
0	#DE	除法错	Fault	无	DIV 和 IDIV 指令
1	#DB	调试异常	Fault/Trap	无	任何代码和数据的访问
2	—	非屏蔽中断	Interrupt	无	非屏蔽外部中断
3	#BP	调试断点	Trap	无	指令 INT 3
4	#OF	溢出	Trap	无	指令 INTO
5	#BR	越界	Fault	无	指令 BOUND
6	#UD	无效（未定义的）操作码	Fault	无	指令 UD2 或者无效指令
7	#NM	设备不可用（无数学协处理器）	Fault	无	浮点或 WAIT/FWAIT 指令
8	#DF	双重错误	Abort	有 (0)	所有能产生异常或 NMI 或 INTR 的指令
9		协处理器段越界（保留）	Fault	无	浮点指令（386 之后的 IA32 处理器不再产生此种异常）
10	#TS	无效 TSS	Fault	有	任务切换或访问 TSS 时
11	#NP	段不存在	Fault	有	加载段寄存器或访问系统段时
12	#SS	堆栈段错误	Fault	有	堆栈操作或加载 SS 时
13	#GP	常规保护错误	Fault	有	内存或其他保护检验
14	#PF	页错误	Fault	有	内存访问
15	—	Intel 保留，未使用			
16	#MF	x87FPU 浮点错（数学错）	Fault	无	x87FPU 浮点指令或 WAIT/FWAIT 指令
17	#AC	对齐检验	Fault	有 (0)	内存中的数据访问（486 开始支持）
18	#MC	Machine Check	Abort	无	错误码（如果有的话）和源依赖于具体模式（奔腾 CPU 开始支持）
19	#XF	SIMD 浮点异常	Fault	无	SSE 和 SSE2 浮点指令（奔腾 III 开始支持）
20~31	—	Intel 保留，未使用			
32~255	—	用户定义中断	Interrupt		外部中断或 int n 指令

看到“助记符”这一栏你可能想起来了，前文中我们对于 #GP、#TS 等异常已经有所提及。而“类型”一栏可能让你有些迷惑，这里之所以笔者没有把它翻译成中文，是怕翻译不准确而造成歧义，而且只有几个单词，理解上不会有什么麻烦。实际上，Fault、Trap 和 Abort 是异常的三种类型，它们的具体解

释如下：

- Fault 是一种可被更正的异常，而且一旦被更正，程序可以不失连续性地继续执行。当一个 fault 发生时，处理器会把产生 fault 的指令之前的状态保存起来。异常处理程序的返回地址将会是产成 fault 的指令，而不是其后的那条指令。
- Trap 是一种在发生 trap 的指令执行之后立即被报告的异常，它也允许程序或任务不失连续性地继续执行。异常处理程序的返回地址将会是产成 trap 的指令之后的那条指令。
- Abort 是一种不总是报告精确异常发生位置的异常，它不允许程序或任务继续执行，而是用来报告严重错误的。

当然，只要你明白了它们分别的含义，当然可以称呼它们为错误、陷阱和终止。而且有一些书籍里面也的确是这样做的。

3.4.2 外部中断

刚才我们着重讨论了异常，现在再来看一下中断。

中断产生的原因有两种，一种是外部中断，也就是由硬件产生的中断，另一种是由指令int n产生的中断。

指令int n产生中断时的情形如图3.37所示，n即为向量号，它类似于调用门的使用。

外部中断的情况则复杂一些，因为需要建立硬件中断与向量号之间的对应关系。外部中断分为不可屏蔽中断（NMI）和可屏蔽中断两种，分别由 CPU 的两根引脚 NMI 和 INTR 来接收，如图3.39所示。

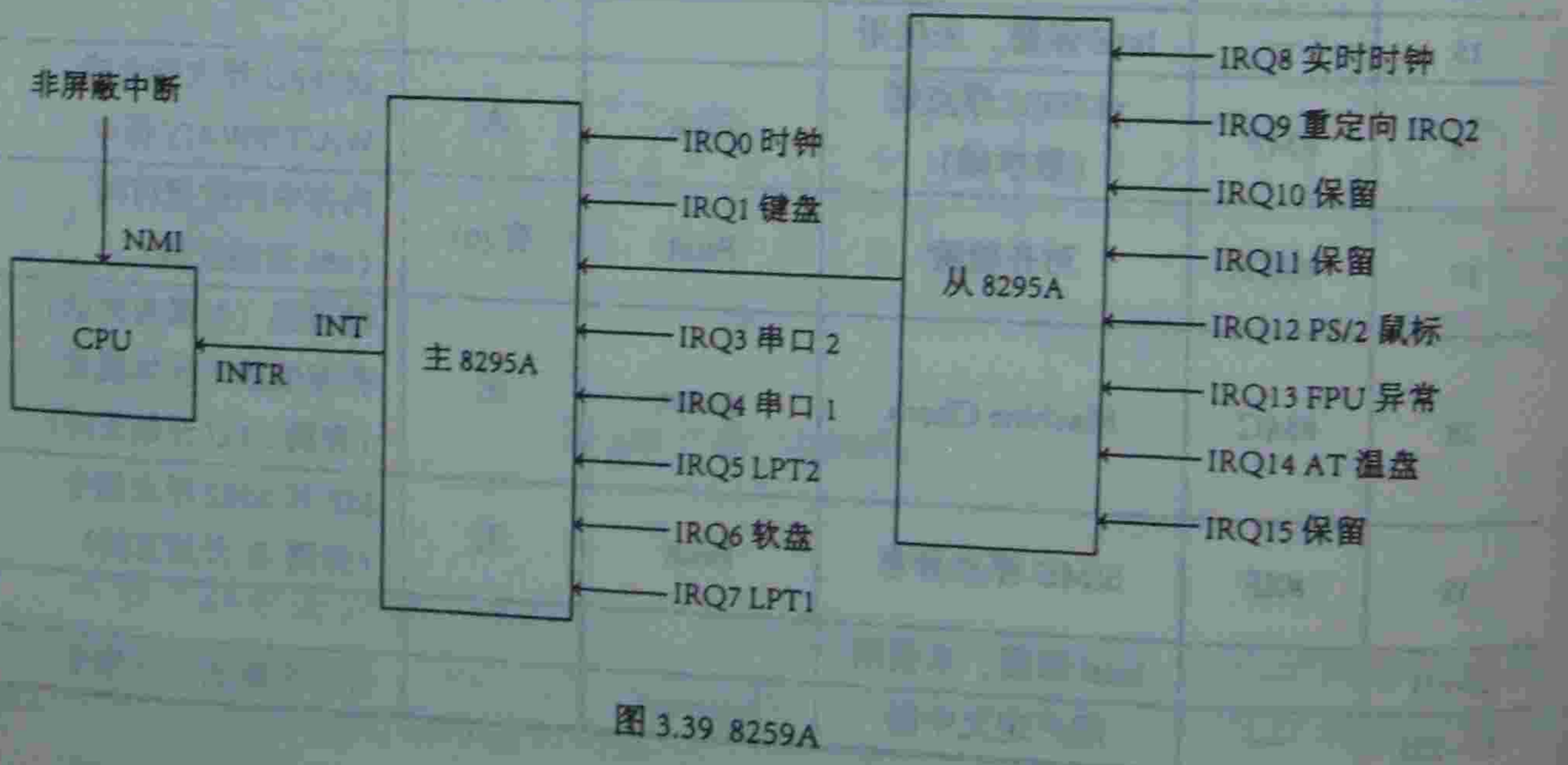


图 3.39 8259A

NMI 不可屏蔽，因为它与 IF 是否被设置无关。NMI 中断对应的中断向量号为2，这在表3.8中已经有所说明。

可屏蔽中断与 CPU 的关系是通过对可编程中断控制器 8259A 建立起来的。如果你是第一次听说 8259A，那么你可以认为它是中断机制中所有外围设备的

一个代理，这个代理不但可以根据优先级在同时发生中断的设备中选择应该处理的请求，而且可以通过对其寄存器的设置来屏蔽或打开相应的中断。它和CPU的连接如图3.39所示。

由图3.39我们知道，与CPU相连的不是一片，而是两片级联的8259A，每个8259A有8根中断信号线，于是两片级联总共可以挂接15个不同的外部设备。那么，这些设备发出的中断请求如何与中断向量对应起来呢？就是通过对8259A的设置完成的。在BIOS初始化它的时候，IRQ0~IRQ7被设置为对应向量号08h~0Fh，而通过表3.8我们知道，在保护模式下向量号08h-0Fh已经被占用了，所以我们不得不重新设置主从8259A。

还好，8259A是可编程中断控制器，对它的设置并不复杂，是通过向相应的端口写入特定的ICW（Initialization Command Word）来实现的。主8259A对应的端口地址是20h和21h，从8259A对应的端口地址是A0h和A1h。ICW共有4个，每一个都是具有特定格式的字节。为了先对初始化8259A的过程有一个概括的了解，我们过一会儿再来关注每一个ICW的格式，现在，先来看一下初始化过程：

1. 往端口20h（主片）或A0h（从片）写入ICW1。
2. 往端口21h（主片）或A1h（从片）写入ICW2。
3. 往端口21h（主片）或A1h（从片）写入ICW3。
4. 往端口21h（主片）或A1h（从片）写入ICW4。

这4步的顺序是不能颠倒的。

我们现在来看一下4个如图3.40所示的ICW的格式。

我们看到，在写入ICW2时涉及与中断向量号的对应，这便是窍门所在了。

原本就迫不及待的你，知道了这一秘密之后是不是想马上一展身手开启中断呢？好，我们现在就开始动手。其实，所要做的工作不外乎设置8259A和建立IDT两大部分，我们以pmtest8.asm为基础对代码进行修改，形成pmtest9.asm。

3.4.3 编程操作8259A

把设置8259A的代码写进一个函数（代码3.34）。

代码3.34 设置8259A（节自chapter3/i/pmtest9a.asm）

```
283 Init8259A:
284         mov     al, 011h
285         out     020h, al        ; 主8259, ICW1.
286         call    io_delay
287
288         out     0A0h, al        ; 从8259, ICW1.
289         call    io_delay
290
291         mov     al, 020h        ; IRQ0 对应中断向量 0x20
292         out     021h, al        ; 主8259, ICW2.
293         call    io_delay
294
```

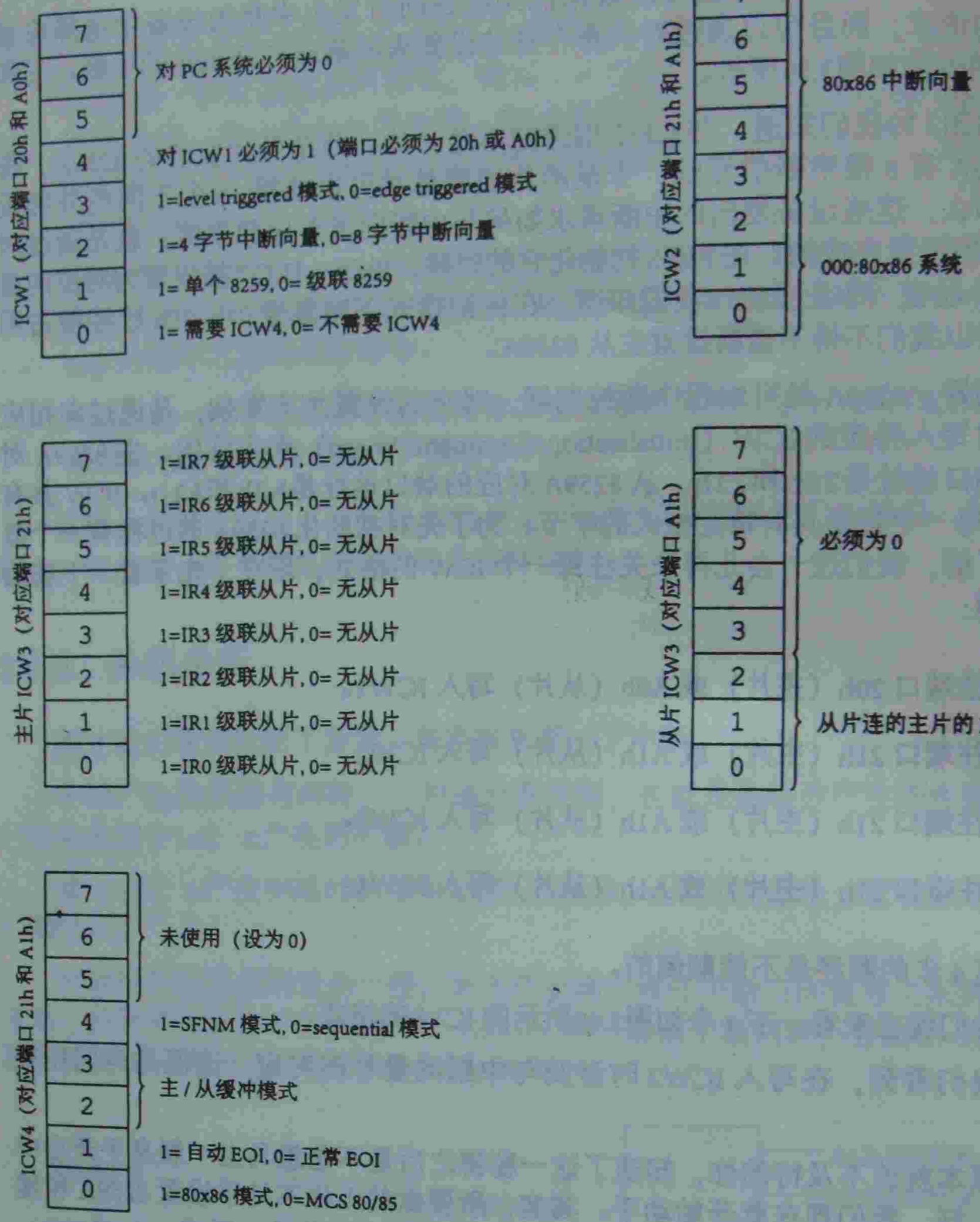


图 3.40 ICW 的格式

```
295     mov     al, 028h        ; IRQ8 对应中断向量 0x28
296     out     0A1h, al        ; 从8259, ICW2.
297     call    io_delay
298
299     mov     al, 004h        ; IR2 对应从8259
300     out     021h, al        ; 主8259, ICW3.
301     call    io_delay
302
303     mov     al, 002h        ; 对应主8259的 IR2
304     out     0A1h, al        ; 从8259, ICW3.
305     call    io_delay
306
307     mov     al, 001h
308     out     021h, al        ; 主8259, ICW4.
309     call    io_delay
310
```

```
311         out     0A1h, al            ; 从8259, ICW4.
312         call    io_delay
313
314         mov     al, 11111110b       ; 仅仅开启定时器中断
315         ;mov    al, 11111111b       ; 屏蔽主8259所有中断
316         out     021h, al            ; 主8259, OCW1.
317         call    io_delay
318
319         mov     al, 11111111b       ; 屏蔽从8259所有中断
320         out     0A1h, al            ; 从8259, OCW1.
321         call    io_delay
322
323         ret
```

这段代码分别往主、从两个 8259A 各写入了 4 个 ICW。在往主 8259A 写入 ICW2 时，我们看到 IRQ0 对应了中断向量号20h，于是，IRQ0~IRQ7 就对应中断向量20h~27h; 类似地，IRQ8~IRQ15 对应中断向量28h~2Fh。对照表3.8我们知道，20h~2Fh处于用户定义中断的范围内。

在这段代码的后半部分，我们通过对端口21h和A1h的操作屏蔽了所有的外部中断，这一次写入的不再是ICW了，而是OCW（Operation Control Word）。OCW 共有 3 个，OCW1、OCW2 和 OCW3。由于我们只在两种情况下用到它，因此并不需要了解所有的内容。这两种情况是：

- 屏蔽或打开外部中断。
- 发送 EOI 给 8259A 以通知它中断处理结束。

若想屏蔽或打开外部中断，只需要往 8259A 写入 OCW1 就可以了，OCW1 的格式如图3.41所示。

图 3.41 OCW1

可见，若想屏蔽某一个中断，将对应那一位设成 1 就可以了。实际上，OCW1 是被写入了中断屏蔽寄存器（IMR，全称 Interrupt Mask Register）中，当一个中断到达，IMR 会判断此中断是否应被丢弃。

说起 EOI，如果你有过在实模式下的汇编经验，那么对它应该不会陌生。当每一次中断处理结束，需要发送一个 EOI 给 8259A，以便继续接收中断。而发送 EOI 是通过往端口20h或A0h写 OCW2 来实现的。OCW2 的格式如图3.42所示。

发送 EOI 给 8259A 可以由如下代码完成：

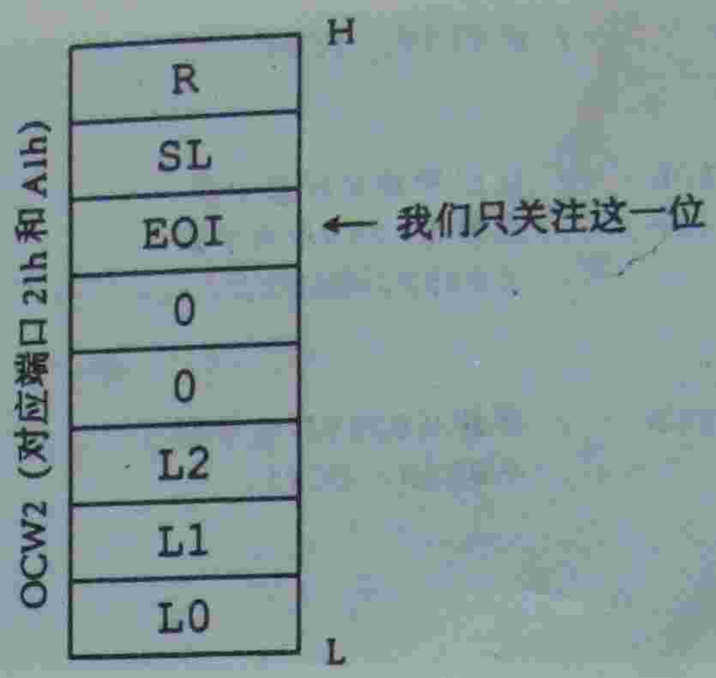


图 3.42 OCW2

```
mov     al, 20h
out     20h或A0h, al
```

而对于 OCW2 其他各位的作用，我们完全可以暂时不予理会。

对于代码3.34，还有一点要说明，每一次 I/O 操作之后都调用了一个延迟函数io_delay以等待操作的完成。函数io_delay很简单，调用了 4 个nop指令（代码3.35）。

代码 3.35 io 延迟函数（节自 chapter3/i/pmtest9a.asm）

```
351 io_delay:
352         nop
353         nop
354         nop
355         nop
356         ret
```

在相应的位置添加调用Init8259A的指令之后，对 8259A 的操作就结束了，我们下面就来建立一个 IDT。

3.4.4 建立 IDT

为了操作方便，我们把 IDT 放进一个单独的段中（代码3.36）。

代码 3.36 IDT（节自 chapter3/i/pmtest9a.asm，后被修改）

```
97  [SECTION .idt]
98  ALIGN   32
99  [BITS   32]
100 LABEL_IDT:
101 ; 门                        目标选择子,              偏移, DCount, 属性
102 %rep 255
103 Gate    SelectorCode32, SpuriousHandler, 0, DA_386IGate
104 %endrep
105
106 IdtLen      equ     $ - LABEL_IDT
107 IdtPtr      dw      IdtLen - 1          ; 段界限
108             dd      0                   ; 基地址
109 ; END of [SECTION .idt]
```

看得出，这个IDT真的是不能再简单了，全部的255个描述符完全相同。这里利用了NASM的%rep预处理指令，将每一个描述符都设置为指向SelectorCode32:SpuriousHandler的中断门。SpuriousHandler也很简单，在屏幕的右上角打印红色的字符“!”，然后进入死循环（代码3.37）。

代码3.37 IDT（节自chapter3/i/pmtest9a.asm）

```
358 _SpuriousHandler:
359 SpuriousHandler equ     _SpuriousHandler - $$
360         mov     ah, 0Ch                 ; 0000: 黑底    1100: 红字
361         mov     al, '!'
362         mov     [gs:((80 * 0 + 75) * 2)], ax    ; 屏幕第 0 行, 第 75 列.
363         jmp     $
364         iretd
```

加载IDT的代码与对GDT的处理非常类似（代码3.38）。

代码3.38 加载IDTR（节自chapter3/i/pmtest9a.asm）

```
201         ; 为加载 IDTR 作准备
202         xor     eax, eax
203         mov     ax, ds
204         shl     eax, 4
205         add     eax, LABEL_IDT              ; eax <- idt 基地址
206         mov     dword [IdtPtr + 2], eax ; [IdtPtr + 2] <- idt 基地址
...
211         ; 关中断
212         cli
213
214         ; 加载 IDTR
215         lidt    [IdtPtr]
```

在执行lidt之前用cli指令清IF位，暂时不响应可屏蔽中断。

其实，到这里为止，我们的中断机制已经初始化完毕了，不过此时运行的话，你会发现程序无法正常回到实模式。因为IDTR以及8259A等内容已经被我们改变，要想顺利跳回实模式还要将它们恢复原样才行。恢复8259A等与回到实模式相关的代码请读者参考附书光盘中的pmtest9.asm。

不过，即便添加了回到实模式的代码，我们仍然看不出任何效果。虽然我们已经完成了保护模式下中断异常处理机制的初始化，但并没有利用中断来做任何事。下面就继续修改代码。

3.4.5 实现一个中断

前文提到过，指令int n用起来很像是调用门的使用。既然我们已经熟悉了调用门，就不妨以此为突破口，试着用一下int指令看看效果如何。

在[SECTION .s32]中添加代码（代码3.39）。

代码3.39 中断（节自chapter3/i/pmtest9b.asm）

```
265         call    Init8259A
266         int     080h
```

由于IDT中所有的描述符都指向SelectorCode32:SpuriousHandler处，所以，无论我们添加的代码调用几号中断，都应该在屏幕的右上角打印出

红色的字符。运行一下，你会看到，屏幕右上角出现红色的“!”，并且程序进入死循环。

我们不妨再往前走一小步，让程序变得稍微优雅一点。

修改一下 IDT，把第80h号中断单独列出来，并新增加一个函数来处理这个中断：UserIntHandler。UserIntHandler与SpuriousHandler类似，只是在函数末尾通过iretd指令返回，而不是进入死循环（代码3.40）。

代码 3.40 UeserIntHandler（节自 chapter3/i/pmtest9c.asm）

```
97  [SECTION .idt]
98  ALIGN   32
99  [BITS   32]
100 LABEL_IDT:
101 ; 门                        目标选择子,             偏移, DCount, 属性
102 %rep 128
103                 Gate    SelectorCode32, SpuriousHandler, 0, DA_386IGate
104 %endrep
105 .080h:          Gate    SelectorCode32, UserIntHandler,  0, DA_386IGate
106
107 IdtLen         equ     $ - LABEL_IDT
108 IdtPtr         dw      IdtLen - 1      ; 段界限
109                 dd      0               ; 基地址
110 ; END of [SECTION .idt]
    ...
363 _UserIntHandler:
364 UserIntHandler  equ     _UserIntHandler - $$
365         mov     ah, 0Ch                         ; 0000: 黑底    1100: 红字
366         mov     al, 'I'
367         mov     [gs:((80 * 0 + 70) * 2)], ax    ; 屏幕第 0 行, 第 70 列。
368         iretd
```

运行结果如图3.43所示，可以看到红色的字符“I”出现在屏幕右上方。

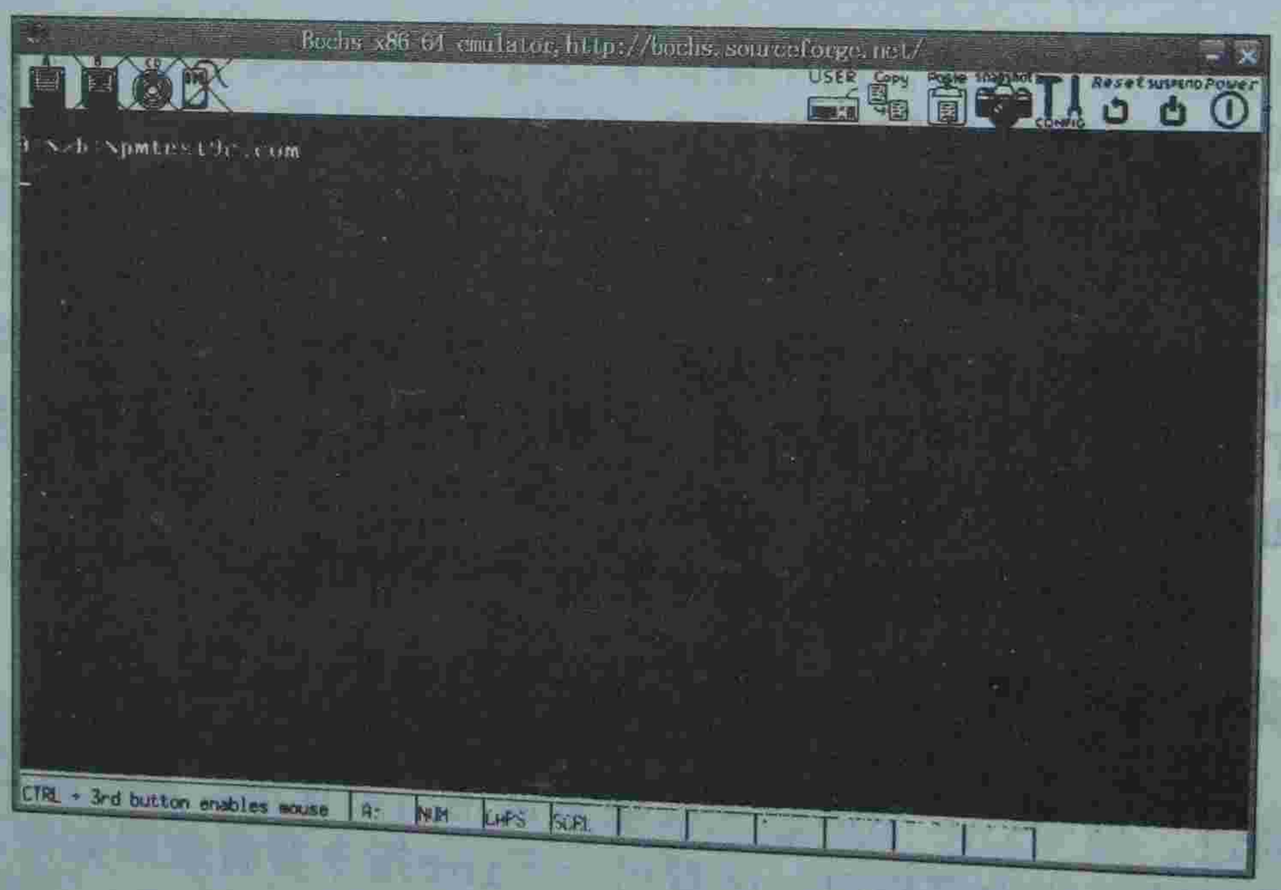


图 3.43 pmtest9c 的执行结果

3.4.6 时钟中断试验

虽然上一节中我们实现了一个中断，但是你可能感觉并没有多少成就感，因为它用起来跟调用门真的差不多。而且，我们辛辛苦苦学习的如何设置

8259A 到现在也没有派上用场。如果你已经感到乏味的话，那么下面的试验你一定会很感兴趣，因为我们即将打开时钟中断（IRQ0），来一点新鲜的体验。

我们提到过，可屏蔽中断与 NMI 的区别在于是否受到 IF 位的影响，而 8259A 的中断屏蔽寄存器（IMR）也影响着中断是否会被响应。所以，外部可屏蔽中断的发生就受到两个因素的影响，只有当 IF 位为 1，并且 IMR 相应位为 0 时才会发生。

那么，如果我们想打开时钟中断的话，一方面不仅要设计一个中断处理程序，另一方面还要设置 IMR，并且设置 IF 位。设置 IMR 可以通过写 OCW2 来完成，而设置 IF 可以通过指令sti来完成。

我们先来写一个时钟中断处理程序，原则仍然是越简单越好（代码3.41）。

代码 3.41 ClockHandler（节自 chapter3/i/pmtest9.asm）

```
387 _ClockHandler:
388 ClockHandler    equ     _ClockHandler - $$
389         inc     byte [gs:((80 * 0 + 70) * 2)]   ; 屏幕第 0 行，第 70 列。
390         mov     al, 20h
391         out     20h, al                         ; 发送 EOI
392         iretd
```

看得出，这个中断处理程序当真是不能再简单了，除了发送 EOI 的两行语句以及iretd，只有一条指令，就是把屏幕第 0 行、第 70 列的字符增一，变成 ASCII 码表中位于它后面的字符。如果我们在调用80h号中断之后打开中断的话，由于第 0 行、第 70 列处已被写入字符 I，所以第一次中断发生时那里会变成字符 J，再一次中断则变成 K，以后每发生一次时钟中断，字符就会变化一次，就会看到不断变化中的字符。

修改初始化 8259A 的代码，时钟中断不再屏蔽（代码3.42）。

代码 3.42 打开时钟中断（节自 chapter3/i/pmtest9.asm）

```
⇨ 343         mov     al, 11111110b   ; 仅仅开启定时器中断
  344         out     021h, al        ; 主8259, OCW1.
  345         call    io_delay
  346
  347         mov     al, 11111111b   ; 屏蔽从8259所有中断
  348         out     0A1h, al        ; 从8259, OCW1.
  349         call    io_delay
  350
  351         ret
```

把 IDT 修改成代码3.43的样子。

代码 3.43 打开时钟中断（节自 chapter3/i/pmtest9.asm）

```
102 [SECTION .idt]
103 ALIGN   32
104 [BITS   32]
105 LABEL_IDT:
106 ; 门                    目标选择子,              偏移, DCount, 属性
107 %rep 32
108                 Gate    SelectorCode32, SpuriousHandler,      0, DA_386IGate
109 %endrep
110 .020h:          Gate    SelectorCode32,    ClockHandler,      0, DA_386IGate
111 %rep 95
112                 Gate    SelectorCode32, SpuriousHandler,      0, DA_386IGate
113 %endrep
```

```
114 .080h:          Gate    SelectorCode32,  UserIntHandler,      0, DA_386IGate
115
116 IdtLen          equ     $ - LABEL_IDT
117 IdtPtr          dw      IdtLen - 1      ; 段界限
118                 dd      0               ; 基地址
119 ; END of [SECTION .idt]
```

按理说，现在在调用80h号中断之后执行sti来打开中断，效果就应该可以看到了。可是有一个问题：程序马上会继续执行，可能没等第一个中断发生程序已经执行完并退出了。所以，我们需要让程序停留在某个地方，干脆让它死循环吧，这样虽然不雅，却简单易行（代码3.44）。

代码 3.44 演示时钟中断（节自 chapter3/i/pmtest9.asm）

```
288         int     080h
289         sti
290         jmp     $
```

运行，效果怎么样？是不是有字符在跳动？

静态的图片无法表示出动态的过程，图3.44是在一瞬间抓拍的图片，刚好显示到字符“z”。

图 3.44 pmtest9 的执行结果

我想此刻你一定像我一样兴奋，因为看到字符跳动的感觉真的不错。是啊，时钟中断的实现也证明了我们先前对于 8259A 的设置是正确无误的。

3.4.7 几点额外说明

3.4.7.1 特权级变换

为简单起见，我们上面的代码始终运行在 ring0，但在实际应用中，中断的产生大多是带有特权级变换的。实际上，通过中断门和陷阱门的中断就相当于用call指令调用一个调用门，涉及的特权级变换的规则是完全一样的。读者可以参考第54页中表3.4以及相关内容。

3.4.7.2 中断或异常发生时的堆栈变化

如图3.45所示，中断或异常发生，以及相应的处理程序结束时，堆栈都会发生变化。

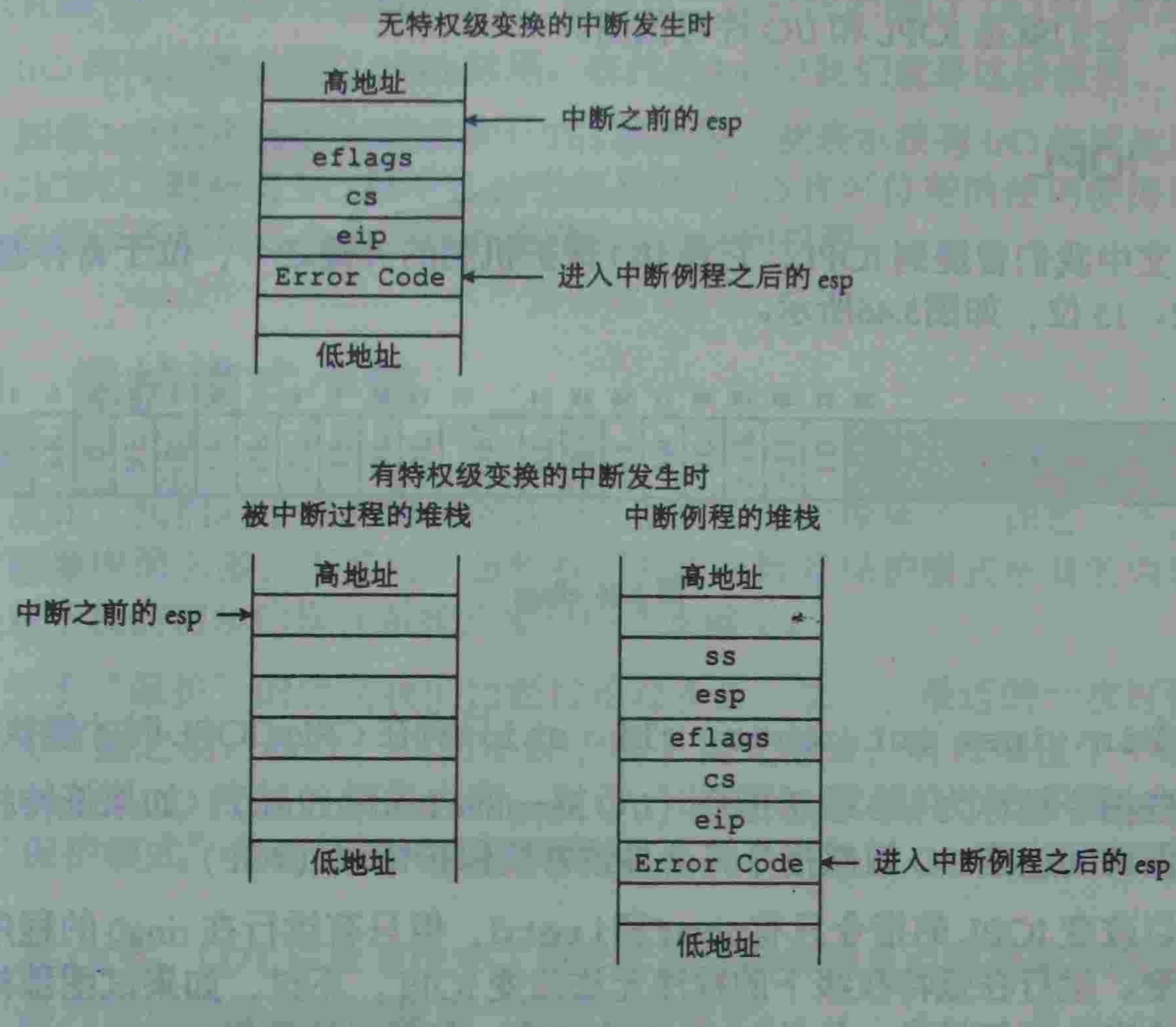


图 3.45 中断或异常发生时的堆栈

如果中断或异常发生时没有特权级变换，那么eflags、cs、eip将依次被压入堆栈，如果有出错码的话，出错码将在最后被压栈。有特权级变换的情况下同样会发生堆栈切换，此时，ss和esp将被压入内层堆栈，然后是eflags、cs、eip、出错码（如果有的话）。

不总是会有出错码（Error Code），具体情况请参考表3.8。

从中断或异常返回时必须使用指令iretd，它与ret很相似，只是它同时会改变eflags的值。需要注意的是，只有当 CPL 为 0 时，eflags中的 IOPL 域才会改变，而且只有当 CPL≤IOPL 时，IF 才会被改变。关于 IOPL 的更多细节请参考3.5节。

另外，iretd执行时 Error Code 不会被自动从堆栈中弹出，所以，执行它之前要先将它从栈中清除掉。

3.4.7.3 中断门和陷阱门的区别

上文中，我们总是把中断门和陷阱门放在一起介绍。实际上，它们之间存在一个微小的差别，就是对中断允许标志 IF 的影响。由中断门向量引起的中断会复位 IF，因为可以避免其他中断干扰当前中断的处理。随后的iret指令会从堆栈上恢复 IF 的原值;而通过陷阱门产生的中断不会改变 IF。

3.5 保护模式下的 I/O

毫无疑问，对 I/O 的控制权限是很重要的一项内容，保护模式对此也做了限制，用户进程如果不被许可是无法进行 I/O 操作的。这种限制通过两个方面来实现，它们就是 IOPL 和 I/O 许可位图。

3.5.1 IOPL

上文中我们曾提到 IOPL，它是 I/O 保护机制的关键之一，位于寄存器 eflags 的第 12、13 位，如图3.46所示。

图 3.46 eflags

指令in、ins、out、outs、cli、sti只有在 CPL≤IOPL 时才能执行。

这些指令被称为 I/O 敏感指令（I/O Sensitive Instructions）。如果低特权级的指令试图访问这些 I/O 敏感指令将会导致常规保护错误（#GP）。

可以改变 IOPL 的指令只有popf和iretd，但只有运行在 ring0 的程序才能将其改变。运行在低特权级下的程序无法改变 IOPL，不过，如果试图那样做的话并不会产生任何异常，只是 IOPL 不会改变，仍然保持原样。

指令popf同样可以用来改变 IF（就好像执行了cli和sti）。然而，在这种情况下，popf也变成了 I/O 敏感指令。只有 CPL≤IOPL 时，popf才可以成功将 IF 改变，否则 IF 将维持原值，不会产生任何异常。

3.5.2 I/O 许可位图（I/O Permission Bitmap）

如果你再回头看图3.20，会发现在 TSS 偏移 102 字节处有一个被称做“I/O 位图基址”的东西，它是一个以 TSS 的地址为基址的偏移，指向的便是 I/O 许可位图。之所以叫做位图，是因为它的每一位表示一个字节的端口地址是否可用。如果某一位为 0，则表示此位对应的端口号可用，为 1 则不可用。由于每一个任务都可以有单独的 TSS，所以每一个任务可以有它单独的 I/O 许可位图。

比如，有一个任务的 TSS 是这样的：

代码 3.45 TSS 举例

```
1  [SECTION .tss3]
2  LABEL_TSS3:
3
4                  ...
5                  DD      SelectorLDT3                    ; LDT
6                  DW      0                               ; 调试陷阱标志
7                  DW      $ - LABEL_TSS3 + 2              ; 指向I/O许可位图
8          times 12 DB     0FFh                            ; 端口 00h--5fh
9                  DB      11111101b                       ; 端口 60h--67h,只允许端口61h的操作
                   DB      0ffh                            ; I/O许可位图结束标志
```

```
10  TSS3Len         equ     $ - LABEL_TSS3
```

由于I/O许可位图开始有12字节内容为0FFh，即有12×8=96位被置为1，所以从端口00h到5Fh共96个端口地址对此任务不可用。同理，接下来的1字节只有第1位（从0开始数）是0，表示这一位对应的端口（61h）可用。

I/O许可位图必须以0FFh结尾。在代码3.45中我们就是这样做的。

如果I/O位图基址大于或等于TSS段界限，就表示没有I/O许可位图，如果CPL≤IOPL，则所有I/O指令都会引起异常。I/O许可位图的使用使得即便在同一特权级下不同的任务也可以有不同的I/O访问权限。

3.6 保护模式小结

至此，我们对于保护模式的认识已经可以告一段落了。回想一下，内容并没有想像中那么多，不是吗？虽然我们并没有涉及保护模式所有的内容，但是这些对于我们初步的操作系统开发已经基本够了。

关于“保护”的含义我们已经讨论过不止一次了，最近的一次讨论是在介绍页式存储之前。现在，我们不仅了解了页式存储，而且知道了中断和异常机制，以及I/O控制的相关内容，对于“保护”，我们终于有了更加全面的理解。“保护模式”包含如下几方面的含义：

- 在GDT、LDT以及IDT中，每一个描述符都有自己的界限和属性等内容，是对描述符所描述对象的一种限定和保护。
- 分页机制中的PDE和PTE都含有R/W以及U/S位，提供了页级保护。
- 页式存储的使用使应用程序使用的是线性地址空间而不是物理地址，于是物理内存就被保护起来。
- 中断不再像实模式下一样使用，也提供特权检验等内容。
- I/O指令不再随便使用，于是端口被保护起来。
- 在程序运行过程中，如果遇到不同特权级间的访问等情况，会对CPL、RPL、DPL、IOPL等内容进行非常严格的检验，同时可能伴随堆栈的切换，这都对不同层级的程序进行了保护。

以上提到的这几个方面可能仍然不能全部概括“保护”二字的含义，但至少可以让我们部分地看到保护模式的真谛所在，以及处理器为操作系统所提供的功能强大的硬件平台。

I like the dreams of the future better than the history of the past.

—— Thomas Jefferson

4 让操作系统走进保护模式

保护模式的内容已讲了这么多，不知你是否还记得我们的操作系统进行到什么地方了。不过，如果你翻翻前面，真的记起来我们已经完成的内容，可能会感到非常失望，因为我们除了写完一个引导扇区，实际上什么都没有做，而且那个引导扇区也非常简陋。

但是，如果你再回想这段时间我们所做的工作，可能就又信心百倍了，因为我们虽然没有直接写自己的 OS，却从头至尾熟悉了保护模式，而且积累了大量的代码，我们完全可以在以后的开发中复用这些代码。同时，保护模式的相关内容已经让我们对于存储管理、特权级控制等内容有了初步的感性认识，这些知识无疑都会在我们今后的开发过程中派上用场。

说到这里，我突然觉得，我们现在的状态就好像箭在弦上，蓄势待发。因为我们已经积累了如此多的知识，并且充满热情，只等待一展身手了。既然已经了解了保护模式，就先让我们的 OS 先进入保护模式吧。

4.1 突破 512 字节的限制

进入保护模式是一件容易的事情，可是我们要做的事情如此之多，总是局限在 512 字节的引导扇区之内总不是长久之计。看来我们需要想个办法。

还好，引导扇区虽小，软盘的容量却大得多。虽然如今硬盘动辄上百 GB，1.44MB 看上去真的很不起眼，但对于我们刚刚开始的操作系统雏形来说已经足够了。所以，可以再建立一个文件，将其通过引导扇区加载入内存，然后将控制权交给它。这样，512 字节的束缚就没有了。

那么，被引导扇区加载进内存的是不是就应该是操作系统内核了呢？我们不妨这样来想，一个操作系统从开机到开始运行，大致经历“引导 → 加载内核入内存 → 跳入保护模式 → 开始执行内核”这样一个过程。也就是说，在内核开始执行之前不但要加载内核，而且还有准备保护模式等一系列工作，如果全都交给引导扇区来做，512 字节很可能是不够用的，所以，不妨把这个过程交给另外的模块来完成，我们把这个模块叫做 Loader。引导扇区负责把 Loader 加载入内存并且把控制权交给它，其他工作放心地交给 Loader 来做，因为它没有 512 字节的限制，将会灵活得多。

在这里，为了操作方便，不妨把软盘做成 FAT12 格式，这样对 Loader 以及今后的 Kernel（内核）的操作将会非常简单易行。

4.1.1 FAT12

FAT12 是 DOS 时代就开始使用的文件系统（File System），直到现在仍然在软盘上使用。几乎所有的文件系统都会把磁盘划分为若干层次以方便组织和管理，这些层次包括：

- 扇区（Sector）：磁盘上的最小数据单元。
- 簇（Cluster）：一个或多个扇区。
- 分区（Partition）：通常指整个文件系统。

我们已经接触过引导扇区，就让我们从这里开始。引导扇区是整个软盘的第 0 个扇区，在这个扇区中有一个很重要的数据结构叫做 BPB（BIOS Parameter Block），引导扇区的格式如表4.1所示，其中名称以BPB_开头的域属于 BPB，以BS_开头的域不属于 BPB，只是引导扇区（Boot Sector）的一部分。

紧接着引导扇区的是两个完全相同的 FAT 表，每个占用 9 个扇区。第二个 FAT 之后是根目录区的第一个扇区。根目录区的后面是数据区，如图4.1所示。

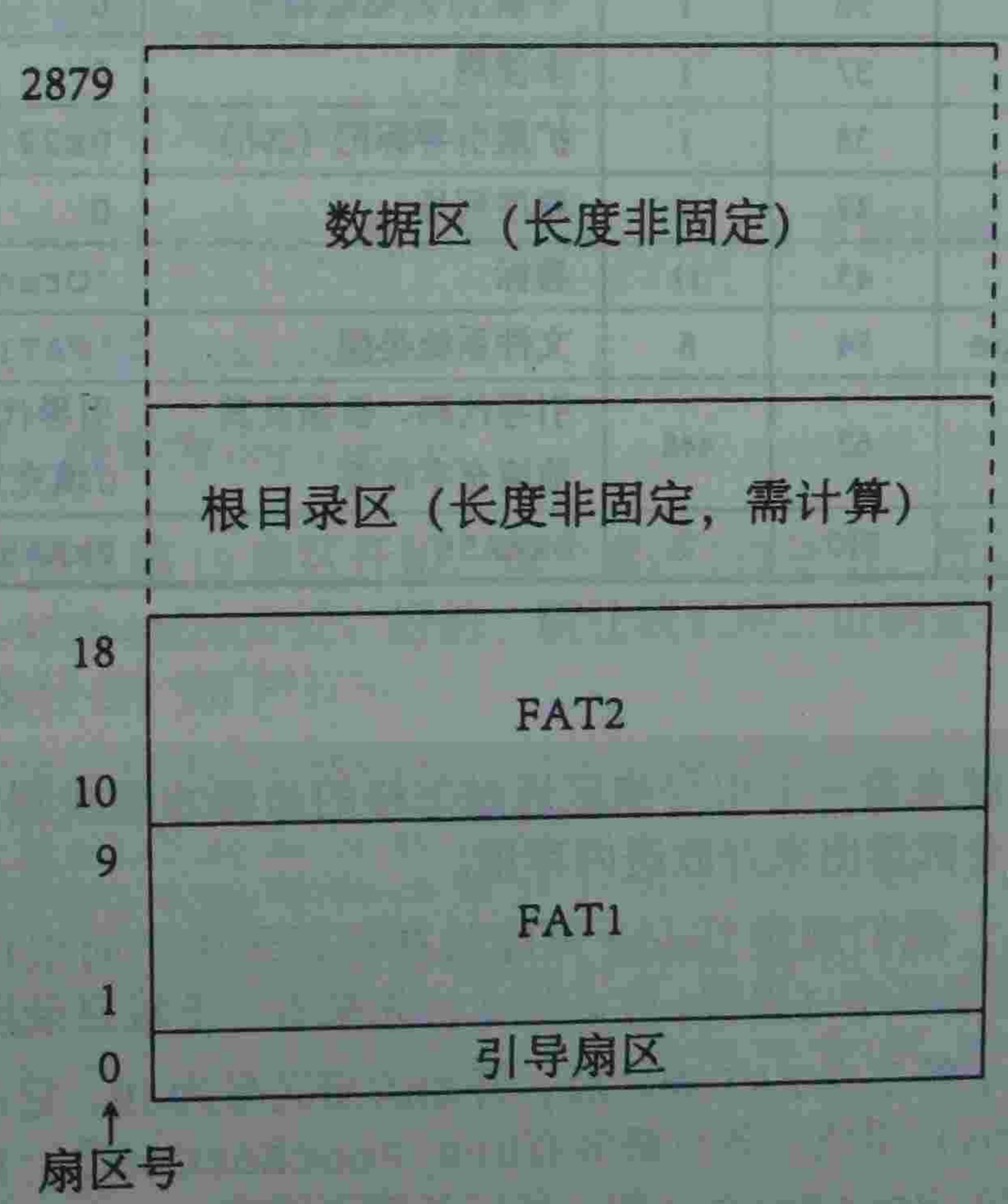


图 4.1 软盘（1.44MB, FAT12）

本来应该讲到 FAT 表了，但单纯数据结构的讲解太不活泼了，我们不妨想一想要用这张软盘做什么。我们是要把 Loader 复制到软盘上并让引导扇区找到

表4.1 FAT12引导扇区的格式

名称	偏移	长度	内容	Orange'S的值
BS_jmpBoot	0	3	一个短跳转指令	jmp LABEL_START nop
BS_OEMName	3	8	厂商名	'ForrestY'
BPB_BytsPerSec	11	2	每扇区字节数	0x200
BPB_SecPerClus	13	1	每簇扇区数	0x1
BPB_RsvdSecCnt	14	2	Boot记录占用多少扇区	0x1
BPB_NumFATs	16	1	共有多少FAT表	0x2
BPB_RootEntCnt	17	2	根目录文件数最大值	0xE0
BPB_TotSec16	19	2	扇区总数	0xB40
BPB_Media	21	1	介质描述符	0xF0
BPB_FATSz16	22	2	每FAT扇区数	0x9
BPB_SecPerTrk	24	2	每磁道扇区数	0x12
BPB_NumHeads	26	2	磁头数（面数）	0x2
BPB_HiddSec	28	4	隐藏扇区数	0
BPB_TotSec32	32	4	如果BPB_TotSec16是0，由这个值记录扇区数	0
BS_DrvNum	36	1	中断13的驱动器号	0
BS_Reserved1	37	1	未使用	0
BS_BootSig	38	1	扩展引导标记（29h）	0x29
BS_VolID	39	4	卷序列号	0
BS_VolLab	43	11	卷标	'OrangeS0.02'
BS_FileSysType	54	8	文件系统类型	'FAT12'
引导代码及其他	62	448	引导代码、数据及其他填充字符等	引导代码（剩余空间被0填充）
结束标志	510	2	0xAA55	0xAA55

并加载它，那么就来看一下引导扇区通过怎样的步骤才能找到文件，以及如何能够把文件内容全都读出来并放进内存里。

为简单起见，我们规定Loader只能放在根目录中，而根目录信息存放在FAT2后面的根目录区中（见图4.1）。那么，先来看一下根目录区。

根目录区位于第二个FAT表之后，开始的扇区号为19，它由若干个目录条目（Directory Entry）组成，条目最多有BPB_RootEntCnt个。由于根目录区的大小是依赖于BPB_RootEntCnt的，所以长度不固定。

根目录区中的每一个条目占用32字节，它的格式如表4.2所示。

看来结构并不复杂，主要定义了文件的名称、属性、大小、日期以及在磁盘中的位置。你可能在想，要是能直观地看到一个真实的目录条目就好了。这并不难做到，我们先来创建一个虚拟软盘，假设是x.img，然后把它作为FreeDos

表 4.2 根目录区中的条目格式

名称	偏移	长度	描述
DIR_Name	0	0xB	文件名 8 字节，扩展名 3 字节
DIR_Attr	0xB	1	文件属性
保留位	0xC	10	保留位
DIR_WrtTime	0x16	2	最后一次写入时间
DIR_WrtDate	0x18	2	最后一次写入日期
DIR_FstClus	0x1A	2	此条目对应的开始簇号
DIR_FileSize	0x1C	4	文件大小

的 B 盘，格式化后就可以方便地往其中添加文件和目录了（比如使用 FreeDos 里的edit.exe）。这样，当我们想查看它的格式时，只需用二进制查看器打开x.img就可以了，跟实际的软盘看起来是一样的（有兴趣的读者可以试着读取一张真实的软盘来对比）。

好了，通过 FreeDos 在这张虚拟软盘中添加以下几个文本文件：

- RIVER.TXT，内容为 riverriverriver。
- FLOWER.TXT，内容为 300 个单词 flower，用来测试文件跨越扇区的情况。（可以先建一个小文件，最后再把它改长，这样可以让它对应的簇不连续，便于观察和理解。）
- TREE.TXT，内容为 treetreetree。

再添加一个 HOUSE 目录，然后在目录\HOUSE 下添加两个文本文件：

- CAT.TXT，内容为 catcatcat。
- DOG.TXT，内容为 dogdogdog。

由于根目录区从第 19 扇区开始，每个扇区 512 字节，所以其第一个字节位于偏移 19×512=9728=0x2600 处。好的，就让我们用二进制查看器来看看x.img的偏移 0x2600 处是什么，如下[1]：

```
▷ xxd -u -a -g 1 -c 16 -s +0x2600 -l 512 x.img
0002600: 52 49 56 45 52 20 20 20 54 58 54 20 00 00 00 00  RIVER   TXT ....
0002610: 00 00 00 00 00 00 74 83 1A 39 02 00 11 00 00 00  ......t..9......
0002620: 46 4C 4F 57 45 52 20 20 54 58 54 20 00 00 00 00  FLOWER  TXT ....
0002630: 00 00 00 00 00 00 72 83 1A 39 03 00 9A 06 00 00  ......r..9......
0002640: 54 52 45 45 20 20 20 20 54 58 54 20 00 00 00 00  TREE    TXT ....
0002650: 00 00 00 00 00 00 62 83 1A 39 04 00 0E 00 00 00  ......b..9......
0002660: 48 4F 55 53 45 20 20 20 20 20 20 10 00 00 00 00  HOUSE       ....
0002670: 00 00 00 00 00 00 74 83 1A 39 05 00 00 00 00 00  ......t..9......
0002680: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00027f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

看到这个画面，你可能一下就注意到了 RIVER、FLOWER、TREE 等单词，是的，它们就在这里。我们以RIVER.TXT为例，它的各项值如表4.3所示。

[1]这里使用 xxd 命令来显示一个扇区的内容。关于 xxd 的详细信息和用法读者可参考其联机文档（使用 man xxd 命令）。

表 4.3 RIVER.TXT 的各项值

名称	值
DIR_Name	RIVER␣␣␣TXT
DIR_Attr	0x20
DIR_WrtTime	0x8374
DIR_WrtDate	0x391A
DIR_FstClus	0x0002
DIR_FileSize	0x00000011

文件名和大小很容易理解，时间和日期信息对我们没有用处，可以忽略掉，我们甚至可以忽略属性。当我们寻找 Loader 时，只要发现文件名正确就认为它是我们要找的那一个文件。最后剩下最重要的信息DIR_FstClus，即文件开始簇号，它告诉我们文件存放在磁盘的什么位置，从而让我们可以找到它。由于一簇只包含一个扇区，所以简化了计算过程，而且下文中说到“簇”的地方，你也可以将它替换成“扇区”。

需要注意的是，数据区的第一个簇的簇号是 2，而不是 0 或者 1。

RIVER.TXT的开始簇号就是 2，也就是说，此文件的数据开始于数据区第一个簇。

数据区第一个簇即第一个扇区在哪里呢？参照图4.1我们发现，必须计算根目录区所占的扇区数才能知道。我们既然已经知道了根目录区条目最多有BPB_RootEntCnt个，扇区数也就容易计算了，假设根目录区共占用$RootDirSectors$个扇区，则有：

$$RootDirSectors = \frac{(BPB_RootEntCnt \times 32) + (BPB_BytsPerSec - 1)}{BPB_BytsPerSec}$$

之所以分子要加上$(BPB_BytsPerSec - 1)$，是为了保证此公式在根目录区无法填满整数个扇区时仍然成立。其实，我们也可以让公式变成这样：

$$RootDirSectors' = \frac{BPB_RootEntCnt \times 32}{BPB_BytsPerSec}$$

如果余数为 0，则：

$$RootDirSectors = RootDirSectors'$$

如果余数不为 0，则：

$$RootDirSectors = RootDirSectors' + 1$$

在本例中，容易算出$RootDirSectors = 14$。所以：

数据区开始扇区号 = 根目录区开始扇区号 + 14 = 19 + 14 = 33

第 33 扇区的偏移量是 0x4200（512×33），让我们看一下这里的内容，如下：

```
▷ xxd -u -s -g 1 -c 16 -s +0x4200 -l 0x11 x.img
0004200: 72 69 76 65 72 72 69 76 65 72 72 69 76 65 72 0D  riverriverriver.
0004210: 0A                                               .
```

果然就是文件的内容，很容易，不是吗？

你可能会想，既然我们通过根目录区已经找到了文件并看到了内容，那我们还要 FAT 表干什么？实际上，对于小于 512 字节的文件来说，FAT 表用处真的不大，但如果文件大于 512 字节，我们需要 FAT 表来找到所有的簇（扇区）。

正如你所看到的，FAT 表有两个，FAT2 可看做是 FAT1 的备份，它们通常是一样的。那么 FAT 表的结构是怎样的呢？我们还是先来一点直观的认识，FAT1 的开始扇区号是 1，偏移为 512 字节（0x200），如下：

```
▷ xxd -u -s -g 1 -c 16 -s +0x200 -l 512 x.img
0000200: F0 FF FF FF 8F 00 FF FF FF FF FF FF 09 A0 00 FF  ................
0000210: 0F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000220: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00003f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

一堆看不懂的符号，好像很多F。其实不复杂，它有点像是一个位图，其中，每 12 位称为一个 FAT 项（FATEntry），代表一个簇。第 0 个和第 1 个 FAT 项始终不使用，从第 2 个 FAT 项开始表示数据区的每一个簇，也就是说，第 2 个 FAT 项表示数据区第一个簇，依此类推。前文说过，数据区的第一个簇的簇号是 2，和这里是相呼应的。

要注意的是，由于每个 FAT 项占 12 位，包含一个字节和另一个字节的一半，所以显得有点别扭。具体情况是这样的，假设连续 3 个字节分别如图4.2所示，那么灰色框表示的是前一个 FAT 项（FATEntry1），BYTE1 是 FATEntry1 的低 8 位，BYTE2 的低 4 位是 FATEntry1 的高 4 位；白色框表示的是后一个 FAT 项（FATEntry2），BYTE2 的高 4 位是 FATEntry2 的低 4 位，BYTE3 是 FATEntry2 的高 8 位。

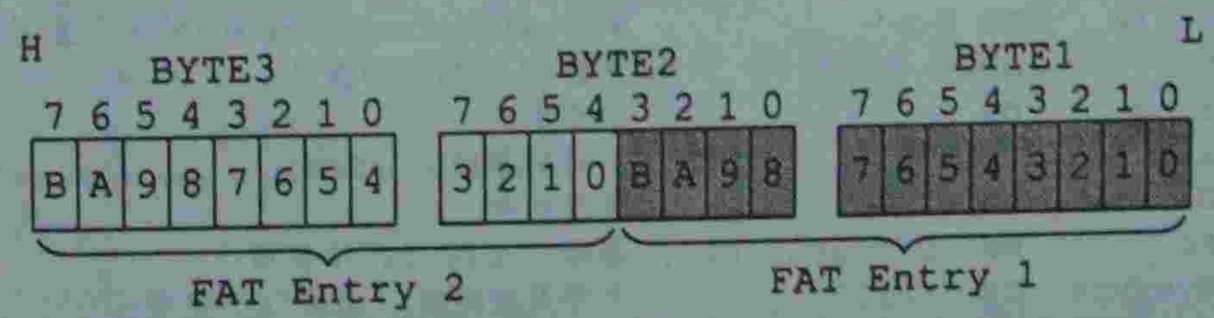


图 4.2 从 FAT 表中得到一个 FAT 项的值

通常，FAT 项的值代表的是文件下一个簇号，但如果值大于或等于0xFF8，则表示当前簇已经是本文件的最后一个簇。如果值为0xFF7，表示它是一个坏簇。

文件RIVER.TXT的开始簇号是 2，对应 FAT 表中的值为0xFFF，表示这个簇已经是最后一个。

我们来看一个长一点的文件FLOWER.TXT，它的DIR_FstClus值为 3，对应第 3 个 FAT 项。结合我们打印出的 FAT 表内容和图4.2我们知道，此 FAT 项值为0x008，也就是说，这个簇不是文件的最后一个簇，下一个簇号为 8。我们再找到第 8 个 FAT 项，发现值为0x009，接下来第 9 个 FAT 项值为0x00A，第

0xA 个 FAT 项值为0xFFF。所以，FLOWER.TXT占用了第 3、8、9、10，共计 4 个簇。

这里需要注意一点，一个 FAT 项可能会跨越两个扇区，这种情况在编码实现的过程中要考虑在内。

好了，到此为止，如何在一个软盘中找到自己想要的文件想必你已经非常清楚了，至于目录\HOUSE以及其他几个文件的情况，读者可以自己研究一下，不但简单而且有趣。下面我们就进入正题，开始修改boot.asm，让它完成寻找 Loader 并加载入内存的任务。我们先给将来的 Loader 起个名字，就叫它LOADER.BIN吧。

4.1.2 DOS 可以识别的引导盘

既然引导扇区需要有 BPB 等头信息才能被微软识别，我们就先加上它，让程序一开头变成下面的形式。

代码 4.1 BPB（节自 chapter4/a/boot.asm）

```
10      jmp short LABEL_START           ; Start to boot.
11      nop                             ; 这个 nop 不可少
12
13      ; 下面是 FAT12 磁盘的头
14      BS_OEMName      DB 'ForrestY'   ; OEM String, 必须 8 个字节
15      BPB_BytsPerSec  DW 512          ; 每扇区字节数
16      BPB_SecPerClus  DB 1            ; 每簇多少扇区
17      BPB_RsvdSecCnt  DW 1            ; Boot 记录占用多少扇区
18      BPB_NumFATs     DB 2            ; 共有多少 FAT 表
19      BPB_RootEntCnt  DW 224          ; 根目录文件数最大值
20      BPB_TotSec16    DW 2880         ; 逻辑扇区总数
21      BPB_Media       DB 0xF0         ; 媒体描述符
22      BPB_FATSz16     DW 9            ; 每FAT扇区数
23      BPB_SecPerTrk   DW 18           ; 每磁道扇区数
24      BPB_NumHeads    DW 2            ; 磁头数(面数)
25      BPB_HiddSec     DD 0            ; 隐藏扇区数
26      BPB_TotSec32    DD 0            ; wTotalSectorCount为0时这个值记录扇区数
27      BS_DrvNum       DB 0            ; 中断 13 的驱动器号
28      BS_Reserved1    DB 0            ; 未使用
29      BS_BootSig      DB 29h          ; 扩展引导标记 (29h)
30      BS_VolID        DD 0            ; 卷序列号
31      BS_VolLab       DB 'OrangeS0.02'; 卷标, 必须 11 个字节
32      BS_FileSysType  DB 'FAT12   '   ; 文件系统类型, 必须 8个字节
33
34  LABEL_START:
```

把生成的Boot.bin写入磁盘引导扇区，运行的效果没有变，仍然会是图1.1的样子。但是，现在的软盘已经能够被 DOS 以及 Linux 识别了，我们已经可以方便地往上添加或删除文件了。

4.1.3 一个最简单的 Loader

要写代码加载 Loader 入内存，可是我们还没有 Loader，怎么办？不要紧，我们先写一个最小的，让它显示一个字符，然后进入死循环，这样，如果加载成功并成功交出了控制权的话，应该可以看到这个字符。

新建一个文件loader.asm，短短几行指令如代码4.2所示。

代码 4.2 最简单的 loader（chapter4/b/loader.asm）

```
1  org     0100h
2
3          mov     ax, 0B800h
4          mov     gs, ax
5          mov     ah, 0Fh                          ; 0000: 黑底    1111: 白字
6          mov     al, 'L'
7          mov     [gs:((80 * 0 + 39) * 2)], ax     ; 屏幕第 0 行, 第 39 列
8
9          jmp     $                                ; 到此停住
```

这段代码可以被编译成.COM文件直接在 DOS 下执行，效果是在屏幕第一行中央出现字符“L”，然后进入死循环。在这里，我们用下面的命令行来编译：

```
▷ nasm loader.asm -o loader.bin
```

需要注意的是，虽然代码4.2编译出的二进制代码加载到内存的任意位置都可以正确执行，但我们要扩展它，为了将来的执行不会出现问题，要保证把它放入某个段内偏移0x100的位置。

4.1.4 加载 Loader 入内存

要加载一个文件入内存的话，免不了要读软盘，这时候就用到 BIOS 中断int 13h。它的用法如表4.4所示。

表 4.4 BIOS 中断 int 13h 的用法

中断号	寄存器		作用
13h	ah=00h	dl=驱动器号（0 表示 A 盘）	复位软驱
	ah=02h ch=柱面（磁道）号 dh=磁头号 es:bx→ 数据缓冲区	al=要读扇区数 cl=起始扇区号 dl=驱动器号（0 表示 A 盘）	从磁盘将数据读入es:bx指向的缓冲区中

我们看到，中断需要的参数不是原来提到的从第 0 扇区开始的扇区号，而是柱面号、磁头号以及在当前柱面上的扇区号 3 个分量，所以需要我们自己来转换一下。对于 1.44MB 的软盘来讲，总共有两面（磁头号 0 和 1），每面 80 个磁道（磁道号 0~79），每个磁道有 18 个扇区（扇区号 1~18）。下面的公式就是软盘容量的由来：

$$2 \times 80 \times 18 \times 512 = 1.44\text{MB}$$

于是，磁头号、柱面（磁道）号和起始扇区号可以用图4.3所示的方法来计算。

我们就先写一个读软盘扇区的函数吧（代码4.3）。

代码 4.3 读软盘扇区（chapter4/b/boot.asm）

```
160 ReadSector:
    ...
```

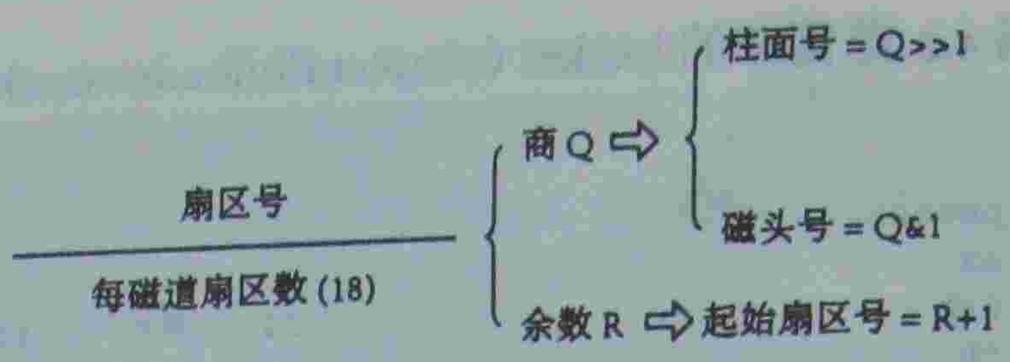


图 4.3 磁头号、柱面号、起始扇区号的计算方法

```
170             push    bp
171             mov     bp, sp
172             sub     esp, 2 ; 辟出两个字节的堆栈区域保存要读的扇区数: byte [bp-2]
173
174             mov     byte [bp-2], cl
175             push    bx                      ; 保存 bx
176             mov     bl, [BPB_SecPerTrk]     ; bl: 除数
177             div     bl                      ; y 在 al 中, z 在 ah 中
178             inc     ah                      ; z ++
179             mov     cl, ah                  ; cl <- 起始扇区号
180             mov     dh, al                  ; dh <- y
181             shr     al, 1                   ; y >> 1 (y/BPB_NumHeads)
182             mov     ch, al                  ; ch <- 柱面号
183             and     dh, 1                   ; dh & 1 = 磁头号
184             pop     bx                      ; 恢复 bx
185             ; 至此, "柱面号, 起始扇区, 磁头号" 全部得到
186             mov     dl, [BS_DrvNum]         ; 驱动器号 (0 表示 A 盘)
187     .GoOnReading:
188             mov     ah, 2                   ; 读
189             mov     al, byte [bp-2]         ; 读 al 个扇区
190             int     13h
191             jc      .GoOnReading            ; 如果读取错误 CF 会被置为 1,
192                                             ; 这时就不停地读, 直到正确为止
193             add     esp, 2
194             pop     bp
195
196             ret
```

如果读者自己试验的时候出现问题，别忘了可以用 Bochs 或者编译成.com 文件进行调试。

还要注意的是，由于这段代码中用到了堆栈，要在程序开头初始化ss和esp（代码4.4）。

代码 4.4 初始化堆栈（chapter4/b/boot.asm）

```
14  BaseOfStack             equ     07c00h  ; 堆栈基地址(栈底, 从这个位置向低地址生长)
    ...
48              mov     ax, cs
49              mov     ds, ax
50              mov     es, ax
51              mov     ss, ax
52              mov     sp, BaseOfStack
```

好了，读扇区的函数写好了，下面我们就开始在软盘中寻找Loader.bin（代码4.5）。

代码 4.5 寻找 loader（chapter4/b/boot.asm）

```
54              xor     ah, ah  ; ┐
55              xor     dl, dl  ; │ 软驱复位
```

```
        int     13h     ; /

; 下面在 A 盘的根目录寻找 LOADER.BIN
;       mov     word [wSectorNo], SectorNoOfRootDirectory
LABEL_SEARCH_IN_ROOT_DIR_BEGIN:
        cmp     word [wRootDirSizeForLoop], 0   ; \. 判断根目录区是不是已经读完
        jz      LABEL_NO_LOADERBIN              ; /  如果读完表示没有找到 LOADER.BIN
        dec     word [wRootDirSizeForLoop]      ; /
        mov     ax, BaseOfLoader
        mov     es, ax                  ; es <- BaseOfLoader
        mov     bx, OffsetOfLoader      ; bx <- OffsetOfLoader
        mov     ax, [wSectorNo]         ; ax <- Root Directory 中的某 Sector 号
        mov     cl, 1
        call    ReadSector

        mov     si, LoaderFileName      ; ds:si -> "LOADER  BIN"
        mov     di, OffsetOfLoader      ; es:di -> BaseOfLoader:0100
        cld
        mov     dx, 10h
LABEL_SEARCH_FOR_LOADERBIN:
        cmp     dx, 0                                   ; \. 循环次数控制,
        jz      LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR      ;  / 如果已经读完了一个 Sector,
        dec     dx                                      ; /  就跳到下一个 Sector
        mov     cx, 11
LABEL_CMP_FILENAME:
        cmp     cx, 0
        jz      LABEL_FILENAME_FOUND    ; 如果比较了 11 个字符都相等, 表示找到
        dec     cx
        lodsb                           ; ds:si -> al
        cmp     al, byte [es:di]
        jz      LABEL_GO_ON
        jmp     LABEL_DIFFERENT         ; 只要发现不一样的字符就表明本 DirectoryEntry
                                        ; 不是我们要找的 LOADER.BIN
LABEL_GO_ON:
        inc     di
        jmp     LABEL_CMP_FILENAME      ; 继续循环

LABEL_DIFFERENT:
        and     di, 0FFE0h              ; else \. di &= E0 为了让它指向本条目开头
        add     di, 20h                 ;      |
        mov     si, LoaderFileName      ;      | di += 20h  下一个目录条目
        jmp     LABEL_SEARCH_FOR_LOADERBIN;    /

LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR:
        add     word [wSectorNo], 1
        jmp     LABEL_SEARCH_IN_ROOT_DIR_BEGIN

LABEL_NO_LOADERBIN:
        mov     dh, 2                   ; "No LOADER."
        call    DispStr                 ; 显示字符串
%ifdef  _BOOT_DEBUG_
        mov     ax, 4c00h               ; \.
        int     21h                     ; /  没有找到 LOADER.BIN, 回到 DOS
%else
        jmp     $                       ; 没有找到 LOADER.BIN, 死循环在这里
%endif

LABEL_FILENAME_FOUND:                   ; 找到 LOADER.BIN 后便来到这里继续
        jmp     $                       ; 代码暂时停在这里
```

这段代码看上去稍微有一点复杂，但逻辑是很清晰的，就是遍历根目录区所有的扇区，将每一个扇区加载入内存，然后从中寻找文件名为Loader.bin的条目，直到找到为止。找到的那一刻，es:di是指向条目中字母 N 后面的那个字符。代码中注释比较多，仔细看还是不困难的，在这里就不做过多解释了。其中的宏定义见代码4.6。

代码 4.6 宏定义（chapter4/b/boot.asm）

```
17 BaseOfLoader             equ  09000h ; LOADER.BIN 被加载到的位置 ---- 段地址
18 OffsetOfLoader           equ  0100h  ; LOADER.BIN 被加载到的位置 ---- 偏移地址
19 RootDirSectors           equ  14     ; 根目录占用空间
20 SectorNoOfRootDirectory  equ  19     ; Root Directory 的第一个扇区号
```

变量和字符串的定义请参见代码4.7。里面还包含其他一点变量和字符串的值，在以后的代码中会用到。

代码 4.7 变量和字符串定义（chapter4/b/boot.asm）

```
119 ;变量
120 wRootDirSizeForLoop  dw  RootDirSectors  ; Root Directory 占用的扇区数，
121                                           ; 在循环中会递减至零
122 wSectorNo            dw  0               ; 要读取的扇区号
123 bOdd                 db  0               ; 奇数还是偶数
124
125 ;字符串
126 LoaderFileName       db  "LOADER␣␣BIN", 0 ; LOADER.BIN 之文件名
127 ; 为简化代码，下面每个字符串的长度均为 MessageLength
128 MessageLength        equ 9
129 BootMessage:         db  "Booting␣␣␣" ; 9字节，不够则用空格补齐。序号 0
130 Message1             db  "Ready.␣␣␣" ; 9字节，不够则用空格补齐。序号 1
131 Message2             db  "No␣LOADER" ; 9字节，不够则用空格补齐。序号 2
```

需要注意的一点是，由于在读取过程中打印一些字符串，我们需要一个函数来做这项工作。为了节省代码长度，字符串的长度都设为 9 字节，不够则用空格补齐，这样就相当于一个二维数组，定位的时候通过数字就可以了，非常方便。显示字符串的函数名叫做DispStr（参见代码4.8），调用它的时候只要保证寄存器dh的值是字符串的序号就可以了。

代码 4.8 显示字符串（chapter4/b/boot.asm）

```
140 DispStr:
141         mov     ax, MessageLength
142         mul     dh
143         add     ax, BootMessage
144         mov     bp, ax                  ; `.
145         mov     ax, ds                  ;   | ES:BP = 串地址
146         mov     es, ax                  ; /
147         mov     cx, MessageLength       ; CX = 串长度
148         mov     ax, 01301h              ; AH = 13,  AL = 01h
149         mov     bx, 0007h               ; 页号为0(BH = 0) 黑底白字(BL = 07h)
150         mov     dl, 0
151         int     10h                     ; int 10h
152         ret
```

同时我们看到，在找到Loader.bin之后，程序死循环在那里，我们暂时先这样做，等到调试通过再继续进行，下面我们先编译一下生成一个Boot.bin。

```
▷ nasm boot.asm -o boot.bin
```

Boot.bin有了，但我们还没有软盘以及其中的 Loader。这很容易，先用 bximage 生成一个软盘映像，然后在 Linux 下这样做：

```
▷ nasm loader.asm -o loader.bin
▷ dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
▷ sudo mount -o loop a.img /mnt/floppy/
▷ sudo cp loader.bin /mnt/floppy/ -v
▷ sudo umount /mnt/floppy/
```

不过直接启动是看不到什么现象的，因为我们仅仅是找到Loader.bin就让程序停在了那里。那么，如何验证程序的正确性呢？我们不妨用 Bochs 调试一下，如下所示：

```
... ... ...
<bochs:1> b 0x7c00 ←在开始处设置断点
<bochs:2> c ←执行到断点
... ... ...
(0) Breakpoint 1, 0x00007c00 in ?? ()
Next at t=866205
(0) [0x00007c00] 0000:7c00 (unk. ctxt): jmp .+0x003c (0x00007c3e) ; eb3c
<bochs:3> n ←跳过 BPB
Next at t=866207
(0) [0x00007c3e] 0000:7c3e (unk. ctxt): mov ax, cs                  ; 8cc8
<bochs:4> u /45 ←反汇编
00007c3e: (                    ): mov ax, cs                        ; 8cc8
00007c40: (                    ): mov ds, ax                        ; 8ed8
... ... ...
00007cad: (                    ): mov dh, 0x02                      ; b602
00007caf: (                    ): call .+0x0030                     ; e83000
00007cb2: (                    ): jmp .+0xfffe                      ; ebfe
00007cb4: (                    ): jmp .+0xfffe                      ; ebfe
<bochs:5> b 0x7cb4 ←在 jmp $ 处设断点
<bochs:6> c ←执行到断点
(0) Breakpoint 2, 0x00007cb4 in ?? ()
Next at t=890512
(0) [0x00007cb4] 0000:7cb4 (unk. ctxt): jmp .+0xfffe (0x00007cb4) ; ebfe
<bochs:7> x /32xb es:di - 16 ←查看 es:di 前后的内存
[bochs]:
0x0009011b <bogus+   0>:  0x00  0xff  0xff  0xff  0xff  0x4c  0x4f  0x41
0x00090123 <bogus+   8>:  0x44  0x45  0x52  0x20  0x20  0x42  0x49  0x4e
0x0009012b <bogus+  16>:  0x20  0x00  0x00  0x8f  0x38  0x1b  0x39  0x1b
0x00090133 <bogus+  24>:  0x39  0x00  0x00  0x8f  0x38  0x1b  0x39  0x03
<bochs:8> x /13xcb es:di - 11 ←容易发现 es:di 前乃我们要找的文件名
[bochs]:
0x00090120 <bogus+       0>:  L    O    A    D    E    R
0x00090128 <bogus+       8>:  B    I    N         \0
<bochs:9> sreg ←查看 es
... ... ...
es:s=0x9000, dl=0x0000ffff, dh=0x00009309, valid=3
... ... ...
<bochs:10> r ←查看 di
CPU 0:
... ... ...
edi: 0x0000012b 299
... ... ...
```

在上面的过程中，反汇编之后对照代码4.5知道断点应该设在0x7cb4处，即第114行，程序执行到那里时，es:di为0x9000:0x12b，即内存0x9012b处，容易发现这里正是Loader.bin这个文件名的后面，di刚刚完成比较所有 11 个字符的使命。看到这里，我们知道我们应该是成功了。有一点不确定的原因是，我们的Loader.bin是拷入此软盘的第一个文件，所以稳妥的做法是在软盘中拷入更多文件进行测试。

等确信可以成功找到文件之后，我们就开始下面的工作——将Loader.bin加载入内存。

现在我们已经有了Loader.bin的起始扇区号，我们需要用这个扇区号来做两件事：一件是把起始扇区装入内存，另一件则是通过它找到 FAT 中的项，从而找到 Loader 占用的其余所有扇区。

在这里，我们把 Loader 装入内存的BaseOfLoader:OffsetOfLoader

处，你可能发现了，我们在代码4.5中根目录区也是装到这个位置，这算是资源的回收利用吧。因为我们装入根目录区仅仅是为了找到相应的目录条目而已，找到之后，根目录区对我们就没有用了。所以，现在我们尽管用 Loader 把它覆盖掉，没有关系。

装入一个扇区对我们来说已经是轻松的事了，可从 FAT 中找到一个项还多少有些麻烦。而且，如果 Loader 占用多个扇区的话，我们可能需要重复从 FAT 中找相应的项，所以，还是写一个函数来做这件事。函数的输入就是扇区号，输出则是其对应的 FAT 项的值（见代码4.9）。

代码 4.9 由扇区号求 FAT 项的值（chapter4/c/boot.asm）

```
22  SectorNoOfFAT1          equ     1       ; FAT1 的第一个扇区号 = BPB_RsvdSecCnt
    ...
264 GetFATEntry:
265         push    es
266         push    bx
267         push    ax
268         mov     ax, BaseOfLoader; ┐
269         sub     ax, 0100h       ; │ 在 BaseOfLoader 后面留出 4K 空间用于存放 FAT
270         mov     es, ax          ; ┘
271         pop     ax
272         mov     byte [bOdd], 0
273         mov     bx, 3
274         mul     bx                      ; dx:ax = ax * 3
275         mov     bx, 2
276         div     bx                      ; dx:ax / 2  ==>  ax <- 商, dx <- 余数
277         cmp     dx, 0
278         jz      LABEL_EVEN
279         mov     byte [bOdd], 1
280 LABEL_EVEN:;偶数
281         ; 现在 ax 中是 FATEntry 在 FAT 中的偏移量,下面来
282         ; 计算 FATEntry 在哪个扇区中(FAT占用不止一个扇区)
283         xor     dx, dx
284         mov     bx, [BPB_BytsPerSec]
285         div     bx ; dx:ax / BPB_BytsPerSec
286                    ;  ax <- 商 (FATEntry 所在的扇区相对于 FAT 的扇区号)
287                    ;  dx <- 余数 (FATEntry 在扇区内的偏移)
288         push    dx
289         mov     bx, 0 ; bx <- 0 于是, es:bx = (BaseOfLoader - 100):00
290         add     ax, SectorNoOfFAT1 ; 此句之后的 ax 就是 FATEntry 所在的扇区号
291         mov     cl, 2
292         call    ReadSector ; 读取 FATEntry 所在的扇区, 一次读两个, 避免在边界
293                            ; 发生错误, 因为一个 FATEntry 可能跨越两个扇区
294         pop     dx
295         add     bx, dx
296         mov     ax, [es:bx]
297         cmp     byte [bOdd], 1
298         jnz     LABEL_EVEN_2
299         shr     ax, 4
300 LABEL_EVEN_2:
301         and     ax, 0FFFh
302
303 LABEL_GET_FAT_ENRY_OK:
304
305         pop     bx
306         pop     es
307         ret
```

我们新增加了宏SectorNoOfFAT1，它与前面提到的RootDirSectors、SectorNoOfRootDirectory等宏一起，与 FAT12 有关的几个数字我们都定义成了宏，而不是在程序中进行计算。一方面，这是为缩小引导扇区代码考虑；

另一方面，这些数字一般情况下是不会变的，写代码计算它们其实是一种浪费。

取得 FAT 项的这段代码中唯一可能让人感到迷惑的地方在于要区别对待扇区号是奇数还是偶数，这在前文中已经提到了，请读者参考图4.2以及相应的文字。

在第108页中我们还提到，由于一个 FAT 项可能跨越两个扇区，所以在代码中一次总是读两个扇区，以免在边界发生错误。

好了，现在一切俱备了，让我们开始加载 Loader 吧（代码4.10）。

代码 4.10 加载 Loader（chapter4/c/boot.asm）

```
127 LABEL_FILENAME_FOUND:                       ; 找到 LOADER.BIN 后便来到这里继续
128         mov     ax, RootDirSectors
129         and     di, 0FFE0h                  ; di -> 当前条目的开始
130         add     di, 01Ah                    ; di -> 首 Sector
131         mov     cx, word [es:di]
132         push    cx                          ; 保存此 Sector 在 FAT 中的序号
133         add     cx, ax
134         add     cx, DeltaSectorNo           ; cl <- LOADER.BIN的起始扇区号(0-based)
135         mov     ax, BaseOfLoader
136         mov     es, ax                      ; es <- BaseOfLoader
137         mov     bx, OffsetOfLoader          ; bx <- OffsetOfLoader
138         mov     ax, cx                      ; ax <- Sector 号
139
140 LABEL_GOON_LOADING_FILE:
141         push    ax                          ; `.
142         push    bx                          ;  |
143         mov     ah, 0Eh                     ;  | 每读一个扇区就在 "Booting  " 后面 '
144         mov     al, '.'                     ;  | 打一个点，形成这样的效果:
145         mov     bl, 0Fh                     ;  | Booting ......
146         int     10h                         ;  |
147         pop     bx                          ;  |
148         pop     ax                          ; /
149
150         mov     cl, 1
151         call    ReadSector
152         pop     ax                          ; 取出此 Sector 在 FAT 中的序号
153         call    GetFATEntry
154         cmp     ax, 0FFFh
155         jz      LABEL_FILE_LOADED
156         push    ax                          ; 保存 Sector 在 FAT 中的序号
157         mov     dx, RootDirSectors
158         add     ax, dx
159         add     ax, DeltaSectorNo
160         add     bx, [BPB_BytsPerSec]
161         jmp     LABEL_GOON_LOADING_FILE
162 LABEL_FILE_LOADED:
```

在代码4.10中我们又看到一个新的宏DeltaSectorNo，那么它是做什么的呢？我们还是拿第4.1.1节中的例子来看，文件RIVER.TXT对应的目录条目中的开始簇号是 2，但我们显然不能根据这个数字 2 来读取扇区。实际上，开始簇号是 2 对应的是数据区的第一个扇区。所以，我们需要有一个方法来计算簇号为 X 代表从引导扇区开始算起是第几个扇区。

我们已经知道，根目录区占用RootDirSectors也即 14 个扇区，根目录区的开始扇区号是 19，于是我们要用“X+RootDirSectors+19-2”来算出“33”这个正确的扇区号。所以，我们又定义了一个宏DeltaSectorNo为 17（即 19−2）来帮助计算正确的扇区号：

```
DeltaSectorNo equ 17
```

4.1.5 向 Loader 交出控制权

上面的代码调试通过后，我们就已经成功地将 Loader 加载入内存，下面让我们来一个跳转，开始执行 Loader（代码4.11）。

代码 4.11 跳入 loader（chapter4/c/boot.asm）

```
168         jmp     BaseOfLoader:OffsetOfLoader     ; 这一句正式跳转到已加载到内
169                                                 ; 存中的 LOADER.BIN 的开始处，
170                                                 ; 开始执行 LOADER.BIN 的代码。
171                                                 ; Boot Sector 的使命到此结束
```

好了，下面用第112页的方法更新引导扇区和 Loader，再运行一下。

再执行一下Boot.com，我们来看一下 Loader 是不是已经在执行了，结果如图4.4所示。

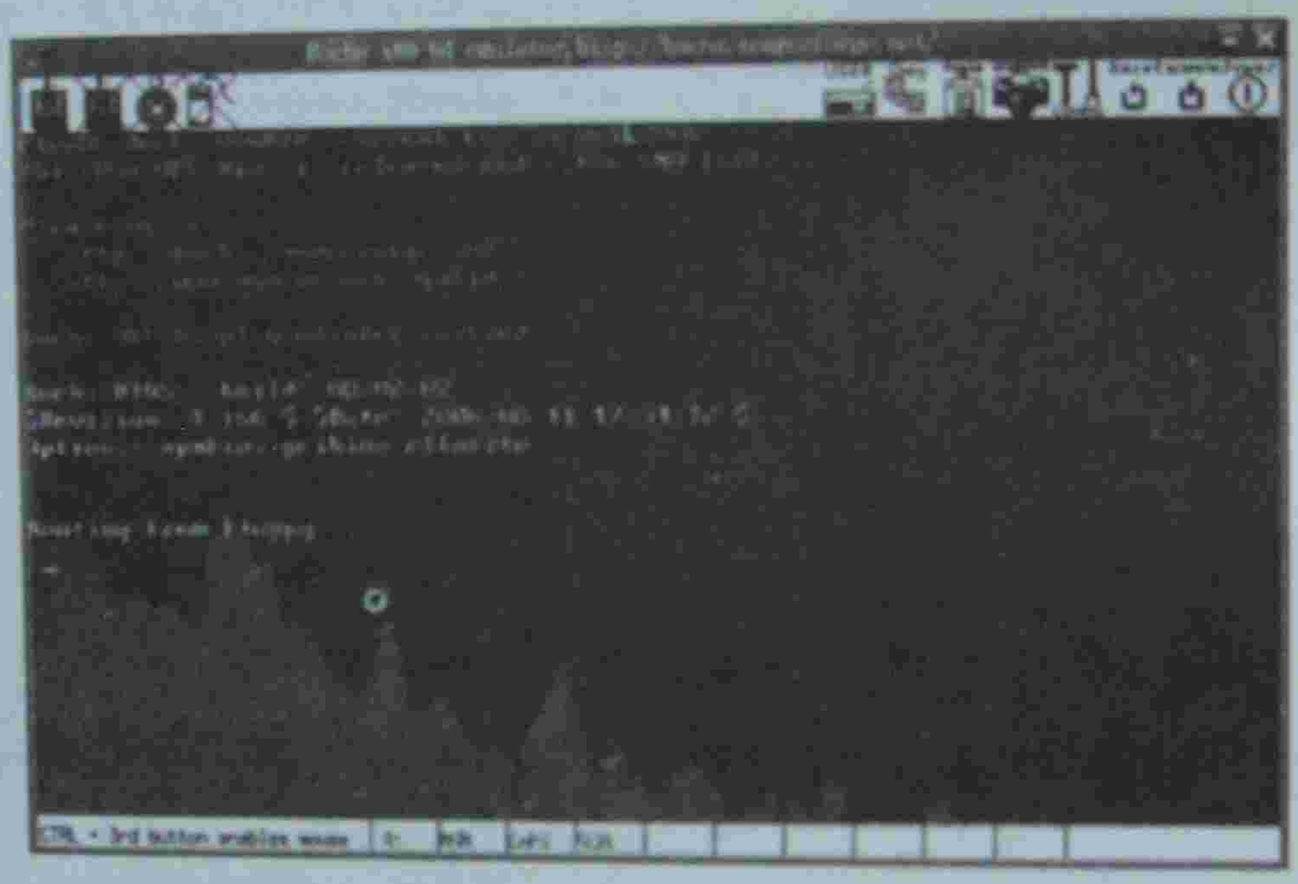


图 4.4 Loader 开始执行

果然，如我们所预料的那样，看到了字符“L”（由于没有清屏，所以你可能需要几秒钟来找到它）。另外，屏幕上的圆点表明我们读了一个扇区就把 Loader 加载完毕。

你一定感到很有趣，是的，而且这意味着我们的引导扇区编写可以告一段落了。

4.1.6 整理 boot.asm

最后，我们来对boot.asm进行整理，以便让执行的效果更好一点。而且，把它写进引导扇区（如果你之前在 DOS 下调试它的话别忘了编译前注释掉_BOOT_DEBUG_宏）。运行结果如图4.5所示。

代码4.12首先清屏，然后显示字符串“Booting”。这样，加载 Loader 时打印的圆点也会出现在这个字符串的后面。

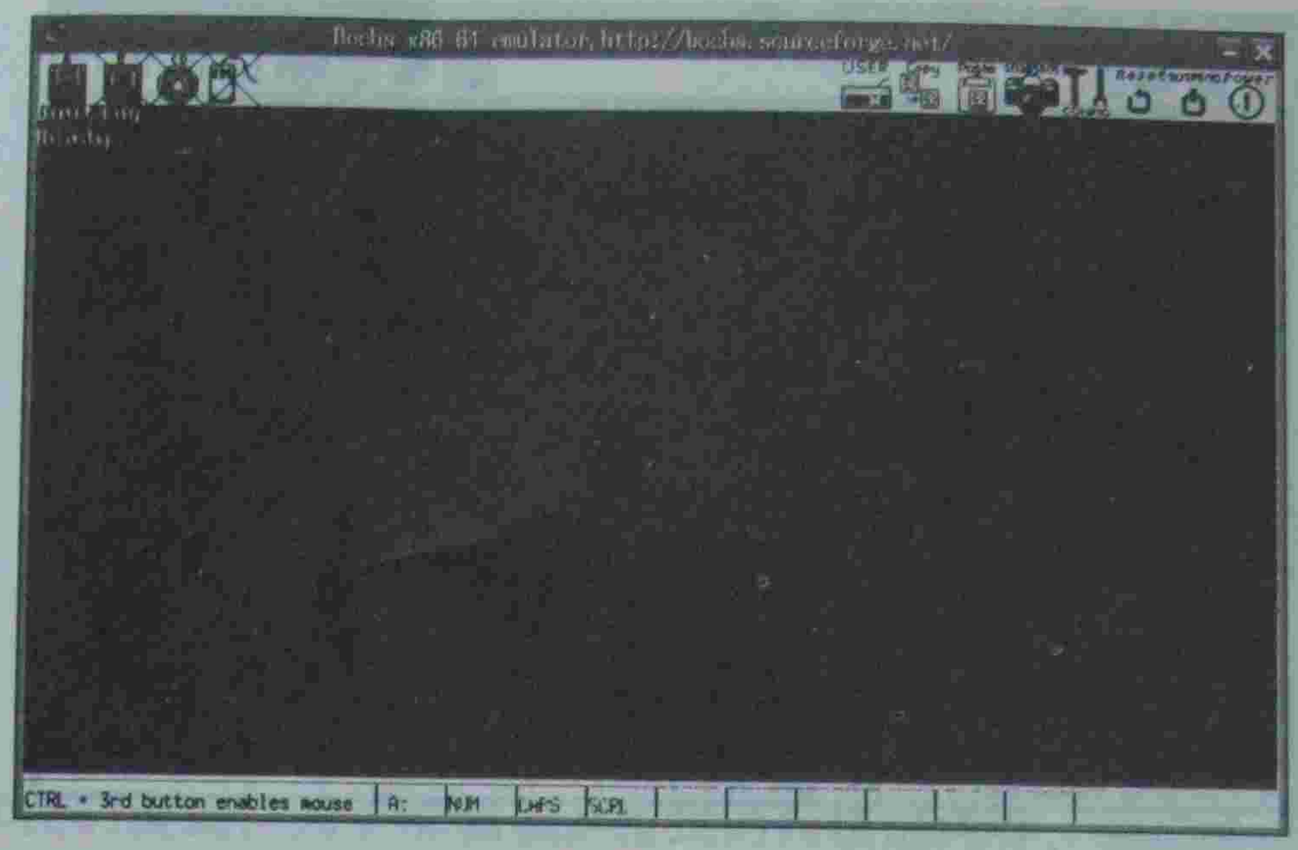


图 4.5 Loader 开始执行

代码 4.12 清屏并显示一个字符串（chapter4/c/boot.asm）

```
58          ; 清屏
59          mov     ax, 0600h               ; AH = 6,  AL = 0h
60          mov     bx, 0700h               ; 黑底白字(BL = 07h)
61          mov     cx, 0                   ; 左上角: (0, 0)
62          mov     dx, 0184fh              ; 右下角: (80, 50)
63          int     10h                     ; int 10h
64
65          mov     dh, 0                   ; "Booting  "
66          call    DispStr                 ; 显示字符串
```

代码4.13实现的是在加载完毕跳入 Loader 之前打印字符串“Ready.”。

代码 4.13 跳入 Loader 之前显示字符串（chapter4/c/boot.asm）

```
164         mov     dh, 1                   ; "Ready."
165         call    DispStr                 ; 显示字符串
```

Loader.bin本质上是个.COM文件，最大也不可能超过 64KB。但是，我们已经成功突破 512 字节限制，这个进步无疑是巨大的。

回想一下，我们做了如此多的工作，但代码长度仍然保持在 512 字节之内，是不是很精妙、很有趣呢？

4.2 保护模式下的“操作系统”

如果你了解 Linux 的话，一定知道它的引导扇区代码Boot.s比我们的代码简单，它直接把内核移动到目标内存。我们的代码之所以复杂一些，是因为我们想和 MSDOS 的磁盘格式兼容，以便调试的时候容易一些。虽然开始时我们多做了工作，但是从长远来看，我认为是值得的。

比如现在，我们就完全可以把第 3 章中的代码pmtest9.asm编译一下，将编译后的二进制命名为Loader.bin并复制到刚刚引导过的软盘中覆盖掉原来简陋的Loader.bin。你会发现程序马上可以执行，结果如图4.6所示。

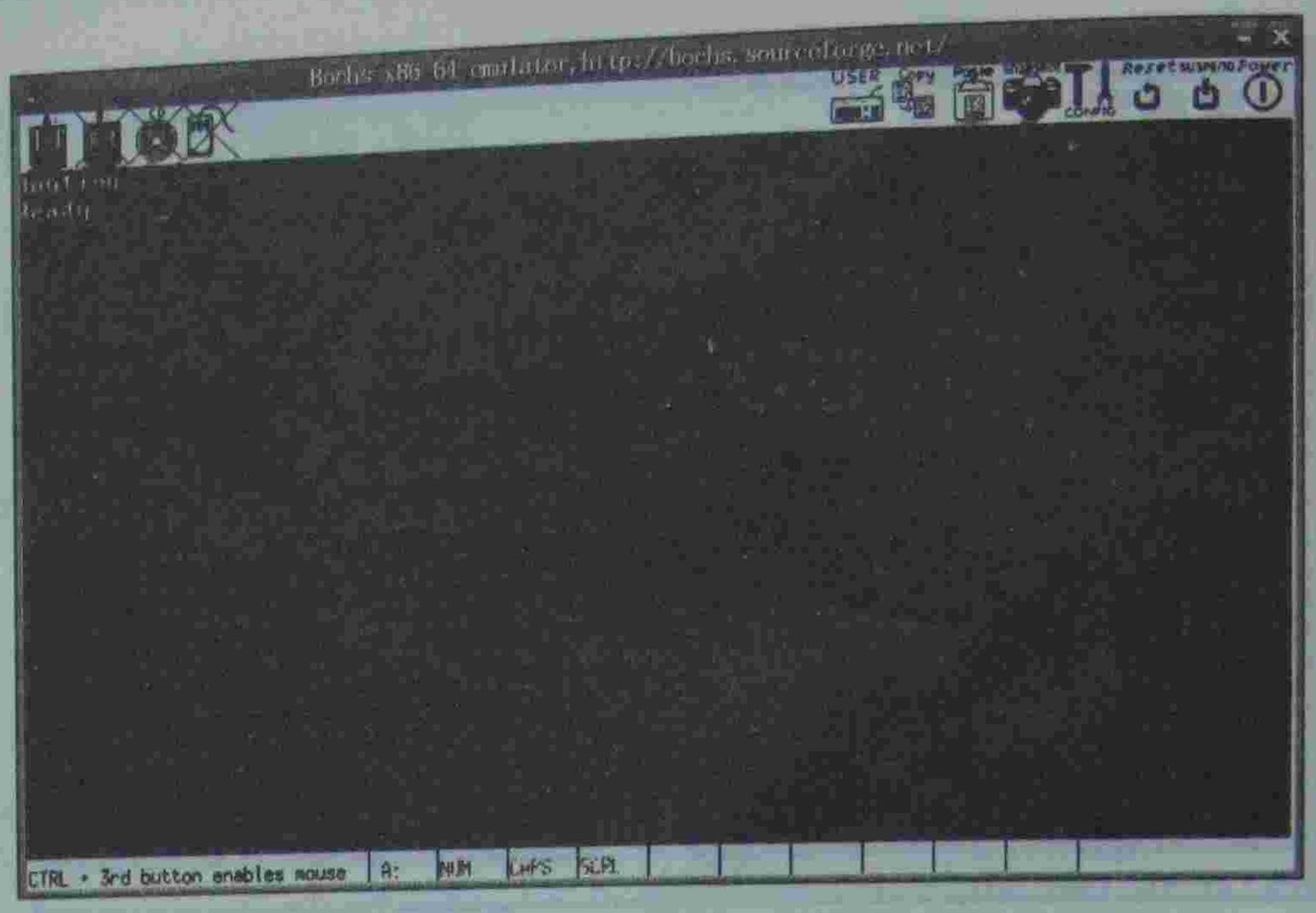


图 4.6 操作系统在保护模式下运行

是不是非常激动人心呢？其实，只要一个.COM文件中不含有 DOS 系统调用，我们就可以把它当做 Loader 来使用，也就是说，现在的引导扇区实际上已经很强大了。图4.6所显示的情况其实已经可以被看做是一个在保护模式下执行的“操作系统”了。

不过，你应该想到一个问题，现在的 Loader 毕竟仅仅是个 Loader，它不是操作系统内核，也不能当做操作系统内核。因为之前我们说过，我们希望自己的操作系统内核至少应该可以在 Linux 下用 GCC 编译链接，要不然，永远用汇编一点一点地写下去实在是太痛苦了。

那么，现在我们假设已经有了一个内核，Loader 肯定要加载它入内存，而且内核开始执行的时候肯定已经在保护模式下了，所以，Loader 要做的事情至少有两件：

- 加载内核入内存。
- 跳入保护模式。

这两件事恰好是我们擅长的。不过，我们用引导扇区加载进内存的 Loader 是个.COM文件，直接把它放进内存就够了，可是将来的内核是在 Linux 下编译链接出的 ELF 格式文件，直接放进内存肯定是不行的，没关系，我们下一章开始就会研究 ELF 的格式。

Premature optimization is the root of all evil.

—— *Donald E. Knuth*

5 内核雏形

上一章我们提到，为了加载 ELF 格式的内核进内存，我们必须研究一下这种格式。可是到目前为止，这种格式的文件我们还没见过呢，而且将来要编译链接内核，我们还不知道用什么命令和选项，看来要做的事情还不少。

那么，从哪一件开始做起呢？研究 ELF 需要一个样本，我们恰好可以通过写一个小的内核来做样本。现在我们就来写一个尽可能小但足以做加载试验的“内核”。

5.1 在 Linux 下用汇编写 Hello World

我们提到，完成从实模式到保护模式跳转这一任务的应该是 Loader，那么 Loader 应该走多远呢？只完成跳转，还是应该把 GDT、IDT、8259A 等内容准备完备？实际上，从逻辑上讲，Loader 不是操作系统的一部分，所以不应该越俎代庖。而且，你一定也希望早早结束 Loader 的工作进入正题，所以，我们还是要让 Loader 尽量简单，其余的工作留给内核来做。

之所以提到这个问题，是因为我们需要知道内核应该由哪些编程语言来完成。我们当然希望尽早摆脱汇编进入 C 的世界，但事实上却的确还有一些功能需要继续使用汇编来完成其中的部分代码，比如时钟中断处理、异常处理等。你将会发现，甚至于进程调度的一部分代码也是用汇编来完成的。

所以，在内核中我们仍然离不开汇编。下面就来体验一下在 Linux 下用汇编编程的感觉，请看代码5.1。

代码 5.1 chapter5/a/hello.asm

```
1  ; 编译链接方法
2  ; (ld 的 '-s' 选项意为 "strip all")
3  ;
4  ; $ nasm -f elf hello.asm -o hello.o
5  ; $ ld -s hello.o -o hello
6  ; $ ./hello
7  ; Hello, world!
8  ; $
9
10 [section .data] ; 数据在此
11
12 strHello      db      "Hello, world!", 0Ah
```

```
13  STRLEN          equ     $ - strHello
14
15  [section .text] ; 代码在此
16
17  global _start   ; 我们必须导出 _start 这个入口，以便让链接器识别
18
19  _start:
20          mov     edx, STRLEN
21          mov     ecx, strHello
22          mov     ebx, 1
23          mov     eax, 4          ; sys_write
24          int     0x80            ; 系统调用
25          mov     ebx, 0
26          mov     eax, 1          ; sys_exit
27          int     0x80            ; 系统调用
```

编译链接（每执行一步都用ls命令查看文件的增加情况）：

```
▷ ls
hello.asm
▷ nasm -f elf hello.asm -o hello.o
▷ ls
hello.asm  hello.o
▷ ld -s hello.o -o hello
▷ ls
hello  hello.asm  hello.o
```

NASM的选项“-f elf”指定了输出文件的格式为ELF格式。

链接选项“-s”意为strip，可以去掉符号表等内容，可起到对生成的可执行代码减肥之功效。

执行：

```
▷ ./hello
Hello, world!
```

成功！怎么样？很简单，不是吗？

我们回头看看代码5.1，程序中定义了两个节（Section），一个放数据，一个放代码。在代码中值得注意的一点是，入口点默认的是“_start”，我们不但要定义它，而且要通过global这个关键字将它导出，这样链接程序才能找到它。至于代码本身，你只需知道它们是两个系统调用（如果你只接触过Windows编程，那么可认为系统调用相当于Windows编程中的API），用来显示字符串并退出就够了。至于为什么这么做倒不用深究，因为在我们自己的OS中根本用不到Linux的系统调用。

5.2 再进一步，汇编和C同步使用

我们轻而易举地就得到了一个ELF格式的文件，它真的很小，只有四百多字节，比一个引导扇区还小。你一定在想，这倒真的很简单。可是同时你心里可能在嘀咕，能行吗？这么小的一个东西，能够胜任我们样本的工作吗？

虽然有可能在杞人忧天，但在对 Linux 下的汇编编程没有足够熟悉之前，这样想也是有道理的。那么，我们就把程序再扩充一下。

我们迟早都要用 C 语言来写程序，并将它与汇编写的程序链接在一起，不如现在就开始尝试一下。这不但是一项非常有趣的工作，而且毫无疑问，它有着非同寻常的意义。

在即将完成的这个小例子中，源代码包含两个文件：foo.asm和bar.c。程序入口_start在foo.asm中，一开始程序将会调用bar.c中的函数choose()，choose()将会比较传入的两个参数，根据比较结果的不同打印出不同的字符串。打印字符串的工作是由foo.asm中的函数myprint()来完成的。整个过程如图5.1所示。

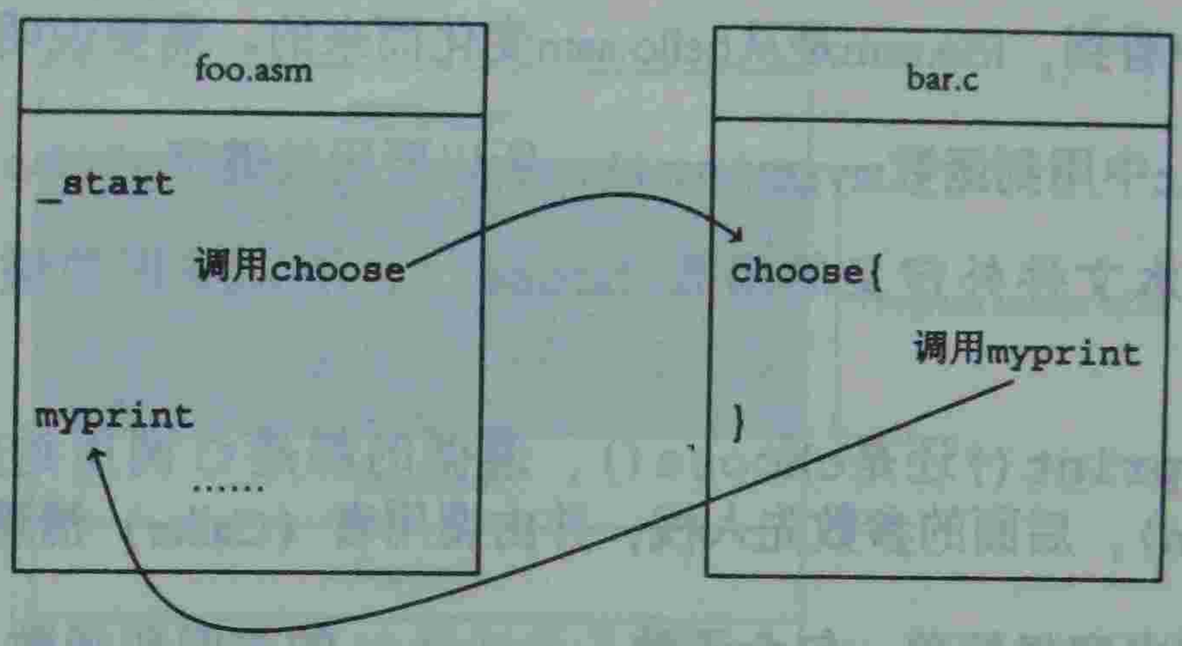


图 5.1 foo.asm 和 bar.c 之间的调用关系

之所以这样安排，是因为它包含了汇编代码和 C 代码之间相互的调用，掌握这一点很重要，在我们今后的工作中会经常用到。

代码 5.2 chapter5/b/foo.asm

```
1  ; 编译链接方法
2  ; (ld 的 '-s' 选项意为 "strip all")
3  ;
4  ; $ nasm -f elf foo.asm -o foo.o
5  ; $ gcc -c bar.c -o bar.o
6  ; $ ld -s hello.o bar.o -o foobar
7  ; $ ./foobar
8  ; the 2nd one
9  ; $
10
11 extern choose    ; int choose(int a, int b);
12
13 [section .data] ; 数据在此
14
15 num1st          dd      3
16 num2nd          dd      4
17
18 [section .text] ; 代码在此
19
20 global _start    ; 我们必须导出 _start 这个入口，以便让链接器识别
21 global myprint   ; 导出这个函数为了让 bar.c 使用
22
23 _start:
24         push    dword [num2nd]  ; '.
25         push    dword [num1st]  ; |
```

```
        call    choose          ;  | choose(num1st, num2nd);
        add     esp, 8          ; /

        mov     ebx, 0
        mov     eax, 1          ; sys_exit
        int     0x80            ; 系统调用

; void myprint(char* msg, int len)
myprint:
        mov     edx, [esp + 8]  ; len
        mov     ecx, [esp + 4]  ; msg
        mov     ebx, 1
        mov     eax, 4          ; sys_write
        int     0x80            ; 系统调用
        ret
```

从代码5.2中看到，foo.asm是从hello.asm变化而来的。需要说明的有三点：

1. 由于在bar.c中用到函数myprint()，所以要用关键字global将其导出。
2. 由于用到本文件外定义的函数choose()，所以要用关键字extern声明。
3. 不管是myprint()还是choose()，遵循的都是C调用约定（C Calling Convention），后面的参数先入栈，并由调用者（Caller）清理堆栈。

文件bar.c的内容很简单，包含函数myprint()的声明和函数choose()的主体（见代码5.3）。

代码 5.3 chapter5/b/bar.c

```
void myprint(char* msg, int len);

int choose(int a, int b)
{
        if(a >= b){
                myprint("the␣1st␣one\n", 13);
        }
        else{
                myprint("the␣2nd␣one\n", 13);
        }

        return 0;
}
```

编译链接和执行的过程：

```
▷ ls
bar.c  foo.asm
▷ nasm -f elf -o foo.o foo.asm
▷ gcc -c -o bar.o bar.c
▷ ld -s -o foobar foo.o bar.o
▷ ls
bar.c  bar.o  foo.asm  foobar  foo.o
▷ ./foobar
the 2nd one
```

读者自己可以通过改变num1st和num2nd的值来试验一下其他可能的输出。

怎么样，是不是很有趣？而且，我猜你一定已经发现了其中的诀窍，不外乎就是关键字global和extern。是的，有了这两个关键字，就可以方便地在汇编和C代码之间自由来去。学会了这一点，你对将来内核的开发工作是不是成竹在胸了呢？

而且，将来我们的内核即便会比 foobar 大很多，也不会有本质的区别，所以，就让我们放心地使用 foobar 作为研究 ELF 的样本吧。

5.3 ELF（Executable and Linkable Format）

ELF 文件的结构如图5.2所示。

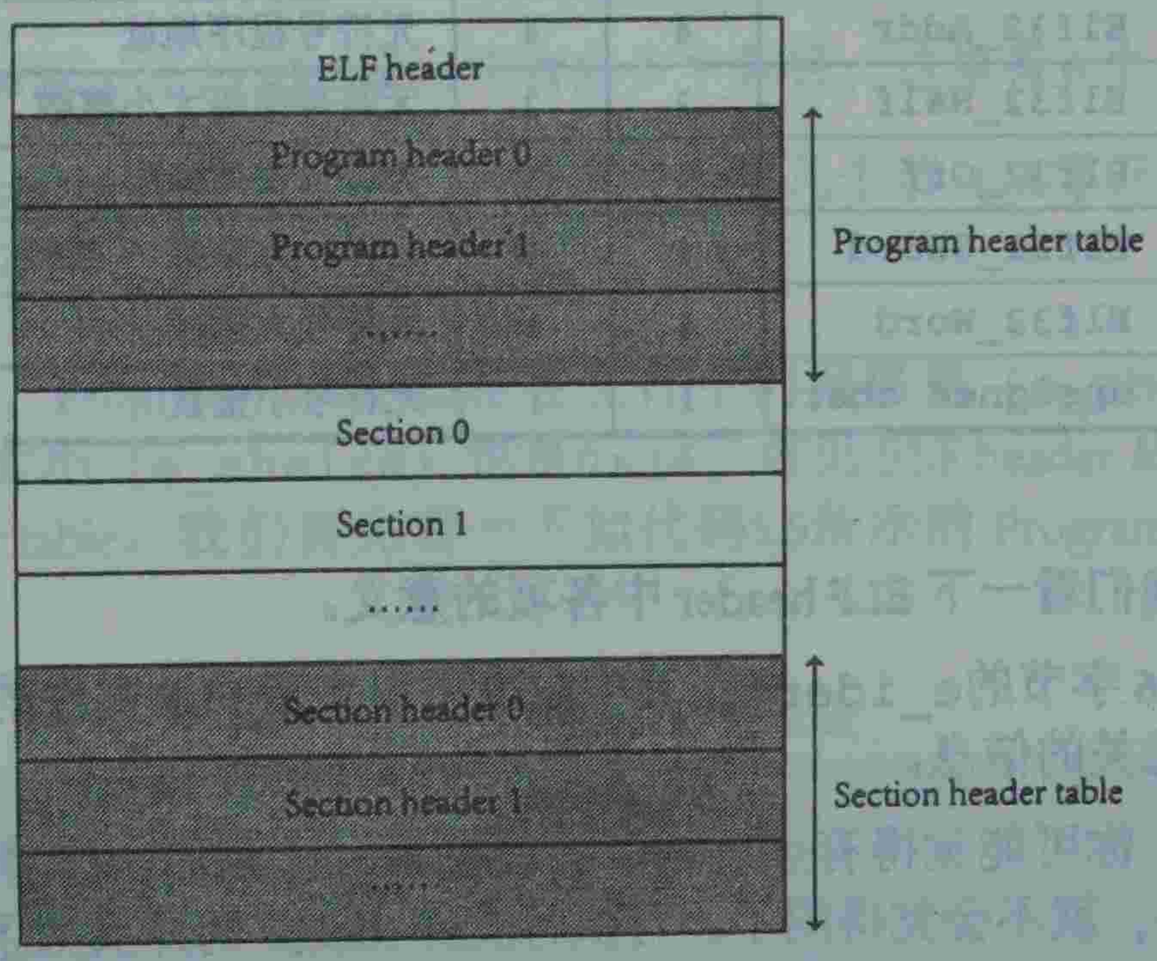


图 5.2 ELF 文件概览

可以看出，ELF 文件由 4 部分组成，分别是 ELF 头（ELF header）、程序头表（Program header table）、节（Sections）和节头表（Section header table）。实际上，一个文件中不一定包含全部这些内容，而且它们的位置也未必如图5.2所示这样安排，只有 ELF 头的位置是固定的，其余各部分的位置、大小等信息由 ELF 头中的各项值来决定。

在下文中，程序头、节头等名词都使用英文原文 Program header、Section header 来表示，你最终会发现，这样写恰恰会帮助你阅读和理解。

ELF header 的格式如代码5.4所示。其中各类型的说明见表5.1。

代码 5.4 ELF header

```
1  #define EI_NIDENT          16
2  typedef struct{
3        unsigned char        e_ident[EI_NIDENT];
4        Elf32_Half           e_type;
5        Elf32_Half           e_machine;
6        Elf32_word           e_version;
7        Elf32_Addr           e_entry;
8        Elf32_Off            e_phoff;
9        Elf32_Off            e_shoff;
```

```
    Elf32_Word          e_flags;
    Elf32_Half          e_ehsize;
    Elf32_Half          e_phentsize;
    Elf32_Half          e_phnum;
    Elf32_Half          e_shentsize;
    Elf32_Half          e_shnum;
    Elf32_Half          e_shstrndx;
}Elf32_Ehdr;
```

由于 ELF 文件力求支持从 8 位到 32 位不同架构的处理器，所以才定义了表5.1中这些数据类型，从而让文件格式与机器无关。

表 5.1 ELF header

名称	大小	对齐	用途
Elf32_Addr	4	4	无符号程序地址
Elf32_Half	2	2	无符号中等大小整数
Elf32_Off	4	4	无符号文件偏移
Elf32_Sword	4	4	有符号大整数
Elf32_Word	4	4	无符号大整数
unsigned char	1	1	无符号小整数

下面就让我们看一下 ELF header 中各项的意义。

最开头是 16 字节的e_ident，其中包含用以表示 ELF 文件的字符，以及其他一些与机器无关的信息。

说到这里，你可能觉得有点乏味了，没关系，让我们一边看着刚才生成的 foobar 一边讲解，就不会觉得闷了。可执行文件 foobar 的开头如下所示：

```
▷ xxd -u -a -g 1 -c 16 -l 80 foobar
0000000: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00  .ELF............
0000010: 02 00 03 00 01 00 00 00 A0 80 04 08 34 00 00 00  ............4...
0000020: C0 01 00 00 00 00 00 00 34 00 20 00 03 00 28 00  ........4. ...(.
0000030: 06 00 05 00 01 00 00 00 00 00 00 00 00 80 04 08  ................
0000040: 00 80 04 08 31 01 00 00 31 01 00 00 05 00 00 00  ....1...1.......
```

开头的 4 字节是固定不变的，第 1 个字节值为0x7F，紧跟着就是 ELF 三个字符，这 4 字节表明这个文件是个 ELF 文件。

下面，我们就以 foobar 为例说明 ELF header 中各项的含义：

- **e_type** 它标识的是该文件的类型，可能的取值在这里就不一一列出了。文件 foobar 的e_type是 2，表明它是一个可执行文件（Executable File）。
- **e_machine** foobar 中此项的值为 3，表明运行该程序需要的体系结构为 Intel 80386。
- **e_version** 这个成员确定文件的版本。
- **e_entry** 程序的入口地址。文件 foobar 的入口地址为0x80480A0。
- **e_phoff** Program header table 在文件中的偏移量（以字节计数）。这里的值是0x34。

- **e_shoff** Section header table 在文件中的偏移量（以字节计数）。这里的值是0x1C0。
- **e_flags** 对 IA32 而言，此项为 0。
- **e_ehsize** ELF header 大小（以字节计数）。这里值为0x34。
- **e_phentsize** Program header table 中每一个条目（一个 Program header）的大小。这里值为0x20。
- **e_phnum** Program header table 中有多少个条目，这里有 3 个。
- **e_shentsize** Section header table 中每一个条目（一个 Section header）的大小，这里值为0x28。
- **e_shnum** Section header table 中有多少个条目，这里有 6 个。
- **e_shstrndx** 包含节名称的字符串表是第几个节（从零开始数）。这里值为 5，表示第 5 个节包含节名称。

我们看到，Program header table 在文件中的偏移量（e_phoff）为0x34，而 ELF header 大小（e_ehsize）也是0x34，可见 ELF header 后面紧接着就是 Program header table。我们首先看一下如代码5.5所示的 Program header 数据结构。

代码 5.5 Program header

```
typedef struct{
      Elf32_Word                p_type;
      Elf32_Off                 p_offset;
      Elf32_Addr                p_vaddr;
      Elf32_Addr                p_paddr;
      Elf32_Word                p_filesz;
      Elf32_Word                p_memsz;
      Elf32_Word                p_flags;
      Elf32_Word                p_align;
}Elf32_Phdr;
```

实际上 Program header 描述的是系统准备程序运行所需的一个段（Segment）或其他信息。这样说有点抽象，让我们来看看 foobar 的情况。程序头表中共有三项（e_phnum=3），偏移分别是0x34～0x53、0x54～0x73和0x74～0x93。

```
▷ xxd -u -a -g 1 -c 16 -s 0x34 -l 0x60 foobar
0000034: 01 00 00 00 00 00 00 00 00 80 04 08 00 80 04 08  ................
0000044: 31 01 00 00 31 01 00 00 05 00 00 00 00 10 00 00  1...1...........
0000054: 01 00 00 00 34 01 00 00 34 91 04 08 34 91 04 08  ....4...4...4...
0000064: 08 00 00 00 08 00 00 00 06 00 00 00 00 10 00 00  ................
0000074: 51 E5 74 64 00 00 00 00 00 00 00 00 00 00 00 00  Q.td............
0000084: 00 00 00 00 00 00 00 00 07 00 00 00 04 00 00 00  ................
```

其中各项的意义如下：

- **p_type** 当前 Program header 所描述的段的类型。
- **p_offset** 段的第一个字节在文件中的偏移。
- **p_vaddr** 段的第一个字节在内存中的虚拟地址。

- **p_paddr** 在物理地址定位相关的系统中，此项是为物理地址保留。
- **p_filesz** 段在文件中的长度。
- **p_memsz** 段在内存中的长度。
- **p_flags** 与段相关的标志。
- **p_align** 根据此项值来确定段在文件以及内存中如何对齐。

读到这里，读者一定已经明白了，Program header 描述的是一个段在文件中的位置、大小以及它被放进内存后所在的位置和大小。如果我们想把一个文件加载进内存的话，需要的正是这些信息。

在 foobar 中共有三个 Program header，其取值如表5.2所示。

表 5.2 Program header

名称	Program header 0	Program header 1	Program header 2
p_type	0x1	0x1	0x6474E551
p_offset	0x0	0x134	0
p_vaddr	0x8048000	0x8049134	0
p_paddr	0x8048000	0x8049134	0
p_filesz	0x131	0x8	0
p_memsz	0x131	0x8	0
p_flags	0x5	0x6	0x7
p_align	0x1000	0x1000	0x4

根据这些信息，我们很容易知道 foobar 在加载进内存之后的情形，正如图5.3所描绘的那样。

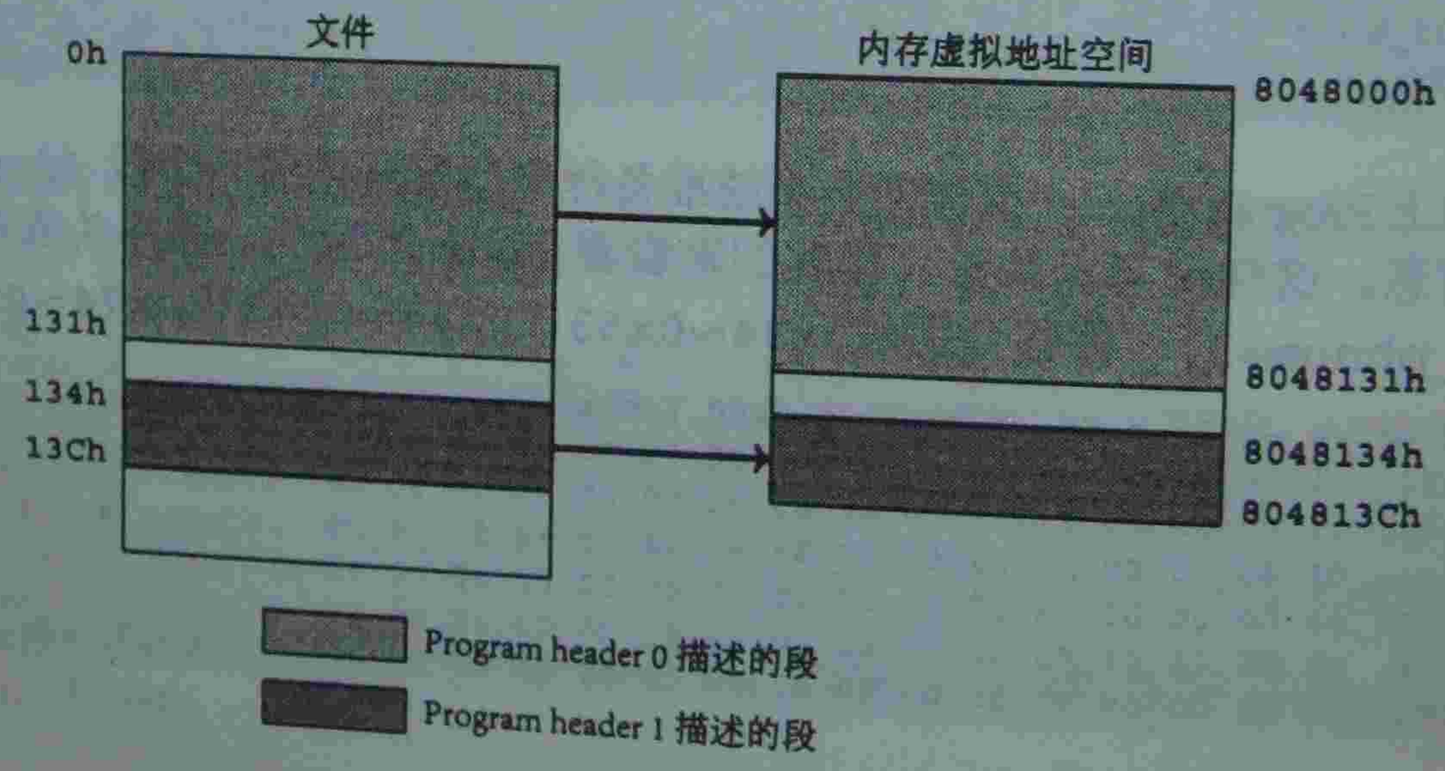


图 5.3 foobar 加载情况

至此，我们已经可以结束对 ELF 文件的研究了。什么，结束了？你一定感到很意外，说不定你正摩拳擦掌准备研究 Section header table 以及其他内容呢。

实际上，我们的确应该先把 ELF 文件的格式研究一下。可是你知道，我实在是懒惰而且性急，我已经迫不及待地要跟你一起扩充我们的 Loader 了。

5.4 从 Loader 到内核

研究过 ELF，你是否还记得 Loader 需要做的工作有哪些？Loader 要做两项工作：

- 加载内核到内存。
- 跳入保护模式。

我们先来做第一项，把内核加载到内存。

5.4.1 用 Loader 加载 ELF

加载内核到内存这一步和引导扇区的工作非常相似，只是处理内核时我们需要根据 Program header table 中的值把内核中相应的段放到正确的位置。我们可以这样来做，首先像引导扇区处理 Loader 那样把内核放入内存，只要内核进入了内存，如何处理它便是一件容易的事情了，我们可以在保护模式下挪动它的位置。

依旧是寻找文件、定位文件以及读入内存，实际上，单就把内核读入内存这一部分，除了文件名和读入的内存地址变了，其余其实都是一样的。之所以没有把它写成一个函数分别在boot.asm和loader.asm中调用，是因为函数在调用时堆栈操作会占用更多的空间，在引导扇区中，每一个字节都是珍贵的。

不过，一些常量的定义却可以在boot.asm和loader.asm之间共享。我们不妨把与 FAT12 文件有关的内容写进一个单独的文件（文件名为fat12hdr.inc，见代码5.6），在两个文件的开头相应的位置分别包含进去。

代码 5.6 chapter5/c/fat12hdr.inc

```
; FAT12 磁盘的头
;-------------------------------------------------------------------------
BS_OEMName      DB 'ForrestY'   ; OEM String, 必须 8 个字节

BPB_BytsPerSec  DW 512          ; 每扇区字节数
BPB_SecPerClus  DB 1            ; 每簇多少扇区
BPB_RsvdSecCnt  DW 1            ; Boot 记录占用多少扇区
BPB_NumFATs     DB 2            ; 共有多少 FAT 表
BPB_RootEntCnt  DW 224          ; 根目录文件数最大值
BPB_TotSec16    DW 2880         ; 逻辑扇区总数
BPB_Media       DB 0xF0         ; 媒体描述符
BPB_FATSz16     DW 9            ; 每FAT扇区数
BPB_SecPerTrk   DW 18           ; 每磁道扇区数
BPB_NumHeads    DW 2            ; 磁头数(面数)
BPB_HiddSec     DD 0            ; 隐藏扇区数
BPB_TotSec32    DD 0            ; 如果 wTotalSectorCount 是 0 由这个值记录扇区数

BS_DrvNum       DB 0            ; 中断 13 的驱动器号
BS_Reserved1    DB 0            ; 未使用
BS_BootSig      DB 29h          ; 扩展引导标记 (29h)
BS_VolID        DD 0            ; 卷序列号
```

```
BS_VolLab           DB 'OrangeS0.02'; 卷标, 必须 11 个字节
BS_FileSysType      DB 'FAT12   '   ; 文件系统类型, 必须 8个字节
;-------------------------------------------------------------------------------


;-------------------------------------------------------------------------------
; 基于 FAT12 头的一些常量定义, 如果头信息改变, 下面的常量可能也要做相应改变
;-------------------------------------------------------------------------------
; BPB_FATSz16
FATSz                   equ     9

; 根目录占用空间:
; RootDirSectors = ((BPB_RootEntCnt*32)+(BPB_BytsPerSec-1))/BPB_BytsPerSec
; 但如果按照此公式代码过长, 故定义此宏
RootDirSectors          equ     14

; Root Directory 的第一个扇区号         = BPB_RsvdSecCnt + (BPB_NumFATs * FATSz)
SectorNoOfRootDirectory equ     19

; FAT1 的第一个扇区号 = BPB_RsvdSecCnt
SectorNoOfFAT1          equ     1

; DeltaSectorNo = BPB_RsvdSecCnt + (BPB_NumFATs * FATSz) - 2
; 文件的开始Sector号 = DirEntry中的开始Sector号 + 根目录占用Sector数目
;                     + DeltaSectorNo
DeltaSectorNo           equ     17
```

这样一来，boot.asm开头部分的代码就应该变成代码5.7所示的样子。

代码 5.7 使用 fat12hdr.inc（节自 chapter5/c/boot.asm）

```
        jmp short LABEL_START           ; Start to boot.
        nop                             ; 这个 nop 不可少

; 下面是 FAT12 磁盘的头, 之所以包含它是因为下面用到了磁盘的一些信息
%include        "fat12hdr.inc"

LABEL_START:
```

下面我们就来修改loader.asm，先让它把内核放进内存（见代码5.8）。

代码 5.8 使用 fat12hdr.inc（节自 chapter5/c/loader.asm）

```
org  0100h

BaseOfStack             equ     0100h

BaseOfKernelFile        equ     08000h ; KERNEL.BIN 被加载到的位置 ----  段地址
OffsetOfKernelFile      equ         0h ; KERNEL.BIN 被加载到的位置 ----  偏移地址

        jmp     LABEL_START                     ; Start

; 下面是 FAT12 磁盘的头, 之所以包含它是因为下面用到了磁盘的一些信息
%include        "fat12hdr.inc"

LABEL_START:
        mov     ax, cs                  ; <--- 从这里开始 ***************
        mov     ds, ax
        mov     es, ax
        mov     ss, ax
        mov     sp, BaseOfStack

        mov     dh, 0
        call    DispStr                 ; "Loading  "
                                        ; 显示字符串
```

```
        ; 下面在 A 盘的根目录寻找 KERNEL.BIN
        mov     word [wSectorNo], SectorNoOfRootDirectory
        xor     ah, ah  ; ┓
        xor     dl, dl  ; ┃ 软驱复位
        int     13h     ; ┛
LABEL_SEARCH_IN_ROOT_DIR_BEGIN:
        cmp     word [wRootDirSizeForLoop], 0   ; ┓
        jz      LABEL_NO_KERNELBIN              ; ┃ 判断根目录区是不是已经读完,
        dec     word [wRootDirSizeForLoop]      ; ┛ 读完表示没有找到 KERNEL.BIN
        mov     ax, BaseOfKernelFile
        mov     es, ax                  ; es <- BaseOfKernelFile
        mov     bx, OffsetOfKernelFile  ; bx <- OffsetOfKernelFile
        mov     ax, [wSectorNo]         ; ax <- Root Directory 中的某 Sector 号
        mov     cl, 1
        call    ReadSector

        mov     si, KernelFileName      ; ds:si -> "KERNEL  BIN"
        mov     di, OffsetOfKernelFile
        cld
        mov     dx, 10h
LABEL_SEARCH_FOR_KERNELBIN:
        cmp     dx, 0                                   ; ┓
        jz      LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR;     ┃ 循环次数控制, 如果已经读完
        dec     dx                                      ; ┛ 了一个 Sector, 就跳到下一个
        mov     cx, 11
LABEL_CMP_FILENAME:
        cmp     cx, 0                           ; ┓
        jz      LABEL_FILENAME_FOUND            ; ┃ 循环次数控制, 如果比较了 11 个字符都
        dec     cx                              ; ┛ 相等, 表示找到
        lodsb                                   ; ds:si -> al
        cmp     al, byte [es:di]                ; if al == es:di
        jz      LABEL_GO_ON
        jmp     LABEL_DIFFERENT
LABEL_GO_ON:
        inc     di
        jmp     LABEL_CMP_FILENAME      ;       继续循环

LABEL_DIFFERENT:
        and     di, 0FFE0h              ; else'. 让 di 是 20h 的倍数
        add     di, 20h                 ;       ┃
        mov     si, KernelFileName      ;       ┃ di += 20h  下一个目录条目
        jmp     LABEL_SEARCH_FOR_KERNELBIN;     ┛

LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR:
        add     word [wSectorNo], 1
        jmp     LABEL_SEARCH_IN_ROOT_DIR_BEGIN

LABEL_NO_KERNELBIN:
        mov     dh, 2                   ; "No KERNEL."
        call    DispStr                 ; 显示字符串
%ifdef  _LOADER_DEBUG_
        mov     ax, 4c00h               ; ┓
        int     21h                     ; ┛ 没有找到 KERNEL.BIN, 回到 DOS
%else
        jmp     $                       ; 没有找到 KERNEL.BIN, 死循环在这里
%endif

LABEL_FILENAME_FOUND:                   ; 找到 KERNEL.BIN 后便来到这里继续
        mov     ax, RootDirSectors
        and     di, 0FFF0h              ; di -> 当前条目的开始

        push    eax
        mov     eax, [es : di + 01Ch]           ; ┓
        mov     dword [dwKernelSize], eax       ; ┛ 保存 KERNEL.BIN 文件大小
        pop     eax
```

```
91          add     di, 01Ah                ; di -> 首 Sector
92          mov     cx, word [es:di]
93          push    cx                      ; 保存此 Sector 在 FAT 中的序号
94          add     cx, ax
95          add     cx, DeltaSectorNo       ; cl <- LOADER.BIN 的起始扇区号(0-based)
96          mov     ax, BaseOfKernelFile
97          mov     es, ax                  ; es <- BaseOfKernelFile
98          mov     bx, OffsetOfKernelFile  ; bx <- OffsetOfKernelFile
99          mov     ax, cx                  ; ax <- Sector 号
100
101 LABEL_GOON_LOADING_FILE:
102         push    ax                      ; ┌
103         push    bx                      ; │
104         mov     ah, 0Eh                 ; │ 每读一个扇区就在 "Loading " 后面
105         mov     al, '.'                 ; │ 打一个点，形成这样的效果:
106         mov     bl, 0Fh                 ; │ Loading ......
107         int     10h                     ; │
108         pop     bx                      ; │
109         pop     ax                      ; └
110
111         mov     cl, 1
112         call    ReadSector
113         pop     ax                      ; 取出此 Sector 在 FAT 中的序号
114         call    GetFATEntry
115         cmp     ax, 0FFFh
116         jz      LABEL_FILE_LOADED
117         push    ax                      ; 保存 Sector 在 FAT 中的序号
118         mov     dx, RootDirSectors
119         add     ax, dx
120         add     ax, DeltaSectorNo
121         add     bx, [BPB_BytsPerSec]
122         jmp     LABEL_GOON_LOADING_FILE
123 LABEL_FILE_LOADED:
124
125         call    KillMotor               ; 关闭软驱马达
126
127         mov     dh, 1                   ; "Ready."
128         call    DispStr                 ; 显示字符串
129
130         jmp     $
...
268 KillMotor:
269         push    dx
270         mov     dx, 03F2h
271         mov     al, 0
272         out     dx, al
273         pop     dx
274         ret
```

这段代码虽然有点长，但仔细读一下就会发现，它和boot.asm其实是差不多的，其中用到的函数如DispStr、ReadSector以及GetFATEntry和boot.asm中是完全一样的，这里就略过了。代码用到的一个新函数是KillMotor，用来关闭软驱马达，不然软驱的灯会一直亮着。

加载内核的代码写好了，可如今我们还没有内核呢，如果现在运行的话，将会出现图5.4所示的情况，“NO KERNEL”字样会被显示出来。

没有内核不要紧，我们马上写一个最简单的，文件名为kernel.asm，我们今后的内核就在它的基础上进行扩充，代码实现的功能照例是显示一个字符（见代码5.9）。

代码 5.9 还算不上内核的内核雏形（chapter5/c/kernel.asm）

图 5.4 Loader 运行情况（无内核时）

```
; $ nasm -f elf kernel.asm -o kernel.o
; $ ld -s kernel.o -o kernel.bin    # '-s' 选项意为 "strip all"

[section .text] ; 代码在此

global _start   ; 导出 _start

_start: ; 跳到这里来的时候，我们假设 gs 指向显存
        mov     ah, 0Fh                         ; 0000: 黑底    1111: 白字
        mov     al, 'K'
        mov     [gs:((80 * 1 + 39) * 2)], ax    ; 屏幕第 1 行，第 39 列
        jmp     $
```

显示字符时涉及内存操作，所以用到 GDT，我们假设在 Loader 中段寄存器gs已经指向显存的开始。

好了，现在“内核”也已经有了，我们来编译它并将其写入软盘映像：

```
▷ nasm -f elf -o kernel.o kernel.asm
▷ ld -s -o kernel.bin kernel.o
▷ sudo mount -o loop a.img /mnt/floppy/
▷ sudo cp kernel.bin /mnt/floppy/ -v
▷ sudo umount /mnt/floppy/
```

再运行一遍，就不再是“No KERNEL”了，结果如图5.5所示。

我们看到，Loading 后面出现一个圆点，说明 Loader 读了一个扇区。不过，由于目前我们除了把内核加载到内存之外没有做其他任何工作，所以除了能看到“Ready.”字样之外，并没有其他现象出现。

5.4.2 跳入保护模式

现在，内核已经被我们加载进内存了，该是跳入保护模式的时候了。

首先是 GDT 以及对应的选择子，我们只定义三个描述符，分别是一个 0~4GB 的可执行段、一个 0~4GB 的可读写段和一个指向显存开始地址的段（见

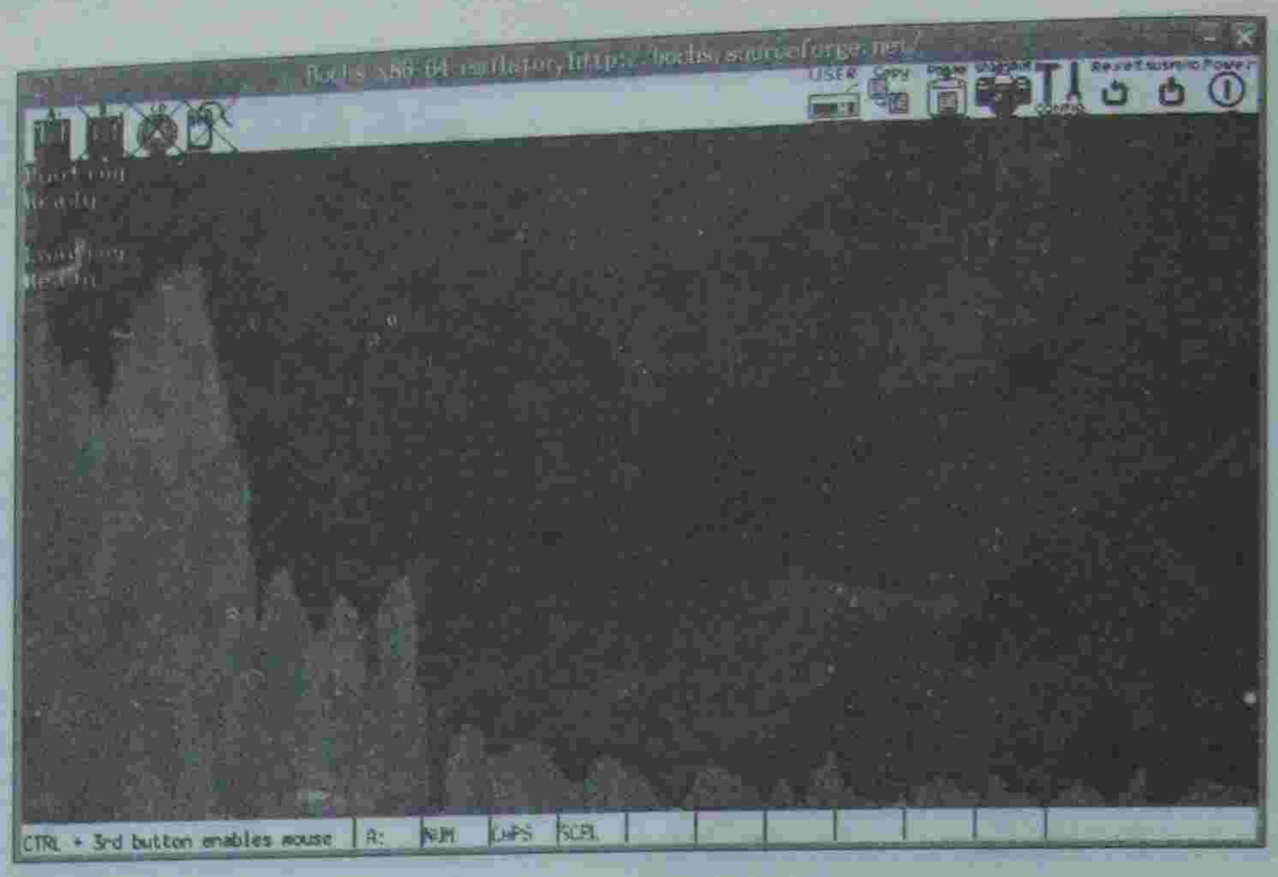


图 5.5 Loader 运行情况（有内核时）

代码5.10)。

代码 5.10 Loader 的 GDT（节自 chapter5/d/loader.asm）

```
%include        "pm.inc"

; GDT
;                                   段基址       段界限, 属性
LABEL_GDT:          Descriptor 0,                0, 0                        ; 空描述符
LABEL_DESC_FLAT_C:  Descriptor 0,          0fffffh, DA_CR|DA_32|DA_LIMIT_4K ;0-4G
LABEL_DESC_FLAT_RW: Descriptor 0,          0fffffh, DA_DRW|DA_32|DA_LIMIT_4K;0-4G
LABEL_DESC_VIDEO:   Descriptor 0B8000h, 0ffffh, DA_DRW|DA_DPL3 ; 显存首地址

GdtLen          equ     $ - LABEL_GDT
GdtPtr          dw      GdtLen - 1                                          ; 段界限
                dd      BaseOfLoaderPhyAddr + LABEL_GDT                     ; 基地址

; GDT 选择子
SelectorFlatC           equ     LABEL_DESC_FLAT_C       - LABEL_GDT
SelectorFlatRW          equ     LABEL_DESC_FLAT_RW      - LABEL_GDT
SelectorVideo           equ     LABEL_DESC_VIDEO        - LABEL_GDT + SA_RPL3
```

在第 3 章我们学习保护模式时，大部分描述符的段基址都是运行时计算后填入相应位置的，因为那时我们的程序是由 BIOS 或者 DOS 加载的，我们不知道段地址，于是也就不知道程序运行时在内存中的位置。如今，Loader 是由我们自己加载的，段地址已经被确定为 BaseOfLoader（见代码4.6），所以在 Loader 中出现的标号（变量）的物理地址可以用下面的公式来表示：

$$\text{标号（变量）的物理地址} = BaseOfLoader \times 10h + \text{标号（变量）的偏移}$$

不过，这样一来，BaseOfLoader就同时在boot.asm和loader.asm两个文件中使用，我们也把它以及相应的声明放在同一个文件load.inc中（见代码5.11）。

代码 5.11 一些宏定义（chapter5/d/load.inc）

```
BaseOfLoader            equ  09000h ; LOADER.BIN 被加载到的位置 ----  段地址
OffsetOfLoader          equ   0100h ; LOADER.BIN 被加载到的位置 ----  偏移地址

BaseOfLoaderPhyAddr equ BaseOfLoader*10h ; LOADER.BIN 被加载到的位置 ---- 物理地址
```

```
5
6  BaseOfKernelFile     equ  08000h ; KERNEL.BIN 被加载到的位置 ----  段地址
7  OffsetOfKernelFile   equ     0h ; KERNEL.BIN 被加载到的位置 ---- 偏移地址
```

同时，在boot.asm和loader.asm中分别用一句%include "load.inc"将其包含。

在代码5.11中可以看到，我们定义了一个宏BaseOfLoaderPhyAddr用以代替BaseOfLoader×10h，它在代码5.10中被用到一次，用来计算 GDT 的基址。

我们即将进入保护模式，仍然像过去一样，进入之后只是打印一个字符，并不做太多工作（代码5.12）。

代码 5.12 Loader 的 32 位代码段（节自 chapter5/d/loader.asm）

```
340 [SECTION .s32]
341
342 ALIGN   32
343
344 [BITS   32]
345
346 LABEL_PM_START:
347         mov     ax, SelectorVideo
348         mov     gs, ax
    ...
364         mov     ah, 0Fh                         ; 0000: 黑底    1111: 白字
365         mov     al, 'P'
366         mov     [gs:((80 * 0 + 39) * 2)], ax    ; 屏幕第 0 行, 第 39 列
367         jmp     $
```

进入保护模式的代码是再熟悉不过了，我们重复过许多次（代码5.13）。

代码 5.13 Loader 进入保护模式（节自 chapter5/d/loader.asm）

```
160 LABEL_FILE_LOADED:
161
162         call    KillMotor               ; 关闭软驱马达
163
164         mov     dh, 1                   ; "Ready."
165         call    DispStrRealMode         ; 显示字符串
166
167         ; 下面准备跳入保护模式
168
169         ; 加载 GDTR
170         lgdt    [GdtPtr]
171
172         ; 关中断
173         cli
174
175         ; 打开地址线A20
176         in      al, 92h
177         or      al, 00000010b
178         out     92h, al
179
180         ; 准备切换到保护模式
181         mov     eax, cr0
182         or      eax, 1
183         mov     cr0, eax
184
185         ; 真正进入保护模式
186         jmp     dword SelectorFlatC:(BaseOfLoaderPhyAddr+LABEL_PM_START)
```

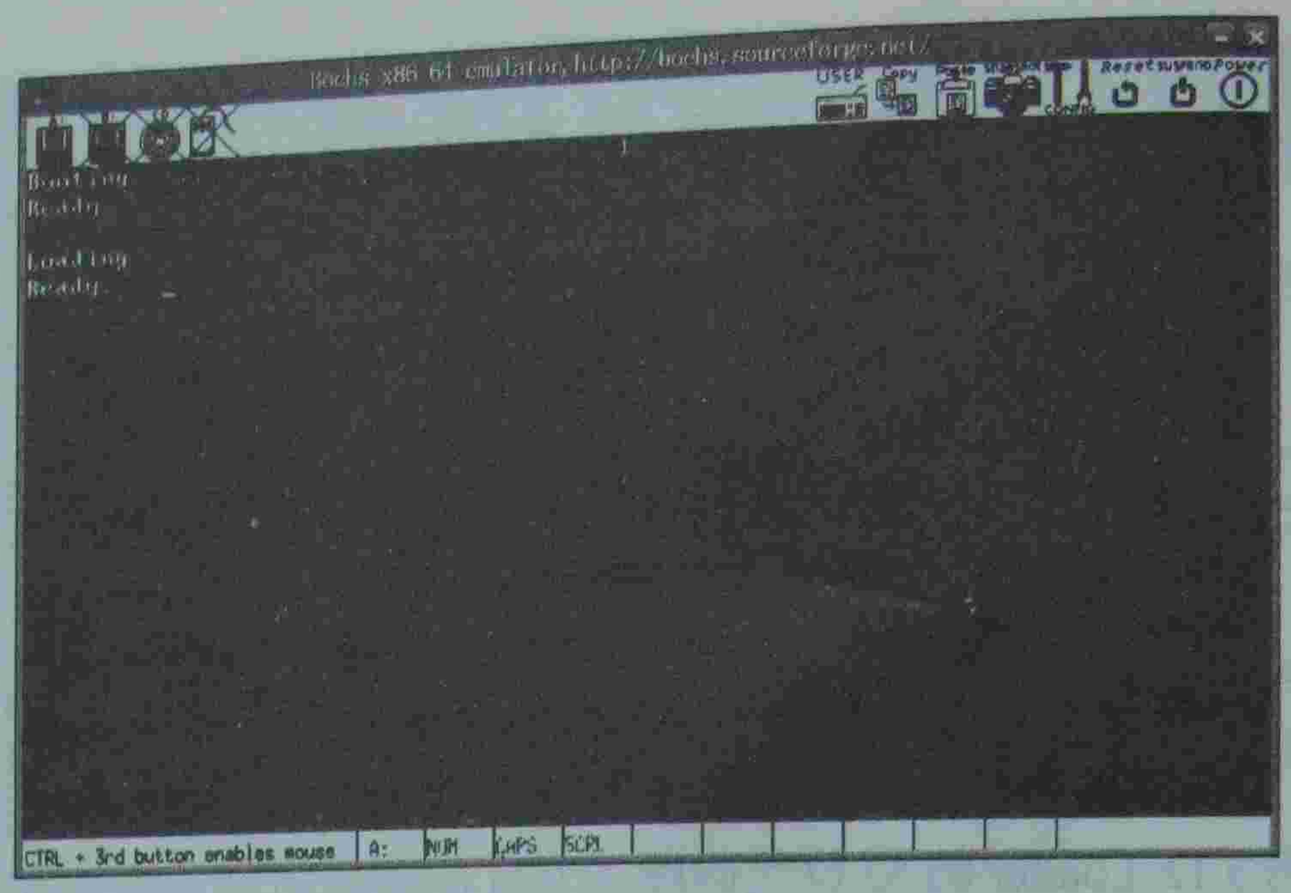


图 5.6 Loader 进入保护模式

运行，结果如图5.6所示。

看到字母“P”，我们已成功进入保护模式，下面可以从容地进行其他各项工作了。首先初始化各个寄存器的值（代码5.14）。

代码 5.14 初始化各个寄存器（节自 chapter5/d/loader.asm）

```
350         mov     ax, SelectorFlatRW
351         mov     ds, ax
352         mov     es, ax
353         mov     fs, ax
354         mov     ss, ax
355         mov     esp, TopOfStack
```

其中，TopOfStack的定义如代码5.15所示。

代码 5.15 堆栈定义（节自 chapter5/d/loader.asm）

```
512 StackSpace:     times   1024    db      0
513 TopOfStack      equ     BaseOfLoaderPhyAddr + $ ; 栈顶
```

目前，在保护模式下我们并不打算做太多工作，所以 1KB 的堆栈就足够了。等到我们进入内核时，可以重新设置堆栈。

下面，我们打开分页机制，打开之前还是应该先知道可使用内存的情况。因此，我们在 Loader 的开头再增加一些内容，如代码5.16所示。

代码 5.16 得到内存信息（节自 chapter5/d/loader.asm）

```
44          ; 得到内存数
45          mov     ebx, 0
46          mov     di, _MemChkBuf          ; ebx = 后续值，开始时需为 0
47  .MemChkLoop:                            ; es:di 指向一个地址范围描述符结构(ARDS)
48
49          mov     eax, 0E820h             ; eax = 0000E820h
50          mov     ecx, 20                 ; ecx = 地址范围描述符结构的大小
51          mov     edx, 0534D4150h         ; edx = 'SMAP'
52          int     15h                     ; int 15h
53          jc      .MemChkFail
54          add     di, 20
            inc     dword [_dwMCRNumber]    ; dwMCRNumber = ARDS 的个数
```

```
55          cmp     ebx, 0
56          jne     .MemChkLoop
57          jmp     .MemChkOK
58  .MemChkFail:
59          mov     dword [_dwMCRNumber], 0
60  .MemChkOK:
```

我们在第 3 章代码pmtest7.asm和pmtest8.asm中不但获得了内存信息，而且把它打印了出来，这里，我们也添加打印内存信息的函数（代码5.17）。

代码 5.17 显示内存信息（节自 chapter5/d/loader.asm）

```
374 DispMemInfo:
375         push    esi
376         push    edi
377         push    ecx
378
379         mov     esi, MemChkBuf
380         mov     ecx, [dwMCRNumber];for(int i=0;i<[MCRNumber];i++)//每次得到一个ARDS
381 .loop:                          ;{
382         mov     edx, 5          ;   for(int j=0;j<5;j++)//每次得到一个ARDS中的成员
383         mov     edi, ARDStruct  ;   {//依次显示:BaseAddrLow,BaseAddrHigh,LengthLow
384 .1:                             ;                   LengthHigh,Type
385         push    dword [esi]     ;
386         call    DispInt         ;     DispInt(MemChkBuf[j*4]); // 显示一个成员
387         pop     eax             ;
388         stosd                   ;     ARDStruct[j*4] = MemChkBuf[j*4];
389         add     esi, 4          ;
390         dec     edx             ;
391         cmp     edx, 0          ;
392         jnz     .1              ;   }
393         call    DispReturn      ;   printf("\n");
394         cmp     dword [dwType], 1 ;  if(Type == AddressRangeMemory)
395         jne     .2              ;   {
396         mov     eax, [dwBaseAddrLow];
397         add     eax, [dwLengthLow];
398         cmp     eax, [dwMemSize]  ;     if(BaseAddrLow + LengthLow > MemSize)
399         jb      .2              ;
400         mov     [dwMemSize], eax  ;     MemSize = BaseAddrLow + LengthLow;
401 .2:                             ;   }
402         loop    .loop           ;}
403                                 ;
404         call    DispReturn      ;printf("\n");
405         push    szRAMSize       ;
406         call    DispStr         ;printf("RAM size:");
407         add     esp, 4          ;
408                                 ;
409         push    dword [dwMemSize] ;
410         call    DispInt         ;DispInt(MemSize);
411         add     esp, 4          ;
412
413         pop     ecx
414         pop     edi
415         pop     esi
416         ret
```

这里用到的DispInt、DispStr、DispReturn等函数直接从第 3 章的代码中拿过来用就可以了，当时我们用单独的文件lib.inc保存这些代码，直接把文件复制过来，将其包含就可以了。只是要注意，一定要在 32 位代码段中包含它。

不过，这时DispStr被重复定义了，因为我们本来已经有一个DispStr了，现在我们把原来的DispStr改成DispStrRealMode，这样就不会冲突

了。

得到内存信息之后，就可以添加启动分页的代码了。这段代码也可以从第3章复制而来，稍做修改便可使用（见代码5.18）。

代码 5.18 启动分页的函数 SetupPaging（节自 chapter5/d/loader.asm）

```
SetupPaging:
        ; 根据内存大小计算应初始化多少PDE以及多少页表
        xor     edx, edx
        mov     eax, [dwMemSize]
        mov     ebx, 400000h    ; 400000h = 4M = 4096 * 1024, 一个页表对应的内存大小
        div     ebx
        mov     ecx, eax        ; 此时 ecx 为页表的个数，也即 PDE 应该的个数
        test    edx, edx
        jz      .no_remainder
        inc     ecx             ; 如果余数不为 0 就需增加一个页表
.no_remainder:
        push    ecx             ; 暂存页表个数

        ; 为简化处理, 所有线性地址对应相等的物理地址. 并且不考虑内存空洞.

        ; 首先初始化页目录
        mov     ax, SelectorFlatRW
        mov     es, ax
        mov     edi, PageDirBase        ; 此段首地址为 PageDirBase
        xor     eax, eax
        mov     eax, PageTblBase | PG_P  | PG_USU | PG_RWW
.1:
        stosd
        add     eax, 4096               ; 为了简化, 所有页表在内存中是连续的
        loop    .1

        ; 再初始化所有页表
        pop     eax                     ; 页表个数
        mov     ebx, 1024               ; 每个页表 1024 个 PTE
        mul     ebx
        mov     ecx, eax                ; PTE个数 = 页表个数 * 1024
        mov     edi, PageTblBase        ; 此段首地址为 PageTblBase
        xor     eax, eax
        mov     eax, PG_P  | PG_USU | PG_RWW
.2:
        stosd
        add     eax, 4096               ; 每一页指向 4K 的空间
        loop    .2

        mov     eax, PageDirBase
        mov     cr3, eax
        mov     eax, cr0
        or      eax, 80000000h
        mov     cr0, eax
        jmp     short .3
.3:
        nop

        ret
```

页目录和页表的定义见代码5.19。

代码 5.19 页目录和页表的位置（节自 chapter5/d/loader.asm）

```
PageDirBase     equ     100000h ; 页目录开始地址: 1M
PageTblBase     equ     101000h ; 页表开始地址:   1M + 4K
```

在以上代码中使用的字符串和变量的定义见代码5.20。

代码 5.20 符号定义（节自 chapter5/d/loader.asm）

```
477 LABEL_DATA:
478 ; 实模式下使用这些符号
479 ; 字符串
480 _szMemChkTitle: db "BaseAddrL BaseAddrH LengthLow LengthHigh    Type", 0Ah, 0
481 _szRAMSize:     db "RAM size:", 0
482 _szReturn:      db 0Ah, 0
483 ;; 变量
484 _dwMCRNumber:   dd 0      ; Memory Check Result
485 _dwDispPos:     dd (80 * 6 + 0) * 2     ; 屏幕第 6 行, 第 0 列
486 _dwMemSize:     dd 0
487 _ARDStruct:     ; Address Range Descriptor Structure
488   _dwBaseAddrLow:                 dd      0
489   _dwBaseAddrHigh:                dd      0
490   _dwLengthLow:                   dd      0
491   _dwLengthHigh:                  dd      0
492   _dwType:                        dd      0
493 _MemChkBuf:     times   256     db      0
494 ;
495 ;; 保护模式下使用这些符号
496 szMemChkTitle           equ     BaseOfLoaderPhyAddr + _szMemChkTitle
497 szRAMSize               equ     BaseOfLoaderPhyAddr + _szRAMSize
498 szReturn                equ     BaseOfLoaderPhyAddr + _szReturn
499 dwDispPos               equ     BaseOfLoaderPhyAddr + _dwDispPos
500 dwMemSize               equ     BaseOfLoaderPhyAddr + _dwMemSize
501 dwMCRNumber             equ     BaseOfLoaderPhyAddr + _dwMCRNumber
502 ARDStruct               equ     BaseOfLoaderPhyAddr + _ARDStruct
503       dwBaseAddrLow     equ     BaseOfLoaderPhyAddr + _dwBaseAddrLow
504       dwBaseAddrHigh    equ     BaseOfLoaderPhyAddr + _dwBaseAddrHigh
505       dwLengthLow       equ     BaseOfLoaderPhyAddr + _dwLengthLow
506       dwLengthHigh      equ     BaseOfLoaderPhyAddr + _dwLengthHigh
507       dwType            equ     BaseOfLoaderPhyAddr + _dwType
508 MemChkBuf              equ     BaseOfLoaderPhyAddr + _MemChkBuf
```

可以看到，在保护模式下使用的地址都被加上了 Loader 的基地址。

显示内存信息和启动分页的函数都准备好了，现在我们来调用它们（代码5.21）。

代码 5.21 显示内存信息并启动分页（节自 chapter5/d/loader.asm）

```
357     push    szMemChkTitle
358     call    DispStr
359     add     esp, 4
360
361     call    DispMemInfo
362     call    SetupPaging
```

这里除了有调用DispMemInfo和SetupPaging的两句代码，还显示了内存信息的一个表头。

运行，结果如图5.7所示。

对于这个图你是不是感到很熟悉呢？是的，我们在第 3 章中见过它，现在它已经成为我们操作系统的一部分，而不再是一个试验了。你是否感到很有成就感呢？

5.4.3 重新放置内核

我们的loader.asm正在飞速膨胀，不要紧，马上就要冲刺了，终点就在前方，因为我们马上就要整理内存中的内核并把控制权交给它了。

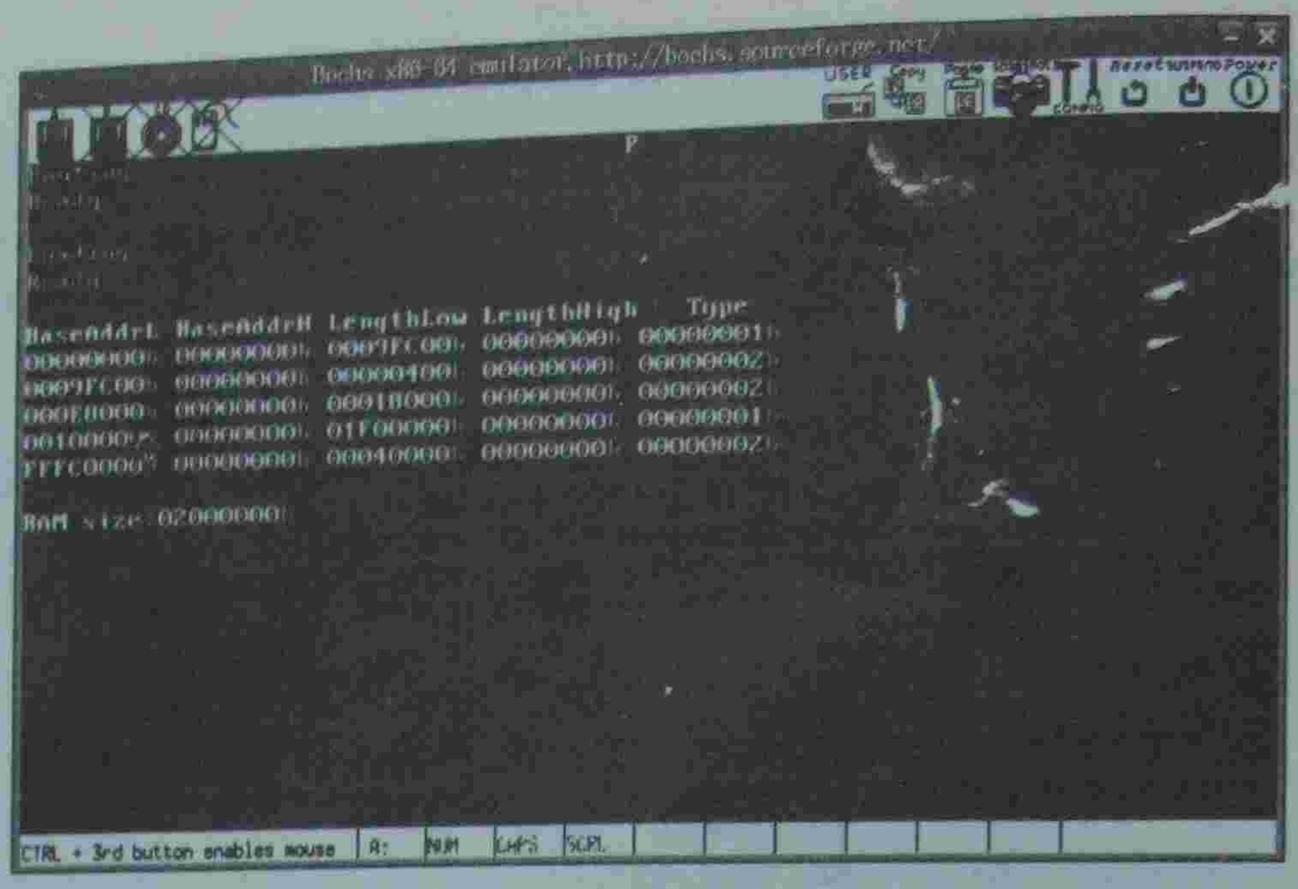


图 5.7 列出内存情况

如果你已经忘记 ELF 文件的格式，建议你马上复习一下。实际上，我们要做的工作是根据内核的 Program header table 的信息进行类似下面这个 C 语言语句的内存复制：

```
memcpy(p_vaddr, BaseOfLoaderPhyAddr + p_offset, p_filesz);
```

复制可能不止一次，如果 Program header 有*n*个，复制就进行*n*次。

我们说过，每一个 Program header 都描述一个段。语句中的`p_offset`为段在文件中的偏移，`p_filesz`为段在文件中的长度，`p_vaddr`为段在内存中的虚拟地址。

说到这里，你可能想起一件事情，就是由 ld 生成的可执行文件中`p_vaddr`的值总是一个类似于`0x8048XXX`的值，至少我们的例子中是一个这样的值。可是我们启动分页机制时地址都是对等映射的，内存地址`0x8048XXX`已经处在 128MB 内存以外（128MB 的十六进制表示是`0x8000000`），如果计算机的内存小于 128MB 的话，这个地址显然已经超出了内存大小。

即便计算机有足够大的内存，显然，我们也不能让编译器来决定内核加载到什么地方。我们得让它受控制，解决它有两个办法，一是通过修改页表让`0x8048XXX`映射到较低的地址，另一种方法就是通过修改 ld 的选项让它生成的可执行代码中`p_vaddr`的值变小。

显然，第二种方法更加简单易行，下面我们就把编译链接时的命令行改为：

```
▷ nasm -f elf -o kernel.o kernel.asm
▷ ld -s -Ttext 0x30400 -o kernel.bin kernel.o
```

程序的入口地址就变成`0x30400`了，ELF header 等信息会位于`0x30400`之前。此时的 ELF header 和 Program header table 的情况如表5.3和表5.4所示。

根据表5.3和表5.4我们知道，我们应该这样放置内核：

```
memcpy(30000h, 90000h + 0, 40Dh);
```

表 5.3 kernel.bin 的 ELF header

项目	值	说明
e_ident	…	
e_type	2H	可执行文件
e_machine	3H	80386
e_version	1H	
e_entry	30400H	入口地址
e_phoff	34H	Program header table 在文件中的偏移量
e_shoff	448H	Section header table 在文件中的偏移量
e_flags	0H	
e_ehsize	34H	ELF header 大小
e_phentsize	20H	每一个 Program header 大小20H字节
e_phnum	1H	Program header table 中只有 1 个条目
e_shentsize	28H	每一个 Section header 大小28H字节
e_shnum	4H	Section header table 中有 4 个条目
e_shstrndx	3H	包含节名称的字符串表是第 3 个节

表 5.4 kernel.bin 的 Program header

项目	值	说明
p_type	1H	PT_LOAD
p_offset	0H	段的第一个字节在文件中的偏移
p_vaddr	30000H	段的第一个字节在内存中的虚拟地址
p_paddr	30000H	
p_filesz	40DH	段在文件中的长度
p_memsz	40DH	段在内存中的长度
p_flags	5H	
p_align	1000H	

也就是说，我们应该把文件从开头开始40Dh字节的内容放到内存30000h处。由于程序的入口在30400h处，所以从这里就可以看出，实际上代码只有0Dh+1个字节。我们来看一下Kernel.bin的内容：

```
▷ xxd -u -a -g 1 -c 16 kernel.bin
0000000: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00  .ELF............
0000010: 02 00 03 00 01 00 00 00 00 04 03 00 34 00 00 00  ............4...
0000020: 48 04 00 00 00 00 00 00 34 00 20 00 01 00 28 00  H.......4. ...(.
0000030: 04 00 03 00 01 00 00 00 00 00 00 00 00 00 03 00  ................
0000040: 00 00 03 00 0D 04 00 00 0D 04 00 00 05 00 00 00  ................
0000050: 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
0000400: B4 0F B0 4B 65 66 A3 EE 00 00 00 EB FE 00 54 68  ...Kef........Th
0000410: 65 20 4E 65 74 77 69 64 65 20 41 73 73 65 6D 62  e Netwide Assemb
0000420: 6C 65 72 20 30 2E 39 38 2E 33 38 00 00 2E 73 68  ler 0.98.38...sh
0000430: 73 74 72 74 61 62 00 2E 74 65 78 74 00 2E 63 6F  strtab..text..co
0000440: 6D 6D 65 6E 74 00 00 00 00 00 00 00 00 00 00 00  mment...........
0000450: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
0000460: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000470: 0B 00 00 00 01 00 00 00 06 00 00 00 00 04 03 00  ................
0000480: 00 04 00 00 0D 00 00 00 00 00 00 00 00 00 00 00  ................
0000490: 10 00 00 00 00 00 00 00 11 00 00 00 01 00 00 00  ................
00004a0: 00 00 00 00 00 00 00 00 0D 04 00 00 1F 00 00 00  ................
00004b0: 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00  ................
00004c0: 01 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00  ................
00004d0: 2C 04 00 00 1A 00 00 00 00 00 00 00 00 00 00 00  ,...............
00004e0: 01 00 00 00 00 00 00 00                          ........
```

上面被星号省去的部分都是0。从中可以看出，从400h到40Dh是仅有的代码，看一下代码5.9，你可能就明白了，0xEBFE正是代码最后的“jmp $”。

代码5.22实现了将 Kernel.bin 根据 ELF 文件信息转移到正确的位置。它很简单，找出每个 Program header，根据其信息进行内存复制。

代码 5.22 InitKernel（节自 chapter5/e/loader.asm）

```
360         call    InitKernel
 ...
713 InitKernel:
714         xor     esi, esi
715         mov     cx, word [BaseOfKernelFilePhyAddr+2Ch]; ecx <- pELFHdr->e_phnum
716         movzx   ecx, cx                                 ;/
717         mov     esi, [BaseOfKernelFilePhyAddr + 1Ch]  ; esi <- pELFHdr->e_phoff
718         add     esi, BaseOfKernelFilePhyAddr;esi<-OffsetOfKernel+pELFHdr->e_phoff
719 .Begin:
720         mov     eax, [esi + 0]
721         cmp     eax, 0                          ; PT_NULL
722         jz      .NoAction
723         push    dword [esi + 010h]      ;size ;'.
724         mov     eax, [esi + 04h]                ; |
725         add     eax, BaseOfKernelFilePhyAddr; | memcpy((void*)(pPHdr->p_vaddr),
726         push    eax                     ;src  ; |       uchCode + pPHdr->p_offset,
727         push    dword [esi + 08h]       ;dst  ; |       pPHdr->p_filesz;
728         call    MemCpy                        ; |
729         add     esp, 12                       ;/
730 .NoAction:
731         add     esi, 020h                     ; esi += pELFHdr->e_phentsize
732         dec     ecx
733         jnz     .Begin
734
735         ret
```

好了，现在万事俱备，只差最后向内核的转移了。不过，我猜有一个问题可能一直在你脑中不曾挥去，就是为什么入口地址是0x30400而不是其他？它看上去不像一个随随便便指定的数字。是的，它的确不是个随便指定的数字，甚至于，在前面章节中我们存放Loader.bin和Kernel.bin的位置也不是随便指定的数字，让我们看一下内核被加载完之后内存的使用情况，你可能就明白了。

图5.8是一个内存使用分布图示。看第一眼的时候你可能有些惊讶，我们不是才往里放了两个文件吗，怎么这么复杂？是的，虽然我们往里存放的内容不多，但它并不单纯。比如我们一直以来用做显示的以0xB8000为开始的内存，显然就不能被 OS 用在常规用途；再比如0x400～0x4FF这段内存，里面存放了许多参数，为了保证在用得着它们的时候它们还在，我们还是暂时保留不覆盖它为妙。

当你看到9FC00h这个数字的时候，不知道你是不是感到面熟，回头看看图3.33和表3.7就明白了，通过中断15h得到的内存信息已经明确地告诉我们，09FC00h～09FFFFh这段内存不能被用做常规使用。即便0h～09FBFFh可以被

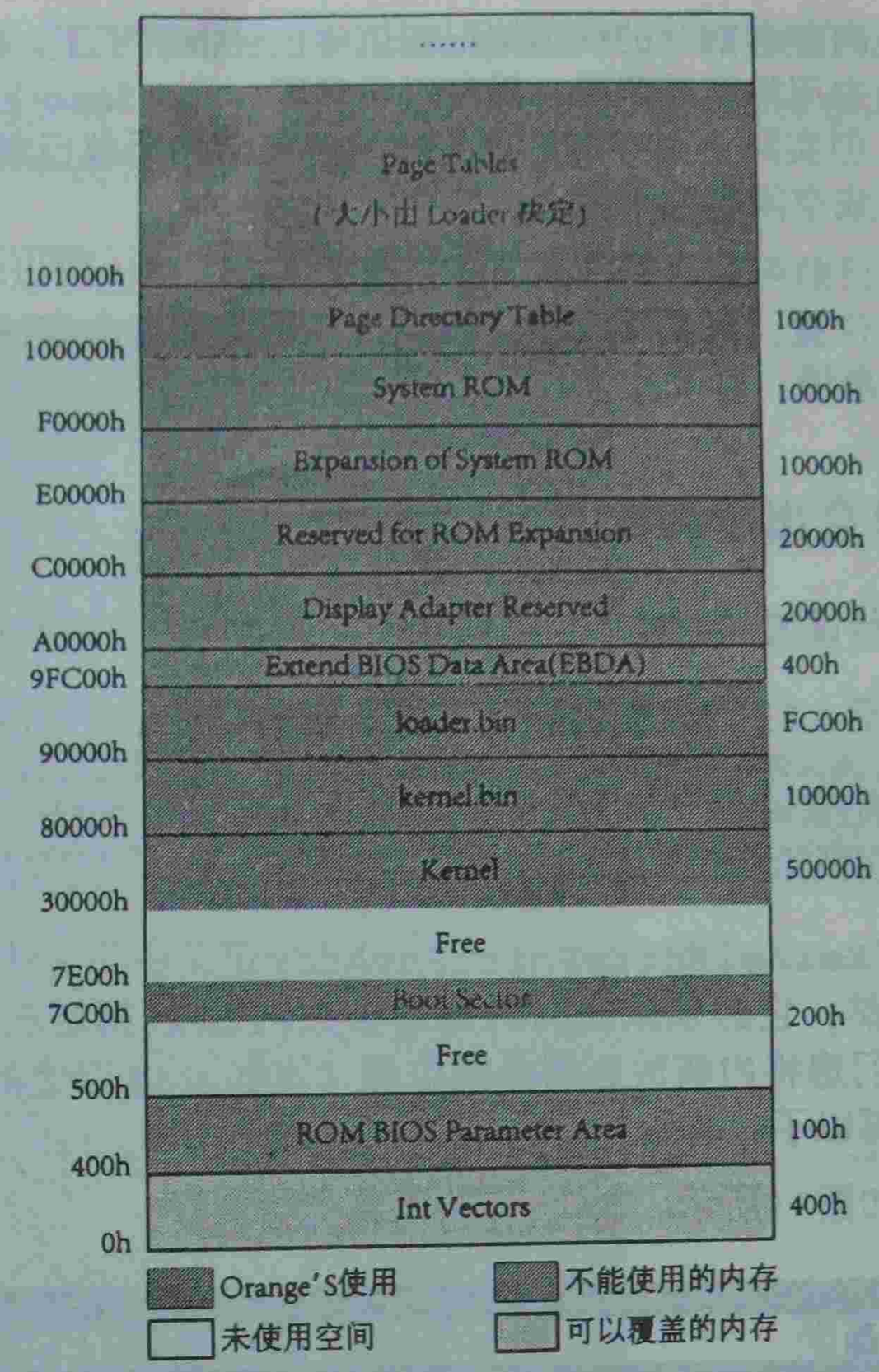


图 5.8 内存使用分布示意图

使用，仍然应该把 BIOS 参数区保护起来以备后用，所以，我们真正可以使用的内存是0500h~09FBFFh这一段。那么，为什么指定的入口地址0x30400离0x500还那么远呢？其实，之所以这么做是为了调试方便。因为大多数的 DOS 都不占用0x30000以上的内存地址，把内核加载到这里，即便在 DOS 下调试也不会覆盖掉 DOS 内存。

现在情况很清楚了，0x90000开始的 63KB 留给了Loader.bin，0x80000开始的 64KB 留给了Kernel.bin，0x30000开始的 320KB 留给整理后的内核，而页目录和页表被放置在了 1MB 以上的内存空间。

我们为Loader.bin留了 63KB 的空间，差一点不到 64KB。一方面因为它本质上是个.COM文件，另一方面我们在写boot.asm时把文件加载在了同一个段中，文件再大也是不允许的，而且，一个 Loader 也不会有那么大，所以，63KB 应该是足够了。

加载文件Kernel.bin到内存时使用的方法跟加载Loader.bin是一样的，也是放在一个段中，所以它也不能超过 64KB。不过，暂时来讲，我们的内核还没有那么大，所以作为权宜之计，倒也未尝不可，况且到时候再对代码进行小的修改并不是一件困难的事情。

好了，现在内存各部分的使用情况相信你已经很明了了。Orange'S放置的位置使得内存看上去用得比较紧凑，虽然引导扇区（Boot Sector）把剩余内存空间分割成了两块，但实际上引导扇区在完成它的使命之后就已经没有用了，所以它本身也可以当成空闲内存来使用。

当然，我们目前可能还用不到那些空闲的内存。你也可以将Orange'S的各个部分放在不同的位置，只要不和图中所示的不能使用的内存冲突就可以了，这不是一件困难的事情，修改几个宏定义就可以了。

5.4.4 向内核交出控制权

该是我们进行试验的时候了，下面我们就试着向内核跳转（代码5.23）。

代码 5.23 向内核跳转（节自 chapter5/e/loader.asm）

```
364     ;*************************************************************************
365     jmp     SelectorFlatC:KernelEntryPointPhyAddr   ; 正式进入内核 *
366     ;*************************************************************************
```

代码5.23中的KernelEntryPointPhyAddr定义在头文件load.inc中，其值为0x30400。当然，它必须跟我们的 ld 的参数-Ttext指定的值是一致的。其实，将来如果我们想将内核放在另外的位置（比如 1MB 以上的内存），只需改动这两个地方就可以了。

运行，结果如图5.9所示。

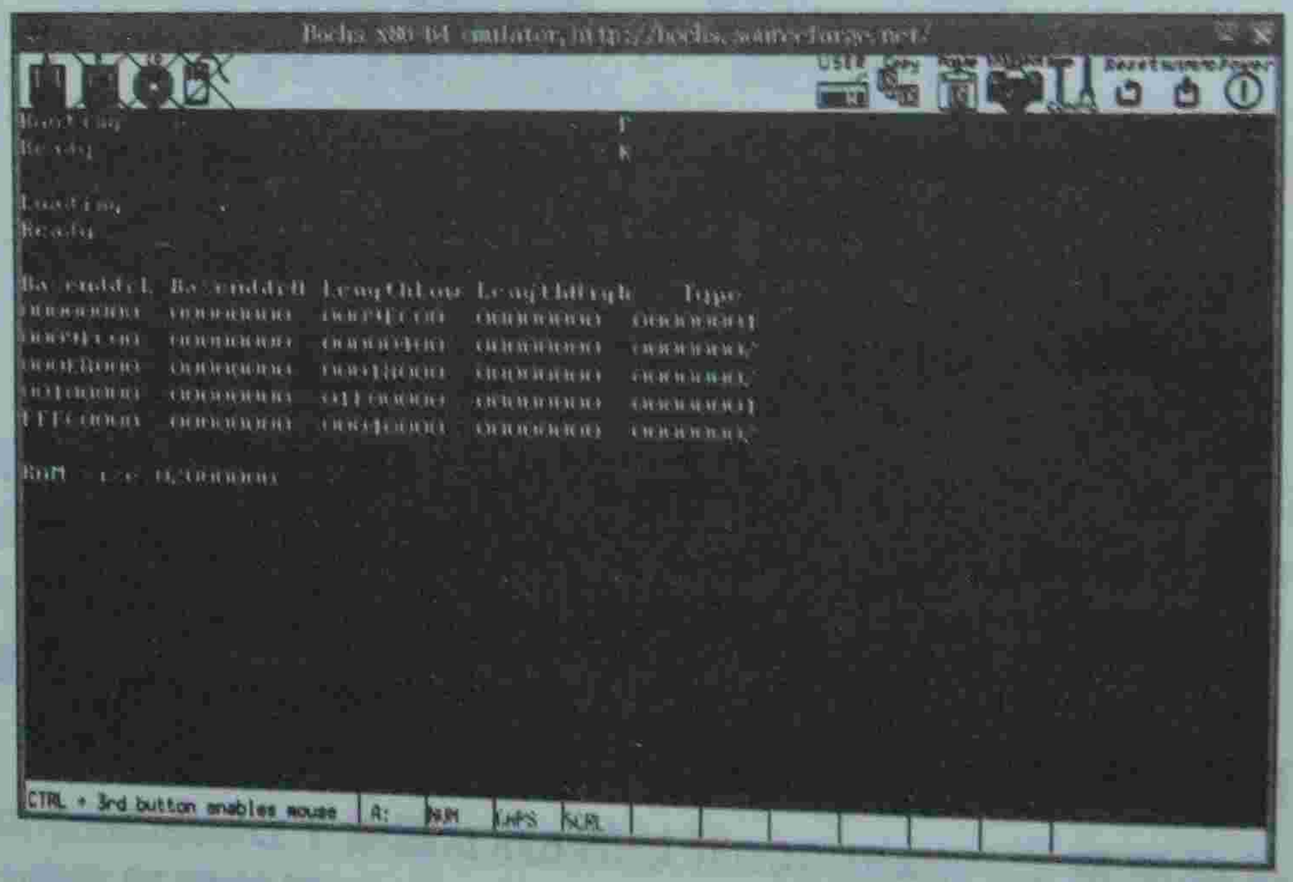


图 5.9 向内核跳转成功

成功了！我们看到，第二行中央出现字符“K”，这表明我们的内核在执行了。

终于迎来了这一刻，Loader 的使命圆满结束，操作系统内核开始运行了。虽然如今我们的“内核”还那么渺小，但你我都知道：来到这里真的很不容易。

然而，正所谓厚积薄发，努力从来不会白费，我们的“内核”虽然简单，我们却有足够的信心将它渐渐扩充，让它慢慢长大。在下面的章节中，你将渐

渐体会到之前不曾有过的畅快——可以使用C语言，当然比汇编来得畅快。

不过，在继续之前，让我们先来回顾一下在内核获得控制权之时各个寄存器的情况，在内核中我们需要这些信息。

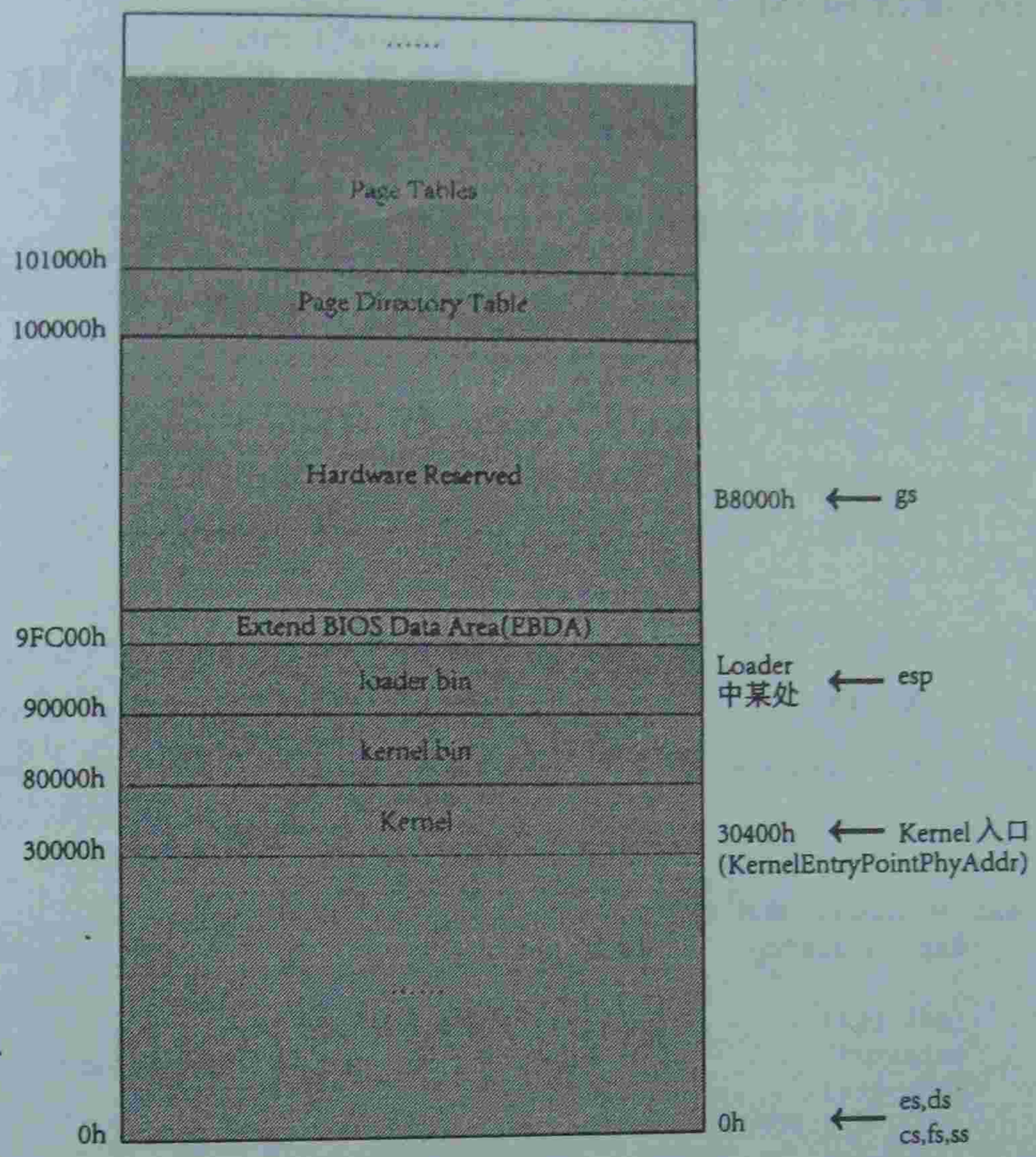


图 5.10 进入内核时寄存器情况示意图

如图5.10所示，cs、ds、es、fs、ss表示的段统统指向内存地址0h，gs表示的段则指向显存，这是我们在进入保护模式之后设置的（参见代码5.14）。同时，esp、GDT等内容也在Loader中，下面对内核进行扩充时，我们会将它们都挪到内核中，以便于控制。

5.5 扩充内核

下面就开始扩充内核，之前要说一句，在前面的章节中，你可能已经发现，我们每向前进行一步，笔者都试图将这样做的原因写清楚，以便让读者不但知其然，更知其所以然，不但看到代码，而且看到在代码没被写出来之前是如何想到这样做的。而在后面的内容中，有一些实现是参考 Minix 来做的，由于懒惰以及自身水平所限，代码有时仅仅是奉行了拿来主义，而没有深究其方法的来源。在享受到前辈优秀代码指引的同时，笔者也借此机会献上对 Andrew S. Tanenbaum 和 Albert S. Woodhull 两位的敬意！

言归正传，我们前面提到，esp、GDT 等内容目前还在 Loader 中，为了方便控制，我们得把它们放进内核中才好，现在我们就来做这项工作。

5.5.1 切换堆栈和 GDT

注意，我们现在可以用 C 语言了，只要能用 C，我们就避免用汇编，这将是我们今后的原则之一。下面我们先来看代码5.24。

代码 5.24 切换堆栈和 GDT（chapter5/f/kernel.asm）

```
20  SELECTOR_KERNEL_CS      equ     8
21
22  ; 导入函数
23  extern  cstart
24
25  ; 导入全局变量
26  extern  gdt_ptr
27
28  [SECTION .bss]
29  StackSpace              resb    2 * 1024
30  StackTop:               ; 栈顶
31
32  [section .text] ; 代码在此
33
34  global _start   ; 导出 _start
35
36  _start:
    ...
78          ; 把 esp 从 LOADER 挪到 KERNEL
79          mov     esp, StackTop   ; 堆栈在 bss 段中
80
81          sgdt    [gdt_ptr]       ; cstart() 中将会用到 gdt_ptr
82          call    cstart          ; 在此函数中改变了gdt_ptr，让它指向新的GDT
83          lgdt    [gdt_ptr]       ; 使用新的GDT
84
85          ;lidt   [idt_ptr]
86
87          jmp     SELECTOR_KERNEL_CS:csinit
88  csinit:         ; “这个跳转指令强制使用刚刚初始化的结构”——<<OS:D&I 2nd>> P90.
89
90          push    0
91          popfd   ; Pop top of stack into EFLAGS
92
93          hlt
```

在这段代码中，用简单的 4 个语句就完成了切换堆栈和更换 GDT 的任务。其中，StackTop定义在.bss段中，堆栈大小为 2KB。操作 GDT 时用到的gdt_ptr和cstart分别是一个全局变量和全局函数，它们定义在start.c中(见代码5.25)。

代码 5.25 start（chapter5/f/start.c）

```
8   #include "type.h"
9   #include "const.h"
10  #include "protect.h"
11
12  PUBLIC  void*   memcpy(void* pDst, void* pSrc, int iSize);
13
    ...
16  PUBLIC  u8              gdt_ptr[6];     /* 0~15:Limit  16~47:Base */
17  PUBLIC  DESCRIPTOR      gdt[GDT_SIZE];
18
```

```
19  PUBLIC void cstart()
20  {
    ...
25          /* 将 LOADER 中的 GDT 复制到新的 GDT 中 */
26          memcpy(&gdt,                                    /* New GDT */
27                 (void*)(*((u32*)(&gdt_ptr[2]))),         /* Base  of Old GDT */
28                 *((u16*)(&gdt_ptr[0])) + 1               /* Limit of Old GDT */
                   );
29          /* gdt_ptr[6] 共 6 个字节: 0~15:Limit  16~47:Base. 用作 sgdt/lgdt 的参数. */
30          u16* p_gdt_limit = (u16*)(&gdt_ptr[0]);
31          u32* p_gdt_base  = (u32*)(&gdt_ptr[2]);
32          *p_gdt_limit = GDT_SIZE * sizeof(DESCRIPTOR) - 1;
33          *p_gdt_base  = (u32)&gdt;
34
35  }
```

函数cstart()首先把位于 Loader 中的原 GDT 全部复制给新的 GDT，然后把gdt_ptr中的内容换成新的 GDT 的基地址和界限。复制 GDT 使用的是函数memcpy，这个函数我们已用过多次了（比如在loader.asm中，当时叫做MemCpy），这次把它的函数体放在string.asm中，这个函数我们已经很熟悉，这里就省去了。

函数cstart()中除了用到的memcpy定义在其他文件之外，还用到了一些新定义的类型、结构体和宏，可以在type.h、const.h以及protect.h中找到。宏PUBLIC定义在const.h中（见代码5.26），同时定义的还有PRIVATE，它们用来区分全局的和局部的符号。

代码 5.26 chapter5/f/const.h

```
8   #ifndef _ORANGES_CONST_H_
9   #define _ORANGES_CONST_H_
10
11
12  /* 函数类型 */
13  #define PUBLIC           /* PUBLIC is the opposite of PRIVATE */
14  #define PRIVATE static   /* PRIVATE x limits the scope of x */
15
16  /* GDT 和 IDT 中描述符的个数 */
17  #define GDT_SIZE         128
18
19
20  #endif /* _ORANGES_CONST_H_ */
```

GDT_SIZE也定义在cosnt.h中。

u8、u16、u32等类型定义在type.h中（见代码5.27），分别代表 8 位、16 位和 32 位的数据类型。定义它们可以让我们的代码增加可读性，一眼看过去就知道类型的长度，在操作gdt_ptr这样的数据时一目了然。

代码 5.27 chapter5/f/type.h

```
8   #ifndef _ORANGES_TYPE_H_
9   #define _ORANGES_TYPE_H_
10
11  typedef unsigned int            u32;
12  typedef unsigned short          u16;
13  typedef unsigned char           u8;
14
15  #endif /* _ORANGES_TYPE_H_ */
```

Descriptor用来表示描述符，它类似于pm.inc中定义的宏Descriptor（见代码5.28）。

代码 5.28 chapter5/f/protect.h

```
8  #ifndef _ORANGES_PROTECT_H_
9  #define _ORANGES_PROTECT_H_
10
11 /* 存储段描述符/系统段描述符 */
12 typedef struct s_descriptor                 /* 共 8 个字节 */
13 {
14         u16     limit_low;                 /* Limit */
15         u16     base_low;                  /* Base */
16         u8      base_mid;                  /* Base */
17         u8      attr1;                     /* P(1) DPL(2) DT(1) TYPE(4) */
18         u8      limit_high_attr2;          /* G(1) D(1) 0(1) AVL(1) LimitHigh(4) */
19         u8      base_high;                 /* Base */
20 }DESCRIPTOR;
21
22 #endif /* _ORANGES_PROTECT_H_ */
```

介绍到这里，想必读者对代码5.25已经完全了解了。虽然头文件和定义看起来比较多，但没什么困难的地方。不把定义放在同一个文件中只是为了使程序结构更好而已。

另外，通过比较代码5.24和代码5.9可知，我们把显示字符“K”的代码去掉了。同时，loader.asm中显示字符“P”的代码也被删除了。我们当时显示它们的目的仅仅是看代码是否执行到了那里，现在我们知道代码运行良好，它们的历史使命也就随之结束了。

好了，现在我们就编译链接：

```
▷ nasm -f elf -o kernel.o kernel.asm
▷ nasm -f elf -o string.o string.asm
▷ gcc -c -o start.o start.c
▷ ld -s -Ttext 0x30400 -o kernel.bin kernel.o \
    string.o start.o
```

运行，结果如图5.11所示。

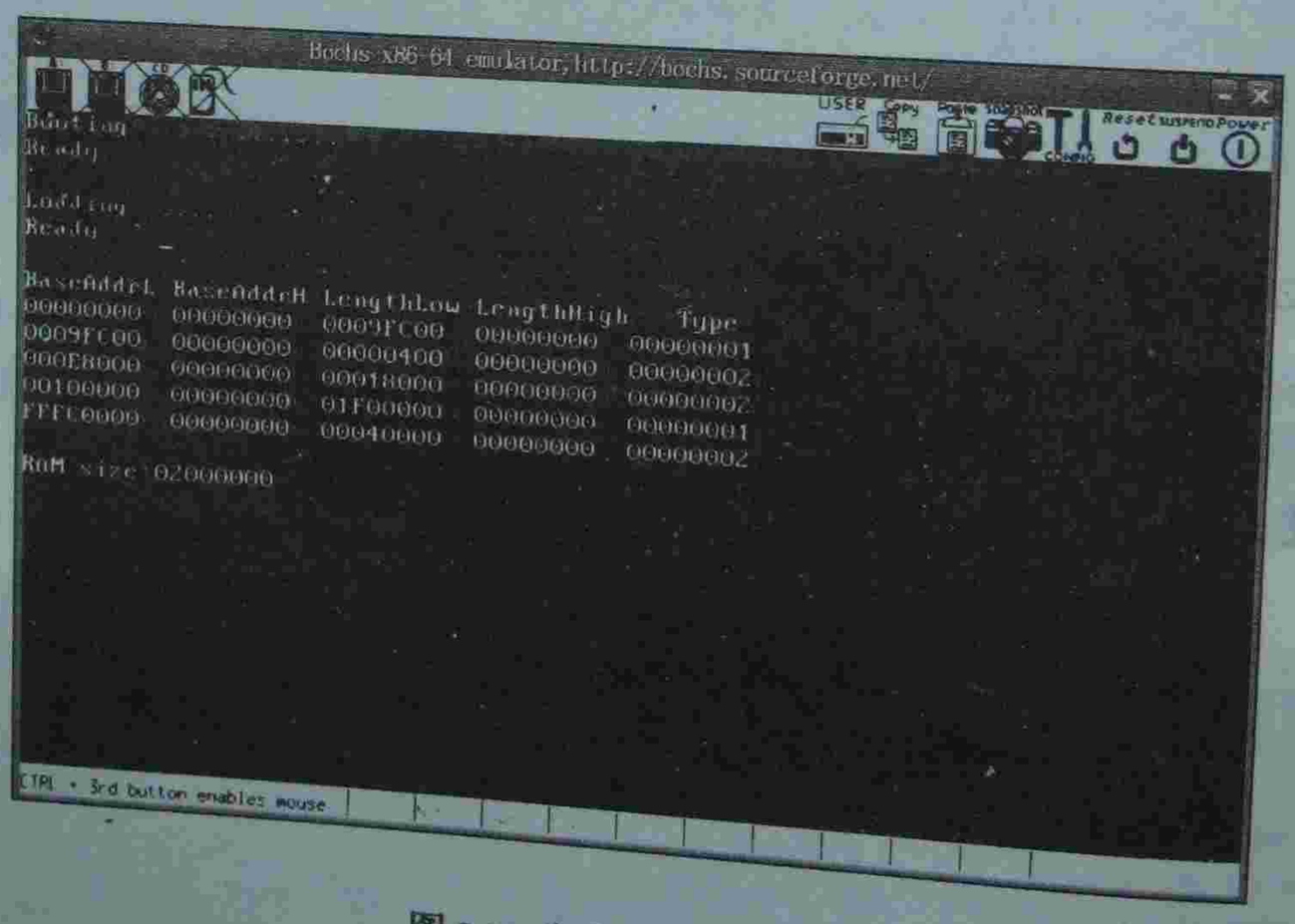


图 5.11 向内核跳转成功

一片寂静，好像什么也没有发生。

这不难理解，我们没有添加任何打印字符或字符串的代码，甚至还删去了“p”和“K”，当然什么也看不到。不要紧，我们有丰富的积累，马上把在第3章中写过的代码复制过来，把它放到一个新的文件kliba.asm中（见代码5.29）。

代码 5.29 chapter5/f/kliba.asm

```
[SECTION .data]
disp_pos        dd      0

[SECTION .text]

; 导出函数
global  disp_str

; ========================================================================
;                  void disp_str(char * info);
; ========================================================================
disp_str:
        push    ebp
        mov     ebp, esp

        mov     esi, [ebp + 8]  ; pszInfo
        mov     edi, [disp_pos]
        mov     ah, 0Fh
.1:
        lodsb
        test    al, al
        jz      .2
        cmp     al, 0Ah ; 是回车吗?
        jnz     .3
        push    eax
        mov     eax, edi
        mov     bl, 160
        div     bl
        and     eax, 0FFh
        inc     eax
        mov     bl, 160
        mul     bl
        mov     edi, eax
        pop     eax
        jmp     .1
.3:
        mov     [gs:edi], ax
        add     edi, 2
        jmp     .1

.2:
        mov     [disp_pos], edi

        pop     ebp
        ret
```

像memcpy一样，只需要简单地声明一下，在C语言代码中就可以方便地使用DispStr了（在这里我们把它改名为disp_str）。马上修改cstart()，添加打印字符串的代码（见代码5.30）。注意，由于变量disp_pos开始被初始化成零，所以如果直接打印字符的话，字符会出现在屏幕左上角，于是代码中disp_str的参数字符串使用了许多个回车（\n），以便让字符串越过已经打印的信息。

代码 5.30 chapter5/f/start.c

```
        disp_str("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"
                 "-----\"cstart\" begins-----\n");
```

下面再来编译一下（别忘了新加了一个源文件kliba.asm）：

```
▷ nasm -f elf -o kernel.o kernel.asm
▷ nasm -f elf -o string.o string.asm
▷ nasm -f elf -o kliba.o kliba.asm
▷ gcc -c -fno-builtin -o start.o start.c
▷ ld -s -Ttext 0x30400 -o kernel.bin kernel.o \
    string.o start.o kliba.o
```

在编译start.c的时候，如果不加参数-fno-builtin，你可能会得到一个警告提示[1]：

```
▷ gcc -c -o start.o start.c
start.c:12: warning: conflicting types for built-in
function 'memcpy'
```

因为memcpy被编译器默认为是“built-in function”。

好了，运行，结果如图5.12所示。

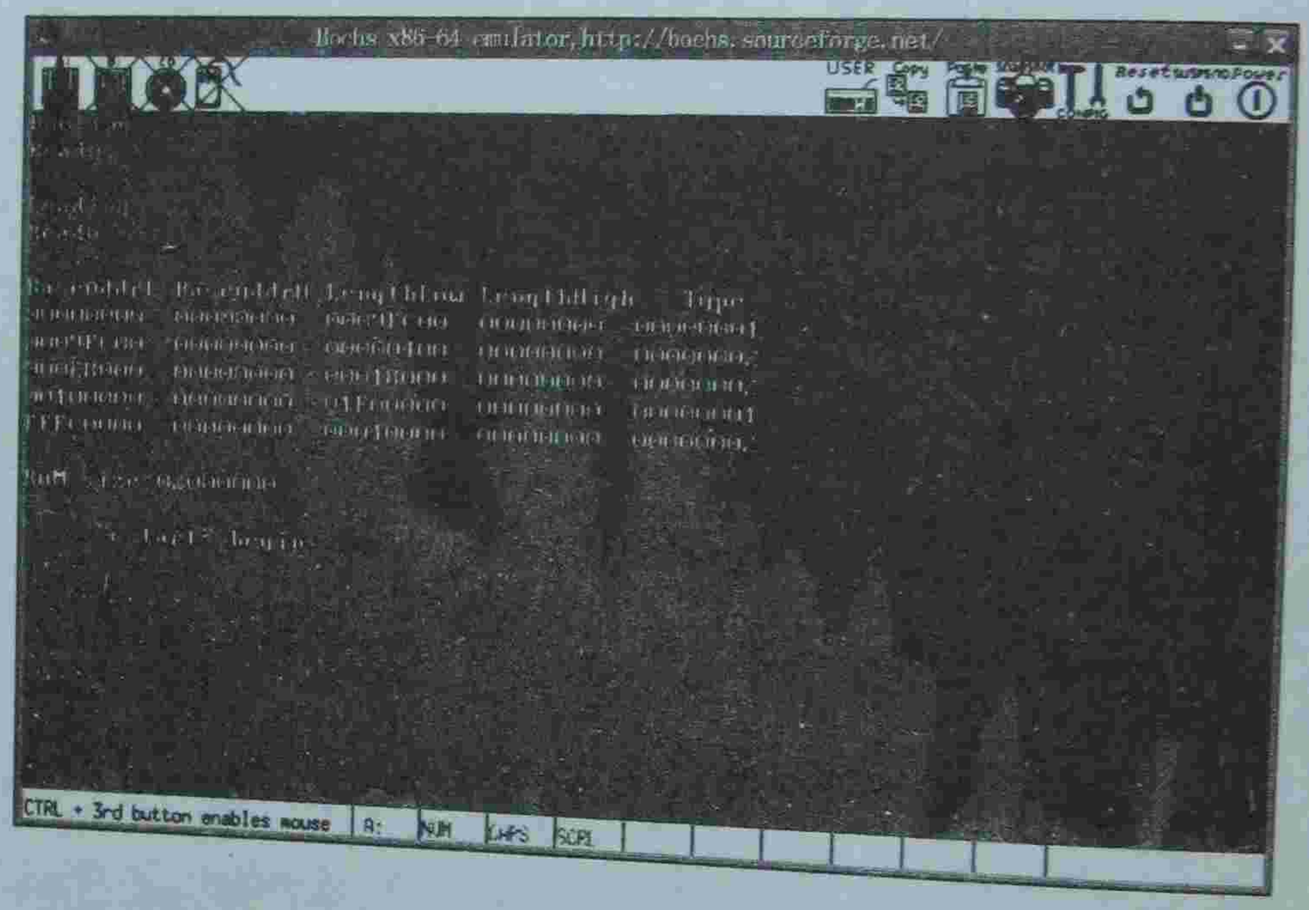


图 5.12 cstart 开始执行

怎么样，我们期待的字符串出现了。

5.5.2 整理我们的文件夹

我想你已经爱上了这种畅快的感觉，只几炷香的工夫，我们的文件飞快增多，代码也多出来不少。可是一个问题出现了，如果你打开放置代码的文件夹，映入眼帘的是.asm、.inc、.c、.h胡乱堆在一起，加上生成的.o、.bin以及bochs用到的.img和bochsrc，简直可以用 A Mess 来形容了。看来我们需要整理一下。

整理目录显然是比写程序简单得多，我们归一下类：

- boot.asm和loader.asm放在单独的目录/boot中，当然它们所需要的头文件也放在里面；

[1]在有些版本的 GCC 中可能不会出现此警告。

- klib.asm和string.asm放在/lib中，作为库的形象出现；
- kernel.asm和start.c放在/kernel里面。

这样一来，结构就清晰多了，目录树如下所示：

```
▷ tree
.
|-- a.img
|-- bochsrc
|-- boot
|   |-- boot.asm
|   |-- include
|   |   |-- fat12hdr.inc
|   |   |-- load.inc
|   |   `-- pm.inc
|   `-- loader.asm
|-- include
|   |-- const.h
|   |-- protect.h
|   `-- type.h
|-- kernel
|   |-- kernel.asm
|   `-- start.c
`-- lib
    |-- kliba.asm
    `-- string.asm
```

怎么样，看上去蛮漂亮的。不知不觉中，我们的代码已经多到变成一棵树了，是不是很有成就感呢？

5.5.3 Makefile

如果读者在跟随本书的介绍一起动手实践的话，可能会想到一个问题，就是随着源代码的增多，编译链接它们的命令行也在慢慢增多，在图5.12的效果出现之前，我们足足输入了5行命令。如果每一次都是这样编译链接的话，真是太痛苦了，而且，现在文件已经分开放置在不同的文件夹，要编译它们无疑变得更加困难。不用担心，运用Makefile，每次只敲入一行命令就可以完成整个过程。

如果你只在Windows下开发过程序，那么很可能没用过Makefile。Makefile内容庞杂——关于它的用法可以单独成书，我们还是遵循够用原则，只学用得着的部分。先来看一个简单的Makefile（代码5.31），我们把它放在目录/boot下，可以用来编译boot.bin和loader.bin。

代码5.31 简单的Makefile（chapter5/g/boot/Makefile）

```
# Makefile for boot

# Programs, flags, etc.
ASM             = nasm
```

```
ASMFLAGS        = -I include/

# This Program
TARGET          = boot.bin loader.bin

# All Phony Targets
.PHONY : everything clean all

# Default starting position
everything : $(TARGET)

clean :
        rm -f $(TARGET)

all : clean everything

boot.bin : boot.asm include/load.inc include/fat12hdr.inc
        $(ASM) $(ASMFLAGS) -o $@ $<

loader.bin : loader.asm include/load.inc include/fat12hdr.inc include/pm.inc
        $(ASM) $(ASMFLAGS) -o $@ $<
```

以字符#开头的行是注释。=用来定义变量，这里，ASM和ASMFLAGS就是两个变量，要注意的是，使用它们的时候要用$(ASM)和$(ASMFLAGS)，而不是它们的原型。其实，明白了这两点，Makefile你就已经明白了一半。.PHONY这个关键字我们暂时不管，来看一下Makefile的最重要的语法：

```
target : prerequisites
        command
```

上面这样的形式代表两层意思：

1. 要想得到 target，需要执行命令 command。
2. target 依赖 prerequisites，当 prerequisites 中至少有一个文件比 target 文件新时，command 才被执行。

比如这个Makefile的最后两行吧，翻译出来就是：

1. 要想得到loader.bin，需要执行“$(ASM) $(ASMFLAGS) -o $@ $<”。
2. loader.bin依赖于以下文件：
   - loader.asm
   - include/load.inc
   - include/pm.inc
   - include/fat12hdr.inc

当它们中至少有一个比loader.bin新时，command 被执行。

那么“$(ASM) $(ASMFLAGS) -o $@ $<”又是什么呢？其实$@和$<意义如下：

- $@代表 target；

- $<代表 prerequisites 的第一个名字。

联系前面我们说过的$(ASM)和$(ASMFLAGS)，这个命令行便等价于：

```
nasm -o loader.bin loader.asm
```

在 Makefile 中我们容易注意到，不但boot.bin和loader.bin两个文件后面有冒号，everything、clean和all后面也有冒号，可是它们 3 个并不是 3 个文件，仅仅是动作名称而已。如果运行“make clean”，将会执行“rm -f $(TARGET)”，也即“rm -f boot.bin loader.bin”。

all后面跟着的是clean和everything，这表明如果执行“make all”，clean和everything所表示的动作将分别被执行。下面就是make all执行的结果：

```
▷ make all
rm -f boot.bin loader.bin
nasm -I include/ -o boot.bin boot.asm
nasm -I include/ -o loader.bin loader.asm
```

刚才被我们忽略过去的关键字.PHONY，其实是表示它后面的名字并不是文件，而仅仅是一种行为的标号。

我们刚才已经运行过make all了，其实直接输入make也是可以的，这时make程序会从第一个名字所代表的动作开始执行。在本例中，第一个标号是everything，所以make和make everything是一样的。下面的过程明白地表示了这一点：

```
▷ make clean
rm -f boot.bin loader.bin
▷ make everything
nasm -I include/ -o boot.bin boot.asm
nasm -I include/ -o loader.bin loader.asm
▷ make clean
rm -f boot.bin loader.bin
▷ make
nasm -I include/ -o boot.bin boot.asm
nasm -I include/ -o loader.bin loader.asm
▷ make
make: Nothing to be done for 'everything'.
```

由于make会自动比较目标和源文件的新旧程度，所以如果运行一个make之后立即运行另一个的话，make程序不会做任何事，因为所有的文件都是新的，不需要生成什么。我们已经看到，第二次运行make时出现

```
make: Nothing to be done for 'everything'.
```

这样的提示。这样就使得我们每一次make时不必把每个源文件都编译一遍（如果一个大型程序有很多源文件的话）。

至此，第一次 Makefile 就已经没有什么陌生的地方了。其实，make 程序的原则就是由果寻因，先看要生成什么，再找生成它需要的条件。这让我想

起一句俗语，叫做“拔出萝卜带出泥”，在本例中，如果把loader.bin比做萝卜，那么loader.asm、include/load.inc、include/fat12hdr.inc、include/pm.inc以及随后的“$(ASM) $(ASMFLAGS) -o $@ $<”都可以被看做是带出的泥。

好了，第一个 Makefile 写成了，我们只需稍微改造和扩充，它就可以发挥强大的作用，用于编译和链接整个操作系统工程了。

我们首先把这个 Makefile 挪到/boot的父目录中，然后把它稍做修改（见代码5.32）。

代码 5.32 chapter5/g/Makefile.boot

```
# Makefile for boot

# Programs, flags, etc.
ASM             = nasm
ASMFLAGS        = -I boot/include/

# This Program
ORANGESBOOT     = boot/boot.bin boot/loader.bin

# All Phony Targets
.PHONY : everything clean all

# Default starting position
everything : $(ORANGESBOOT)

clean :
        rm -f $(ORANGESBOOT)

all : clean everything

boot/boot.bin : boot/boot.asm boot/include/load.inc boot/include/fat12hdr.inc
        $(ASM) $(ASMFLAGS) -o $@ $<

boot/loader.bin : boot/loader.asm boot/include/load.inc \
                  boot/include/fat12hdr.inc boot/include/pm.inc
        $(ASM) $(ASMFLAGS) -o $@ $<
```

跟代码5.31比较起来，代码5.32并没有大的改变，主要是把其中的文件统统加上了路径“boot/”。再运行make：

```
▷ make all -f Makefile.boot
rm -f boot/boot.bin boot/loader.bin
nasm -I boot/include/ -o boot/boot.bin boot/boot.asm
nasm -I boot/include/ -o boot/loader.bin boot/loader.asm
```

注意这里使用参数“-f”，指定使用Makefile.boot，而不是默认的Makefile、makefile或GNUmakefile[2]。

到这里我们对Makefile已经不再陌生了。让我们再接再厉，在Makefile.boot的基础上扩展Makefile（代码5.33）。

代码 5.33 chapter5/g/Makefile

```
############################
# Makefile for Orange'S #
############################
```

[2]具体情况可用 man 命令查看 make 的联机手册。

```
# Entry point of Orange'S
# It must have the same value with 'KernelEntryPointPhyAddr' in load.inc!
ENTRYPOINT      = 0x30400

# Offset of entry point in kernel file
# It depends on ENTRYPOINT
ENTRYOFFSET     =   0x400

# Programs, flags, etc.
ASM             = nasm
DASM            = ndisasm
CC              = gcc
LD              = ld
ASMBFLAGS       = -I boot/include/
ASMKFLAGS       = -I include/ -f elf
CFLAGS          = -I include/ -c -fno-builtin
LDFLAGS         = -s -Ttext $(ENTRYPOINT)
DASMFLAGS       = -u -o $(ENTRYPOINT) -e $(ENTRYOFFSET)

# This Program
ORANGESBOOT     = boot/boot.bin boot/loader.bin
ORANGESKERNEL   = kernel.bin
OBJS            = kernel/kernel.o kernel/start.o lib/kliba.o lib/string.o
DASMOUTPUT      = kernel.bin.asm

# All Phony Targets
.PHONY : everything final image clean realclean disasm all buildimg

# Default starting position
everything : $(ORANGESBOOT) $(ORANGESKERNEL)

all : realclean everything

final : all clean

image : final buildimg

clean :
        rm -f $(OBJS)

realclean :
        rm -f $(OBJS) $(ORANGESBOOT) $(ORANGESKERNEL)

disasm :
        $(DASM) $(DASMFLAGS) $(ORANGESKERNEL) > $(DASMOUTPUT)

# We assume that "a.img" exists in current folder
buildimg :
        dd if=boot/boot.bin of=a.img bs=512 count=1 conv=notrunc
        sudo mount -o loop a.img /mnt/floppy/
        sudo cp -fv boot/loader.bin /mnt/floppy/
        sudo cp -fv kernel.bin /mnt/floppy
        sudo umount /mnt/floppy

boot/boot.bin : boot/boot.asm boot/include/load.inc boot/include/fat12hdr.inc
        $(ASM) $(ASMBFLAGS) -o $@ $<

boot/loader.bin : boot/loader.asm boot/include/load.inc \
                  boot/include/fat12hdr.inc boot/include/pm.inc
        $(ASM) $(ASMBFLAGS) -o $@ $<

$(ORANGESKERNEL) : $(OBJS)
        $(LD) $(LDFLAGS) -o $(ORANGESKERNEL) $(OBJS)

kernel/kernel.o : kernel/kernel.asm
        $(ASM) $(ASMKFLAGS) -o $@ $<
```

```
kernel/start.o : kernel/start.c include/type.h include/const.h include/protect.h
        $(CC) $(CFLAGS) -o $@ $<

lib/kliba.o : lib/kliba.asm
        $(ASM) $(ASMKFLAGS) -o $@ $<

lib/string.o : lib/string.asm
        $(ASM) $(ASMKFLAGS) -o $@ $<
```

可以看到，因为目录层次的原因，我们把GCC的选项也增加了对头文件目录的指定“-I include”。

这个Makefile虽然比原来长出不少，但并没有任何困难之处。不过，它的功能已经非常强大，通过make disasm我们可以反汇编内核到一个文件。甚至于，通过make buildimg或者make image，我们可以直接把引导扇区、loader.bin和kernel.bin写入虚拟软盘。

既然如此神奇，我们就来试一下它的效果，输入make image，执行情况如下：

```
▷ make image
rm -f kernel/kernel.o kernel/start.o lib/kliba.o lib/string.o boot/boot.bin
boot/loader.bin kernel.bin
nasm -I boot/include/ -o boot/boot.bin boot/boot.asm
nasm -I boot/include/ -o boot/loader.bin boot/loader.asm
nasm -I include/ -f elf -o kernel/kernel.o kernel/kernel.asm
gcc -I include/ -c -fno-builtin -o kernel/start.o kernel/start.c
nasm -I include/ -f elf -o lib/kliba.o lib/kliba.asm
nasm -I include/ -f elf -o lib/string.o lib/string.asm
ld -s -Ttext 0x30400 -o kernel.bin kernel/kernel.o kernel/start.o lib/kliba.o
lib/string.o
rm -f kernel/kernel.o kernel/start.o lib/kliba.o lib/string.o
dd if=boot/boot.bin of=a.img bs=512 count=1 conv=notrunc
1+0 records in
1+0 records out
512 bytes (512 B) copied, 4.8611e-05 seconds, 10.5 MB/s
sudo mount -o loop a.img /mnt/floppy/
sudo cp -fv boot/loader.bin /mnt/floppy/ “
boot/loader.” bin -> “/mnt/floppy/loader.” bin
sudo cp -fv kernel.bin /mnt/floppy “
kernel.” bin -> “/mnt/floppy/kernel.” bin
sudo umount /mnt/floppy
```

真的是太棒了，只需要一个命令，make程序就按照预先的步骤将所有工作搞定。不过，你现在可能还不太信任这个新玩意儿，不要紧，我们试验一下就知道它的运行状况了。

来到start.c，在cstart()的结束处添加一行程序（代码5.34），如果我们运行时看到效果改变，就说明make运行正确。

代码5.34 节自 chapter5/g/kernel/start.c

```
22  PUBLIC void cstart()
23  {
24
25          disp_str("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"
                     "-----\"cstart\" begins-----\n");
...
⇒ 38
39          disp_str("-----\"cstart\" ends-----\n");
    }
```

重新make一下，启动，怎么样？预期的字符串出现了（如图5.13所示）！

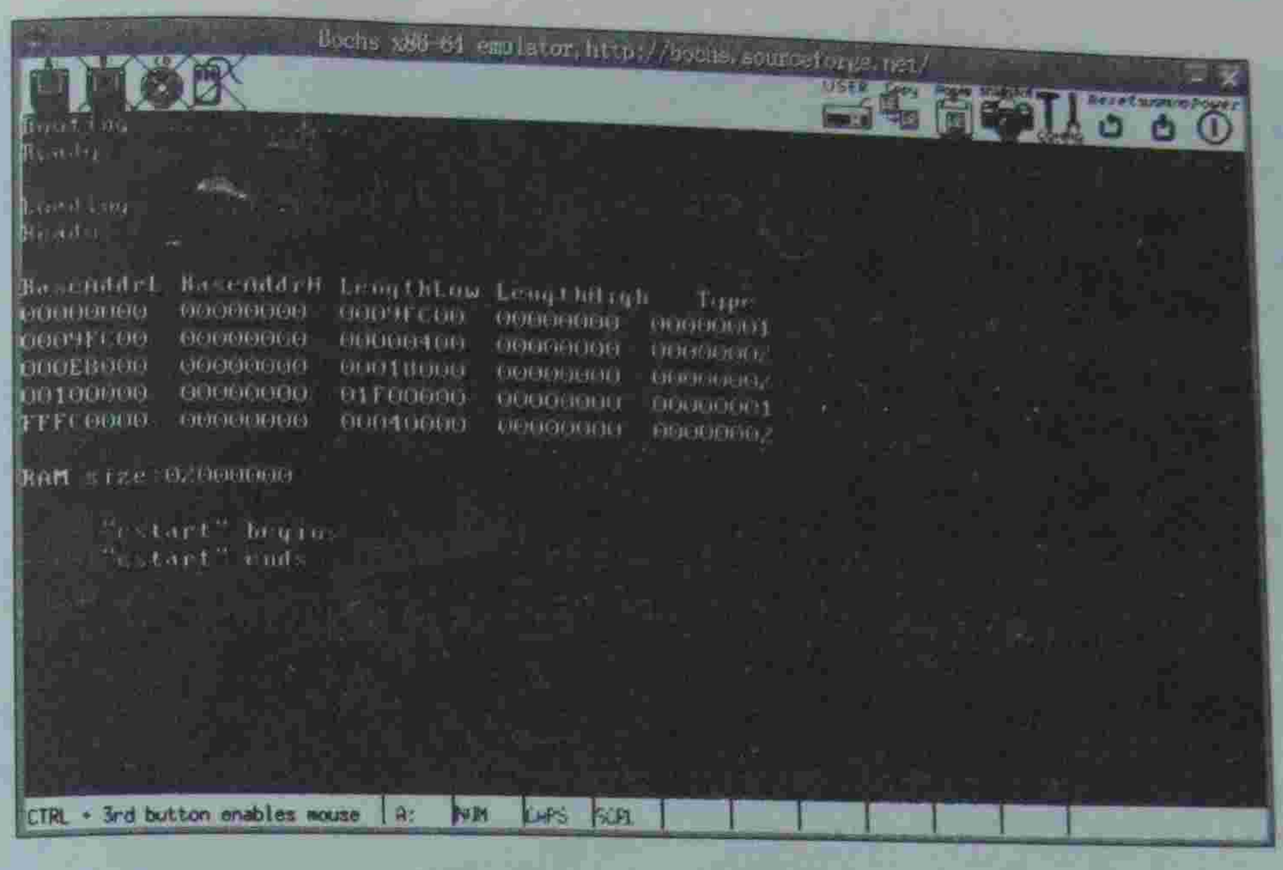


图 5.13 在 cstart() 中打印字符串

我们的Makefile运行正常。这真的是太好了，今后我们重新编译链接的速度不但会大大加快，而且诸如写引导扇区、复制内核文件这样的工作也可以交给make来完成，甚至于你可以根据自己的需要自由添加功能，真是太棒了。

5.5.4 添加中断处理

虽然我们的内核目前还没有做任何实质性的工作，但我们的代码组织结构却已经有了一个雏形，而且我们还拥有好用的Makefile，即便继续增加更多代码的话，我们也可以容易地将它们组织起来。

那么接下来做什么呢？高兴之余，你是不是变得迷惑，不知道接下来该做什么了。不要紧，我们想得稍微远一点，作为一个操作系统，进程毫无疑问是最基本也最重要的东西，于是我们的下一个重大目标应该是实现一个进程。再进一步，我们应该逐渐拥有多个进程。如果从进程本身的角度来看，它只不过是一段执行中的代码，这样看起来它跟我们已经实现的代码没有本质的区别。可是，如果从操作系统角度来看，进程必须是可控制的，所以这就涉及到进程和操作系统之间执行的转换。因为 CPU 只有一个，同一时刻要么是客户进程在运行，要么是操作系统在运行。这个问题我们在后面的章节中还有更加详细的论述。不过我们现在应该清楚，如果实现进程，需要一种控制权转换机制，这种机制便是中断。

中断我们并不陌生，在第 3 章中，我们甚至已经看到了时钟中断发生的效果。好的，就让我们一边复习一边把中断处理添加到我们的 Baby OS 中。

你一定已经回忆起来了，要做的工作有两项：设置 8259A 和建立 IDT。我们先来写一个函数设置 8259A（见代码5.35）。

代码 5.35 初始化 8259A（节自 chapter5/h/kernel/i8259.c）

```
18 PUBLIC void init_8259A()
19 {
20         /* Master 8259, ICW1. */
21         out_byte(INT_M_CTL,     0x11);
```

```
22          /* Slave  8259, ICW1. */
23          out_byte(INT_S_CTL,     0x11);
24
25          /* Master 8259, ICW2. 设置 '主8259' 的中断入口地址为 0x20. */
26          out_byte(INT_M_CTLMASK, INT_VECTOR_IRQ0);
27
28          /* Slave  8259, ICW2. 设置 '从8259' 的中断入口地址为 0x28 */
29          out_byte(INT_S_CTLMASK, INT_VECTOR_IRQ8);
30
31          /* Master 8259, ICW3. IR2 对应 '从8259'. */
32          out_byte(INT_M_CTLMASK, 0x4);
33
34          /* Slave  8259, ICW3. 对应 '主8259' 的 IR2. */
35          out_byte(INT_S_CTLMASK, 0x2);
36
37          /* Master 8259, ICW4. */
38          out_byte(INT_M_CTLMASK, 0x1);
39
40          /* Slave  8259, ICW4. */
41          out_byte(INT_S_CTLMASK, 0x1);
42
43          /* Master 8259, OCW1.  */
44          out_byte(INT_M_CTLMASK, 0xFF);
45
46          /* Slave  8259, OCW1.  */
47          out_byte(INT_S_CTLMASK, 0xFF);
48
49  }
```

我们把初始化8259A的函数命名为init_8259A，它本质上跟第3章中的代码3.34是一样的，只是由汇编代码换成C代码，而且相应的端口被定义成宏。宏定义请见代码5.36和代码5.37。

代码5.36 8259A的端口（节自chapter5/h/include/const.h）

```
28  /* 8259A interrupt controller ports. */
29  #define INT_M_CTL     0x20 /* I/O port for interrupt controller       <Master> */
30  #define INT_M_CTLMASK 0x21 /* setting bits in this port disables ints <Master> */
31  #define INT_S_CTL     0xA0 /* I/O port for second interrupt controller<Slave>  */
32  #define INT_S_CTLMASK 0xA1 /* setting bits in this port disables ints <Slave>  */
```

代码5.37 中断向量（节自chapter5/h/include/protect.h）

```
95  /* 中断向量 */
96  #define INT_VECTOR_IRQ0                 0x20
97  #define INT_VECTOR_IRQ8                 0x28
```

函数init_8259A中只用到一个函数，就是用来写端口的out_byte，它的函数体位于kliba.asm中（见代码5.38）。其中，不但有out_byte，用于对端口进行写操作，还有in_byte，用来对端口进行读操作。由于端口操作可能需要时间，所以两个函数中都加了点空操作以便有微小的延迟。

代码5.38 chapter5/h/lib/kliba.asm

```
96  ; ========================================================================
97  ;                 void out_byte(u16 port, u8 value);
98  ; ========================================================================
99  out_byte:
100         mov     edx, [esp + 4]
101         mov     al, [esp + 4 + 4]       ; port
102         out     dx, al                  ; value
103         nop     ; 一点延迟
```

```
        nop
        ret

; ========================================================================
;                  u8 in_byte(u16 port);
; ========================================================================
in_byte:
        mov     edx, [esp + 4]          ; port
        xor     eax, eax
        in      al, dx
        nop     ; 一点延迟
        nop
        ret
```

这两个函数的原型放在了include/proto.h中，这是一个新建立的头文件，用来存放函数声明（见代码5.39）。可以看到，start.c中函数disp_str的声明也被挪到了里面。

代码 5.39 函数生命（节自 chapter5/h/include/proto.h）

```
PUBLIC void     out_byte(u16 port, u8 value);
PUBLIC u8       in_byte(u16 port);
PUBLIC void     disp_str(char * info);
```

在挪动disp_str的函数声明时，一定也注意到了和它挨着的memcpy，我们把它也放进一个头文件，这个头文件是新建立的，取名为string.h。

当然，由于新增加了头文件，在相应的.c文件中不能忘了包含它们。

这些地方都改完之后，最后一件重要的事情就是修改Makefile。不但要添加新的目标kernel/i8259.o，而且由于头文件的变化，kernel/start.o的依赖关系也稍有变化（见代码5.40）。

代码 5.40 chapter5/h/Makefile

```
OBJS = kernel/kernel.o kernel/start.o kernel/i8259.o lib/kliba.o lib/string.o
 ...
kernel/start.o: kernel/start.c include/type.h include/const.h \
                include/protect.h include/proto.h include/string.h
        $(CC) $(CFLAGS) -o $@ $<

kernel/i8259.o : kernel/i8259.c include/type.h include/const.h \
                         include/protect.h include/proto.h
        $(CC) $(CFLAGS) -o $@ $<
```

当确定依赖关系的时候，你可能觉得有点麻烦，尤其是当头文件越来越多，可以想像到时候可能更让人眼花缭乱。不要紧，GCC 提供了一个参数“-M”，可以自动生成依赖关系。下面是“gcc -M”的典型用法：

```
▷ gcc -M kernel/start.c -I include
start.o: kernel/start.c include/type.h include/const.h include/ protect.h \
 include/proto.h include/string.h
```

我们直接把输出复制到Makefile中就可以了。

其实，现在我们已经可以make一下了。虽然目前还没有完成任何实质性的工作，但是make一下，测试一下自己的工作有没有错误还是可以的。通过之后运行我们的操作系统并不会有什么新鲜效果出现，我们甚至还没有添加调用init_8259A的代码。不要紧，我们继续往下走来初始化 IDT。

说起 IDT 让我们不能不想起 GDT，当初初始化它所用的方法，我们同样可以拿过来用。首先修改start.c（见代码5.41）。

代码 5.41 初始化 IDT（节自 chapter5/h/kernel/start.c）

```
13  #include "global.h"
    ...
35          /* idt_ptr[6] 共 6 个字节：0~15:Limit  16~47:Base。用作 sidt/lidt 的参数。*/
36          u16* p_idt_limit = (u16*)(&idt_ptr[0]);
37          u32* p_idt_base  = (u32*)(&idt_ptr[2]);
38          *p_idt_limit = IDT_SIZE * sizeof(GATE) - 1;
39          *p_idt_base  = (u32)&idt;
```

代码跟先前初始化 GDT 的部分基本上是一样的，只是所有的 GDT 字眼变成了 IDT。不过你会发现，原来位于start.c开头的gdt[]和gdt_ptr[]的声明不在了，取而代之的是对头文件global.h的包含。gdt[]、gdt_ptr[]以及新增加的变量idt[]和idt_ptr[]都放在了这个新建的头文件中。之所以把全局变量声明都放在其中是为了代码的美感和可读性（见代码5.42）。

代码 5.42 全局变量的声明（节自 chapter5/h/include/global.h）

```
8   /* EXTERN is defined as extern except in global.c */
9   #ifdef  GLOBAL_VARIABLES_HERE
10  #undef  EXTERN
11  #define EXTERN
12  #endif
13
14  EXTERN  int             disp_pos;
15  EXTERN  u8              gdt_ptr[6];     /* 0~15:Limit  16~47:Base */
16  EXTERN  DESCRIPTOR      gdt[GDT_SIZE];
17  EXTERN  u8              idt_ptr[6];     /* 0~15:Limit  16~47:Base */
18  EXTERN  GATE            idt[IDT_SIZE];
```

EXTERN定义在const.h中（代码5.44），通常情况下它被定义成extern。但是在global.h中你会发现，如果宏GLOBAL_VARIABLES_HERE被定义的话，EXTERN将会被定义成空值。这样做的意图联系global.c（见代码5.43）你就全明白了。你会发现，通过宏GLOBAL_VARIABLES_HERE的使用，在让所有变量只出现一次（在global.h中）的同时，预编译结束后，global.c和其他.c文件中的结果不同。在global.c中，变量前面没有extern关键字，而在其他文件中，变量前将会有extern关键字。

代码 5.43 全局变量（节自 chapter5/h/kernel/global.c）

```
8   #define GLOBAL_VARIABLES_HERE
9
10  #include "type.h"
11  #include "const.h"
12  #include "protect.h"
13  #include "proto.h"
14  #include "global.h"
```

代码 5.44 EXTERN（节自 chapter5/h/include/const.h）

```
12  /* EXTERN is defined as extern except in global.c */
13  #define EXTERN extern
    ...
21  #define IDT_SIZE        256
```

可以看到，IDT_SIZE的定义也在const.h中。

另外，GATE的定义在protect.h中（代码5.45）。

代码 5.45 GATE（节自 chapter5/h/include/protect.h）

```
/* 门描述符 */
typedef struct s_gate
{
        u16     offset_low;     /* Offset Low */
        u16     selector;       /* Selector */
        u8      dcount;         /* 该字段只在调用门描述符中有效。如果在利用
                                   调用门调用子程序时引起特权级的转换和堆栈
                                   的改变，需要将外层堆栈中的参数复制到内层
                                   堆栈。该双字计数字段就是用于说明这种情况
                                   发生时，要复制的双字参数的数量。*/
        u8      attr;           /* P(1) DPL(2) DT(1) TYPE(4) */
        u16     offset_high;    /* Offset High */
}GATE;
```

好了，start.c修改完之后，我们在kernel.asm中添加两句，导入idt_ptr这个符号（代码5.46第17行）并加载 IDT（代码5.46第97行）。

现在，加载 IDT 的代码已经写完了。不过，现在 IDT 内还没有任何内容呢，要抓紧添加。我们在第 3 章的代码中写得比较简单，仅有两个中断门而已，我们要让 IDT 充实起来。

我们曾经在第 3 章的表3.8中给出了处理器可以处理的中断和异常列表，现在，该是把这些中断和异常的处理程序统统添加上的时候了。虽然它们总数有十几个，但我们却可以用相似的方法来处理它们（见代码5.46）。

如果你已经忘记异常发生时堆栈的变化情况，请翻回第 3 章看一下图3.45。从中可以看到，中断或异常发生时eflags、cs、eip已经被压栈，如果有错误码的话，错误码也已经被压栈。所以我们对异常处理的总体思想是，如果有错误码，则直接把向量号压栈，然后执行一个函数exception_handler；如果没有错误码，则先在栈中压入一个0xFFFFFFFF，再把向量号压栈并随后执行exception_handler。

函数exception_handler()的原型是这样的：

```
void exception_handler(int vec_no, int err_code,
                       int eip, int cs, int eflags);
```

由于C调用约定是调用者恢复堆栈，所以不用担心exception_handler会破坏堆栈中的eip、cs以及eflags。

代码 5.46 中断和异常（chapter5/h/kernel/kernel.asm）

```
17  extern  idt_ptr
    ...
27  global _start   ; 导出 _start
28
29  global  divide_error
30  global  single_step_exception
31  global  nmi
32  global  breakpoint_exception
33  global  overflow
34  global  bounds_check
35  global  inval_opcode
36  global  copr_not_available
37  global  double_fault
38  global  copr_seg_overrun
```

```
39  global  inval_tss
40  global  segment_not_present
41  global  stack_exception
42  global  general_protection
43  global  page_fault
44  global  copr_error
 ...
97          lidt    [idt_ptr]
 ...
112 divide_error:
113         push    0xFFFFFFFF      ; no err code
114         push    0               ; vector_no     = 0
115         jmp     exception
116 single_step_exception:
117         push    0xFFFFFFFF      ; no err code
118         push    1               ; vector_no     = 1
119         jmp     exception
120 nmi:
121         push    0xFFFFFFFF      ; no err code
122         push    2               ; vector_no     = 2
123         jmp     exception
124 breakpoint_exception:
125         push    0xFFFFFFFF      ; no err code
126         push    3               ; vector_no     = 3
127         jmp     exception
128 overflow:
129         push    0xFFFFFFFF      ; no err code
130         push    4               ; vector_no     = 4
131         jmp     exception
132 bounds_check:
133         push    0xFFFFFFFF      ; no err code
134         push    5               ; vector_no     = 5
135         jmp     exception
136 inval_opcode:
137         push    0xFFFFFFFF      ; no err code
138         push    6               ; vector_no     = 6
139         jmp     exception
140 copr_not_available:
141         push    0xFFFFFFFF      ; no err code
142         push    7               ; vector_no     = 7
143         jmp     exception
144 double_fault:
145         push    8               ; vector_no     = 8
146         jmp     exception
147 copr_seg_overrun:
148         push    0xFFFFFFFF      ; no err code
149         push    9               ; vector_no     = 9
150         jmp     exception
151 inval_tss:
152         push    10              ; vector_no     = A
153         jmp     exception
154 segment_not_present:
155         push    11              ; vector_no     = B
156         jmp     exception
157 stack_exception:
158         push    12              ; vector_no     = C
159         jmp     exception
160 general_protection:
161         push    13              ; vector_no     = D
162         jmp     exception
163 page_fault:
164         push    14              ; vector_no     = E
165         jmp     exception
166 copr_error:
167         push    0xFFFFFFFF      ; no err code
168         push    16              ; vector_no     = 10h
169         jmp     exception
```

```
exception:
        call    exception_handler
        add     esp, 4*2            ; 让栈顶指向 EIP，堆栈中从顶向下依次是：EIP、CS、EFLAGS
        hlt
```

我们看到，在代码5.46的最后，栈顶被调整为指向eip，堆栈中从顶向下依次是：eip、cs、eflags。虽然目前我们到这里就让程序停住了，但这样做有利于提醒我们以后修改时注意，用iretd返回前的样子应该是这样的。

我们下面就来看一下函数exception_handler（见代码5.47），它的实现实际上也很简单，首先把屏幕的前5行通过打印空格的方式清空，然后把堆栈中的参数打印出来。

代码 5.47 异常处理函数（节自 chapter5/h/kernel/protect.c）

```
PUBLIC void exception_handler(int vec_no,int err_code,int eip,int cs,int eflags)
{
        int i;
        int text_color = 0x74; /* 灰底红字 */

        char * err_msg[] = {"#DE Divide Error",
                            "#DB RESERVED",
                            "--  NMI Interrupt",
                            "#BP Breakpoint",
                            "#OF Overflow",
                            "#BR BOUND Range Exceeded",
                            "#UD Invalid Opcode (Undefined Opcode)",
                            "#NM Device Not Available (No Math Coprocessor)",
                            "#DF Double Fault",
                            "    Coprocessor Segment Overrun (reserved)",
                            "#TS Invalid TSS",
                            "#NP Segment Not Present",
                            "#SS Stack-Segment Fault",
                            "#GP General Protection",
                            "#PF Page Fault",
                            "--  (Intel reserved. Do not use.)",
                            "#MF x87 FPU Floating-Point Error (Math Fault)",
                            "#AC Alignment Check",
                            "#MC Machine Check",
                            "#XF SIMD Floating-Point Exception"
        };

        /* 通过打印空格的方式清空屏幕的前五行，并把 disp_pos 清零 */
        disp_pos = 0;
        for(i=0;i<80*5;i++){
                disp_str(" ");
        }
        disp_pos = 0;

        disp_color_str("Exception! --> ", text_color);
        disp_color_str(err_msg[vec_no], text_color);
        disp_color_str("\n\n", text_color);
        disp_color_str("EFLAGS:", text_color);
        disp_int(eflags);
        disp_color_str("CS:", text_color);
        disp_int(cs);
        disp_color_str("EIP:", text_color);
        disp_int(eip);

        if(err_code != 0xFFFFFFFF){
                disp_color_str("Error code:", text_color);
                disp_int(err_code);
        }
}
```

在这里，我们新建立了一个文件protect.c用来放置exception_handler。需要提醒的是，每新建一个源文件，我们都要考虑在Makefile做出相应改变。

为了突出显示，exception_handler中打印字符串不再使用disp_str而使用了函数disp_color_str()，它和disp_str()基本上是一样的，区别在于增加了一个设置颜色的参数，见代码5.48。

代码 5.48 函数 disp_color_str（chapter5/h/lib/kliba.asm）

```
61  disp_color_str:
62          push    ebp
63          mov     ebp, esp
64
65          mov     esi, [ebp + 8]   ; pszInfo
66          mov     edi, [disp_pos]
⇒ 67        mov     ah, [ebp + 12]   ; color
68  .1:
69          lodsb
70          test    al, al
71          jz      .2
72          cmp     al, 0Ah ; 是回车吗?
73          jnz     .3
74          push    eax
75          mov     eax, edi
76          mov     bl, 160
77          div     bl
78          and     eax, 0FFh
79          inc     eax
80          mov     bl, 160
81          mul     bl
82          mov     edi, eax
83          pop     eax
84          jmp     .1
85  .3:
86          mov     [gs:edi], ax
87          add     edi, 2
88          jmp     .1
89
90  .2:
91          mov     [disp_pos], edi
92
93          pop     ebp
94          ret
```

另外，为了显示整数，我们新编写了函数disp_int()，它被定义在新建的文件klib.c中，见代码5.49。

代码 5.49 函数 disp_int（节自 chapter5/h/lib/klib.c）

```
16  /*======================================================================*
17                               itoa
18   *======================================================================*/
19  /* 数字前面的 0 不被显示出来，比如 0000B800 被显示成 B800 */
20  PUBLIC char * itoa(char * str, int num)
21  {
22          char *  p = str;
23          char    ch;
24          int     i;
25          int     flag = 0;
26
27          *p++ = '0';
28          *p++ = 'x';
29
```

```
        if(num == 0){
                *p++ = '0';
        }
        else{
                for(i=28;i>=0;i-=4){
                        ch = (num >> i) & 0xF;
                        if(flag || (ch > 0)){
                                flag = 1;
                                ch += '0';
                                if(ch > '9'){
                                        ch += 7;
                                }
                                *p++ = ch;
                        }
                }
        }

        *p = 0;

        return str;
}

/*======================================================================*
                                disp_int
 *======================================================================*/
PUBLIC void disp_int(int input)
{
        char output[16];
        itoa(output, input);
        disp_str(output);
}
```

disp_int很简单，用itoa()将整数转换成字符串后显示出来。itoa()也定义在klib.c中，不过它和C库函数itoa()比起来要简单得多，目的只是把一个32位的数值用十六进制的方式显示出来，既不支持其他进制的转换，也不考虑有符号数等情况。

现在我们已经有了异常处理函数，该是设置IDT的时候了。我们把设置IDT的代码放进函数init_prot()中（见代码5.51），它也位于protect.c中。

protect.c通篇几乎只调用一个函数，就是init_idt_desc()（代码5.50），它用来初始化一个门描述符。其中用到的函数指针类型是这样定义的（位于type.h）：

```
typedef void (*int_handler) ();
```

所有的异常处理程序都必须与此声明完全一致（见代码5.51）。

代码5.50 函数init_idt_desc（节自chapter5/h/kernel/protect.c）

```
PRIVATE void init_idt_desc(unsigned char vector, u8 desc_type,
                           int_handler handler, unsigned char privilege)
{
        GATE *  p_gate  = &idt[vector];
        u32     base    = (u32)handler;
        p_gate->offset_low      = base & 0xFFFF;
        p_gate->selector        = SELECTOR_KERNEL_CS;
        p_gate->dcount          = 0;
        p_gate->attr            = desc_type | (privilege << 5);
        p_gate->offset_high     = (base >> 16) & 0xFFFF;
}
```

在init_prot()中，所有描述符都被初始化成中断门。函数中用到了若干宏，其中以INT_VECTOR_开头的宏表示中断向量，DA_386IGate表示中断门，在定义protect.h中定义，PRIVILEGE_KRNL和PRIVILEGE_USER定义在const.h中。

另外，调用init_8259A()的语句也放在了这个函数中。

代码5.51 函数init_prot（节自chapter5/h/kernel/protect.c）

```
19  void    divide_error();
20  void    single_step_exception();
21  void    nmi();
22  void    breakpoint_exception();
23  void    overflow();
24  void    bounds_check();
25  void    inval_opcode();
26  void    copr_not_available();
27  void    double_fault();
28  void    copr_seg_overrun();
29  void    inval_tss();
30  void    segment_not_present();
31  void    stack_exception();
32  void    general_protection();
33  void    page_fault();
34  void    copr_error();
35
36  /*======================================================================*
37                               init_prot
38   *======================================================================*/
39  PUBLIC void init_prot()
40  {
41      init_8259A();
42
43      // 全部初始化成中断门(没有陷阱门)
44      init_idt_desc(INT_VECTOR_DIVIDE,        DA_386IGate,
45                    divide_error,             PRIVILEGE_KRNL);
46
47      init_idt_desc(INT_VECTOR_DEBUG,         DA_386IGate,
48                    single_step_exception,    PRIVILEGE_KRNL);
49
50      init_idt_desc(INT_VECTOR_NMI,           DA_386IGate,
51                    nmi,                      PRIVILEGE_KRNL);
52
53      init_idt_desc(INT_VECTOR_BREAKPOINT,    DA_386IGate,
54                    breakpoint_exception,     PRIVILEGE_USER);
55
56      init_idt_desc(INT_VECTOR_OVERFLOW,      DA_386IGate,
57                    overflow,                 PRIVILEGE_USER);
58
59      init_idt_desc(INT_VECTOR_BOUNDS,        DA_386IGate,
60                    bounds_check,             PRIVILEGE_KRNL);
61
62      init_idt_desc(INT_VECTOR_INVAL_OP,      DA_386IGate,
63                    inval_opcode,             PRIVILEGE_KRNL);
64
65      init_idt_desc(INT_VECTOR_COPROC_NOT,    DA_386IGate,
66                    copr_not_available,       PRIVILEGE_KRNL);
67
68      init_idt_desc(INT_VECTOR_DOUBLE_FAULT,  DA_386IGate,
69                    double_fault,             PRIVILEGE_KRNL);
70
71      init_idt_desc(INT_VECTOR_COPROC_SEG,    DA_386IGate,
72                    copr_seg_overrun,         PRIVILEGE_KRNL);
73
74      init_idt_desc(INT_VECTOR_INVAL_TSS,     DA_386IGate,
75                    inval_tss,                PRIVILEGE_KRNL);
```

```
76          init_idt_desc(INT_VECTOR_SEG_NOT,       DA_386IGate,
77                        segment_not_present,      PRIVILEGE_KRNL);
78
79          init_idt_desc(INT_VECTOR_STACK_FAULT,   DA_386IGate,
80                        stack_exception,          PRIVILEGE_KRNL);
81
82          init_idt_desc(INT_VECTOR_PROTECTION,    DA_386IGate,
83                        general_protection,       PRIVILEGE_KRNL);
84
85          init_idt_desc(INT_VECTOR_PAGE_FAULT,    DA_386IGate,
86                        page_fault,               PRIVILEGE_KRNL);
87
88          init_idt_desc(INT_VECTOR_COPROC_ERR,    DA_386IGate,
89                        copr_error,               PRIVILEGE_KRNL);
90
91  }
```

至此设置IDT的代码总算添加得差不多了，我们现在来调用init_prot()(代码5.52)。

代码 5.52 函数 init_prot（节自 chapter5/h/kernel/start.c）

```
⇒ 41      init_prot();
42
43      disp_str("-----\"cstart\" ends-----\n");
```

对Makefile进行相应的修改之后，我们就可以先make一下了，通过之后运行，你会发现什么效果也没有。是啊，我们添加了异常处理程序，但是没有异常发生，怎么能有效果呢？好的，我们就制造一个异常来试试看。Intel 为我们准备了一个指令叫做ud2，能够产生一个 #UD 异常，我们就在kernel.asm中添加一条ud2指令（代码5.53）。

代码 5.53 ud2（chapter5/h/kernel/kernel.asm）

```
100 csinit:         ; 这个跳转指令强制使用刚刚初始化的结构
101
102     ud2
```

再make，然后运行，怎么样，看到效果了吗？

不出意外的话，你应该可以看到图5.14所示的效果了，异常的助记符、名字以及eflags、cs、eip的值都被打印了出来。

这是个没有错误码的异常，你可能觉得还不过瘾，我们再来产生一个有错误码的异常，把ud2这行指令修改成jmp 0x40:0。运行，你会发现错误码也显示出来了，如图5.15所示。

虽然只是初始化 8259A 和设置 IDT 这两项任务，却也费了我们这么多的精力，零碎的东西还真不少。不过，现在我们已经有了异常处理机制，今后，即便出了错，我们也能方便地知道错误出在什么地方以及错误的类型。与 Bochs 的调试功能结合起来，这简直可以称得上是操作系统编写的双保险，从此我们可以放心地开发了。

不过，8259A 虽然已经设置完成，但是我们还没有真正开始使用它呢。不要紧，有了异常处理的经验，这项工作就变得轻车熟路了。

复习一下图3.39，我们知道，两片级联的 8259A 可以挂接 15 个不同的外部设备，我们也理应有 15 个中断处理程序。为简单起见，我们写两个带参数的宏，用它们作为中断处理程序。代码5.54就是 8259A 的中断例程。

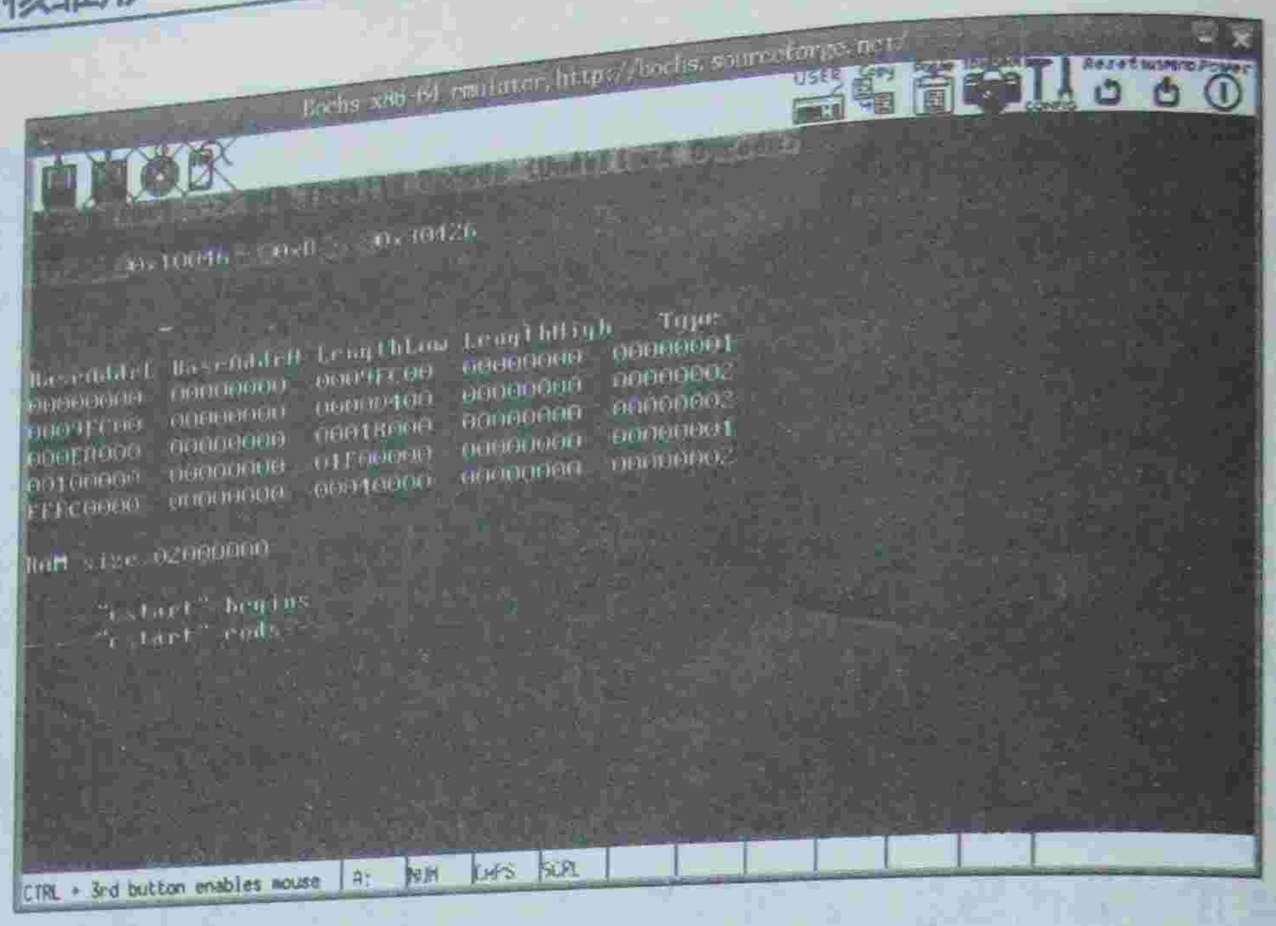


图 5.14 #UD 演示

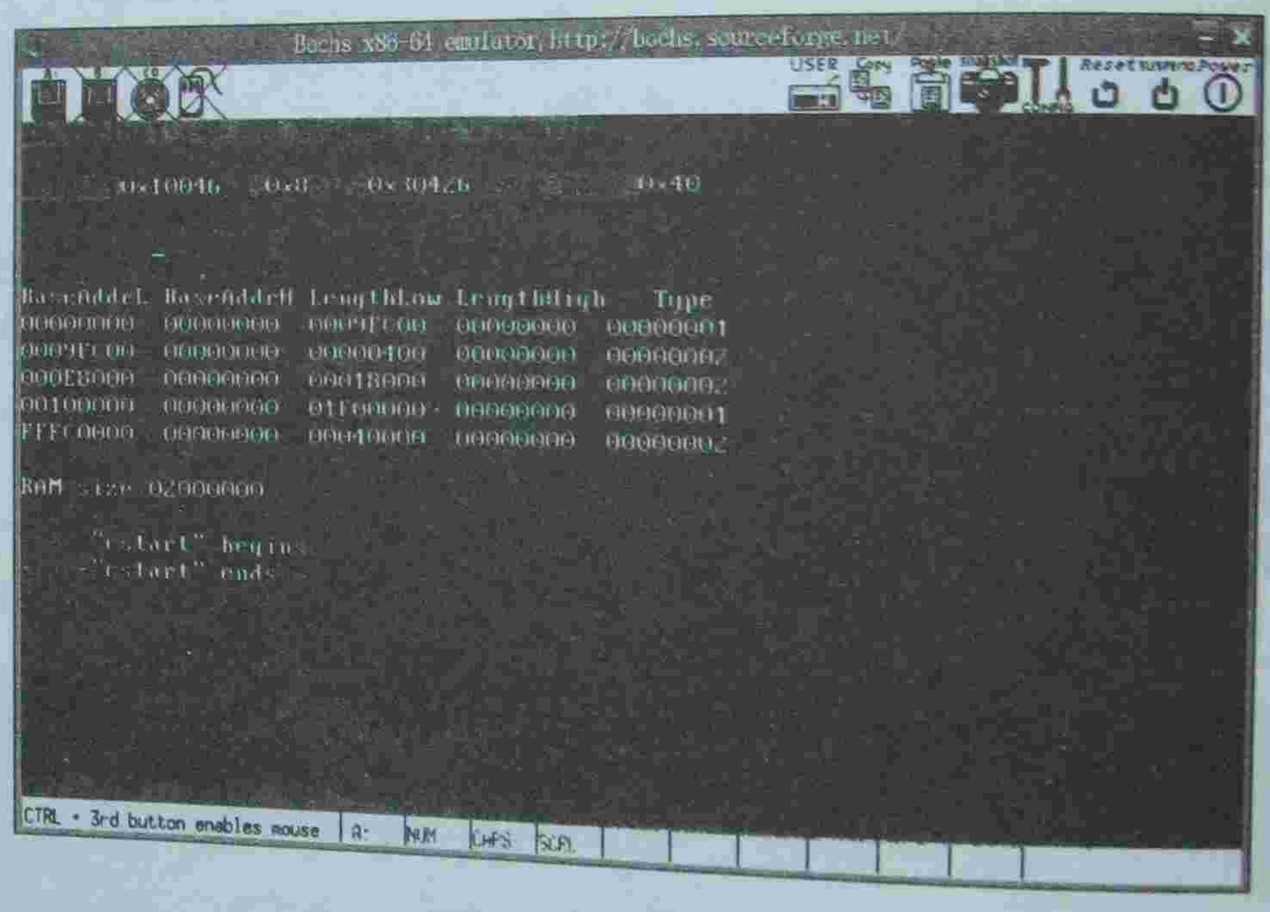


图 5.15 #GP 演示

代码 5.54 对应 8259A 的中断例程（节自 chapter5/i/kernel/kernel.asm）

```
14  extern  spurious_irq
    ...
46  global  hwint00
47  global  hwint01
48  global  hwint02
49  global  hwint03
50  global  hwint04
51  global  hwint05
52  global  hwint06
53  global  hwint07
54  global  hwint08
55  global  hwint09
56  global  hwint10
57  global  hwint11
58  global  hwint12
59  global  hwint13
60  global  hwint14
61  global  hwint15
```

在这里，所有的中断都会触发一个函数spurious_irq()，这个函数的定义如代码5.55所示。

代码 5.55 函数 spurious_irq（节自 chapter5/i/kernel/i8259.c）

```
54  PUBLIC void spurious_irq(int irq)
55  {
56          disp_str("spurious_irq: ");
57          disp_int(irq);
58          disp_str("\n");
59  }
```

spurious_irq()其实什么也不做，仅仅是把IRQ号打印出来而已。下面我们就来设置IDT（代码5.56）。

代码 5.56 设置 IDT（节自 chapter5/i/kernel/protect.c）

```
35  void    hwint00();
36  void    hwint01();
37  void    hwint02();
38  void    hwint03();
39  void    hwint04();
40  void    hwint05();
41  void    hwint06();
42  void    hwint07();
43  void    hwint08();
44  void    hwint09();
45  void    hwint10();
46  void    hwint11();
47  void    hwint12();
48  void    hwint13();
49  void    hwint14();
50  void    hwint15();
51
52  /*======================================================================*
53                               init_prot
54   *======================================================================*/
55  PUBLIC void init_prot()
56  {
     ...
108         init_idt_desc(INT_VECTOR_IRQ0 + 0,      DA_386IGate,
109                       hwint00,                  PRIVILEGE_KRNL);
110
111         init_idt_desc(INT_VECTOR_IRQ0 + 1,      DA_386IGate,
112                       hwint01,                  PRIVILEGE_KRNL);
113
114         init_idt_desc(INT_VECTOR_IRQ0 + 2,      DA_386IGate,
115                       hwint02,                  PRIVILEGE_KRNL);
116
117         init_idt_desc(INT_VECTOR_IRQ0 + 3,      DA_386IGate,
118                       hwint03,                  PRIVILEGE_KRNL);
119
120         init_idt_desc(INT_VECTOR_IRQ0 + 4,      DA_386IGate,
121                       hwint04,                  PRIVILEGE_KRNL);
122
123         init_idt_desc(INT_VECTOR_IRQ0 + 5,      DA_386IGate,
124                       hwint05,                  PRIVILEGE_KRNL);
125
126         init_idt_desc(INT_VECTOR_IRQ0 + 6,      DA_386IGate,
127                       hwint06,                  PRIVILEGE_KRNL);
128
129         init_idt_desc(INT_VECTOR_IRQ0 + 7,      DA_386IGate,
130                       hwint07,                  PRIVILEGE_KRNL);
131
132         init_idt_desc(INT_VECTOR_IRQ8 + 0,      DA_386IGate,
```

```
                hwint08,                                PRIVILEGE_KRNL);

        init_idt_desc(INT_VECTOR_IRQ8 + 1,      DA_386IGate,
                hwint09,                                PRIVILEGE_KRNL);

        init_idt_desc(INT_VECTOR_IRQ8 + 2,      DA_386IGate,
                hwint10,                                PRIVILEGE_KRNL);

        init_idt_desc(INT_VECTOR_IRQ8 + 3,      DA_386IGate,
                hwint11,                                PRIVILEGE_KRNL);

        init_idt_desc(INT_VECTOR_IRQ8 + 4,      DA_386IGate,
                hwint12,                                PRIVILEGE_KRNL);

        init_idt_desc(INT_VECTOR_IRQ8 + 5,      DA_386IGate,
                hwint13,                                PRIVILEGE_KRNL);

        init_idt_desc(INT_VECTOR_IRQ8 + 6,      DA_386IGate,
                hwint14,                                PRIVILEGE_KRNL);

        init_idt_desc(INT_VECTOR_IRQ8 + 7,      DA_386IGate,
                hwint15,                                PRIVILEGE_KRNL);
}
```

其实，现在已经可以make并运行了，但是不会有什么效果，因为我们不但没有通过任何方式设置IF位，而且在init_8259A()中把所有中断都屏蔽掉了。

那么，我们先来到i8259.c处做代码5.57这样的修改：

代码5.57 打开键盘中断（节自chapter5/i/kernel/i8259.c）

```
        /* Master 8259, OCW1.  */
        out_byte(INT_M_CTLMASK, 0xFD);

        /* Slave  8259, OCW1.  */
        out_byte(INT_S_CTLMASK, 0xFF);
}
```

在这里，我们向主8259A相应端口写入了0xFD，由于0xFD对应的二进制是11111101，于是键盘中断被打开，而其他中断仍然处于屏蔽状态。最后，在kernel.asm中添加sti指令设置IF位（代码5.58）：

代码5.58 置IF位（节自chapter5/i/kernel/kernel.asm）

```
csinit:
        sti
        hlt
```

make，运行，开始没有什么特殊的现象，但当我们敲击键盘的任意键时，字符串“spurious_irq: 0x1”就出现了，这表明当前的IRQ号为1，正是对应的键盘中断，如图5.16所示。

5.5.5 两点说明

1. 我们多次用到disp_str()、disp_color_str()、disp_int()等函数。如果你读过Minix或者Linux的代码的话，可能发现在它们的代码中并没有与之相类似的函数。这些函数既不强大也不唯美，为什么我们还要

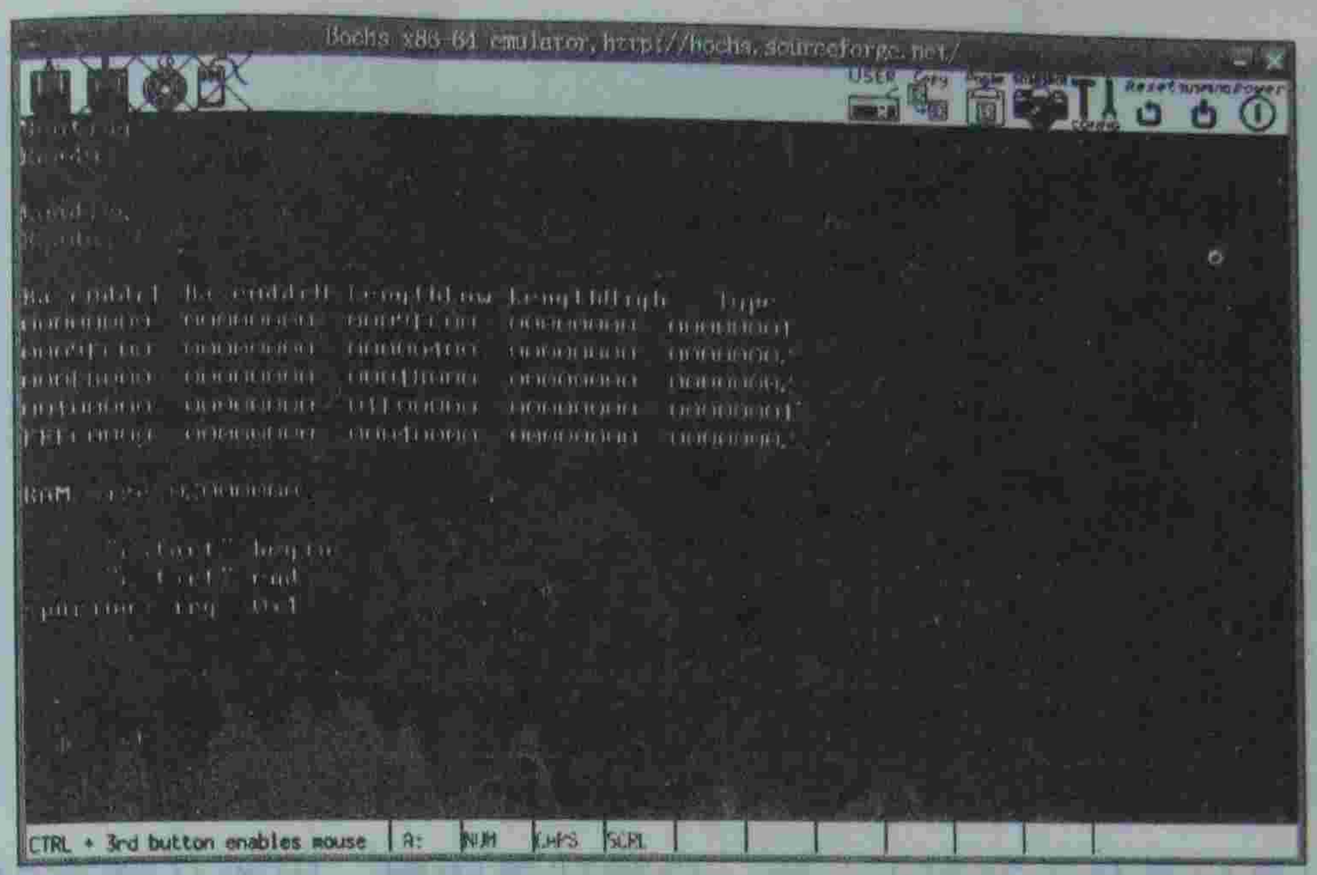


图 5.16 键盘中断发生

用它们呢？其实在可预见的将来，当我们的控制台等模块完善之时，也想写一个漂亮的printf()，但是目前来看，好像还比较遥远。但是，我们又不能没有一个打印函数，所以就先将就着用它们吧。

2. 在上面的代码中，有一些简单明了的符号声明、导入，以及对Makefile的修改等内容并没有列在书中，建议读者在读书的过程中同时参考附书光盘中相应的代码。

5.6 小结

有时，当你埋头走路的时候，猛然回头，发现自己居然已经走出这么远了。我猜你看了下面的目录树之后可能会有这样的感觉。

```
▷ tree
.
|-- Makefile
|-- a.img
|-- bochsrc
|-- boot
|   |-- boot.asm
|   |-- include
|   |   |-- fat12hdr.inc
|   |   |-- load.inc
|   |   `-- pm.inc
|   `-- loader.asm
|-- include
|   |-- const.h
|   |-- global.h
|   |-- protect.h
|   |-- proto.h
```

```
|   |-- string.h
|   `-- type.h
|-- kernel
|   |-- global.c
|   |-- i8259.c
|   |-- kernel.asm
|   |-- protect.c
|   `-- start.c
`-- lib
    |-- klib.c
    |-- kliba.asm
    `-- string.asm
```

我们的目录树又长出了不少，你一定感到很欣慰，因为从结束对保护模式的学习到这里，我们的进展真的是不曾想像过的快。想到这你的信心一定比以往任何时候都要足，因为你知道，拥有的不仅是一个由这样的目录树构成的内核雏形，而且我们还知道如何调试，甚至于我们还学习了ELF文件格式、Makefile的编写等一系列内容。我们已经有了异常处理，设置了8259A并可以接收外部中断，实际上，虽然这个操作系统什么都不能干，但它的潜能已经不再让人质疑。

而且，从编程语言的角度来看，今后我们可能还会用到汇编，但是我们再也不用大段大段用汇编编程，写那些晦涩和容易出错的代码。我们今后会因为C语言的使用而更得心应手，就好比从马车时代进入蒸汽机车时代。

还有一点，笔者觉得前面的工作最好是在连续的时间内完成，因为难懂、难写的汇编语句在工作中断之后难以重新拾起，而且框架的搭建过程也适合一气呵成。所以，在过去的日子中你可能感到有些疲惫。不要紧，现在你可以稍微休息一下，喘口气了，因为接下来的工作是在已经搭建好的框架上完成，并且大部分将用可读性较好的C语言编写，所以即便有些许中断，也不会有很严重的影响。

最困难的日子已经过去，虽然眼前的路仍然很长，但是我们不再感觉是在无边的黑暗中摸索，眼前是一条光明大道，等待我们踏入新的征程。

An expert is a man who has stopped thinking – he knows!

—— Frank Lloyd Wright

6 进程

进程是操作系统中最重要的概念之一，实际上，我们的工作成果在实现进程之前是不能被称做“操作系统”的。进程是一个比较复杂的概念，读者从下文中可以看到，即便是最简单的进程雏形，仍然需要考虑很多的因素。所以本章的开头部分节奏有些慢，希望读者也能以较慢的节奏来阅读，从而可以获得更全面、细致的认识。

6.1 迟到的进程

通常，操作系统教科书都是从进程开始讲解的，可是本书却到现在才将它请出，可谓姗姗来迟。那么，为什么那些教科书不愿意讲述前面这些内容呢？很大一个原因是由于这些内容太过底层，以至于在不同的机器上实现起来完全不同。

由于本书旨在实践，所以肯定不会脱离具体的平台。而且在本章中读者可以看到，进程的切换及调度等内容是和保护模式的相关技术紧密相连的，这些代码量可能并不多，但却至关重要。读者只有了解了它们，才可以彻底地理解进程的运转过程。对于进程的概念，只有在有了基于具体平台的感性认识之后，才有可能对形而上的理论有更踏实的理解。

所以笔者认为，对于想深入了解操作系统的读者，至少接触一种平台上的具体实现是很有必要的，比如我们讲到的最普及的 IBM PC。毕竟，没有实践的理论便如海市蜃楼，美丽，却永远难以看清。而且，有了一种机型的经验，不但有利于在学习理论时形成形象思维，更有触类旁通的能力，面对任何类型的机器和操作系统都能成竹在胸。

6.2 概述

在继续之前，我们需要对进程有一个大概的认识。下面我们首先大致看一下进程是什么，并在开始前对将来要做的工作来一点猜想和瞻望。

6.2.1 进程介绍

在我们的例子中，本书不打算将太多的特性加入到进程中来，那样容易让人陷入过多的细节中，并不利于入门。我们不妨把系统中运行的若干进程想像成一个人在一天内要做的若干样工作：总体来看，每样工作相对独立，并可产生某种结果；从细节上看，每样工作都具有自己的方法、工具和需要的资源；从时间上看，每一个时刻只能有一项工作正在处理中（所谓一心不能二用），各项工作可以轮换来做，这对于最终结果没有影响。

进程与此是类似的，从宏观来看，它有自己的目标，或者说功能，同时又能受控于进程调度模块（类似于工作受控于人）；从微观来看，它可以利用系统的资源，有自己的代码（类似于做事的方法）和数据，同时拥有自己的堆栈（数据和堆栈类似于做事需要的资源和工具）；进程需要被调度，就好比一个人轮换着做不同的工作。进程示意如图6.1所示。

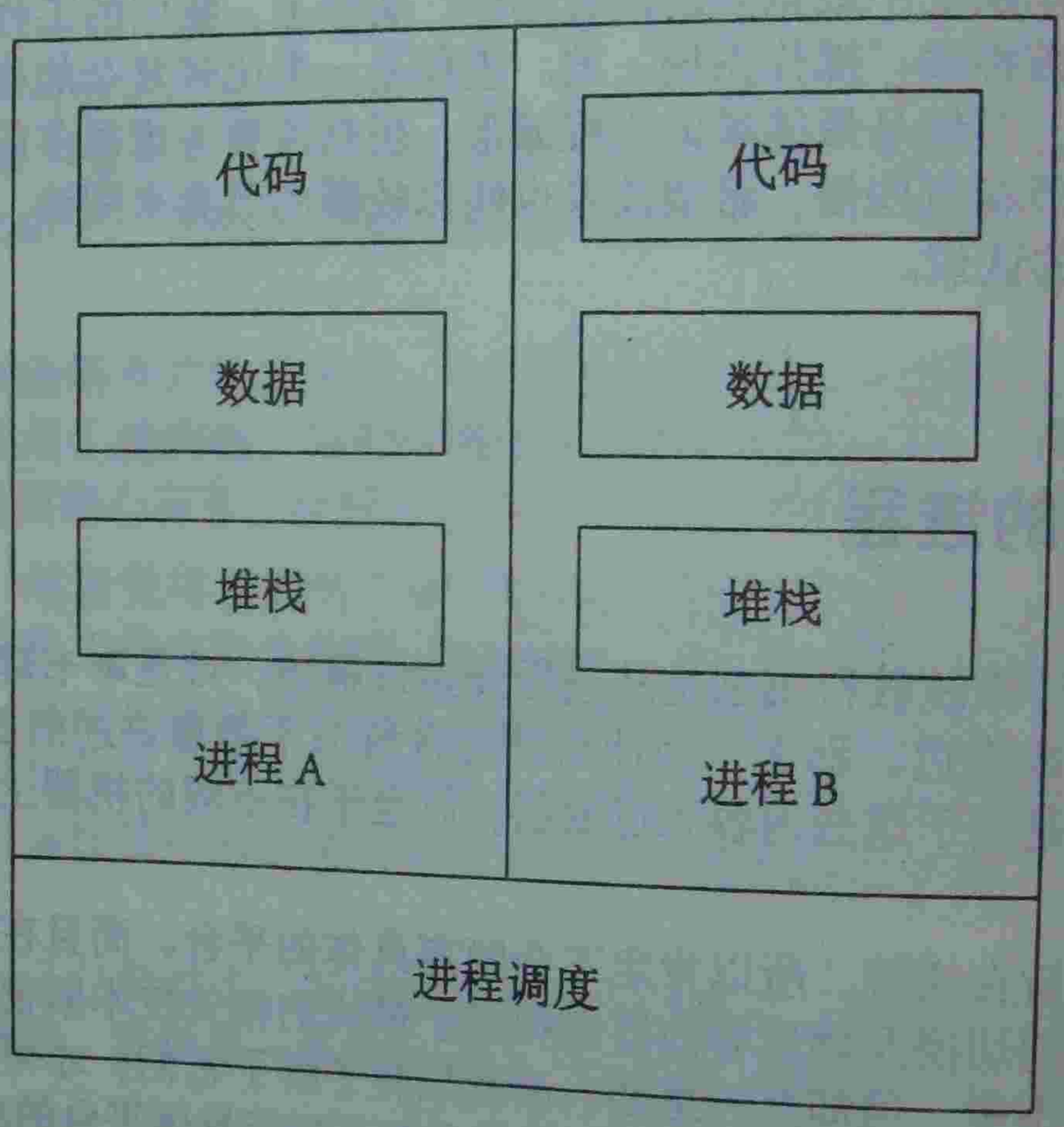


图 6.1 进程示意

我们还是遵循过去的原则，先是形成一个最简陋的进程，然后模仿它再写一个，变成两个。我们试着让它们同时运行，并让我们的系统试着对它们进行调度，当然，使用的是最简单的调度算法。最后，再试着扩展进程的功能。

6.2.2 未雨绸缪——形成进程的必要考虑

你可能会这样想，进程不就是一块或大或小的代码吗，应该很简单吧，随便写几句，想要执行它的时候跳转过去不就行了吗。可是我要提醒你，我们将面对一个无法避免的麻烦，那就是进程调度（这个问题在前面章节中我们已经稍有提及）。将来我们会有很多个进程，它们看上去就好像在同时运行，但是我们知道，CPU 只有一个——我们只考虑单 CPU 系统，实际上哪怕我们有多个

CPU，我们也不能每增加一个进程就增加一个 CPU。也就是说，CPU 的个数通常总是小于进程的个数，于是在同一时刻，总是有“正在运行的”和“正在休息的”进程。所以，对于“正在休息的”进程，我们需要让它在重新醒来时记住自己挂起之前的状态，以便让原来的任务继续执行下去。

所以，我们需要一个数据结构记录一个进程的状态，在进程要被挂起的时候，进程信息就被写入这个数据结构，等到进程重新启动的时候，这个信息重新被读出来（见图6.2）。

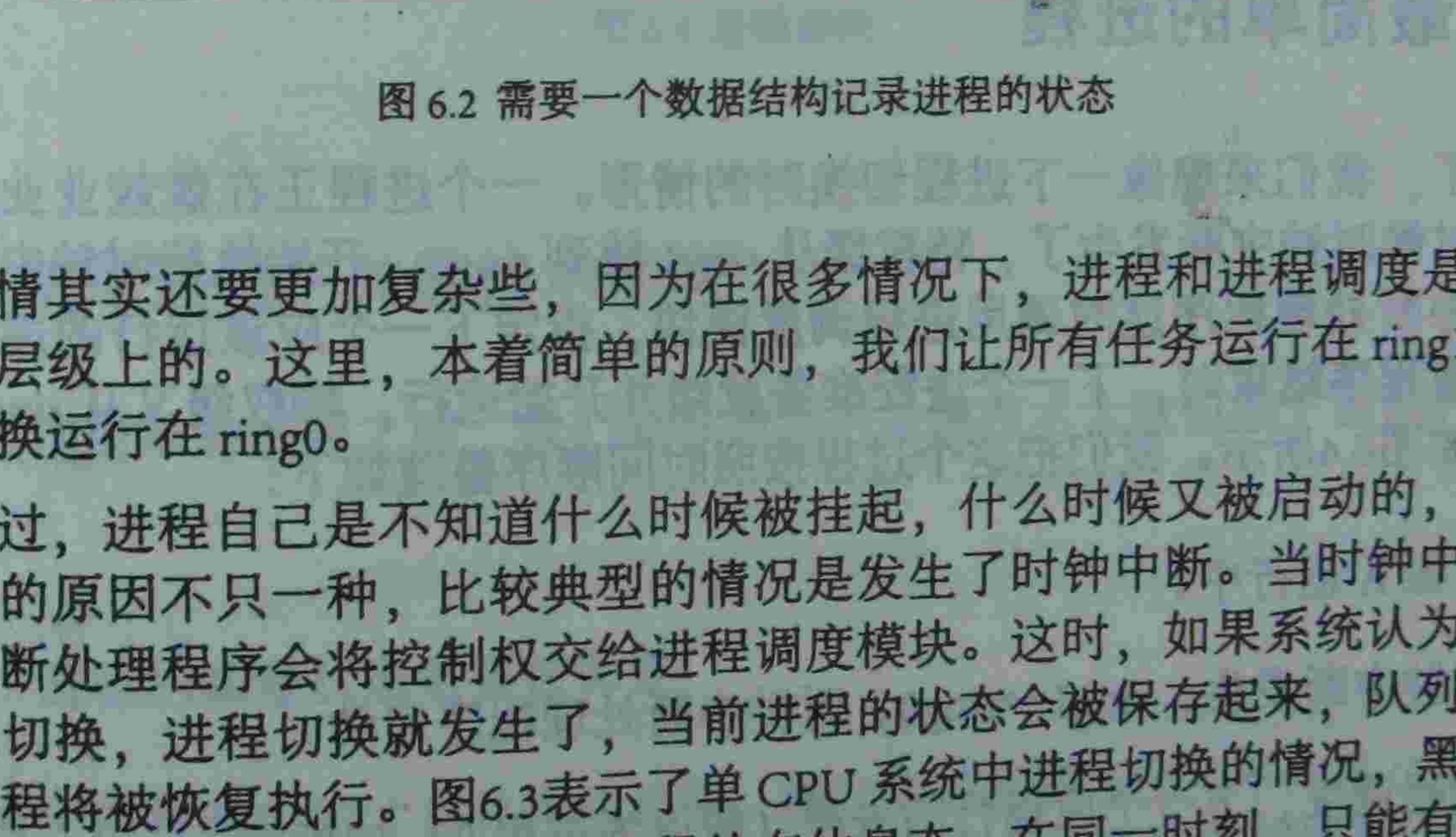


图 6.2 需要一个数据结构记录进程的状态

事情其实还要更加复杂些，因为在很多情况下，进程和进程调度是运行在不同的层级上的。这里，本着简单的原则，我们让所有任务运行在 ring1，而让进程切换运行在 ring0。

不过，进程自己是不知道什么时候被挂起，什么时候又被启动的，诱发进程切换的原因不只一种，比较典型的情况是发生了时钟中断。当时钟中断发生时，中断处理程序会将控制权交给进程调度模块。这时，如果系统认为应该进行进程切换，进程切换就发生了，当前进程的状态会被保存起来，队列中的下一个进程将被恢复执行。图6.3表示了单 CPU 系统中进程切换的情况，黑色条表示进程处在运行态，白色条表示进程处在休息态。在同一时刻，只能有一个进程处在运行态。进程切换的操作者是操作系统的进程调度模块。

这里要说明的一点是，并非在每一次时钟中断时都一定会发生进程切换，不过为了容易理解和实现，我们在6.6节之前暂时让每次非重入的（关于中断重入下文中会有详细介绍）中断都切换一次进程。

6.2.3 参考的代码

我或许没有提到过，笔者写自己的操作系统的初衷，本来是要弄懂 Minix。所以，其中的很多代码是向 Minix 学习的结果，我们的进程就是在它的基础上进行的简化。如果你也打算阅读这本经典著作，本书可以起到敲门砖的作用，并加速你理解 Minix 进程管理的过程。不过，如果你不愿意，你完全可以不阅读 Minix 的代码，Orange'S的进程实现要简单得多，自成体系，而且本书给出了

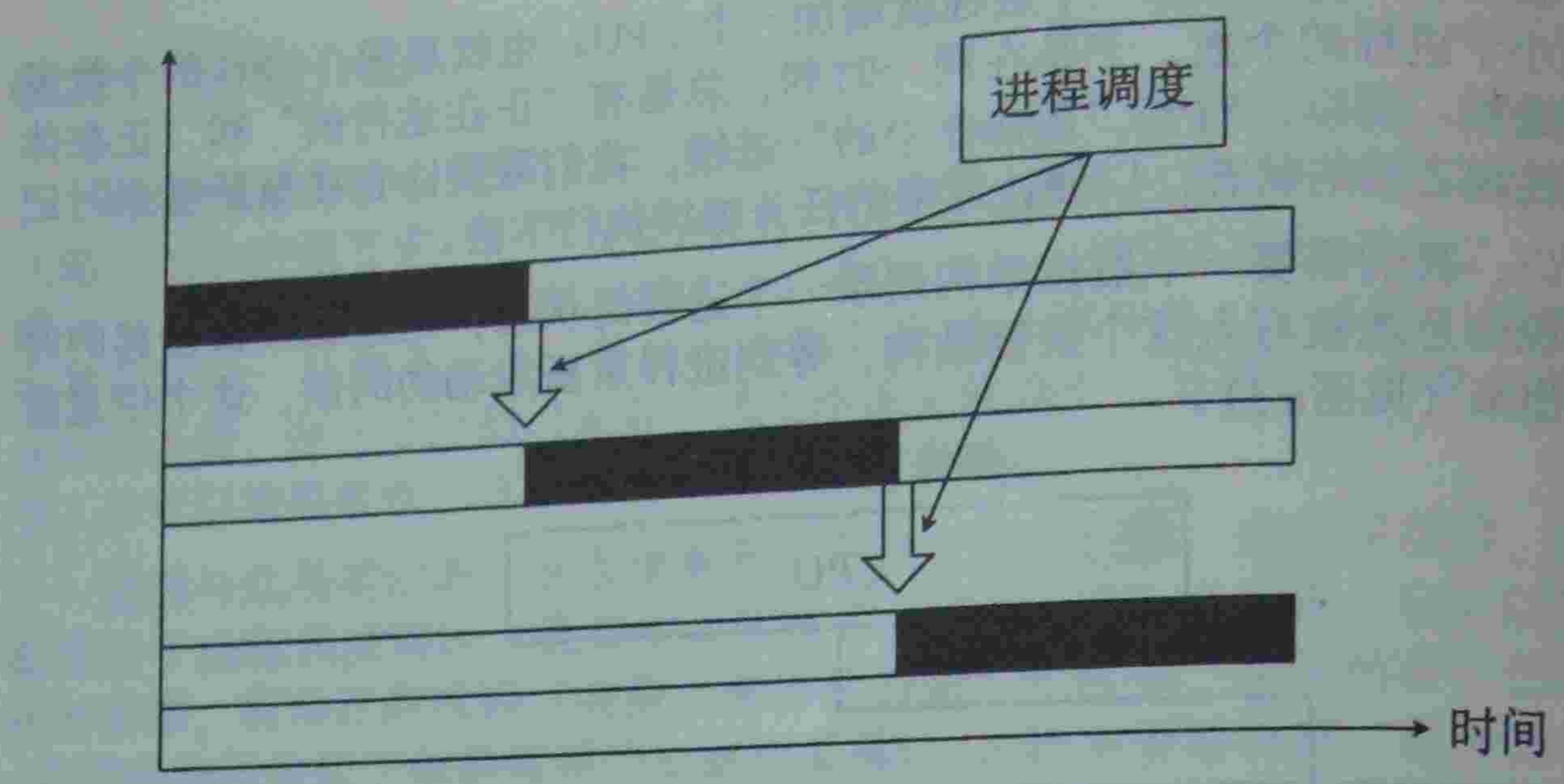


图 6.3 进程切换

所有细节上的说明，这些说明并不依赖于 Minix。

6.3 最简单的进程

好了，我们来想像一下进程切换时的情形。一个进程正在兢兢业业地运行着，这时候时钟中断发生了，特权级从 ring1 跳到 ring0，开始执行时钟中断处理程序，中断处理程序这时调用进程调度模块，指定下一个应该运行的进程，当中断处理程序结束时，下一个进程准备就绪并开始运行，特权级又从 ring0 跳回 ring1，如图6.4所示。我们把这个过程按照时间顺序整理如下：

1. 进程 A 运行中。
2. 时钟中断发生，ring1→ring0，时钟中断处理程序启动。
3. 进程调度，下一个应运行的进程（假设为进程 B）被指定。
4. 进程 B 被恢复，ring0→ring1。
5. 进程 B 运行中。

要想实现这些功能，我们必须完成的应该有以下几项：

- 时钟中断处理程序
- 进程调度模块
- 两个进程

内容看上去并不怎么多，我们先来分析一下，以进程 A 到进程 B 切换为例，其中有哪些关键技术需要解决。然后用代码分别实现这几个部分。

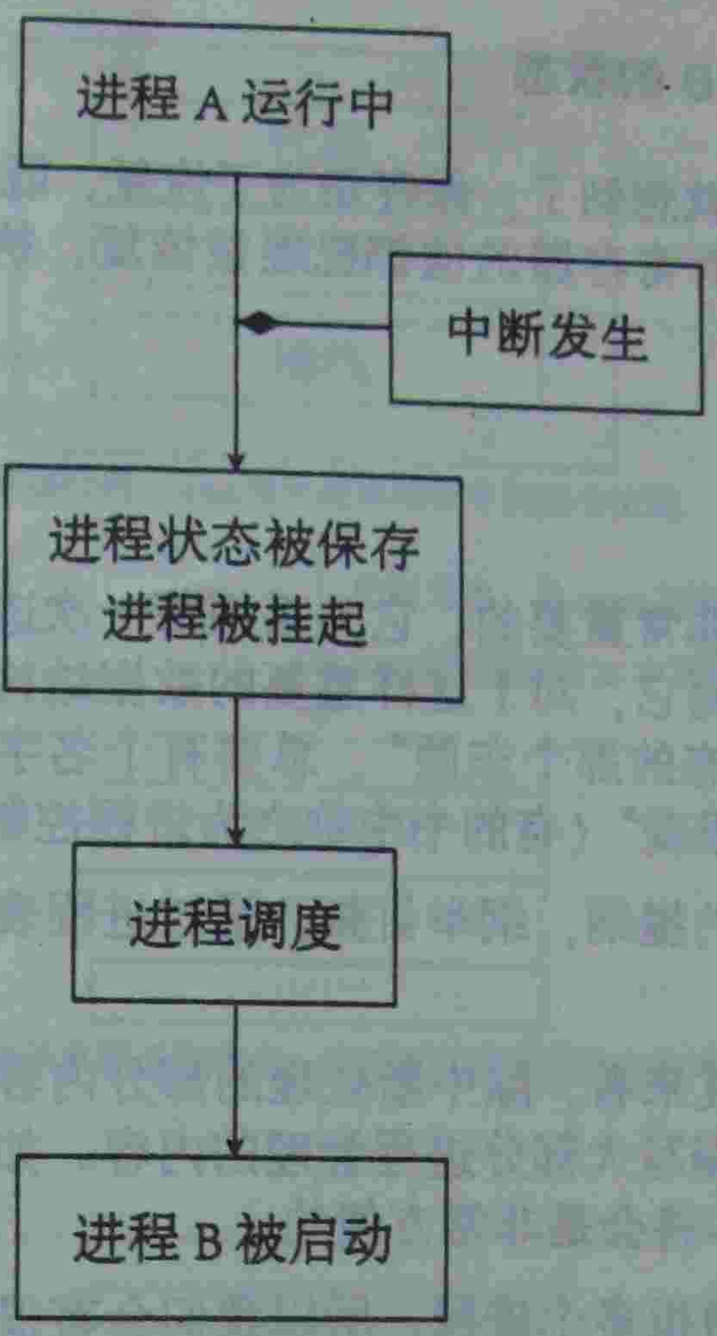


图 6.4 进程切换

6.3.1 简单进程的关键技术预测

在实现简单的进程之前，我们能够想到的关键技术大致包括下面的内容。

6.3.1.1 进程的哪些状态需要被保存

只有可能被改变的才有保存的必要。我们的进程要运行，不外乎 CPU 和内存在相互协作，而不同进程的内存互不干涉（我们考虑最简单的情况，假设内存足够大），但是我们提到过，CPU 只有一个，不同进程共用一个 CPU 的一套寄存器。所以，我们要把寄存器的值统统保存起来，准备进程被恢复执行时使用。

6.3.1.2 进程的状态需要何时以及怎样被保存

为了保证进程状态完整，不被破坏，我们当然希望在进程刚刚被挂起时保存所有寄存器的值。你一定在想，保存寄存器我已经很拿手，push就可以了，没错，push就够了。不过，Intel 想得更周到，不但有push，更有pushad，一条指令可以保存许多寄存器值。而这些代码，我们应该把它写在时钟中断例程的最顶端，以便中断发生时马上被执行。

6.3.1.3 如何恢复进程 B 的状态

不用说，你一定早就想到了，保存是为了恢复，既然保存用的是push，恢复一定用pop了。等所有寄存器的值都已经被恢复，执行指令iretd，就回到了进程 B。

6.3.1.4 进程表的引入

进程的状态无疑是非常重要的，它关系到每一次进程挂起和恢复。可以预见，我们今后将多次提到它，对于这样重要的数据结构，我们总不能在每次提到时叫它“保存进程状态的那个东西”，总要有个名字。还好，前人已经替我们起过了，那就是“进程表”(有的书中称之为进程控制块，也即 PCB)。

进程表相当于进程的提纲，纲举目张，通过进程表，我们可以非常方便地进行进程管理。

从代码编写这个角度来看，除中断处理的部分内容我们不得不使用汇编之外，我们还是要用 C 来编写大部分进程管理的内容。如果把进程表定义成一个结构体的话，对它的操作将会是非常方便的。

毫无疑问，我们会有很多个进程，所以我们会有很多个进程表，形成一个进程表数组。进程表数组如图6.5所示。

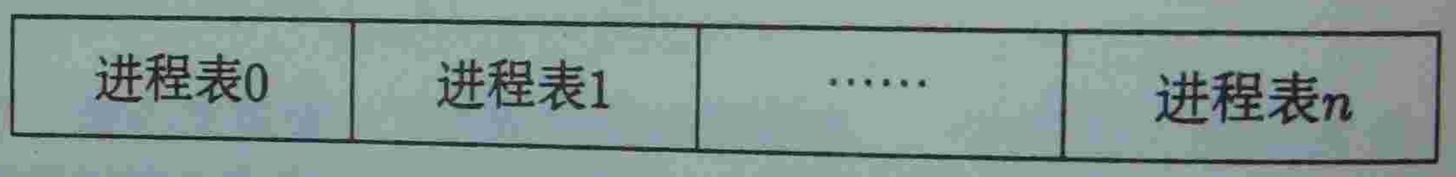


图 6.5 进程表数组

进程表是用来描述进程的，所以它必须独立于进程之外。所以，当我们把寄存器值压到进程表内的时候，已经处在进程管理模块中了。

6.3.1.5 进程栈和内核栈

当寄存器的值已经被保存到进程表内，进程调度模块就开始执行了。但这时有一个很重要的问题容易被忽视，就是esp现在指向何处。

毫无疑问，我们在进程调度模块中会用到堆栈，而寄存器被压到进程表之后，esp是指向进程表某个位置的。这就有了问题，如果接下来进行任何的堆栈操作，都会破坏掉进程表的值，从而在下一次进程恢复时产生严重的错误。

为解决这个问题，避免错误的出现，一定要记得将esp指向专门的内核栈区域。这样，在短短的进程切换过程中，esp的位置出现在 3 个不同的区域(图6.6是整个过程的示意)。

其中：

- 进程栈——进程运行时自身的堆栈。
- 进程表——存储进程状态信息的数据结构。

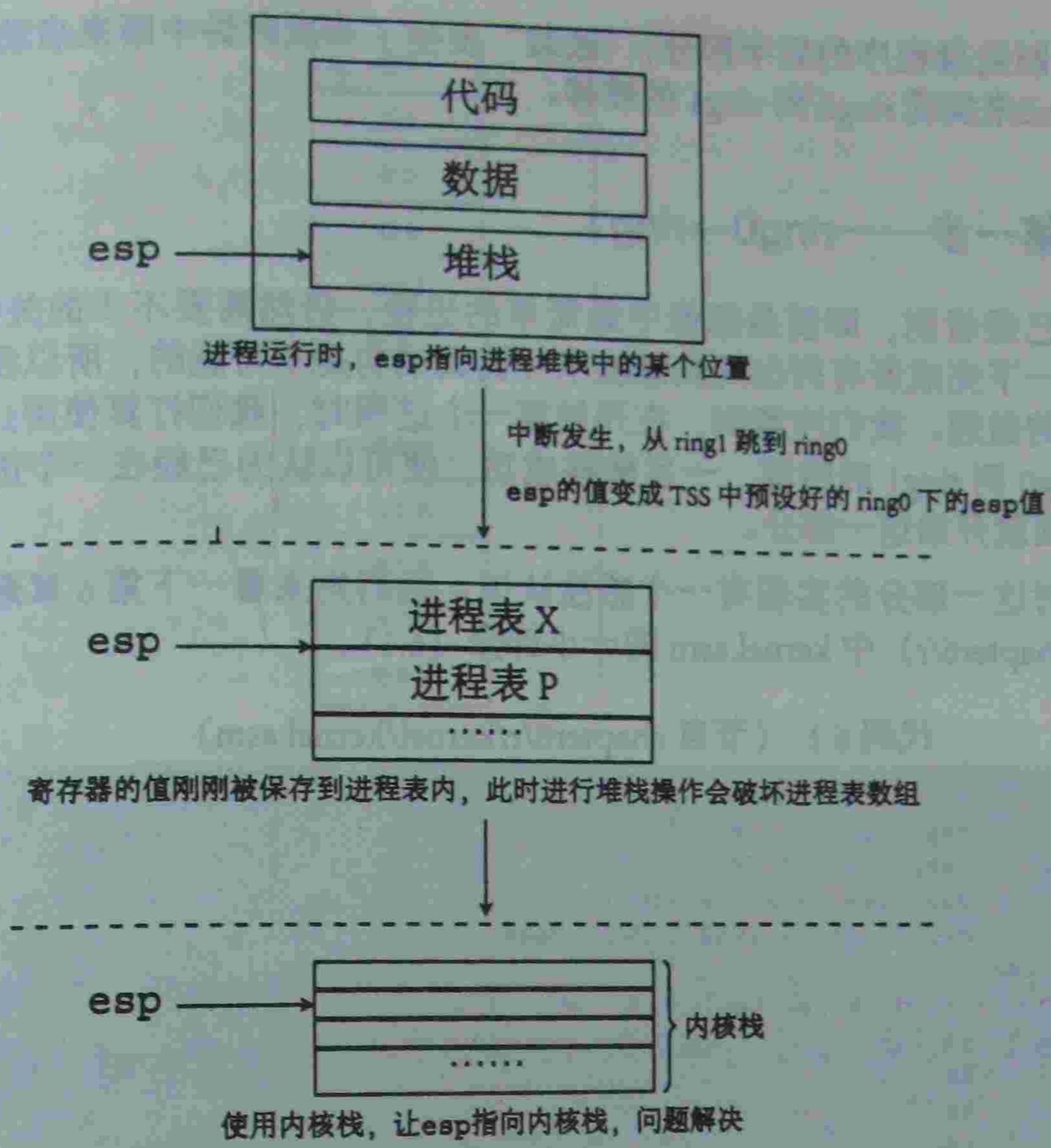


图 6.6 内核栈

- 内核栈——进程调度模块运行时使用的堆栈。

在具体编写代码的过程中，一定要清楚当前使用的是哪个堆栈，以免破坏掉不应破坏的数据。

6.3.1.6 特权级变换：ring1→ring0

在我们以前的代码中，还没有使用过除 ring0 之外的其他特权级。你应该还记得，对于有特权级变换的转移，如果由外层向内层转移时，需要从 TSS 中取得从当前 TSS 中取出内层ss和esp作为目标代码的ss和esp。所以，我们必须事先准备好 TSS。由于每个进程相对独立，我们把涉及到的描述符放在局部描述符表 LDT 中，所以，我们还需要为每个进程准备 LDT。

6.3.1.7 特权级变换：ring0→ring1

在我们刚才的分析过程中，我们假设的初始状态是“进程 A 运行中”。可是我们知道，到目前为止我们的代码完全运行在 ring0。所以，可以预见，当我们准备开始第一个进程时，我们面临一个从 ring0 到 ring1 的转移，并启动进程 A。这跟我们从进程 B 恢复的情形很相似，所以我们完全可以在准备就绪之后

跳转到中断处理程序的后半部分，“假装”发生了一次时钟中断来启动进程A，利用iretd来实现ring0到ring1的转移。

6.3.2 第一步——ring0→ring1

我们已经看到，即便是想像中最简单的进程，仍然需要不少的关键技术。而且，要一下完成所有列出的关键技术并调试成功是不可能的，所以我们还是从最容易的做起。我们注意到，在开始第一个进程时，我们打算使用iretd来实现由ring0到ring1的转移，一旦转移成功，便可以认为已经在一个进程中运行了。下面就开始这一部分。

为了对这一部分的实现有一个感性认识，我们先来看一下第6章最终实现的代码（chapter6/r）中kernel.asm的一小部分（6.1）。

代码6.1 （节自chapter6/r/kernel/kernel.asm）

```
357 restart:
358         mov     esp, [p_proc_ready]
359         lldt    [esp + P_LDT_SEL]
360         lea     eax, [esp + P_STACKTOP]
361         mov     dword [tss + TSS3_S_SP0], eax
362 restart_reenter:
363         dec     dword [k_reenter]
364         pop     gs
365         pop     fs
366         pop     es
367         pop     ds
368         popad
369         add     esp, 4
370         iretd
```

为了容易理解，先来看一看本章所附的代码的部分内容。因为进程毕竟是个新鲜事物，它涉及若干方面，如果一开始就下手行动，很可能会无所适从。

在/kernel中你会发现多了一个main.c，里面有个函数kernel_main()，从中可以找到这样一行：

```
restart();
```

它调用的便是代码6.1这一段，它是进程调度的一部分，同时也是我们的操作系统启动第一个进程时的入口。

第358行设置了esp的值，而在下方不远处就是若干个pop以及一个popad指令。结合过去的分析我们不难推断，p_proc_ready应该是一个指向进程表的指针，存放的便是下一个要启动进程的进程表的地址。而且，其中的内容必然是以图6.7所示的顺序进行存放。这样，才会使pop和popad指令执行后各寄存器的内容更新一遍。

我们来验证一下。在头文件global.h中可以找到p_proc_ready类型是一个结构类型指针：struct s_proc*。再打开proc.h，可以看到s_proc这个结构体的第一个成员也是一个结构，叫做s_stackframe。顺藤摸瓜，我们找到s_stackframe这个结构体的声明，它的内容安排与我们的推断完全一致。

现在我们知道了，原来进程的状态统统被存放在s_proc这个结构体中，而且位于前部的是所有相关寄存器的值，s_proc这个结构就应该是我们提到过

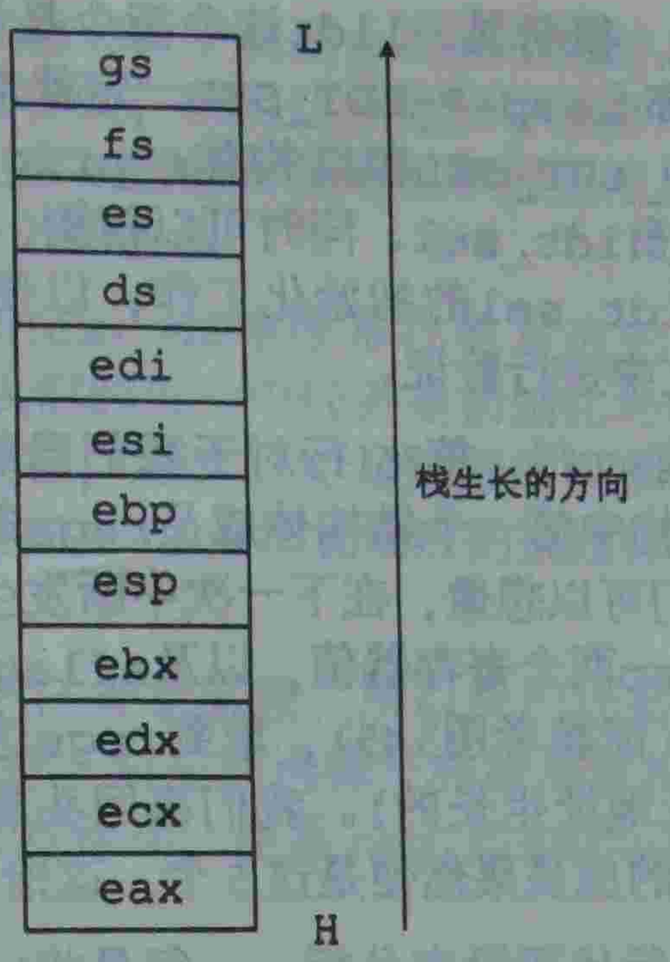


图 6.7 进程表推想

的“进程表”。当要恢复一个进程时，便将esp指向这个结构体的开始处，然后运行一系列的pop命令将寄存器值弹出。进程表的开始位置结构图示如图6.8所示。

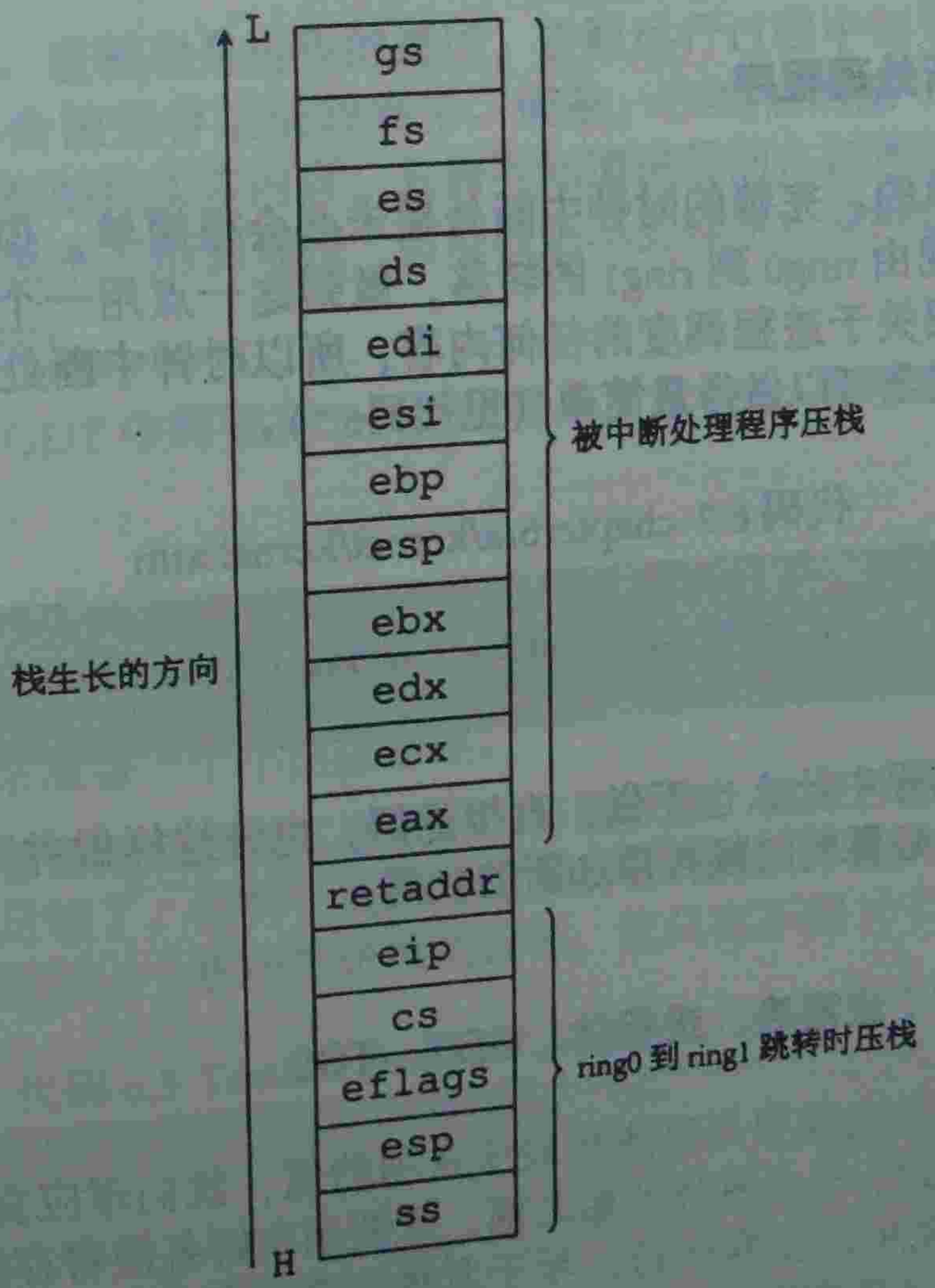


图 6.8 进程表的开始位置结构图示

我们再来看第359行，很明显，lldt这个指令是设置ldtr的。既然esp等同于p_proc_ready，那么esp+P_LDT_SEL一定就是s_proc的一个成员，我们通过对比sconst.inc中P_LDT_SEL和结构体s_proc可知，esp+P_LDT_SEL恰好就是s_proc中的成员ldt_sel。同时可以猜测，在执行restart()之前，在某个地方一定是做了ldt_sel的初始化工作，以便lldt可以正确执行。对于这一点，我们留到下文中进行验证。

经过上面的分析，第360行、第361行对于我们已经很容易理解了，它们的作用是将s_proc这个结构中第一个结构体成员regs的末地址赋给 TSS 中 ring0 堆栈指针域（esp）。我们可以想像，在下一次中断发生时，esp将变成regs的末地址，然后进程ss和esp两个寄存器值，以及eflags，还有cs、eip这几个寄存器值将依次被压栈（请参考图3.45），放到regs这个结构的最后面（不要忘记堆栈是从高地址向低地址生长的）。我们再回头看s_stackframe这个结构的定义时，发现最末端的成员果然便是这 5 个，这恰好验证了我们的想法。

至此，我们只剩下两行代码没有分析，一行是将k_reenter的值减 1，而另一行则是将esp加 4。结合s_stackframe的结构定义不难发现，其实esp加 4 恰好跳过了retaddr这个成员，以便执行iretd这个指令，之前堆栈内恰好是eip、cs、eflags、esp和ss的值。那么，究竟retaddr是用来干什么的呢？k_reenter这个变量又起什么作用呢？我们留到后面慢慢说明。

这段代码我们基本上已经弄明白了，对于进程我们也已经有了一定程度的感性认识。你一定还记得在第6.3节中我们说过，要想实现进程必须完成这几项：时钟中断处理程序、进程调度模块和进程体。我们就依次来做这些工作。

6.3.2.1 时钟中断处理程序

先来做最简单的。完善的时钟中断处理未必会很简单，但前面说过，我们打算且只打算实现由 ring0 到 ring1 的转移，做到这一点用一个iretd指令就够了。此时并不需要关于进程调度的任何内容，所以时钟中断处理程序在这一步并不重要，我们完全可以做得最简单（见代码6.2）。

代码 6.2 chapter6/a/kernel/kernel.asm

```
150 ALIGN   16
151 hwint00:
152         iretd                   ; Interrupt routine for irq 0 (the clock).
```

在这段中断例程中什么也不做，直接返回，也许这样做并不好，暂且不管它，等到我们觉得必要的时候再添加新的代码。

6.3.2.2 化整为零：进程表、进程体、GDT、TSS

既然在进程开始之前要用到进程表中各项的值，我们理应首先将这些值进行初始化。不难想到，一个进程开始之前，只要指定好各段寄存器、eip、esp以及eflags，它就可以正常运行，至于其他寄存器是用不到的，所以我们得出这样的必须初始化的寄存器列表：cs、ds、es、fs、gs、ss、esp、eip、eflags。

你大概还记得，我们在 Loader 中就把 gs 对应的描述符 DPL 设为 3，所以进程中的代码是有权限访问显存的；我们让其他段寄存器对应的描述符基地址和段界限与先前的段寄存器对应的描述符基地址和段界限相同，只是改变它们的 RPL 和 TI，以表示它们运行的特权级。

值得注意的是，这里的cs、ds等段寄存器对应的将是 LDT 中而不再是 GDT 中的描述符。所以，我们的另一个任务是初始化局部描述符表。可以把它放置在进程表中，从逻辑上看，由于 LDT 是进程的一部分，所以这样安排也是合理的。同时，我们还必须在 GDT 中增加相应的描述符，并在合适的时间将相应的选择子加载给ldtr。

另外，由于我们用到了任务状态段，所以我们还必须初始化一个 TSS，并且在 GDT 中添加一个描述符，对应的选择子将被加载给tr这个寄存器。其实，TSS 中我们所有能用到的只有两项，便是 ring0 的ss和esp，所以我们只需要初始化它们两个就够了。

在第一个进程正式开始之前，我们的准备工作已经做得差不多了，其核心内容便是一个进程表以及与之相关的 TSS 等内容。它们之间的对应关系如图6.9所示。

这个图看起来有点复杂，但是如果将其化整为零，可以分为 4 个部分，那就是进程表、进程体、GDT 和 TSS。它们之间的关系大致分为三个部分：

1. 进程表和 GDT。进程表内的 LDT Selector 对应 GDT 中的一个描述符，而这个描述符所指向的内存空间就存在于进程表内。

2. 进程表和进程。进程表是进程的描述，进程运行过程中如果被中断，各个寄存器的值都会被保存进进程表中。但是，在我们的第一个进程开始之前，并不需要初始化太多内容，只需要知道进程的入口地址就足够了。另外，由于程序免不了用到堆栈，而堆栈是不受程序本身控制的，所以还需要事先指定esp。

3. GDT 和 TSS。GDT 中需要有一个描述符来对应 TSS，需要事先初始化这个描述符。

好了，这 4 个部分的相互关系读者应该已经弄清楚了，那么现在，就让我们分别来做这 4 个部分的初始化工作。

第一步，首先来准备一个小的进程体。

虽然你身上穿着经过无数道工序做出的华丽衣装，但是若想遮蔽风寒的话，一张兽皮其实已经足够了。同样，我们知道以后会有无所不能的进程，但此刻，并不需要，我们只需要一个极小的进程执行体，它只有不到 10 行（代码6.3）。

代码 6.3 TestA（chapter6/a/kernel/main.c）

```
51 void TestA()
52 {
53
54         int i = 0;
55         while(1){
56                 disp_str("A");
57                 disp_int(i++);
58                 disp_str(".");
                   delay(1);
```

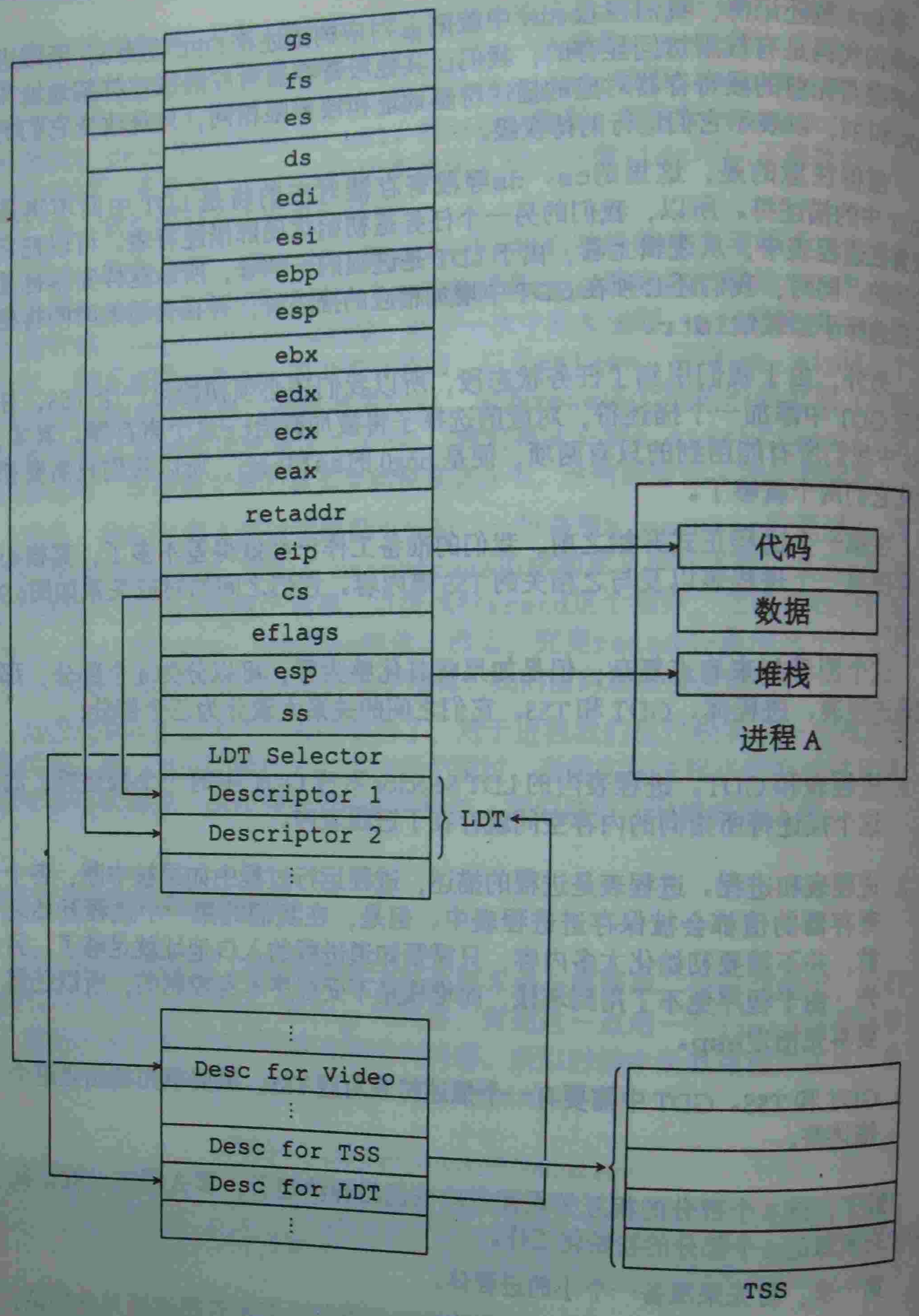


图 6.9 进程表及相关数据结构对应关系示意

```
59              }
60      }
```

看到这个“进程”，你会说，这是个函数。是的，不但是个函数，而且是个极其简单的函数，但已经可以满足它作为一个进程执行体的功能。在它执行时会不停地循环，每循环一次就打印一个字符和一个数字，并且稍停片刻。

我们希望进程开始运行时能看到屏幕上打印出源源不断的“A”——那无

疑将是个激动人心的时刻。

注意，这段代码被放置在main.c这个文件中。我们上文提到过，这是个新建立的文件，在第5章的最后，我们调用指令sti打开中断之后就用hlt指令让程序停止以等待中断的发生。显然，在本章中，我们将最终让进程运行起来，而不能仅仅停在那里，所以程序需要继续进行下去。我们将hlt注释掉，并让程序跳转到kernel_main()这个函数中，见代码6.4，这个函数放在main.c中，目前除了显示一行字符之外并不完成其他工作。不过，由于在完成进程的编写之前，要让程序停在这里，所以我们用一个死循环作为它的结束。

代码 6.4 函数 kernel_main（chapter6/a/kernel/main.c）

```
20  PUBLIC int kernel_main()
21  {
22          disp_str("-----\"kernel_main\" begins-----\n");
 ...
45          while(1){}
46  }
```

在kernel.asm的最后，我们跳转到kernel_main()中（代码6.5）。

代码 6.5 chapter6/a/kernel/kernel.asm

```
13  extern  kernel_main
 ...
134         ;sti
135         jmp     kernel_main
```

进程A中的函数delay()我们也让它尽量简单，写一个循环（代码6.6）。

代码 6.6 chapter6/a/lib/klib.c

```
66  PUBLIC void delay(int time)
67  {
68          int i, j, k;
69          for (k = 0; k < time; k++) {
70                  for (i = 0; i < 10; i++) {
71                          for (j = 0; j < 10000; j++) {}
72                  }
73          }
74  }
```

运行的时候，如果发现两次打印之间的间隔不理想，可以调整这里循环的次数。

第二步，初始化进程表。

要初始化进程表，首先要有进程表结构的定义，如代码6.8所示。其中，结构体STACK_FRAME的定义见代码6.7。

代码 6.7 chapter6/a/include/proc.h

```
9   typedef struct s_stackframe {                                   */
10          u32     gs;             /*  \                          */
11          u32     fs;             /*   |                         */
12          u32     es;             /*   |                         */
13          u32     ds;             /*   |                         */
14          u32     edi;            /*   |  pushed by save()       */
15          u32     esi;            /*   |                         */
16          u32     ebp;            /*   |                         */
17          u32     kernel_esp;     /* <- 'popad' will ignore it   */
```

```
        u32     ebx;            /* |                                    */
        u32     edx;            /* |                                    */
        u32     ecx;            /* /                                    */
        u32     eax;            /* return addr for kernel.asm::save()   */
        u32     retaddr;        /* \                                    */
        u32     eip;            /* |                                    */
        u32     cs;             /* | pushed by CPU during interrupt     */
        u32     eflags;         /* |                                    */
        u32     esp;            /* /                                    */
        u32     ss;
}STACK_FRAME;
```

如果完全“自己动手”编写这部分代码的话，显然不会在STACK_FRAME中添加retaddr作为一个成员。不过，我们现在已经看过最终的代码了，知道它将来是有用的，所以不如暂时妥协一下，把它加在里面。

代码 6.8 chapter6/a/include/proc.h

```
typedef struct s_proc {
        STACK_FRAME regs;           /* process registers saved in stack frame */

        u16 ldt_sel;                /* gdt selector giving ldt base and limit */
        DESCRIPTOR ldts[LDT_SIZE]; /* local descriptors for code and data */
        u32 pid;                    /* process id passed in from MM */
        char p_name[16];            /* name of the process */
}PROCESS;
```

现在，结构体的定义有了，我们在global.c中声明一个进程表：

```
PUBLIC PROCESS proc_table[NR_TASKS];
```

其中，NR_TASKS定义了最大允许进程，我们把它设为1。虽然目前只试验一个进程的运行，但为了以后的扩展，我们还是声明了一个数组而不是单独一个变量，当NR_TASKS为1的时候数组和变量是一样的。

好了，进程表有了，我们就来初始化它。在整个过程中我建议你同时对照图6.9，这样有利于理清思路。

由于kernel_main()是最后一部分被执行的代码，那么初始化进程表的代码理应添加在这里。

代码 6.9 初始化进程表（chapter6/a/kernel/main.c）

```
        PROCESS* p_proc = proc_table;

        p_proc->ldt_sel = SELECTOR_LDT_FIRST;
        memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS>>3], sizeof(DESCRIPTOR));
        p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5;     // change the DPL
        memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS>>3], sizeof(DESCRIPTOR));
        p_proc->ldts[1].attr1 = DA_DRW | PRIVILEGE_TASK << 5;   // change the DPL

        p_proc->regs.cs = (0 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
        p_proc->regs.ds = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
        p_proc->regs.es = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
        p_proc->regs.fs = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
        p_proc->regs.ss = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
        p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK) | RPL_TASK;
        p_proc->regs.eip= (u32)TestA;
        p_proc->regs.esp= (u32) task_stack + STACK_SIZE_TOTAL;
        p_proc->regs.eflags = 0x1202;   // IF=1, IOPL=1, bit 2 is always 1.
```

从图6.9或者代码6.8可以看出，进程表需要初始化的主要有3个部分：寄存器、LDT Selector和LDT。明白了这一点，代码6.9就很容易理解了，LDT Selector被赋值为SELECTOR_LDT_FIRST，这个宏的定义在代码6.10中。LDT里面共有两个描述符，为简化起见，分别被初始化成内核代码段和内核数据段，只是改变了一下DPL以让其运行在低的特权级下。

要初始化的寄存器比较多，我们看到，cs指向LDT中第一个描述符，ds、es、fs、ss都设为指向LDT中的第二个描述符，gs仍然指向显存，只是其RPL发生改变。

接下来，eip指向TestA，这表明进程将从TestA的入口地址开始运行。另外，esp指向了单独的栈，栈的大小为STACK_SIZE_TOTAL。

最后一行是设置eflags，结合图3.46可以知道，0x1202恰好设置了IF位并把IOPL设为1。这样，进程就可以使用I/O指令，并且中断会在iretd执行时被打开（kernel.asm中的sti指令已经在代码6.4中被注释掉了）。

代码中用到的宏大部分定义在protect.h中，见代码6.10。

代码6.10 部分宏定义（chapter6/a/include/protect.h）

```
/* GDT */
/* 描述符索引 */
#define INDEX_DUMMY             0       /* \                                */
#define INDEX_FLAT_C            1       /* | LOADER 里面已经确定了的 */
#define INDEX_FLAT_RW           2       /* |                                */
#define INDEX_VIDEO             3       /* /                                */
#define INDEX_TSS               4
#define INDEX_LDT_FIRST         5
/* 选择子 */
#define SELECTOR_DUMMY            0     /* \                                */
#define SELECTOR_FLAT_C         0x08    /* | LOADER 里面已经确定了的 */
#define SELECTOR_FLAT_RW        0x10    /* |                                */
#define SELECTOR_VIDEO          (0x18+3)/* /<-- RPL=3                       */
#define SELECTOR_TSS            0x20    /* TSS                              */
#define SELECTOR_LDT_FIRST      0x28

#define SELECTOR_KERNEL_CS      SELECTOR_FLAT_C
#define SELECTOR_KERNEL_DS      SELECTOR_FLAT_RW
#define SELECTOR_KERNEL_GS      SELECTOR_VIDEO

/* 每个任务有一个单独的 LDT, 每个 LDT 中的描述符个数: */
#define LDT_SIZE                2

/* 选择子类型值说明 */
/* 其中, SA_ : Selector Attribute */
#define SA_RPL_MASK     0xFFFC
#define SA_RPL0         0
#define SA_RPL1         1
#define SA_RPL2         2
#define SA_RPL3         3

#define SA_TI_MASK      0xFFFB
#define SA_TIG          0
#define SA_TIL          4
```

这里不但定义了SELECTOR_LDT_FIRST，而且定义了SELECTOR_TSS，因为从图6.9中可知，我们还需要一个用来使用TSS的描述符。

这里，一定要记得LDT跟GDT是联系在一起的，别忘了填充GDT中进程的LDT的描述符（代码6.11）。

代码 6.11 填充 GDT 中进程 LDT 的描述符（chapter6/a/kernel/protect.c）

```
109         init_descriptor(&gdt[INDEX_LDT_FIRST],
110                         vir2phys(seg2phys(SELECTOR_KERNEL_DS), proc_table[0].ldts),
111                         LDT_SIZE * sizeof(DESCRIPTOR) - 1,
112                         DA_LDT);
```

这段代码放在init_prot()中。init_descriptor和init_idt_desc有些类似（代码6.12）。

代码 6.12 函数 init_descriptor（chapter6/a/kernel/protect.c）

```
149 PRIVATE void init_descriptor(DESCRIPTOR *p_desc,u32 base,u32 limit,u16 attribute)
150 {
151         p_desc->limit_low       = limit & 0x0FFFF;
152         p_desc->base_low        = base & 0x0FFFF;
153         p_desc->base_mid        = (base >> 16) & 0x0FF;
154         p_desc->attr1           = attribute & 0xFF;
155         p_desc->limit_high_attr2= ((limit>>16) & 0x0F) | (attribute>>8) & 0xF0;
156         p_desc->base_high       = (base >> 24) & 0x0FF;
157 }
```

seg2phys的定义如代码6.13所示。

代码 6.13 由段名求绝对地址（chapter6/a/kernel/protect.c）

```
138 PUBLIC u32 seg2phys(u16 seg)
139 {
140         DESCRIPTOR* p_dest = &gdt[seg >> 3];
141         return (p_dest->base_high<<24 | p_dest->base_mid<<16 | p_dest->base_low);
142 }
```

vir2phys是一个宏，定义在protect.h中（代码6.14）。

代码 6.14 vir2phys（chapter6/a/include/protect.h）

```
147 /* 线性地址 → 物理地址 */
148 #define vir2phys(seg_base, vir) (u32)(((u32)seg_base) + (u32)(vir))
```

第三步，准备 GDT 和 TSS。

现在，再看一下图6.9，会发现剩下的没有初始化的只有 TSS（定义请见代码6.17）和它对应的描述符了。让我们来到init_prot()，填充 TSS 以及对应的描述符（代码6.15）。

代码 6.15 TSS 相关（chapter6/a/kernel/protect.c）

```
100         memset(&tss, 0, sizeof(tss));
101         tss.ss0 = SELECTOR_KERNEL_DS;
102         init_descriptor(&gdt[INDEX_TSS],
103                         vir2phys(seg2phys(SELECTOR_KERNEL_DS), &tss),
104                         sizeof(tss) - 1,
105                         DA_386TSS);
106         tss.iobase = sizeof(tss); /* 没有I/O许可位图 */
```

如今 TSS 已经准备好了，我们需要添加加载tr的代码。这更简单，只要在kernel.asm中添加几行就好了（代码6.16）。

代码 6.16 节自（chapter6/a/kernel/kernel.asm）

```
130         xor     eax, eax
131         mov     ax, SELECTOR_TSS
132         ltr     ax
```

代码 6.17 TSS（chapter6/a/include/protect.h）

```
typedef struct s_tss {
        u32     backlink;
        u32     esp0;   /* stack pointer to use during interrupt */
        u32     ss0;    /*   "   segment  "  "    "        "    */
        u32     esp1;
        u32     ss1;
        u32     esp2;
        u32     ss2;
        u32     cr3;
        u32     eip;
        u32     flags;
        u32     eax;
        u32     ecx;
        u32     edx;
        u32     ebx;
        u32     esp;
        u32     ebp;
        u32     esi;
        u32     edi;
        u32     es;
        u32     cs;
        u32     ss;
        u32     ds;
        u32     fs;
        u32     gs;
        u32     ldt;
        u16     trap;
        u16     iobase; /* I/O位图基址大于或等于TSS段界限，就表示没有I/O许可位图 */
}TSS;
```

6.3.2.3 iretd

在本节的开头，我们已经事先分析过restart这个函数，其实可以直接把它复制到我们的kernel.asm中。不过，由于我们只是想完成 ring0 到 ring1 的跳转，那个restart仍稍嫌复杂，让我们做一个简化的，像代码6.18这样就足够了。

代码 6.18 restart（chapter6/a/kernel/kernel.asm）

```
restart:
        mov     esp, [p_proc_ready]
        lldt    [esp + P_LDT_SEL]
        lea     eax, [esp + P_STACKTOP]
        mov     dword [tss + TSS3_S_SP0], eax

        pop     gs
        pop     fs
        pop     es
        pop     ds
        popad

        add     esp, 4

        iretd
```

其中，p_proc_ready是指向进程表结构的指针：

```
EXTERN PROCESS* p_proc_ready;
```

P_LDT_SEL、P_STACKTOP、TSS3_S_SP0和SELECTOR_TSS（代码6.16中用到）都定义在新建立的文件sconst.inc中。一定要注意，这里的选择子必须

与protect.h中的值保持一致。

由于进程的各寄存器值如今已经在进程表里面保存好了，现在我们只需要让esp指向栈顶，然后将各个值弹出就行了。最后一句iretd执行以后，eflags会被改变成pProc->regs.eflags的值。我们事先置了IF位，所以进程开始运行之时，中断其实也已经被打开了，虽然暂时来讲这没有意义，但了解这一点对以后的程序很重要。

好了，如果我们正在制造一把枪，那么现在大部分的工作已经完成，只剩下扳机了。是的，扳机。让我们回到kernel_main()，添加两行（代码6.19）。

代码 6.19 扳机（chapter6/a/kernel/main.c）

```
42      p_proc_ready    = proc_table;
43      restart();
```

最后的工作完成了，现在一切准备就绪！

6.3.2.4 进程启动（参考代码 chapter6/a）

可以看到，仅仅为了一个跳转，我们做了如此多的工作，如今是检验工作成果的时候了。make，运行，结果如图6.10所示。

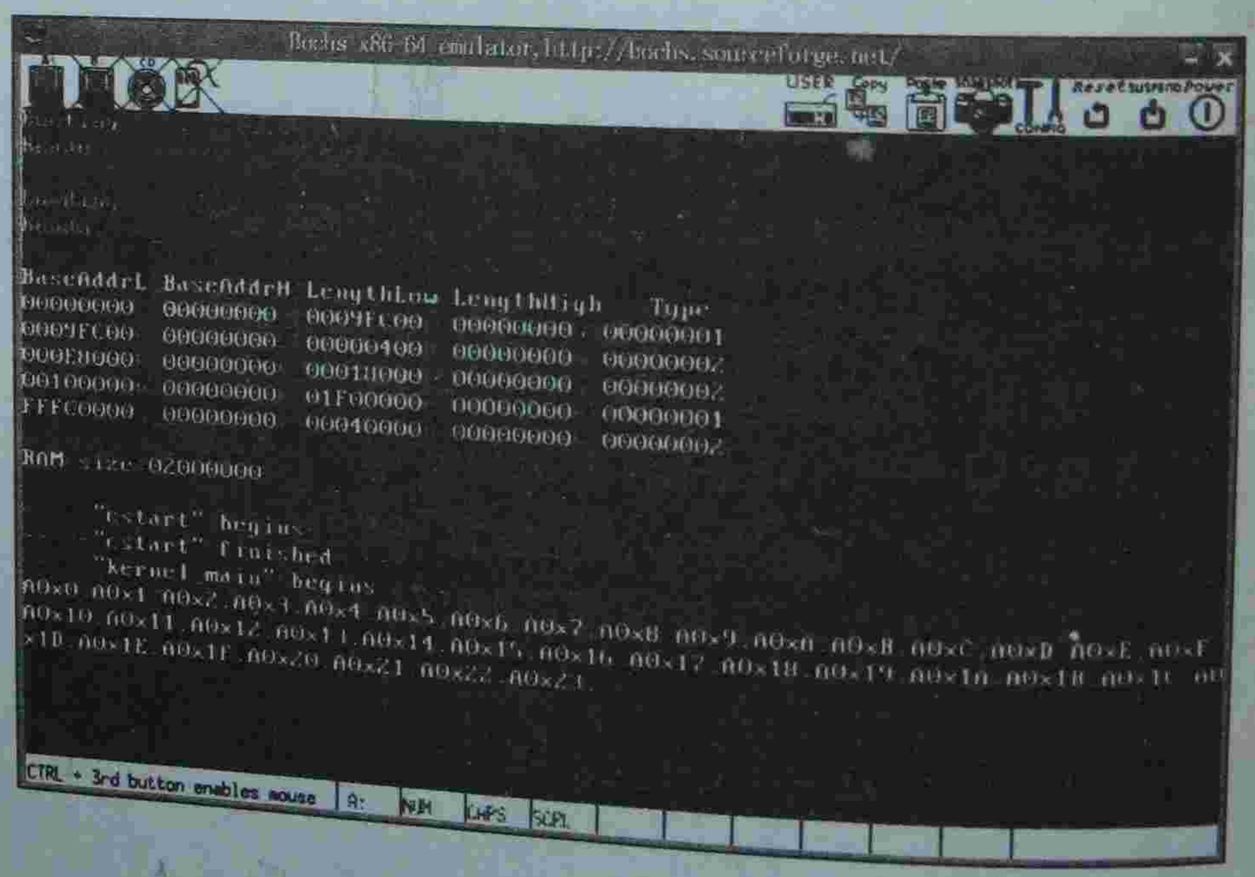


图 6.10 进程开始运行

我们的进程在运行了！

不是吗，我们看到了不断出现的字符“A”和不断增加的数字。虽然是一个再普通不过的函数在运行，对我们却有着不同寻常的意义。这意味着我们实现了 ring0 到 ring1 的跳转，再进一步，这意味着我们的进程在运行，而这一切意味着我们辛辛苦苦编写的这个东西已经可以称之为一个“操作系统”了。因为它已经有了“进程”，无论它是多么简陋。

6.3.2.5 第一个进程回顾

兴奋过后，让我们来进一步研究一下程序走过的过程。第一个进程的启动过程示意如图6.11所示。

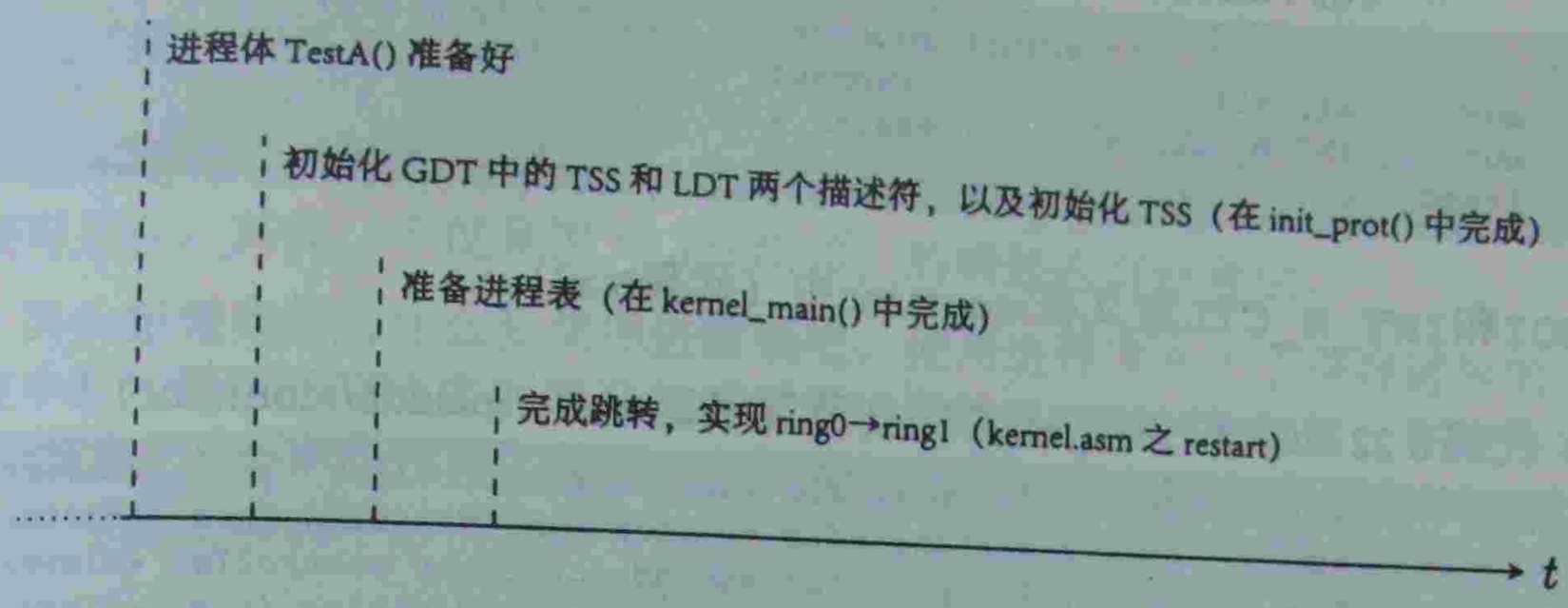


图 6.11 第一个进程的启动过程示意图

图6.11有助于让我们更清晰地了解程序的执行过程。横轴表示时间，从左到右是按照时间顺序依次完成的工作，其中进程体`TestA()`在内核被 LOADER 放置到内存中之后就准备好了。从图中可以看到，第181页中提到的进程表、进程体、GDT 和 TSS 这 4 个部分的初始化工作都已经完成。

进程已经开始执行了，此刻你在想什么呢？是仍在回味整个过程，还是心痒难耐地想要进一步完善进程调度？你一定已经考虑到，我们虽然已经用到了进程表，但毫无疑问，这离我们对它的期望还很远。我们希望进程表能够担当起保存并恢复进程状态的重任，而现在，我们的进程开启之后就再不停息，因为我们根本不曾开启时钟中断（上一章的最后我们只打开了键盘中断）。

其实，即便我们打开了时钟中断，时钟中断也只会发生一次，因为我们没有将中断结束位 EOI 置为 1，告知 8259A 当前中断结束。

不过没关系，我们第一步只想完成从 ring0 到 ring1 的转移。下面，就让我们一点点的完善，打开进程调度的大门。

6.3.3 第二步——丰富中断处理程序

我们在本章的开头就曾说过，中断在进程实现中扮演着重要的角色（看看图6.4和图6.5就知道了）。所以不开启中断显然是不行的，现在我们就慢慢把中断处理模块完善起来。

6.3.3.1 让时钟中断开始起作用

刚才提到，我们还没有打开时钟中断，现在就在i8259.c的`init_8259A()`中把它打开（代码6.20）。

代码 6.20 打开时钟中断（chapter6/b/kernel/i8259.c）

```
29      out_byte(INT_M_CTLMASK, 0xFE);  // Master 8259, OCW1.
30      out_byte(INT_S_CTLMASK, 0xFF);  // Slave  8259, OCW1.
```

为了让时钟中断可以不停地发生而不是只发生一次，还需要设置EOI（代码6.21）。

代码6.21 设置EOI（chapter6/b/kernel/kernel1.asm）

```
151 hwint00:                ; Interrupt routine for irq 0 (the clock).
⇒ 152         mov     al, EOI         ; `. reenable
⇒ 153         out     INT_M_CTL, al   ; / master 8259
154         iretd
```

EOI和INT_M_CTL定义在sconst.inc中（代码6.22）。

代码6.22 EOI和INT_M_CTL（节自 chapter6/b/include/sconst.inc）

```
33 INT_M_CTL     equ   0x20 ; I/O port for interrupt controller         <Master>
34 INT_M_CTLMASK equ   0x21 ; setting bits in this port disables ints  <Master>
35 INT_S_CTL     equ   0xA0 ; I/O port for second interrupt controller <Slave>
36 INT_S_CTLMASK equ   0xA1 ; setting bits in this port disables ints  <Slave>
37
38 EOI           equ   0x20
```

运行后发现结果和原来没有任何区别，这是意料之中的事，因为我们只是可以继续接受中断而已，其余并没有做什么。不过心里还是不太放心，我们甚至不知道中断处理程序到底是不是在运行。因此，可以在中断例程中再添加些东西，以便看到些效果（代码6.23）。

代码6.23 时钟中断处理程序（chapter6/b/kernel/kernel2.asm）

```
151 hwint00:                ; Interrupt routine for irq 0 (the clock).
⇒ 152         inc     byte [gs:0]     ; 改变屏幕第 0 行，第 0 列的字符
153         mov     al, EOI         ; `. reenable
154         out     INT_M_CTL, al   ; / master 8259
155         iretd
```

很明显，代码6.23参照了代码3.41的做法，通过改变屏幕第0行、第0列字符的方式来说明中断例程正在运行。本来这个位置是Boot的首字母“B”，如果发生中断，它会不断变化。运行一下，结果如图6.12所示。

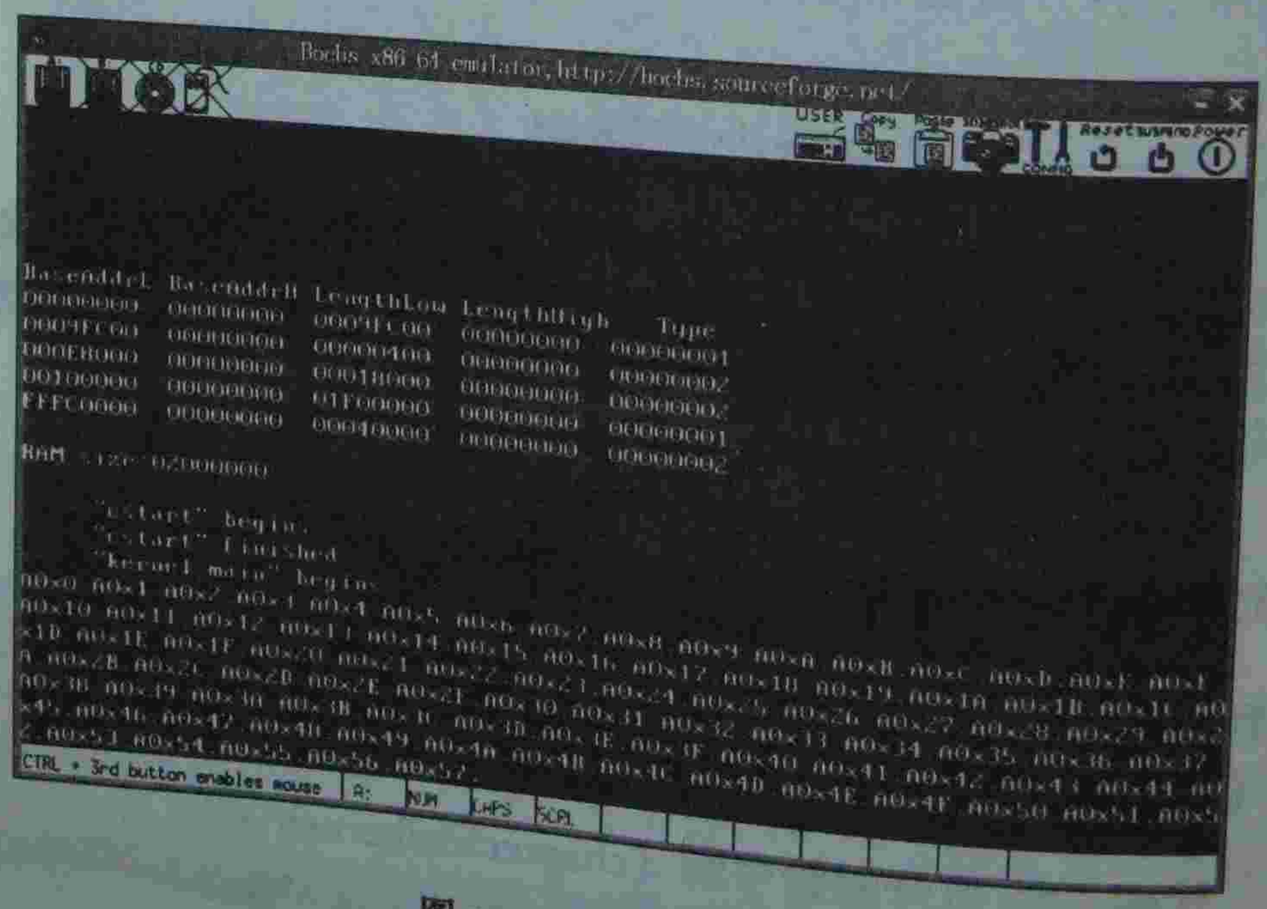


图6.12 进程开始运行

预期的结果出现了！黑色屏幕的左上角，不断变化的字符按照 ASCII 码的顺序在跳动，这说明中断处理程序的确是在运行的。本图正好显示到了字符“/”。

6.3.3.2 现场的保护与恢复

说不定你已经猜到了，为什么我们不用disp_str这个函数而用mov指令直接写显存。其实，不仅是因为这样简单，的确还有其他理由。

回头想想我们为什么要使用进程表吧。使用进程表是为了保存进程的状态，以便中断处理程序完成之后需要被恢复的进程能够被顺利地恢复。在进程表中，我们给每一个寄存器预留了位置，以便把它们所有的值都保存下来。这样就可以在进程调度模块中尽情地使用这些寄存器，而不必担心会对进程产生不良影响。

可是在现在这个很短的中断例程中，我们却在事先没有保存的情况下改变了al这个寄存器的值。al很小，但改变它毕竟是有风险的。之所以没用复杂一点的disp_str这个函数，是为了不会改变更多寄存器的值而产生更大的风险。从程序运行的情况来看，对al的改变并没有影响到进程的运行，但它仍让我们感到有些担心，现在我们就来把程序改进一下，改成代码6.24的样子。

代码 6.24 时钟中断处理程序（chapter6/b/kernel/kernel3.asm）

```
150 ALIGN   16
151 hwint00:                ; Interrupt routine for irq 0 (the clock).
152         pushad          ; `.
153         push    ds      ;  |
154         push    es      ;  | 保存原寄存器值
155         push    fs      ;  |
156         push    gs      ; /
157
158         inc     byte [gs:0]     ; 改变屏幕第 0 行，第 0 列的字符
159
160         mov     al, EOI         ; `. reenable
161         out     INT_M_CTL, al   ; /  master 8259
162
163         pop     gs      ; `.
164         pop     fs      ;  |
165         pop     es      ;  | 恢复原寄存器值
166         pop     ds      ;  |
167         popad           ; /
168
169         iretd
```

从现在开始每进行一次代码修改我都建议你make并运行一下，以便看到效果，下文中有些地方就不再提醒了。在这里运行，仍可以看到进程的运行以及跳动的字符。

6.3.3.3 赋值tss.esp0

现在的中断处理程序看上去像样多了，寄存器先是被保存，后又被恢复，进程被很好地保护起来。不过，有一点不知道你有没有想到，中断现在已经被打开，于是就存在 ring0 和 ring1 之间频繁的切换。两个层级之间的切换包含两方面，一是代码的跳转，还有一个不容忽视的方面，就是堆栈也在切换。

由ring0到ring1时，堆栈的切换直接在指令iretd被执行时就完成了，目标代码的cs、eip、ss、esp等都是从堆栈中得到，这很简单。但ring1到ring0切换时就免不了用到TSS了。其实到目前为止，TSS对于我们的用处也只是保存ring0堆栈信息，而堆栈的信息也不外乎就是ss和esp两个寄存器。回想一下，在上一节中，为了搭建一个进程调度的大致框架，我们已经做了一些TSS的初始化工作，并且已经给TSS中用于ring0的ss赋了值（代码6.15第101行），那么，tss.esp0应该在什么时候被赋值呢？其实我们在上一节分析restart代码的时候已经涉及过这个问题了。由于要为下一次ring1→ring0做准备，所以用iretd返回之前要保证tss.esp0是正确的。

我们分析过，当进程被中断切到内核态，当前的各个寄存器应该被立即保存（压栈）。也就是说，每个进程在运行时，tss.esp0应该是当前进程的进程表中保存寄存器值的地方，即struct s_proc中struct s_stackframe的最高地址处。这样，进程被挂起后才恰好保存寄存器到正确的位置。我们假设进程A在运行，那么tss.esp0的值应该是进程表A中regs的最高处，因为我们是不可能在进程A运行时来设置tss.esp0的值的，所以必须在A被恢复运行之前，即iretd执行之前做这件事。换句话说，我们应该在时钟中断处理结束之前做这件事。

代码6.25给出了实现方法。

代码6.25 修改时钟中断处理（chapter6/b/kernel/kernel4.asm）

```
150  ALIGN   16
151  hwint00:                ; Interrupt routine for irq 0 (the clock).
⇒152         sub     esp,4
153          pushad          ; .
154          push    ds      ;  |
155          push    es      ;  | 保存原寄存器值
156          push    fs      ;  |
157          push    gs      ; /
⇒158         mov     dx, ss
⇒159         mov     ds, dx
⇒160         mov     es, dx
161
162          inc     byte [gs:0]     ; 改变屏幕第 0 行，第 0 列的字符
163
164          mov     al, EOI         ; .reenable
165          out     INT_M_CTL, al   ; /  master 8259
166
⇒167         lea     eax, [esp + P_STACKTOP]
⇒168         mov     dword [tss + TSS3_S_SP0], eax
169
170          pop     gs      ; .
171          pop     fs      ;  |
172          pop     es      ;  | 恢复原寄存器值
173          pop     ds      ;  |
174          popad           ; /
⇒175         add     esp,4
176
177          iretd
```

你可能注意到，在这里不仅增加了给tss.esp0赋值的语句，而且还额外增加了几句代码。你可以看到，sub/add esp这两句代码实际上是跳过了4字节，结合进程表的定义知道，被跳过的这4字节实际上就是那个retaddr，我们还是先不管这个值。

另外3行mov是令ds和es指向与ss相同的段。

现在我们的中断例程变成了这样：在中断发生的开始，esp的值是刚刚从TSS里面取到的进程表A中regs的最高地址，然后各寄存器值被压栈入进程表，最后esp指向regs的最低地址处，然后设置tss.esp0的值，准备下一次进程被中断时使用。

如今我们只有一个进程，第二次时钟中断之后对tss.esp0的赋值其实就是在重复。但以后我们会实现多个进程，在进程B或者C将要获得CPU之前，tss.esp0的值会被修改成进程表B或者C中相应的地址。

看到这里你可能会想，刚开始添加两行置EOI位的地址代码时中断就已经打开，从那时起就存在了ring0到ring1的切换，可直到现在我们才把tss.esp0的值补全。也就是说，当前面的程序发生ring1→ring0跳转时，esp一定指向了一个错误而且有风险的地方。实际上，我们是冒了一个险，因为我们在不知道esp指向何处时就使用了它。

6.3.3.4 内核栈

我们曾经提到过内核栈的问题，如今这个问题真的出现了。现在esp指向的是进程表，如果此时我们要执行复杂的进程调度程序呢？最简单的例子，如果我们想调用一个函数，这时一定会用到堆栈操作，那么，我们的进程表立刻会被破坏掉。所以我们需要切换堆栈，将esp指向另外的位置。

在引入内核栈时笔者曾经提醒过，在具体编写代码的过程中，一定要清楚当前使用的是哪个堆栈，以免破坏掉不应破坏的数据。现在就到了该用内核栈的时候了（代码6.26）。

代码6.26 修改时钟中断处理（chapter6/b/kernel/kernel5.asm）

```
150 ALIGN   16
151 hwint00:                ; Interrupt routine for irq 0 (the clock).
152         sub     esp, 4
153         pushad          ; `.
154         push    ds      ;  |
155         push    es      ;  | 保存原寄存器值
156         push    fs      ;  |
157         push    gs      ; /
158         mov     dx, ss
159         mov     ds, dx
160         mov     es, dx
161
⇒ 162       mov     esp, StackTop                   ; 切到内核栈
163
164         inc     byte [gs:0]                     ; 改变屏幕第 0 行，第 0 列的字符
165
166         mov     al, EOI                         ; `. reenable
167         out     INT_M_CTL, al                   ; /  master 8259
168
⇒ 169       mov     esp, [p_proc_ready]             ; 离开内核栈
170
171         lea     eax, [esp + P_STACKTOP]
172         mov     dword [tss + TSS3_S_SP0], eax
173
174         pop     gs      ; `.
175         pop     fs      ;  |
176         pop     es      ;  | 恢复原寄存器值
177         pop     ds      ;  |
178         popad           ; /
179         add     esp, 4
```

```
180
181         iretd
```

切到内核栈和重新将esp切到进程表都很简单，一个mov语句就够了，但是它却非常关键。如果没有这个简单的mov，随着中断例程越来越大，出错的时候，你可能都不知道错在哪里。

在这里我们尽可能地把代码放在使用内核栈的过程中来执行，只留下跳回进程所必需的代码。其实是想暗示，使用内核栈后，我们的中断例程可以变得很复杂。我们不妨在这里试一下，把这段打印字符的代码替换成使用DispStr这个函数（代码6.27）。

代码6.27 修改时钟中断处理（chapter6/b/kernel/kernel.final.asm）

```
16  extern  disp_str
    ...
27  [SECTION .data]
28  clock_int_msg           db      "^", 0
    ...
155 hwint00:                ; Interrupt routine for irq 0 (the clock).
156         sub     esp, 4
157         pushad          ; `.
158         push    ds      ;  |
159         push    es      ;  | 保存原寄存器值
160         push    fs      ;  |
161         push    gs      ; /
162         mov     dx, ss
163         mov     ds, dx
164         mov     es, dx
165
166         mov     esp, StackTop           ; 切到内核栈
167
168         inc     byte [gs:0]             ; 改变屏幕第 0 行, 第 0 列的字符
169
170         mov     al, EOI
171         out     INT_M_CTL, al           ; `. reenable
172                                         ; /  master 8259
⇒ 173       push    clock_int_msg
⇒ 174       call    disp_str
⇒ 175       add     esp, 4
176
177         mov     esp, [p_proc_ready]     ; 离开内核栈
178
179         lea     eax, [esp + P_STACKTOP]
180         mov     dword [tss + TSS3_S_SP0], eax
181
182         pop     gs      ; `.
183         pop     fs      ;  |
184         pop     es      ;  | 恢复原寄存器值
185         pop     ds      ;  |
186         popad           ; /
187         add     esp, 4
188
189         iretd
```

其中，clock_int_msg的定义如代码6.27所示，仅仅是个字符“^”而已。

此时，如果运行的话，就应该看到图6.13所示的画面了。

我们看到不断出现的字符“^”，说明函数disp_str运行正常，而且没有影响到中断处理的其他部分以及进程A。之所以在两次字符A的打印中间有多

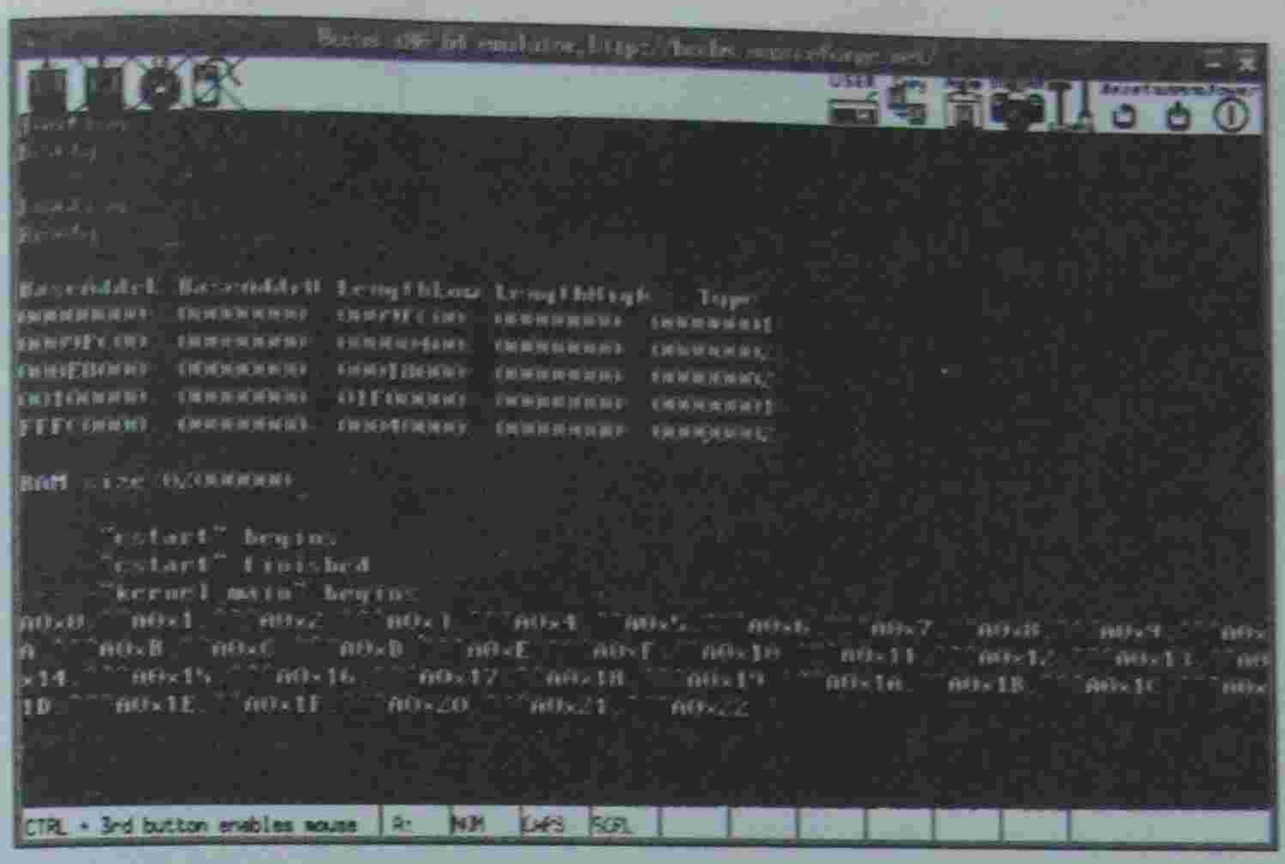


图 6.13 时钟中断模块运行正常

个“^”，是因为我们的进程执行体中加入了delay()函数，在此函数的执行过程中发生了多次中断。

6.3.3.5 中断重入

从开始只有一句iretd的中断处理程序到现在，我们已经增加了许多内容。而且我们知道，在将寄存器的值保存好以及使用了内核栈之后，可以认为将更加复杂的内容添加进去都是没有问题的。但是，现在又出现了另外一个的问题，由于中断处理程序的内容变得愈来愈复杂，我们是否应该允许中断嵌套？也就是说，在中断处理过程中，是否应该允许下一个中断发生？不允许肯定是不行的，因为你一定不希望在进程调度时你的按键就不再响应。于是，我们必须用合适的机制来应付嵌套的情况。那么，嵌套会是怎样的情形呢？让我们修改一下代码，以便让系统可以在时钟中断的处理过程中接受下一个时钟中断。这听起来不是个很好的主意，但是可以借此来做个试验。

首先，因为CPU在响应中断的过程中会自动关闭中断，我们需要人为地打开中断，加入sti指令；然后，为保证中断处理过程足够长，以至于在它完成之前就会有下一个中断产生，我们在中断处理例程中调用一个延迟函数。具体修改如代码6.28所示。

代码 6.28 修改时钟中断处理（chapter6/c/kernel/kernel1.asm）

```
17  extern  delay
    ...
156 hwint00:                ; Interrupt routine for irq 0 (the clock).
157
158         sub     esp, 4
159         pushad          ; .
160         push    ds      ; |
161         push    es      ; | 保存原寄存器值
162         push    fs      ; |
163         push    gs      ; /
164         mov     dx, ss
165         mov     ds, dx
166         mov     es, dx
```

```
167     mov     esp, StackTop                   ; 切到内核栈
168
169     inc     byte [gs:0]                     ; 改变屏幕第 0 行, 第 0 列的字符
170
171     mov     al, EOI                         ; .. reenable
172     out     INT_M_CTL, al                   ; /  master 8259
173
⇒ 174   sti
175
176     push    clock_int_msg
177     call    disp_str
178     add     esp, 4
179
⇒ 180   push    1
⇒ 181   call    delay
⇒ 182   add     esp, 4
183
⇒ 184   cli
185
186     mov     esp, [p_proc_ready]             ; 离开内核栈
187
188     lea     eax, [esp + P_STACKTOP]
189     mov     dword [tss + TSS3_S_SP0], eax
190
191     pop     gs          ; .
192     pop     fs          ; |
193     pop     es          ; | 恢复原寄存器值
194     pop     ds          ; |
195     popad               ; /
196     add     esp, 4
197
198     iretd
```

运行，你会发现，在打印了一个A0x0之后就不停打印“^”，再也进不到进程里面，如图6.14所示。

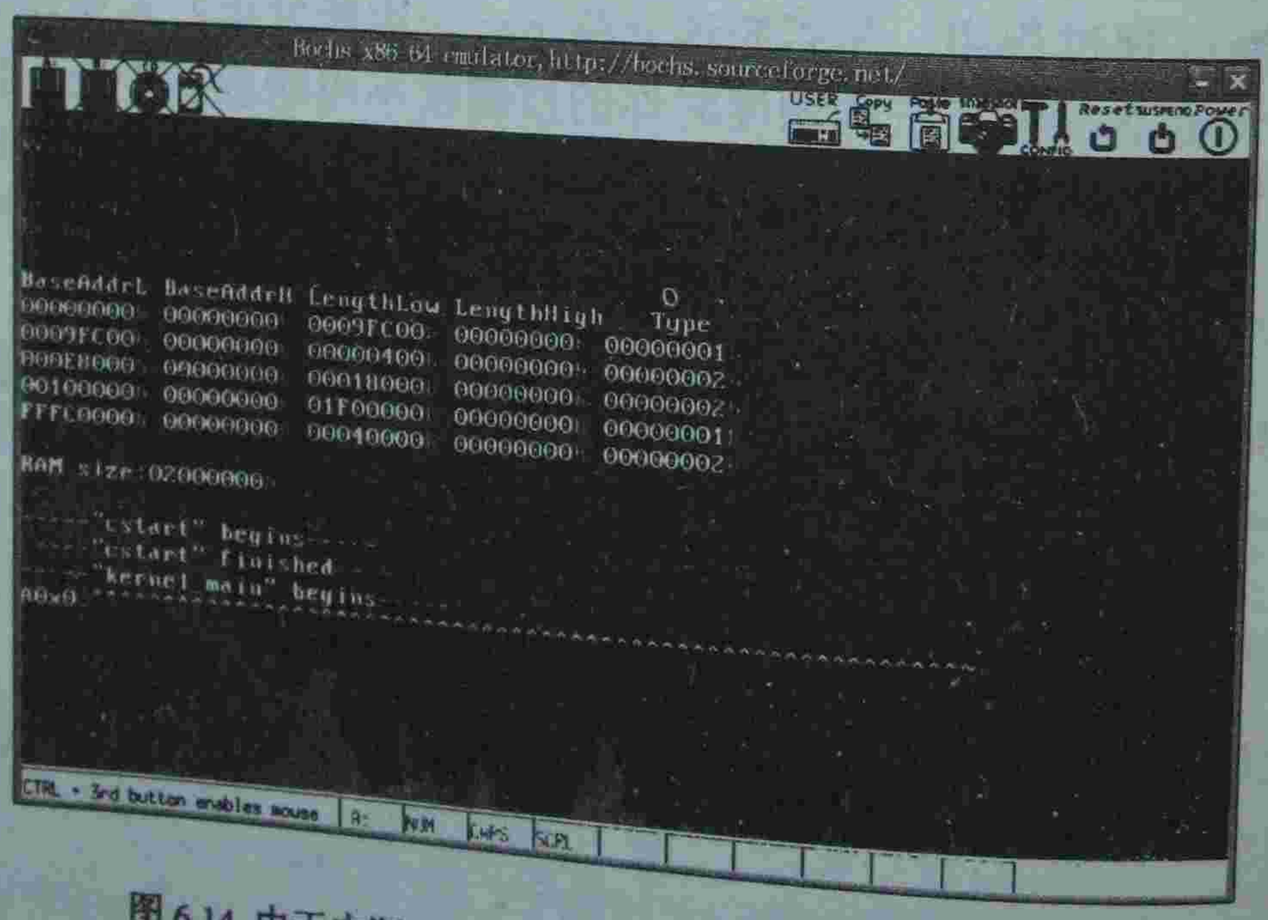


图 6.14 由于中断重入的发生，进程挂起后无法再被恢复

之所以会产生这种情况，是因为在一次中断还未处理完时，又一次中断发生了。这时程序又跳到中断处理程序的开头，如此反复，永远也执行不到中断处理程序的结尾——跳回进程继续执行。而且，由于压栈操作多而出栈操作少，随着时间的继续，当堆栈溢出的时候，意料不到的事情就可能发生了。

中断处理程序是被动的，它只知道当忠实的中断发生时执行那段代码，完全不理会中断在何时发生。可是，为了避免这种嵌套现象的发生，我们必须想一个办法让中断处理程序知道自己是不是在嵌套执行。

这个问题并不难解决，只要设置一个全局变量就可以了。这个全局变量有一个初值−1，当中断处理程序开始执行时它自加，结束时自减。在处理程序开头处这个变量需要被检查一下，如果值不是0(0 = −1 + 1)，则说明在一次中断未处理完之前就又发生了一次中断，这时直接跳到最后，结束中断处理程序的执行。当然，武断地结束新的中断并不是一个好的办法，但在这里，我们姑且这样来做。

好了，我们按照这个思路把程序修改一下（代码6.29）。

代码 6.29 k_reenter（chapter6/c/kernel/main.c）

```
20 PUBLIC int kernel_main()
21 {
 ...
40         k_reenter = -1;
 ...
47 }
```

然后在中断例程中加入k_reenter自加以及判断是否为 0 的代码（代码6.30）。

代码 6.30 修改时钟中断处理（chapter6/c/kernel/kernel2.asm）

```
25 extern  k_reenter
 ...
157 hwint00:                ; Interrupt routine for irq 0 (the clock).
158         sub     esp, 4
159         pushad          ; .
160         push    ds      ; |
161         push    es      ; | 保存原寄存器值
162         push    fs      ; |
163         push    gs      ; /
164         mov     dx, ss
165         mov     ds, dx
166         mov     es, dx
167
168         inc     byte [gs:0]           ; 改变屏幕第 0 行，第 0 列的字符
169
170         mov     al, EOI               ; .. reenable
171         out     INT_M_CTL, al         ; /  master 8259
172
⇒ 173       inc     dword [k_reenter]
⇒ 174       cmp     dword [k_reenter], 0
⇒ 175       jne     .re_enter
176
177         mov     esp, StackTop         ; 切到内核栈
178
179         sti
180
181         push    clock_int_msg
182         call    disp_str
183         add     esp, 4
184
185         push    1
186         call    delay
187         add     esp, 4
188
189         cli
190
```

```
191         mov     esp, [p_proc_ready]     ; 离开内核栈
192
193         lea     eax, [esp + P_STACKTOP]
194         mov     dword [tss + TSS3_S_SP0], eax
195
196 .re_enter:     ; 如果(k_reenter != 0)，会跳转到这里
197         dec     dword [k_reenter]
198         pop     gs      ; ┓
199         pop     fs      ; ┃
200         pop     es      ; ┃ 恢复原寄存器值
201         pop     ds      ; ┃
202         popad           ; ┛
203         add     esp, 4
204
205         iretd
```

我们再make一下，运行，结果如图6.15所示。

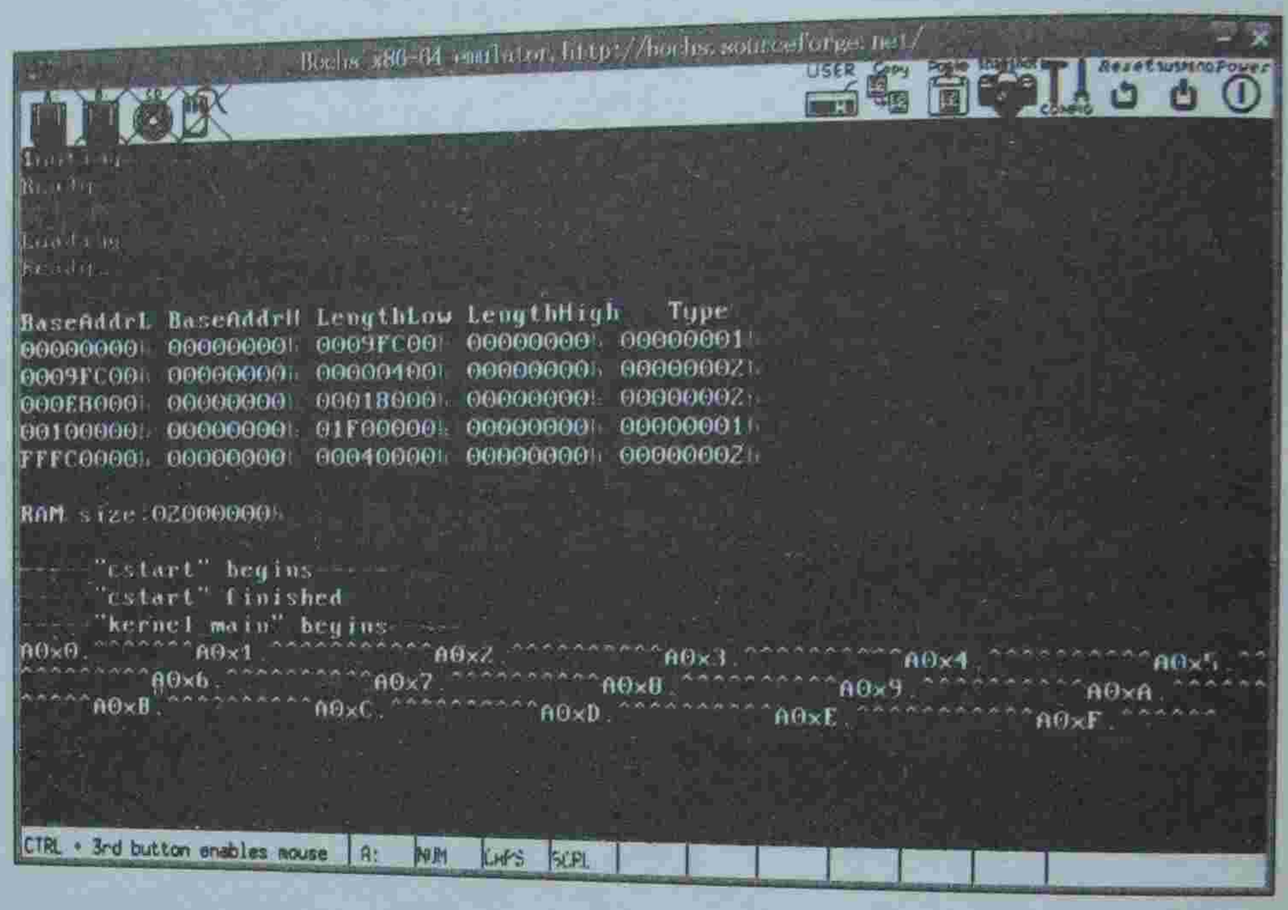


图 6.15 解决中断重入问题

可以看到，字符 A 和相应的数字又在不停出现了，这说明我们的修改生效了。而且，你可以发现，屏幕左上角的字母跳动速度快而字符“^”打印速度慢，说明有很多时候程序在执行了`inc byte [gs:0]`之后并没有执行`disp_str`，这也说明中断重入的确发生了。如果你想做对比的话，建议你现在执行一下chapter6/b中的程序，你会发现打印“^”的速度和左上角的字母跳动的速度是一样的。

好了，我们已经有了一个办法来解决中断重入这个问题，那么让我们注释掉刚才的打印字符以及Delay等语句（代码6.31）。

代码 6.31 修改时钟中断处理（chapter6/c/kernel/kernel3.asm）

```
157 hwint00:                ; Interrupt routine for irq 0 (the clock).
158         sub     esp, 4
159         pushad          ; ┓
160         push    ds      ; ┃
161         push    es      ; ┃ 保存原寄存器值
162         push    fs      ; ┃
163         push    gs      ; ┛
164         mov     dx, ss
165         mov     ds, dx
166         mov     es, dx
```

```
167         inc     byte [gs:0]                     ; 改变屏幕第 0 行, 第 0 列的字符
168
169         mov     al, EOI                         ; .
170         out     INT_M_CTL, al                   ; | reenable
171                                                 ; / master 8259
172         inc     dword [k_reenter]
173         cmp     dword [k_reenter], 0
174         jne     .re_enter
175
176         mov     esp, StackTop                   ; 切到内核栈
177
178         sti
179
180         push    clock_int_msg
181         call    disp_str
182         add     esp, 4
183
184
185 ;;;     push    1
186 ;;;     call    delay
187 ;;;     add     esp, 4
188
189         cli
190
191         mov     esp, [p_proc_ready]     ; 离开内核栈
192
193         lea     eax, [esp + P_STACKTOP]
194         mov     dword [tss + TSS3_S_SP0], eax
195
196 .re_enter:      ; 如果(k_reenter != 0), 会跳转到这里
197         dec     dword [k_reenter]
198         pop     gs          ; .
199         pop     fs          ; |
200         pop     es          ; | 恢复原寄存器值
201         pop     ds          ; |
202         popad               ; /
203         add     esp, 4
204
205         iretd
```

再次运行，结果如图6.16所示。

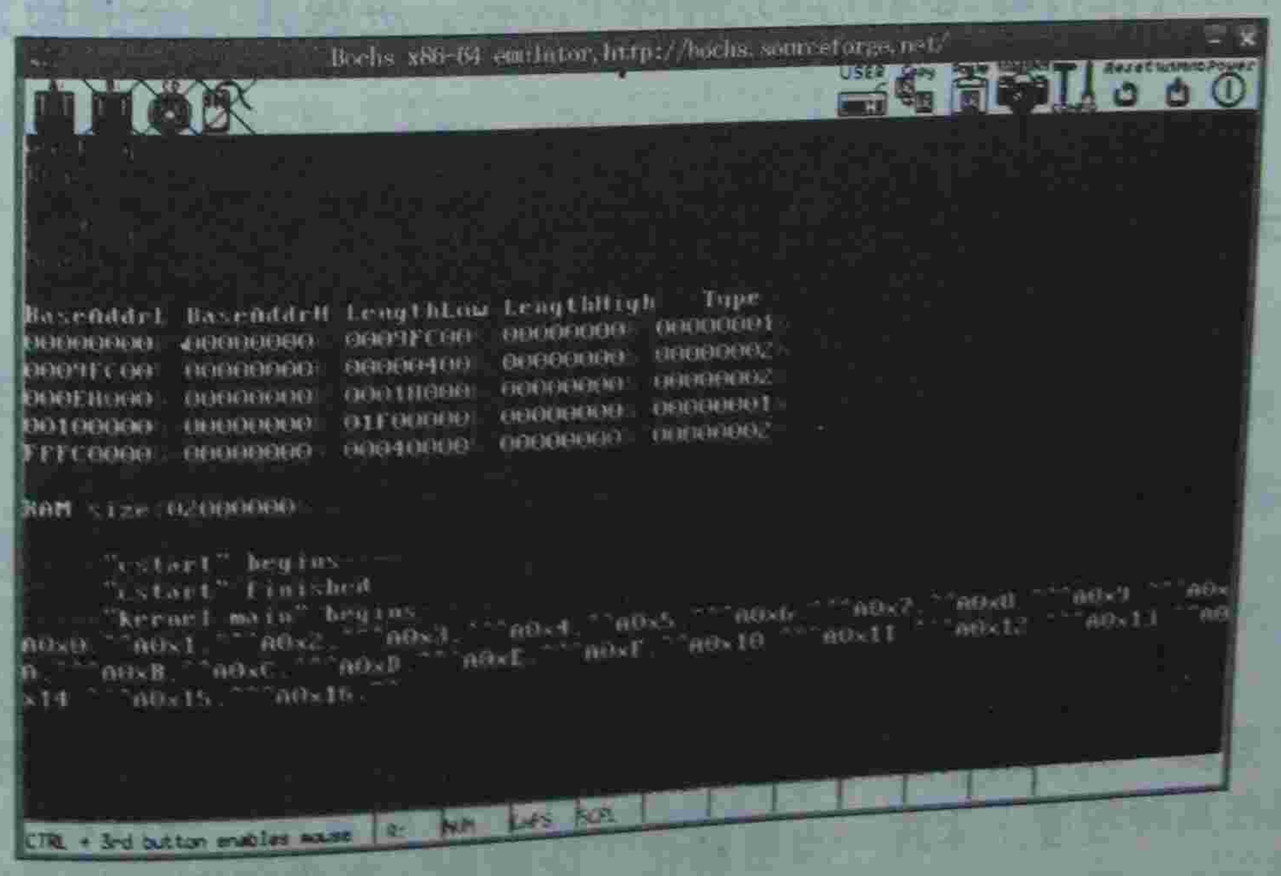


图 6.16 考虑中断重入之后

你可能觉得图6.16非常眼熟，是的，在我们没有考虑中断重入的时候，画

面跟现在是一样的。但是你我都知道，如今我们的代码考虑了更多的情况，于是可以适应更多的条件变化，它比原先更加成熟。

6.4 多进程

虽然在刚刚完成 ring0 到 ring1 跳转时就说，我们的“进程”正在运行。但是直到现在，我们才能在说这句话时感到坦然，因为，进程此时不仅是在运行而已，它可以随时被中断，可以在中断处理程序完成之后被恢复。进程此时已经有了两种状态：运行和睡眠。由此不难想到，我们已经具备了处理多个进程的能力，只需要让其中一个进程处在运行态，其余进程处在睡眠态就可以了。

那么，现在我们就来试一试实现多个进程。

6.4.1 添加一个进程体

别的不用说，添加一个进程体当然是少不了的，让我们先把这个最简单的工作做完，在main.c中进程 A 的代码的下面添加进程 B（代码6.32）。

代码 6.32 k_reenter（chapter6/d/kernel/main.c）

```
89  void TestB()
90  {
91          int i = 0x1000;
92          while(1){
93                  disp_str("B");
94                  disp_int(i++);
95                  disp_str(".");
96                  delay(1);
97          }
98  }
```

在这里，除了打印的字母换成了 B 之外，i的初始值被设成了0x1000，为的就是在将来程序运行时能清晰地分辨两个进程。进程体其余的地方跟进程A完全一样。

6.4.2 相关的变量和宏

让我们再来回忆一下当初准备第一个进程时还做了哪些工作。我们前面提到，进程不外乎 4 个要素：进程表、进程体、GDT、TSS。

进程体我们已经有了，下一个要关注的问题是进程表如何初始化。让我们再一次来到kernel_main()函数中看一下进程 A 的初始化工作是怎么做的。说起来很简单，将其中的几个关键成员赋值就可以了。我们当然可以将这份代码复制一份，将其中涉及进程 A 的内容统统改成与进程 B 相关的代码。可是显然那样有些蹩脚，因为我们不可能每增加一个进程就复制一份代码，最好我们能够让代码在某种程度上实现一点自动化，让增加一个进程变得简单而迅速。

拿来主义最好不过，Minix 中定义了一个数组叫做tasktab[1]。这个数组的每一项定义好一个任务（在本章中“任务”和“进程”是可以互换的）的开始

[1]如果有兴趣，读者可以自行查看其源码，数组定义在 Minix 源码的src/kernel/table.c中。

地址、堆栈等，在初始化的时候，只要用一个for循环依次读取每一项，然后填充到相应的进程表项中就可以了。

我们首先在proc.h中声明一个数组类型（代码6.33）。

代码 6.33 k_reenter（chapter6/d/include/proc.h）

```
42 typedef struct s_task {
43         task_f  initial_eip;
44         int     stacksize;
45         char    name[32];
46 }TASK;
```

其中，task_f是这样定义的（在type.h中）：

```
typedef void (*task_f) ();
```

一个进程只要有一个进程体和堆栈就可以运行了，所以，这个数组只要有前两个成员其实就已经够了。这里，我们还定义了name，以便给每个进程起一个名字。

好了，下面我们在 global.c 中增加这样一个定义（代码6.34）。

代码 6.34 task_table（chapter6/d/kernel/global.c）

```
22 PUBLIC  TASK    task_table[NR_TASKS] = {{TestA, STACK_SIZE_TESTA, "TestA"},
23                                        {TestB, STACK_SIZE_TESTB, "TestB"}};
```

别忘了同时在global.h中加入这么一行：

```
extern  TASK    task_table[];
```

此外你一定已经想起来一件事，就是我们当初就考虑到了以后的扩充，虽然只有一个进程，我们还是安排了一个进程表数组proc_table[NR_TASKS]，只是NR_TASKS为 1 罢了。

显然，进程表里有几项task_table也应该有几项。在这里，我们已经有两个进程了，所以先把NR_TASKS的值修改为 2（见代码6.35）。

代码6.34中的STACK_SIZE_TESTB还没有定义。我们到proc.h中添加下面的定义（代码6.35）。

代码 6.35 k_reenter（chapter6/d/include/proc.h）

```
49 /* Number of tasks */
50 #define NR_TASKS        2
51
52 /* stacks of tasks */
53 #define STACK_SIZE_TESTA        0x8000
54 #define STACK_SIZE_TESTB        0x8000
55
56 #define STACK_SIZE_TOTAL        (STACK_SIZE_TESTA + \
57                                 STACK_SIZE_TESTB)
```

最后，在proto.h中加入TestB的函数声明：

```
void TestB();
```

好了，围绕task_table，与新添加进程相关的定义已经完成，可这不足以马上让新进程运转起来，下面我们就来做进程表的初始化工作。

6.4.3 进程表初始化代码扩充

现在可以用for循环来做进程表的初始化工作了（代码6.36）。

代码 6.36 初始化进程表（chapter6/d/kernel/main.c）

```
PUBLIC int kernel_main()
{
        disp_str("-----\"kernel_main\" begins-----\n");

        TASK*           p_task          = task_table;
        PROCESS*        p_proc          = proc_table;
        char*           p_task_stack    = task_stack + STACK_SIZE_TOTAL;
        u16             selector_ldt    = SELECTOR_LDT_FIRST;
        int i;
        for(i=0;i<NR_TASKS;i++){
                strcpy(p_proc->p_name, p_task->name);   // name of the process
                p_proc->pid = i;                        // pid

                p_proc->ldt_sel = selector_ldt;

                memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS >> 3],
                       sizeof(DESCRIPTOR));
                p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5;
                memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS >> 3],
                       sizeof(DESCRIPTOR));
                p_proc->ldts[1].attr1 = DA_DRW | PRIVILEGE_TASK << 5;
                p_proc->regs.cs = ((8 * 0) & SA_RPL_MASK & SA_TI_MASK)
                        | SA_TIL | RPL_TASK;
                p_proc->regs.ds = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
                        | SA_TIL | RPL_TASK;
                p_proc->regs.es = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
                        | SA_TIL | RPL_TASK;
                p_proc->regs.fs = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
                        | SA_TIL | RPL_TASK;
                p_proc->regs.ss = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK)
                        | SA_TIL | RPL_TASK;
                p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK)
                        | RPL_TASK;

                p_proc->regs.eip = (u32)p_task->initial_eip;
                p_proc->regs.esp = (u32)p_task_stack;
                p_proc->regs.eflags = 0x1202; /* IF=1, IOPL=1 */

                p_task_stack -= p_task->stacksize;
                p_proc++;
                p_task++;
                selector_ldt += 1 << 3;
        }

        k_reenter = -1;

        p_proc_ready    = proc_table;
        restart();

        while(1){}
}
```

在我们这个简单的例子中，进程之间区别真的不大。每一次循环的不同在于，从TASK结构中读取不同的任务入口地址、堆栈栈顶和进程名，然后赋给相应的进程表项。需要注意的地方有两点。

- 由于堆栈是从高地址往低地址生长的，所以在给每一个进程分配堆栈空间的时候也是从高地址往低地址进行。

- 在这里，我们为每一个进程都在GDT中分配一个描述符用来对应进程的LDT。在protect.h中可以看到，SELECTOR_LDT_FIRST是GDT中被定义的最后一个描述符，但是正如它的名字所表示的，它仅仅是“第一个”和“唯一一个”被明白指出来的而已。实际上，我们在task_table中定义了几个任务，通过上文的for循环中的代码，GDT中就会有几个描述符被初始化，它们列在SELECTOR_LDT_FIRST之后。

此外，对于进程的名称（p_proc->p_name）和序号（p_proc->pid）的设置，目前来看是锦上添花的事情，并没有什么实际的作用。

6.4.4 LDT

我们刚刚解释过，每一个进程都会在GDT中对应一个LDT描述符。于是在for循环中，我们将每个进程表项中的成员p_proc->ldt_sel赋值。可是，选择子仅仅是解决了where问题，通过它，我们能在GDT中找到相应的描述符，但描述符的具体内容是什么，即what问题还没有解决。

当初只有一个进程时，我们是在init_prot()这个函数中通过一个语句解决了what的问题。现在，我们同样需要把它变成一个循环（代码6.37）。

代码6.37 初始化LDT（chapter6/d/kernel/protect.c）

```
109         int i;
110         PROCESS* p_proc = proc_table;
111         u16 selector_ldt = INDEX_LDT_FIRST << 3;
112         for(i=0;i<NR_TASKS;i++){
113                 init_descriptor(&gdt[selector_ldt>>3],
114                                 vir2phys(seg2phys(SELECTOR_KERNEL_DS),
115                                         proc_table[i].ldts),
116                                 LDT_SIZE * sizeof(DESCRIPTOR) - 1,
117                                 DA_LDT);
118                 p_proc++;
119                 selector_ldt += 1 << 3;
120         }
```

这个循环显然比代码6.36中的那个简单多了，只是注意几次位移就可以了。

另外，每个进程都有自己的LDT，所以当进程切换时需要重新加载ldtr，见代码6.39第187行。

6.4.5 修改中断处理程序

这样是不是就可以了呢？make，然后运行一下，没有变化？跟原先一模一样？原来每次发生时钟中断的时候，被恢复的进程还是原来的A，我们还没有编写任何进程切换代码。

时钟中断处理程序位于kernel.asm中，除了保存和恢复进程信息，我们只做了一件简单的事，就是在屏幕上打印一个字符“^”。显然，进程切换的代码就应该添加在这个位置才对。

回忆一下，一个进程如何由“睡眠”状态变成“运行”状态？无非是将esp指向进程表项的开始处，然后在执行lldt之后经历一系列pop指令恢复各个寄

存器的值。一切信息都包含在进程表中，所以，要想恢复不同的进程，只需要将esp指向不同的进程表就可以了。

在离开内核栈的时候，有一个语句是为esp赋值的：

```
mov     esp, [p_proc_ready]
```

全局变量p_proc_ready是指向进程表结构的指针，我们只需要在这一句执行之前把它赋予不同的值就可以了。

可以想见，进程切换是一个稍稍有点复杂的过程，因为涉及进程调度等内容。一方面，这涉及算法等一些复杂的内容，另一方面，它应该是与硬件无关的，所以我们用C语言编写这个模块。

那么，代码如何组织呢？这块内容既是时钟中断的一部分，又关乎进程调度。所以我们可以创建一个clock.c，也可以创建一个proc.c。我们再来学习一下Minix，创建一个clock.c。目前clock.c里面只有一个函数（代码6.38）。

代码 6.38 时钟中断处理程序（chapter6/d/kernel/clock.c）

```
20  PUBLIC void clock_handler(int irq)
21  {
22          disp_str("#");
23  }
```

函数只有一个语句，就是打印一个字符“#”。下面我们在时钟中断例程中调用这个函数（见代码6.39）。

代码 6.39 加载 ldtr（chapter6/d/kernel/kernel.asm）

```
158 hwint00:                ; Interrupt routine for irq 0 (the clock).
159         sub     esp, 4
160         pushad          ; `.
161         push    ds      ;  |
162         push    es      ;  | 保存原寄存器值
163         push    fs      ;  |
164         push    gs      ; /
165         mov     dx, ss
166         mov     ds, dx
167         mov     es, dx
168
169         inc     byte [gs:0]             ; 改变屏幕第 0 行，第 0 列的字符
170
171         mov     al, EOI
172         out     INT_M_CTL, al           ; `. reenable
173                                         ; /  master 8259
174         inc     dword [k_reenter]
175         cmp     dword [k_reenter], 0
176         jne     .re_enter
177
178         mov     esp, StackTop           ; 切到内核栈
179
180         sti
⇒ 181       push    0
⇒ 182       call    clock_handler
⇒ 183       add     esp, 4
184         cli
185
186
⇒ 187       mov     esp, [p_proc_ready]
188         lldt    [esp + P_LDT_SEL]       ; 离开内核栈
189         lea     eax, [esp + P_STACKTOP]
190         mov     dword [tss + TSS3_S_SP0], eax
```

```
.re_enter:      ; 如果(k_reenter != 0)，会跳转到这里
        dec     dword [k_reenter]
        pop     gs      ; ┐
        pop     fs      ; │
        pop     es      ; │ 恢复原寄存器值
        pop     ds      ; │
        popad           ; ┘
        add     esp, 4
        iretd
```

由于增加了一个文件，所以不要忘记修改Makefile。make，运行，看到图6.17所示的情景。图中打印的字符“#”说明我们刚刚所增加的代码已经在正确运行。

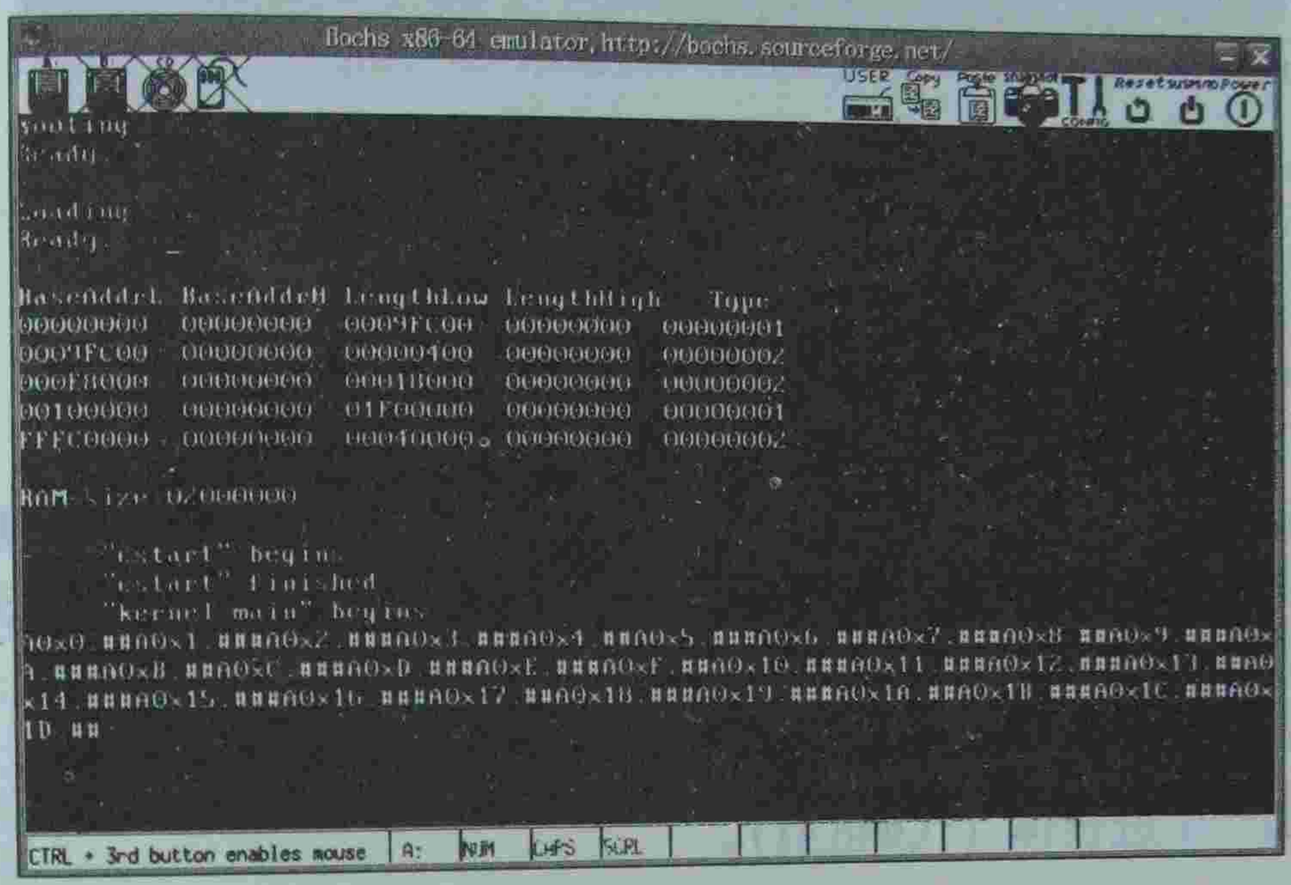


图 6.17 做好切换的准备但还未添加进程切换代码时的情形

下面该进程切换了（代码6.40）。

代码 6.40 时钟中断处理程序（chapter6/e/kernel/clock.c）

```
PUBLIC void clock_handler(int irq)
{
        disp_str("#");
        p_proc_ready++;
        if (p_proc_ready >= proc_table + NR_TASKS)
                p_proc_ready = proc_table;
}
```

在代码6.40中，每一次我们让p_proc_ready指向进程表中的下一个表项，如果切换前已经到达进程表结尾则回到第一个表项。最关键的这几行添加完后就可以看到结果了，如图6.18所示。

成功了！我们看到了交替出现的“A”和“B”，还有各自不断增加的数字。这表明我们的第二个进程运行成功了，我们已经成功实现了多进程。

毫无疑问，这又是一个历史性的时刻。因为到此为止，一个多进程的框架已经基本完成，在此基础上，你可以方便地添加任务，并且方便地设计调度算法对这些任务进行管理。从此以后，操作系统课本上的调度算法不再是空洞的理论，而变成了你手中可以随意指挥和试验的代码，任由驱使，近在眼前。

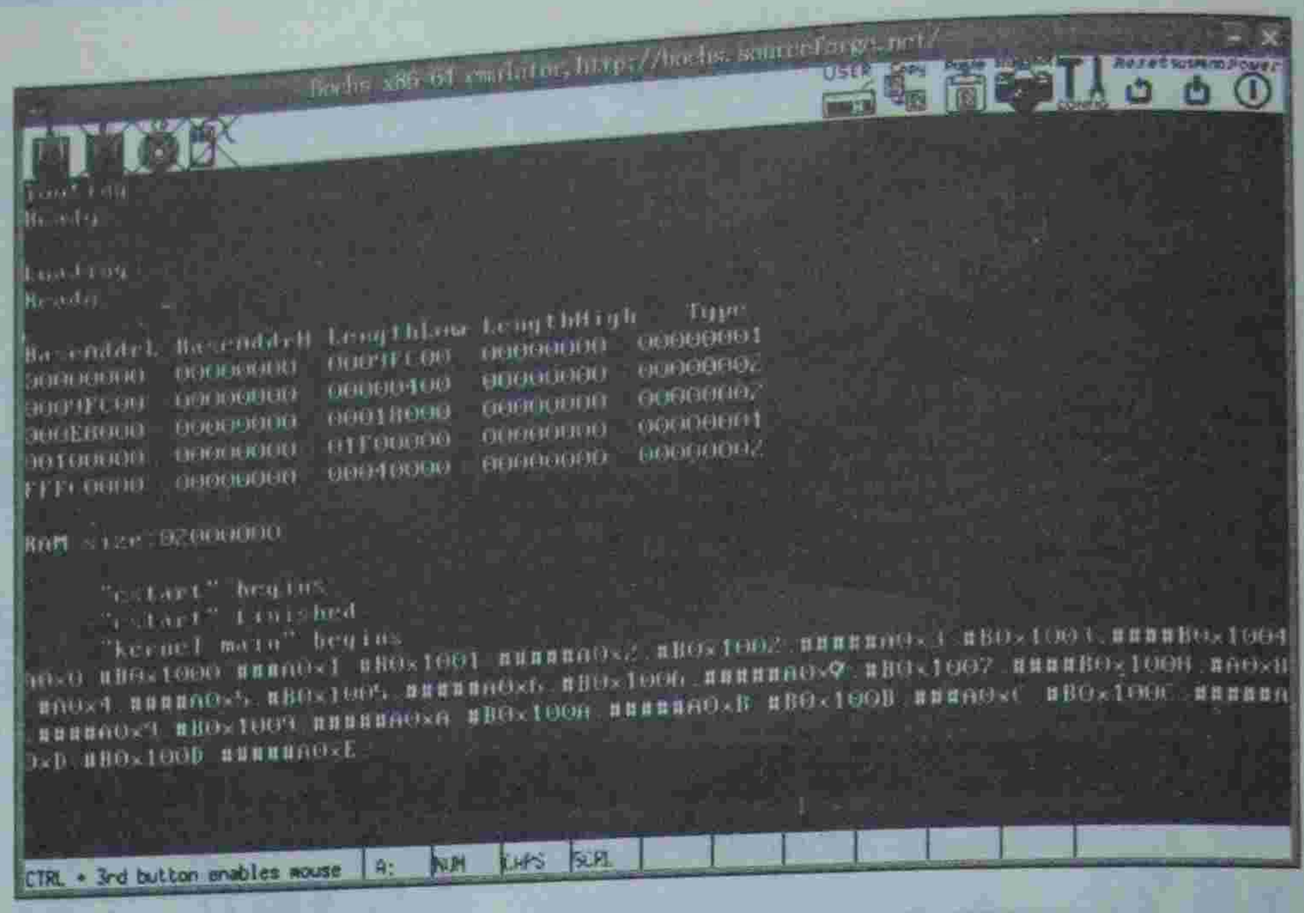


图 6.18 实现多进程

6.4.6 添加一个任务的步骤总结

如今我们已经有两个进程在运行了，一个到两个是质的飞跃，两个到三个则仅仅是量的积累，应该是容易得多了。话虽如此，但如果时隔几天，回头再来添加一个进程，没准还会丢三落四。好，我们再来添加一个任务，并且把添加一个任务的步骤总结一下。

首先添加一个进程体（代码6.41）。

代码 6.41 TestC（chapter6/f/kernel/main.c）

```
103 void TestC()
104 {
105         int i = 0x2000;
106         while(1){
107                 disp_str("C");
108                 disp_int(i++);
109                 disp_str(".");
110                 delay(1);
111         }
112 }
```

然后在task_table中添加一项进程（代码6.42）。

代码 6.42 task_table（chapter6/f/kernel/global.c）

```
22 PUBLIC  TASK    task_table[NR_TASKS] = {{TestA, STACK_SIZE_TESTA, "TestA"},
23                                        {TestB, STACK_SIZE_TESTB, "TestB"},
24                                        {TestC, STACK_SIZE_TESTC, "TestC"}};
```

然后是proc.h（代码6.43）。

代码 6.43 宏（chapter6/f/include/proc.h）

```
49 /* Number of tasks */
50 #define NR_TASKS        3
51
52 /* stacks of tasks */
53 #define STACK_SIZE_TESTA        0x8000
```

```
#define STACK_SIZE_TESTB        0x8000
#define STACK_SIZE_TESTC        0x8000

#define STACK_SIZE_TOTAL        (STACK_SIZE_TESTA + \
                                 STACK_SIZE_TESTB + \
                                 STACK_SIZE_TESTC)
```

在proto.h中添加函数声明：

```
void TestC();
```

make，成功！运行结果如图6.19所示。

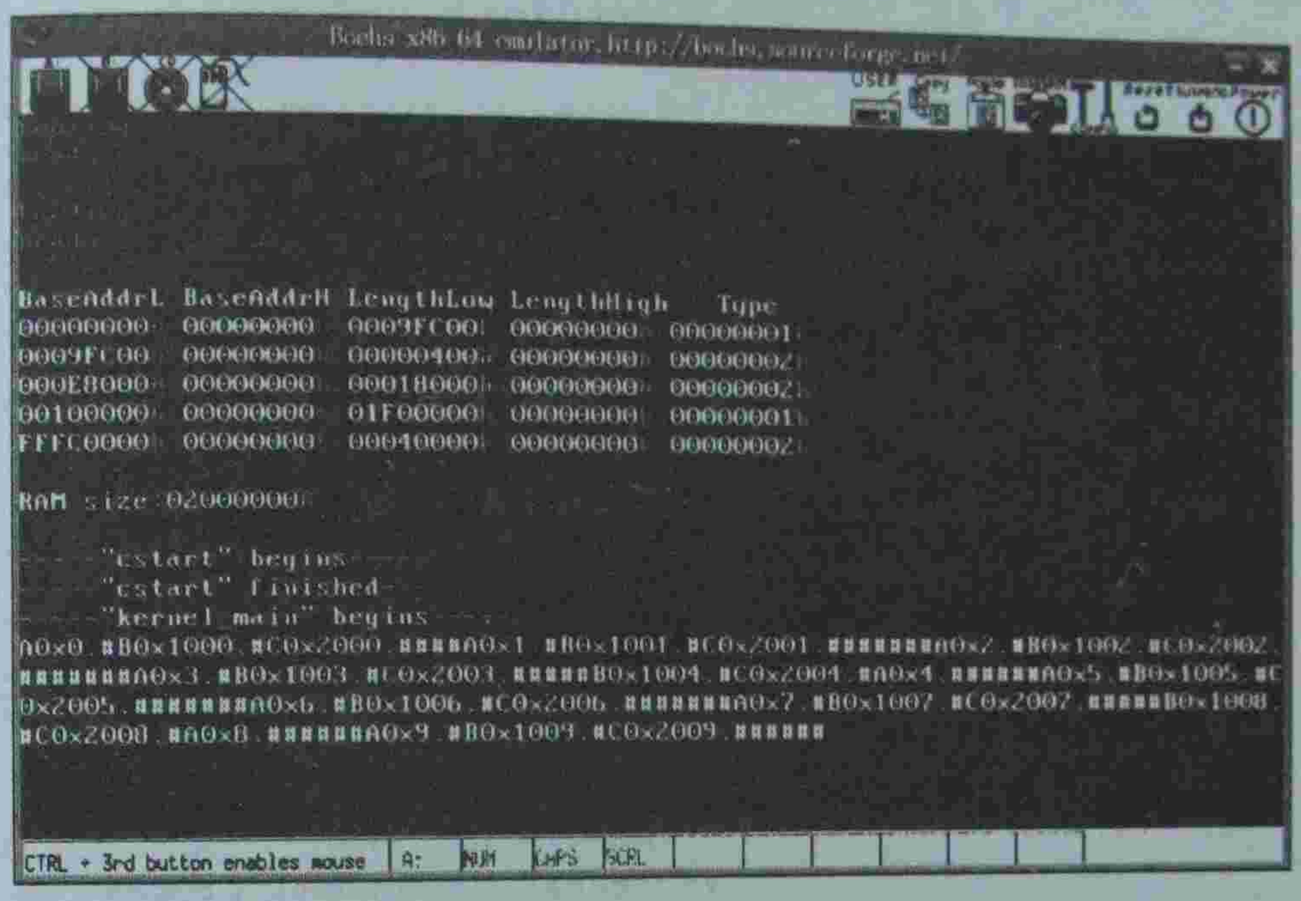


图 6.19 实现多进程

果然，简单几步就成功地添加了一个任务，我们把添加任务的步骤总结一下。增加一个任务需要的步骤：

1. 在task_table中增加一项（global.c）。
2. 让NR_TASKS加 1（proc.h）。
3. 定义任务堆栈（proc.h）。
4. 修改STACK_SIZE_TOTAL（proc.h）。
5. 添加新任务执行体的函数声明（proto.h）。

除了任务本身的代码和一些宏定义之外，原来的代码几乎不需要做任何改变，看来我们的代码的自动化程度还是不错的，这真让人高兴！

6.4.7 号外：Minix 的中断处理

在当初编写时钟中断例程的时候，不知道你有没有想过，其他的中断例程可能跟它是差不多的，因为它们也会在开始处保存当前进程的信息，在结束处

恢复一个进程，中间也会遇到中断重入、内核栈的问题等。也就是说，整个的框架是差不多的。

由于这种相似性，所有的中断例程一定可以有一种统一起来的方法。虽然这并不必要，但如果你看过 Minix 代码的话，你可能发现，这种统一充满了美感。现在我们就学习一下 Minix 是如何处理中断的。你没有读过 Minix 的代码？没有关系，下文中列出的节选足够让你理解它的中断处理机制。

代码 6.44 Minix 的中断处理（src/kernel/mpx386.s）

```
240 !*===========================================================================*
241 !*                              hwint00 - 07                                 *
242 !*===========================================================================*
243 ! Note this is a macro, it looks like a subroutine.
244 #define hwint_master(irq)       \
245         call    save                            ;\
246         inb     INT_CTLMASK                     ;\
247         orb     al, [1<<irq]                    ;\
248         outb    INT_CTLMASK     /* disable the irq          */;\
249         movb    al, ENABLE                      ;\
250         outb    INT_CTL         /* reenable master 8259     */;\
251         sti                     /* enable interrupts        */;\
252         push    irq             /* irq                      */;\
253         call    (_irq_table + 4*irq)    /* eax = (*irq_table[irq])(irq) */;\
254         pop     ecx                             ;\
255         cli                     /* disable interrupts       */;\
256         test    eax, eax        /* need to reenable irq?    */;\
257         jz      0f                              ;\
258         inb     INT_CTLMASK                     ;\
259         andb    al, ~[1<<irq]                   ;\
260         outb    INT_CTLMASK     /* enable the irq           */;\
261 0:      ret                     /* restart (another) process    */
262
263 ! Each of these entry points is an expansion of the hwint_master macro
264         .align  16
265 _hwint00:               ! Interrupt routine for irq 0 (the clock).
266         hwint_master(0)
267
268         .align  16
269 _hwint01:               ! Interrupt routine for irq 1 (keyboard)
270         hwint_master(1)
271
272         .align  16
273 _hwint02:               ! Interrupt routine for irq 2 (cascade!)
274         hwint_master(2)
275
276         .align  16
277 _hwint03:               ! Interrupt routine for irq 3 (second serial)
278         hwint_master(3)
279
280         .align  16
281 _hwint04:               ! Interrupt routine for irq 4 (first serial)
282         hwint_master(4)
283
284         .align  16
285 _hwint05:               ! Interrupt routine for irq 5 (XT winchester)
286         hwint_master(5)
287
288         .align  16
289 _hwint06:               ! Interrupt routine for irq 6 (floppy)
290         hwint_master(6)
291
292         .align  16
293 _hwint07:               ! Interrupt routine for irq 7 (printer)
294         hwint_master(7)
295
```

```
296 !*==========================================================================*
297 !*                                hwint08 - 15                              *
298 !*==========================================================================*
299 ! Note this is a macro, it looks like a subroutine.
300 #define hwint_slave(irq)        \
301         call    save                            /* save interrupted process state */;\
302         inb     INT2_CTLMASK                                                    ;\
303         orb     al, [1<<[irq-8]]                                                ;\
304         outb    INT2_CTLMASK                    /* disable the irq            */;\
305         movb    al, ENABLE                                                      ;\
306         outb    INT_CTL                         /* reenable master 8259       */;\
307         jmp     .+2                             /* delay                      */;\
308         outb    INT2_CTL                        /* reenable slave 8259        */;\
309         sti                                     /* enable interrupts          */;\
310         push    irq                             /* irq                        */;\
311         call    (_irq_table + 4*irq)            /* eax = (*irq_table[irq])(irq) */;\
312         pop     ecx                                                             ;\
313         cli                                     /* disable interrupts         */;\
314         test    eax, eax                        /* need to reenable irq?      */;\
315         jz      0f                                                              ;\
316         inb     INT2_CTLMASK                                                    ;\
317         andb    al, ~[1<<[irq-8]]                                               ;\
318         outb    INT2_CTLMASK                    /* enable the irq             */;\
319 0:      ret                                     /* restart (another) process  */
320
321 ! Each of these entry points is an expansion of the hwint_slave macro
322         .align  16
323 _hwint08:                       ! Interrupt routine for irq 8 (realtime clock)
324         hwint_slave(8)
325
326         .align  16
327 _hwint09:                       ! Interrupt routine for irq 9 (irq 2 redirected)
328         hwint_slave(9)
329
330         .align  16
331 _hwint10:                       ! Interrupt routine for irq 10
332         hwint_slave(10)
333
334         .align  16
335 _hwint11:                       ! Interrupt routine for irq 11
336         hwint_slave(11)
337
338         .align  16
339 _hwint12:                       ! Interrupt routine for irq 12
340         hwint_slave(12)
341
342         .align  16
343 _hwint13:                       ! Interrupt routine for irq 13 (FPU exception)
344         hwint_slave(13)
345
346         .align  16
347 _hwint14:                       ! Interrupt routine for irq 14 (AT winchester)
348         hwint_slave(14)
349
350         .align  16
351 _hwint15:                       ! Interrupt routine for irq 15
352         hwint_slave(15)
```

代码后面部分的_hwint00、_hwint01等就是中断例程的入口。可以看到，所有的中断例程看上去都差不多，都使用hwint_master(irq)这个宏。跟函数比较起来，使用宏虽然浪费了一些空间，但是由于避免了使用函数所必需的压栈、出栈，所以节省了时间。

hwint_master首先调用一个函数save，将寄存器的值保存起来，然后操纵8259A避免在处理当前中断的同时发生同样类型的中断。紧接着，给8259A

的中断命令寄存器发出中断结束命令（EOI）。然后，用sti指令打开中断，调用函数(*irq_table[irq])(irq)，这是与当前中断相关的一个例程。再用cli关中断、test指令判断函数(*irq_table[irq])(irq)的返回值。如果非零的话就重新打开当前发生的中断（比如发生的是时钟中断就重新打开时钟中断）；如果是零的话就直接ret。注意，最后一个指令是ret而不是iret！你可能感到有点奇怪，什么时候从中断返回呢？不着急，我们先来分析一下第一句里面调用的save语句，过一会儿你就明白了。

代码6.45 Minix的中断处理（src/kernel/mpx386.s）

```
354 /*===========================================================================*
355  *                                save                                       *
356  *===========================================================================*/
357 ! Save for protected mode.
358 ! This is much simpler than for 8086 mode, because the stack already points
359 ! into the process table, or has already been switched to the kernel stack.
360
361         .align  16
362 save:
363         cld                     ! set direction flag to a known value
364         pushad                  ! save "general" registers
365     o16 push    ds              ! save ds
366     o16 push    es              ! save es
367     o16 push    fs              ! save fs
368     o16 push    gs              ! save gs
369         mov     dx, ss          ! ss is kernel data segment
370         mov     ds, dx          ! load rest of kernel segments
371         mov     es, dx          ! kernel does not use fs, gs
372         mov     eax, esp        ! prepare to return
373         incb    (_k_reenter)    ! from -1 if not reentering
374         jnz     set_restart1    ! stack is already kernel stack
375         mov     esp, k_stktop
376         push    _restart        ! build return address for int handler
377         xor     ebp, ebp        ! for stacktrace
378         jmp     RETADR-P_STACKBASE(eax)
379
380         .align  4
381 set_restart1:
382         push    restart1
383         jmp     RETADR-P_STACKBASE(eax)
```

对于这段代码你一定觉得非常眼熟，是的，大部分的代码都是我们在中断例程中实现过的，只是需要注意的是，如果发生中断重入的话，就跳过切换内核栈的代码（因为已经在内核栈了），并且把不同的地址压栈。接下来的jmp指令也有点令人奇怪，看上去是跳到了[eax+RETADR-P_STACKBASE]处，那么eax是什么呢？向前找到第372行，这一行把esp的值赋给eax，那么esp的值是什么呢？由于当时刚刚做完寄存器的保存工作，所以esp恰恰指向进程表的起始地址。

关于RETADR-P_STACKBASE，让我们看一下RETADR和P_STACKBASE的定义（代码6.46）。

代码6.46 Minix的一些宏定义（src/kernel/sconst.h）

```
P_STACKBASE     =       0
GSREG           =       P_STACKBASE
FSREG           =       GSREG + 2
ESREG           =       FSREG + 2       ! 386 introduces FS and GS segments
DSREG           =       ESREG + 2
DIREG           =       DSREG + 2
```

```
SIREG     =     DIREG + W
BPREG     =     SIREG + W
STREG     =     BPREG + W       ! hole for another SP
BXREG     =     STREG + W
DXREG     =     BXREG + W
CXREG     =     DXREG + W
AXREG     =     CXREG + W
RETADR    =     AXREG + W       ! return address for save() call
PCREG     =     RETADR + W
CSREG     =     PCREG + W
PSWREG    =     CSREG + W
SPREG     =     PSWREG + W
SSREG     =     SPREG + W
P_STACKTOP =    SSREG + W
P_LDT_SEL =     P_STACKTOP
P_LDT     =     P_LDT_SEL + W
```

原来RETADR-P_STACKBASE就是执行call save这条指令时压栈的返回地址相对于进程表起始地址的偏移。所以[eax+RETADR-P_STACKBASE]就是返回地址，即inb INT_CTLMASK这条指令的地址。

jmp RETADR-P_STACKBASE(eax)实际上是从save函数（如果你想把它称为函数的话）返回，继续从inb INT_CTLMASK向下执行。

另外，中断重入与否的区别除了是否切换内核栈之外，还有一个push语句也不相同。我们拿其中一种情况来看一下，假设非中断重入，将会执行push _restart这一句。这是什么意思呢？还记得hwint_master(irq)最后的ret吗？原来ret指令是要跳转到_restart处。好了，我们就来看看_restart做了些什么（代码6.47）。

代码 6.47 Minix 的中断处理（src/kernel/mpx386.s）

```
420  !*===========================================================================*
421  !*                                restart                                    *
422  !*===========================================================================*
423  _restart:
424
425  ! Flush any held-up interrupts.
426  ! This reenables interrupts, so the current interrupt handler may reenter.
427  ! This does not matter, because the current handler is about to exit and no
428  ! other handlers can reenter since flushing is only done when k_reenter == 0.
429
430          cmp     (_held_head), 0 ! do fast test to usually avoid function call
431          jz      over_call_unhold
432          call    _unhold         ! this is rare so overhead acceptable
433  over_call_unhold:
434          mov     esp, (_proc_ptr)        ! will assume P_STACKBASE == 0
435          lldt    P_LDT_SEL(esp)          ! enable segment descriptors for task
436          lea     eax, P_STACKTOP(esp)    ! arrange for next interrupt
437          mov     (_tss+TSS3_S_SP0), eax  ! to save state in process table
438  restart1:
439          decb    (_k_reenter)
440      o16 pop     gs
441      o16 pop     fs
442      o16 pop     es
443      o16 pop     ds
444          popad
445          add     esp, 4          ! skip return adr
446          iretd                   ! continue process
```

看到这里，你一定已经明白了一切，因为我们的代码几乎与它相同，在这里就不必多说了。相应地，如果发生中断重入的话，就会跳到restart1处执

行，不再进行进程的切换。

虽然这些代码看上去不多，但每一句代码都很重要，甚至有着深意。一段复杂的代码不可能是一蹴而就的，本章用大量的篇幅，本着循序渐进的原则，就是在努力将每一个语句形成的原因讲述清楚，希望读者可以用比较短的时间理解这些内容，而且理解它们的由来。

6.4.8 代码回顾与整理

之所以在这里加入对 Minix 代码的解释，是因为我想在自己的操作系统中模仿它的这种中断处理方式。显然，我们先前的中断处理有点太粗糙了，而 Minix 这部分代码则不但显得优雅，而且思路更加清晰。好了，我们就着手对代码进行改造。

首先，我们已经有了一个restart，而且与中断例程的最后一部分基本一致，那么我们就先把这两块合二为一。

首先修改中断例程（代码6.48）。

代码 6.48 修改时钟中断处理程序（chapter6/g/kernel/kernel.asm）

```
158 hwint00:                ; Interrupt routine for irq 0 (the clock).
159         sub     esp, 4
160         pushad          ; `.
161         push    ds      ;  |
162         push    es      ;  | 保存原寄存器值
163         push    fs      ;  |
164         push    gs      ; /
165         mov     dx, ss
166         mov     ds, dx
167         mov     es, dx
168
169         inc     byte [gs:0]             ; 改变屏幕第 0 行，第 0 列的字符
170
171         mov     al, EOI
172         out     INT_M_CTL, al           ; `. reenable
173                                         ; /  master 8259
174
175         inc     dword [k_reenter]
⇒ 176       cmp     dword [k_reenter], 0
177         jne     .1      ; 重入时就到.1，通常情况下顺序执行
178         ;jne    .re_enter
179
180         mov     esp, StackTop           ; 切到内核栈
⇒ 181       push    .restart_v2
⇒ 182       jmp     .2
⇒ 183 .1:
⇒ 184
⇒ 185       push    .restart_reenter_v2     ; 中断重入
186 .2:
187         sti                             ; 没有中断重入
188
189         push    0
190         call    clock_handler
191         add     esp, 4
192
193         cli
194
⇒ 195       ret     ; 重入时就到.restart_reenter_v2，通常情况下到.restart_v2
196 .restart_v2:
197         mov     esp, [p_proc_ready]
198         lldt    [esp + P_LDT_SEL]       ; 离开内核栈
```

```
199         lea     eax, [esp + P_STACKTOP]
200         mov     dword [tss + TSS3_S_SP0], eax
201
202 .restart_reenter_v2:                         ; 如果(k_reenter != 0)，会跳转到这里
203 ;.re_enter:     ; 如果(k_reenter != 0)，会跳转到这里
204         dec     dword [k_reenter]
205         pop     gs          ; '.
206         pop     fs          ;  |
207         pop     es          ;  | 恢复原寄存器值
208         pop     ds          ;  |
209         popad               ; /
210         add     esp, 4
211
212         iretd
```

需要注意的是，这里不仅仅是形式上的修改，内容也有了一些变化：原先的程序当发生中断重入的时候是不执行clock_handler的，而现在则总是在执行。所以，我们还需要在clock_handler中稍做修改（代码6.49）。

代码 6.49 时钟中断处理程序（chapter6/g/kernel/clock.c）

```
20  PUBLIC void clock_handler(int irq)
21  {
22          disp_str("#");
23
24          if (k_reenter != 0) {
25                  disp_str("!");
26                  return;
27          }
28
29          p_proc_ready++;
30          if (p_proc_ready >= proc_table + NR_TASKS) {
31                  p_proc_ready = proc_table;
32          }
33  }
```

当发生中断重入的时候，本函数会直接返回，不做任何操作。在返回前加入了打印字符“!”的代码，只是为了发生中断重入的时候可以直观地看到而已。

此时make并运行一下，结果仍如图6.19所示。

下面我们再来修改restart。

为了将来合二为一，我们将它修改得几乎与中断例程中的最后一段一模一样，增加了一行代码和一个标号（代码6.51）。需要注意的是，既然在进程第一次运行之前执行了dec dword [k_reenter]，就必须把k_reenter的初始化值修改一下（代码6.50）。

代码 6.50 k_reenter（chapter6/h/kernel/main.c）

```
20  PUBLIC int kernel_main()
21  {
    ...
64          k_reenter = 0;
    ...
70  }
```

代码 6.51 修改时钟中断处理程序（chapter6/h/kernel/kernel.asm）

```
353 restart:
```

```
354         mov     esp, [p_proc_ready]
355         lldt    [esp + P_LDT_SEL]
356         lea     eax, [esp + P_STACKTOP]
357         mov     dword [tss + TSS3_S_SP0], eax
358 restart_reenter:
359         dec     dword [k_reenter]
360         pop     gs
361         pop     fs
362         pop     es
363         pop     ds
364         popad
365         add     esp, 4
366         iretd
```

现在如果对比代码6.48和代码6.51就会发现，两段代码的最后部分除了标号的名字不同，其余都是相同的，我们完全可以删掉其中一段，把代码6.48中重复的部分删掉，同时修改用到标号.restart_v2和.restart_reenter_v2的地方。

代码 6.52 修改时钟中断处理程序（chapter6/h/kernel/kernel.asm）

```
158 hwint00:                ; Interrupt routine for irq 0 (the clock).
159         sub     esp, 4
160         pushad          ; `.
161         push    ds      ;  |
162         push    es      ;  | 保存原寄存器值
163         push    fs      ;  |
164         push    gs      ; /
165         mov     dx, ss
166         mov     ds, dx
167         mov     es, dx
168
169         inc     byte [gs:0]           ; 改变屏幕第 0 行，第 0 列的字符
170
171         mov     al, EOI
172         out     INT_M_CTL, al         ; `. reenable
173                                       ; /  master 8259
174         inc     dword [k_reenter]
175         cmp     dword [k_reenter], 0
176         jne     .1          ; 重入时跳到.1，通常情况下顺序执行
177
178         mov     esp, StackTop         ; 切到内核栈
179
180         push    restart
181         jmp     .2
182 .1:
183         push    restart_reenter       ; 中断重入
184 .2:
185         sti                           ; 没有中断重入
186
187         push    0
188         call    clock_handler
189         add     esp, 4
190
191         cli
192
193         ret
194
195 ;; .restart_v2:
196 ;;         mov     esp, [p_proc_ready]     ; 离开内核栈
197 ;;         lldt    [esp + P_LDT_SEL]
198 ;;         lea     eax, [esp + P_STACKTOP]
199 ;;         mov     dword [tss + TSS3_S_SP0], eax
200
201 ;; .restart_reenter_v2:                    ; 如果(k_reenter != 0)，会跳转到这里
```

```
;; ;.re_enter:   ; 如果(k_reenter != 0)，会跳转到这里
;;        dec     dword [k_reenter]
;;        pop     gs          ; `.
;;        pop     fs          ;  |
;;        pop     es          ;  | 恢复原寄存器值
;;        pop     ds          ;  |
;;        popad               ; /
;;        add     esp, 4
;;
          iretd
;;
```

现在，我们删除掉了hwint00中的最后一段代码，并修改了涉及删除代码中标号的两句指令。

在这里编译并运行，成功，看到的现象与原先别无二致。

原来长长的中断例程，如今已经被分离出了一个restart。现在我们再来分离save。当准备开始时，我想你一定又一次注意到了开头第一个语句：

```
sub esp, 4
```

我们已经提过，这个语句是跳过了进程表中的一个成员。如今我们已经读过了 Minix 的代码，应该已经明白了，这个成员其实就是由call save语句产生的、被压栈的返回地址。现在就把开头这部分代码挪进save函数中（代码6.53）。

代码 6.53 修改时钟中断处理程序（chapter6/i/kernel/kernel.asm）

```
hwint00:                ; Interrupt routine for irq 0 (the clock).
        call    save

        mov     al, EOI                 ; `. reenable
        out     INT_M_CTL, al           ; /  master 8259

        sti
        push    0
        call    clock_handler
        add     esp, 4
        cli

        ret
...
save:
        pushad          ; `.
        push    ds      ;  |
        push    es      ;  | 保存原寄存器值
        push    fs      ;  |
        push    gs      ; /
        mov     dx, ss
        mov     ds, dx
        mov     es, dx

        mov     eax, esp                ;eax = 进程表起始地址

        inc     dword [k_reenter]       ;k_reenter++;
        cmp     dword [k_reenter], 0    ;if(k_reenter ==0)
        jne     .1                      ;{
        mov     esp, StackTop           ;  mov esp, StackTop <--切换到内核栈
        push    restart                 ;  push restart
        jmp     [eax + RETADR - P_STACKBASE]; return;
.1:                                     ;} else { 已经在内核栈，不需要再切换
        push    restart_reenter         ;  push restart_reenter
        jmp     [eax + RETADR - P_STACKBASE]; return;
                                        ;}
```

现在来考虑一下，为什么这个save与我们以前的函数看起来是如此的不同？一般的函数最后都是以ret指令结尾，跳回调用处继续执行，那是因为函数所使用的堆栈最后都被释放了，调用时call指令的下一条指令地址被压栈，最后ret指令将这条指令从堆栈中弹出，函数调用前后esp的值是一样的。可是我们这里的save函数则不同，调用前后esp的值是完全不同的，甚至是否发生中断重入也影响着esp的值。所以我们必须事先将返回地址保存起来，最后用jmp指令跳转回去。

在上面这段代码中，注释被写成了C语言的格式，这样读者可以比较清晰地了解代码的逻辑。

save函数准备好之后，让我们继续修改中断例程（代码6.54）。

代码6.54 修改时钟中断处理程序（chapter6/j/kernel/kernel.asm）

```
158 hwint00:                ; Interrupt routine for irq 0 (the clock).
159         call    save
160
⇒ 161       in      al, INT_M_CTLMASK       ; .
⇒ 162       or      al, 1                   ;  | 不允许再发生时钟中断
⇒ 163       out     INT_M_CTLMASK, al       ; /
164
165         mov     al, EOI                 ; . reenable
166         out     INT_M_CTL, al           ; /  master 8259
167
168         sti
169         push    0
170         call    clock_handler
171         add     esp, 4
172         cli
173
⇒ 174       in      al, INT_M_CTLMASK       ; .
⇒ 175       and     al, 0xFE                ;  | 又允许时钟中断发生
⇒ 176       out     INT_M_CTLMASK, al       ; /
177
178         ret
```

在这里添加了两段代码，在调用clock_handler之前屏蔽掉时钟中断，在调用之后重又打开。这样，在只打开时钟中断的时候不再会发生中断重入的情况，但是可以预料，当其他中断被打开的时候，中断重入的情况仍然可能出现，我们对它的处理仍然有必要。

到这里，其实我们的时钟中断处理程序已经与Minix的hwint_master差不多了，现在，我们也把它改成一个类似的宏，用它替换原有的宏，并且修改中断例程（代码6.55）。

代码6.55 修改时钟中断处理程序（chapter6/k/kernel/kernel.asm）

```
170 hwint00:                ; Interrupt routine for irq 0 (the clock).
171         hwint_master    0
```

新的宏如代码6.56所示。

代码6.56 修改时钟中断处理程序（chapter6/k/kernel/kernel.asm）

```
19  extern  irq_table
    ...
150 %macro  hwint_master    1
151         call    save
152         in      al, INT_M_CTLMASK
```

```
        or      al, (1 << %1)           ;   | 屏蔽当前中断
        out     INT_M_CTLMASK, al       ; /
        mov     al, EOI                 ; `. 置EOI位
        out     INT_M_CTL, al           ; /
        sti     ; CPU在响应中断的过程中会自动关中断，这句之后就允许响应新的中断
        push    %1                      ; .
        call    [irq_table + 4 * %1]    ;   | 中断处理程序
        pop     ecx                     ; /
        cli
        in      al, INT_M_CTLMASK       ; .
        and     al, ~(1 << %1)          ;   | 恢复接受当前中断
        out     INT_M_CTLMASK, al       ; /
        ret
endmacro
```

这里，新引入了一个函数指针数组irq_table（定义在global.c中）：

```
PUBLIC irq_handler irq_table[NR_IRQ];
```

别忘了在global.h中加入声明：

```
extern irq_handler irq_table[];
```

其中，irq_handler在type.h中这样定义：

```
typedef void (*irq_handler) (int irq);
```

这与我们的clock_handler类型是完全一致的。

NR_IRQ的值定义为16，以对应主从两个8259A（定义在const.h中）：

```
#define NR_IRQ 16
```

现在，虽然已经定义了irq_table，但它还没有被赋以任何的值，我们需要有16个函数来初始化它，可目前只有一个clock_handler。不要紧，我们把剩余的元素全部赋值为spurious_irq就行了。

好了，现在我们就来初始化irq_table。这项工作分为两部分，首先将所有的元素初始化为spurious_irq，然后再处理需要单独赋值的元素（代码6.57）。

代码6.57 初始化irq_table（chapter6/k/kernel/i8259.c）

```
PUBLIC void init_8259A()
{
 ...

        int i;
        for (i = 0; i < NR_IRQ; i++)
                irq_table[i] = spurious_irq;
}
```

下面到了单独为irq_table[0]，也就是时钟中断赋值的时候了。先写一个函数put_irq_handler来为irq_table赋值（代码6.58）。

代码6.58 put_irq_handler（chapter6/k/kernel/i8259.c）

```
PUBLIC void put_irq_handler(int irq, irq_handler handler)
{
        disable_irq(irq);
        irq_table[irq] = handler;
}
```

之所以新添加一个函数来做这项工作，一方面是因为这个操作不能用一条语句完成；另一方面，这样的思想有点类似于 C++ 中的封装机制。

这里，我们新添加的一个函数disable_irq和我们马上要见到的另一个函数enable_irq都在klib.asm中（用汇编语言编写，也是从 Minix 相应代码学习而来，详见代码6.59）。

代码 6.59 disable_irq 和 enable_irq（chapter6/k/lib/kliba.asm）

```
8   %include "sconst.inc"
    ...
21  global  enable_irq
22  global  disable_irq
    ...
123 ; ========================================================================
124 ;                   void disable_irq(int irq);
125 ; ========================================================================
126 ; Disable an interrupt request line by setting an 8259 bit.
127 ; Equivalent code:
128 ;       if(irq < 8)
129 ;               out_byte(INT_M_CTLMASK, in_byte(INT_M_CTLMASK) | (1 << irq));
130 ;       else
131 ;               out_byte(INT_S_CTLMASK, in_byte(INT_S_CTLMASK) | (1 << irq));
132 disable_irq:
133         mov     ecx, [esp + 4]          ; irq
134         pushf
135         cli
136         mov     ah, 1
137         rol     ah, cl                  ; ah = (1 << (irq % 8))
138         cmp     cl, 8
139         jae     disable_8               ; disable irq >= 8 at the slave 8259
140 disable_0:
141         in      al, INT_M_CTLMASK
142         test    al, ah
143         jnz     dis_already             ; already disabled?
144         or      al, ah
145         out     INT_M_CTLMASK, al       ; set bit at master 8259
146         popf
147         mov     eax, 1                  ; disabled by this function
148         ret
149 disable_8:
150         in      al, INT_S_CTLMASK
151         test    al, ah
152         jnz     dis_already
153         or      al, ah                  ; already disabled?
154         out     INT_S_CTLMASK, al
155         popf                            ; set bit at slave 8259
156         mov     eax, 1
157         ret                             ; disabled by this function
158 dis_already:
159         popf
160         xor     eax, eax
161         ret                             ; already disabled
162
163 ; ========================================================================
164 ;                   void enable_irq(int irq);
165 ; ========================================================================
166 ; Enable an interrupt request line by clearing an 8259 bit.
167 ; Equivalent code:
168 ;       if(irq < 8)
169 ;               out_byte(INT_M_CTLMASK, in_byte(INT_M_CTLMASK) & ~(1 << irq));
170 ;       else
171 ;               out_byte(INT_S_CTLMASK, in_byte(INT_S_CTLMASK) & ~(1 << irq));
172 ;
173 enable_irq:
174         mov     ecx, [esp + 4]          ; irq
```

```
        pushf
        cli
        mov     ah, ~1
        rol     ah, cl                  ; ah = ~(1 << (irq % 8))
        cmp     cl, 8
        jae     enable_8                ; enable irq >= 8 at the slave 8259
enable_0:
        in      al, INT_M_CTLMASK
        and     al, ah
        out     INT_M_CTLMASK, al       ; clear bit at master 8259
        popf
        ret
enable_8:
        in      al, INT_S_CTLMASK
        and     al, ah
        out     INT_S_CTLMASK, al       ; clear bit at slave 8259
        popf
        ret
```

现在在kernel_main()中指定时钟中断处理程序（代码6.60）。

代码 6.60 时钟中断相关（chapter6/k/kernel/main.c）

```
PUBLIC int kernel_main()
{
 ...
        put_irq_handler(CLOCK_IRQ, clock_handler); /* 设定时钟中断处理程序 */
        enable_irq(CLOCK_IRQ);                     /* 让8259A可以接收时钟中断 */
 ...
}
```

这两行不但指定了时钟中断处理程序，而且让 8259A 可以接收时钟中断。

既然我们用disable_irq和enable_irq这两个函数来控制 8259A 对中断的接收情况了，那就应该在init_8259A()中屏蔽 8259A 的所有中断：

```
out_byte(INT_M_CTLMASK, 0xFF);
```

到此为止，代码的修改就告一段落了。注意添加相应的函数声明，编译并运行，结果如图6.20所示。

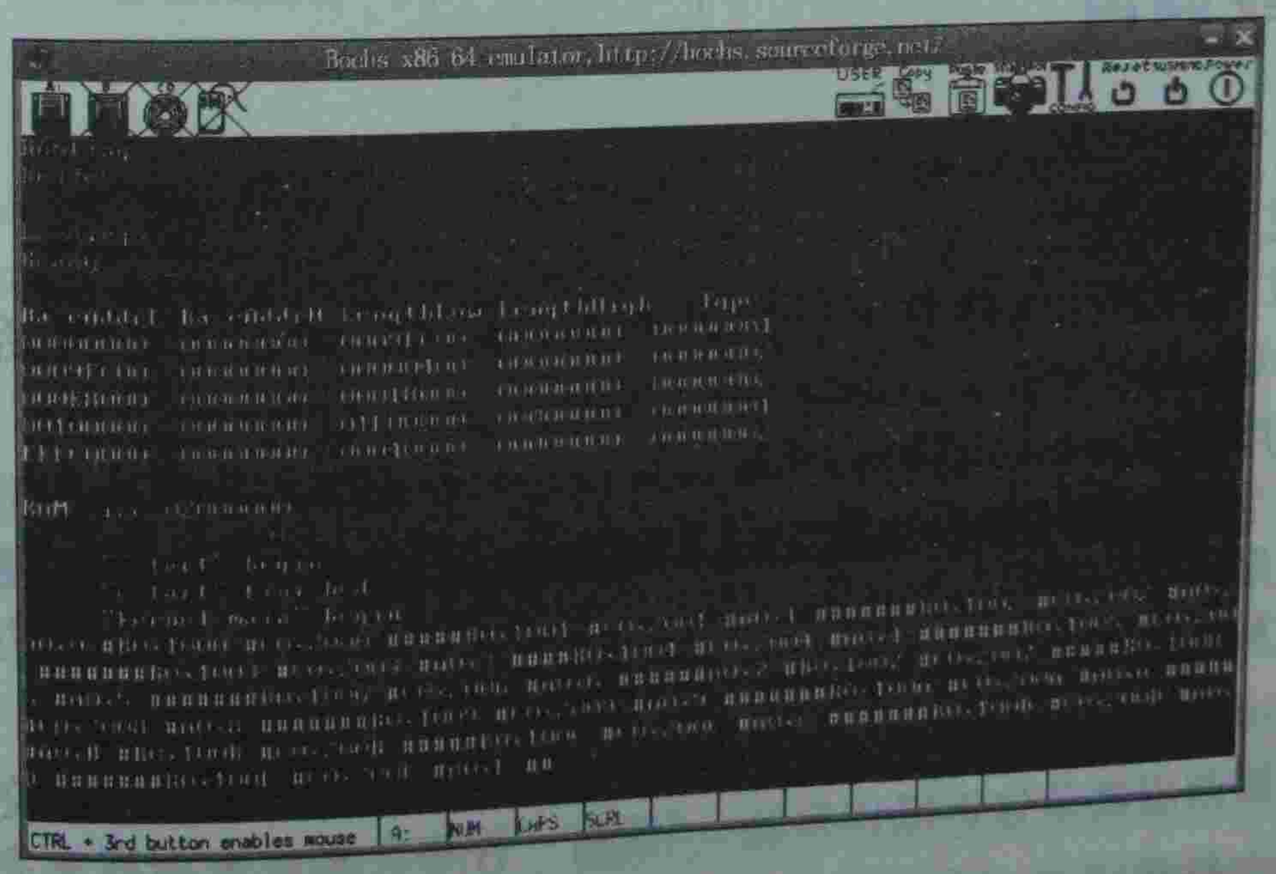


图 6.20 实现多进程

结果虽然跟原先大致相同，但是现在的代码不但更有条理，而且更容易扩展。实际上，现在我们完成的绝不仅仅是一个时钟中断处理程序而已，同时也是一套方便扩展的中断处理的接口。现在，若想添加某个中断处理模块，只需要将完成中断处理的函数入口地址赋给irq_table中相应的元素就够了。这个函数可能仍然非常底层，但已经可以完全用C语言编写。

应该说，到这里才真正算是里程碑式的成果。我们之前已经有过太多的中间成果，但是回头想想，现在的Orange'S已经可以随意地增加进程的数目，已经预留了足够方便的中断处理接口。也就是说，虽然它仍算不上是完整意义上的操作系统，但是一个粗糙的框架已经形成。我们历尽艰辛，到这里终于可以稍微喘一口气了。好，现在就让我们一起回忆一下我们可爱的操作系统是怎样运转起来的，如图6.21所示。

如果顺着这个图表把整个过程重新过一遍的话，你可能发现，涉及的代码并没有感觉上那么多，但是，要彻底把它写出来并不是一件非常容易的事情，其中最困难的就是时钟中断处理程序围绕进程表项进行进程切换的过程，这一点我们已经深有体会。还好我们一步一步走过来了，而且，可以想见，时钟中断处理程序在以后应该不需要很大的改变，这多少让我们感到欣慰。

不过我们的系统无论如何还是非常幼稚的。回头想想，幼稚的原因在于我们在进程本身方面考虑得还比较少，比如，未曾考虑过进程优先级问题，未曾考虑进程间通信的问题等。现在让我们对照图6.21所示的图表来想像一下，如果增加这些内容的话，大致应该是怎样的情形。

切换到Kernel的GDT的代码通常情况下是不需要改动的，init_prot()中的init_8259A()看上去是比较稳定的代码，kernel_main()结构很简单，但由于在这里初始化了进程表，所以若对进程功能进行扩展的话，会有一些改动。我们在本来就不多的代码中，只想到这一处地方可能会在进程功能扩展时有所改动，这真是一件让人高兴的事情，因为这意味着，即便目前的工作告一段落，等到下一次想要进一步完善它的时候，上手也会比较容易，因为接口已经足够简单了。

6.5 系统调用

如果你对Linux不太熟悉而只在Windows下写过程序，那么可能你不熟悉“系统调用”这个概念，不过你一定听说过API。在Windows中，应用程序通过调用API与操作系统建立联系，比如，弹出一个对话框可以使用MessageBoxA。

系统调用与此类似。在我们的操作系统中，已经存在的3个进程是运行在ring1上的，它们已经不能任意地使用某些指令，不能访问某些权限更高的内存区域，但如果一项任务需要这些使用指令或者内存区域时，该怎么办呢？这时候只能通过系统调用来实现，它是应用程序和操作系统之间的桥梁。

换句话说，应用程序的能力是有限的，很多事情做不了，只能交给操作系统来做。系统调用就是告诉操作系统：“我有一件事，请你来帮我来完成。”

所以，一件事情就可能是应用程序做一部分，操作系统做一部分。这样，问题就又涉及特权级变换。

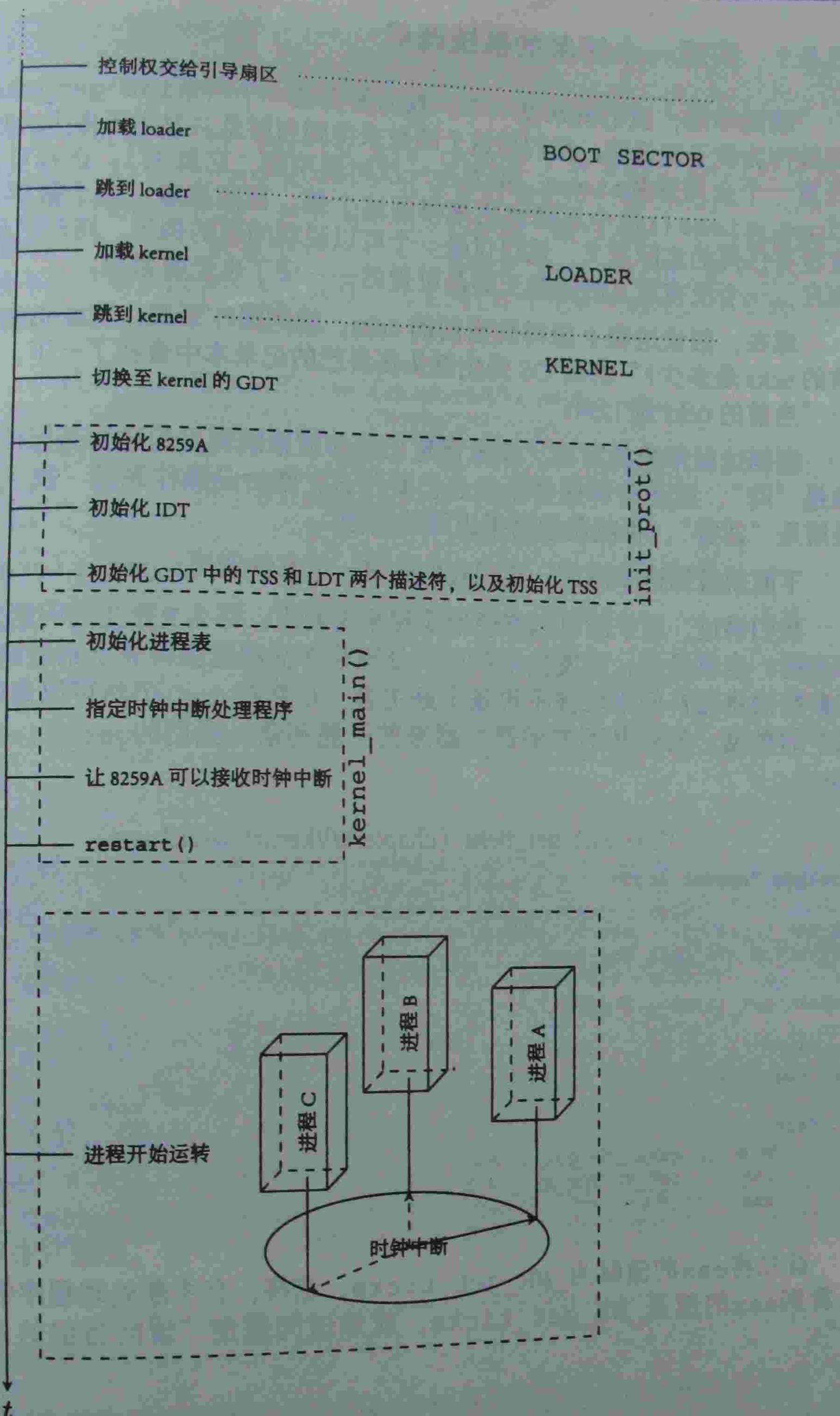


图 6.21 Orange'S的运转过程

很明显，这已经难不倒我们了，因为进程的切换就是不停地在重复这么一个特权级变换的过程。在那里，触发变换的是外部中断，我们把这个诱因换一下就可以了，变成“int nnn”，一切都就解决了。

6.5.1 实现一个简单的系统调用

既然不难，我们就来做一下。我们用实现一个叫做int get_ticks()的函数作为例子来说明。我们用这个函数来得到当前总共发生了多少次时钟中断。设置一个全局变量ticks，每发生一次时钟中断，它就加1。进程可以随时通过get_ticks()这个系统调用来得到这个值。目前来看，这个数字对我们而言没有任何的实际意义，这仅仅是一个可以说明问题的例子，同时它足够简单。而且，一个没有意义的函数无疑是可爱的——错了也无关紧要。

现在，假设进程P想得到当前的ticks，就问操作系统："OS先生，请问当前的ticks是多少？"然后OS先生低头在自己的记录本中查找了一下，然后告诉P："当前的ticks是1234。"

根据这段形象的对话，读者肯定已经知道系统调用的过程是怎样的了。首先是"问"，就是告诉操作系统自己要什么；然后是操作系统"找"，即处理；最后是"回答"，也就是把结果返回给进程。

下面就按照这个顺序，实现我们的第一个系统调用：get_ticks。

我们讲过，用中断可以方便地实现系统调用。那么P先生的问题怎么让OS知道呢？也就是说，当发生中断后，处理程序从哪里获得关于系统调用本身及其参数信息呢？使用堆栈不再是个好主意，虽然在ring0仍然可以读取ring1堆栈中的数据，但还是不如用寄存器来的干脆利落。我们的get_ticks()见代码6.61。

代码6.61 get_ticks（chapter6/l/kernel/syscall.asm）

```
8   %include "sconst.inc"
9
10  _NR_get_ticks       equ 0 ; 要跟 global.c 中 sys_call_table 的定义相对应!
11  INT_VECTOR_SYS_CALL equ 0x90
12
13  global  get_ticks ; 导出符号
14
15  bits 32
16  [section .text]
17
18  get_ticks:
19          mov     eax, _NR_get_ticks
20          int     INT_VECTOR_SYS_CALL
21          ret
```

首先将eax的值赋为_NR_get_ticks，这样，在中断处理程序中，OS先生看到eax的值是_NR_get_ticks，就知道问题是"请问当前的ticks是多少"。

这里将系统调用对应的中断号设为0x90，它只要不和原来的中断号重复即可，Linux相应的中断号是0x80。

马上来定义INT_VECTOR_SYS_CALL对应的中断门，在init_prot()中，紧跟初始化其他中断门的语句（代码6.62）。

代码6.62 初始化系统调用的中断门（chapter6/l/kernel/protect.c）

```
61  PUBLIC void init_prot()
62  {
    ...
162         init_idt_desc(INT_VECTOR_SYS_CALL,      DA_386IGate,
```

```
163                     sys_call,                       PRIVILEGE_USER);
    ...
187 }
```

这样我们就将第INT_VECTOR_SYS_CALL号中断与sys_call对应起来了。那么sys_call应该如何实现呢？其实我们完全可以模仿hwint_master宏来做：先保存寄存器的值，然后调用相应的函数，最后返回。不过看看save，我们注意到一个细节，就是里面有一条语句mov eax, esp改变了eax的值。这显然是不允许的，因为eax存放着 P 先生问 OS 先生的问题呢。不要紧，我们把eax统统改成esi（代码6.63）。

代码 6.63 修改后的 save（chapter6/l/kernel/kernel.asm）

```
314 save:
315         pushad          ; `.
316         push    ds      ;  |
317         push    es      ;  | 保存原寄存器值
318         push    fs      ;  |
319         push    gs      ; /
320         mov     dx, ss
321         mov     ds, dx
322         mov     es, dx
323
⇒ 324       mov     esi, esp                    ;esi = 进程表起始地址
325
326         inc     dword [k_reenter]           ;k_reenter++;
327         cmp     dword [k_reenter], 0        ;if(k_reenter ==0)
328         jne     .1                          ;{
329         mov     esp, StackTop               ;  mov esp, StackTop <--切换到内核栈
330         push    restart                     ;  push restart
⇒ 331       jmp     [esi + RETADR - P_STACKBASE];  return;
332 .1:                                         ;} else { 已经在内核栈，不需要再切换
333         push    restart_reenter             ;  push restart_reenter
⇒ 334       jmp     [esi + RETADR - P_STACKBASE];  return;
335                                             ;}
```

现在可以写sys_call了，请看代码6.64。

函数sys_call基本上是hwint_master的简化，甚至连对相应处理程序的调用都类似，在hwint_master中是call [irq_table+4*%1]（即调用了irq_table[%1]），这里变成了call [sys_call_table+eax*4]（调用的是sys_call_table [eax]）。与irq_table类似，sys_call_table是一个函数指针数组，每一个成员都指向一个函数，用以处理相应的系统调用。

代码 6.64 sys_call（chapter6/l/kernel/kernel.asm）

```
28  extern  sys_call_table
    ...
44  global sys_call
    ...
341 sys_call:
342         call    save
343
344         sti
345
346         call    [sys_call_table + eax * 4]
347         mov     [esi + EAXREG - P_STACKBASE], eax
348
349         cli
350
351         ret
```

sys_call_table的定义在global.c中，实际上目前它只有一个成员：

```
PUBLIC system_call sys_call_table[NR_SYS_CALL] =
                        {sys_get_ticks};
```

其中，system_call是在type.h中定义的：

```
typedef void* system_call;
```

这样，无论系统调用何种函数，都不会有编译时错误。

前面eax已被赋值为_NR_get_ticks（即0），而sys_call_table[0]已经初始化为sys_get_ticks，所以call [sys_call_table+eax*4]这一句调用的便是sys_get_ticks。由于ticks看上去是与进程相关的东西，我们就单独建立一个文件proc.c，把sys_get_ticks放在里面。为简单起见，暂时让这个函数最简，打印一个字符“+”后就返回，不做其他任何操作（代码6.65）。

代码6.65 sys_get_ticks（chapter6/l/kernel/proc.c）

```
20  PUBLIC int sys_get_ticks()
21  {
22          disp_str("+");
23          return 0;
24  }
```

继续看代码6.64，如果mov [esi+EAXREG-P_STACKBASE],eax这一句你还是不明白，回顾一下save中的jmp [esi+RETADR-P_STACKBASE]就知道了，其实它是把函数sys_call_table[eax]的返回值放在进程表中eax的位置，以便进程P被恢复执行时eax中装的是正确的返回值。

下面在proto.h中添加函数声明（代码6.66）。

代码6.66 函数声明（chapter6/l/include/proto.h）

```
37  /* 以下是系统调用相关 */
38
39  /* proc.c */
40  PUBLIC  int     sys_get_ticks();        /* sys_call */
41
42  /* syscall.asm */
43  PUBLIC  void    sys_call();             /* int_handler */
44  PUBLIC  int     get_ticks();
```

好，现在可以在进程中添加调用get_ticks的代码了，来到TestA中添加如下语句（代码6.67）。

代码6.67 TestA（chapter6/l/kernel/main.c）

```
79  void TestA()
80  {
81          int i = 0;
82          while (1) {
83                  get_ticks();
84                  disp_str("A");
85                  disp_int(i++);
86                  disp_str(".");
87                  delay(1);
88          }
89  }
```

再次提醒一句，别忘了在kernel.asm和syscall.asm中导入和导出相应符号，并且修改Makefile（增加了一个文件proc.c）。然后就可以make并运行了，结果如图6.22所示。

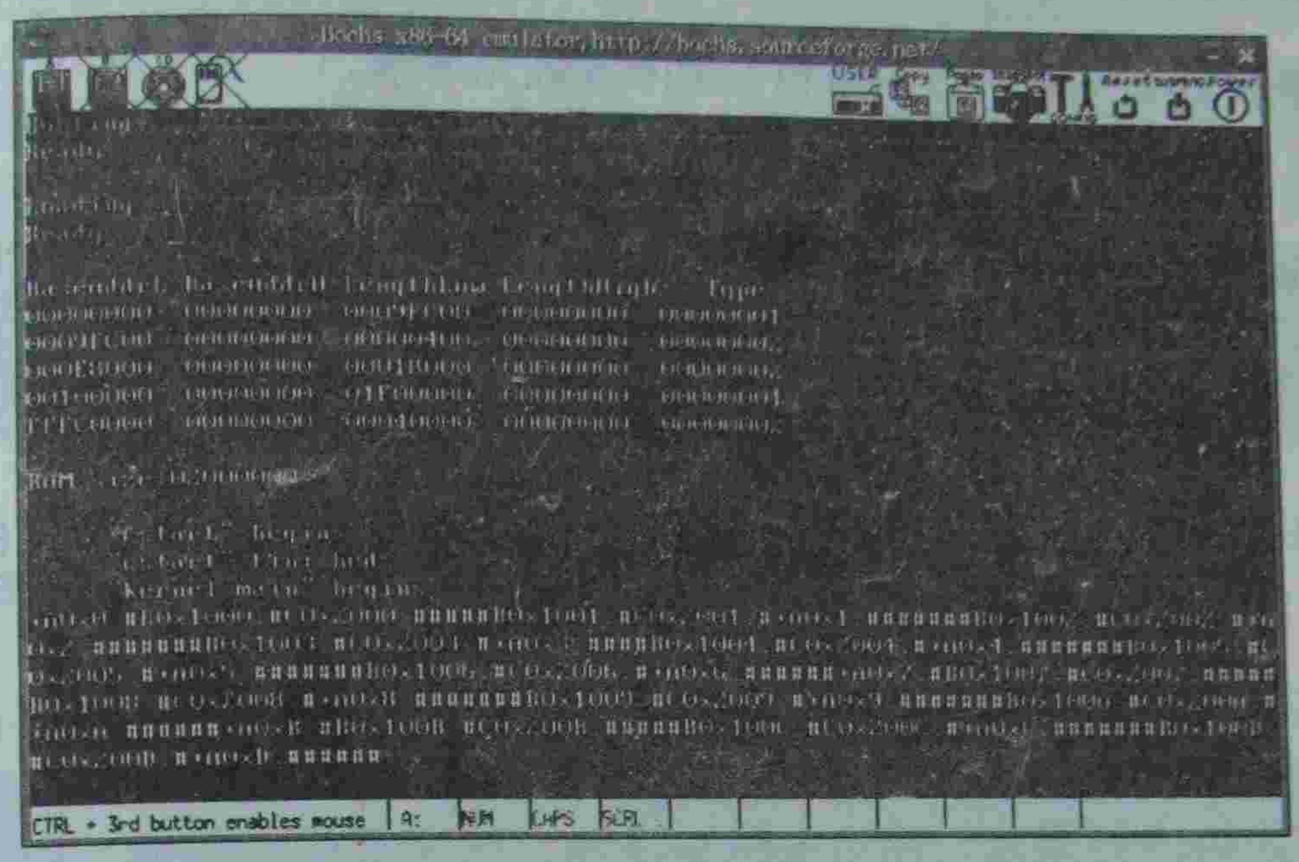


图 6.22 第一个系统调用开始运行

从图6.22中我们看到，加号出现在字符 A 的前面。这说明，我们的第一个系统调用成功了！

下面我们来改进一下函数sys_get_ticks()，让它发挥应有的作用。它要返回的是当前的 ticks，但是我们还没有声明这样的全局变量呢，如何是好？马上来到global.h：

```
EXTERN int ticks;
```

在main.c中初始化之（代码6.68）。

代码 6.68 初始化 ticks（chapter6/m/kernel/main.c）

```
20  PUBLIC int kernel_main()
21  {
    ...
65          ticks = 0;
    ...
75  }
```

在clock_handler(int irq)中添加一句（代码6.69）。

代码 6.69 时钟中断处理程序（chapter6/m/kernel/clock.c）

```
20  PUBLIC void clock_handler(int irq)
21  {
22
23          disp_str("#");
24          ticks++;
25
26          if (k_reenter != 0) {
27                  disp_str("!");
28                  return;
29          }
30
31          p_proc_ready++;
```

```
32          if (p_proc_ready >= proc_table + NR_TASKS) {
33                  p_proc_ready = proc_table;
34          }
35  }
```

然后修改sys_get_ticks()，如代码6.70所示。

代码 6.70 sys_get_ticks（chapter6/m/kernel/proc.c）

```
20  PUBLIC int sys_get_ticks()
21  {
22          return ticks;
23  }
```

最后修改TestA()。我们不再打印递增的i了，改成打印当前的 ticks（代码6.71）。

代码 6.71 TestA（chapter6/m/kernel/main.c）

```
80  void TestA()
81  {
82          int i = 0;
83          while (1) {
84                  disp_str("A");
⇒ 85                disp_int(get_ticks());
86                  disp_str(".");
87                  delay(1);
88          }
89  }
```

顺便提一下，在 Linux 中有一个变量jiffies类似于我们的ticks；而在 Minix 中，显然作者更喜欢ticks这个词，我们选择了ticks，因为我们离 Minix 更近。

好了，下面我们可以试一下了，结果如图6.23所示。

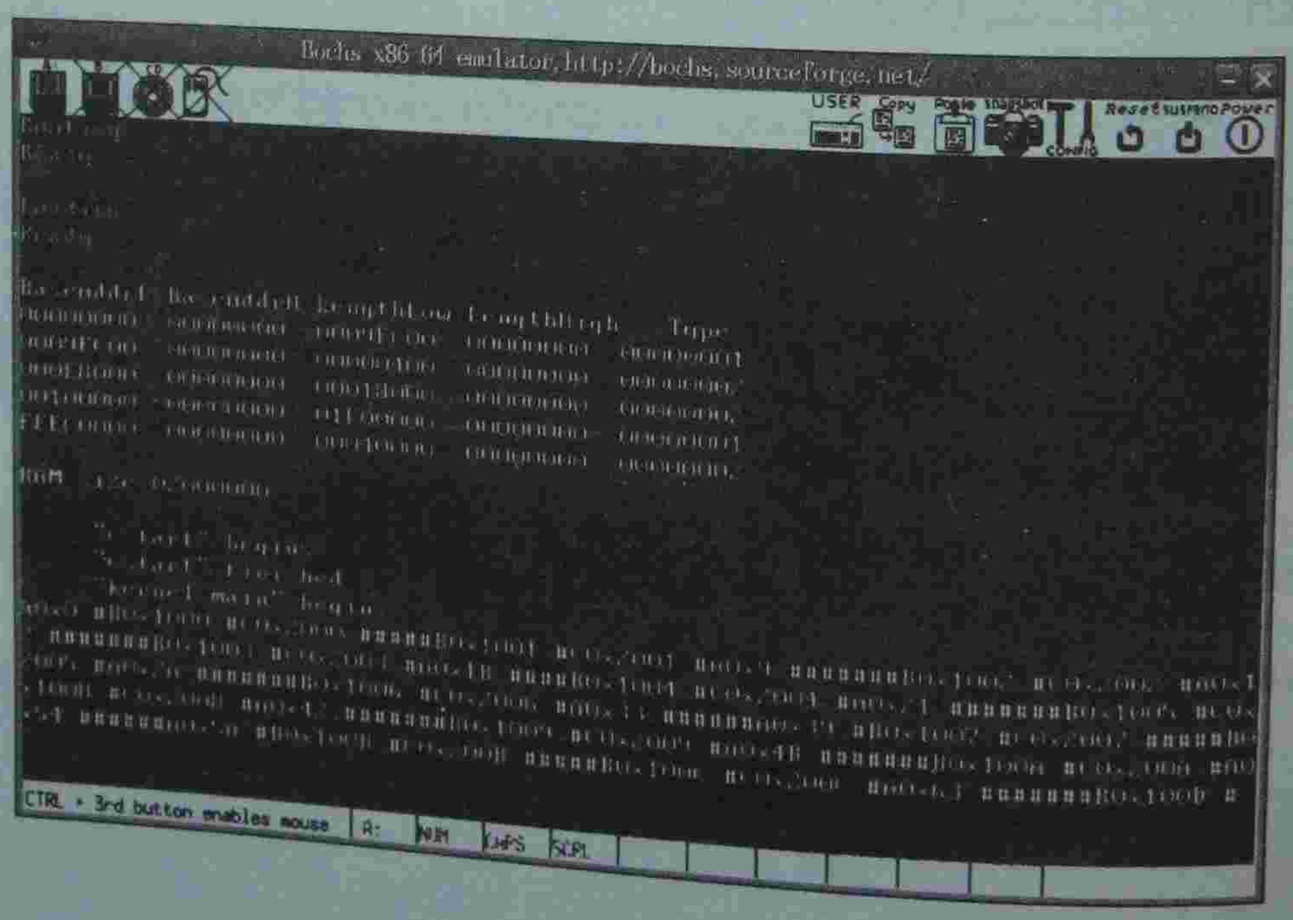


图 6.23 第一个系统调用运行良好

第一次打印出的是A0x0，第二次打印出的是A0x9，如果你注意数一下，会发现两次打印之间的“#”恰好是9个，我们的get_ticks一切正常！

说明：我们已经成功增加了一个系统调用，但要注意，虽然我们在学习Minix；但Minix的系统调用并不是这样，它实际上只有3个系统调用：send、receive和sendrec，并以此为基础建立了一套消息机制，需要系统支持的功能都是通过这套消息机制来做到的，所以，很显然它是微内核的。在这里，我们把get_ticks这样一个普通的功能直接用系统调用来实现，看上去与Minix不同，好像有点像Linux的宏内核的样子。在此笔者要说明一点，之所以这么做，完全是因为在目前的基础上如此实现一个get_ticks函数最为简单，而且又能说明系统调用的原理。它看上去有点像是要实现一个宏内核，这既不代表笔者在微内核、宏内核这个问题上的立场，也不代表Orange'S将来会试图发展成宏内核。

6.5.2 get_ticks的应用

我们当初写get_ticks的时候并没考虑它有什么实际用处，只是觉得它足够简单（函数sys_get_ticks只用一行就搞定了）。可是在写的时候，你有没有想一个问题，时钟中断发生的时间间隔是一定的，如果我们知道了这个时间间隔，就可以用get_ticks函数来写一个判断时间的函数，进而替代我们曾经使用的丑陋的delay()。

那么时钟中断隔多长时间发生一次呢？我们下面就来看一下。

6.5.2.1 8253/8254 PIT

我们一直在讲时钟中断，但好像到目前为止，我们还没考虑过它为什么发生，以及由谁来产生。

中断当然不是凭空产生的。实际上它是由一个被称做PIT（Programmable Interval Timer）的芯片来触发的。在IBM XT中，这个芯片用的是Intel 8253，在AT以及以后换成了Intel 8254。8254功能更强一些，但对于增强的功能，我们并不一定涉及，在下面的陈述中，我们只称呼它8253。

8253有3个计数器（Counter），它们都是16位的，各有不同的作用，如表6.1所示。

表6.1 8253的计数器

计数器	作用
Counter0	输出到IRQ0，以便每隔一段时间让系统产生一次时钟中断
Counter1	通常被设为18，以便大约每15μs做一次RAM刷新
Counter2	连接PC喇叭

从表6.1中看到，时钟中断实际上是由8253的Counter 0产生的。

计数器的工作原理是这样的：它有一个输入频率，在PC上是1193180Hz。在每一个时钟周期（CLK cycle），计数器值会减1，当减到0时，就会触发一个输出。由于计数器是16位的，所以最大值是65535，因此，默认的时钟中断的发生频率就是1193180/65536≈18.2Hz。

我们可以通过编程来控制 8253。因为如果改变计数器的计数值，那么中断产生的时间间隔也就相应改变了。

比如，如果想让系统每 10ms 产生一次中断，也就是让输出频率为 100Hz，那么需要为计数器赋值为 1193180/100≈11931。

已经明白了原理，那么怎么来改变计数器的计数值呢？是通过对相应端口的写操作来实现的。我们来看一下 8253 的端口情况，如表6.2所示。

表 6.2 8253 端口

端口	描述
40h	8253 Counter0
41h	8253 Counter1
42h	8253 Counter2
43h	8253 模式控制寄存器（Mode Control Register）

从表6.2我们知道，改变 Counter 0 计数值需要操作端口 40h。但是这个操作稍微有一点复杂，因为我们需要先通过端口 43h 写 8253 模式控制寄存器。先来看一下它的数据格式，如图6.24所示。

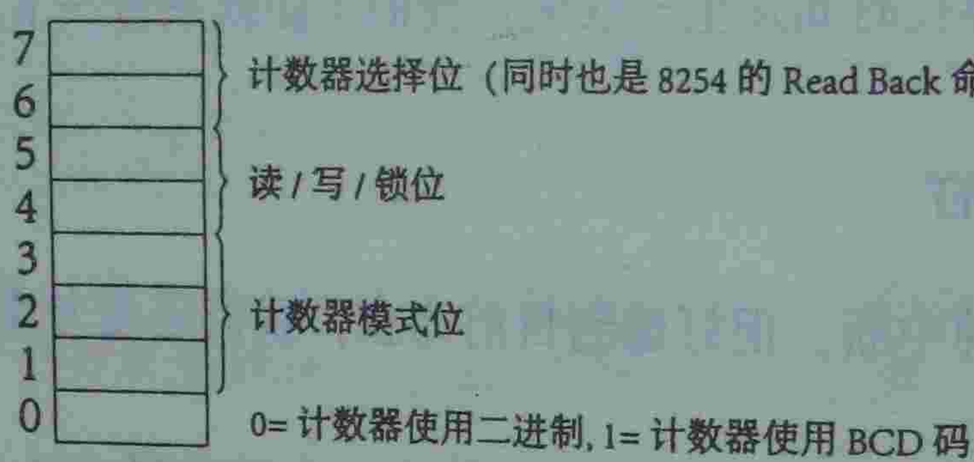


图 6.24 8253 模式控制寄存器

计数器模式位如表6.3所示。

表 6.3 计数器模式位

模式位值			模式	名称
3	2	1		
0	0	0	模式 0	interrupt on terminal count
0	0	1	模式 1	programmable one-shot
0	1	0	模式 2	rate generator ← 我们的时钟中断采用此模式
0	1	1	模式 3	square wave rate generator
1	0	0	模式 4	software triggered strobe
1	0	1	模式 5	hardware triggered strobe

注意：模式的选择不是唯一的，Minix 和 Linux 就分别使用不同的模式。

读/写/锁（Read/Write/Latch）位如表6.4所示。

表6.4 读/写/锁（Read/Write/Latch）位

位		描述
5	4	
0	0	锁住当前记数值（以便于读取）
0	1	只读写高字节
1	0	只读写低字节
1	1	先读写低字节，再读写高字节

注意：锁住（Latch）当前计数器值并不是让计数停止，而仅仅是为了便于读取。相反，如果不锁住直接读取会影响计数。

计数器选择位如表6.5所示。

表6.5 计数器选择位

位		描述
7	6	
0	0	选择 Counter0
0	1	选择 Counter1
1	0	选择 Counter2
1	1	对 8253 而言非法，对 8254 是 Read Back 命令

了解了各部分的意义之后，如何写模式控制寄存器就很明确了。我们要操作的是Counter0，所以第7、6位应该是“00”；计数值是16位的，所以低字节和高字节都要写入，于是第5、4位应该是“11”；使用模式2，所以第3、2、1位应该是“010”；第0位设为“0”。这样，整个字节就变成“00110100”，也就是十六进制的0x34。

下面来设置计数值（代码6.72）。

代码6.72 设置8253（chapter6/n/kernel/main.c）

```
20  PUBLIC int kernel_main()
21  {
    ...
69          /* 初始化 8253 PIT */
70          out_byte(TIMER_MODE, RATE_GENERATOR);
71          out_byte(TIMER0, (u8) (TIMER_FREQ/HZ) );
72          out_byte(TIMER0, (u8) ((TIMER_FREQ/HZ) >> 8));
    ...
80  }
```

其中各个宏的定义如代码6.73所示。

代码 6.73 有关 8253 的宏定义（chapter6/n/include/const.h）

```
42 /* 8253/8254 PIT (Programmable Interval Timer) */
43 #define TIMER0          0x40 /* I/O port for timer channel 0 */
44 #define TIMER_MODE      0x43 /* I/O port for timer mode control */
45 #define RATE_GENERATOR 0x34 /* 00-11-010-0 :
46                              * Counter0 - LSB then MSB - rate generator - binary
47                              */
48 #define TIMER_FREQ      1193182L/* clock frequency for timer in PC and AT */
49 #define HZ              100  /* clock freq (software settable on IBM-PC) */
```

6.5.2.2 不太精确的延迟函数

通过代码6.72，我们已经把两次时钟中断的间隔改成了 10ms，如果现在运行程序，你会看到如图6.25所示的情形：在很短的时间内打印出很多“#”，这说明中断发生快了很多。也难怪，原来一秒钟 18.2 次中断，大约 55ms 发生一次，现在一秒钟 100 次，10ms 发生一次，所以区别才会这么明显。

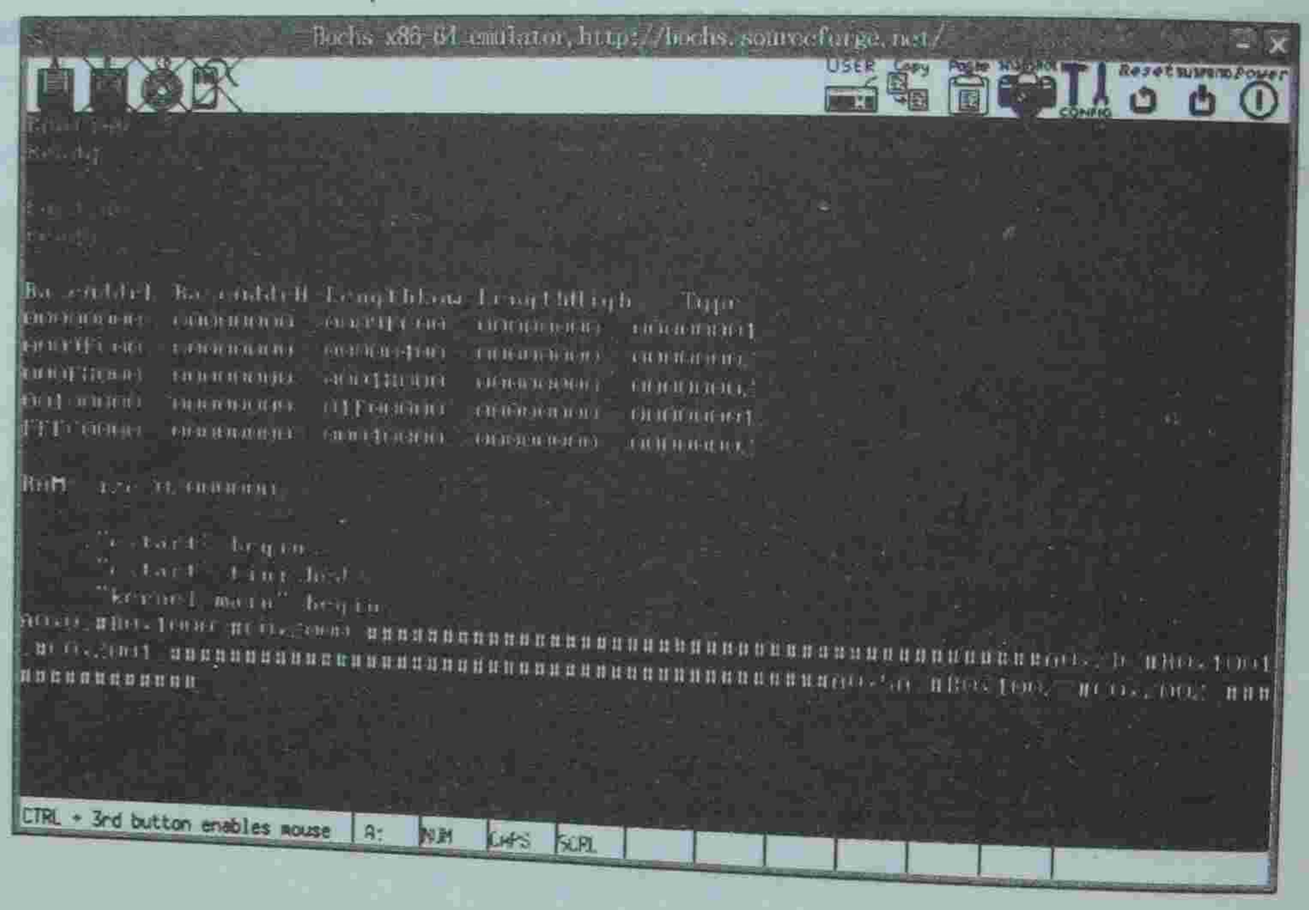


图 6.25 重新设置 8253 之后

现在我们可以编写新的延迟函数了，因为中断 10ms 发生一次，所以ticks也是 10ms 增加一次，延迟函数可以这样来写，见代码6.74。

代码 6.74 精确到 10ms 的延迟函数（chapter6/n/kernel/clock.c）

```
40 PUBLIC void milli_delay(int milli_sec)
41 {
42         int t = get_ticks();
43
44         while(((get_ticks() - t) * 1000 / HZ) < milli_sec) {}
45 }
```

函数一开始得到当前的ticks值，然后开始循环，每次循环的时候看已经过去了多少ticks（假设是 Δt 个）。因为ticks之间的间隔时间是 (1000/Hz)ms，所以 Δt 个ticks相当于 (Δt*1000/Hz)ms，循环会在这个毫秒数大于要求的毫秒数时退出。

接下来我们修改进程 A 的进程体（代码6.75）。

代码 6.75 修改进程 A（chapter6/n/kernel/main.c）

```
85  void TestA()
86  {
87      int i = 0;
88      while (1) {
89          disp_str("A");
90          disp_int(get_ticks());
91          disp_str(".");
92          milli_delay(1000);
93      }
94  }
```

同时让进程 B 和 C 的进程体与此相似，只是打印的字母不同。然后make，运行，呈现图6.26所示的结果。

图 6.26 使用 milli_delay 的进程

从图6.26中看出，发生了很多次中断重入。由于现在进入内核态要么是发生了时钟中断，要么是调用了get_ticks，所以重入发生的唯一情况是调用get_ticks时发生了时钟中断。不过这倒不是什么大问题，我们已经有了相应的处理机制。

读者可以根据打印出的ticks值来计算两次打印“A”之间发生了多少此中断，在图6.26中，第 3 次和第 2 次打印 A 之间发生了 0x64（0xC8-0x64），也就是100次中断，这是很完美的。但是如果读者亲自实验并实验多次的话，可能发现并不是每次运行都完美，误差有时的确是存在的。根据笔者的试验，最后发现误差有时候很小，比如 0ms 或者在 10ms 以内，有时候则比较大，比如 70ms。也就是说，虽然中断 10ms 发生一次，但通过这种方式写出来的milli_delay误差却不止 10ms，而是“10ms 级”的。

究其原因，一个很重要的方面在于，现在有不止一个进程在运行，当时间满足条件之后，CPU 控制权可能恰好交给了其他进程，这时其他进程可能耗费掉若干的 ticks。另外，打印这些字符和数字也用掉一些 ticks。

为了排除其他因素的影响，我们把进程数减为 1（可以通过修改NR_TASKS和task_table[NR_TASKS]来实现），然后把中断例程中打印“#”和“!”的代码也去掉，再运行一次，会发现每一次的间隔都是 0x64，也就是 100 个 ticks

(如图6.27所示)。

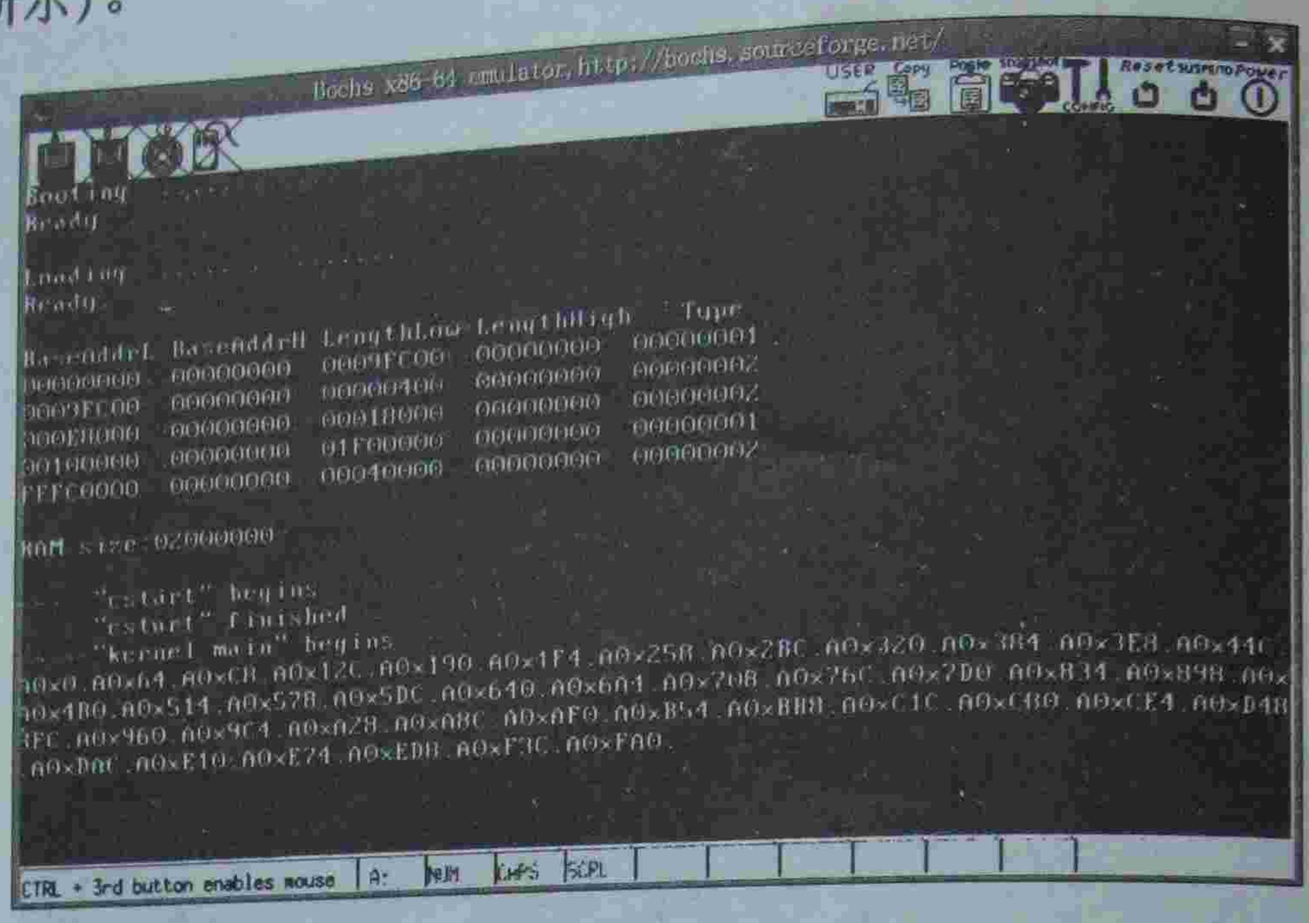


图 6.27 只有一个进程运行时 milli_delay 的执行情况

虽然存在误差的可能，虽然精度不够高，但比起原来那个野蛮的循环，却已经有很大进步了。而且我们把第一个系统调用派上了用场，同时还掌握了如何操作 8253。这些收获无疑让这个不完美的函数价值大增。

在不同的甚至是同一个操作系统中，都会有不同的延迟函数，这些函数实现的机制各不相同，而且有一些还很精妙。比如 Linux 中的udelay，是通过计算循环次数和时间之间的关系，用一定次数的循环来延迟一定的时间（我们当初的delay也是循环，不过太简陋了）；Minix 中的milli_delay通过读取 8253 的计数值来得到比较精确的延迟时间，但是它只能运行在核心态。

我们的延迟函数是新的发明，它不够精妙，但是足够简单，milli_delay的函数体只有两行这是其他延迟函数做不到的。而且它运行在用户态，使用十分方便。

6.6 进程调度

6.6.1 避免对称——进程的节奏感

上面的 3 个进程都是延迟相同的时间，让我们修改一下，尝试让它们延迟不同的时间（代码6.76）。

代码 6.76 修改 3 个进程的延迟时间（chapter6/o/kernel/main.c）

```
82  /*======================================================================*
83                                 TestA
84   *======================================================================*/
85  void TestA()
86  {
87          int i = 0;
88          while (1) {
89                  disp_str("A.");
90                  milli_delay(300);
```

```
        }
}

/*======================================================================*
                               TestB
 *======================================================================*/
void TestB()
{
        int i = 0x1000;
        while(1){
                disp_str("B.");
                milli_delay(900);
        }
}

/*======================================================================*
                               TestB
 *======================================================================*/
void TestC()
{
        int i = 0x2000;
        while(1){
                disp_str("C.");
                milli_delay(1500);
        }
}
```

修改后的代码让A、B、C三个进程分别延迟300、900、1500ms。容易猜到，这样一来，打印出来的3个字母的数量肯定应该有较大的差别了，进程执行时间上的对称从此被打破。我们来看看输出的情况，如图6.28所示。

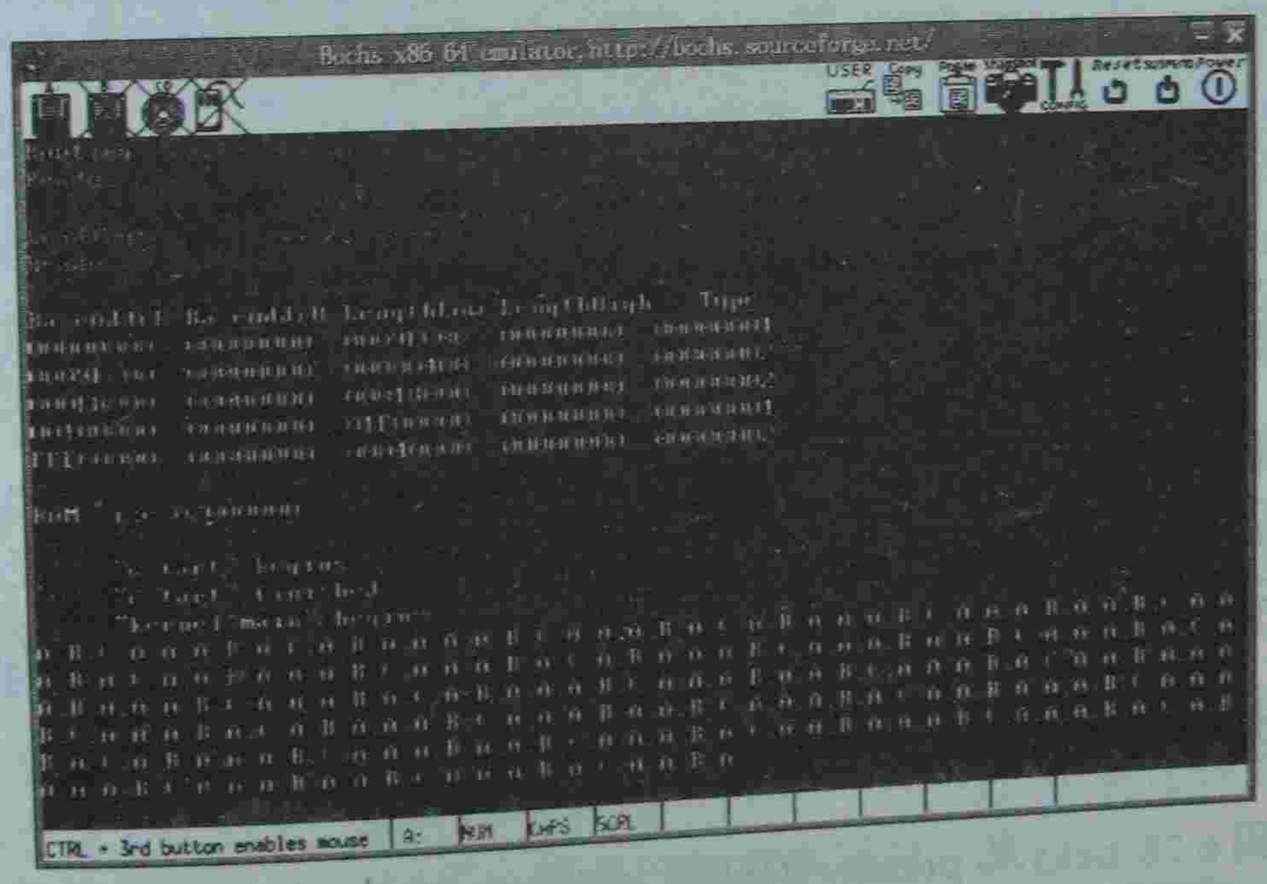


图 6.28 让不同的进程休息不同的时间

数一数[2]可以知道，输出中共有A字母140个，B字母51个，C字母32个，所以A和B的个数之比是2.745，A和C的个数之比是4.345，这两个数字与3（进程B和A的延迟时间之比）和5（进程C和A的延迟时间之比）是基本吻合的。

[2]用眼睛数不是个好主意，可以用Bochs的snapshot功能将屏幕抓成一个文本文件，然后用一个命令来计算字母个数，比如这样：`cat snapshot.txt | sed -e 's/\([ABC]\)\./\1\n/g' | sort | grep A | wc -l`。

这样的输出应该是在预料之中的，延迟的时间越长，干的活当然就越少，于是输出就会越少。

不过，这个输出倒是给了我们一个启示，进程干活的时间长短不一，这不正好和“优先级”的概念暗合了吗？

是的，在上面的例子中，完全可以认为我们通过设置延迟时间的长短而为不同的进程赋予了不同的优先级，当然这种说法不确切，这种方法也不值得提倡。但我们可以进一步思考，既然延迟是通过得到 ticks 来实现的，如果我们把延迟的过程拿到进程调度模块中来实现，不就真的实现了进程的优先级了吗？因为 ticks 在哪里都是可以得到的。

实现的方法并不困难，我们只需要稍微修改调度算法就可以了。过去，我们在发生时钟中断选择下一个执行的进程时，直接选择进程表中的下一个进程，这种时间片轮转的方式给了每个进程均等的机会。我们现在不再给每个进程以相等的机会了。具体的方法是，给每一个进程都添加一个变量（可以放在进程表中），在一段时间的开头，这个变量的值有大有小，进程每获得一个运行周期，这个变量就减 1，当减到 0 时，此进程就不再获得执行的机会，直到所有进程的变量都减到 0 为止。这样，每个进程获得的执行时间就不一样了。我们现在修改一下代码，看看执行的结果怎么样。先修改进程表（代码6.77）。

代码 6.77 为进程表添加新的成员（chapter6/p/include/proc.h）

```
31  typedef struct s_proc {
32          STACK_FRAME regs;          /* process registers saved in stack frame */
33
34          u16 ldt_sel;               /* gdt selector giving ldt base and limit */
35          DESCRIPTOR ldts[LDT_SIZE]; /* local descriptors for code and data */
36
⇒ 37        int ticks;                 /* remained ticks */
⇒ 38        int priority;
39
40          u32 pid;                   /* process id passed in from MM */
41          char p_name[16];           /* name of the process */
42  }PROCESS;
```

在进程表中添加了两个成员：ticks是递减的，从某个初值到 0。为了记住ticks的初值，我们另外定义一个变量priority，它是恒定不变的。当所有的进程ticks都变为 0 之后，再把各自的ticks赋值为priority，然后继续执行。

ticks和priority最初赋值如代码6.78所示。

代码 6.78 ticks 和 priority 的初值（chapter6/p/kernel/main.c）

```
20  PUBLIC int kernel_main()
21  {
    ...
64
65          proc_table[0].ticks = proc_table[0].priority = 150;
66          proc_table[1].ticks = proc_table[1].priority =  50;
            proc_table[2].ticks = proc_table[2].priority =  30;
    ...
84  }
```

对于进程调度，我们可以单独写一个函数，叫做schedule()，放在proc.c

代码 6.79 进程调度（chapter6/p/kernel/proc.c）

```
19  PUBLIC void schedule()
20  {
21          PROCESS* p;
22          int      greatest_ticks = 0;
23
24          while (!greatest_ticks) {
25                  for (p = proc_table; p < proc_table+NR_TASKS; p++) {
26                          if (p->ticks > greatest_ticks) {
27                                  greatest_ticks = p->ticks;
28                                  p_proc_ready = p;
29                          }
30                  }
31
32                  if (!greatest_ticks) {
33                          for (p = proc_table; p < proc_table+NR_TASKS; p++) {
34                                  p->ticks = p->priority;
35                          }
36                  }
37          }
38  }
```

同时修改时钟中断处理函数为代码6.80的样子。

代码 6.80 修改后的时钟中断（chapter6/p/kernel/clock.c）

```
20  PUBLIC void clock_handler(int irq)
21  {
22          ticks++;
23          p_proc_ready->ticks--;
24
25          if (k_reenter != 0) {
26                  return;
27          }
28
29          schedule();
30  }
```

同时，我们将所有进程的延迟时间全改为相同的值，把所有milli_delay的参数改成 200。

make，运行，出现如图6.29所示的情形。

从图6.29中我们看到，虽然各个进程延迟的时间都相同，但由于改变了它们的优先级，运行的时间明显不同，这说明我们的优先级策略生效了！

不过细心一点可以发现，当前的 A、B、C三个字母的个数之比是 139:71:54，大体相当于 2.57:1.31:1，与进程优先级 5:1.67:1（15:5:3）相差比较大。为什么会出现这样的情形呢？我们打印出更多的信息来研究一下。

首先修改各个进程，让它们各自打印一个当前的ticks（代码6.81）。

代码 6.81 修改后的进程（chapter6/q/kernel/main.c）

```
92  /*======================================================================*
93                               TestA
94   *======================================================================*/
95  void TestA()
96  {
97          int i = 0;
98          while (1) {
99                  disp_color_str("A.", BRIGHT | MAKE_COLOR(BLACK, RED));
100                 disp_int(get_ticks());
```

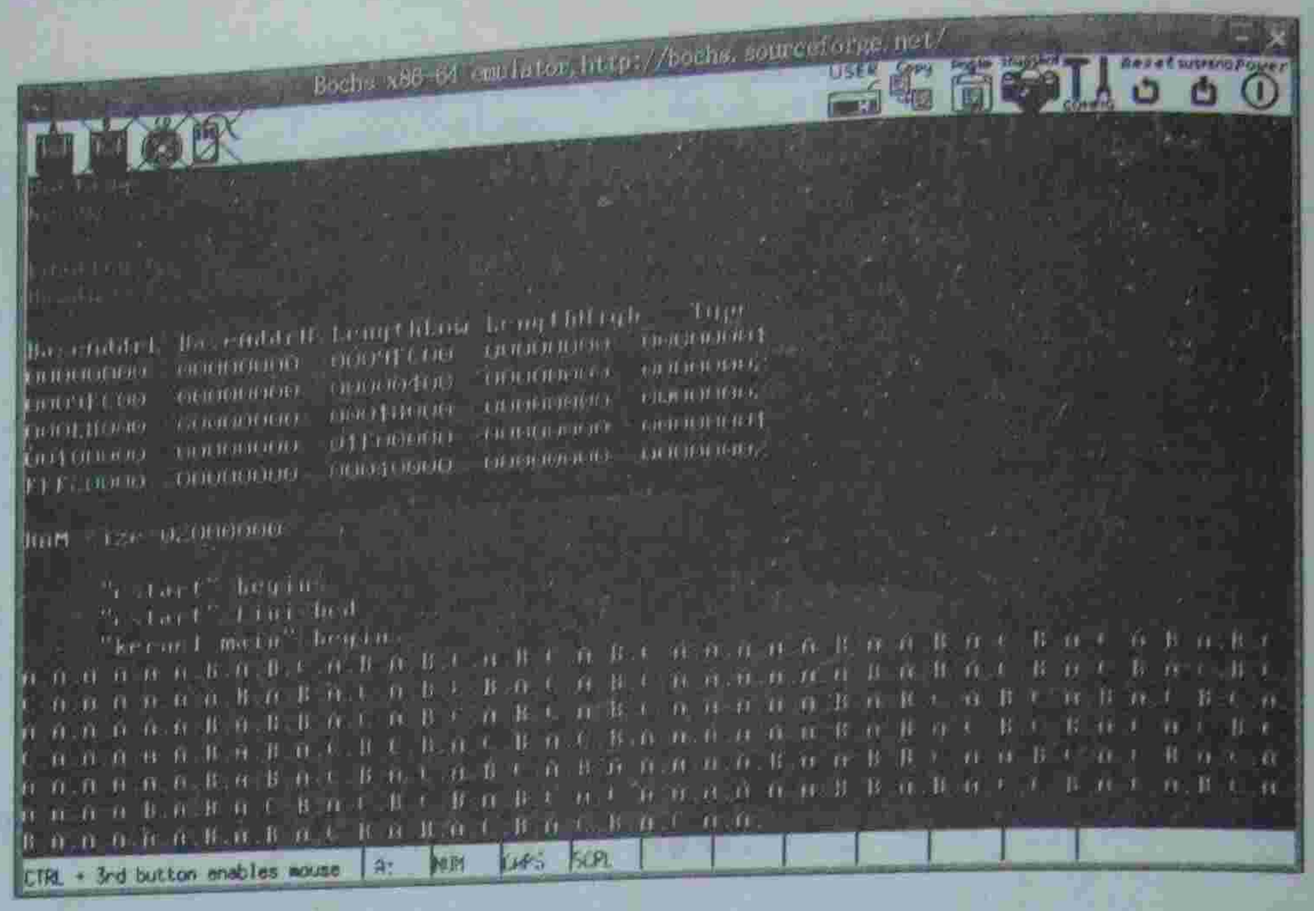


图 6.29 基于优先级的进程调度

```
101                 milli_delay(200);
102         }
103 }
104
105 /*======================================================================*
106                                TestB
107  *======================================================================*/
108 void TestB()
109 {
110     int i = 0x1000;
111     while(1){
⇒ 112         disp_color_str("B.", BRIGHT | MAKE_COLOR(BLACK, RED));
⇒ 113         disp_int(get_ticks());
114         milli_delay(200);
115     }
116 }
117
118 /*======================================================================*
119
120                                TestB
121  *======================================================================*/
122 void TestC()
123 {
124     int i = 0x2000;
125     while(1){
⇒ 126         disp_color_str("C.", BRIGHT | MAKE_COLOR(BLACK, RED));
127         disp_int(get_ticks());
128         milli_delay(200);
129     }
```

为了让A、B、C三个字母看上去醒目一些，这里用disp_color_str函数把它们打印成红色。

然后修改一下proc.c中的schedule()，加上几条打印语句，同时注释掉为进程表中的成员ticks重新赋值的代码，以便让进程不至于永远执行下去（当所有进程的ticks都减为0时程序就停止了），这样比较有利于观察（代码6.82）。

代码 6.82 修改后的 schedule（chapter6/q/kernel/proc.c）

```
19 PUBLIC void schedule()
```

```
{
        PROCESS* p;
        int      greatest_ticks = 0;

        while (!greatest_ticks) {
                for (p = proc_table; p < proc_table+NR_TASKS; p++) {
                        if (p->ticks > greatest_ticks) {
                                disp_str("<");
                                disp_int(p->ticks);
                                disp_str(">");
                                greatest_ticks = p->ticks;
                                p_proc_ready = p;
                        }
                }

                /* if (!greatest_ticks) { */
                /*      for (p = proc_table; p < proc_table+NR_TASKS; p++) { */
                /*              p->ticks = p->priority; */
                /*      } */
                /* } */
        }
}
```

由于现在打印的东西比较多了，我们在kernel_main()中添加清空屏幕的函数，以便让输出从屏幕左上角开始，否则我们无法看到所有的输出（代码6.83）。

代码 6.83 修改后的 kernel_main（chapter6/q/kernel/main.c）

```
PUBLIC int kernel_main()
{
 ...
        disp_pos = 0;
        for (i = 0; i < 80*25; i++) {
                disp_str("␣");
        }
        disp_pos = 0;
 ...
}
```

再次运行，结果如图6.30所示。

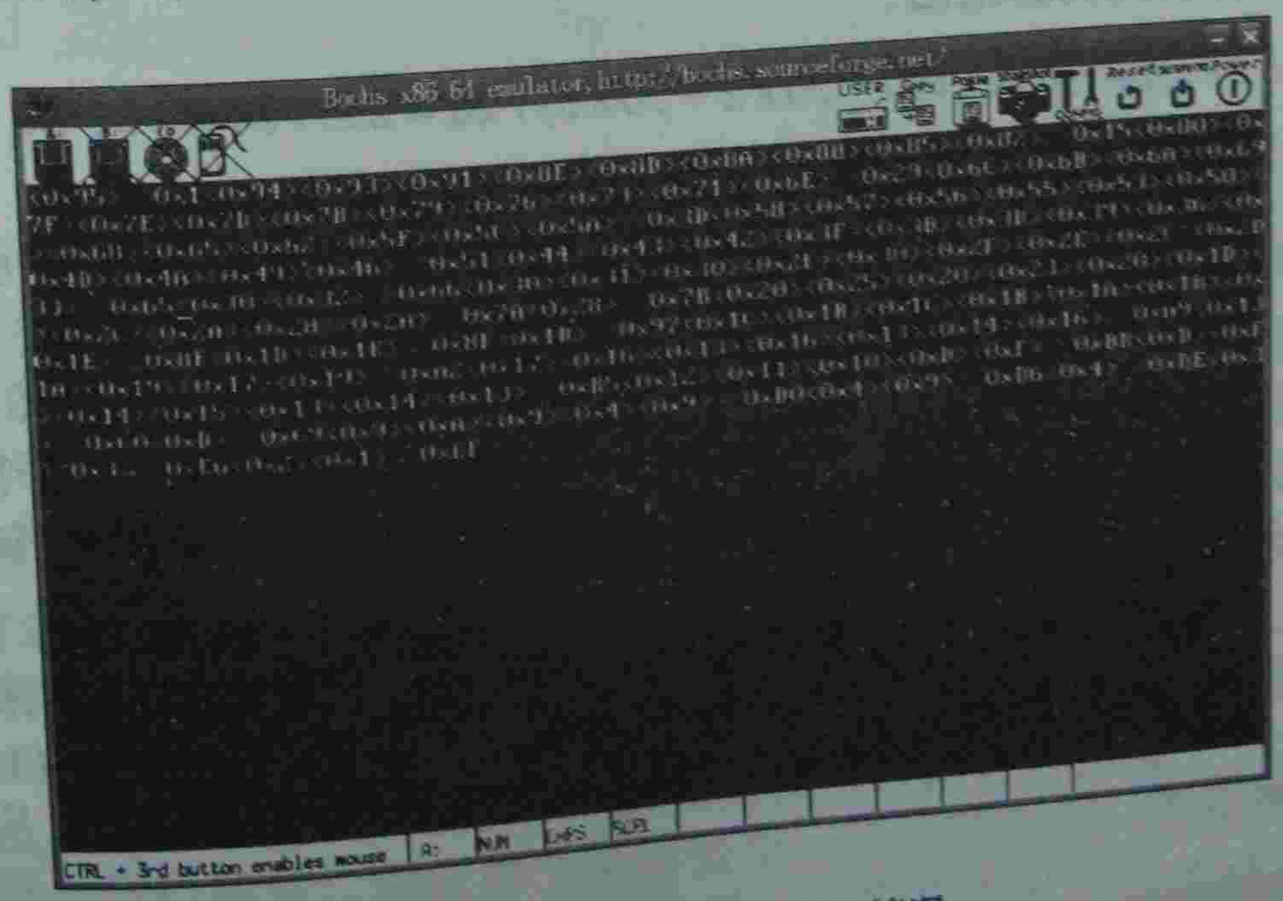


图 6.30 打印出的进程调度情况

稍加分析会发现，整个执行过程可以划分成3个阶段：最开始只有进程A自己在运行，后来A和B同时运行，再后来A、B、C同时运行，如表6.6所示。

表6.6 对结果的分析

进程	每次循环耗费	阶段一	阶段二	阶段三	总共	原因
A	20 ticks	5	2	4	5+2+4(11)	230/20 = 11.5
B	20 ticks	由于优先级低而未被调度	2	5	2+5(7)	130/20 = 6.5
C	20 ticks	由于优先级低而未被调度	由于优先级低而未被调度	5	5(5)	90/20 = 4.5

在表6.6中，除了最右边一列“结果的产生原因”外，都是图6.30所示执行结果的真实记录。由于进程的每一次循环都延迟200ms（即20ticks），所以，在最开始的100ticks中，进程A循环5次，在后面的20×2ticks中每个进程循环2次，最后的30×3ticks中每个进程循环4次或5次都很容易理解。

其实现在已经很明白了，在3个阶段中，最初阶段的时间跨度为100ticks，之后，由于进程A的ticks值已经小于50，已经与进程B的ticks值相当，所以以后就同时有A和B受到调度。在最后一个阶段，就变成A、B、C三个进程同时受到调度。

由于每一次进程调度的时候只有某一个进程的ticks会减1（而不是3个进程ticks同时减1），所以，总共调度的次数应该是3个进程的ticks之和（150+50+30=230）。这个规律放在中间某个过程中也是适用的，比如到最后阶段，当A和B的ticks都减到30时，3个进程同时运行，总共运行的时间将是30×3=90ticks。所以我们总结出：

进程A执行的循环数 = (100 + 20×2 + 30×3) / 20 = 230 / 20 = 11.5次

进程B执行的循环数 = (0 + 20×2 + 30×3) / 20 = 130 / 20 = 6.5次

进程C执行的循环数 = (0 + 0×2 + 30×3) / 20 = 90 / 20 = 4.5次

这个结论与我们的试验结果11、7、5是相吻合的。

根据这个分析也可以知道，基于现在的调度算法，A、B、C三个进程的执行时间之比，理论值应该是230:130:90，即2.56:1.44:1。我们两次的试验结果（图6.29和图6.30）结论都与此相吻合。

现在，从实践到理论，我们第一阶段的调度算法试验就算是结束了。可以看到，虽然这种算法能分出定性的优先级关系，但是从数字上（150、50、30）不容易一下子看出各自执行的时间定量关系（150:50:30和11:7:5是很不相同的）。这就意味着，当我们给予一个进程某个优先级，需要经过计算才能知道它们各自运行的时间比例是多少。这让我们感到，有必要在此基础上改进一下程序。其实，只要在clock_handler()中添加一个判断，问题便告解决（代

码6.84）。

代码 6.84 修改后的 clock_handler（chapter6/r/kernel/clock.c）

```
20  PUBLIC void clock_handler(int irq)
21  {
22          ticks++;
23          p_proc_ready->ticks--;
24
25          if (k_reenter != 0) {
26                  return;
27          }
28
⇨ 29          if (p_proc_ready->ticks > 0) {
⇨ 30                  return;
⇨ 31          }
32
33          schedule();
34
35  }
```

这样，在一个进程的 ticks 还没有变成 0 之前，其他进程就不会有机会获得执行，结果如图6.31所示。

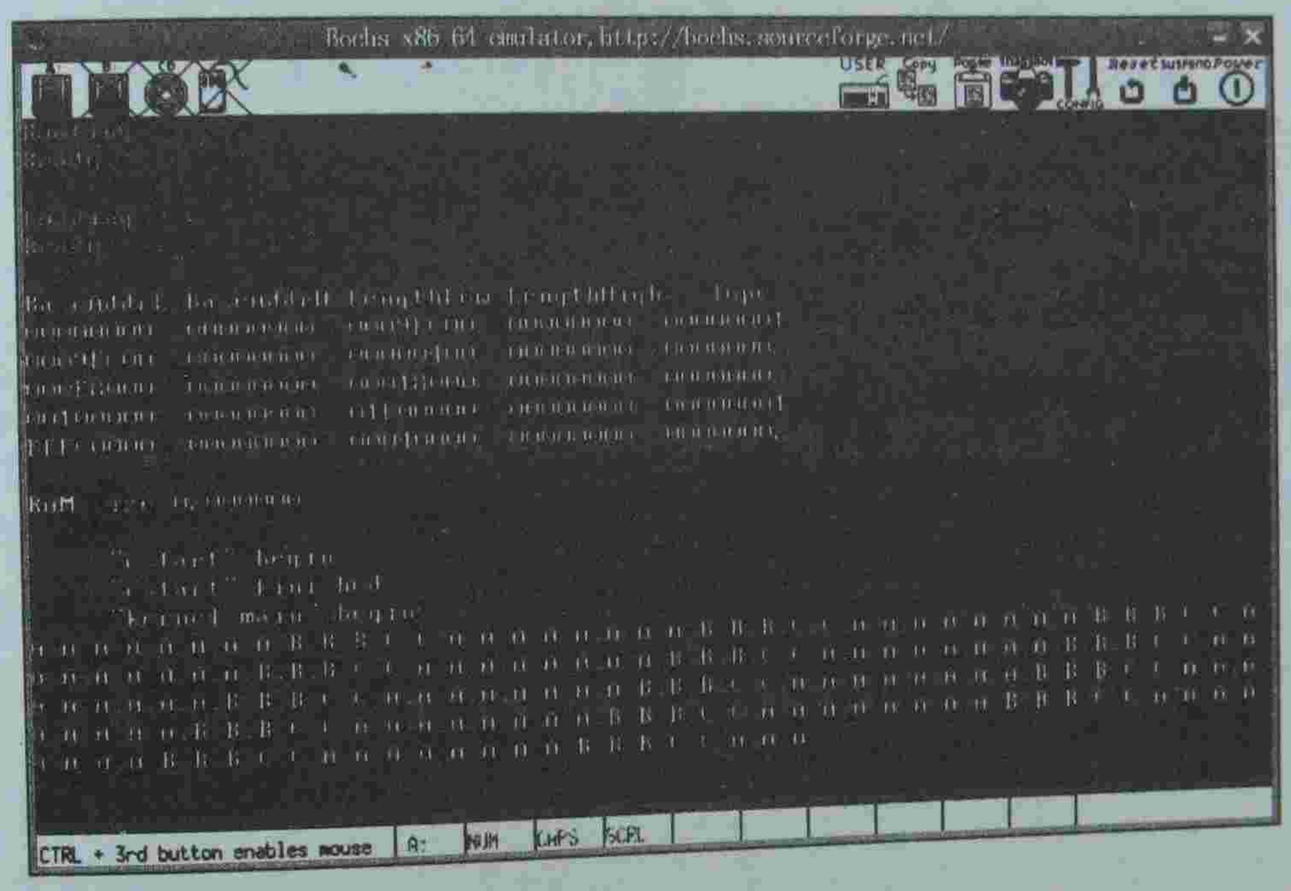


图 6.31 进程调度策略改进后的调度情况

从6.31图可以明显看出，进程 A 先执行，然后是 B，再然后是 C，与原先有了很大的差别。原因在于进程 A 的 ticks 从 150 递减至 0 之后，才把控制权给 B，B 用完它的 ticks（50）之后再给 C，然后各自的 ticks 被重置，继续下一个类似的过程。可以看到，进程 A 在 150ticks 内执行 8 次循环，B 在 50ticks 内执行 3 次循环，C 在 30ticks 内执行 2 次循环。这样就很直观了。

不过这样看上去进程各自运行的时间有一点长，我们把它们的优先级改小一点（代码6.85）。

代码 6.85 修改后的优先级（chapter6/r/kernel/main.c）

```
20  PUBLIC int kernel_main()
21  {
    ...
64          proc_table[0].ticks = proc_table[0].priority = 15;
65          proc_table[1].ticks = proc_table[1].priority =  5;
```

```
66          proc_table[2].ticks = proc_table[2].priority =  3;
...
84  }
```

然后把各个进程的延迟时间改成 10ms（代码6.86）。

代码 6.86 修改后的延迟时间（chapter6/r/kernel/main.c）

```
86  /*======================================================================*
87                               TestA
88   *======================================================================*/
89  void TestA()
90  {
91          int i = 0;
92          while (1) {
93                  disp_str("A.");
⇒ 94                  milli_delay(10);
95          }
96  }
97
98  /*======================================================================*
99                               TestB
100  *======================================================================*/
101 void TestB()
102 {
103         int i = 0x1000;
104         while(1){
105                 disp_str("B.");
⇒ 106                 milli_delay(10);
107         }
108 }
109
110 /*======================================================================*
111                              TestB
112  *======================================================================*/
113 void TestC()
114 {
115         int i = 0x2000;
116         while(1){
117                 disp_str("C.");
⇒ 118                 milli_delay(10);
119         }
120 }
```

再运行一下，结果如图6.32所示。

从这次结果可以看出，打印出的字符的个数之比非常接近 15:5:3。

6.6.2 优先级调度总结

至此，基于简单优先级的进程调度算法已经完成了。它很简单，目前看来运行得还是可以的。顺便要告诉读者的是，它实际上是从 Linux 借鉴而来，不过并不是为了借鉴而去借鉴，而是笔者在写系统调用get_ticks()以及随后的代码时偶然发现自己的部分代码与 Linux 相关代码有一定联系，然后参考了它的代码，最后形成现在的情形。

计算机世界跟现实世界在很多时候都是类似的，在各个层面上，你都会经常有所体会，或者这样类比。远些说，比如面向对象技术中关于类、实例等的编程思想，近些说，比如本章开头提到的进程和人的工作之间的相似性。

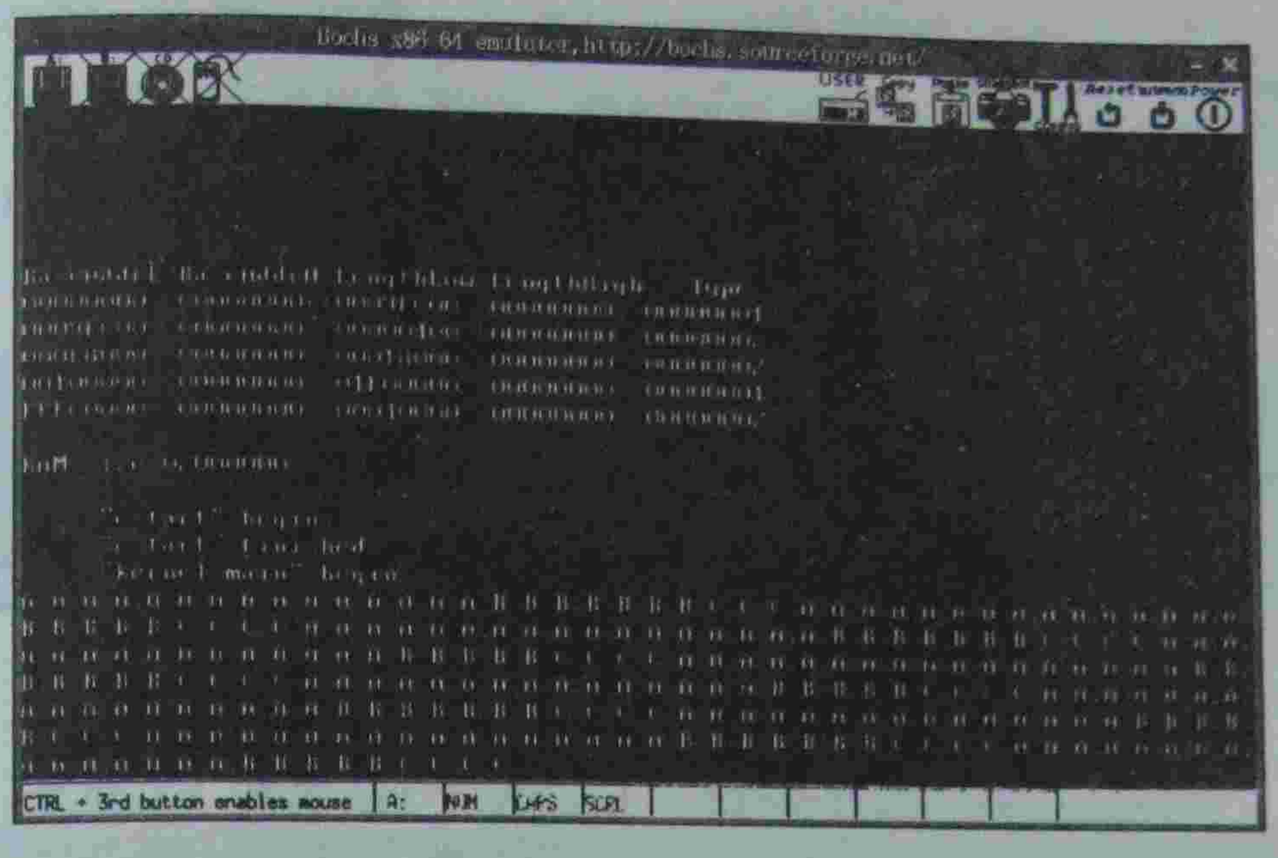


图 6.32 优先级和进程延迟时间改变后的执行情况

优先级调度也是来源于现实世界，所谓“轻重缓急”四个字，恰好可以用来表达优先级调度的思想。最重要的事情总是应该被赋予更高的优先级，应该给予更多的时间，以及尽早地进行处理。

在 Minix 中，进程分为任务（Task）、服务（Server）和用户进程（User）三种，进程调度也据此设置了 3 个不同的优先级队列，我们目前并没有使用优先级队列来实现调度策略，是因为一方面那样会使程序实现的复杂度大大增加，另一方面目前的算法是在系统调用get_ticks()的使用中顺理成章地形成，虽然它很简陋，但在“更早地处理”和“更多的时间”这两方面，都已经给予了高优先级的进程以很大的照顾。而且毫无疑问，我们已经通过它进入了进程调度算法这个领域的大门。它虽远称不上好，但却功莫大焉。

By far the best proof is experience. —— *Francis Bacon*

7 输入/输出系统

完美主义者常常因试图努力把一件事做好而放弃对新领域的尝试，从而使做事的机会成本增加。有时候回头一看才发现，自己在某件事情上已花费了太多时间。而实际上，暂时的妥协可能并不会影响到最终完美结果的呈现。因为不但知识需要沉淀，事情之间也总是有关联的。

我们刚刚实现了简单的进程，你现在可能很想把它做得更加完善，比如进一步改进调度算法、增加通信机制等。但是这些工作不但做起来没有尽头，而且有些也是难以实现的，因为进程必须与I/O、内存管理等其他模块一起工作。而且，简单的进程更有利于我们思考和控制。

那么现在，我们就来实现简单的I/O。

7.1 键盘

到目前为止，我们的操作系统一旦启动就不再受我们的控制了，只能静静地等待结果的出现。但正如眼下人们看新闻的方式正在从电视转向因特网一样，我们的操作系统显然也更需要交互。而交互的手段，首先当然是键盘。

7.1.1 从中断开始——键盘初体验

说起键盘，你可能想起8259A的IRQ1对应的就是键盘，我们在第5章中甚至做过一个小小的试验（见代码5.57和图5.16）。那时我们没有为键盘中断指定专门的处理程序，所以当按下键盘时只能打印一行“spurious_irq:0x1”。现在我们来写一个专门的处理程序。

新建一个文件keyboard.c，添加一个非常简单的键盘中断处理程序（代码7.1）。

代码7.1 键盘中断处理程序（chapter7/a/kernel/keyboard.c）

```
18 PUBLIC void keyboard_handler(int irq)
19 {
20         disp_str("*");
21 }
```

结果是每按一次键，打印一个星号，有点像在输入密码。

为了不受其他进程输出的影响，我们把其他进程的输出都注释掉。

然后添加指定中断处理程序的代码并打开键盘中断（代码7.2）。

代码7.2 打开键盘中断（chapter7/a/kernel/keyboard.c）

```
PUBLIC void init_keyboard()
{
        put_irq_handler(KEYBOARD_IRQ, keyboard_handler);/*设定键盘中断处理程序*/
        enable_irq(KEYBOARD_IRQ);                       /*开键盘中断*/
}
```

不要忘了在proto.h中声明init_keyboard()并调用之（代码7.3）。

代码7.3 调用init_keyboard（chapter7/a/kernel/main.c）

```
PUBLIC int kernel_main()
{
...
        init_keyboard();
...
}
```

修改Makefile之后，就可以make并运行了，如图7.1所示。

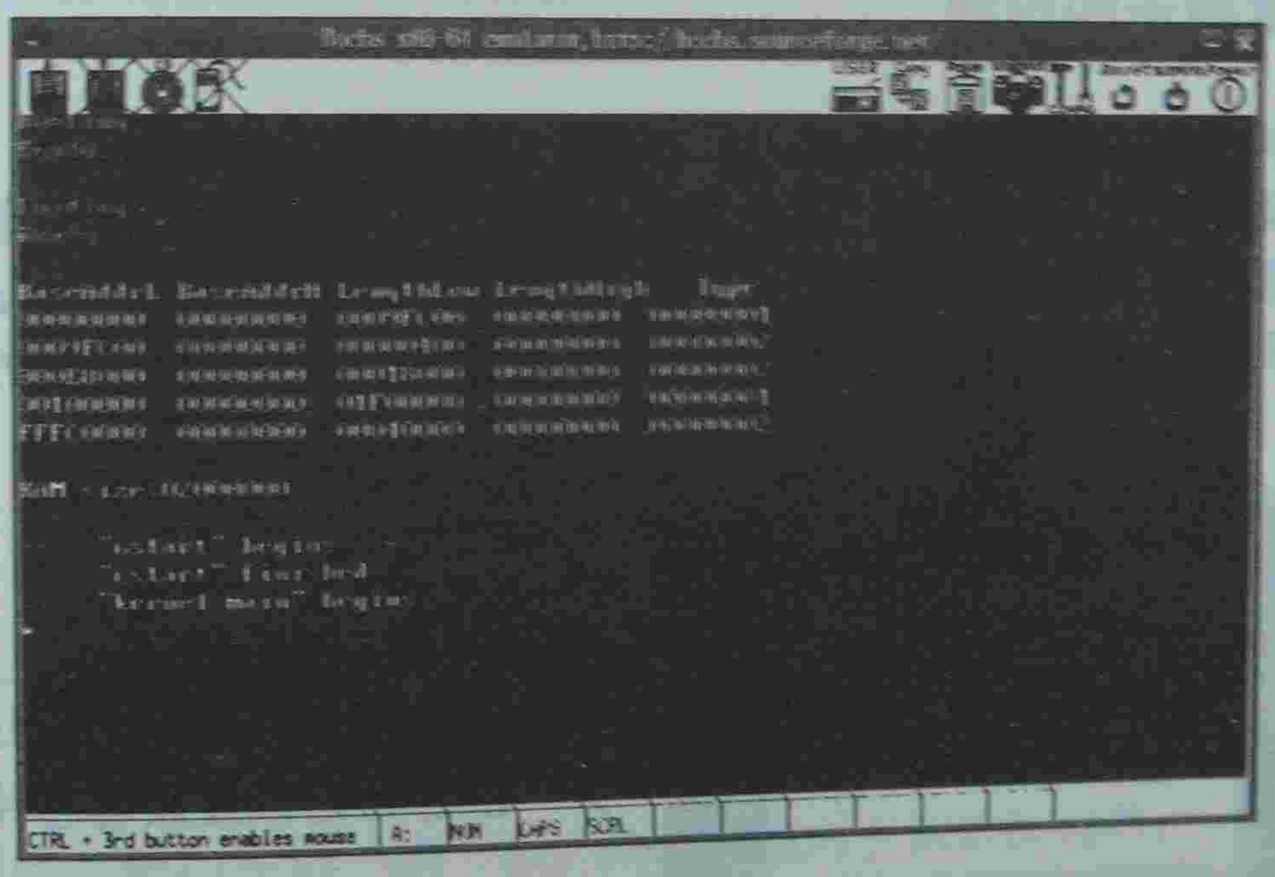


图7.1 在键盘中断处理程序中打印星号

怎么出现一个星号之后键盘就不再响应了？这倒是比较奇怪的，看来事情比我们想像的要复杂一些。我们还是要从头说起。

7.1.2 AT、PS/2键盘

PS/2键盘和USB键盘是当今最流行的两种键盘。现在，看看你的键盘连接计算机的接口，如果看上去很像图7.2左图的样子，那么你用的应该是个PS/2键盘；如果是一个有棱有角的扁口，那么它很可能是个USB键盘（如果你居然不熟悉USB口的话）。再有一种可能，你还在使用一种稍微老一点的AT键盘，它看上去就像图7.2右图的形状。有人把AT键盘称为“大口”键盘，而把PS/2称

为“小口”键盘。因为AT键盘的接口稍微大一些而PS/2的稍微小一些。它们的接口分别叫做“5-pin DIN”和“6-pin Mini-DIN”。

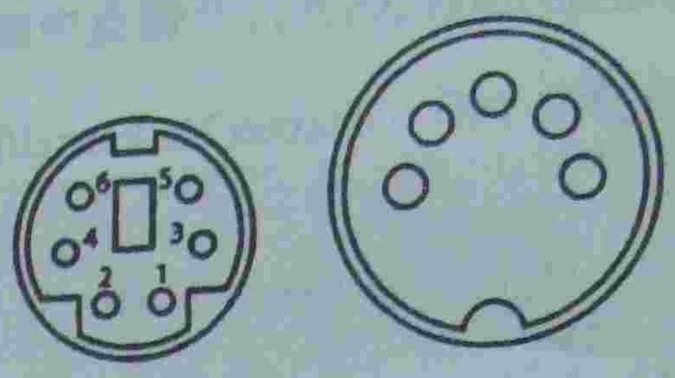


图7.2 两种键盘接口

实际上在更早时候，IBM还推出过XT键盘，它也使用5-pin DIN，不过现在早就不再使用了。如今的主流键盘主要是AT、PS/2、USB三种。在本书中，我们不讨论USB键盘，只介绍AT和PS/2两种。

实际上，PS/2键盘只是在AT键盘的基础上做了一点点的扩展而已，在大多数情况下，尤其是在最初接触它们时，你可以认为它们是一样的。

7.1.3 键盘敲击的过程

在键盘中存在一枚叫做键盘编码器（Keyboard Encoder）的芯片，它通常是Intel 8048以及兼容芯片，作用是监视键盘的输入，并把适当的数据传送给计算机。另外，在计算机主板上还有一个键盘控制器（Keyboard Controller），用来接收和解码来自键盘的数据，并与8259A以及软件等进行通信（如图7.3所示）。

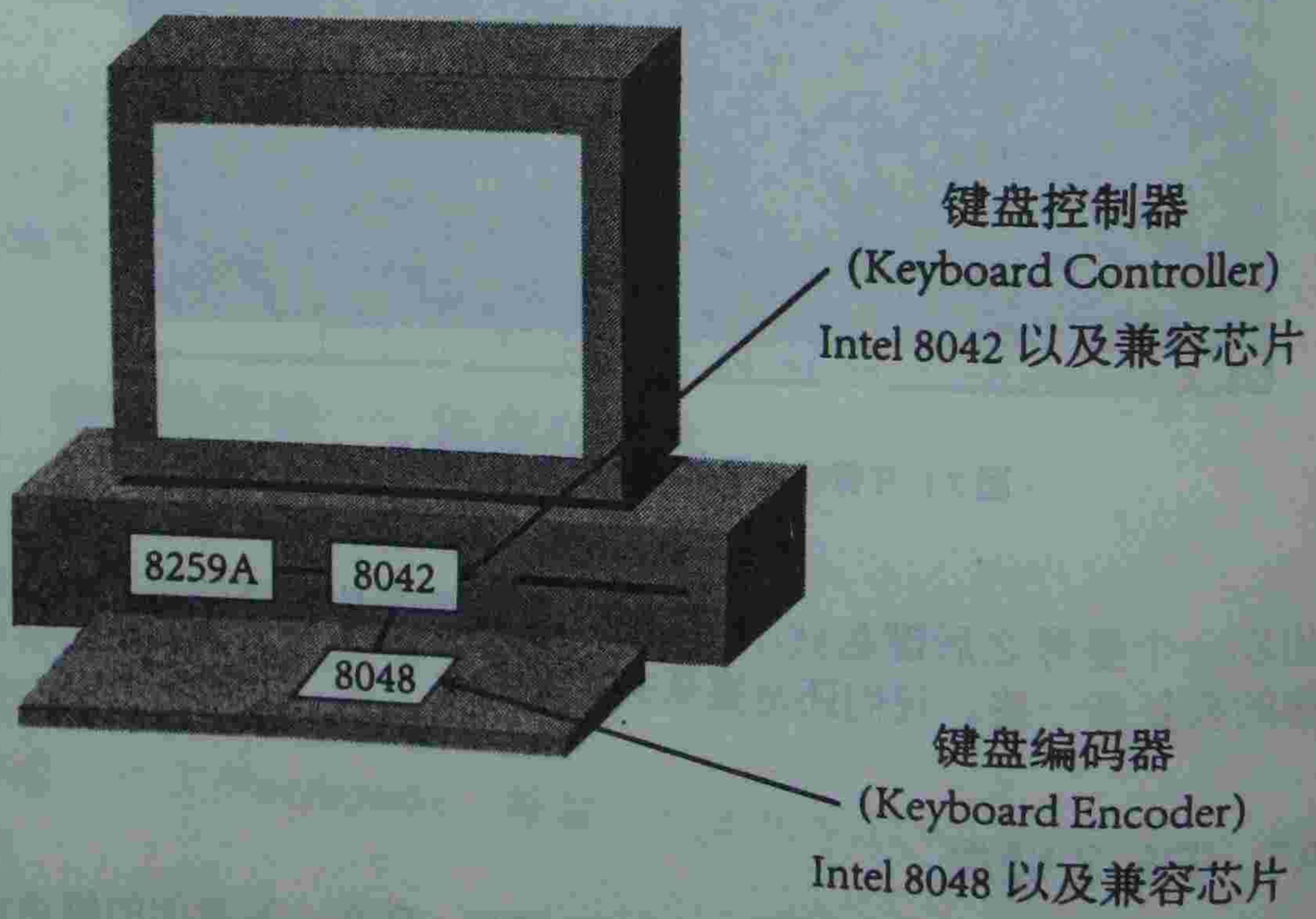


图7.3 键盘和主机连接示意图

敲击键盘有两个方面的含义：动作和内容。动作可以分解成三类：按下、保持按住的状态以及放开；内容则是键盘上不同的键，字母键还是数字键，回

车键还是箭头键。所以，根据敲击动作产生的编码，8048既要反映“哪个”按键产生了动作，还要反映产生了“什么”动作。

敲击键盘所产生的编码被称作扫描码（Scan Code），它分为Make Code和Break Code两类。当一个键被按下或者保持住按下时，将会产生Make Code，当键弹起时，产生Break Code。除了Pause键之外，每一个按键都对应一个Make Code和一个Break Code。

扫描码总共有三套，叫做Scan code set 1、Scan code set 2和Scan code set 3。Scan code set 1是早期的XT键盘使用的，现在的键盘默认都支持Scan code set 2，而Scan code set 3很少使用。

当8048检测到一个键的动作后，会把相应的扫描码发送给8042，8042会把它转换成相应的Scan code set 1扫描码，并将其放置在输入缓冲区中，然后8042告诉8259A产生中断（IRQ1）。如果此时键盘又有新的键被按下，8042将不再接收，一直到缓冲区被清空，8042才会收到更多的扫描码。

现在，你一定明白了为什么图7.1中只打印了一个字符，因为我们的键盘中断处理例程什么都没做。只有我们把扫描码从缓冲区中读出来后，8042才能继续响应新的按键。

那么如何从缓冲区中读取扫描码呢？这就需要我们来看一看8042了。

8042包含表7.1所示的一些寄存器。

表7.1 8042的寄存器

寄存器名称	寄存器大小	端口	R/W	用法
输出缓冲区	1 BYTE	0x60	Read	读输出缓冲区
输入缓冲区	1 BYTE	0x60	Write	写输入缓冲区
状态寄存器	1 BYTE	0x64	Read	读状态寄存器
控制寄存器	1 BYTE	0x64	Write	发送命令

注意：由于8042处在8048和系统的中间，所以输入和输出是相对的。比如，8042从8048输入数据，然后输出到系统。因此，表7.1中的“出”和“入”是相对于系统而言的。

对于输入和输出缓冲区，可以通过in和out指令来进行相应的读取操作。也就是说，一个in al, 0x60指令就可以读取扫描码了。

事不宜迟，马上修改程序。在keyboard_handler中添加下面一句：

```
in_byte(0x60);
```

运行，结果如图7.4所示。

按一下键，出现两个星号，再按一下，又出现两个。按了两下按键的结果是出现了4个星号！你马上明白过来了，每次按键都产生一个Make Code和一个Break Code，所以总共产生了4次中断。

只打印星号显然不够有趣，我们把收到的扫描码打印一下看看，进一步修改keyboard_handler（代码7.4）。

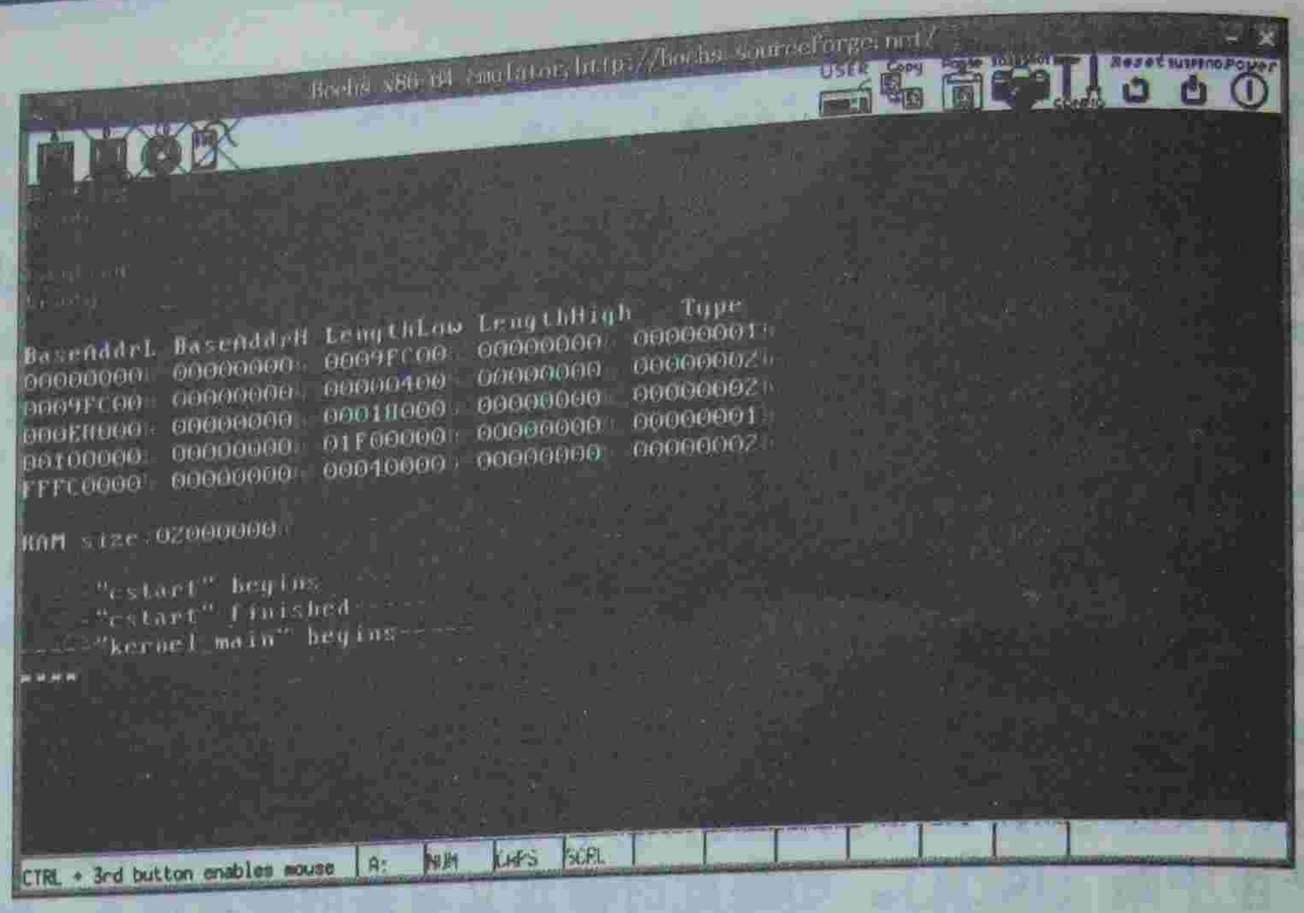


图 7.4 每发生一次键盘中断就读取一次 8042 输出缓冲区并打印一次星号

代码 7.4 修改键盘中断（chapter7/b/kernel/keyboard.c）

```
18  PUBLIC void keyboard_handler(int irq)
19  {
20          /* disp_str("*"); */
21          u8 scan_code = in_byte(0x60);
22          disp_int(scan_code);
23  }
```

接下来在运行的时候按两个键：字符“a”和“9”，看出现什么？结果如图7.5所示。

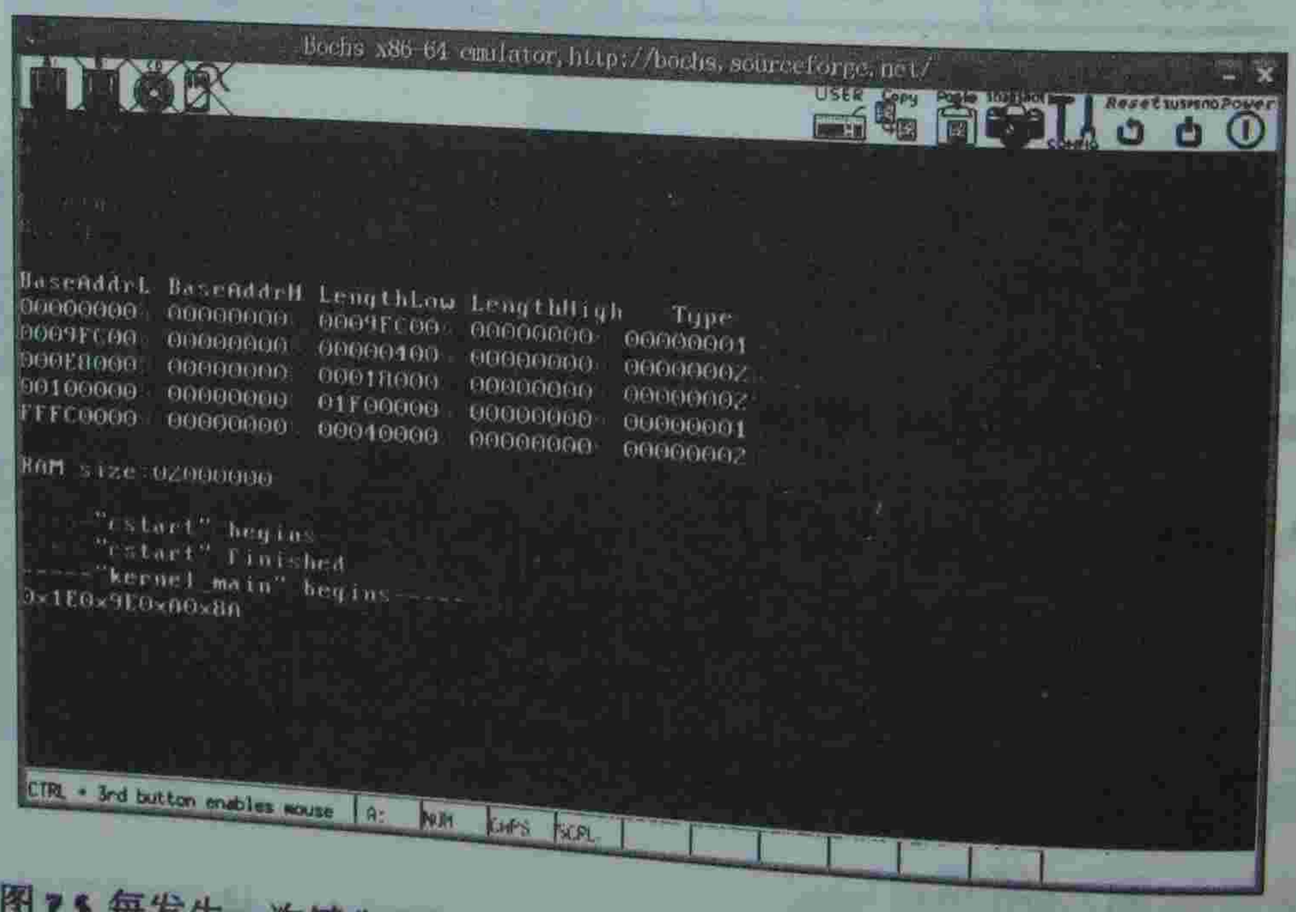


图 7.5 每发生一次键盘中断就读取一次 8042 输出缓冲区并打印读取的值

总共出现 4 组代码：0x1E、0x9E、0xA、0x8A。对照表7.2发现，它们恰恰就是字符“a”和“9”的 Make Code 和 Break Code。

注意：由于 a 和 A 是同一个键，所以它们的扫描码是一样的（事实上根本就不是“它们”而是“它”，因为是同一个键），如果按下“左 Shift+a”，将得到这样的输出：0x2A0x1E0x9E0xAA，分别是左 Shift 键的 Make Code、a 的 Make Code、a 的 Break Code 以及左 Shift 键的 Break Code。所以，按下“Shift+a”得到 A 是软件的功劳，键盘和 8042 是不管这些的，在你自己的操作系统中，甚至可以让“Shift+a”去对应 S 或者 T，只要你习惯就行。同理，按下任何的键，不管是单键还是组合键，想让屏幕输出什么，或者产生什么反应，都是由软件来控制的。虽然增加了操作系统的复杂性，但这种机制无疑是相当灵活的。

同时你也看到，虽然键盘支持的是 Scan code set 2，最终又转化成了 Scan code set 1。这是基于为 XT 键盘写的程序兼容性而考虑的。又是兼容性问题，你脑海里会飞速闪过 A20、内存空洞、被分割得乱七八糟的描述符等一系列令人厌恶的回忆。是啊，有时候，软硬件的设计者对兼容性的考虑的确占去了太多的精力，可是又有什么办法呢？

不过，我们马上就可以根据扫描码来处理键盘输入了，只要根据表格建立一种对应关系就够了。因此，忘掉这些不快吧。

表 7.2 Scan code set 1 扫描码

Key	Make	Break	Key	Make	Break
A	1E	9E	Tab	0F	8F
B	30	B0	C Lock	3A	BA
C	2E	AE	L Shift	2A	AA
D	20	A0	L Ctrl	1D	9D
E	12	92	L GUI	E0,5B	E0,DB
F	21	A1	L Alt	38	B8
G	22	A2	R Shift	36	B6
H	23	A3	R Ctrl	E0,1D	E0,9D
I	17	97	R GUI	E0,5C	E0,DC
J	24	A4	R Alt	E0,38	E0,B8
K	25	A5	APPS	E0,5D	E0,DD
L	26	A6	Enter	1C	9C
M	32	B2	Esc	1	81
N	31	B1	F1	3B	BB
O	18	98	F2	3C	BC
P	19	99	F3	3D	BD
Q	10	19	F4	3E	BE
R	13	93	F5	3F	BF
S	1F	9F	F6	40	C0
T	14	94	F7	41	C1
U	16	96	F8	42	C2
V	2F	AF	F9	43	C3
W	11	91	F10	44	C4
X	2D	AD	F11	57	D7
Y	15	95	F12	58	D8
Z	2C	AC	Print Screen	E0,2A E0,37	E0,B7 E0,AA

（续表）

Key	Make	Break
0	0B	8B
1	2	82
2	3	83
3	4	84
4	5	85
5	6	86
6	7	87*
7	8	88
8	9	89
9	0A	8A
'	29	89
-	0C	8C
=	0D	8D
\	2B	AB
Space	39	B9
KP/	E0,35	E0,B5
KP-	4A	CA
KP.	53	D3
KP0	52	D2
KP1	4F	CF
KP2	50	D0
KP3	51	D1
KP4	4B	CB
KP5	4C	CC
KP6	4D	CD
KP7	47	C7
KP8	48	C8
KP9	49	C9
]	1B	9B
;	27	A7
'	28	A8
,	33	B3
.	34	B4
/	35	B5
Power	E0,5E	E0,DE
Sleep	E0,5F	E0,DF
Wake	E0,63	E0,E3

Key	Make	Break
Scroll	46	C6
Pause	E1,1D 45,E1 9D,C5	NONE
[	1A	9A
Insert	E0,52	E0,D2
Home	E0,47	E0,C7
PgUp	E0,49	E0,C9
Delete	E0,53	E0,D3
End	E0,4F	E0,CF
PgDn	E0,51	E0,D1
↑	E0,48	E0,C8
←	E0,4B	E0,CB
↓	E0,50	E0,D0
→	E0,4D	E0,CD
Num	45	C5
Backspace	0E	8E
KP*	37	B7
KP+	4E	CE
KP EN	E0,1C	E0,9C
Next Track*	E0,19	E0,99
Previous Track*	E0,10	E0,90
Stop*	E0,24	E0,A4
Play/Pause*	E0,22	E0,A2
Mute*	E0,20	E0,A0
Volume Up*	E0,30	E0,B0
Volume Down*	E0,2E	E0,AE
Media Select*	E0,6D	E0,ED
E-mail*	E0,6C	E0,EC
Calculator*	E0,21	E0,A1
My Computer*	E0,6B	E0,EB
WWW Search*	E0,65	E0,E5
WWW Home*	E0,32	E0,B2
WWW Back*	E0,6A	E0,EA
WWW Forward*	E0,69	E0,E9
WWW Stop*	E0,68	E0,E8
WWW Refresh*	E0,67	E0,E7
WWW Favorites*	E0,66	E0,E6

带*号的为Windows多媒体键盘专有按钮

7.1.4 用数组表示扫描码

现在扫描码已被轻松获得，可是我们该如何将扫描码和相应字符对应起来呢？从表7.2中可以看出，Break Code是Make Code与0x80进行“或（OR）”操作的结果。可是Make Code和相应键的对应关系好像找不到什么规律。不过还好，扫描码是一些数字，我们可以建立一个数组，以扫描码为下标，对应的元素就是相应的字符。要注意的是，其中以0xE0以及0xE1开头的扫描码要区别对待。

我们把这个数组做成代码7.5这个样子。其中每3个值一组（MAP_COLS被定义成3），分别是单独按某键、Shift＋某键和有0xE0前缀的扫描码对应的字符。Esc、Enter等被定义成了宏，宏的具体数值无所谓，只要不会造成冲突和混淆，让操作系统认识就可以。

代码7.5 扫描码解析数组（chapter7/c/include/keymap.h）

```
u32 keymap[NR_SCAN_CODES * MAP_COLS] = {

/* scan-code                    !Shift          Shift           E0 XX   */
/* ============================================================================ */
/* 0x00 - none          */     0,              0,              0,
/* 0x01 - ESC           */     ESC,            ESC,            0,
/* 0x02 - '1'           */     '1',            '!',            0,
/* 0x03 - '2'           */     '2',            '@',            0,
/* 0x04 - '3'           */     '3',            '#',            0,
/* 0x05 - '4'           */     '4',            '$',            0,
/* 0x06 - '5'           */     '5',            '%',            0,
/* 0x07 - '6'           */     '6',            '^',            0,
/* 0x08 - '7'           */     '7',            '&',            0,
/* 0x09 - '8'           */     '8',            '*',            0,
/* 0x0A - '9'           */     '9',            '(',            0,
/* 0x0B - '0'           */     '0',            ')',            0,
/* 0x0C - '-'           */     '-',            '_',            0,
/* 0x0D - '='           */     '=',            '+',            0,
/* 0x0E - BS            */     BACKSPACE,      BACKSPACE,      0,
/* 0x0F - TAB           */     TAB,            TAB,            0,
/* 0x10 - 'q'           */     'q',            'Q',            0,
/* 0x11 - 'w'           */     'w',            'W',            0,
/* 0x12 - 'e'           */     'e',            'E',            0,
/* 0x13 - 'r'           */     'r',            'R',            0,
/* 0x14 - 't'           */     't',            'T',            0,
/* 0x15 - 'y'           */     'y',            'Y',            0,
/* 0x16 - 'u'           */     'u',            'U',            0,
/* 0x17 - 'i'           */     'i',            'I',            0,
/* 0x18 - 'o'           */     'o',            'O',            0,
/* 0x19 - 'p'           */     'p',            'P',            0,
/* 0x1A - '['           */     '[',            '{',            0,
/* 0x1B - ']'           */     ']',            '}',            0,
/* 0x1C - CR/LF         */     ENTER,          ENTER,          PAD_ENTER,
/* 0x1D - l. Ctrl       */     CTRL_L,         CTRL_L,         CTRL_R,
/* 0x1E - 'a'           */     'a',            'A',            0,
/* 0x1F - 's'           */     's',            'S',            0,
/* 0x20 - 'd'           */     'd',            'D',            0,
/* 0x21 - 'f'           */     'f',            'F',            0,
/* 0x22 - 'g'           */     'g',            'G',            0,
/* 0x23 - 'h'           */     'h',            'H',            0,
/* 0x24 - 'j'           */     'j',            'J',            0,
/* 0x25 - 'k'           */     'k',            'K',            0,
/* 0x26 - 'l'           */     'l',            'L',            0,
/* 0x27 - ';'           */     ';',            ':',            0,
/* 0x28 - '\''          */     '\'',           '"',            0,
/* 0x29 - '`'           */     '`',            '~',            0,
/* 0x2A - l. SHIFT      */     SHIFT_L,        SHIFT_L,        0,
/* 0x2B - '\'           */     '\\',           '|',            0,
/* 0x2C - 'z'           */     'z',            'Z',            0,
/* 0x2D - 'x'           */     'x',            'X',            0,
/* 0x2E - 'c'           */     'c',            'C',            0,
/* 0x2F - 'v'           */     'v',            'V',            0,
/* 0x30 - 'b'           */     'b',            'B',            0,
/* 0x31 - 'n'           */     'n',            'N',            0,
/* 0x32 - 'm'           */     'm',            'M',            0,
/* 0x33 - ','           */     ',',            '<',            0,
/* 0x34 - '.'           */     '.',            '>',            PAD_SLASH,
/* 0x35 - '/'           */     '/',            '?',            0,
/* 0x36 - r. SHIFT      */     SHIFT_R,        SHIFT_R,
```

```
78  /* 0x37 - '*'      */  '*',        ALT_L,      '*',        0,
79  /* 0x38 - ALT      */  ' ',        CAPS_LOCK,  ' ',        ALT_R,
80  /* 0x39 - ' '      */  CAPS_LOCK,  F1,         CAPS_LOCK,  0,
81  /* 0x3A - CapsLock */  F1,         F2,         F1,         0,
82  /* 0x3B - F1       */  F2,         F3,         F2,         0,
83  /* 0x3C - F2       */  F3,         F4,         F3,         0,
84  /* 0x3D - F3       */  F4,         F5,         F4,         0,
85  /* 0x3E - F4       */  F5,         F6,         F5,         0,
86  /* 0x3F - F5       */  F6,         F7,         F6,         0,
87  /* 0x40 - F6       */  F7,         F8,         F7,         0,
88  /* 0x41 - F7       */  F8,         F9,         F8,         0,
89  /* 0x42 - F8       */  F9,         F10,        F9,         0,
90  /* 0x43 - F9       */  F10,        NUM_LOCK,   F10,        0,
91  /* 0x44 - F10      */  NUM_LOCK,   SCROLL_LOCK, NUM_LOCK,  0,
92  /* 0x45 - NumLock  */  SCROLL_LOCK, PAD_HOME,   SCROLL_LOCK, 0,
93  /* 0x46 - ScrLock  */  PAD_HOME,   PAD_UP,      '7',        HOME,
94  /* 0x47 - Home     */  PAD_UP,     PAD_PAGEUP, '8',        UP,
95  /* 0x48 - CurUp    */  PAD_PAGEUP, PAD_MINUS,  '9',        PAGEUP,
96  /* 0x49 - PgUp     */  PAD_MINUS,  PAD_LEFT,   '-',        0,
97  /* 0x4A - '-'      */  PAD_LEFT,   PAD_MID,    '4',        LEFT,
98  /* 0x4B - Left     */  PAD_MID,    PAD_RIGHT,  '5',        0,
99  /* 0x4C - MID      */  PAD_RIGHT,  PAD_PLUS,   '6',        RIGHT,
100 /* 0x4D - Right    */  PAD_PLUS,   PAD_END,    '+',        0,
101 /* 0x4E - '+'      */  PAD_END,    PAD_DOWN,   '1',        END,
102 /* 0x4F - End      */  PAD_DOWN,   PAD_PAGEDOWN, '2',      DOWN,
103 /* 0x50 - Down     */  PAD_PAGEDOWN, PAD_INS,  '3',        PAGEDOWN,
104 /* 0x51 - PgDown   */  PAD_INS,    PAD_DOT,    '0',        INSERT,
105 /* 0x52 - Insert   */  PAD_DOT,    0,          '.',        DELETE,
106 /* 0x53 - Delete   */  0,          0,          0,          0,
107 /* 0x54 - Enter    */  0,          0,          0,          0,
108 /* 0x55 - ???      */  0,          0,          0,          0,
109 /* 0x56 - ???      */  F11,        F11,        0,          0,
110 /* 0x57 - F11      */  F12,        F12,        0,          0,
111 /* 0x58 - F12      */  0,          0,          0,          0,
112 /* 0x59 - ???      */  0,          0,          0,          0,
113 /* 0x5A - ???      */  0,          0,          GUI_L,      0,
114 /* 0x5B - ???      */  0,          0,          GUI_R,      0,
115 /* 0x5C - ???      */  0,          0,          APPS,       0,
116 /* 0x5D - ???      */  0,          0,          0,
117 /* 0x5E - ???      */  0,
    ...
150 /* 0x7F - ???      */  0,          0,          0
151 };
```

举个例子，如果获得的扫描码是0x1F，我们应该在代码7.5中很容易看到它对应的是字母“s”。在写程序的时候，应该用keymap[0x1F*MAP_COLS]来表示0x1F对应的字符。如果获得的扫描码是0x2A0x1E，它是左 Shift 键的 Make Code 和字符“a”的 Make Code 连在一起，说明按下 Shift 还没有弹起的时候 a 又被按下，因此应该用keymap[0x1E*MAP_COLS+1]表示这一行为的结果，即大写字母 A。

存在0xE0的情况也是类似的，如果我们收到的扫描码是0xE00x47，就应该去找keymap[0x47*MAP_COLS+2]，它对应的是 Home。

但是问题出现了。从表7.2可以知道，8042 的输入缓冲区大小只有一个字节，所以当一个扫描码有不止一个字符时，实际上会产生不止一次中断。也就是说，如果我们按一下 Shift+A，产生的0x2A0x1E0x9E0xAA是 4 次中断接收来的。这就给我们的程序实现带来了困难，因为第一次中断时收到的0x2A无法让我们知道用户最终会完成什么，说不定是按下 Shift 又释放，也可能是 Shift+Z 而不是 Shift+A。

于是，当接收到类似0x2A这样的值的时候，需要先把它保存起来，在随后的过程中慢慢解析用户到底做了什么。

保存一个字符可以用全局变量来完成。可是，由于扫描码的值和长度都不一样，这项工作做起来可能并不简单。而且我们可以想像，键盘操作必将是频繁而且复杂的，如果把得到扫描码之后相应的后续操作都放在键盘中断处理中，最后keyboard_handler会变得很大，这不是一个好主意。在这里，向前辈学习，建立一个缓冲区，让keyboard_handler将每次收到的扫描码放入这个缓冲区，然后建立一个新的任务专门用来解析它们并做相应处理。

注意，这一章的代码很多地方也是从 Minix 借鉴的，只是我们的代码要简单得多。

7.1.5 键盘输入缓冲区

我们先来建立一个缓冲区（这个缓冲区的结构是借鉴来的），用以放置中断例程收到的扫描码（代码7.6）。

代码 7.6 键盘缓冲区（chapter7/c/include/keyboard.h）

```
122 /* Keyboard structure, 1 per console. */
123 typedef struct s_kb {
124         char*   p_head;                 /* 指向缓冲区中下一个空闲位置 */
125         char*   p_tail;                 /* 指向键盘任务应处理的字节 */
126         int     count;                  /* 缓冲区中共有多少字节 */
127         char    buf[KB_IN_BYTES];       /* 缓冲区 */
128 }KB_INPUT;
```

这个缓冲区的用法如图7.6所示，白色框表示空闲字节，灰色框表示已用字节。在执行写操作的时候要注意，如果已经到达缓冲区末尾，则应将指针移到开头。

对照图7.6，我们可以容易地对缓冲区进行添加操作，如代码7.7所示。

代码 7.7 修改后的 keyboard_handler（chapter7/c/kernel/keyboard.c）

```
16 PRIVATE KB_INPUT      kb_in;
17
18 /*======================================================================*
19                             keyboard_handler
20  *======================================================================*/
21 PUBLIC void keyboard_handler(int irq)
22 {
23         u8 scan_code = in_byte(KB_DATA);
24
25         if (kb_in.count < KB_IN_BYTES) {
26                 *(kb_in.p_head) = scan_code;
27                 kb_in.p_head++;
28                 if (kb_in.p_head == kb_in.buf + KB_IN_BYTES) {
29                         kb_in.p_head = kb_in.buf;
30                 }
31                 kb_in.count++;
32         }
33 }
```

代码很简单，但要注意，如果缓冲区已满，这里使用的策略是直接就把收到的字节丢弃。其中的kb_in，由于我们只在keyboard.c中使用，于是把它声明成一个PRIVATE变量（PRIVATE的定义位于const.h中，被定义成了static）。

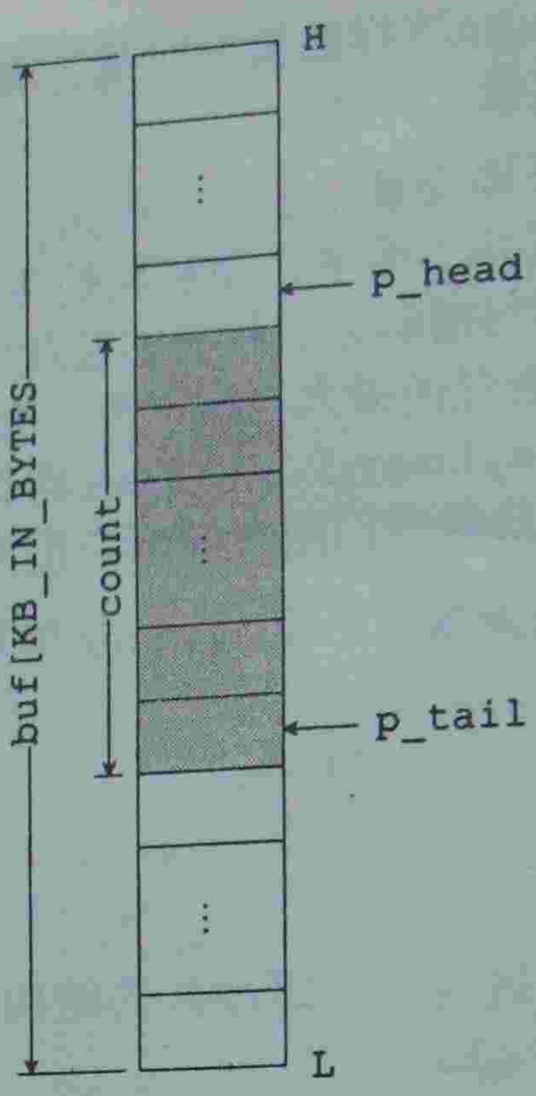


图 7.6 键盘缓冲区示意图

注意，kb_in的成员需要初始化，初始化的代码放在init_keyboard()中（代码7.8）。

代码 7.8 修改后的 keyboard_handler（chapter7/c/kernel/keyboard.c）

```
39  PUBLIC void init_keyboard()
40  {
⇒ 41        kb_in.count = 0;
⇒ 42        kb_in.p_head = kb_in.p_tail = kb_in.buf;
43
44        put_irq_handler(KEYBOARD_IRQ, keyboard_handler);/*设定键盘中断处理程序*/
45        enable_irq(KEYBOARD_IRQ);                       /*开键盘中断*/
46  }
```

为了保持kernel_main()的整洁，我们把时钟中断的设定和开启也放到单独的函数init_clock()中（代码7.9）。

代码 7.9 init_clock（chapter7/c/kernel/clock.c）

```
50  PUBLIC void init_clock()
51  {
52        /* 初始化 8253 PIT */
53        out_byte(TIMER_MODE, RATE_GENERATOR);
54        out_byte(TIMER0, (u8) (TIMER_FREQ/HZ) );
55        out_byte(TIMER0, (u8) ((TIMER_FREQ/HZ) >> 8));
56
57        put_irq_handler(CLOCK_IRQ, clock_handler);      /* 设定时钟中断处理程序 */
58        enable_irq(CLOCK_IRQ);                          /* 让8259A可以接收时钟中断 */
59  }
```

这样，在kernel_main()中调用这两个函数就可以了。

7.1.6 用新加的任务处理键盘操作

添加一个任务是很简单的，我们在6.4.6节中已经做过总结。可是我必须提前告诉你，我们下面要添加的这个任务将来不仅会处理键盘操作，还将处理诸如屏幕输出等内容，这些操作共同组成同一个任务：终端任务。

关于终端任务所做的其他工作我们留到后面再介绍，现在，你可以认为它只处理键盘输入。

为了简化程序，在这个任务中，我们只是不停地调用keyboard.c中的函数keyboard_read()（代码7.10）。

代码 7.10 tty 任务（chapter7/c/kernel/tty.c）

```
21 PUBLIC void task_tty()
22 {
23         while (1) {
24                 keyboard_read();
25         }
26 }
```

我们暂时把所有对扫描码的处理都写进keyboard.c中。代码7.10中被tty使用的keyboard_read()可以如代码7.11这样来定义。

代码 7.11 keyboard_read（chapter7/c/kernel/keyboard.c）

```
52 PUBLIC void keyboard_read()
53 {
54         u8 scan_code;
55
56         if(kb_in.count > 0){
57                 disable_int();
58                 scan_code = *(kb_in.p_tail);
59                 kb_in.p_tail++;
60                 if (kb_in.p_tail == kb_in.buf + KB_IN_BYTES) {
61                         kb_in.p_tail = kb_in.buf;
62                 }
63                 kb_in.count--;
64                 enable_int();
65
66                 disp_int(scan_code);
67         }
68 }
```

其中，disable_int()和enable_int()的定义很简单（代码7.12）。

代码 7.12 disable_int 和 enable_int（chapter7/c/lib/kliba.asm）

```
201 ; ========================================================================
202 ;                  void disable_int();
203 ; ========================================================================
204 disable_int:
205         cli
206         ret
207
208 ; ========================================================================
209 ;                  void enable_int();
210 ; ========================================================================
211 enable_int:
212         sti
213         ret
```

其实像这样简单的两个汇编函数，我们完全可以用 C 语句中嵌入汇编的方式来做，而且，由于避免了调用函数的call指令和返回时的ret指令，因此更加节省时间。但是说实话，笔者本人比较讨厌 AT&T 语法的汇编，所以对于内联汇编，除非没有其他办法，否则笔者是不会使用的，即便它极其简单也避免使用。

回过头再说keyboard_read()，函数首先判断kb_in.count是否为 0，如不为 0，表明缓冲区中有扫描码，就开始读。读缓冲区开始时关闭了中断，到结束时才打开，因为kb_in作为一个整体，对其中的成员的操作应该是一气呵成不受打扰的。读操作相当于写操作的反过程。

好了，这样我们就完成了通过任务来处理扫描码（其实还没有开始处理，仅仅是打印数字而已）的代码，修改Makefile之后就可以 make 并运行了，不过运行结果显然与过去一样，因为我们并没有对扫描码进行解析。

7.1.7 解析扫描码

对扫描码的解析工作有一点烦琐，所以我们还是分步骤来完成它。

7.1.7.1 让字符显示出来

虽然我们已经有了一个数组keymap[]，但是请读者还是不要低估了解析扫描码的复杂性，因为它不但分为 Make Code 和 Break Code，而且有长有短，功能也很多样，比如 Home 键对应的是一种功能而不是一个 ASCII 码，所以要区别对待。我们还是由简到繁，先挑能打印的打印一下，请看代码7.13。

代码 7.13 解析扫描码（chapter7/d/kernel/keyboard.c）

```
53  PUBLIC void keyboard_read()
54  {
55          u8      scan_code;
56          char    output[2];
57          int     make;   /* TRUE: make;  FALSE: break. */
58
59          memset(output, 0, 2);
60
61          if(kb_in.count > 0){
62                  disable_int();
63                  scan_code = *(kb_in.p_tail);
64                  kb_in.p_tail++;
65                  if (kb_in.p_tail == kb_in.buf + KB_IN_BYTES) {
66                          kb_in.p_tail = kb_in.buf;
67                  }
68                  kb_in.count--;
69                  enable_int();
70
71                  /* 下面开始解析扫描码 */
72                  if (scan_code == 0xE1) {
73                          /* 暂时不做任何操作 */
74                  }
75                  else if (scan_code == 0xE0) {
76                          /* 暂时不做任何操作 */
77                  }
78                  else {  /* 下面处理可打印字符 */
79
80                          /* 首先判断Make Code 还是 Break Code */
81                          make = (scan_code & FLAG_BREAK ? FALSE : TRUE);
```

```
                        /* 如果是Make Code 就打印，是 Break Code 则不做处理 */
                        if(make) {
                                output[0] = keymap[(scan_code&0x7F)*MAP_COLS];
                                disp_str(output);
                        }
                }
                /* disp_int(scan_code); */
        }
}
```

由于要判断的东西太多，所以一下子把这个函数写得完善几乎是不可能的。我们就一步一步来，每走一步就 make 并运行一下看看效果，然后就知道接下来该怎么做了。

比如在代码7.11中，总体的思想就是0xE0和0xE1单独处理，因为从表7.2中知道，除去以这两个数字开头的扫描码，其余的都是单字节的。

暂时对0xE0和0xE1不加理会。如果遇到不是以它们开头的，则判断是 Make Code 还是 Break Code，如果是后者同样不加理会，如果是前者就打印出来。我们前文中讲过，Break Code 是 Make Code 与0x80进行“或（OR）”操作的结果，代码中的FLAG_BREAK就是被定义成了0x80。

你可能注意到一个细节，从keymap[]中取出字符的时候进行了一个“与”操作（scan_code&0x7F）。一方面，如果当前扫描码是 Break Code，“与”操作之后就变成 Make Code 了；另一方面，这样做也是为了避免越界的发生，因为数组keymap[]的大小是0x80。

接下来就可以 make 并运行了，图7.7就是初步运行的结果。

图 7.7 可以显示键入的小写字母和数字

运行时，我们敲入了“abc123”共计6个字母，它们被显示在了屏幕上。

7.1.7.2 处理 Shift、Alt、Ctrl

现在可以输入简单的字符和数字了，你一定迫不及待地想搞点更复杂的输入，比如按个 Shift 组合什么的。现在按下 Shift 只能看到一个奇怪的字符。下面

就来添加代码，使其能够响应这些功能键。

代码 7.14 解析扫描码（chapter7/e/kernel/keyboard.c）

```
19  PRIVATE int     code_with_E0 = 0;
20  PRIVATE int     shift_l;        /* l shift state */
21  PRIVATE int     shift_r;        /* r shift state */
22  PRIVATE int     alt_l;          /* l alt state   */
23  PRIVATE int     alt_r;          /* r left state  */
24  PRIVATE int     ctrl_l;         /* l ctrl state  */
25  PRIVATE int     ctrl_r;         /* l ctrl state  */
26  PRIVATE int     caps_lock;      /* Caps Lock     */
27  PRIVATE int     num_lock;       /* Num Lock      */
28  PRIVATE int     scroll_lock;    /* Scroll Lock   */
29  PRIVATE int     column;
    ...
69  PUBLIC void keyboard_read()
70  {
71          u8      scan_code;
72          char    output[2];
73          int     make;   /* TRUE: make;  FALSE: break. */
74
75          u32     key = 0;/* 用一个整型来表示一个键。比如，如果 Home 被按下，
76                           * 则 key 值将为定义在 keyboard.h 中的 'HOME'。
77                           */
78          u32*    keyrow; /* 指向 keymap[] 的某一行 */
79          /* memset(output, 0, 2); */
80
81          if(kb_in.count > 0){
82                  disable_int();
83                  scan_code = *(kb_in.p_tail);
84                  kb_in.p_tail++;
85                  if (kb_in.p_tail == kb_in.buf + KB_IN_BYTES) {
86                          kb_in.p_tail = kb_in.buf;
87                  }
88                  kb_in.count--;
89                  enable_int();
90
91                  /* 下面开始解析扫描码 */
92                  if (scan_code == 0xE1) {
93                          /* 暂时不做任何操作 */
94                  }
95                  else if (scan_code == 0xE0) {
96                          code_with_E0 = 1;
97                  }
98                  else {  /* 下面处理可打印字符 */
99
100                         /* 首先判断Make Code 还是 Break Code */
101                         make = (scan_code & FLAG_BREAK ? 0 : 1);
102
103                         /* 先定位到 keymap 中的行 */
104                         keyrow = &keymap[(scan_code & 0x7F) * MAP_COLS];
105
106                         column = 0;
107                         if (shift_l || shift_r) {
108                                 column = 1;
109                         }
110                         if (code_with_E0) {
111                                 column = 2;
112                                 code_with_E0 = 0;
113                         }
114
115                         key = keyrow[column];
116
117                         switch(key) {
118                         case SHIFT_L:
119                                 shift_l = make;
120                                 key = 0;
```

```
                break;
            case SHIFT_R:
                shift_r = make;
                key = 0;
                break;
            case CTRL_L:
                ctrl_l = make;
                key = 0;
                break;
            case CTRL_R:
                ctrl_r = make;
                key = 0;
                break;
            case ALT_L:
                alt_l = make;
                key = 0;
                break;
            case ALT_R:
                alt_l = make;
                key = 0;
                break;
            default:
                if (!make) {    /* 如果是 Break Code */
                    key = 0;        /* 忽略之 */
                }
                break;
            }

            /* 如果 Key 不为0说明是可打印字符，否则不做处理 */
            if(key){
                output[0] = key;
                disp_str(output);
            }
        }
    }
}
```

在代码7.14中，我们不但添加了处理 Shift 的代码，而且也对 Alt 和 Ctrl 键的状态进行了判断，只是暂时对它们还没有做任何的处理。

Shift、Alt、Ctrl 键共有 6 个（左右各 3 个），注意，最好不要把左右两个键不加区分，因为有一些软件需要区分对待，最简单而且经典的一个例子是超级玛丽，其中左右 Shift 功能是不一样的。为了充分识别它们而不把左右键混为一谈，我们声明了 6 个变量来记录它们的状态。当其中的某一个键被按下时，相应的变量值变为TRUE。比如，当我们按下左 Shift 键，shift_l就变为TRUE，如果它立即被释放，则shift_l又变回FALSE。如果当左 Shift 键被按下且未被释放时，又按下 a 键，则if (shift_l || shift_r)成立，于是column值变为 1，keymap[column]的取值就是keymap[]中第二列中相应的值，即大写字母 A。

同时，在这段代码中对以0xE0开头的扫描码也做了处理。其实它与按下 Shift 键类似，甚至还要更简单。当检测到一个扫描码的第一个字节是0xE0时，在将code_with_E0赋值为TRUE之后整个函数实际上就返回了。但动作显然没有结束，下一个字节马上进入处理过程，由于code_with_E0为TRUE，所以column值变成 2，于是 key 就变成keymap[]中第二列的值了。

在整个过程中我们还看到，虽然开始变量key的值是从keymap[]中得到的，但从整个函数执行的角度来看，遵循这样的原则：如果一个完整的操作还未结束（比如一个 2 字节的扫描码还未完全读入），则key赋值为 0，等到下一

次keyboard_read()被执行时再继续处理。也就是说，目前的情况是，一个完整的操作需要在keyboard_read()多次调用时完成。

好了，现在运行一下，结果如图7.8所示。

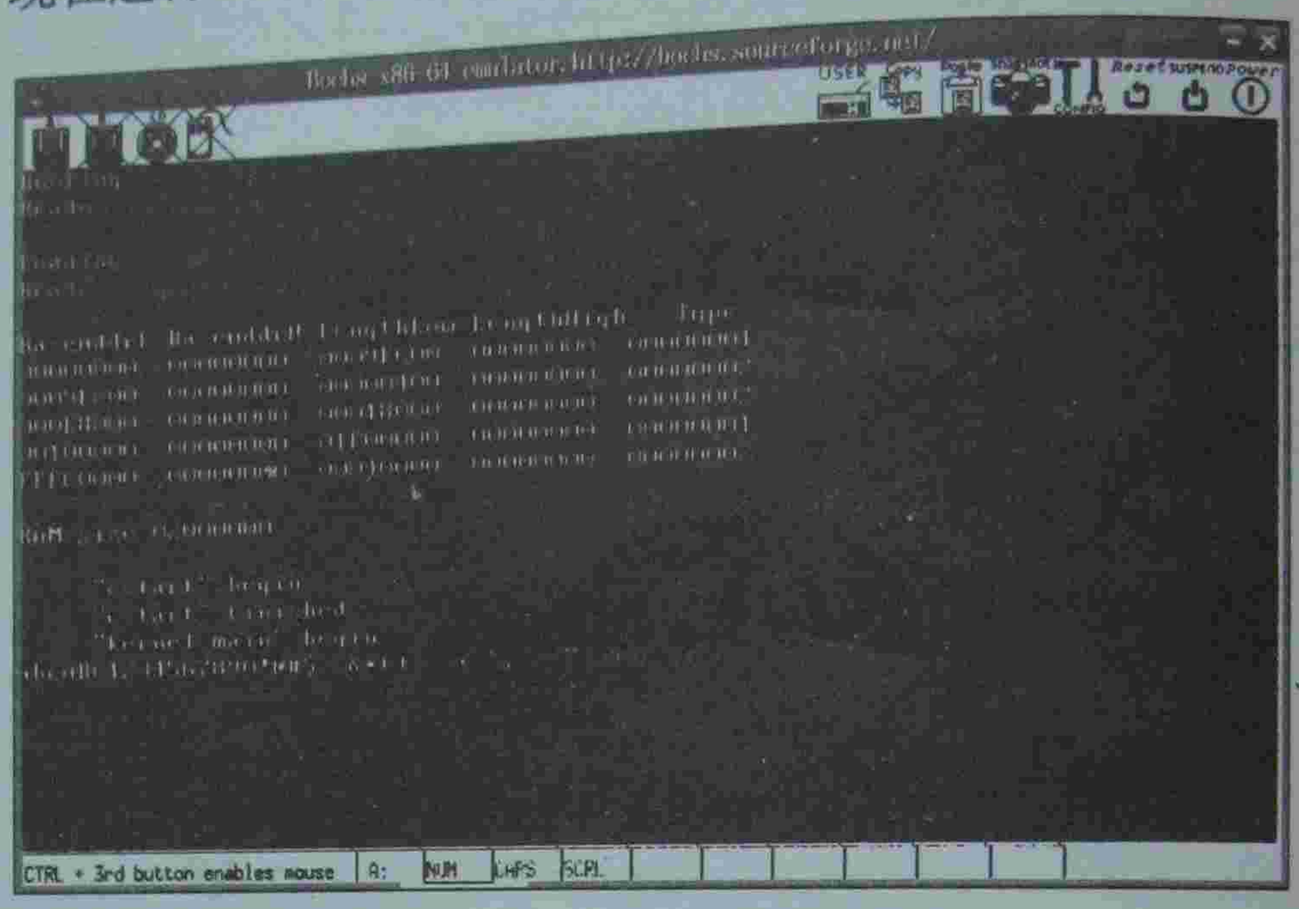


图7.8 可以显示大写字母和特殊符号

从图7.8看到，如今我们的OS已经可以识别大写字母，以及“!”、“@”等字符了。

7.1.7.3 处理所有按键

到现在为止，我们已经可以处理大部分的按键了，但是至少还存在两个问题。

1. 如果扫描码更加复杂一些，比如超过3个字符，如今的程序还不足以很好地处理。
2. 如果按下诸如F1、F2这样的功能键，系统会试图把它当做可打印字符来处理，从而打印出一个奇怪的符号。

我们首先来解决第一个问题。记得前面刚刚讲过，目前的情况是，一个完整的操作需要在keyboard_read()多次调用时完成。这不但让我们增加了一些全局变量，比如code_with_E0，而且让keyboard_read()理解起来也有些困难。符合逻辑的方法是，既然按下一个键会产生一到几字节的扫描码，就最好能够在一个过程中把它们全都读出来。这其实并不困难，只需要将从kb_in中读取字符的代码单独拿出来作为一个函数，在用到的时候调用它就可以了。看看代码7.15就明白了。

代码7.15 解析扫描码（chapter7/f/kernel/keyboard.c）

```
19 PRIVATE int    code_with_E0;
20 PRIVATE int    shift_l;      /* l shift state */
21 PRIVATE int    shift_r;      /* r shift state */
22 PRIVATE int    alt_l;        /* l alt state   */
23 PRIVATE int    alt_r;        /* r left state  */
```

```
24  PRIVATE int     ctrl_l;         /* l ctrl state  */
25  PRIVATE int     ctrl_r;         /* l ctrl state  */
26  PRIVATE int     caps_lock;      /* Caps Lock     */
27  PRIVATE int     num_lock;       /* Num Lock      */
28  PRIVATE int     scroll_lock;    /* Scroll Lock   */
29  PRIVATE int     column;
30
31  PRIVATE u8      get_byte_from_kbuf();
    ...
68  /*======================================================================*
69                               keyboard_read
70  *======================================================================*/
71  PUBLIC void keyboard_read()
72  {
73          u8      scan_code;
74          char    output[2];
75          int     make;   /* 1: make;  0: break. */
76
77          u32     key = 0;/* 用一个整型来表示一个键。比如，如果 Home 被按下，
78                           * 则 key 值将为定义在 keyboard.h 中的 'HOME'。
79                           */
80          u32*    keyrow; /* 指向 keymap[] 的某一行 */
81
82          if(kb_in.count > 0){
83                  code_with_E0 = 0;
84
85                  scan_code = get_byte_from_kbuf();
86
87                  /* 下面开始解析扫描码 */
88                  if (scan_code == 0xE1) {
89                          int i;
90                          u8 pausebrk_scode[] = {0xE1, 0x1D, 0x45,
91                                                 0xE1, 0x9D, 0xC5};
92                          int is_pausebreak = 1;
93                          for(i=1;i<6;i++){
94                                  if (get_byte_from_kbuf() != pausebrk_scode[i]) {
95                                          is_pausebreak = 0;
96                                          break;
97                                  }
98                          }
99                          if (is_pausebreak) {
100                                 key = PAUSEBREAK;
101                         }
102                 }
103         else if (scan_code == 0xE0) {
104                 scan_code = get_byte_from_kbuf();
105
106                 /* PrintScreen 被按下 */
107                 if (scan_code == 0x2A) {
108                         if (get_byte_from_kbuf() == 0xE0) {
109                                 if (get_byte_from_kbuf() == 0x37) {
110                                         key = PRINTSCREEN;
111                                         make = 1;
112                                 }
113                         }
114                 }
115                 /* PrintScreen 被释放 */
116                 if (scan_code == 0xB7) {
117                         if (get_byte_from_kbuf() == 0xE0) {
118                                 if (get_byte_from_kbuf() == 0xAA) {
119                                         key = PRINTSCREEN;
120                                         make = 0;
121                                 }
122                         }
123                 }
124                 /* 不是PrintScreen，此时scan_code为0xE0紧跟的那个值。 */
125                 if (key == 0) {
```

```
                code_with_E0 = 1;
            }
        }
        if ((key != PAUSEBREAK) && (key != PRINTSCREEN)) {
            /* 首先判断Make Code 还是 Break Code */
            make = (scan_code & FLAG_BREAK ? 0 : 1);

            /* 先定位到 keymap 中的行 */
            keyrow = &keymap[(scan_code & 0x7F) * MAP_COLS];

            column = 0;
            if (shift_l || shift_r) {
                column = 1;
            }
            if (code_with_E0) {
                column = 2;
                code_with_E0 = 0;
            }

            key = keyrow[column];

            switch(key) {
            case SHIFT_L:
                shift_l = make;
                key = 0;
                break;
            case SHIFT_R:
                shift_r = make;
                key = 0;
                break;
            case CTRL_L:
                ctrl_l = make;
                key = 0;
                break;
            case CTRL_R:
                ctrl_r = make;
                key = 0;
                break;
            case ALT_L:
                alt_l = make;
                key = 0;
                break;
            case ALT_R:
                alt_l = make;
                key = 0;
                break;
            default:
                if (!make) {    /* 如果是 Break Code */
                    key = 0;/* 忽略之 */
                }
                break;
            }

            /* 如果 Key 不为0说明是可打印字符，否则不做处理 */
            if (key) {
                output[0] = key;
                disp_str(output);
            }
        }
    }
}

/*======================================================================*
                           get_byte_from_kbuf
 *======================================================================*/
PRIVATE u8 get_byte_from_kbuf()         /* 从键盘缓冲区中读取下一个字节 */
```

```
193         u8 scan_code;
194
195         while (kb_in.count <= 0) {}   /* 等待下一个字节到来 */
196
197         disable_int();
198         scan_code = *(kb_in.p_tail);
199         kb_in.p_tail++;
200         if (kb_in.p_tail == kb_in.buf + KB_IN_BYTES) {
201                 kb_in.p_tail = kb_in.buf;
202         }
203         kb_in.count--;
204         enable_int();
205
206         return scan_code;
207 }
```

这段代码有点长，但实际上除了把函数get_byte_from_kbuf()单独拿出来之外，只是单独处理了 Pause 和 PrintScreen 两个按键而已。不过这样一来，整个处理过程就发生了变化，每一次调用keyboard_read()都可以处理一个相对完整的过程。比如按下右侧的 Alt 键产生的0xE0、0x38，不必调用两次keyboard_read()。而且，如今所有的按键都已经在处理范围之内了。

不过，组合键的情况还是要多次调用keyboard_read()。想想也难怪，多个按键调用多次读操作，这也是合情合理的。

下面来解决刚刚提出的第二个问题，就是关于非打印字符的处理。

keyboard_read()这个函数只是负责读取扫描码就可以了，至于如何处理，不应该是它的职责，因为只有更高层次的软件才能根据具体情况做出不同的反应。这样看来，之前我们的打印操作其实也是越俎代庖了。

那么我们再将代码进行如下修改（代码7.16）。

代码 7.16 解析扫描码（chapter7/g/kernel/keyboard.c）

```
71  PUBLIC void keyboard_read()
72  {
    ...
129             if ((key != PAUSEBREAK) && (key != PRINTSCREEN)) {
130                     /* 首先判断Make Code 还是 Break Code */
131                     make = (scan_code & FLAG_BREAK ? 0 : 1);
132
133                     /* 先定位到 keymap 中的行 */
134                     keyrow = &keymap[(scan_code & 0x7F) * MAP_COLS];
135
136                     column = 0;
137                     if (shift_l || shift_r) {
138                             column = 1;
139                     }
140                     if (code_with_E0) {
141                             column = 2;
142                             code_with_E0 = 0;
143                     }
144
145                     key = keyrow[column];
146
147                     switch(key) {
148                     case SHIFT_L:
149                             shift_l = make;
150                             break;
151                     case SHIFT_R:
152                             shift_r = make;
153                             break;
```

```
                case CTRL_L:
                        ctrl_l = make;
                        break;
                case CTRL_R:
                        ctrl_r = make;
                        break;
                case ALT_L:
                        alt_l = make;
                        break;
                case ALT_R:
                        alt_l = make;
                        break;
                default:
                        break;
                }

                if (make) { /* 忽略 Break Code */
                        key |= shift_l  ? FLAG_SHIFT_L  : 0;
                        key |= shift_r  ? FLAG_SHIFT_R  : 0;
                        key |= ctrl_l   ? FLAG_CTRL_L   : 0;
                        key |= ctrl_r   ? FLAG_CTRL_R   : 0;
                        key |= alt_l    ? FLAG_ALT_L    : 0;
                        key |= alt_r    ? FLAG_ALT_R    : 0;

                        in_process(key);
                }
        }
}
```

这样，无论是Pause、Print Screen，还是其他以0xE0开头的扫描码或普通的单字符扫描码，都会交给函数in_process()来处理。而且，Shift、Alt、Ctrl键的状态会用设置相应位的方式通过Key表现出来。我们马上写一个简单的in_process()（代码7.17）。

代码7.17 in_process（chapter7/g/kernel/tty.c）

```
PUBLIC void in_process(u32 key)
{
        char output[2] = {'\0', '\0'};

        if (!(key & FLAG_EXT)) {
                output[0] = key & 0xFF;
                disp_str(output);
        }
}
```

注意，这里有一个小技巧。如果你打开keyboard.h，可以看到如代码7.18所示的情形。

代码7.18 键盘缓冲区（chapter7/g/include/keyboard.h）

```
/* Special keys */
#define ESC             (0x01 + FLAG_EXT)       /* Esc        */
#define TAB             (0x02 + FLAG_EXT)       /* Tab        */
#define ENTER           (0x03 + FLAG_EXT)       /* Enter      */
#define BACKSPACE       (0x04 + FLAG_EXT)       /* BackSpace  */

#define GUI_L           (0x05 + FLAG_EXT)       /* L GUI      */
#define GUI_R           (0x06 + FLAG_EXT)       /* R GUI      */
#define APPS            (0x07 + FLAG_EXT)       /* APPS */

/* Shift, Ctrl, Alt */
#define SHIFT_L         (0x08 + FLAG_EXT)       /* L Shift    */
```

```
#define SHIFT_R         (0x09 + FLAG_EXT)       /* R Shift      */
#define CTRL_L          (0x0A + FLAG_EXT)       /* L Ctrl       */
#define CTRL_R          (0x0B + FLAG_EXT)       /* R Ctrl       */
#define ALT_L           (0x0C + FLAG_EXT)       /* L Alt        */
#define ALT_R           (0x0D + FLAG_EXT)       /* R Alt        */
```

在所有的不可打印字符的定义中，都加了一个FLAG_EXT，这就使得我们在程序中可以非常容易地识别出来。所以当(!(key & FLAG_EXT))真时，就表明当前字符是一个可打印字符。

执行后的效果如图7.9所示。

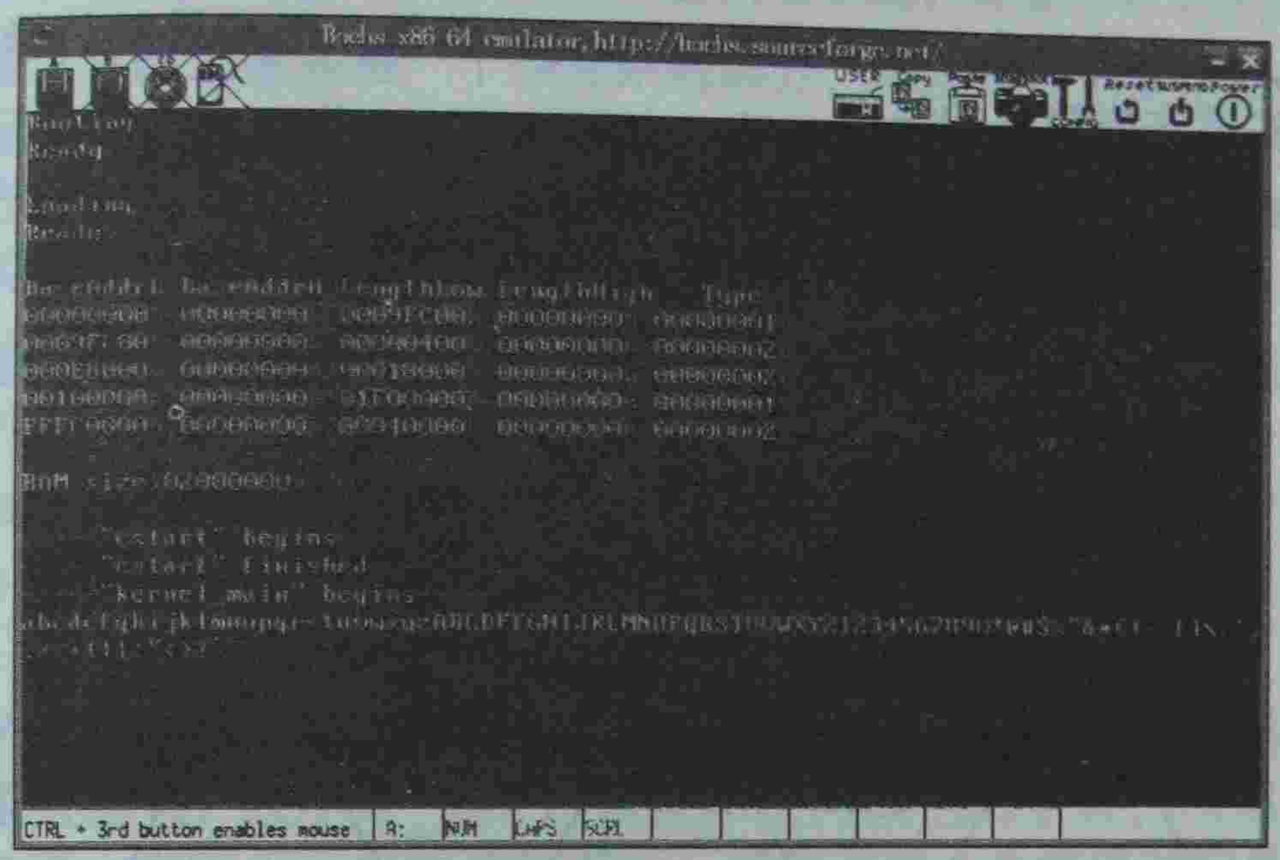


图 7.9 敲击无法处理的按键不再打印奇怪的符号

我们成功地输出了 26 个字母，包括大写和小写，同时输出了数字，以及其他一些字符。当你按下 F1、F2 等功能键时，程序并不做出反应。这些都表明我们的修改是成功的。

另外，不管是单键还是组合键，都使用一个 32 位整型数 key 来表示。因为可打印字符的 ASCII 码是 8 位，而我们将特殊的按键定义成了FLAG_EXT和一个单字节数的和，也不超过 9 位（可参考keyboard.h），这样，我们还剩余很多位来表示 Shift、Alt、Ctrl 等键的状态，一个整型记载的信息足够我们了解当前的按键情况。

7.2 显示器

你一定感到有点奇怪，我们刚刚还在讨论键盘，怎么又讲显示器了？笔者并不是有意打断你编写键盘驱动的兴致，只是，随着键盘模块的逐渐完善，我们越来越需要考虑它与屏幕输出之间的关系。I/O 包含两个方面——Input 和 Output——它们总是放在一起的。

记得我们在新添加终端进程的时候讲过，这个进程不仅处理键盘操作，还将处理诸如屏幕输出等内容。所以，在彻底完成键盘驱动之前，我们不得不先来了解一下终端的概念以及显示器的驱动方式。

那么终端到底指的是什么？我们现在先来认识一下。

7.2.1 初识 TTY

如果你用过 Linux 或者 UNIX，对 TTY 就一定不会陌生。很多时候，我们也称之为终端。对于终端最简单而形象的认识是，当你按 Alt+F1、Alt+F2、Alt+F3 等组合键时，会切换到不同的屏幕。在这些不同的屏幕中，可以分别有不同的输入和输出，相互之间并不受到彼此影响。在某个终端中，如果键入命令 tty，执行的结果将是当前的终端号。

实际上，终端的概念不仅仅是 Alt+Fn 这么简单，但在目前的操作系统中，我们暂时只实现这样简单的终端。

对于不同的 TTY，我们可以理解成图7.10的样式。

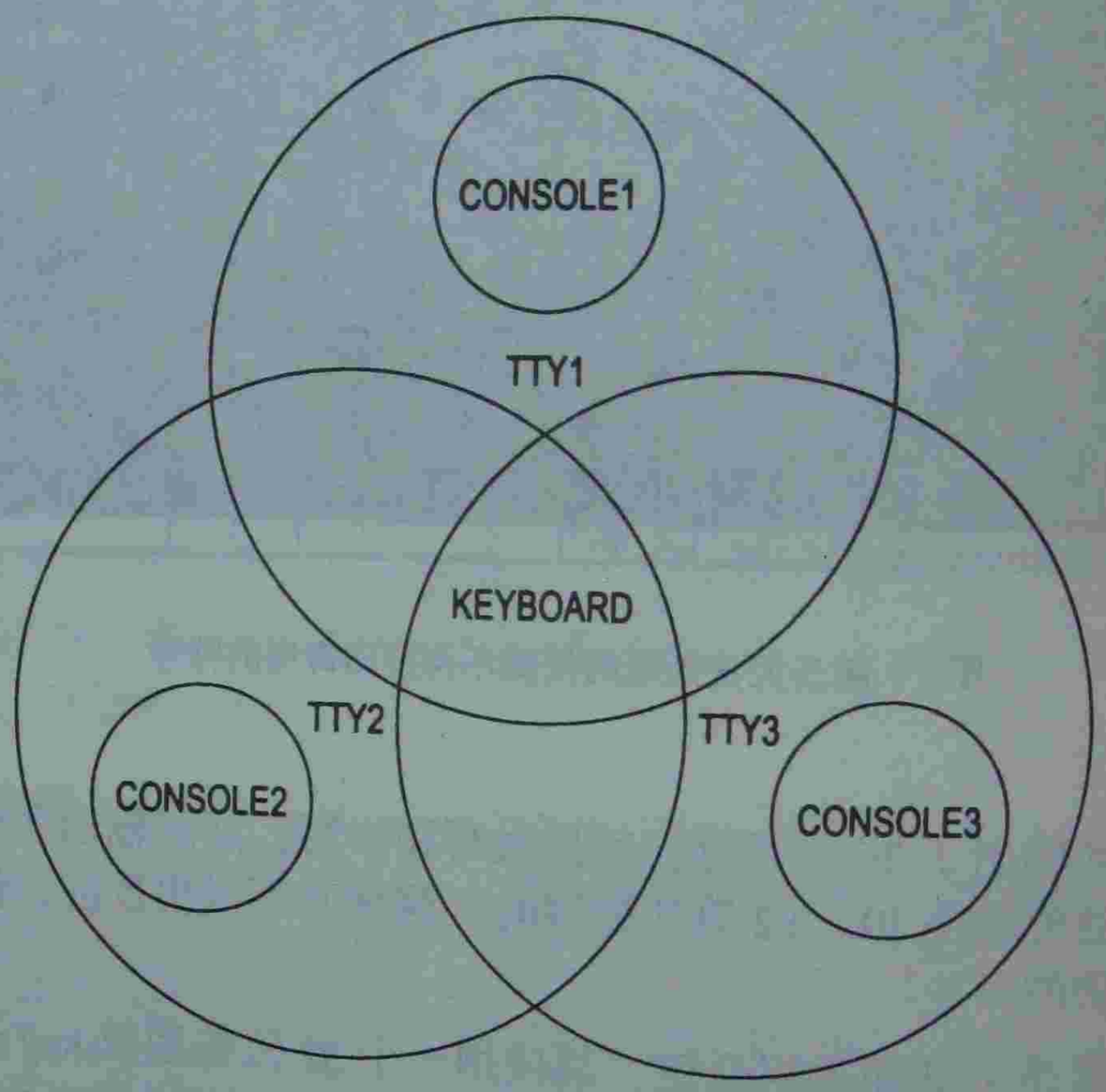


图 7.10 TTY

虽然不同的 TTY 对应的输入设备是同一个键盘，但输出却好比是在不同的显示器上，因为不同的 TTY 对应的屏幕画面可能是迥然不同的。实际上，我们当然是在使用同一个显示器，画面的不同只不过是因为显示了显存的不同位置罢了。

既然 3 个 CONSOLE 公用同一块显存，就必须有一种方式，在切换 CONSOLE 的瞬间，让屏幕显示显存中某个位置的内容。不用担心，通过简单的端口操作，这很容易做到。我们马上就会介绍。

7.2.2 基本概念

虽然在题目以及文中我们使用“显示器”这个字眼，但它并不是一个精确的称呼，因为我们操作的对象可能是显卡，或者仅仅是显存。不过没关系，开

始的不精确不代表我们不严谨，因为随着认识的深入，这些概念最终会清晰起来。目前，在模糊的地方我们可以暂时使用“视频”这个词。

到现在才来仔细介绍视频好像有点晚，因为从一开始我们写那个简单的Boot Sector的时候，就从来没有离开过对视频的操作——如果不是通过屏幕的反馈，我们怎么知道计算机在做些什么呢？

在最初那个Boot Sector中，打印字符是通过BIOS中断来实现的。但到了保护模式中，BIOS中断不再能用了，我们就在GDT中建立了一个段，它的开始地址是0xB8000，通过段寄存器gs对它进行写操作，从而实现数据的显示。到目前为止，我们对于视频模块的操作也仅限于此，想显示什么就mov而已。

但是，实际上视频是一个很复杂的部分，很多操作的复杂程度非mov能比。显示适配器可以被设置成不同的模式，用来显示更多的色彩、更华丽的图像和动画。当你面对色彩斑斓的图形界面时，当你用PC欣赏电影时，大概能够猜想出要实现它一定很不容易。

不过，目前我们还没必要搞得那么复杂，我们只要认识开机看到的默认模式就够了，那就是80×25文本模式。在这种模式下，显存大小为32KB，占用的范围为0xB8000～0xBFFFF。每2字节代表一个字符，其中低字节表示字符的ASCII码，高字节表示字符的属性。一个屏幕总共可以显示25行，每行80个字符。

虽然我们没有仔细介绍字符属性，不过却设置过显示字符的颜色，还编写了一个函数disp_color_str()来显示不同颜色的字符，所以，你大概已经了解一二了。实际上，在默认情况下，屏幕上每一个字符对应的2字节的定义如图7.11所示。

图7.11 屏幕上每一个字符对应的2字节

可以看到，低字节表示的是字符本身，高字节用来定义字符的颜色。颜色分前景和背景两部分，各占4位，其中低三位意义是相同的，表示颜色，但最高位作用不同。如果前景最高位为1的话，字符的颜色会比此位为0时亮一些；如果背景最高位为1，则显示出的字符将是闪烁的（注意是字符闪烁而不是背景闪烁）。更多细节请参考表7.3。

表7.3 字符属性位颜色详解

属性位				十六进制	意义	
B/I	R	G	B		作为背景	作为前景
0	0	0	0	0h	黑色	黑色
0	0	0	1	1h	蓝色	蓝色
0	0	1	0	2h	绿色	绿色
0	0	1	1	3h	青色	青色
0	1	0	0	4h	红色	红色
0	1	0	1	5h	洋红	洋红
0	1	1	0	6h	棕色	棕色
0	1	1	1	7h	白色	白色
1	0	0	0	8h	黑色（闪烁）	灰色
1	0	0	1	9h	蓝色（闪烁）	亮蓝
1	0	1	0	Ah	绿色（闪烁）	亮绿
1	0	1	1	Bh	青色（闪烁）	亮青
1	1	0	0	Ch	红色（闪烁）	亮红
1	1	0	1	Dh	洋红（闪烁）	亮紫
1	1	1	0	Eh	棕色（闪烁）	黄色
1	1	1	1	Fh	白色（闪烁）	亮白

现在我们来看一下第3章中代码3.1中的这几行代码：

```
mov ah, 0Ch ; 0000: 黑底    1100: 红字
mov al, 'P'
mov [gs:edi], ax
```

再对照图7.11和表7.3，是不是就全明白了？

如果你想实际看一下各种颜色的效果，可以通过调用disp_color_str()并改变其参数去试一下就知道了。

显示不同颜色的字符，然后把屏幕搞得色彩缤纷，这有时显得有点小儿科，不过笔者的体会是，在刚开始学习的时候，这种简单色彩所带来的乐趣也很容易让人兴奋，从而得到满足和成就感并提高我们的学习兴趣。所以，如果你也在亲手实践的话，尽管把屏幕搞得花花绿绿吧！

好，我们已经知道一个屏幕可以显示几行几列，又知道了一个字符占用几个字节，那么，一个屏幕映射到显存中所占的空间大小就很容易计算了：

$$80 \times 25 \times 2 = 4000\text{字节}$$

刚才我们讲过，显存有32KB，每个屏幕才占4KB，所以显存中足以存放8

个屏幕的数据。这就好办了，如果我们有 3 个 TTY，分别让它们使用 10KB 的空间还有剩余。而且在每一个 TTY 内，还可以实现简单的滚屏功能。

那么，如何能够让系统显示指定位置的内容呢？其实很简单，通过端口操作设置相应的寄存器就可以了。我们马上就来介绍。

在继续之前要说明的一点是，本书假定系统使用的是 VGA 以上的视频子系统，并假定不使用单色模式。VGA 早在 1987 年就被推出了，所以这样的假设不会有什么问题。

7.2.3 寄存器

我们又要跟硬件打交道了。说起来，从 8259A 到 8042，再到现在的显示器，对硬件的操作也做了不少，写程序其实没什么新鲜内容，端口操作而已。不过每种硬件各不相同，我们不得不了解其具体细节。比如 VGA 系统，它有 6 组寄存器，如表7.4所示，我们先来一点感性认识。

表 7.4 VGA 寄存器

寄存器		读端口	写端口
General Registers	Miscellaneous Output Register	0x3CC	0x3C2
	Input Status Register 0	0x3C2	-
	Input Status Register 1	0x3DA	-
	Feature Control Register	0x3CA	0x3DA
	Video Subsystem Enable Register	0x3C3	
Sequencer Registers	Address Register	0x3C4	
	Data Registers	0x3C5	
CRT Controller Registers	Address Register	0x3D4	
	Data Registers	0x3D5	
Graphics Controller Registers	Address Register	0x3CE	
	Data Registers	0x3CF	
Attribute Controller Registers	Address Register	0x3C0	
	Data Registers	0x3C1	0x3C0
Video DAC Palette Registers	Write Address	0x3C8	
	Read Address	-	0x3C7
	DAC State	0x3C7	-
	Data	0x3C9	
	Pel Mask	0x3C6	-

从这个表格看出，寄存器还真的是不少，而且有些寄存器读和写的端口是不同的。寄存器名称全部使用的是英文，这样做不但避免了翻译的偏差，而且有个好处是，我们可以通过 Register 这个单词是否使用复数来判断寄存器是否只有一个。比如 CRT Controller Registers 这一组，其中的 Data Registers 使用的是复数，说明数据寄存器不止一个，如表7.5所示。

这么多寄存器，只有一个端口0x3D5，怎么来操作其中某一个呢？这就用到 Address Register 了。我们看到表7.5中每一个寄存器都对应一个索引值，当

表 7.5 CRT Controller Data Registers

寄存器名称	索引	寄存器名称	索引
Horizontal Total Register	00h	Start Address Low Register	0Dh
End Horizontal Display Register	01h	Cursor Location High Register	0Eh
Start Horizontal Blanking Register	02h	Cursor Location Low Register	0Fh
End Horizontal Blanking Register	03h	Vertical Retrace Start Register	10h
Start Horizontal Retrace Register	04h	Vertical Retrace End Register	11h
End Horizontal Retrace Register	05h	Vertical Display End Register	12h
Vertical Total Register	06h	Offset Register	13h
Overflow Register	07h	Underline Location Register	14h
Preset Row Scan Register	08h	Start Vertical Blanking Register	15h
Maximum Scan Line Register	09h	End Vertical Blanking	16h
Cursor Start Register	0Ah	CRTC Mode Control Register	17h
Cursor End Register	0Bh	Line Compare Register	18h
Start Address High Register	0Ch		

想要访问其中一个时，只需要先向 Address Register 写对应的索引值（通过端口0x3D4），然后再通过端口0x3D5进行的操作就是针对索引值对应的寄存器了。如果我们把 Data Registers 看作一个数组，那么 Address Register 就相当于数组的下标。

举个例子，假如想把索引号为idx的寄存器的值改为new_value，可以这样来做：

```
out_byte(0x3D4, idx);
out_byte(0x3D5, new_value);
```

可以看到，只是多了一次端口操作而已。

我们马上来试一下。从字面意思可以知道，Cursor Location High Register 和 Cursor Location Low Register 是用来设置光标位置的，索引号分别是0Eh和0Fh。很久以来我们都没有理会光标位置这个问题，自从 Loader 中调用 BIOS 中断显示完第 5 行的Ready后，它就一直停在那里。现在我们就来修改一下程序，让它跟随我们敲入的每一个字符（代码7.19）。

代码 7.19 设置光标位置（chapter7/h/kernel/tty.c）

```
39
⇒ 40        disable_int();
⇒ 41        out_byte(CRTC_ADDR_REG, CURSOR_H);
⇒ 42        out_byte(CRTC_DATA_REG, ((disp_pos/2)>>8)&0xFF);
⇒ 43        out_byte(CRTC_ADDR_REG, CURSOR_L);
   44       out_byte(CRTC_DATA_REG, (disp_pos/2)&0xFF);
            enable_int();
```

其中，几个宏的定义如代码7.20所示（其中还定义了我们之后会用到的另外两个寄存器的索引值）。

代码 7.20 CRTC 相关的宏（chapter7/h/include/const.h）

```
72  #define CRTC_ADDR_REG   0x3D4   /* CRT Controller Registers - Addr Register */
```

```
#define CRTC_DATA_REG   0x3D5   /* CRT Controller Registers - Data Register */
#define START_ADDR_H    0xC     /* reg index of video mem start addr (MSB) */
#define START_ADDR_L    0xD     /* reg index of video mem start addr (LSB) */
#define CURSOR_H        0xE     /* reg index of cursor position (MSB) */
#define CURSOR_L        0xF     /* reg index of cursor position (LSB) */
#define V_MEM_BASE      0xB8000 /* base of color video memory */
#define V_MEM_SIZE      0x8000  /* 32K: B8000H -> BFFFFH */
```

之所以disp_pos被2除，是因为屏幕上每个字符对应2字节。

好了，make并运行，效果如图7.12所示。可以看到，光标开始跟随字符了。

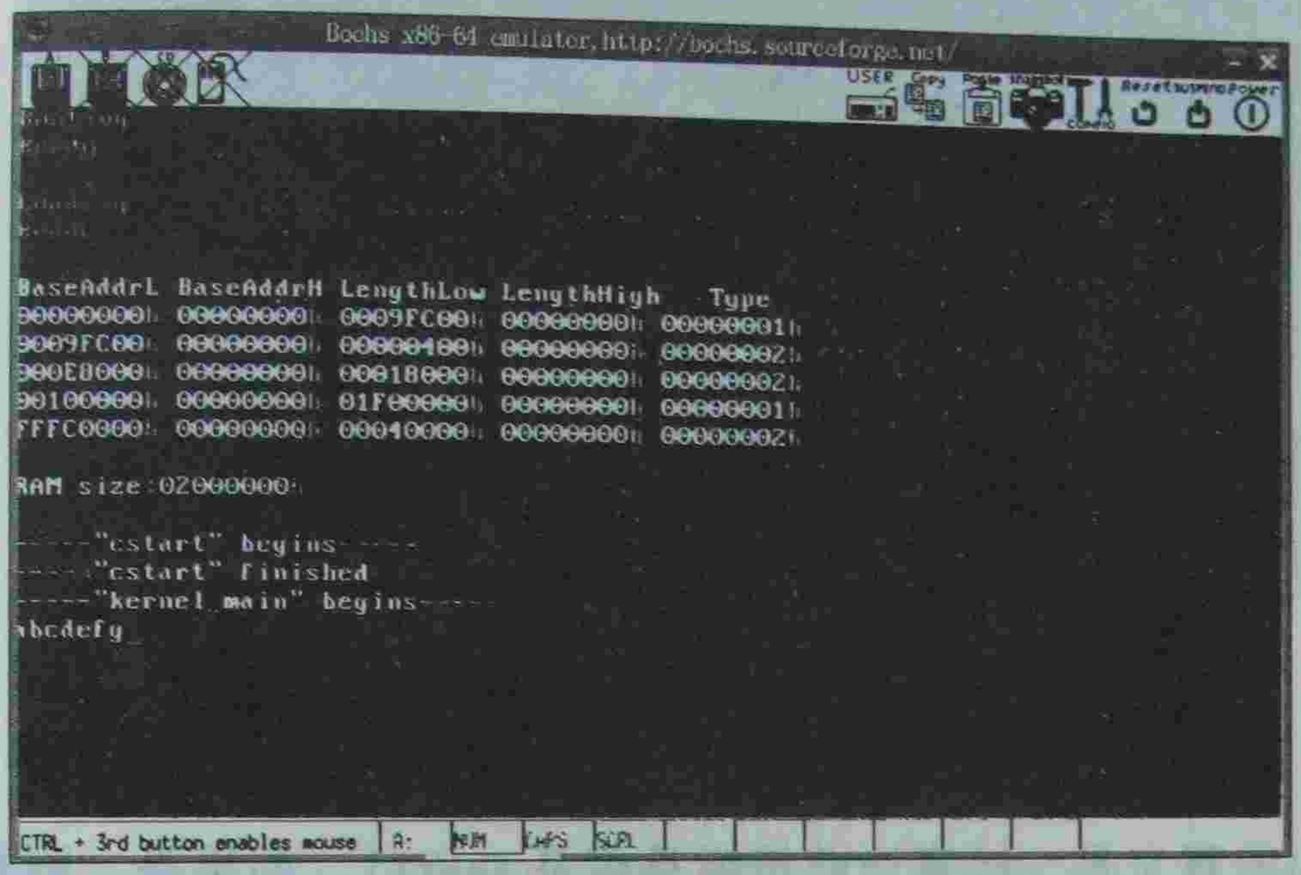


图7.12 光标跟随字符

我们不妨乘胜追击，进一步做试验。通过设置Start Address High Register和Start Address Low Register来重新设置显示开始地址，从而实现滚屏的功能。看看代码7.21和图7.13就知道了。

代码7.21 重新设置显示开始地址（chapter7/h/kernel/tty.c）

```
PUBLIC void in_process(u32 key)
{
        char output[2] = {'\0', '\0'};

        if (!(key & FLAG_EXT)) {
...
        else {
                int raw_code = key & MASK_RAW;
                switch(raw_code) {
                case UP:
                        if ((key & FLAG_SHIFT_L) || (key & FLAG_SHIFT_R)) {
                                disable_int();
                                out_byte(CRTC_ADDR_REG, START_ADDR_H);
                                out_byte(CRTC_DATA_REG, ((80*15) >> 8) & 0xFF);
                                out_byte(CRTC_ADDR_REG, START_ADDR_L);
                                out_byte(CRTC_DATA_REG, (80*15) & 0xFF);
                                enable_int();

                        }
                        break;
                case DOWN:
                        if ((key & FLAG_SHIFT_L) || (key & FLAG_SHIFT_R)) {
                                /* Shift+Down, do nothing */
```

```
62                          }
63                          break;
64                  default:
65                          break;
66                  }
67          }
68  }
```

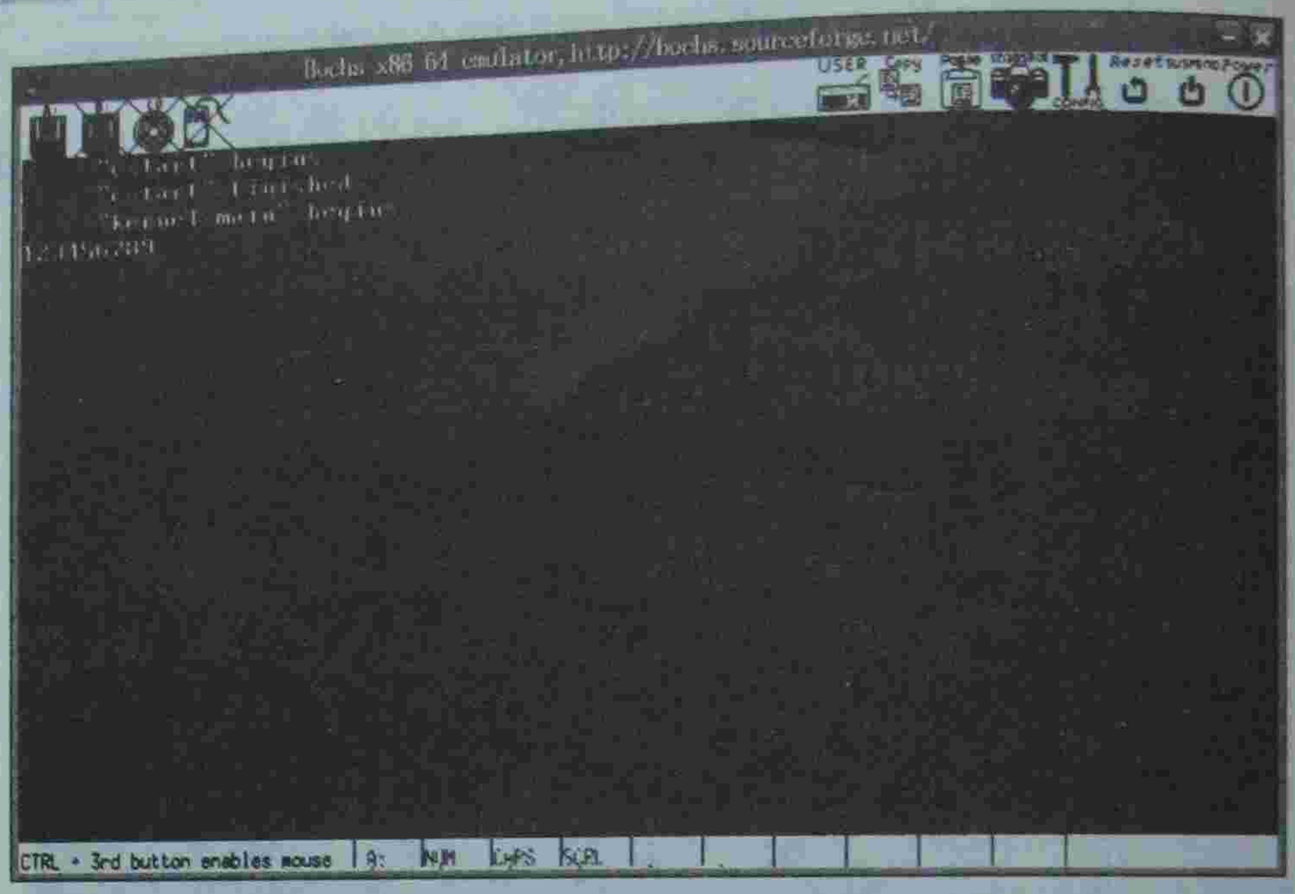


图 7.13 显示开始地址被重新设置

代码7.21的意义是，检查当前按键是否是 Shift+↑，如果是，则卷动屏幕至80×15处，即向上卷动 15 行。让我们来试一下，运行，按 Shift+↑，果然呈现出图7.13的样子。也就是说，Start Address High Register 和 Start Address Low Register 两个寄存器可以用来设置从显存的某个位置开始显示。这个特性允许我们把显存划分成不同的部分，然后只需要简单的寄存器设置就可以显示相应位置的内容。

我们已经通过改变 VGA 寄存器的值实现了光标的移动和屏幕滚动。不过，看到表7.4和表7.5中密密麻麻的寄存器名称，心中还是有点发怵，寄存器太多了！不要担心，寄存器虽多，我们暂时用到的却没有多少。反正我们已经了解了它们的访问方式，等到需要某个功能时，查一下手册就可以了。

7.3 TTY 任务

现在，对键盘和显示器的操作我们都已经了解了，那么，实现多个 TTY 仅仅是设计的问题了。我们可以让 TTY 任务以图7.14的形式运行。

在 TTY 任务中执行一个循环，这个循环将轮询每一个 TTY，处理它的事件，包括从键盘缓冲区读取数据、显示字符等内容。

需要说明如下几点：

1. 并非每轮询到某个 TTY 时，箭头所对应的全部事件都会发生，只有当某个 TTY 对应的控制台是当前控制台时，它才可以读取键盘缓冲区（所以图中读取过程使用了虚线）。

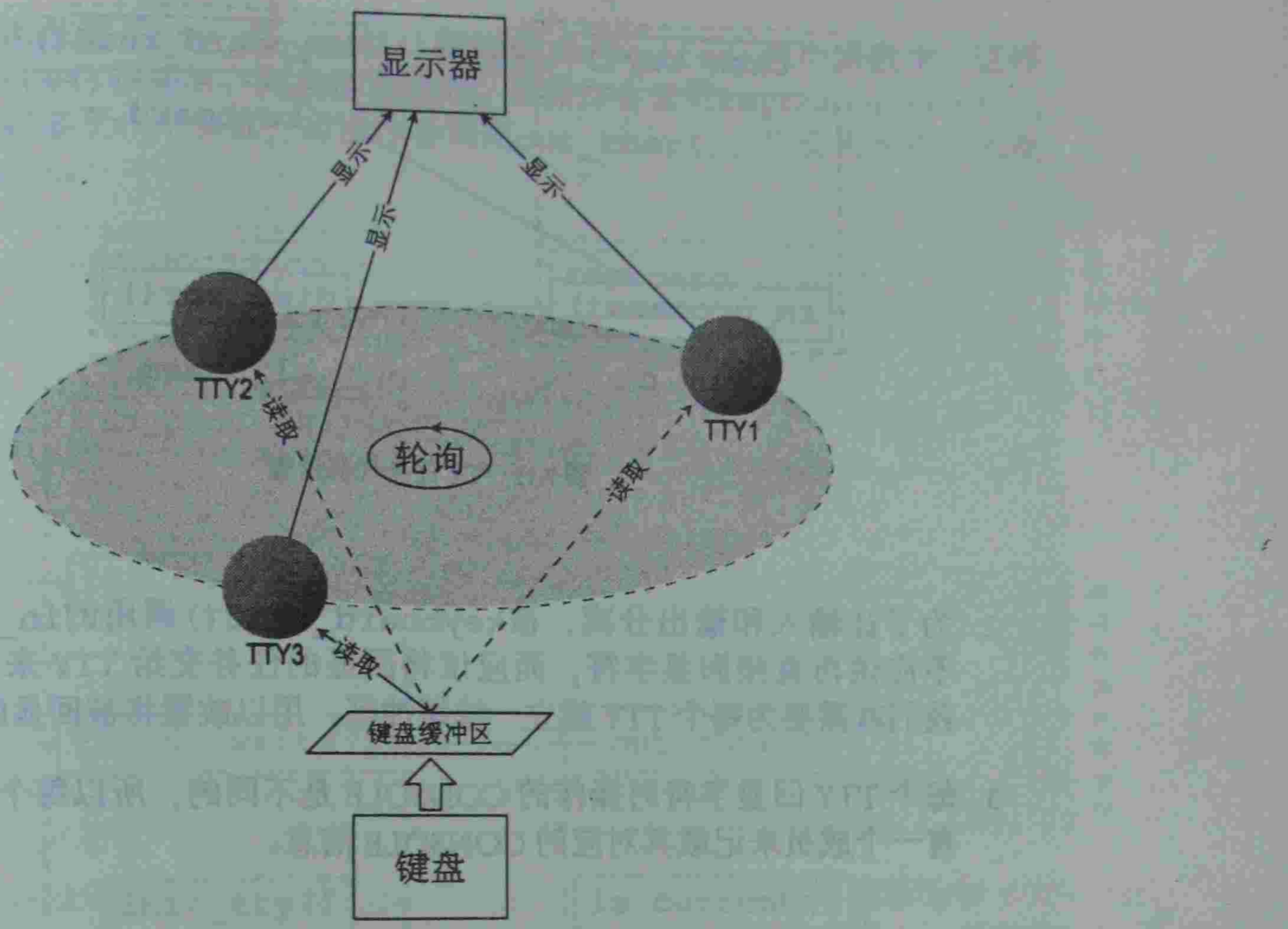


图 7.14 TTY 任务示意图

2. TTY 可以对输入的数据做更多的处理，但在这里，我们只把它简化为“显示”一项。

3. 虽然在图中，键盘和显示器的图示画在了 TTY 的外面，但正如图7.10所要表达的，我们应该把键盘和显示器算做每一个 TTY 的一部分，它们是公用的。

运行的过程已经清楚了，其实轮询到每一个 TTY 时不外乎做两件事：

1. 处理输入——查看是不是当前 TTY，如果是则从键盘缓冲区读取数据。

2. 处理输出——如果有要显示的内容则显示它。

在前面的程序中，TTY 任务是很简单的。如图7.15所示，箭头指的是函数间的调用关系。`task_tty()`是一个循环，它不断调用`keyboard_read()`，而`keyboard_read()`从键盘缓冲区得到数据后会调用`in_process()`，将字符直接显示出来。

我们下面要做的工作不能再这么简单了，它与原先程序实现的区别主要表现在如下几个方面：

1. 每一个 TTY 都应该有自己的读和写的动作。所以在`keyboard_read()`内部，函数需要了解自己是被哪一个 TTY 调用。我们通过为函数传入一个参数来做到这一点，这个参数是指向当前 TTY 的指针。

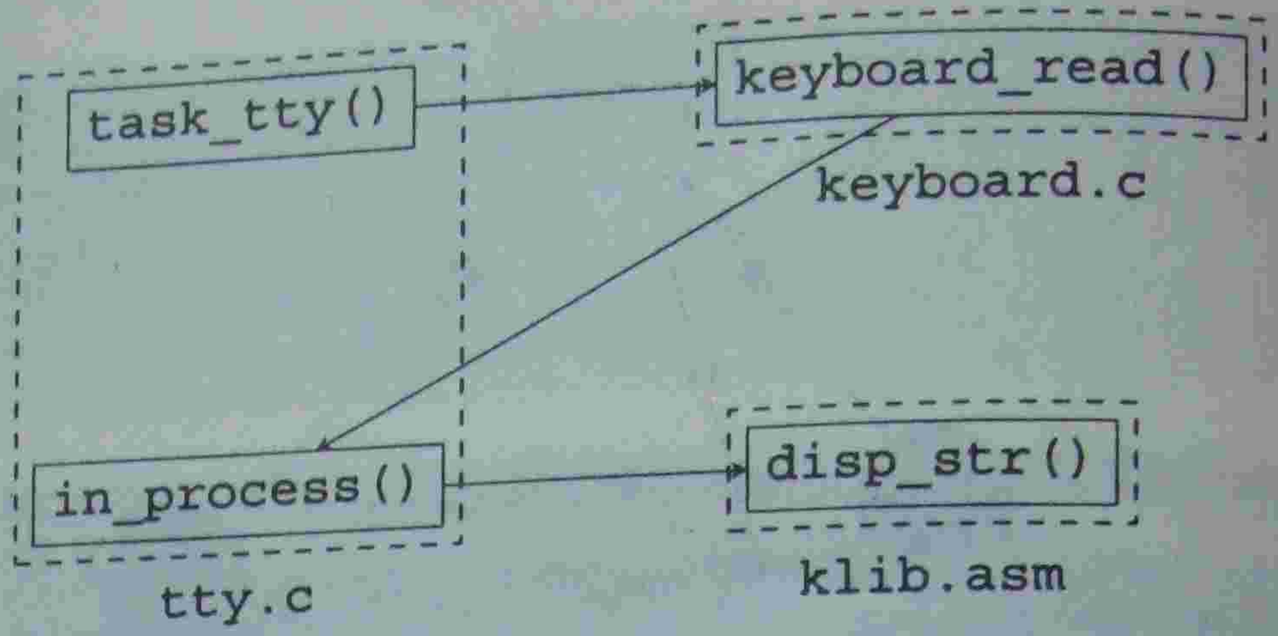


图7.15 TTY任务代码示意

2. 为了让输入和输出分离，被keyboard_read()调用的in_process()不应该再直接回显字符，而应该将回显的任务交给TTY来完成，这样，我们就需要为每个TTY建立一块缓冲区，用以放置将被回显的字符。

3. 每个TTY回显字符时操作的CONSOLE是不同的，所以每个TTY都应该有一个成员来记载其对应的CONSOLE信息。

7.3.1 TTY任务框架的搭建

基于上面的考虑，我们新建两个结构体，分别表示TTY和CONSOLE，见代码7.22和代码7.23。

代码7.22 TTY结构（chapter7/i/include/tty.h）

```
#define TTY_IN_BYTES    256     /* tty input queue size */

struct s_console;

/* TTY */
typedef struct s_tty
{
        u32     in_buf[TTY_IN_BYTES];   /* TTY 输入缓冲区 */
        u32*    p_inbuf_head;           /* 指向缓冲区中下一个空闲位置 */
        u32*    p_inbuf_tail;           /* 指向键盘任务应处理的键值 */
        int     inbuf_count;            /* 缓冲区中已经填充了多少 */

        struct s_console *      p_console;
}TTY;
```

代码7.23 CONSOLE结构（chapter7/i/include/console.h）

```
typedef struct s_console
{
        unsigned int    current_start_addr;     /* 当前显示到了什么位置     */
        unsigned int    original_addr;          /* 当前控制台对应显存位置 */
        unsigned int    v_mem_limit;            /* 当前控制台占的显存大小 */
        unsigned int    cursor;                 /* 当前光标位置 */
}CONSOLE;
```

由于下面要添加的内容有一点多，我们需要先来看一下整个的程序流程，如图7.16所示。在task_tty()中，通过循环来处理每一个TTY的读和写操作，

读写操作全都放在了tty_do_read()和tty_do_write()两个函数中，这样就让task_tty()很简洁，而且逻辑清晰。读操作会调用keyboard_read()，当然此时已经多了一个参数；写操作会调用out_char()，它会将字符写入指定的 CONSOLE。

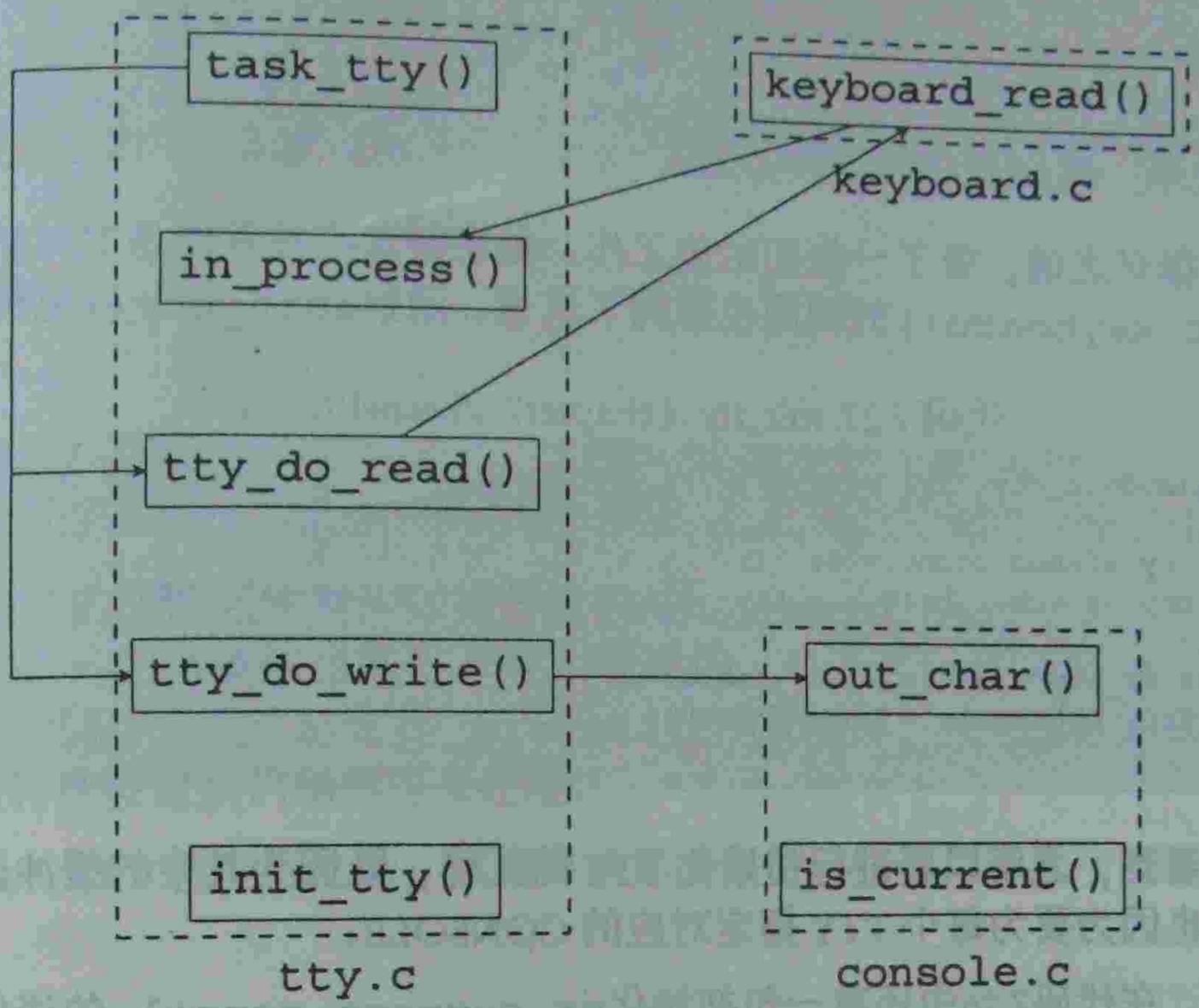


图 7.16 TTY 任务代码示意

对照图7.16，具体实现就变得容易了。之前分析过，32KB 的显存同时存在 3 个控制台是允许的，那么我们就先声明 3 个 TTY 以及对应的 3 个 CONSOLE（见代码7.24和代码7.25）。

代码 7.24 控制台个数（也是终端个数，chapter7/i/include/const.h）

```
47  /* TTY */
48  #define NR_CONSOLES     3       /* consoles */
```

代码 7.25 TTY 和 CONSOLE（chapter7/i/kernel/global.c）

```
29  PUBLIC  TTY             tty_table[NR_CONSOLES];
30  PUBLIC  CONSOLE         console_table[NR_CONSOLES];
```

下面来看一下框架性的task_tty()（代码7.26）。

代码 7.26 task_tty（chapter7/i/kernel/tty.c）

```
19  #define TTY_FIRST       (tty_table)
20  #define TTY_END         (tty_table + NR_CONSOLES)
    ...
29  PUBLIC void task_tty()
30  {
31
32          TTY*    p_tty;
33
34          init_keyboard();
35
            for (p_tty=TTY_FIRST;p_tty<TTY_END;p_tty++) {
```

```
                init_tty(p_tty);
        }
        nr_current_console = 0;
        while (1) {
                for (p_tty=TTY_FIRST;p_tty<TTY_END;p_tty++) {
                        tty_do_read(p_tty);
                        tty_do_write(p_tty);
                }
        }
}
```

在主循环之前，做了一些初始化工作。由于键盘应被看作是TTY的一部分，所以init_keyboard()的调用也挪到了这里。函数init_tty()见代码7.27。

代码7.27 init_tty（chapter7/i/kernel/tty.c）

```
PRIVATE void init_tty(TTY* p_tty)
{
        p_tty->inbuf_count = 0;
        p_tty->p_inbuf_head = p_tty->p_inbuf_tail = p_tty->in_buf;

        int nr_tty = p_tty - tty_table;
        p_tty->p_console = console_table + nr_tty;
}
```

可以看到，之所以要进行初始化TTY的工作，既因为其中的缓冲区需要设置初值，也因为要为每个TTY指定对应的CONSOLE。

另外，在代码7.26中还有一句初始化nr_current_console的语句，这是一个全局变量，定义在global.h中：

```
EXTERN int nr_current_console;
```

从字面意思可以知道，这个变量用来记录当前的控制台是哪一个，只有当某个TTY对应的控制台是当前控制台时，它才可以读取键盘缓冲区。所以，在tty_do_read()中要判断这个变量的值，进行控制台切换时也要记得改变它。

判断是否为当前控制台的代码见代码7.28。源文件console.c是新建立的。

代码7.28 判断是否为当前控制台（chapter7/i/kernel/console.c）

```
PUBLIC int is_current_console(CONSOLE* p_con)
{
        return (p_con == &console_table[nr_current_console]);
}
```

这样，tty_do_read()就很容易写了（代码7.29）。

代码7.29 tty_do_read（chapter7/i/kernel/tty.c）

```
PRIVATE void tty_do_read(TTY* p_tty)
{
        if (is_current_console(p_tty->p_console)) {
                keyboard_read(p_tty);
        }
}
```

注意，keyboard_read()发生了改变，要对其函数体做相应修改，同时，in_process()也要增加一个参数（代码7.30）。

代码 7.30 in_process（chapter7/i/kernel/tty.c）

```
62  PUBLIC void in_process(TTY* p_tty, u32 key)
63  {
64          char output[2] = {'\0', '\0'};
65
66          if (!(key & FLAG_EXT)) {
67                  if (p_tty->inbuf_count < TTY_IN_BYTES) {
68                          *(p_tty->p_inbuf_head) = key;
69                          p_tty->p_inbuf_head++;
70                          if (p_tty->p_inbuf_head == p_tty->in_buf + TTY_IN_BYTES) {
71                                  p_tty->p_inbuf_head = p_tty->in_buf;
72                          }
73                          p_tty->inbuf_count++;
74                  }
75          }
76          else {
...
97          }
98  }
```

往 TTY 的缓冲区中写入数据的代码与keyboard_handler()中的代码差不多。我们只把需要输出的字符写入缓冲区，如果遇到诸如 Alt+Fn 这样的切换控制台的操作，就让它在in_process()中处理掉，我想这样也是符合逻辑的。不过特殊按键我们稍侯再来处理。

在写入 TTY 缓冲区之后，读操作就算结束了。我们再来看一下写操作（代码7.31）。

代码 7.31 tty_do_write（chapter7/i/kernel/tty.c）

```
115 PRIVATE void tty_do_write(TTY* p_tty)
116 {
117         if (p_tty->inbuf_count) {
118                 char ch = *(p_tty->p_inbuf_tail);
119                 p_tty->p_inbuf_tail++;
120                 if (p_tty->p_inbuf_tail == p_tty->in_buf + TTY_IN_BYTES) {
121                         p_tty->p_inbuf_tail = p_tty->in_buf;
122                 }
123                 p_tty->inbuf_count--;
124
125                 out_char(p_tty->p_console, ch);
126         }
127 }
```

这段代码从 TTY 缓冲区中取出键值（类似get_byte_from_kbuf()），然后用out_char显示在对应的 CONSOLE 中。

我们暂时这样实现out_char()，见代码7.32。

代码 7.32 往控制台输出字符（chapter7/i/kernel/console.c）

```
36  /*======================================================================*
37                                 out_char
38   *======================================================================*/
39  PUBLIC void out_char(CONSOLE* p_con, char ch)
40  {
41          u8* p_vmem = (u8*)(V_MEM_BASE + disp_pos);
42
43          *p_vmem++ = ch;
44          *p_vmem++ = DEFAULT_CHAR_COLOR;
45          disp_pos += 2;
46
47          set_cursor(disp_pos/2);
```

```
}

/*======================================================================*
                           set_cursor
 *======================================================================*/
PRIVATE void set_cursor(unsigned int position)
{
        disable_int();
        out_byte(CRTC_ADDR_REG, CURSOR_H);
        out_byte(CRTC_DATA_REG, (position >> 8) & 0xFF);
        out_byte(CRTC_ADDR_REG, CURSOR_L);
        out_byte(CRTC_DATA_REG, position & 0xFF);
        enable_int();
}
```

V_MEM_BASE在const.h被定义为0xB8000，所以V_MEM_BASE+disp_pos就变成当前显示位置的地址。在这里，我们不再使用disp_str()来显示字符，而直接写字符到特定地址，这样做的前提是当前的ds指向的段的基址为0。

现在回顾一下我们上面所做的这些工作。当TTY任务开始运行时，所有TTY都将被初始化，并且全局变量nr_current_console会被赋值为0。然后循环开始并一直进行下去。对于每一个TTY，首先执行tty_do_read()，它将调用keyboard_read()并将读入的字符交给函数in_process()来处理，如果是需要输出的字符，会被in_process()放入当前接受处理的TTY的缓冲区中。然后tty_do_write()会接着执行，如果缓冲区中有数据，就被送入out_char显示出来。

由于nr_current_console初始化之后再没改变过，所以只是初始TTY在接受处理。其他TTY在做is_current_console(p_tty->p_console)这个判断后就被忽略掉了。所以，尽管在out_char()中将所有字符不加区分地顺序显示出来，但这是没有关系的。运行结果如图7.17所示。

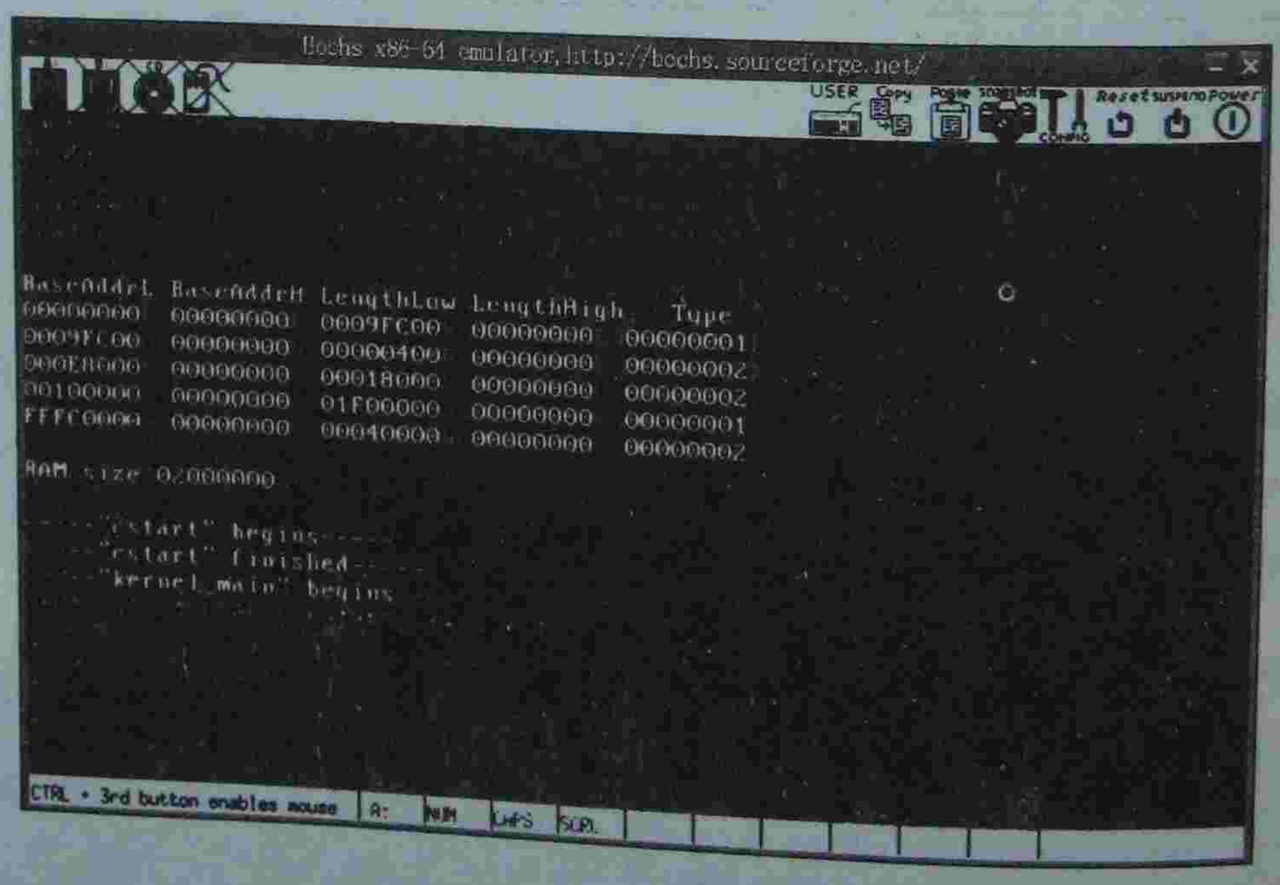


图7.17 显示开始地址被重新设置

一切良好！不过，框架虽然搭建起来了，我们仍只是在使用一个CONSOLE，下面我们就来实现多个CONSOLE。

7.3.2 多控制台

其实在写 TTY 和 CONSOLE 两个结构的时候已经为多控制台留下了足够的接口，只是我们还没实现它们而已，比如，CONSOLE 这个结构中的成员其实根本没有用到，具体如图7.18所示。

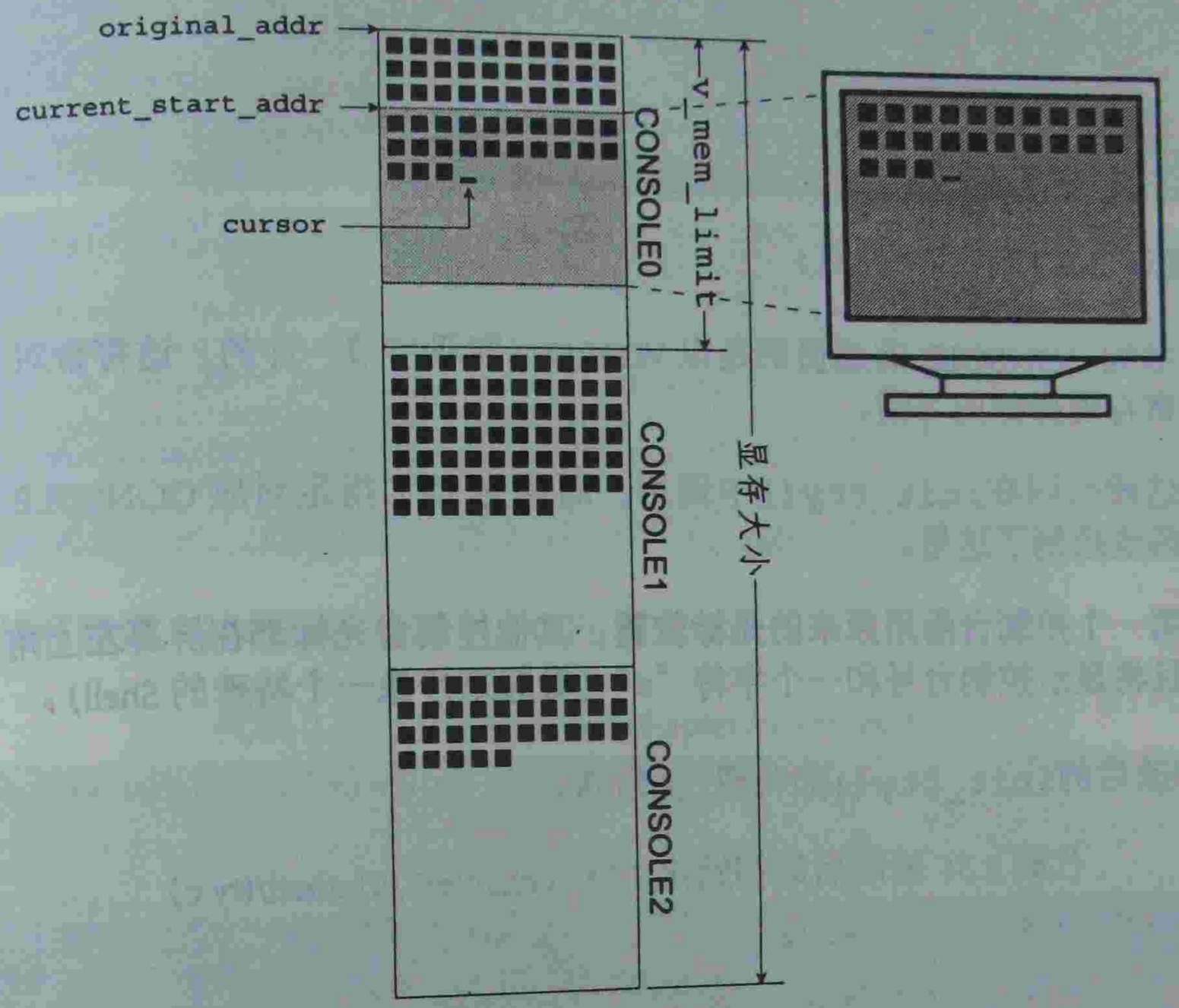


图 7.18 多控制台示意图

图7.18表示了某时刻显存的使用情况。其中灰色框表示当前屏幕，黑色小方格表示显存中已写入的字符。可以看出，original_addr和v_mem_limit用做定义控制台所占显存的总体情况，它们是静态的，一经初始化就不再改变；current_start_addr将随着屏幕卷动而变化，cursor变化更频繁，每输出一个字符就更新一次。

下面我们先来为这些成员设置初值（代码7.33）。

代码 7.33 往控制台输出字符（chapter7/j/kernel/console.c）

```
31  PUBLIC void init_screen(TTY* p_tty)
32  {
33      int nr_tty = p_tty - tty_table;
34      p_tty->p_console = console_table + nr_tty;
35
36      int v_mem_size = V_MEM_SIZE >> 1;       /* 显存总大小 (in WORD) */
37
38      int con_v_mem_size                   = v_mem_size / NR_CONSOLES;
39      p_tty->p_console->original_addr      = nr_tty * con_v_mem_size;
40      p_tty->p_console->v_mem_limit        = con_v_mem_size;
41      p_tty->p_console->current_start_addr = p_tty->p_console->original_addr;
42
43      /* 默认光标位置在最开始处 */
44      p_tty->p_console->cursor = p_tty->p_console->original_addr;
```

```
        if (nr_tty == 0) {
                /* 第一个控制台沿用原来的光标位置 */
                p_tty->p_console->cursor = disp_pos / 2;
                disp_pos = 0;
        }
        else {
                out_char(p_tty->p_console, nr_tty + '0');
                out_char(p_tty->p_console, '#');
        }

        set_cursor(p_tty->p_console->cursor);
}
```

值得注意的有如下几点：

1. 结构CONSOLE的成员都是以WORD（即双字节）计的，这符合对VGA寄存器操作的习惯。
2. 这段代码在init_tty()中调用，而且为TTY指定对应CONSOLE的代码也挪到了这里。
3. 第一个控制台沿用原来的光标位置，其他控制台光标都在屏幕左上角，并且将显示控制台号和一个字符“#”（看起来好像一个特殊的Shell）。

修改后的init_tty()如代码7.34所示。

代码7.34 修改后的初始化TTY（chapter7/j/kernel/tty.c）

```
PRIVATE void init_tty(TTY* p_tty)
{
        p_tty->inbuf_count = 0;
        p_tty->p_inbuf_head = p_tty->p_inbuf_tail = p_tty->in_buf;

        init_screen(p_tty);
}
```

原来的函数out_char()尚未考虑多控制台的情况，如今要改变一下了（代码7.35）。

代码7.35 修改后的out_char（chapter7/j/kernel/console.c）

```
PUBLIC void out_char(CONSOLE* p_con, char ch)
{
        u8* p_vmem = (u8*)(V_MEM_BASE + p_con->cursor * 2);

        *p_vmem++ = ch;
        *p_vmem++ = DEFAULT_CHAR_COLOR;
        p_con->cursor++;

        set_cursor(p_con->cursor);
}
```

为了能够看到效果，我们还需要一个切换控制台的函数（代码7.36）。

代码7.36 切换控制台的代码（chapter7/j/kernel/console.c）

```
PUBLIC void select_console(int nr_console)      /* 0 ~ (NR_CONSOLES - 1) */
{
        if ((nr_console < 0) || (nr_console >= NR_CONSOLES)) {
```

```
117             return;
118         }
119
120         nr_current_console = nr_console;
121
122         set_cursor(console_table[nr_console].cursor);
123         set_video_start_addr(console_table[nr_console].current_start_addr);
124 }
```

其中，函数set_video_start_addr()我们已经很熟悉了（代码7.37）。

代码 7.37 切换控制台（chapter7/j/kernel/console.c）

```
99  PRIVATE void set_video_start_addr(u32 addr)
100 {
101         disable_int();
102         out_byte(CRTC_ADDR_REG, START_ADDR_H);
103         out_byte(CRTC_DATA_REG, (addr >> 8) & 0xFF);
104         out_byte(CRTC_ADDR_REG, START_ADDR_L);
105         out_byte(CRTC_DATA_REG, addr & 0xFF);
106         enable_int();
107 }
```

按照惯例，我们应该在按下 Alt+Fn 时做切换工作（代码7.38）。

代码 7.38 处理 Alt+Fn（chapter7/j/kernel/tty.c）

```
60  PUBLIC void in_process(TTY* p_tty, u32 key)
61  {
    ...
74          else {
    ...
92                  case F1:
93                  case F2:
94                  case F3:
95                  case F4:
96                  case F5:
97                  case F6:
98                  case F7:
99                  case F8:
100                 case F9:
101                 case F10:
102                 case F11:
103                 case F12:
104                     /* Alt + F1-F12 */
105                     if ((key & FLAG_ALT_L) || (key & FLAG_ALT_R)) {
106                         select_console(raw_code - F1);
107                     }
108                     break;
    ...
113 }
```

这样，我们就可以把原来tty_task()中直接将nr_current_console赋值为 0 的语句换成对select_console()的调用了（代码7.39）。

代码 7.39 选择第 0 个 console（chapter7/j/kernel/tty.c）

```
28  PUBLIC void task_tty()
29  {
    ...
37          select_console(0);
    ...
44  }
```

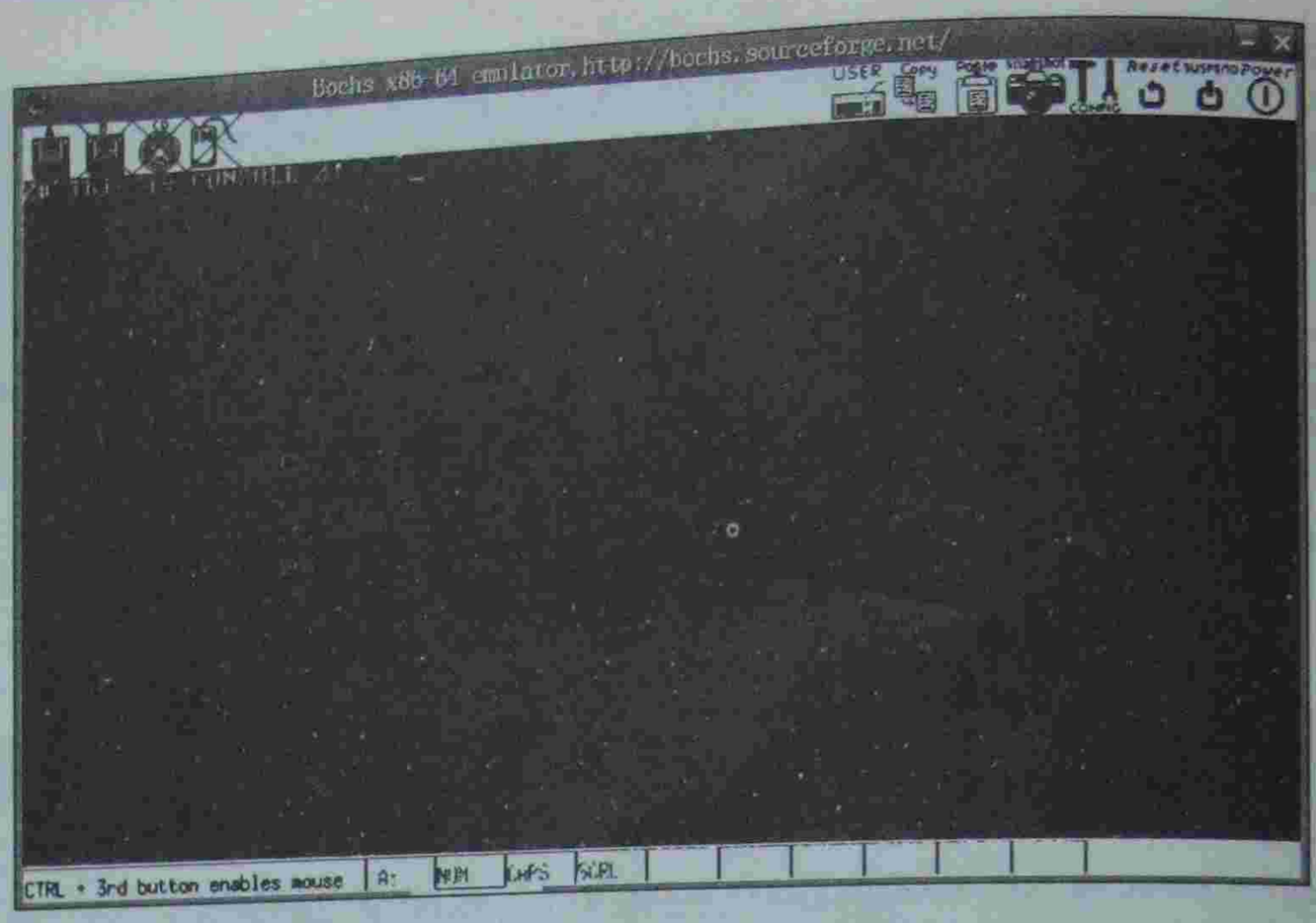


图7.19 显示开始地址被重新设置

这样，我们就可以运行一下看看了，结果如图7.19所示。

图7.19所示的画面是在控制台2（即通过Alt+F3切换到的控制台）中。其中“This is CONSOLE 2! ^-^”这行字是笔者打上去的。

怎么样，你是不是开始觉得我们的OS越来越好玩了呢？

如果你切换到控制台0的话，可能发现屏幕快满了。我们现在就来添加屏幕滚动的代码（代码7.40）。

代码7.40 scroll_screen（chapter7/k/kernel/console.c）

```
136 PUBLIC void scroll_screen(CONSOLE* p_con, int direction)
137 {
138         if (direction == SCR_UP) {
139                 if (p_con->current_start_addr > p_con->original_addr) {
140                         p_con->current_start_addr -= SCREEN_WIDTH;
141                 }
142         }
143         else if (direction == SCR_DN) {
144                 if (p_con->current_start_addr + SCREEN_SIZE <
145                     p_con->original_addr + p_con->v_mem_limit) {
146                         p_con->current_start_addr += SCREEN_WIDTH;
147                 }
148         }
149         else{
150         }
151
152         set_video_start_addr(p_con->current_start_addr);
153         set_cursor(p_con->cursor);
154 }
```

为了简化程序，当屏幕滚动到最下端后，再试图向下滚动时按键将不再响应，最上端时也是这样。下面来看响应Shift+↑和Shift+↓的代码（代码7.41）。

代码7.41 响应Shift+↑和Shift+↓（chapter7/k/kernel/tty.c）

```
60 PUBLIC void in_process(TTY* p_tty, u32 key)
61 {
   ...
74         else {
```

```
                int raw_code = key & MASK_RAW;
                switch(raw_code) {
                case UP:
                        if ((key & FLAG_SHIFT_L) || (key & FLAG_SHIFT_R)) {
                                scroll_screen(p_tty->p_console, SCR_DN);
                        }
                        break;
                case DOWN:
                        if ((key & FLAG_SHIFT_L) || (key & FLAG_SHIFT_R)) {
                                scroll_screen(p_tty->p_console, SCR_UP);
                        }
                        break;
...
                }
```

好了，现在我们运行一下，在控制台0按Shift+↑数次，会呈现图7.20所示的情形。

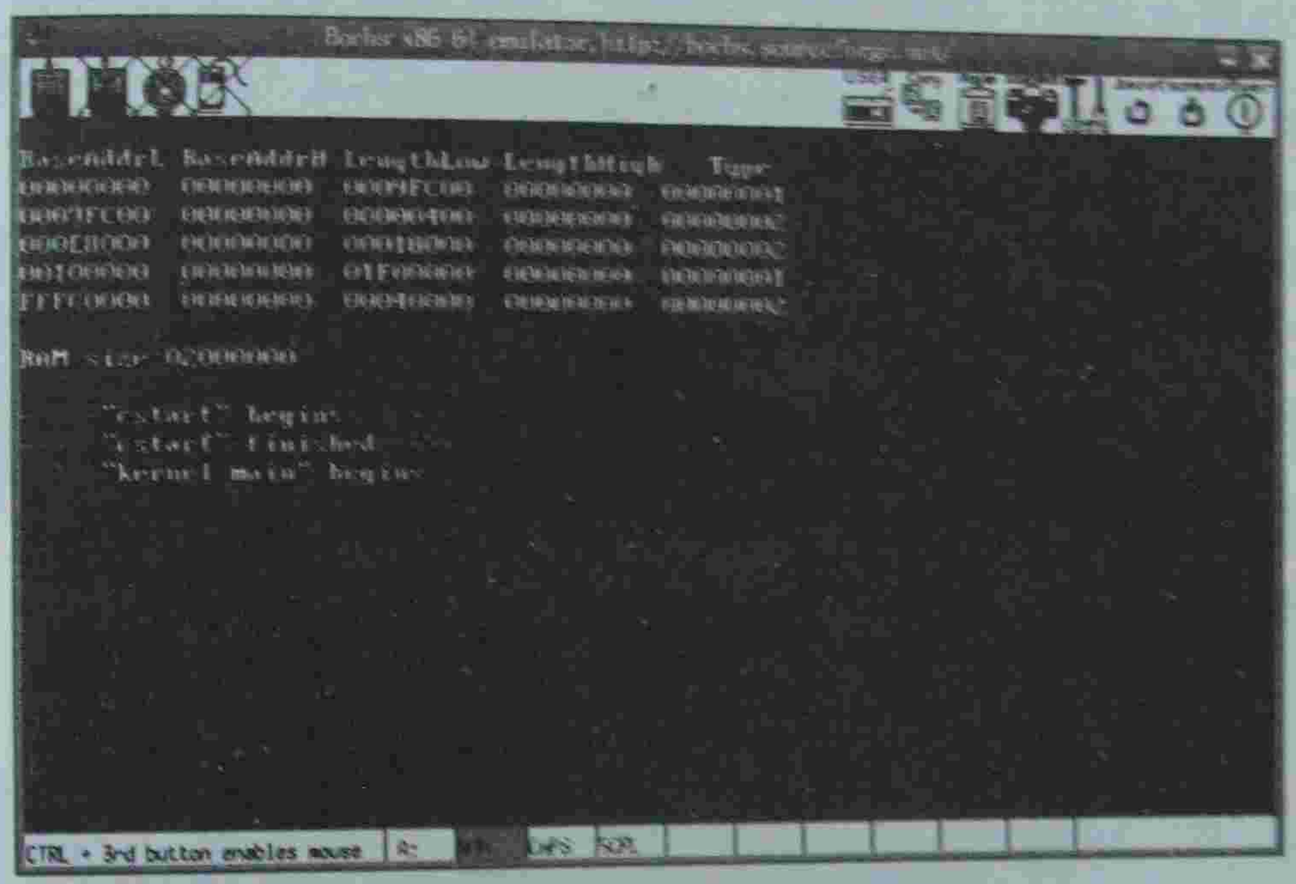


图7.20 显示开始地址被重新设置

到现在为止，多控制台已经被我们实现了。虽然涉及的函数稍微有些繁杂，但如果读者对照图7.16来看的话，会发现理解起来是很容易的。

7.3.3 完善键盘处理

在上面的运行过程中你可能已经发现，现在我们的系统对键盘的支持是很差的，比如你甚至不能使用CapsLock，更不用说BackSpace、小键盘等了。之所以迟迟不加入这些内容，完全是为了让我们的TTY任务最简单。如今，任务的框架已经搭建起来了，现在就可以添加处理其他按键的代码了。

7.3.3.1 回车键和退格键

当敲击回车键和退格键时，我们往TTY缓冲区中写入“\n”和“\b”，然后在out_char中做相应处理，请看代码7.42。

代码7.42 响应回车键和退格键（chapter7/l/kernel/tty.c）

```
PUBLIC void in_process(TTY* p_tty, u32 key)
{
```

```
64          char output[2] = {'\0', '\0'};
65
66          if (!(key & FLAG_EXT)) {
67  ⇒               put_key(p_tty, key);
68          }
69          else {
70                  int raw_code = key & MASK_RAW;
71                  switch(raw_code) {
72                  case ENTER:
73  ⇒                       put_key(p_tty, '\n');
74                          break;
75                  case BACKSPACE:
76  ⇒                       put_key(p_tty, '\b');
77                          break;
...
109 }
...
114 PRIVATE void put_key(TTY* p_tty, u32 key)
115 {
116         if (p_tty->inbuf_count < TTY_IN_BYTES) {
117                 *(p_tty->p_inbuf_head) = key;
118                 p_tty->p_inbuf_head++;
119                 if (p_tty->p_inbuf_head == p_tty->in_buf + TTY_IN_BYTES) {
120                         p_tty->p_inbuf_head = p_tty->in_buf;
121                 }
122                 p_tty->inbuf_count++;
123         }
124 }
```

然后修改out_char()（代码7.43）。

代码 7.43 修改 out_char（chapter7/l/kernel/console.c）

```
73  PUBLIC void out_char(CONSOLE* p_con, char ch)
74  {
75          u8* p_vmem = (u8*)(V_MEM_BASE + p_con->cursor * 2);
76
77          switch(ch) {
78          case '\n':
79                  if (p_con->cursor < p_con->original_addr +
80                      p_con->v_mem_limit - SCREEN_WIDTH) {
81                          p_con->cursor = p_con->original_addr + SCREEN_WIDTH *
82                                  ((p_con->cursor - p_con->original_addr) /
83                                   SCREEN_WIDTH + 1);
84                  }
85                  break;
86          case '\b':
87                  if (p_con->cursor > p_con->original_addr) {
88                          p_con->cursor--;
89                          *(p_vmem-2) = ' ';
90                          *(p_vmem-1) = DEFAULT_CHAR_COLOR;
91                  }
92                  break;
93          default:
94                  if (p_con->cursor <
95                      p_con->original_addr + p_con->v_mem_limit - 1) {
96                          *p_vmem++ = ch;
97                          *p_vmem++ = DEFAULT_CHAR_COLOR;
98                          p_con->cursor++;
99                  }
100                 break;
101         }
102
103         while (p_con->cursor >= p_con->current_start_addr + SCREEN_SIZE) {
104                 scroll_screen(p_con, SCR_DN);
105         }
106 }
```

```
107             flush(p_con);
108     }
        ...
113     PRIVATE void flush(CONSOLE* p_con)
114     {
115             set_cursor(p_con->cursor);
116             set_video_start_addr(p_con->current_start_addr);
117     }
```

可以看到，回车键直接把光标挪到了下一行的开头，而退格键则把光标挪到上一个字符的位置，并在那里写一个空格，以便清除原来的字符。

由于不断的回车会让光标快速地移动到屏幕底端，所以在这里还判断了光标是否已经移出了屏幕，如果是的话，将会触发屏幕滚动。

另外，输出任何类型的字符时，都做了边界检验，以防止影响到别的控制台，甚至试图写到显存之外的内存。

图7.21是控制台 1 中以下按键序列的结果：

键入Welcome Enter and BackSpace!－回车－回车－键入“a”－回车－回车－键入“b”－回车－回车－键入“c”－回车－回车－回车－键入“123456”－退格－退格－键入“789”。

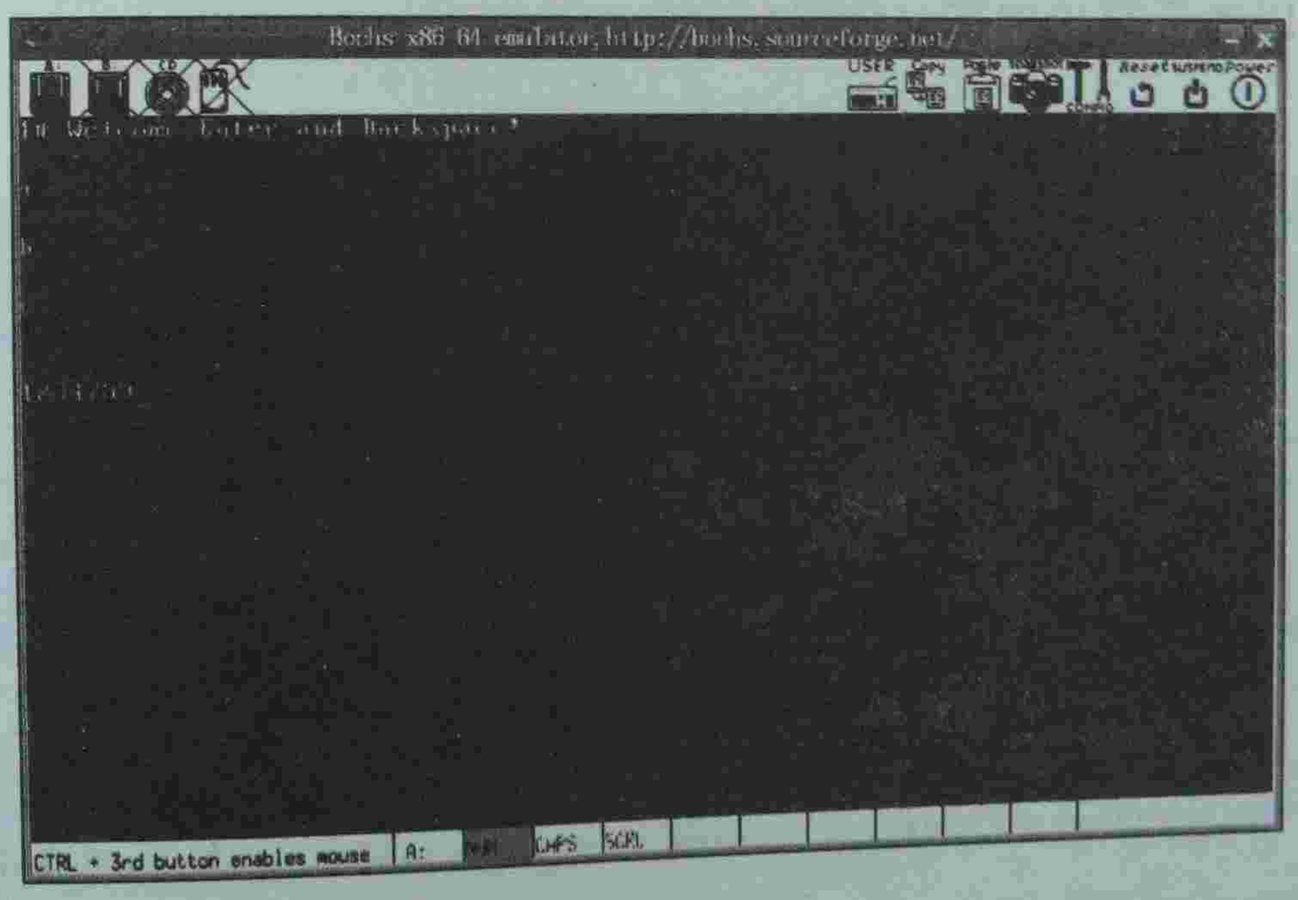


图 7.21 回车键和退格键可用

7.3.3.2 Caps Lock、Num Lock、Scroll Lock

键盘上这 3 个键有一点特殊，因为每一个都对应一个小灯（LED）。实际上，不但通过敲击键盘可以控制这些灯的亮灭，通过写入 8042 的输入缓冲区也可以做到这一点。这样，我们可以维持 3 个全局变量，用以表示 3 个灯的状态，在键盘初始化的时候给它们任意赋我们想要的初值，并同时设置灯的相应状态。

先来看看如何通过端口操作控制它们。从表7.1可以看到，输入缓冲区和控制寄存器都是可写的，但它们的作用是不同的，写入输入缓冲区用来往 8048 发送命令，而写入控制寄存器是往 8042 本身发送命令。

我们的目的是往 8048 发送命令，使用端口0x60。设置 LED 的命令是0xED。

当键盘接收到这个命令后，会回复一个 ACK（0xFA），然后等待从端口0x60写入的 LED 参数字节，这个参数字节定义如图7.22所示。

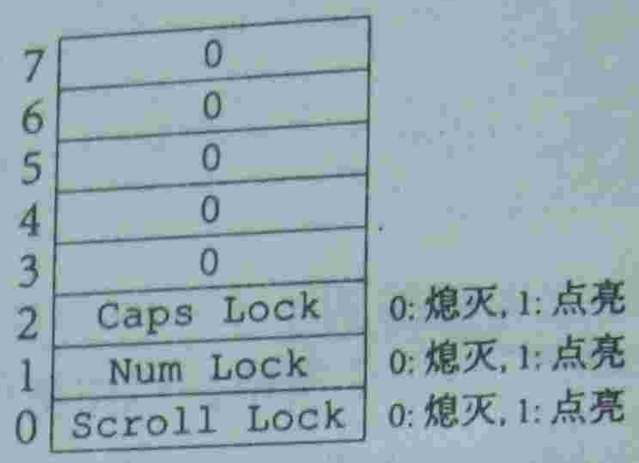


图 7.22 设置 LED 的参数字节

当键盘收到参数字节后，会再回复一个 ACK，并根据参数字节的值来设置 LED。

要注意的是，在向 8042 输入缓冲区写数据时，要先判断一下输入缓冲区是否为空，方法是通过端口0x64读取状态寄存器。状态寄存器的第 1 位如果为0，表示输入缓冲区是空的，可以向其写入数据。

现在可以动手写代码了（代码7.44）。

代码 7.44 设置 LED（chapter7/m/kernel/keyboard.c）

```
317 /*======================================================================*
318                                kb_wait
319  *======================================================================*/
320 PRIVATE void kb_wait()  /* 等待 8042 的输入缓冲区空 */
321 {
322         u8 kb_stat;
323
324         do {
325                 kb_stat = in_byte(KB_CMD);
326         } while (kb_stat & 0x02);
327 }
328
329
330 /*======================================================================*
331
332                                kb_ack
333  *======================================================================*/
334 PRIVATE void kb_ack()
335 {
336         u8 kb_read;
337
338         do {
339                 kb_read = in_byte(KB_DATA);
340         } while (kb_read =! KB_ACK);
341 }
342
343 /*======================================================================*
344
345                                set_leds
346  *======================================================================*/
347 PRIVATE void set_leds()
348 {
349         u8 leds = (caps_lock << 2) | (num_lock << 1) | scroll_lock;
350
351         kb_wait();
352         out_byte(KB_DATA, LED_CODE);
         kb_ack();
```

```
        kb_wait();
        out_byte(KB_DATA, leds);
        kb_ack();
}
```

其中的LED_CODE和KB_ACK分别被定义成0xED和0xFA（代码7.45）。

代码 7.45 LED_CODE 和 KB_ACK（chapter7/m/include/const.h）

```
#define LED_CODE        0xED
#define KB_ACK          0xFA
```

而caps_lock，num_lock和scroll_lock的声明和初始化如代码7.46所示。

代码 7.46 初始化 LEDs 的状态（chapter7/m/kernel/keyboard.c）

```
PRIVATE int     caps_lock;      /* Caps Lock     */
PRIVATE int     num_lock;       /* Num Lock      */
PRIVATE int     scroll_lock;    /* Scroll Lock   */
 ...
PUBLIC void init_keyboard()
{
        kb_in.count = 0;
        kb_in.p_head = kb_in.p_tail = kb_in.buf;

        shift_l = shift_r = 0;
        alt_l   = alt_r   = 0;
        ctrl_l  = ctrl_r  = 0;

        caps_lock   = 0;
        num_lock    = 1;
        scroll_lock = 0;

        set_leds();

        put_irq_handler(KEYBOARD_IRQ, keyboard_handler);/*设定键盘中断处理程序*/
        enable_irq(KEYBOARD_IRQ);                       /*开键盘中断*/
}
```

之所以把num_lock的初值设为1，是因为据笔者观察，好像大多数人使用小键盘的时候都是使用其数字功能而非箭头等功能。

现在运行程序，你会发现系统启动之后 NumLock 被点亮。

虽然灯亮了，但却还未起到作用，我们现在就来修改keyboard_read()（代码7.47）。

代码 7.47 使用 CapsLock 和小键盘（chapter7/m/kernel/keyboard.c）

```
PUBLIC void keyboard_read(TTY* p_tty)
{
 ...
                if ((key != PAUSEBREAK) && (key != PRINTSCREEN)) {
                        /* 首先判断Make Code 还是 Break Code */
                        make = (scan_code & FLAG_BREAK ? 0 : 1);

                        /* 先定位到 keymap 中的行 */
                        keyrow = &keymap[(scan_code & 0x7F) * MAP_COLS];

                        column = 0;

                        int caps = shift_l || shift_r;
```

```
⇒ 154                   if (caps_lock) {
⇒ 155                           if ((keyrow[0] >= 'a') && (keyrow[0] <= 'z')){
⇒ 156                                   caps = !caps;
⇒ 157                           }
⇒ 158                   }
⇒ 159                   if (caps) {
⇒ 160                           column = 1;
⇒ 161                   }
    162
⇒ 163                   if (code_with_E0) {
⇒ 164                           column = 2;
⇒ 165                   }
    166
    167                   key = keyrow[column];
    168
    169                   switch(key) {
    170                   case SHIFT_L:
    171                           shift_l = make;
    172                           break;
    173                   case SHIFT_R:
    174                           shift_r = make;
    175                           break;
    176                   case CTRL_L:
    177                           ctrl_l = make;
    178                           break;
    179                   case CTRL_R:
    180                           ctrl_r = make;
    181                           break;
    182                   case ALT_L:
    183                           alt_l = make;
    184                           break;
    185                   case ALT_R:
    186                           alt_l = make;
    187                           break;
⇒ 188                   case CAPS_LOCK:
⇒ 189                           if (make) {
⇒ 190                                   caps_lock   = !caps_lock;
⇒ 191                                   set_leds();
⇒ 192                           }
⇒ 193                           break;
⇒ 194                   case NUM_LOCK:
⇒ 195                           if (make) {
⇒ 196                                   num_lock    = !num_lock;
⇒ 197                                   set_leds();
⇒ 198                           }
⇒ 199                           break;
⇒ 200                   case SCROLL_LOCK:
⇒ 201                           if (make) {
⇒ 202                                   scroll_lock = !scroll_lock;
⇒ 203                                   set_leds();
⇒ 204                           }
⇒ 205                           break;
    206                   default:
    207                           break;
    208                   }
    209
⇒ 210                   if (make) { /* 忽略 Break Code */
⇒ 211                           int pad = 0;
⇒ 212
⇒ 213                           /* 首先处理小键盘 */
⇒ 214                           if ((key >= PAD_SLASH) && (key <= PAD_9)) {
⇒ 215                                   pad = 1;
⇒ 216                                   switch(key) {
⇒ 217                                   case PAD_SLASH:
⇒ 218                                           key = '/';
⇒ 219                                           break;
⇒ 220                                   case PAD_STAR:
```

```
                key = '*';
                break;
            case PAD_MINUS:
                key = '-';
                break;
            case PAD_PLUS:
                key = '+';
                break;
            case PAD_ENTER:
                key = ENTER;
                break;
            default:
                if (num_lock &&
                    (key >= PAD_0) &&
                    (key <= PAD_9)) {
                    key = key - PAD_0 + '0';
                }
                else if (num_lock &&
                        (key == PAD_DOT)) {
                    key = '.';
                }
                else{
                        switch(key) {
                        case PAD_HOME:
                                key = HOME;
                                break;
                        case PAD_END:
                                key = END;
                                break;
                        case PAD_PAGEUP:
                                key = PAGEUP;
                                break;
                        case PAD_PAGEDOWN:
                                key = PAGEDOWN;
                                break;
                        case PAD_INS:
                                key = INSERT;
                                break;
                        case PAD_UP:
                                key = UP;
                                break;
                        case PAD_DOWN:
                                key = DOWN;
                                break;
                        case PAD_LEFT:
                                key = LEFT;
                                break;
                        case PAD_RIGHT:
                                key = RIGHT;
                                break;
                        case PAD_DOT:
                                key = DELETE;
                                break;
                        default:
                                break;
                        }
                }
                break;
        }
    }

    key |= shift_l  ? FLAG_SHIFT_L  : 0;
    key |= shift_r  ? FLAG_SHIFT_R  : 0;
    key |= ctrl_l   ? FLAG_CTRL_L   : 0;
    key |= ctrl_r   ? FLAG_CTRL_R   : 0;
    key |= alt_l    ? FLAG_ALT_L    : 0;
    key |= alt_r    ? FLAG_ALT_R    : 0;
```

```
                                key |= pad     ? FLAG_PAD     : 0;
⇒ 288
289                             in_process(p_tty, key);
290                         }
291                     }
292                 }
293             }
294 }
```

代码虽长，但很容易理解。唯一要提的是，这里增加了一个pad变量，并在key中设置了一个相应的位。这样做是考虑将来可能需要区分普通的数字键和小键盘上的数字键。

好了，现在运行一下，你会发现 CapsLock 已经开始起作用了，小键盘也能用了。如图7.23所示。

图 7.23 改进后的键盘处理演示

图中的字符是在点亮 CapsLock 之后键入的，数字键以及 /、* 等是用小键盘键入的。

7.3.4 TTY 任务总结

至此，我们的 TTY 任务可以暂告一段落了。它的优点是足够小巧，而且结构很清晰，很易懂，加上我们上面对于细节的介绍以及辅助的图表，理解起来应该是很容易的。

不过有个问题，终端任务虽然运行在 ring1，但我们可以看到，它与运行在 ring0 的keyboard_handler有些混淆，终端任务可以访问所有内存。kb_in 这个变量在 ring0 下写，在 ring1 下读。不过 Minix 也是这样来做的，这是向它学习的结果。

7.4 区分任务和用户进程

现在，我们有了4个进程，分别是TTY、A、B、C。其中A、B和C是可有可无的，它其实不是操作系统的一部分，而更像用户在执行的程序。而TTY则不同，它肩负着重大的职责，没有它我们连键盘都无法使用。

所以，我们有必要把它们区分开来，分为两类。我们称TTY为“任务”，而称A、B、C为“用户进程”。

在具体的实现上，也来做一些相应的改变，让用户进程运行在ring3，任务继续留在ring1。这样就形成了图7.24所示的情形。

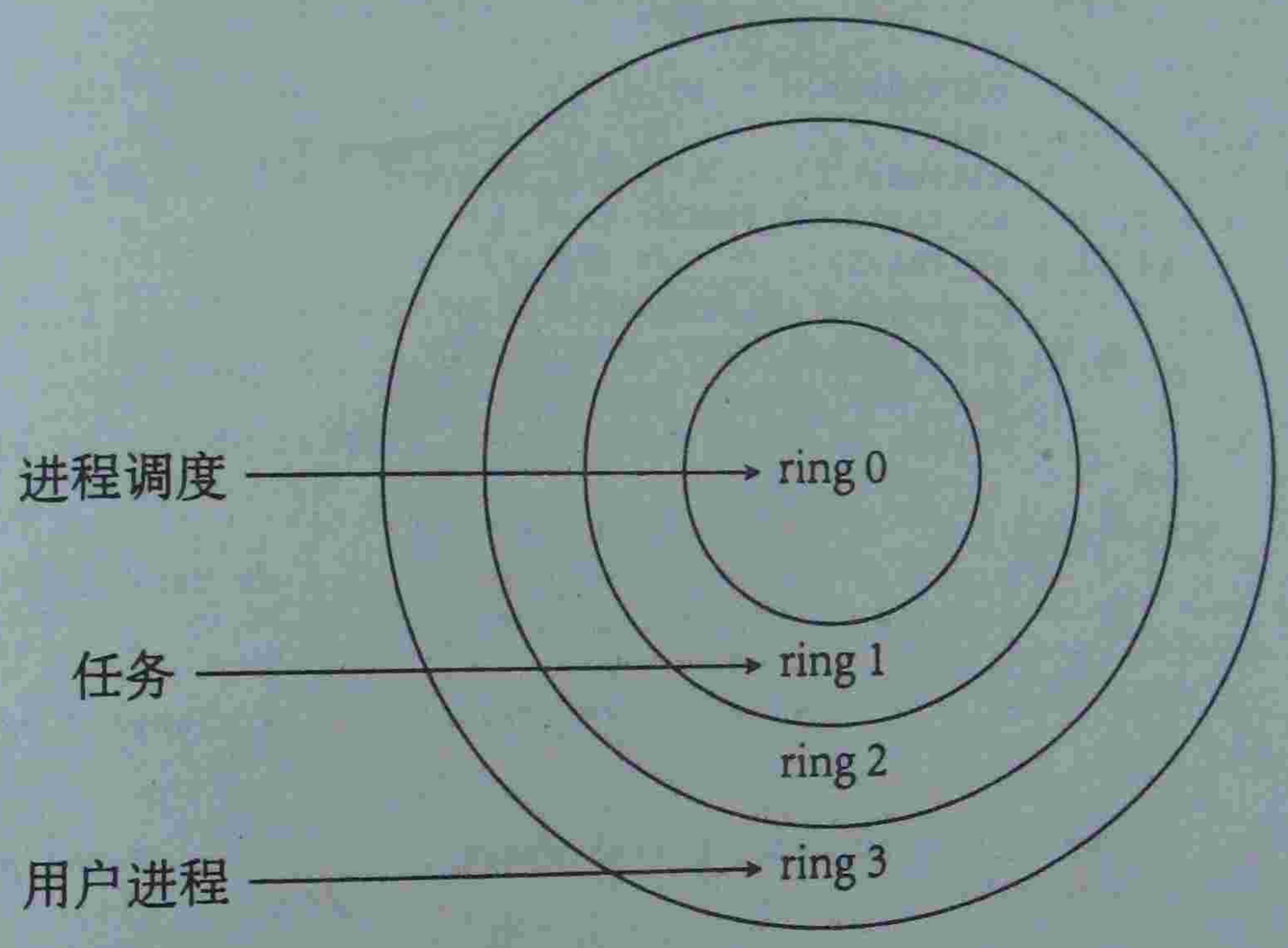


图7.24 区分任务和用户进程

除了ring2未加使用之外，这个图和图3.12表达的意思基本相同。不同的是，那时我们还只是模糊地知道特权级的概念，而如今我们马上要实现这样的系统了。好，现在就来修改。首先增加对NR_PROCS的定义（代码7.48）。

代码7.48 增加NR_PROC（chapter7/n/include/proc.h）

```
51  /* Number of tasks & procs */
52  #define NR_TASKS        1
53  #define NR_PROCS        3
```

增加NR_PROCS的同时，将NR_TASKS修改为1。

然后在所有用到NR_TASKS的地方都要做相应修改，首先是proc_table和task_table（代码7.49）。

代码7.49 区分task和proc（chapter7/n/kernel/global.c）

```
20  PUBLIC  PROCESS proc_table[NR_TASKS + NR_PROCS];
21
22  PUBLIC  TASK    task_table[NR_TASKS] = {
23          {task_tty, STACK_SIZE_TTY, "tty"}};
24
25  PUBLIC  TASK    user_proc_table[NR_PROCS] = {
```

```
        {TestA, STACK_SIZE_TESTA, "TestA"},
        {TestB, STACK_SIZE_TESTB, "TestB"},
        {TestC, STACK_SIZE_TESTC, "TestC"}};
```

我们新声明了一个数组user_proc_table[]，实际上这是权宜之计，因为完善的操作系统应该有专门的方法来新建一个用户进程，不过目前使用与任务相同的方法来做无疑是简单的。

初始化进程表的地方当然也要进行修改（代码7.50）。

代码 7.50 初始化进程表时区分 task 和 proc（chapter7/n/kernel/main.c）

```
PUBLIC int kernel_main()
{
...
        u8              privilege;
        u8              rpl;
        int             eflags;
        for (i = 0; i < NR_TASKS+NR_PROCS; i++) {
                if (i < NR_TASKS) {     /* 任务 */
                        p_task    = task_table + i;
                        privilege = PRIVILEGE_TASK;
                        rpl       = RPL_TASK;
                        eflags    = 0x1202; /* IF=1, IOPL=1, bit 2 is always 1 */
                }
                else {                  /* 用户进程 */
                        p_task    = user_proc_table + (i - NR_TASKS);
                        privilege = PRIVILEGE_USER;
                        rpl       = RPL_USER;
                        eflags    = 0x202; /* IF=1, bit 2 is always 1 */
                }

                strcpy(p_proc->p_name, p_task->name);   // name of the process
                p_proc->pid = i;                        // pid

                p_proc->ldt_sel = selector_ldt;

                memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS >> 3],
                       sizeof(DESCRIPTOR));
                p_proc->ldts[0].attr1 = DA_C | privilege << 5;
                memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS >> 3],
                       sizeof(DESCRIPTOR));
                p_proc->ldts[1].attr1 = DA_DRW | privilege << 5;
                p_proc->regs.cs = (0 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | rpl;
                p_proc->regs.ds = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | rpl;
                p_proc->regs.es = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | rpl;
                p_proc->regs.fs = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | rpl;
                p_proc->regs.ss = (8 & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | rpl;
                p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK) | rpl;

                p_proc->regs.eip = (u32)p_task->initial_eip;
                p_proc->regs.esp = (u32)p_task_stack;
                p_proc->regs.eflags = eflags;

                p_task_stack -= p_task->stacksize;
                p_proc++;
                p_task++;
                selector_ldt += 1 << 3;
        }
...
}
```

这里不但改变了用户进程的特权级，而且通过改变eflags，还剥夺了用户进程所有的 I/O 权限。

另外，还有protect.c中初始化 GDT 中 LDT 描述符的代码和proc.c中进程调度的相关代码也进行了修改，参见代码7.51和代码7.52。

代码 7.51 区分 task 和 proc 后初始化 GDT 中的 LDT 描述符（chapter7/n/kernel/protect.c）

```
PUBLIC void init_prot()
{
...
        for (i = 0; i < NR_TASKS+NR_PROCS; i++){
                init_descriptor(&gdt[selector_ldt>>3],
                                vir2phys(seg2phys(SELECTOR_KERNEL_DS),
                                         proc_table[i].ldts),
                                LDT_SIZE * sizeof(DESCRIPTOR) - 1,
                                DA_LDT);
                p_proc++;
                selector_ldt += 1 << 3;
        }
}
```

代码 7.52 区分 task 和 proc 后的进程调度（chapter7/n/kernel/proc.c）

```
PUBLIC void schedule()
{
        PROCESS* p;
        int      greatest_ticks = 0;

        while (!greatest_ticks) {
                for (p = proc_table; p < proc_table+NR_TASKS+NR_PROCS; p++) {
                        if (p->ticks > greatest_ticks) {
                                greatest_ticks = p->ticks;
                                p_proc_ready = p;
                        }
                }

                if (!greatest_ticks) {
                        for(p=proc_table;p<proc_table+NR_TASKS+NR_PROCS;p++) {
                                p->ticks = p->priority;
                        }
                }
        }
}
```

做了上述修改之后，就可以 make 并运行了。虽然它的运行结果与之前是一样的，但我们知道，这次的改动将又是一次标志性事件。它标志着Orange'S现在已运行在了 3 个特权级之上，普通的用户进程从此和系统任务区分开来。

7.5 printf

如今我们已经有一个任务和三个用户进程，但已经好久没有看见过 A、B、C三个进程的运行情况了。你一定也很想看到进程在特定终端运行的情景，而且，由于我们的 TTY 已具雏形，也是该编写一个供输出使用的printf()的时候了。

由于printf()要完成屏幕输出的功能，需要调用控制台模块中的相应代码，所以，它必须通过系统调用才能完成。

7.5.1 为进程指定 TTY

可以想像，当某个进程调用printf()时，操作系统必须知道往哪个控制台输出才行。而当系统调用发生，ring3 跳入 ring0 时，系统只能知道当前系统调用是由哪个进程触发的。所以，我们必须为每个进程指定一个与之相对应的TTY，这可以通过在进程表中增加一个成员来实现（代码7.53）。

代码 7.53 进程表中增加一个成员（chapter7/o/include/proc.h）

```
31  typedef struct s_proc {
32          STACK_FRAME regs;          /* process registers saved in stack frame */
33          u16 ldt_sel;               /* gdt selector giving ldt base and limit */
34          DESCRIPTOR ldts[LDT_SIZE]; /* local descriptors for code and data */
35
36          int ticks;                 /* remained ticks */
37          int priority;
38
39          u32 pid;                   /* process id passed in from MM */
40          char p_name[16];           /* name of the process */
41
42          int nr_tty;
43  }PROCESS;
```

我们还是用与初始化PROCESS的ticks和priority成员时相同的方法来为nr_tty设置初值（代码7.54）。

代码 7.54 为 nr_tty 设置初值（chapter7/o/kernel/main.c）

```
22  PUBLIC int kernel_main()
23  {
    ...
34          for (i = 0; i < NR_TASKS+NR_PROCS; i++) {
    ...
70                  p_proc->nr_tty = 0;
    ...
76          }
    ...
83          proc_table[1].nr_tty = 0;
84          proc_table[2].nr_tty = 1;
85          proc_table[3].nr_tty = 1;
    ...
98  }
```

可以看到，在for循环中，所有进程的nr_tty被初始化成 0，这样，所有进程默认与第 0 个 TTY 绑定。不过在后面，B 和 C 两个进程与第 1 个 TTY 绑定。这意味着，将来 B 和 C 的输出将同时出现在控制台 1，而 A 的输出出现在控制台 0。

7.5.2 printf()的实现

函数printf()对于我们来说肯定是非常熟悉，从学习 HelloWorld 的时候就开始用它了。但它的实现却并不简单，首先是它的参数个数和类型都可变，而且其表示格式的参数（比如“%d”、“%x”等）形式多样，在printf()中都要加以识别。

不过，按照我们一贯的风格，开始时只实现一个简单点的。下面的printf只支持“%x”一种格式（代码7.55）。

代码 7.55 printf（chapter7/o/kernel/printf.c）

```
52 int printf(const char *fmt, ...)
53 {
54         int i;
55         char buf[256];
56
57         va_list arg = (va_list)((char*)(&fmt) + 4); /*4是参数fmt所占堆栈中的大小*/
58         i = vsprintf(buf, fmt, arg);
59         write(buf, i);
60
61         return i;
62 }
```

其中，vsprintf()的实现方法如代码7.56所示。

代码 7.56 vsprintf（chapter7/o/kernel/vsprintf.c）

```
19 int vsprintf(char *buf, const char *fmt, va_list args)
20 {
21         char*   p;
22         char    tmp[256];
23         va_list p_next_arg = args;
24
25         for (p=buf;*fmt;fmt++) {
26                 if (*fmt != '%') {
27                         *p++ = *fmt;
28                         continue;
29                 }
30
31                 fmt++;
32
33                 switch (*fmt) {
34                 case 'x':
35                         itoa(tmp, *((int*)p_next_arg));
36                         strcpy(p, tmp);
37                         p_next_arg += 4;
38                         p += strlen(tmp);
39                         break;
40                 case 's':
41                         break;
42                 default:
43                         break;
44                 }
45         }
46
47         return (p - buf);
48 }
```

虽然本书不是介绍C语言的，但这里遇到可变参数这个问题了，对于其原理就简单介绍一下。

我们知道，调用一个函数时，总是先把参数压栈，然后通过call指令转移到被调用者，在完成后清理堆栈。但这里遇到两个问题，一是如果有多个参数，哪个参数先入栈，是前面的还是后面的？二是由谁来清理堆栈，调用者还是被调用者？

这两个方面的问题其实被称为“调用约定”（Calling Conventions），这个概念在第5.2节中曾经提过一次。调用约定有若干种，每一种都规定参数入栈的顺序以及谁来清理堆栈。我们已经用汇编语言写过不少的函数，都是后面的参数先入栈，并且由调用者清理堆栈。这种约定被称做C调用约定。

C调用约定的好处在处理可变参数函数时得到了充分体现，因为只有调用

者知道此次调用包含几个参数，于是可以方便地清理堆栈。

C 调用约定让使用可变参数的函数成为可能，可具体怎么做呢？首先是它的声明，过去我们写的函数，都有确定类型的参数，可现在不同了，参数的个数和类型都不知道，于是，省略号就派上了用场，正如代码7.55所示，一个省略号，表示参数不知道有多少，更不知道是什么。

可是在每一次调用过程中，printf必须有一种方法来使用这些参数才行。从代码7.55可以看到，printf使用了它的第一个参数fmt作为基准，得到了后面若干参数的开始地址，这样，其值也就容易得到了。

举一个例子，假设我们调用printf(fmt, var1, var2, var3)，则堆栈情况将如图7.25所示。

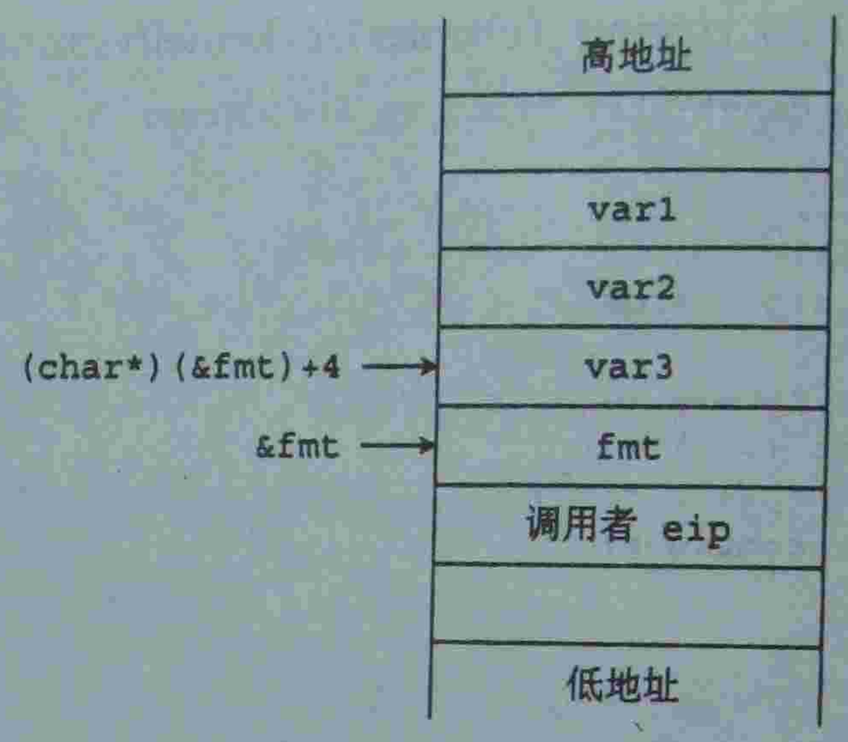


图 7.25 printf 调用后的堆栈情况

一目了然，&fmt表示fmt的地址，(char*)(&fmt)+4则表示紧随fmt之后的参数，即var1的地址。所以，接下来实际上是将var1的地址传递给了紧接着调用的vsprintf。va_list其实就是char*，它的定义在type.h中。

函数vsprintf的实现见代码7.56，虽然它只识别“%x”这一种格式，但其他格式的原理也是一样的，即根据%后的格式字符就能判断下一个参数的类型，从而知道从堆栈中取出什么。

经过这一番说明，读者一定已经理解了代码7.55和代码7.56这两段代码。值得说明的是，如果你阅读 Minix 或者 Linux 源代码的话，会发现printf中有va_start、va_end这样的宏，进一步阅读它们的定义你会发现，它们本质上与我们的代码是一样的，只是我们的代码更“赤裸裸”，从而更直观，更容易理解。

代码7.55中的write()便是我们即将完成的系统调用了，它把vsprintf输出的字符串打印到屏幕上。

7.5.3 系统调用write()

我们已经实现过一个系统调用get_ticks()，所以再增加一个不再是难事。增加一个系统调用（假设为foo）的过程如表7.6所示。

表 7.6 增加一个系统调用的过程

步骤	内容	文件
1	NR_SYS_CALL 加一	const.h
2	为sys_call_table[]增加一个成员，假设是sys_foo	global.c
3	sys_foo的函数体	因具体情况而异
4	sys_foo的函数声明	proto.h
5	foo的函数声明	proto.h
6	_NR_foo 的定义	syscall.asm
7	foo的函数体	syscall.asm
8	添加global foo	syscall.asm
9	如果参数个数与以前的系统调用比有所增加，则需要修改sys_call	kernel.asm

我们把这个新增的系统调用取名为write()，把它对应的内核部分取名为sys_write()，它们的声明在proto.h中（代码7.57）。

代码 7.57 关于 write() 的函数声明（chapter7/o/include/proto.h）

```
59 PUBLIC  int     sys_write(char* buf, int len, PROCESS* p_proc);
    ...
65 PUBLIC  void    write(char* buf, int len);
```

这样其实第 4、5 步已经做好，而且步骤 1、2、6、8 都是很容易的，剩下的工作就是添加write()和sys_write()这两个函数体了。先来看write()，见代码7.58。

代码 7.58 write()（节自 chapter7/o/kernel/syscall.asm）

```
12 _NR_write            equ 1
    ...
16 global  write
    ...
32 write:
33         mov     eax, _NR_write
34         mov     ebx, [esp + 4]
35         mov     ecx, [esp + 8]
36         int     INT_VECTOR_SYS_CALL
37         ret
```

这里使用了ebx和ecx来传递两个参数。由于我们已有的系统调用是没有参数的，所以一会儿我们还需要修改sys_call。

再来看一下sys_write()，见代码7.59。

代码 7.59 sys_write（chapter7/o/kernel/tty.c）

```
158 PUBLIC void tty_write(TTY* p_tty, char* buf, int len)
159 {
160         char* p = buf;
161         int i = len;
162
163         while (i) {
164                 out_char(p_tty->p_console, *p++);
```

```
165                     i--;
166             }
167     }
        ...
172     PUBLIC int sys_write(char* buf, int len, PROCESS* p_proc)
173     {
174             tty_write(&tty_table[p_proc->nr_tty], buf, len);
175             return 0;
176     }
```

sys_write()通过调用新增加的简单函数tty_write()来实现字符的输出。注意，sys_write()比write()多了一个参数，这个参数也是在我们即将要修改的sys_call中压栈的（代码7.60）。

代码7.60 修改sys_call()（节自chapter7/o/kernel/kernel.asm）

```
341     sys_call:
342             call    save
⇒ 343           push    dword [p_proc_ready]
344             sti
345
⇒ 346           push    ecx
⇒ 347           push    ebx
348             call    [sys_call_table + eax * 4]
⇒ 349           add     esp, 4 * 3
350
351             mov     [esi + EAXREG - P_STACKBASE], eax
352             cli
353             ret
```

由于当前运行的进程就是通过设置p_proc_ready来恢复执行的，所以当进程切换到未发生之前，p_proc_ready的值就是指向当前进程的指针。把它压栈就将当前进程，即write()的调用者指针传递给了sys_write()。

7.5.4 使用printf()

这样，我们的第二个系统调用printf()就完成了。下面在3个用户进程中调用它（代码7.61）。

代码7.61 使用printf（chapter7/o/kernel/main.c）

```
103     void TestA()
104     {
105             int i = 0;
106             while (1) {
⇒ 107                   printf("<Ticks:%x>", get_ticks());
108                     milli_delay(200);
109             }
110     }
        ...
115     void TestB()
116     {
117
118             int i = 0x1000;
⇒ 119           while(1){
120                     printf("B");
121                     milli_delay(200);
122             }
        ...
127     void TestC()
128     {
```

```
129         int i = 0x2000;
130         while(1){
131                 printf("C");
132                 milli_delay(200);
133         }
134 }
```

运行，成功了！图7.26和图7.27便是控制台 0 和控制台 1 的情景。

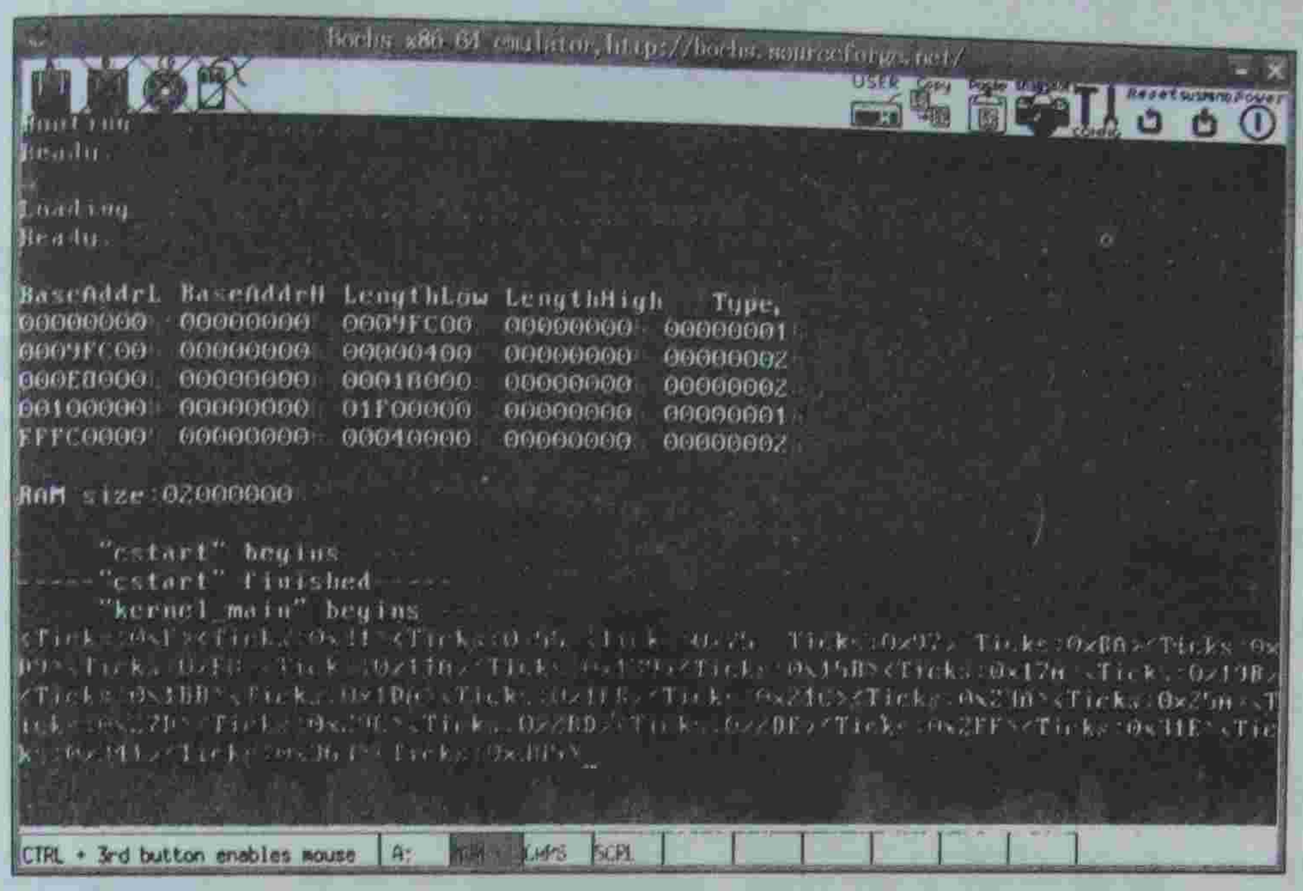


图 7.26 printf打印出的字符和数字（控制台 0）

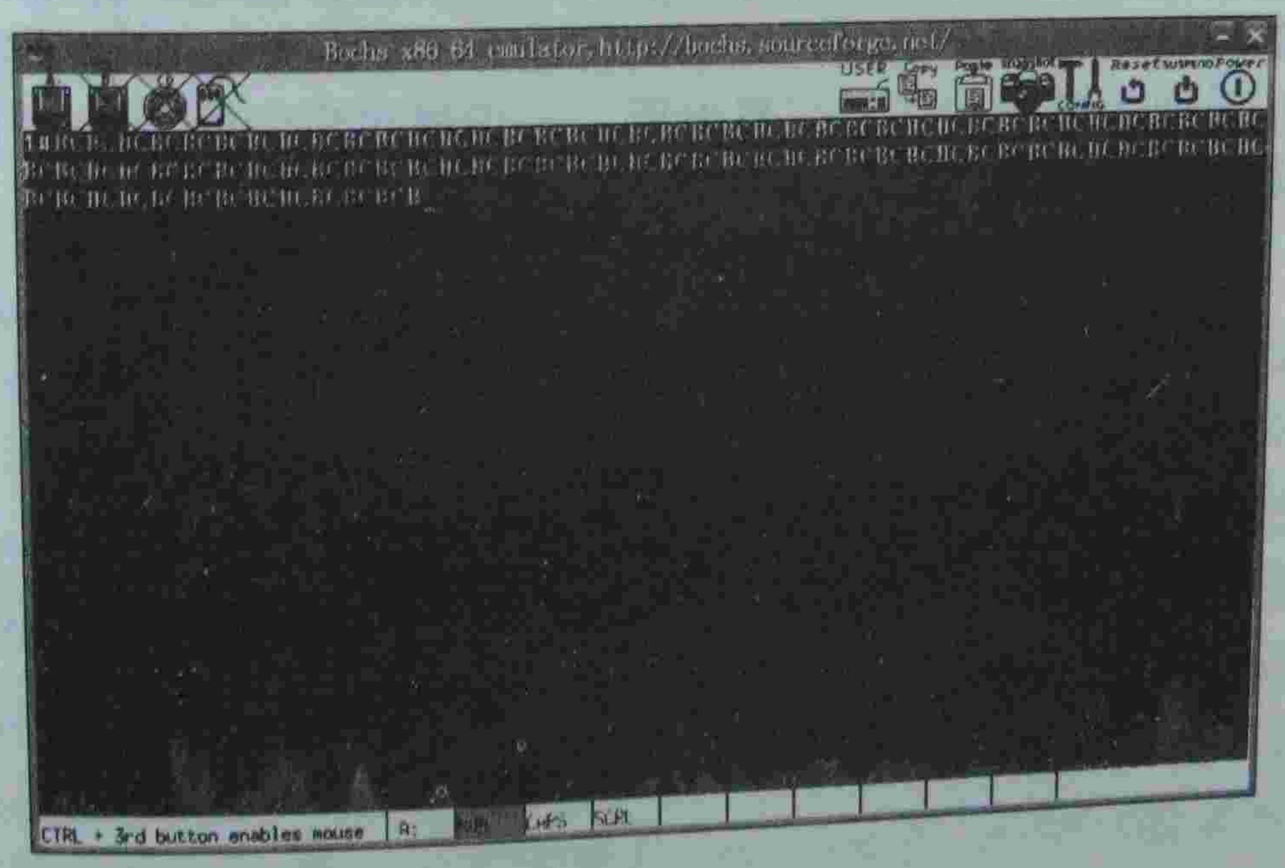


图 7.27 printf打印出的字符（控制台 1）

太棒了，我们终于有了自己的printf，从此不但可以告别disp_str，而且，它是一个用户态的程序，可以被普通的用户进程调用。

喜悦之余，让我们回顾一下printf的调用过程，如图7.28所示。

一个系统调用涉及特权级的切换，所以从实现到运行还是有一点复杂的。不过图7.28清晰地表示了这一过程，其中箭头表示函数之间的调用关系。

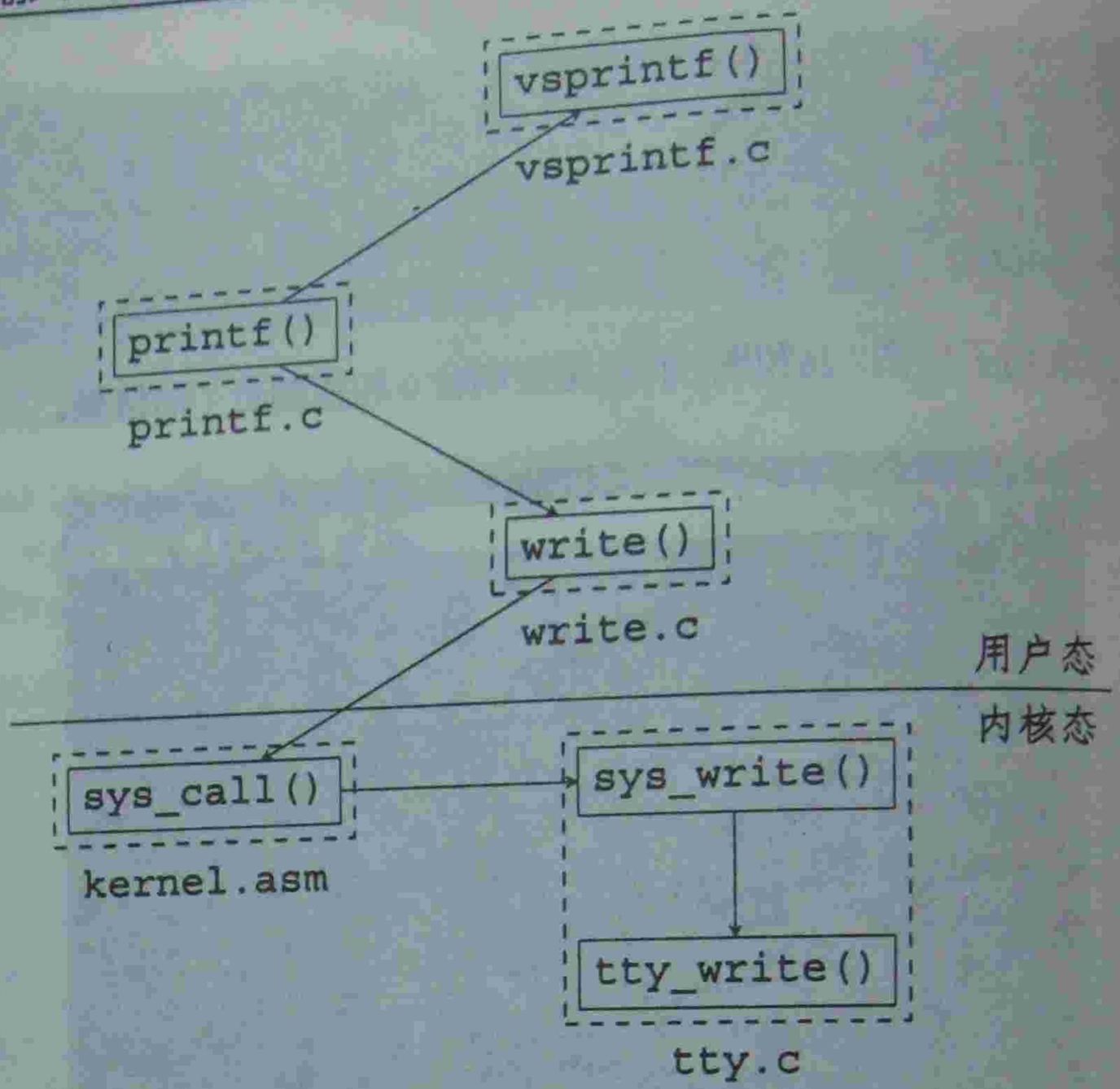


图 7.28 printf 调用过程示意图

一切都明白之后，就让我们好好享受一下我们的作品吧。你既可以来到控制台 2 的空白屏幕敲击键盘，也可以来到控制台 0 看当前打印的数字，也可以来到控制台 1 看两个进程的字母交替出现。一切都向我们展示着一个多任务多控制台操作系统的特性。

下篇

在本书的序言中笔者已经提到，下篇将介绍进程间通信、文件系统和内存管理。这些概念在任何一本操作系统中都能找到，而且通常它们登场时都是浓墨重彩。本书不会重复概念性的叙述，而仍然是以“动手写”为基调，并以极其简化的设计来尽量减少初学时可能遇到的困扰。与此同时，笔者会分享自己的调试技巧，并期待达到抛砖引玉的功效。

第8章介绍进程间通信。在这一章的开头，额外做了两件事情，一件是通过分析具体实例来比较宏内核和微内核，并做出我们的选择；另一件是增加两个函数：assert()和panic()，以便在运行过程中随时发现错误，从而减轻调试的难度。进程间通信，顾名思义，必然涉及许多进程间相互收发消息，在此过程中难免遇到一些随机情况，这使得调试的难度增加了。如果亲自实践的话，读者会发现 assert()和 panic()这两个函数能帮上大忙。

进程间通信的方法有若干，本书采用的是同步消息机制。具体的原理书中有详细的介绍。读者需要提前了解的是，这部分代码虽然并不多，但需要格外的细心和耐心。必要时，请别忘了我们有 Bochs 这一调试利器。

第9章介绍文件系统。书中采用了十分简陋的设计，甚至不惜牺牲文件系统的性能和部分功能。然而这一简陋的设计目前还是够用了，更加关键的是，我们可以从一个简单的设计中先得以管窥全豹。实现一个相对较好的文件系统，可以作为下一步的目标。

第10章介绍内存管理。这一章的主线并不是教读者在进程中分配和释放内存，这或许多少出乎你的意料。在这一章的最后，我们实现了一个简单的 shell，并可以执行通过交叉编译的方式编写的应用程序了。至于具体的实现过程，读者阅读之后便知。

第11章是比较轻松的一章，主要是将我们自己的操作系统与现实世界接轨。在这一章中，你将了解如何将自己编写的操作系统安装到硬盘，以及如何使之与现有的操作系统共存等内容。

总体来看，下篇避免重复在其他书中容易找到的概念，而着重于如何一步一步去将概念转化成现实的代码，这一思想是跟上篇一脉相承的。不过跟上篇的事无巨细相比，下篇还是稍有简略。原因是笔者相信通过上篇的实践，读者已经对开发环境和编译运行等步骤已经熟悉，所以有些琐碎的地方就不多赘言。如果读者有哪些地方不明白，没关系，我们网络上见。再次提醒读者，本书的官方网站位于：http://www.osfromscratch.org。

Love your neighbor, yet don't pull down your hedge.

—— B. Franklin

8 进程间通信

我们提到过，当一个进程需要操作系统的帮助，它可以通过系统调用让内核来替它完成一些工作。迄今为止，我们已经熟悉了系统调用的工作机制，并且已经实现了不止一个系统调用。接下来你会发现，用户进程将会有更多事情依赖于内核。比如我们想实现一个文件系统，最起码读写硬盘的工作要求助于内核。这里我们可以逐渐地增加系统调用，但也可以采用另一种方案，就是将这些工作剥离出来，交给一些系统进程来完成，让内核只负责它必须负责的工作，比如进程调度。这种将内核工作简单化的思想，便是微内核的基本思想。而所有工作通过系统调用扔给内核态的做法，被称为宏内核。

在基于宏内核的操作系统中，完成具体任务时，用户进程通过系统调用让内核来做事，直来直去，我们之前已经很熟悉了。在基于微内核的操作系统中，这个过程稍微复杂一些。在完成具体任务时，内核的角色很像是个中介。就比如我们将要实现的文件系统吧，设想用户进程 P 读取一个文件，首先通过内核告诉进程 FS，然后 FS 再通过内核告诉驱动程序（也是一个独立的进程），驱动程序读取硬盘，返回结果。这样一来，一项工作的完成变得有些曲折，需要多个进程协同工作。于是，进程间通信也就变得至关重要了。

到如今，我们的操作系统慢慢长大，接下来我们要用它来管理磁盘和磁盘上的文件并管理内存等，这些都要向应用程序提供接口，到了必须决定用微内核还是宏内核的时候了。怎么办呢？当然不能抛个硬币了事。我们不妨先找两个具体的例子来看看它们分别是怎么回事，看完了，明白了，再做决定也不迟。

8.1 微内核还是宏内核

微内核和宏内核的例子都非常好找。我们一直拿在手边的 Minix，以及每天在用的 Linux，便是两者的典型例子。Minix 是微内核的，Linux 则是宏内核的。

说起这两个例子，有一段轶事不能不提。那就是当年 Tanenbaum 和 Linus 一老一少的口舌之争。话说 Linus 写了个操作系统叫做 Linux，使用的是宏内核，他把这个消息发在了 comp.os.minix 新闻组上，这时 Tanenbaum 说话了，把 Linux 批评了一通，年轻气盛的 Linus 于是发信回击，这样一来二去，为我们留下一段微内核与宏内核的经典争论。

争论的全部内容在这里我们就不全部转述了，读者感兴趣的话可以用搜索引擎很容易地搜到[1]，我们把其中的重点说一下。在谈到微内核和宏内核时，Andy（Andrew S. Tanenbaum）是这样说的：

> 老一点的操作系统都是宏内核的，也就是说，整个操作系统是一个运行在核心态的单独的 a.out 文件，这个二进制文件包含进程管理、内存管理、文件系统以及其他。具体实例包括 UNIX、MS-DOS、VMS、MVS、OS/360、MULTICS 等。
>
> 另一种便是微内核，在这种系统中操作系统的大部分都运行在单独的进程，而且多数在内核之外。它们之间通过消息传递来通信。内核的任务是处理消息传递、中断处理、底层的进程管理，以及可能的 I/O。这种设计的实例有 RC4000、Amoeba、Chorus、Mach，以及还没有发布的 Windows/NT。
>
> 我完全可以（但不必）再讲述一段关于两者之间相对优势的很长的故事，然而在实际设计操作系统的人中间说说就够了，争论实际上已经结束。微内核已经取得了胜利。对于宏内核而言唯一的争论焦点在于效率，不过已经有足够的证据表明微内核可以像宏内核一样快（比如 Rick Rashid 已经发表了 Mach 3.0 和宏内核系统的比较报告）所以那不过是喊喊而已罢了。
>
> Minix 是微内核的，文件系统和内存管理是单独的进程，它们运行在内核之外。I/O 驱动也是单独的进程（在内核之内，但仅仅是因为 Intel CPU 的糟糕设计使得很难不这样做）。Linux 是个宏内核的系统。这相当于向七十年代倒退了一大步。就好比将一个已存在的工作得很好的 C 程序用 Basic 重写一遍。在我看来，在 1991 年写一个宏内核的系统真不是个好主意。

以上前两段基本上可以被认为是宏内核和微内核的基本概念。从概念上我们不难猜到，宏内核看上去试图包办一切，而微内核恰恰相反，它的任务只是“处理消息传递、中断处理、底层的进程管理，以及可能的 I/O”，而其他事情都交给内核之外单独的进程来完成。

在这段文字中 Andy 不但阐述了宏内核和微内核的概念，摆明了对于这个问题鲜明的观点，而且他也毫不掩饰自己对宏内核的不屑。而且这种不屑让他认为 Linux 简直是技术的倒退。在随后的文字中，对于 Linux 的可移植性 Andy 也做了不客气的批评。也难怪 Linus 对此非常恼火。从 Linus 的第一个回复开始，这场争论开始变得精彩起来。

Linus 的回复是这样开始的：

> 好吧，既然是这么一个主题，我恐怕不能不做回答了。向已经听了太多关于 Linux 的 Minix 使用者们道歉。我很乐意上钩（Andy 说了这些话，好像在引诱 Linus 开始一场争论——笔者注），该是吵一架的时候了！
>
> 啊哈，看来 Linus 真的被激怒了，我仿佛看到了他挽起袖子的样子☺。是

[1] 或者在维基百科上看一下：http://en.wikipedia.org/wiki/Tanenbaum-Torvalds_debate。

啊，看到自己辛辛苦苦的劳动成果被人冠以“过时了”的形容，谁还能平心静气呢？

针对微内核和宏内核之争，他是这样回应的：

> 是的，Linux 是宏内核，我同意微内核更好些。如果不是你使用了具有争论性的主题，我可能会同意你大部分的观点。从理论上（以及美学上）讲 Linux 是输了。如果去年春天 GNU 内核已经做好，我可能不会这么麻烦地开始我的工作：问题是它没有做好而且到现在都没有。在已经实现这一方面 Linux 赢大了。
>
> » Minix 是微内核系统……Linux 则是宏内核的。
>
> 如果这是评价内核好坏的唯一标准，那么你是对的。你没有提到的是 Minix 的微内核实现得并不好，而且（内核内）多任务存在问题。如果我做一个多线程文件系统存在问题的操作系统，我可能不会这么快就声讨别人：事实上我会尽最大努力让别人忘掉我的失败。

这一段我觉得非常重要，因为看得出来，Linus 内心还是承认微内核的优势的，而且他提到了“美学”（aesthetical）这个词，因为的确，微内核的思想更加优雅，这在我们下文中的分析中也可以看到。不过尽管如此，他还是批评了 Minix 本身，认为它的微内核实现的并不令人满意。

在后来谈到可移植性的时候，Linus 的话也颇具初生牛犊不怕虎的劲头：

> 可移植性是为不能编写新程序的人设计的
>
> ——我，现在（使用傲慢的语气）

真的很精彩不是么？我甚至感觉有点像在看武侠片，一老一少，出招拆招，虽是打架，但颇有章法。不难看出，刚刚这句“我，现在（使用傲慢的语气）”甚至带有一丝挑衅意味，看这句话我甚至在想像着 Linus 敲出这行字的时候该是带着怎样傲慢的神色——不过谁在年轻的时候不是这样气盛的呢？呵呵。

看过热闹之后，让我们来实地勘查一下，两种内核看上去是什么样子的。我们就以系统调用作为突破口，看看它们的代码。

8.1.1 Linux 的系统调用

为了简单起见，我们拿 Linux 0.01 作为 Linux 方代表——最新的 Linux 内核代码量太大了，不利于短时间内弄懂。其中的系统调用不止一个，我们以`fork()`为例来分析一下。为了让读者一下子就理清这个系统调用的脉络，对于代码细节这里就不细言了，我们直接来看程序的流程图（如图8.1所示）。相关的具体代码读者可以在 Linux 0.01 的以下文件中找到：

- init/main.c
- include/unistd.h
- kernel/sched.c

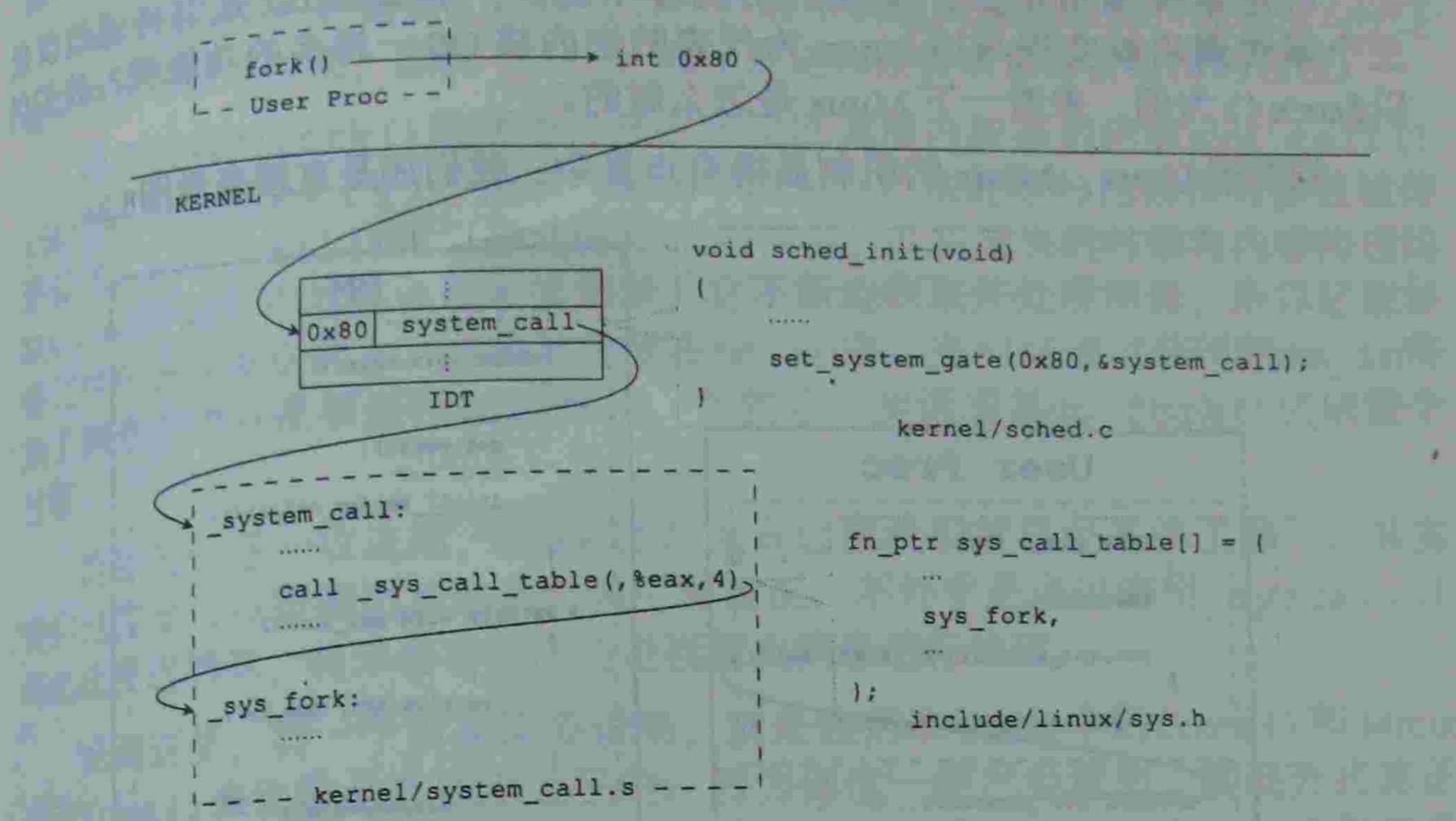


图 8.1 Linux 0.01 的 fork 系统调用

从图8.1中我们可以看出，调用fork()实际上是调用了中断0x80，通过事先初始化好的 IDT，程序转移到了_system_call，最终通过一个函数指针数组sys_call_table[]转化成了调用sys_fork()。这跟我们实现过的系统调用是很相似的，此处不再赘述。

8.1.2 Minix 的系统调用

Linux 的fork系统调用很容易理解，但 Minix 的就不这么简单了，它刚开始甚至可能让你感到迷惑。我们来打开 Minix 代码文件src/kernel/proc.c，看一下函数sys_call()的开头（见代码8.1）。

代码 8.1 Minix 的 sys_call()

```
PUBLIC int sys_call(function, src_dest, m_ptr)
int function;                   /* SEND, RECEIVE, or BOTH */
int src_dest;                   /* source to receive from or dest to send to */
message *m_ptr;                 /* pointer to message */
{
/* The only system calls that exist in MINIX are sending and receiving
 * messages.  These are done by trapping to the kernel with an INT instruction.
 * The trap is caught and sys_call() is called to send or receive a message
 * (or both). The caller is always given by proc_ptr.
 */
...
}
```

开头这段注释非常重要，一个“only”道破天机（或者将你搞晕）：在 Minix 中，不再像 Linux 那样有许许多多的系统调用（sys_call_table[]中

列出的有几十个），而是仅有发送和接收消息的系统调用。通过sys_call的参数function的注释我们可以知道，系统调用的种类总共有三个，那就是SEND、RECEIVE以及BOTH。

可是系统调用虽少，实现的功能却不能少，那么 Minix 是怎样通过仅仅三个系统调用就实现与以 Linux 为代表的宏内核 OS 一样多的功能呢？我们仍以fork()为例，来看一下 Minix 是怎么做的。

相对于 Linux，Minix 的机制显得有点复杂，我们还是直接来看图8.2。

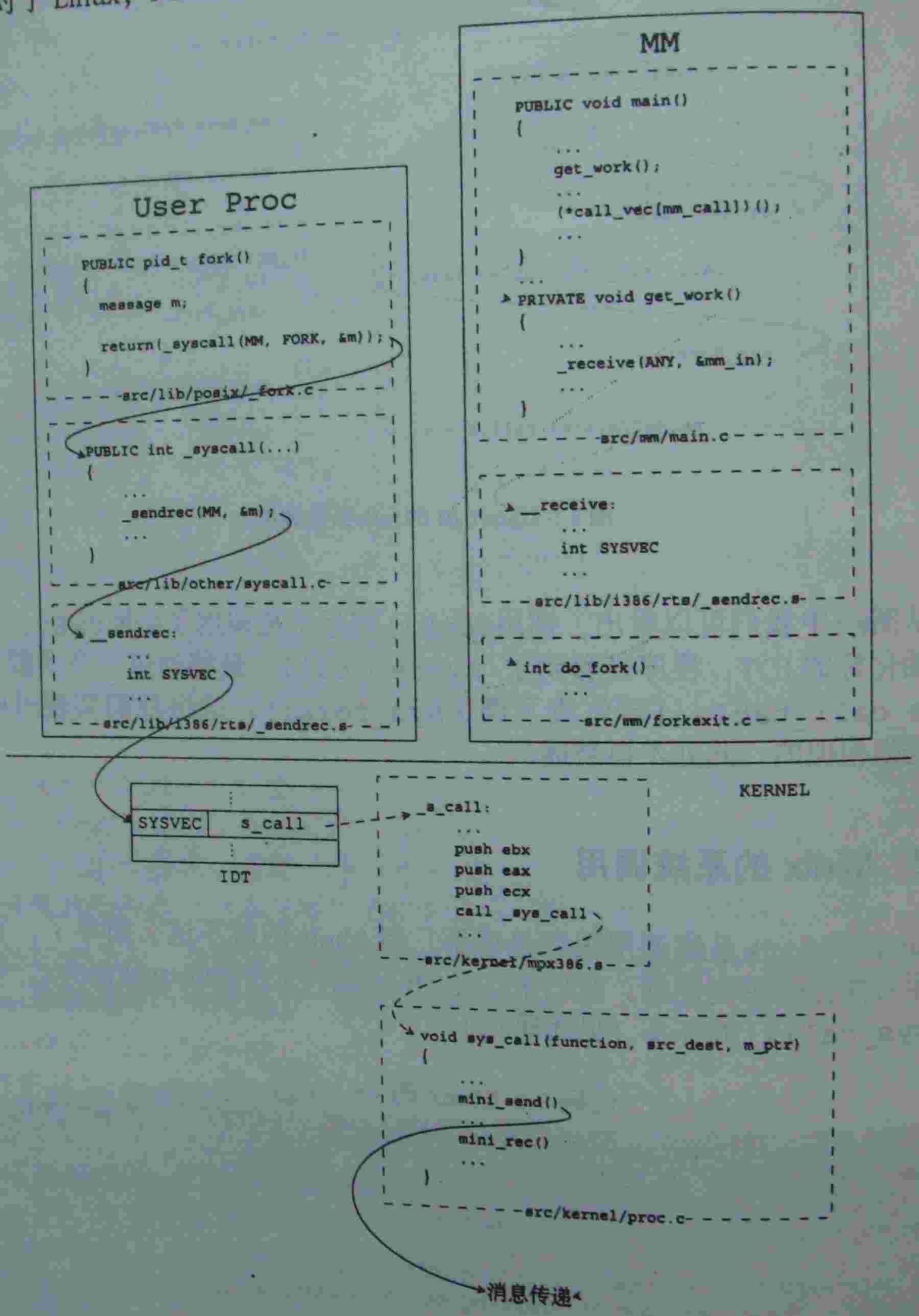


图 8.2 Minix 的 fork "系统调用"

跟图8.1所示的 Linux 不同，这里多出来一个内存管理器（MM），fork()所要做的工作是由它来负责的（如果是另外的系统调用，那么具体工作可能就不是由 MM 来负责，比如系统调用read()就是由 FS 来负责的，跟 MM 类似，

FS是单独运行的另一个进程），那么MM是如何得到用户进程的通知的呢？正是消息机制在进程之间起到了重要作用，它类似于邮政系统，在信封（或包裹单）上写明目的地，消息就送达了。

图中使用了三种箭头，实线（——→）表示消息的发送过程，点线（……→）表示消息的获取过程，虚线（- - -→）表示发送和接收消息都会经历的过程。

用户进程对fork()的调用将最终转化成调用内核态的函数sys_call()（位于src/kernel/proc.c中），消息（即图中的m）的地址这时已经作为参数被传递进来，sys_call()可以据此得知m的内容，并在适当的时候将内容传递给MM。MM的工作其实说起来很简单，它不断地获取并处理消息，所以它能够得到用户进程发送的m，并将其存放在mm_in中。当MM通过获得的mm_in得知了消息的内容是要进行fork操作，它就进一步调用其do_fork()完成整个过程。

消息的一送一收之间，fork()的大致过程我们就已经基本了解了。其实我们也完全可以猜测出其他系统调用的情况。不外乎是通过调用_syscall()转化成发送消息，将来会有相应的进程取出消息进行处理。

说到这里，有一个情况需要说明，就是我们拿Linux中的fork()和Minix中的fork()来比较是有些不公平的。因为你也一定已经看到，实现方式真正与Linux的fork()相同的是Minix的_sendrec()和_receive()，它们都是通过中断进入内核，在内核中完成任务。而Minix中的fork()是通过两个进程分别调用_sendrec()和_receive()这两个系统调用来实现的，从这个意义上来说，Minix中的fork()其实不算系统调用（这也是函数sys_call()的开头注释中说Minix的系统调用只有三个的原因），它只是在完成一个紧密依赖系统调用的工作罢了。不过从用户的角度，这种差别是看不到的，而且只要调用fork()时能实现需要的功能，这种差别就无关紧要。因此用户完全可以称fork()为一个系统调用。

实现方法的差别源于设计思路的不同。在Minix中，真正的系统调用只有三个，这意味着内核不必事无巨细地处理用户进程要求的所有工作，只需要做好其“邮局”的职能，将消息按照需求来回传送就够了。在Linux中内核所做的工作，在Minix中被交给专门的进程来完成。你可能一下子就明白了，原来微内核的“微”字是让内核功能最简化的意思。

8.1.3 我们的选择

到这里，微内核和宏内核各自的工作原理我想读者已经明白了。同时它们的优缺点也基本上清楚了。宏内核的优势在于其逻辑简单，直截了当，实现起来也容易，而且也因为它的直接，避免了像微内核那样在消息传递时占用资源。而微内核的优势在于，它的逻辑虽相对复杂但非常严谨，结构上显得非常优雅和精致，而且程序更容易模块化，从而更容易移植。

从编程的难易程度上来看，宏内核看上去具有一定优势，因为它很直接，不需要绕弯子，但从长期来看，当内核逐渐变大，微内核的结构会更加清晰。虽然选择微内核意味着有调试起来有些困难的消息机制摆在面前，但从设计理念上来看，微内核更加“摩登”，更“酷”。而且，从学习编程的角度看，搞一个微内核可以为将来架构其他东西作为很有益的参考。基于这些原因，我们选

择微内核。

选择了微内核，首要的任务就比较明显了，那就是实现一个进程间通信（IPC）机制。其实这也没有想像中那么难，我们下面就来看一看。

8.2 IPC

IPC 是 Inter-Process Communication 的缩写，直译为进程间通信，说白了就是进程间发消息。我们在上一节中把这种消息传递比作邮政系统，但实际上这种比喻并不全对。有的消息机制是很像收发邮件的，这种叫做异步 IPC，意思是说，发信者发完就去干别的了，收信者也一样，看看信箱里没信，也不坐在旁边傻等。而有另一种消息机制正好相反，被称为同步 IPC，它不像邮寄，倒像接力赛，发送者一直等到接收者收到消息才肯放手，接收者也一样，接不到就一直等着，不干别的。

当然你可以把同步 IPC 也比作邮寄，只不过寄信的人从把信投到信箱里的那一刻开始，就住在邮局不走了，其他什么也不干了，就等着邮局说：“哥们儿，你的信对方已经收到了，放心回家吧！”这才恋恋不舍地离开。收信的也一样，一旦决定收信，就守在自家信箱前面不走了，一直等，连觉也不睡，望穿秋水，等信拿在手里了，这才回屋，每收一次信，就得瘦个十几斤。

我们都是性情中人，我们选择傻等，或曰同步 IPC。

同步 IPC 有若干的好处，比如：

- 操作系统不需要另外维护缓冲区来存放正在传递的消息；
- 操作系统不需要保留一份消息副本；
- 操作系统不需要维护接收队列（发送队列还是需要的）；
- 发送者和接收者都可在任何时刻清晰且容易地知道消息是否送达；
- 从实现系统调用的角度来看，同步 IPC 更加合理——当使用系统调用时，我们的确需要等待内核返回结果之后再继续。

这些特性读者可能无法一下子全部明白，不要紧，我们接下来写完代码，你就全都明白了。

8.3 实现 IPC

Minix 的 IPC 机制我们已经明白了，它的核心乃在于“`int SYSVEC`”这个软中断以及与之对应的`sys_call()`这个函数。增加一个系统调用对我们来讲已是信手拈来的事，按照表7.6一步一步来就好了。我们把这个新的系统调用起名为`sendrec`。`sendrec`和`sys_sendrec`的函数体分别见代码8.2和代码8.3。

代码 8.2 sendrec（节自 chapter8/a/kernel/syscall.asm）

```
25 sendrec:
```

```
        mov     eax, _NR_sendrec
        mov     ebx, [esp + 4]  ; function
        mov     ecx, [esp + 8]  ; src_dest
        mov     edx, [esp + 12] ; p_msg
        int     INT_VECTOR_SYS_CALL
        ret
```

代码 8.3 sys_sendrec（chapter8/a/kernel/proc.c）

```
/*****************************************************************************
 *                                sys_sendrec
 *****************************************************************************/
/**
 * <Ring 0> The core routine of system call 'sendrec()'.
 *
 * @param function SEND or RECEIVE
 * @param src_dest To/From whom the message is transferred.
 * @param m        Ptr to the MESSAGE body.
 * @param p        The caller proc.
 *
 * @return Zero if success.
 *****************************************************************************/
PUBLIC int sys_sendrec(int function, int src_dest, MESSAGE* m, struct proc* p)
{
        assert(k_reenter == 0); /* make sure we are not in ring0 */
        assert((src_dest >= 0 && src_dest < NR_TASKS + NR_PROCS) ||
               src_dest == ANY ||
               src_dest == INTERRUPT);

        int ret = 0;
        int caller = proc2pid(p);
        MESSAGE* mla = (MESSAGE*)va2la(caller, m);
        mla->source = caller;

        assert(mla->source != src_dest);

        /**
         * Actually we have the third message type: BOTH. However, it is not
         * allowed to be passed to the kernel directly. Kernel doesn't know
         * it at all. It is transformed into a SEND followed by a RECEIVE
         * by 'send_recv()'.
         */
        if (function == SEND) {
                ret = msg_send(p, src_dest, m);
                if (ret != 0)
                        return ret;
        }
        else if (function == RECEIVE) {
                ret = msg_receive(p, src_dest, m);
                if (ret != 0)
                        return ret;
        }
        else {
                panic("{sys_sendrec} invalid function: "
                      "%d (SEND:%d, RECEIVE:%d).", function, SEND, RECEIVE);
        }

        return 0;
}
```

表7.6的最后一步中提到，如果参数个数与以前的系统调用比有所增加，则需要修改kernel.asm中的sys_call。额外要注意，我们新加的参数是通过edx这个参数传递的，而save这个函数中也用到了寄存器dx，所以我们同时需要修改save（代码8.4）。

代码 8.4 修改 save 和 sys_call（节自 chapter8/a/kernel/kernel.asm）

```
311 ; ====================================================================================
312 ;                                              save
313 ; ====================================================================================
314 save:
315         pushad          ; `.
316         push    ds      ;  |
317         push    es      ;  | 保存原寄存器值
318         push    fs      ;  |
319         push    gs      ; /
320
321         ;; 注意，从这里开始，一直到 'mov esp, StackTop'，中间坚决不能用 push/pop 指令，
322         ;; 因为当前 esp 指向 proc_table 里的某个位置，push 会破坏掉进程表，导致灾难性后果！
323
324         mov     esi, edx        ; 保存 edx，因为 edx 里保存了系统调用的参数
325                                 ; (没用栈，而是用了另一个寄存器 esi)
326         mov     dx, ss
327         mov     ds, dx
328         mov     es, dx
329         mov     fs, dx
330
331         mov     edx, esi        ; 恢复 edx
332
333         mov     esi, esp                    ;esi = 进程表起始地址
334
335         inc     dword [k_reenter]           ;k_reenter++;
336         cmp     dword [k_reenter], 0        ;if(k_reenter ==0)
337         jne     .1                          ;{
338         mov     esp, StackTop               ;  mov esp, StackTop <--切换到内核栈
339         push    restart                     ;  push restart
340         jmp     [esi + RETADR - P_STACKBASE];  return;
341 .1:                                         ;} else { 已经在内核栈，不需要再切换
342         push    restart_reenter             ;  push restart_reenter
343         jmp     [esi + RETADR - P_STACKBASE];  return;
344                                             ;}
345
346
347 ; ====================================================================================
348 ;                                          sys_call
349 ; ====================================================================================
350 sys_call:
351         call    save
352
353         sti
354         push    esi
355
356         push    dword [p_proc_ready]
357         push    edx
358         push    ecx
359         push    ebx
360         call    [sys_call_table + eax * 4]
361         add     esp, 4 * 4
362
363         pop     esi
364         mov     [esi + EAXREG - P_STACKBASE], eax
365         cli
366
367         ret
```

sys_sendrec()这个函数被设计得相当简单，它可以描述为：把SEND消息交给msg_send()处理，把RECEIVE消息交给msg_receive()处理。

msg_send()和msg_receive()这两个函数我们过一会儿细细分解，先来看看之前没出现过的assert()和panic()。这两个函数虽然起的是辅助作用，但绝对不是可有可无，因为在我们接下来要处理的消息收发中，有一些编

程细节还真容易让人迷糊，这时候assert()就大显神威了，它会在错误被放大之前通知你。panic()的作用也类似，用于通知你发生了严重的错误。

8.3.1 assert()和panic()

先来看assert()。你或许早就开始使用这个函数，但之前你使用的都是现成的assert，只要包含一个头文件，就可以方便地使用。如今什么都得自力更生了，不过不用怕，写一个assert函数并非难事，见代码8.5。

代码 8.5 assert（chapter8/a/include/const.h）

```
#define ASSERT
#ifdef ASSERT
void assertion_failure(char *exp, char *file, char *base_file, int line);
#define assert(exp)  if (exp) ; \
        else assertion_failure(#exp, __FILE__, __BASE_FILE__, __LINE__)

#else
#define assert(exp)
#endif
```

注意其中的__FILE__、__BASE_FILE__和__LINE__这三个宏，它们的意义如下[2]：

- __FILE__ 将被展开成当前输入的文件。在这里，它告诉我们哪个文件中产生了异常。
- __BASE_FILE__ 可被认为是传递给编译器的那个文件名。比如你在m.c中包含了n.h，而n.h中的某一个assert函数失败了，则__FILE__为n.h，__BASE_FILE__为m.c。
- __LINE__ 将被展开成当前的行号。

明白了这几个宏的意义，剩下的assertion_failure()这个函数就显得容易了，它的作用就是将错误发生的位置打印出来（代码8.6）。

代码 8.6 assertion_failure（chapter8/a/lib/misc.c）

```
PUBLIC void assertion_failure(char *exp, char *file, char *base_file, int line)
{
        printl("%c  assert(%s) failed: file: %s, base_file: %s, ln%d",
               MAG_CH_ASSERT,
               exp, file, base_file, line);

        /**
         * If assertion fails in a TASK, the system will halt before
         * printl() returns. If it happens in a USER PROC, printl() will
         * return like a common routine and arrive here.
         * @see sys_printx()
         *
         * We use a forever loop to prevent the proc from going on:
         */
        spin("assertion_failure()");

        /* should never arrive here */
        __asm__ __volatile__("ud2");
}
```

[2]更详细的解释可参考 GCC 官方文档之 The C Preprocessor。

注意这里使用了一点点小伎俩，那就是使用了一个改进后的打印函数，叫做printl()，它其实就是一个定义成printf的宏，不过这里的printf跟上一章中的稍有不同，它将调用一个叫做printx的系统调用，并最终调用函数sys_printx()，它位于tty.c中（代码8.7）。

代码 8.7 sys_printx（chapter8/a/kernel/tty.c）

```
181 PUBLIC int sys_printx(int _unused1, int _unused2, char* s, struct proc* p_proc)
182 {
183         const char * p;
184         char ch;
185
186         char reenter_err[] = "? k_reenter is incorrect for unknown reason";
187         reenter_err[0] = MAG_CH_PANIC;
188
189     /**
190      * @note Code in both Ring 0 and Ring 1~3 may invoke printx().
191      * If this happens in Ring 0, no linear-physical address mapping
192      * is needed.
193      *
194      * @attention The value of 'k_reenter' is tricky here. When
195      *   -# printx() is called in Ring 0
196      *      - k_reenter > 0. When code in Ring 0 calls printx(),
197      *        an 'interrupt re-enter' will occur (printx() generates
198      *        a software interrupt). Thus 'k_reenter' will be increased
199      *        by 'kernel.asm::save' and be greater than 0.
200      *   -# printx() is called in Ring 1~3
201      *      - k_reenter == 0.
202      */
203     if (k_reenter == 0)  /* printx() called in Ring<1~3> */
204         p = va2la(proc2pid(p_proc), s);
205     else if (k_reenter > 0) /* printx() called in Ring<0> */
206         p = s;
207     else    /* this should NOT happen */
208         p = reenter_err;
209
210     /**
211      * @note if assertion fails in any TASK, the system will be halted;
212      * if it fails in a USER PROC, it'll return like any normal syscall
213      * does.
214      */
215     if ((*p == MAG_CH_PANIC) ||
216         (*p == MAG_CH_ASSERT && p_proc_ready < &proc_table[NR_TASKS])) {
217         disable_int();
218         char * v = (char*)V_MEM_BASE;
219         const char * q = p + 1; /* +1: skip the magic char */
220
221         while (v < (char*)(V_MEM_BASE + V_MEM_SIZE)) {
222             *v++ = *q++;
223             *v++ = RED_CHAR;
224             if (!*q) {
225                 while (((int)v - V_MEM_BASE) % (SCR_WIDTH * 16)) {
226                     /* *v++ = ' '; */
227                     v++;
228                     *v++ = GRAY_CHAR;
229                 }
230                 q = p + 1;
231             }
232         }
233
234         __asm__ __volatile__("hlt");
235     }
236
237     while ((ch = *p++) != 0) {
238         if (ch == MAG_CH_PANIC || ch == MAG_CH_ASSERT)
239             continue; /* skip the magic char */
```

```
240                 out_char(tty_table[p_proc->nr_tty].p_console, ch);
241         }
242
243         return 0;
244 }
245
```

容易看到，sys_printx()将首先判断首字符是否为预先设定的“Magic Char”，如果是的话，则做响应的特殊处理。我们的assertion_failure()就使用了MAG_CH_ASSERT作为“Magic Char”。当sys_printx()发现传入字符串的第一个字符是MAG_CH_ASSERT时，会同时判断调用系统调用的进程是系统进程（TASK）还是用户进程（USER PROC），如果是系统进程，则停止整个系统的运转，并将要打印的字符串打印在显存的各处；如果是用户进程，则打印之后像一个普通的printx调用一样返回，届时该用户进程会因为assertion_failure()中对函数spin()的调用而进入死循环。换言之，系统进程的assert失败会导致系统停转，用户进程的失败仅仅使自己停转。

到这里读者应该很清楚assert()函数的实现方法了，我们不妨来试验一下，在系统进程 TTY 中添加一句“assert(0);”，运行，读者将看到如图8.3所示的画面。再在用户进程 TestC 中添加一句“assert(0);”，将看到如图8.4所示的画面。

图 8.3 系统任务中 assert 失败

panic()跟assert()类似，也用到了sys_printx()和“Magic Char”，不过它要更简单一些，见代码8.8。

代码 8.8 panic（chapter8/a/kernel/main.c）

```
159 PUBLIC void panic(const char *fmt, ...)
160 {
161         int i;
162         char buf[256];
163
164         /* 4 is the size of fmt in the stack */
165         va_list arg = (va_list)((char*)&fmt + 4);
166
167         i = vsprintf(buf, fmt, arg);
168
```

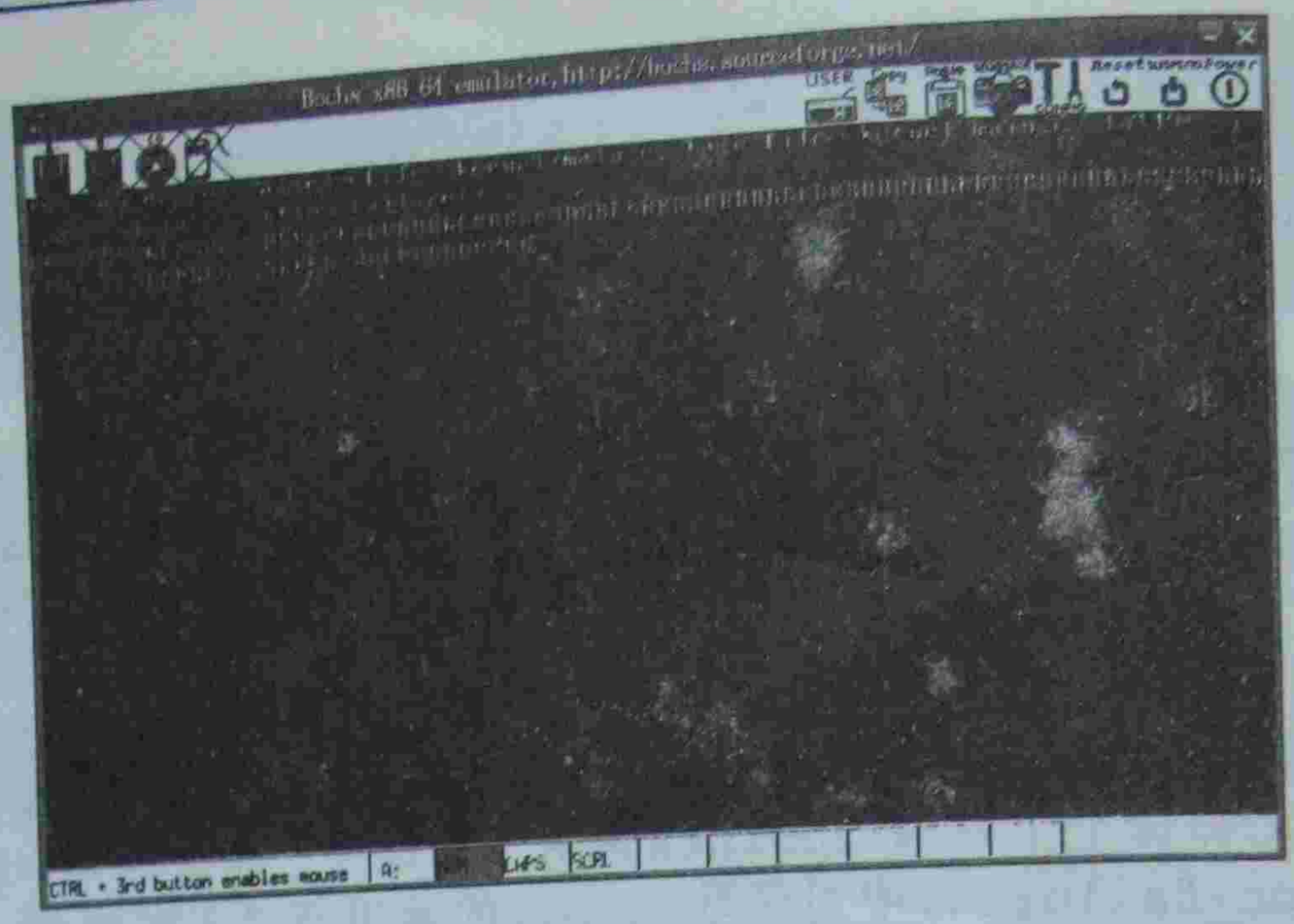


图 8.4 用户进程中 assert 失败

```
        printl("%c %!panic!! %s", MAG_CH_PANIC, buf);

        /* should never arrive here */
        __asm__ __volatile__("ud2");
}
```

由于panic只会用在系统任务所处的 Ring1 或 Ring0，所以sys_printx()遇到MAG_CH_PANIC就直接叫停整个系统，因为我们使用panic的时候，必是发生了严重错误的时候。

我们同样可以在 TTY 中试验一下panic的效果，比如添加这么一行：

```
panic("in TTY");
```

运行，会看到如图8.5所示的效果。

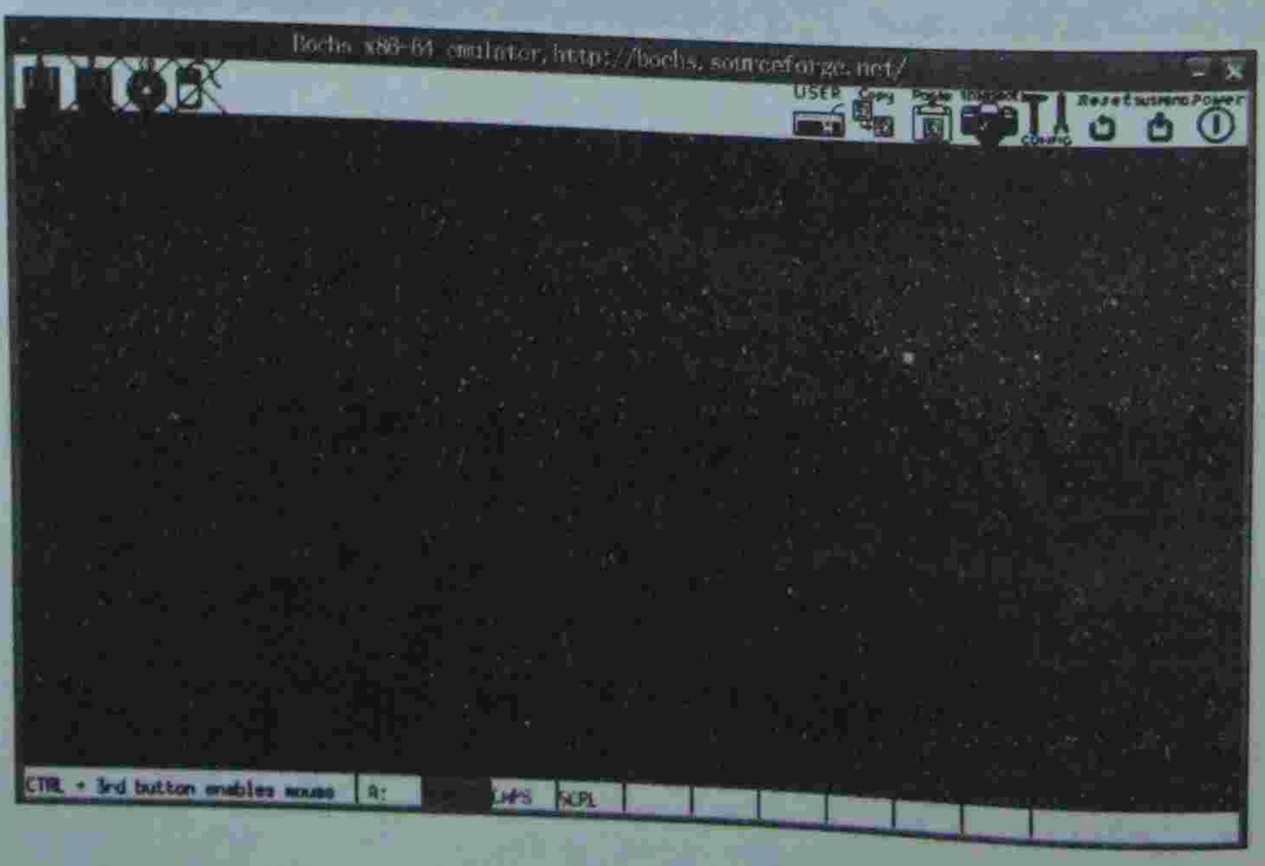


图 8.5 panic

在我们接下来的代码中，很多地方用到了assert()和panic()，其实有些地方完全可以不用这两个函数，而是以返回值的形式向上层函数传递的，但使用assert()和panic()可以减少代码量，并在第一时间通知我们哪里出了问题，作为一个试验性的操作系统，笔者认为这样做比使用某种方法来“消除”错误还要好。

8.3.2 msg_send()和msg_receive()

话题岔开这么久，让我们回到代码8.3，既然关键的函数是msg_send()和msg_receive()，那我们就来看一下，见代码8.9，它们是IPC的核心代码。

代码 8.9 msg_send 和 msg_receive（chapter8/a/kernel/proc.c）

```
/*****************************************************************************
 *                                ldt_seg_linear
 *****************************************************************************/
/**
 * <Ring 0~1> Calculate the linear address of a certain segment of a given
 * proc.
 *
 * @param p   Whose (the proc ptr).
 * @param idx Which (one proc has more than one segments).
 *
 * @return  The required linear address.
 *****************************************************************************/
PUBLIC int ldt_seg_linear(struct proc* p, int idx)
{
	struct descriptor * d = &p->ldts[idx];

	return d->base_high << 24 | d->base_mid << 16 | d->base_low;
}

/*****************************************************************************
 *                                va2la
 *****************************************************************************/
/**
 * <Ring 0~1> Virtual addr --> Linear addr.
 *
 * @param pid  PID of the proc whose address is to be calculated.
 * @param va   Virtual address.
 *
 * @return The linear address for the given virtual address.
 *****************************************************************************/
PUBLIC void* va2la(int pid, void* va)
{
	struct proc* p = &proc_table[pid];

	u32 seg_base = ldt_seg_linear(p, INDEX_LDT_RW);
	u32 la = seg_base + (u32)va;

	if (pid < NR_TASKS + NR_PROCS) {
		assert(la == (u32)va);
	}

	return (void*)la;
}

/*****************************************************************************
 *                                reset_msg
 *****************************************************************************/
/**
```

```
 * <Ring 0~3> Clear up a MESSAGE by setting each byte to 0.
 *
 * @param p  The message to be cleared.
 *****************************************************************************/
PUBLIC void reset_msg(MESSAGE* p)
{
        memset(p, 0, sizeof(MESSAGE));
}

/*****************************************************************************
 *                                block
 *****************************************************************************/
/**
 * <Ring 0> This routine is called after 'p_flags' has been set (!= 0), it
 * calls 'schedule()' to choose another proc as the 'proc_ready'.
 *
 * @attention This routine does not change 'p_flags'. Make sure the 'p_flags'
 * of the proc to be blocked has been set properly.
 *
 * @param p The proc to be blocked.
 *****************************************************************************/
PRIVATE void block(struct proc* p)
{
        assert(p->p_flags);
        schedule();
}

/*****************************************************************************
 *                                unblock
 *****************************************************************************/
/**
 * <Ring 0> This is a dummy routine. It does nothing actually. When it is
 * called, the 'p_flags' should have been cleared (== 0).
 *
 * @param p The unblocked proc.
 *****************************************************************************/
PRIVATE void unblock(struct proc* p)
{
        assert(p->p_flags == 0);
}

/*****************************************************************************
 *                                deadlock
 *****************************************************************************/
/**
 * <Ring 0> Check whether it is safe to send a message from src to dest.
 * The routine will detect if the messaging graph contains a cycle. For
 * instance, if we have procs trying to send messages like this:
 * A -> B -> C -> A, then a deadlock occurs, because all of them will
 * wait forever. If no cycles detected, it is considered as safe.
 *
 * @param src   Who wants to send message.
 * @param dest  To whom the message is sent.
 *
 * @return Zero if success.
 *****************************************************************************/
PRIVATE int deadlock(int src, int dest)
{
        struct proc* p = proc_table + dest;
        while (1) {
                if (p->p_flags & SENDING) {
                        if (p->p_sendto == src) {
                                /* print the chain */
                                p = proc_table + dest;
                                printl("=_=%s", p->name);
                                do {
                                        assert(p->p_msg);
```

```
                                p = proc_table + p->p_sendto;
                                printl("->%s", p->name);
                            } while (p != proc_table + src);
                            printl("=_=");

                            return 1;
                        }
                        p = proc_table + p->p_sendto;
                    }
                    else {
                        break;
                    }
                }
                return 0;
}

/*****************************************************************************
 *                                msg_send
 *****************************************************************************/
/**
 * <Ring 0> Send a message to the dest proc. If dest is blocked waiting for
 * the message, copy the message to it and unblock dest. Otherwise the caller
 * will be blocked and appended to the dest's sending queue.
 *
 * @param current  The caller, the sender.
 * @param dest     To whom the message is sent.
 * @param m        The message.
 *
 * @return Zero if success.
 *****************************************************************************/
PRIVATE int msg_send(struct proc* current, int dest, MESSAGE* m)
{
        struct proc* sender = current;
        struct proc* p_dest = proc_table + dest; /* proc dest */

        assert(proc2pid(sender) != dest);

        /* check for deadlock here */
        if (deadlock(proc2pid(sender), dest)) {
                panic(">>DEADLOCK<<␣%s->%s", sender->name, p_dest->name);
        }

        if ((p_dest->p_flags & RECEIVING) && /* dest is waiting for the msg */
            (p_dest->p_recvfrom == proc2pid(sender) ||
             p_dest->p_recvfrom == ANY)) {
                assert(p_dest->p_msg);
                assert(m);

                phys_copy(va2la(dest, p_dest->p_msg),
                          va2la(proc2pid(sender), m),
                          sizeof(MESSAGE));
                p_dest->p_msg = 0;
                p_dest->p_flags &= ~RECEIVING; /* dest has received the msg */
                p_dest->p_recvfrom = NO_TASK;
                unblock(p_dest);

                assert(p_dest->p_flags == 0);
                assert(p_dest->p_msg == 0);
                assert(p_dest->p_recvfrom == NO_TASK);
                assert(p_dest->p_sendto == NO_TASK);
                assert(sender->p_flags == 0);
                assert(sender->p_msg == 0);
                assert(sender->p_recvfrom == NO_TASK);
                assert(sender->p_sendto == NO_TASK);
        }
        else { /* dest is not waiting for the msg */
                sender->p_flags |= SENDING;
```

```
327             assert(sender->p_flags == SENDING);
328             sender->p_sendto = dest;
329             sender->p_msg = m;
330
331             /* append to the sending queue */
332             struct proc * p;
333             if (p_dest->q_sending) {
334                     p = p_dest->q_sending;
335                     while (p->next_sending)
336                             p = p->next_sending;
337                     p->next_sending = sender;
338
339             }
340             else {
341                     p_dest->q_sending = sender;
342                     }
343                     sender->next_sending = 0;
344
345                     block(sender);
346
347                     assert(sender->p_flags == SENDING);
348                     assert(sender->p_msg != 0);
349                     assert(sender->p_recvfrom == NO_TASK);
350                     assert(sender->p_sendto == dest);
351             }
352
353             return 0;
354     }
355
356     /*****************************************************************************
357      *                                msg_receive
358      *****************************************************************************/
359     /**
360      * <Ring 0> Try to get a message from the src proc. If src is blocked sending
361      * the message, copy the message from it and unblock src. Otherwise the caller
362      * will be blocked.
363      *
364      * @param current The caller, the proc who wanna receive.
365      * @param src     From whom the message will be received.
366      * @param m       The message ptr to accept the message.
367      *
368      * @return  Zero if success.
369      *****************************************************************************/
370     PRIVATE int msg_receive(struct proc* current, int src, MESSAGE* m)
371     {
372             struct proc* p_who_wanna_recv = current; /**
373                                                       * This name is a little bit
374                                                       * wierd, but it makes me
375                                                       * think clearly, so I keep
376                                                       * it.
377                                                       */
378             struct proc* p_from = 0; /* from which the message will be fetched */
379             struct proc* prev = 0;
380             int copyok = 0;
381
382             assert(proc2pid(p_who_wanna_recv) != src);
383
384             if ((p_who_wanna_recv->has_int_msg) &&
385                 ((src == ANY) || (src == INTERRUPT))) {
386                     /* There is an interrupt needs p_who_wanna_recv's handling and
387                      * p_who_wanna_recv is ready to handle it.
388                      */
389
390                     MESSAGE msg;
391                     reset_msg(&msg);
392                     msg.source = INTERRUPT;
393                     msg.type = HARD_INT;
```

```
            assert(m);
            phys_copy(va2la(proc2pid(p_who_wanna_recv), m), &msg,
                      sizeof(MESSAGE));

            p_who_wanna_recv->has_int_msg = 0;

            assert(p_who_wanna_recv->p_flags == 0);
            assert(p_who_wanna_recv->p_msg == 0);
            assert(p_who_wanna_recv->p_sendto == NO_TASK);
            assert(p_who_wanna_recv->has_int_msg == 0);

            return 0;
    }


    /* Arrives here if no interrupt for p_who_wanna_recv. */
    if (src == ANY) {
            /* p_who_wanna_recv is ready to receive messages from
             * ANY proc, we'll check the sending queue and pick the
             * first proc in it.
             */
            if (p_who_wanna_recv->q_sending) {
                    p_from = p_who_wanna_recv->q_sending;
                    copyok = 1;

                    assert(p_who_wanna_recv->p_flags == 0);
                    assert(p_who_wanna_recv->p_msg == 0);
                    assert(p_who_wanna_recv->p_recvfrom == NO_TASK);
                    assert(p_who_wanna_recv->p_sendto == NO_TASK);
                    assert(p_who_wanna_recv->q_sending != 0);
                    assert(p_from->p_flags == SENDING);
                    assert(p_from->p_msg != 0);
                    assert(p_from->p_recvfrom == NO_TASK);
                    assert(p_from->p_sendto == proc2pid(p_who_wanna_recv));
            }
    }
    else {
            /* p_who_wanna_recv wants to receive a message from
             * a certain proc: src.
             */
            p_from = &proc_table[src];

            if ((p_from->p_flags & SENDING) &&
                (p_from->p_sendto == proc2pid(p_who_wanna_recv))) {
                    /* Perfect, src is sending a message to
                     * p_who_wanna_recv.
                     */
                    copyok = 1;

                    struct proc* p = p_who_wanna_recv->q_sending;
                    assert(p); /* p_from must have been appended to the
                                * queue, so the queue must not be NULL
                                */
                    while (p) {
                            assert(p_from->p_flags & SENDING);
                            if (proc2pid(p) == src) { /* if p is the one */
                                    p_from = p;
                                    break;
                            }
                            prev = p;
                            p = p->next_sending;
                    }

                    assert(p_who_wanna_recv->p_flags == 0);
                    assert(p_who_wanna_recv->p_msg == 0);
                    assert(p_who_wanna_recv->p_recvfrom == NO_TASK);
                    assert(p_who_wanna_recv->p_sendto == NO_TASK);
```

```
                assert(p_who_wanna_recv->q_sending != 0);
                assert(p_from->p_flags == SENDING);
                assert(p_from->p_msg != 0);
                assert(p_from->p_recvfrom == NO_TASK);
                assert(p_from->p_sendto == proc2pid(p_who_wanna_recv));
        }
    }

    if (copyok) {
        /* It's determined from which proc the message will
         * be copied. Note that this proc must have been
         * waiting for this moment in the queue, so we should
         * remove it from the queue.
         */
        if (p_from == p_who_wanna_recv->q_sending) { /* the 1st one */
            assert(prev == 0);
            p_who_wanna_recv->q_sending = p_from->next_sending;
            p_from->next_sending = 0;
        }
        else {
            assert(prev);
            prev->next_sending = p_from->next_sending;
            p_from->next_sending = 0;
        }

        assert(m);
        assert(p_from->p_msg);
        /* copy the message */
        phys_copy(va2la(proc2pid(p_who_wanna_recv), m),
                  va2la(proc2pid(p_from), p_from->p_msg),
                  sizeof(MESSAGE));

        p_from->p_msg = 0;
        p_from->p_sendto = NO_TASK;
        p_from->p_flags &= ~SENDING;
        unblock(p_from);
    }
    else {  /* nobody's sending any msg */
        /* Set p_flags so that p_who_wanna_recv will not
         * be scheduled until it is unblocked.
         */
        p_who_wanna_recv->p_flags |= RECEIVING;

        p_who_wanna_recv->p_msg = m;

        if (src == ANY)
            p_who_wanna_recv->p_recvfrom = ANY;
        else
            p_who_wanna_recv->p_recvfrom = proc2pid(p_from);

        block(p_who_wanna_recv);

        assert(p_who_wanna_recv->p_flags == RECEIVING);
        assert(p_who_wanna_recv->p_msg != 0);
        assert(p_who_wanna_recv->p_recvfrom != NO_TASK);
        assert(p_who_wanna_recv->p_sendto == NO_TASK);
        assert(p_who_wanna_recv->has_int_msg == 0);
    }

    return 0;
}
```

围绕msg_send()和msg_receive()，代码中还列出了其他几个必要的函数，它们是：

- **ldt_seg_linear()** 每个进程都有自己的LDT，位于进程表的中间，这

个函数就是根据 LDT 中描述符的索引来求得描述符所指向的段的基地址。

- **va2la()** 用来由虚拟地址求线性地址，它用到了ldt_seg_linear()。
- **reset_msg()** 用于把一个消息的每个字节清零。
- **block()** 阻塞一个进程。
- **unblock()** 解除一个进程的阻塞。
- **deadlock()** 简单地判断是否发生死锁。方法是判断消息的发送是否构成一个环，如果构成环则意味着发生死锁，比如 A 试图发消息给 B，同时 B 试图给 C，C 试图给 A 发消息，那么死锁就发生了，因为 A、B 和 C 三个进程都将无限等待下去（如图8.6所示）。

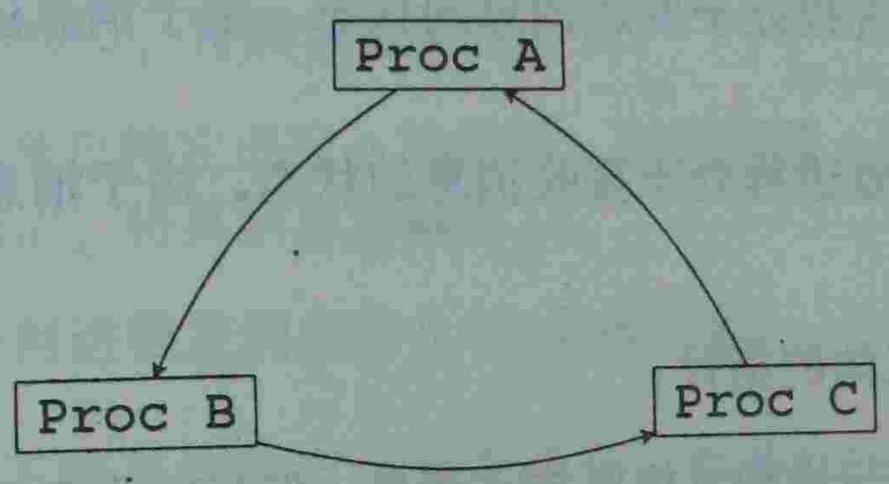


图 8.6 发消息时可能发生的死锁

在block()、unblock()和deadlock()中，都出现了struct proc这个结构体的一个新成员：p_flag。其实增加的新成员还有几个，见代码8.10。

代码 8.10 进程表的新成员（chapter8/a/include/proc.h）

```
struct proc {
        struct stackframe regs;          /* process registers saved in stack frame */

        u16 ldt_sel;                     /* gdt selector giving ldt base and limit */
        struct descriptor ldts[LDT_SIZE]; /* local descs for code and data */

        int ticks;                       /* remained ticks */
        int priority;

        u32 pid;                         /* process id passed in from MM */
        char name[16];                   /* name of the process */

        int  p_flags;                    /**
                                          * process flags.
                                          * A proc is runnable iff p_flags==0
                                          */


        MESSAGE * p_msg;
        int p_recvfrom;
        int p_sendto;

        int has_int_msg;                 /**
                                          * nonzero if an INTERRUPT occurred when
                                          * the task is not ready to deal with it.
                                          */
```

```
⇒ 57        struct proc * q_sending;    /**
                                         * queue of procs sending messages to
                                         * this proc
                                         */
58
59
60          struct proc * next_sending;/**
⇒ 61                                     * next proc in the sending
                                         * queue (q_sending)
62                                       */
63
64
65          int nr_tty;
66
67  };
```

所有增加的这些成员都是跟消息机制有关的。

- **p_flags** 用于标明进程的状态。目前它的取值可以有三种：
  - 0 进程正在运行或准备运行。
  - **SENDING** 进程处于发送消息的状态。由于消息还未送达，进程被阻塞。
  - **RECEIVING** 进程处于接收消息的状态。由于消息还未收到，进程被阻塞。
- **p_msg** 指向消息体的指针。
- **p_recvfrom** 假设进程 P 想要接收消息，但目前没有进程发消息给它，本成员记录 P 想要从谁那里接收消息。
- **p_sendto** 假设进程 P 想要发送消息，但目前没有进程接收它，本成员记录 P 想要发送消息给谁。
- **has_int_msg** 如果有一个中断需要某进程来处理，或者换句话说，某进程正在等待一个中断发生——比如硬盘驱动可能会等待硬盘中断的发生，系统在得知中断发生后会将此位置为 1。
- **q_sending** 如果有若干进程——比如 A、B 和 C——都向同一个进程 P 发送消息，而 P 此时并未准备接收消息，那么 A、B 和 C 将会排成一个队列。进程 P 的 q_sending 指向第一个试图发送消息的进程。
- **next_sending** 试图发送消息的 A、B 和 C（依时间顺序）三进程排成的队列的实现方式是这样的：目的进程 P 的进程表的q_sending指向 A，进程 A 的进程表的next_sending指向 B，进程 B 的进程表的next_sending指向 C，进程 C 的进程表的next_sending指向空。

假设有进程 A 想要向 B 发送消息 M，那么过程将会是这样的：

1. A 首先准备好 M。
2. A 通过系统调用sendrec，最终调用msg_send。
3. 简单判断是否发生死锁。
4. 判断目标进程 B 是否正在等待来自 A 的消息：

- 如果是：消息被复制给 B，B 被解除阻塞，继续运行；
- 如果否：A 被阻塞，并被加入到 B 的发送队列中。

假设有进程 B 想要接收消息（来自特定进程、中断或者任意进程），那么过程将会是这样的：

1. B 准备一个空的消息结构体 M，用于接收消息。
2. B 通过系统调用sendrec，最终调用msg_receive。
3. 判断 B 是否有个来自硬件的消息（通过has_int_msg），如果是，并且 B 准备接收来自中断的消息或准备接收任意消息，则马上准备一个消息给 B，并返回。
4. 如果 B 想接收来自任意进程的消息，则从自己的发送队列中选取第一个（如果队列非空的话），将其消息复制给 M。
5. 如果 B 是想接收来自特定进程 A 的消息，则先判断 A 是否正在等待向 B 发送消息，若是的话，将其消息复制给 M。
6. 如果此时没有任何进程发消息给 B，B 会被阻塞。

值得说明的是，不管是接收方还是发送方，都各自维护一个消息结构体，只不过发送方的结构体是携带了消息内容的而接收方是空的。由于我们使用同步 IPC，一方的需求——发送或接收——只有被满足之后才会继续运行，所以操作系统不需要维护任何的消息缓冲，实现起来也就相对简单。

8.3.3 增加消息机制之后的进程调度

在上一节中我们提到，如今的每个进程增加了两种可能的状态：SENDING 和RECEIVING。相应的，我们需要在进程调度的时候区别对待了。凡是处于SENDING或RECEIVING状态的进程，我们就不再让它们获得 CPU 了，也就是说，将它们“阻塞”了。这也解释了为什么block()和unblock()两个函数本质上没做任何工作——一个进程是否阻塞，已经由进程表中的p_flags项决定，我们不需要额外做什么工作。不过我们还是应该保留这两个函数，一方面将来可能要扩展它们，另一方面它们也有助于理清编程的思路。

代码8.11就是修改后的调度函数。

代码 8.11 增加消息机制之后的进程调度（chapter8/a/kernel/proc.c）

```
PUBLIC void schedule()
{
    struct proc*    p;
    int             greatest_ticks = 0;

    while (!greatest_ticks) {
        for (p = &FIRST_PROC; p <= &LAST_PROC; p++) {
            if (p->p_flags == 0) {
                if (p->ticks > greatest_ticks) {
                    greatest_ticks = p->ticks;
                    p_proc_ready = p;
                }
```

```
                }
            }
            if (!greatest_ticks)
                for (p = &FIRST_PROC; p <= &LAST_PROC; p++)
                    if (p->p_flags == 0)
                        p->ticks = p->priority;

        }
}
```

可以看到，当且仅当p_flags为零时，一个进程才可能获得运行的机会。

8.4 使用 IPC 来替换系统调用get_ticks

到这里我们的消息机制已经可以用了，如果读者亲自实践的话，别忘了一些细枝末节的东西，比如在初始化进程时给新增加的进程表成员赋值，再比如增加一些必要的函数声明以及修改Makefile等零碎工作。

为验证消息机制是否工作正常，我们还是从最简单的工作着手，删掉原先的系统调用get_ticks，用收发消息的方法重新实现之。

不过且慢，既然是收发消息，必然是有两方参与。想想便知，我们需要一个系统进程来接收用户进程的消息，并且返回ticks值。我们就来建立一个新的系统进程，就叫它“SYSTASK”。

添加一个任务的工作还是按照第6.4.6节中所述步骤进行。它的主循环如代码8.12所示。

代码 8.12 系统进程（chapter8/a/kernel/systask.c）

```
/*****************************************************************************
 *                                task_sys
 *****************************************************************************/
/**
 * <Ring 1> The main loop of TASK SYS.
 *
 *****************************************************************************/
PUBLIC void task_sys()
{
        MESSAGE msg;
        while (1) {
                send_recv(RECEIVE, ANY, &msg);
                int src = msg.source;

                switch (msg.type) {
                case GET_TICKS:
                        msg.RETVAL = ticks;
                        send_recv(SEND, src, &msg);
                        break;
                default:
                        panic("unknown msg type");
                        break;
                }
        }
}
```

代码很简单，不过要留心一下其中用到的函数send_recv()，它其实就是把sendrec这个系统调用给封装了一下，见代码8.13。

代码 8.13 send_recv（chapter8/a/kernel/proc.c）

```
/*****************************************************************************
 *                                send_recv
 *****************************************************************************/
/**
 * <Ring 1~3> IPC syscall.
 *
 * It is an encapsulation of 'sendrec',
 * invoking 'sendrec' directly should be avoided
 *
 * @param function  SEND, RECEIVE or BOTH
 * @param src_dest  The caller's proc_nr
 * @param msg       Pointer to the MESSAGE struct
 *
 * @return always 0.
 *****************************************************************************/
PUBLIC int send_recv(int function, int src_dest, MESSAGE* msg)
{
        int ret = 0;

        if (function == RECEIVE)
                memset(msg, 0, sizeof(MESSAGE));

        switch (function) {
        case BOTH:
                ret = sendrec(SEND, src_dest, msg);
                if (ret == 0)
                        ret = sendrec(RECEIVE, src_dest, msg);
                break;
        case SEND:
        case RECEIVE:
                ret = sendrec(function, src_dest, msg);
                break;
        default:
                assert((function == BOTH) ||
                       (function == SEND) || (function == RECEIVE));
                break;
        }

        return ret;
}
```

我们知道，一个完整的系统调用需要一个来回，那就是用户进程向内核请求一个东西，然后内核返回给它。我们用消息机制来实现这个过程同样需要一个来回，这意味着用户进程发送一个消息之后需要马上等待接收一个消息，以便收到内核（其实是某个系统任务）给它的返回值。这个发送然后马上接收的行为被send_recv()这个函数包装了一下，并在SEND和RECEIVE之外又提供了一个叫做BOTH的消息类型。今后我们想要收发消息时，就直接使用这个send_recv()，而不再直接使用系统调用sendrec。

好了，系统进程 SYSTASK 已经就绪，下面就来修改一下函数get_ticks（代码8.14）。

代码 8.14 get_ticks（chapter8/a/kernel/main.c）

```
PUBLIC int get_ticks()
{
        MESSAGE msg;
        reset_msg(&msg);
        msg.type = GET_TICKS;
        send_recv(BOTH, TASK_SYS, &msg);
        return msg.RETVAL;
}
```

```
...
void TestA()
{
        while (1) {
                printf("<Ticks:%d>", get_ticks());
                milli_delay(200);
        }
}
```

我们以GET_TICKS为消息类型，不夹带其他任何信息地传递给 SYSTASK，SYSTASK 收到这个消息之后，把当前的ticks值放入消息并发给用户进程，用户进程会接收到它，完成整个任务。

我们来运行一下，结果如图8.7所示。

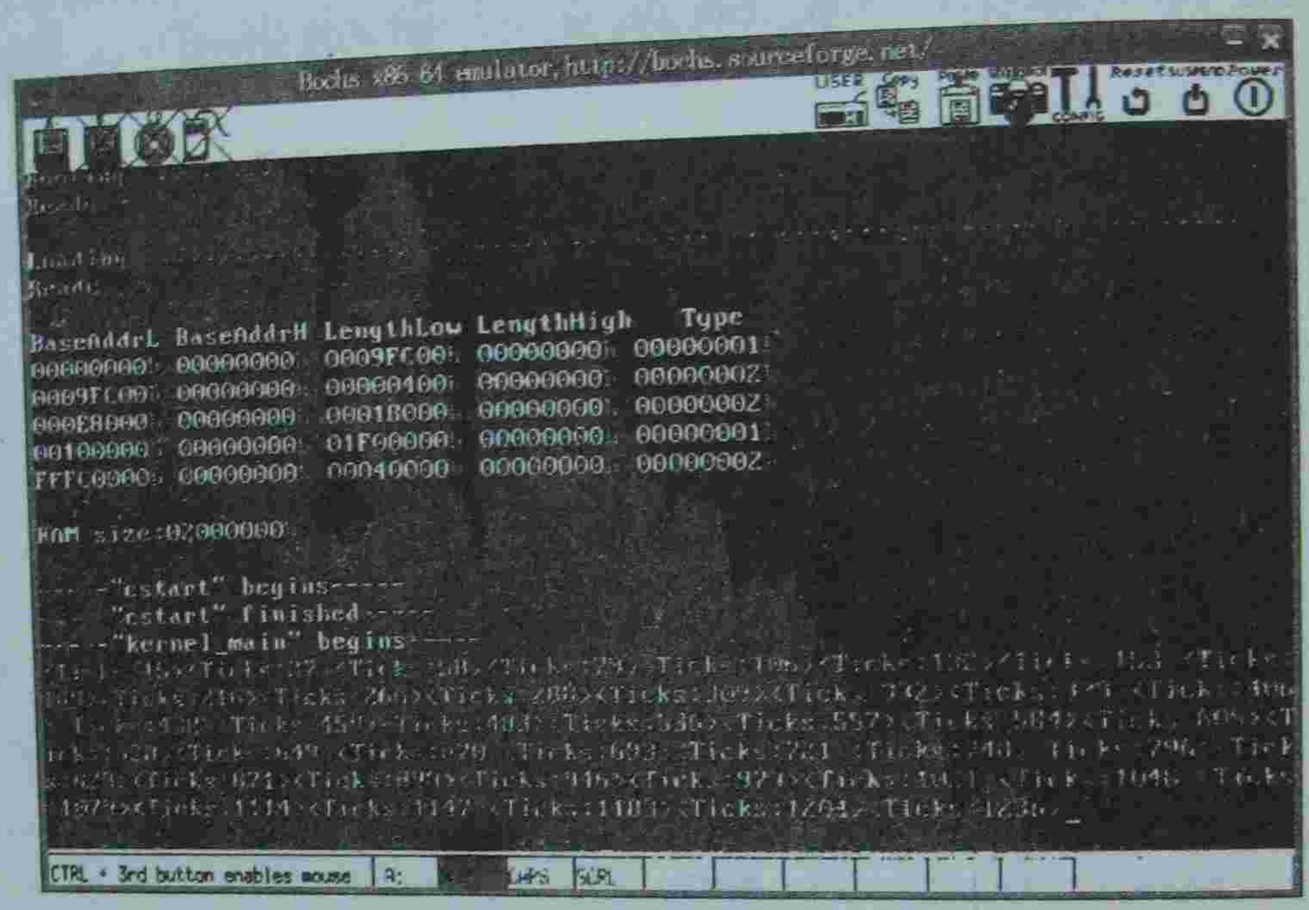


图 8.7 使用 IPC 实现 get_ticks

成功了！进程 TestA 调用get_ticks之后，成功地打印出了它们的值，这表明我们的消息机制工作良好！

8.5 总结

虽然运行结果没有很大改变，但是如今我们的操作系统已经确立了微内核的路线，并且成功地实现了 IPC，即便这算不上是一个质的飞跃，至少我们已经走上了另一个台阶。接下来，基于消息机制，我们将逐步实现硬盘驱动程序、文件系统等内容。而且你将逐步发现微内核的优点，那就是代码相对很独立，结构很清晰，并且内核态的代码今后将很少需要大的改动了。

People who are very quick are apt to be too sure. Slow and steady win the race.

—— *Aesop (6th C.B.C.)*

9 文件系统

在本章中，让我们来实现一个非常简单的文件系统。这个文件系统将建立在真正的硬盘之上——它不是个假惺惺的 RAM Disk，它通过硬盘驱动程序直接操作硬盘。在编写这个文件系统的过程中，我们上一章所实现的 IPC 机制将大显神通，你会发现，通过几个消息，用户进程、文件系统和驱动程序之间可以方便地协同工作。顺带着，我们还将建立一个驱动程序框架的雏形。

9.1 硬盘简介

既然我们将对硬盘进行操作，我们需要先来认识一下硬盘。虽然读者想必已经比较熟悉硬盘，但对于一些术语，我们还是有必要先来澄清一下。

PC 发展到今天，每一个部分细讲起来都是个很长的故事，无论其发展历史还是技术细节，都一言难尽，硬盘也是如此。更加令人迷惑的是，硬盘标准中有太多的术语和名词，让人眼花缭乱，甚至不知所云。在这里，笔者试图用比较简略的文字，将一些重要的术语描述清楚，当然，读者若想了解所有细节，可能需要翻阅一些标准文档，最重要的文档目前由 T13 技术委员会维护，他们的官方网站位于http://www.t13.org。

很久以前，硬盘控制器和硬盘本身是分离的，直到有一天，西数（Western Digital）、康柏、CDC 的一个事业部 Imprimis（后来被希捷收购）等合作开发了一种新的接口标准，将硬盘控制器和硬盘合在一起，这一标准被称为 IDE (Integrated Drive Electronics）或者 ATA（AT Attachment，AT 指的是 IBM 的 PC/AT 个人计算机)。事实上 ATA 这一名称更加“正确”，它是接口的“真正的”名字，而 IDE 这一名称仅仅是在当时区分开了那些控制器和硬盘分离的设计，它不是个标准的名称。不过 IDE 这个名称被使用得非常广泛，或许比 ATA 流传还广，有些人干脆称这一接口为 IDE/ATA 或者 ATA/IDE。后来 ATA 被标准化，今天我们称之为 ATA-1。

再后来 ATA 标准不断发展，陆续出现了 ATA-2、ATA-3、ATA/ATAPI-4、ATA/ATAPI-5、ATA/ATAPI-6，以及 ATA/ATAPI-7。所谓 ATAPI，其实是 CD-ROM 的接口的真正名称，它是 AT Attachment Packet Interface 的缩写。ATAPI 使得硬盘和 CD-ROM 的接口统一起来。

你或许还听过其他一些名词，诸如EIDE、FASTATA、FASTATA-2、ULTRA ATA等，不要被迷惑，所有这些都是为了让产品听上去比较有新意而起的名字，也就是说，它们都是营销手段，或者是遵循某个标准的产品别名。它们并不是另外的标准。

你或许还听过“温盘”这个名称，它所指的温彻斯特硬盘（Winchester Disk）其实是1973年IBM一款硬盘驱动器的代号。我们现在的硬盘都是直接或间接建立在温彻斯特技术（Winchester Technology）之上。了解这一事实很重要，因为在一些操作系统的代码中，仍使用Winchester这一名称指代硬盘驱动器。

近几年，一种新的标准出现了，它就是SATA（Serial ATA），或被称为“串口硬盘”，而且它正在逐渐在取代原先的ATA。为了区分开来，之前的硬盘则被称为“并口硬盘”，原先ATA的叫法也相应地改为PATA（Parallel ATA）。ATA/ATAPI-7的文档中包含了SATA的内容。从我们想要编写相当低级的驱动程序的角度来看，由于SATA通常兼容PATA的操作方式，所以暂时地，我们姑且认为所有的硬盘都是无差别的ATA硬盘。

如果此刻你忍不住下载了ATA/ATAPI的技术文档的话，千万不要被吓怕了，标准内容虽多，我们能用到的却很少，我们马上就来逐一介绍。

9.2 硬盘操作的I/O端口

跟我们之前接触过的键盘控制器、VGA控制器等硬件类似，对硬盘控制器的操作仍是通过I/O端口来进行，这些端口分为两组，它们对应命令块寄存器（Command Block Registers）和控制块寄存器（Control Block Register），如表9.1所示。

表9.1 硬盘I/O端口及寄存器

组别	I/O端口		读时	写时
	Primary	Secondary		
Command Block Registers	1F0h	170h	Data	Data
	1F1h	171h	Error	Features
	1F2h	172h	Sector Count	Sector Count
	1F3h	173h	LBA Low	LBA Low
	1F4h	174h	LBA Mid	LBA Mid
	1F5h	175h	LBA High	LBA High
	1F6h	176h	Device	Device
	1F7h	177h	Status	Command
Control Block Register	3F6h	376h	Alternate Status	Device Control

表9.1中的Primary和Secondary指的是ATA接口通道（Channel），通俗地说就是主板上的IDE口。一个普通的PC主板上通常有两个IDE口，分别对应两个IDE通道：Primary和Secondary，它们有时也被标注为IDE0和IDE1。每个

IDE 通道又能连接两个设备，称为主设备（Master）和从设备（Slave）。对不同的 IDE 通道的访问是通过 I/O 端口来区分的，对同一 IDE 通道上的主从设备的访问是通过 Device 寄存器上的第 4 位的值来区分的——第 4 位为 0 时操作主设备，为 1 时操作从设备。事实上一个机器不只允许有两个 IDE 通道，但超过两个的情况非常罕见，在此不做介绍。在本书中，我们只考虑硬盘接在 Primary 通道的情况。

对硬盘的操作并不复杂，只需先往命令块寄存器（Command Block Registers）写入正确的值，再通过控制块寄存器（Control Block Register）发送命令就可以了。当然或许每个命令会有不同的编程细节，我们在遇见具体情况时再做介绍。

9.3 硬盘驱动程序

驱动程序的作用在于隐藏硬件细节，向上层进程提供统一的接口。由于我们的进程通过收发消息来相互通信，那么驱动程序的接口自然也是消息了。所以只要我们定义了驱动程序可以接收什么消息，也就定义了驱动程序的接口。为简单起见，我们先只定义一种消息：DEV_OPEN。我们过会儿通过 FS 任务向硬盘驱动程序发送一个DEV_OPEN消息。可是硬盘驱动程序收到这个消息之后干点什么呢？我们还是先干点简单工作：向硬盘驱动器发送一个 IDENTIFY 命令，这个命令可用来获取硬盘参数。

向硬盘发送 IDENTIFY 命令很简单，只需要通过 Device 寄存器的第 4 位指定驱动器——0 表示 Master，1 表示 Slave——然后往 Command 寄存器写入十六进制ECh就可以。硬盘准备好参数之后，会产生一个中断，这时我们就可以通过 Data 寄存器读取数据了。参数有很多，总共是 256 个字（WORD），在下面的程序中，我们仅仅取出其中的几个值来显示。我们来看代码9.1。

代码 9.1 硬盘驱动（chapter9/a/kernel/hd.c）

```
/*****************************************************************************
 *                                task_hd
 *****************************************************************************/
/**
 * Main loop of HD driver.
 *
 *****************************************************************************/
PUBLIC void task_hd()
{
	MESSAGE msg;

	init_hd();

	while (1) {
		send_recv(RECEIVE, ANY, &msg);

		int src = msg.source;

		switch (msg.type) {
		case DEV_OPEN:
			hd_identify(0);
			break;

		default:
			dump_msg("HD driver::unknown msg", &msg);
```

```
                spin("FS::main_loop (invalid msg.type)");
                break;
            }

            send_recv(SEND, src, &msg);
        }
    }

/*****************************************************************************
 *                                init_hd
 *****************************************************************************/
/**
 * <Ring 1> Check hard drive, set IRQ handler, enable IRQ and initialize data
 *          structures.
 *****************************************************************************/
PRIVATE void init_hd()
{
        /* Get the number of drives from the BIOS data area */
        u8 * pNrDrives = (u8*)(0x475);
        printl("NrDrives:%d.\n", *pNrDrives);
        assert(*pNrDrives);

        put_irq_handler(AT_WINI_IRQ, hd_handler);
        enable_irq(CASCADE_IRQ);
        enable_irq(AT_WINI_IRQ);
}

/*****************************************************************************
 *                                hd_identify
 *****************************************************************************/
/**
 * <Ring 1> Get the disk information.
 *
 * @param drive  Drive Nr.
 *****************************************************************************/
PRIVATE void hd_identify(int drive)
{
        struct hd_cmd cmd;
        cmd.device  = MAKE_DEVICE_REG(0, drive, 0);
        cmd.command = ATA_IDENTIFY;
        hd_cmd_out(&cmd);
        interrupt_wait();
        port_read(REG_DATA, hdbuf, SECTOR_SIZE);

        print_identify_info((u16*)hdbuf);
}

/*****************************************************************************
 *                                print_identify_info
 *****************************************************************************/
/**
 * <Ring 1> Print the hdinfo retrieved via ATA_IDENTIFY command.
 *
 * @param hdinfo  The buffer read from the disk i/o port.
 *****************************************************************************/
PRIVATE void print_identify_info(u16* hdinfo)
{
        int i, k;
        char s[64];

        struct iden_info_ascii {
                int idx;
                int len;
                char * desc;
        } iinfo[] = {{10, 20, "HD SN"}, /* Serial number in ASCII */
                     {27, 40, "HD Model"} /* Model number in ASCII */ };
```

```
        for (k = 0; k < sizeof(iinfo)/sizeof(iinfo[0]); k++) {
                char * p = (char*)&hdinfo[iinfo[k].idx];
                for (i = 0; i < iinfo[k].len/2; i++) {
                        s[i*2+1] = *p++;
                        s[i*2] = *p++;
                }
                s[i*2] = 0;
                printl("%s: %s\n", iinfo[k].desc, s);
        }

        int capabilities = hdinfo[49];
        printl("LBA supported: %s\n",
                (capabilities & 0x0200) ? "Yes" : "No");

        int cmd_set_supported = hdinfo[83];
        printl("LBA48 supported: %s\n",
                (cmd_set_supported & 0x0400) ? "Yes" : "No");

        int sectors = ((int)hdinfo[61] << 16) + hdinfo[60];
        printl("HD size: %dMB\n", sectors * 512 / 1000000);
}

/*****************************************************************************
                                hd_cmd_out
 *****************************************************************************/
/**
 * <Ring 1> Output a command to HD controller.
 *
 * @param cmd  The command struct ptr.
 *****************************************************************************/
PRIVATE void hd_cmd_out(struct hd_cmd* cmd)
{
        /**
         * For all commands, the host must first check if BSY=1,
         * and should proceed no further unless and until BSY=0
         */
        if (!waitfor(STATUS_BSY, 0, HD_TIMEOUT))
                panic("hd error.");

        /* Activate the Interrupt Enable (nIEN) bit */
        out_byte(REG_DEV_CTRL, 0);
        /* Load required parameters in the Command Block Registers */
        out_byte(REG_FEATURES, cmd->features);
        out_byte(REG_NSECTOR,  cmd->count);
        out_byte(REG_LBA_LOW,  cmd->lba_low);
        out_byte(REG_LBA_MID,  cmd->lba_mid);
        out_byte(REG_LBA_HIGH, cmd->lba_high);
        out_byte(REG_DEVICE,   cmd->device);
        /* Write the command code to the Command Register */
        out_byte(REG_CMD,      cmd->command);
}

/*****************************************************************************
                                interrupt_wait
 *****************************************************************************/
/**
 * <Ring 1> Wait until a disk interrupt occurs.
 *
 *****************************************************************************/
PRIVATE void interrupt_wait()
{
        MESSAGE msg;
        send_recv(RECEIVE, INTERRUPT, &msg);
}

/*****************************************************************************
                                waitfor
```

```
192  *****************************************************************************/
193 /**
194  * <Ring 1> Wait for a certain status.
195  *
196  * @param mask    Status mask.
197  * @param val     Required status.
198  * @param timeout Timeout in milliseconds.
199  *
200  * @return One if sucess, zero if timeout.
201  *****************************************************************************/
202 PRIVATE int waitfor(int mask, int val, int timeout)
203 {
204         int t = get_ticks();
205
206         while(((get_ticks() - t) * 1000 / HZ) < timeout)
207                 if ((in_byte(REG_STATUS) & mask) == val)
208                         return 1;
209
210         return 0;
211 }
212
213 /*****************************************************************************
214  *                                hd_handler
215  *****************************************************************************/
216 /**
217  * <Ring 0> Interrupt handler.
218  *
219  * @param irq  IRQ nr of the disk interrupt.
220  *****************************************************************************/
221 PUBLIC void hd_handler(int irq)
222 {
223         /*
224          * Interrupts are cleared when the host
225          *   - reads the Status Register,
226          *   - issues a reset, or
227          *   - writes to the Command Register.
228          */
229         hd_status = in_byte(REG_STATUS);
230
231         inform_int(TASK_HD);
232 }
```

在代码9.1中总共有这么几个函数：

- task_hd()
- init_hd()
- hd_identify()
- print_identify_info()
- hd_cmd_out()
- interrupt_wait()
- waitfor()
- hd_handler()

task_hd()是硬盘驱动的主循环，它跟我们之前的任务没什么两样。开头调用init_hd()做一些初始化工作，然后开始不停地处理消息。我们目前只接

收DEV_OPEN消息，收到它之后将会调用hd_identify()来获取并打印部分硬盘参数。

init_hd()中所做的工作分两部分。它先是从物理地址0x475处获取系统内硬盘数量——这个地址是由 BIOS 指定的。然后它指定hd_handler()为硬盘中断处理程序，并且打开硬盘中断。注意，这里不仅仅需要打开硬盘对应的中断信号线，还要打开主 8259A 用于级联从片的信号线，读者若不理解的话，看看图3.39便一目了然。

hd_identify()便是获取硬盘参数的函数了。我们先来说说 Device 寄存器的格式，如图9.1所示。

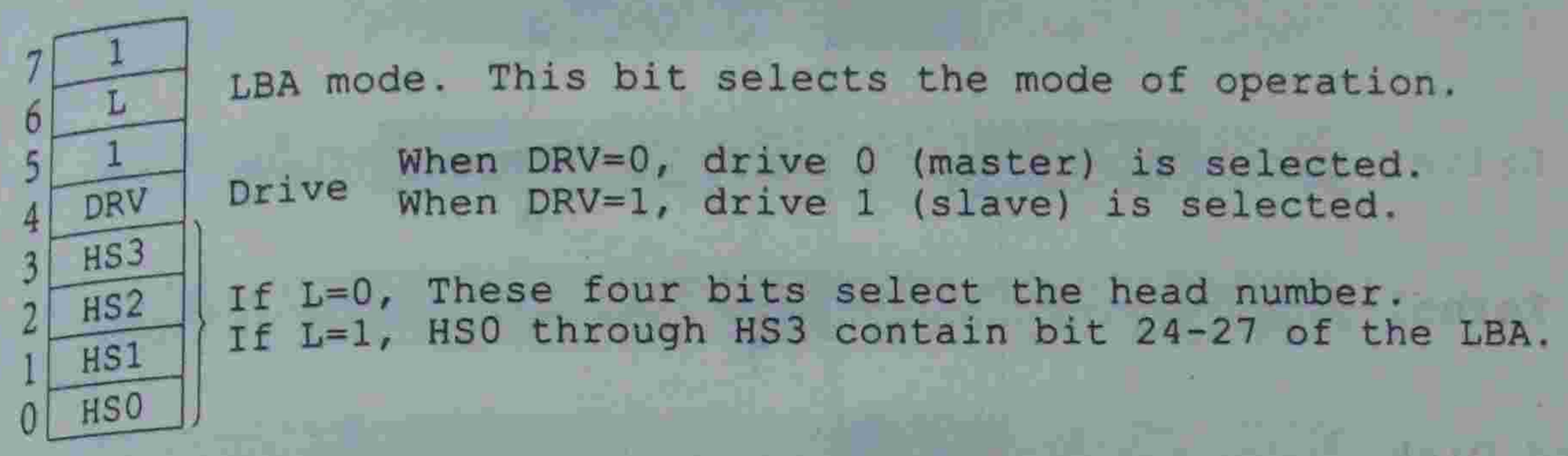


图 9.1 Device Register

可以看到，寄存器主要有三部分：LBA 模式位、DRV 位和低四位。我们之前已经提到两次，DRV 位（第 4 位）用于指定主盘或从盘，0 表示主盘，1 表示从盘。LBA 模式位用于指定操作模式，当此位为 0 时，对磁盘的操作使用 CHS 模式，也即用“柱面 / 磁头 / 扇区号”来定位扇区；当此位为 1 时，对磁盘的操作使用 LBA（Logical Block Address）模式。由于现代硬盘都支持 LBA 模式，在本书中我们都使用 LBA 模式。寄存器的低四位，在 CHS 模式中表示磁头号，在 LBA 模式中表示 LBA 的 24 到 27 位。也就是说，LBA 的地址被拆分成了四部分，第 0 到 7 位由 LBA Low 寄存器指定，第 8 到 15 位由 LBA Mid 寄存器指定，第 16 到 23 位由 LBA High 寄存器指定。整个 LBA 是 28 位的，经过计算可得知，2^{28}=268435456，这么多个扇区意味着 128GB 的容量，这也是 28 位 LBA 的最大寻址能力。

可是目前市面上的硬盘容量早就超过 128GB 了，怎么突破限制呢？原理也很简单，每次读写的时候，三个寄存器——LBA Low、LBA Mid、LBA High——每个使用两次，这样就相当于有了六个寄存器来存储地址，这就是 LBA48 (ATA-6 开始支持)。48 位 LBA 的寻址上限为 128PB，我们觉得并希望它是足够了，但谁知道呢，或许不久以后每个寄存器将不得不使用三次——我们已经见到太多例子，人们不停在想办法突破自己之前设下的限制，但当下一次做决定时，还是会低估技术的发展速度。

好了言归正传，由于 Device 这个寄存器有这么一点点复杂，我们于是用一个宏来合成它，见代码9.2。

代码 9.2 MAKE_DEVICE_REG（chapter9/a/include/hd.h）

```
/* for DEVICE register. */
#define MAKE_DEVICE_REG(lba,drv,lba_highest) (((lba) << 6) |          \
                                              ((drv) << 4) |          \
                                              (lba_highest & 0xF) | 0xA0)
```

我们在代码9.1第96行用到了这个宏。

在接下来的第98行，我们用了一个函数hd_cmd_out()来进行实际的向硬盘驱动器发送命令的工作，因为今后我们还会用到它。当发送命令之后，我们用interrupt_wait()这一函数来等待中断的发生，而等待的方法是调用了一个接收消息的函数（第187行）。说到这里，读者或许需要翻一翻上一章的代码。我们在进行消息处理时，有一种单独的消息来源叫做INTERRUPT，在这里就用到了。之所以发生中断后还要通过消息来通知驱动程序，是因为驱动程序是运行在非内核态的独立的进程，而发生中断这一事件是独立于进程的。驱动程序获知中断发生的情形可参考图9.2，图中列出了两种情形。

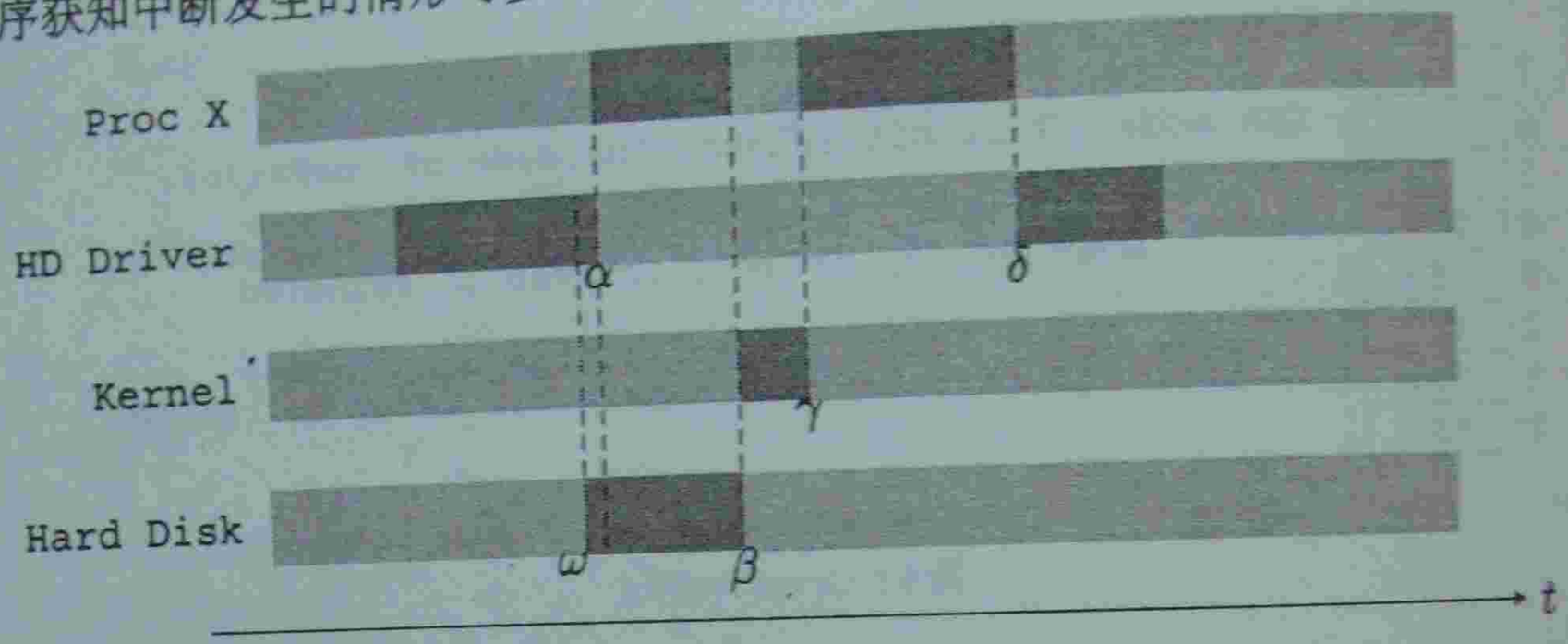


图 9.2 驱动程序如何获知硬盘中断

第一种情况中各时间点/段的状况是这样的：

ω 硬盘驱动调用hd_cmd_out()。

ω~α 硬盘已处在工作状态，同时硬盘驱动也在运行——执行hd_cmd_out()到send_recv()之间的少量代码。

α 硬盘驱动调用send_recv()，由于此时硬盘尚未完成工作，所以被阻塞，控制权于是交给 Proc X。

α~β Proc X 运行中，同时硬盘也在工作。

β 硬盘完成工作，并触发中断，执行中断处理程序hd_handler()（它应被视为内核的一部分，因为它工作在 Ring0），它最终调用inform_int()，这会解除硬盘驱动的阻塞。

γ hd_handler()结束，控制权回到 Proc X 手中。

δ Proc X 的时间片用完，调度程序将控制权交给硬盘驱动（它早在β时已被解除阻塞）。

第二种情况是这样的：

α' 硬盘驱动执行完hd_cmd_out()之后便由调度程序将控制权交给另一个进程 Proc X。

α'~β' Proc X 运行的同时，硬盘在工作。

β' 硬盘完成工作，并触发中断，执行中断处理程序hd_handler()，它最终调用inform_int()。

γ' hd_handler()结束，控制权还给 Proc X。

δ' Proc X 的时间片用完，调度程序将控制权交给硬盘驱动。

α' 硬盘驱动执行send_recv()得到消息，并继续执行。

这两种情况的区别在于，第一种情况调用send_recv()时无法立刻返回，因为那时硬盘还没有完成任务，而第二种情况调用send_recv()时直接返回，因为硬盘已经把数据准备好了。不管是哪种情况，从驱动程序的角度来看，都是等待一个中断的发生，而且它应该能够等到。

当中断发生以后，接下来的第99行从 DATA 端口读取数据，这样硬盘参数就得到了。

接下来的print_identify_info()选取了其中几个参数打印了出来。它们的偏移和解释见表9.2，如果读者有兴趣，可从 T13 网站上下载 AT Attachment with Packet Interface 文档以获得更详细的参数列表。

到这里我们获取硬盘参数并打印的整体流程已经很清楚了，下面我们就来看一看用来向硬盘发送命令的函数hd_cmd_out()。在它的一开头先判断硬盘 Status 寄存器（请参考图9.3）的 BSY 位，因为只有这一位为零时我们才能行动。若 BSY 为零则先通过 Device Control 寄存器（请参考图9.4）打开中断，再依次写 Features、Sector Count、LBA Low、LBA Mid、LBA High、Device 和 Command 等寄存器。一旦写入 Command 寄存器，命令便被执行了。函数hd_cmd_out()接受的参数类型为一结构体，它里面保存的是我们要写入的寄存器应有的值。

代码9.1第221行处是一个只有两行代码的硬盘中断处理程序，它先是通过读 Status 寄存器来恢复中断响应，然后调用inform_int()来通知驱动程序。

表9.2 通过IDENTIFY命令得到的硬盘参数（节选）

偏移	描述
10~19	序列号(20个ASCII字符)
27~46	型号(40个ASCII字符)
60~61	用户可用的最大扇区数
49	功能（Capabilities） (bit 9为1表示支持LBA)
83	支持的命令集 (bit 10为1表示支持48位寻址)

```
7  BSY     Busy. If BSY=1, no other bits in the register are valid.
6  DRDY    Drive Ready.
5  DF/SE   Device Fault / Stream Error.
4  #       Command dependent. (formerly DSC bit)
3  DRQ     Data Request. (ready to transfer data)
2  -       Obsolete.
1  -       Obsolete.
0  ERR     Error. (an error occurred)
```

图9.3 Status Register

```
7  HOB     High Order Byte (defined by 48-bit Address feature set).
6  -
5  -
4  -
3  -
2  SRST    Software Rest.
1  -IEN    Interrupt Enable.
0  0
```

图9.4 Device Control Register

再次提醒读者，硬盘中断处理程序虽然也放在hd.c中，但它跟其他代码有着本质的不同。中断处理程序运行在Ring0，是内核的一部分，而其他代码都是驱动程序进程运行时调用，都运行在Ring1。另外，从图3.39可以看到，硬盘的中断信号线是连在8259A从片上的，这是我们第一次用到从片，而第6章中我们只是编写了处理主片的宏（代码6.56），现在我们就照葫芦画瓢写一个处理从片的宏，见代码9.3。

代码9.3 hwint_slave（chapter9/a/kernel/kernel.asm）

```
204 %macro  hwint_slave     1
205         call    save
206         in      al, INT_S_CTLMASK       ; .
207         or      al, (1 << (%1 - 8))     ; | 屏蔽当前中断
208         out     INT_S_CTLMASK, al       ; /
209         mov     al, EOI                 ; .
210         out     INT_M_CTL, al           ; | 置EOI位(master)
211         nop                             ; /
212         out     INT_S_CTL, al           ; .
                                            ; | 置EOI位(slave)
                                            ; / 一定注意：slave和master都要置EOI
```

```
        sti     ; CPU在响应中断的过程中会自动关中断，这句之后就允许响应新的中断
        push    %1                              ; \.
        call    [irq_table + 4 * %1]            ;  | 中断处理程序
        pop     ecx                             ; /
        cli
        in      al, INT_S_CTLMASK               ; \.
        and     al, ~(1 << (%1 - 8))            ;  | 恢复接受当前中断
        out     INT_S_CTLMASK, al               ; /
        ret
endmacro
```

一定要记得，从片连接的设备发生中断之后，从片和主片都要置EOI，忘掉主片的话，硬盘就再也响应不了中断了。

好了，硬盘驱动程序暂时就这些内容了，我们来看看使用它的FS进程，见代码9.4。

代码 9.4 新建立的文件系统进程（chapter9/a/fs/main.c）

```
/*****************************************************************************
                                  task_fs
 *****************************************************************************/
/**
 * <Ring 1> The main loop of TASK FS.
 *
 *****************************************************************************/
PUBLIC void task_fs()
{
        printl("Task␣FS␣begins.\n");

        /* open the device: hard disk */
        MESSAGE driver_msg;
        driver_msg.type = DEV_OPEN;
        send_recv(BOTH, TASK_HD, &driver_msg);

        spin("FS");
}
```

可以看到，我们新建立的文件系统进程其实就是个空壳，它什么都不做，只是向硬盘驱动程序发送一个DEV_OPEN消息。跟之前的所有进程一样，FS进程会在系统启动时开始运行，所以我们在启动操作系统之后很快硬盘驱动就会接收到DEV_OPEN消息，并且获取硬盘参数并进行一些打印工作。

下面我们该运行一下看看了，不过我们的“机器”还没有硬盘呢，不要紧，马上造出一个硬盘：

```
▷ bximage
========================================================================
                                bximage
                  Disk Image Creation Tool for Bochs
        $Id: bximage.c,v 1.32 2006/06/16 07:29:33 vruppert Exp $
========================================================================

Do you want to create a floppy disk image or a hard disk image?
Please type hd or fd. [hd] ↩

What kind of image should I create?
Please type flat, sparse or growing. [flat] ↩

Enter the hard disk size in megabytes, between 1 and 129023
[10] 80 ↩

I will create a 'flat' hard disk image with
```

```
  cyl=162
  heads=16
  sectors per track=63
  total sectors=163296
  total size=79.73 megabytes

What should I name the image?
[c.img] 80m.img ↩

Writing: [] Done.

I wrote 83607552 bytes to 80m.img.

The following line should appear in your bochsrc:
  ata0-master: type=disk, path="80m.img", mode=flat, cylinders=162, heads=16,
spt=63
```

用bximage，顷刻之间，一块80MB的硬盘就造好了。更酷的是，bximage把应该把什么放进Bochs配置文件都给列出来了，我们马上修改bochsrc，相当于把新造好的硬盘插入机器。添加下面两行即可：

```
ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata0-master: type=disk, path="80m.img", mode=flat, cylinders=162, heads=16,
spt=63
```

下面我们就运行来看一看，见图9.5。

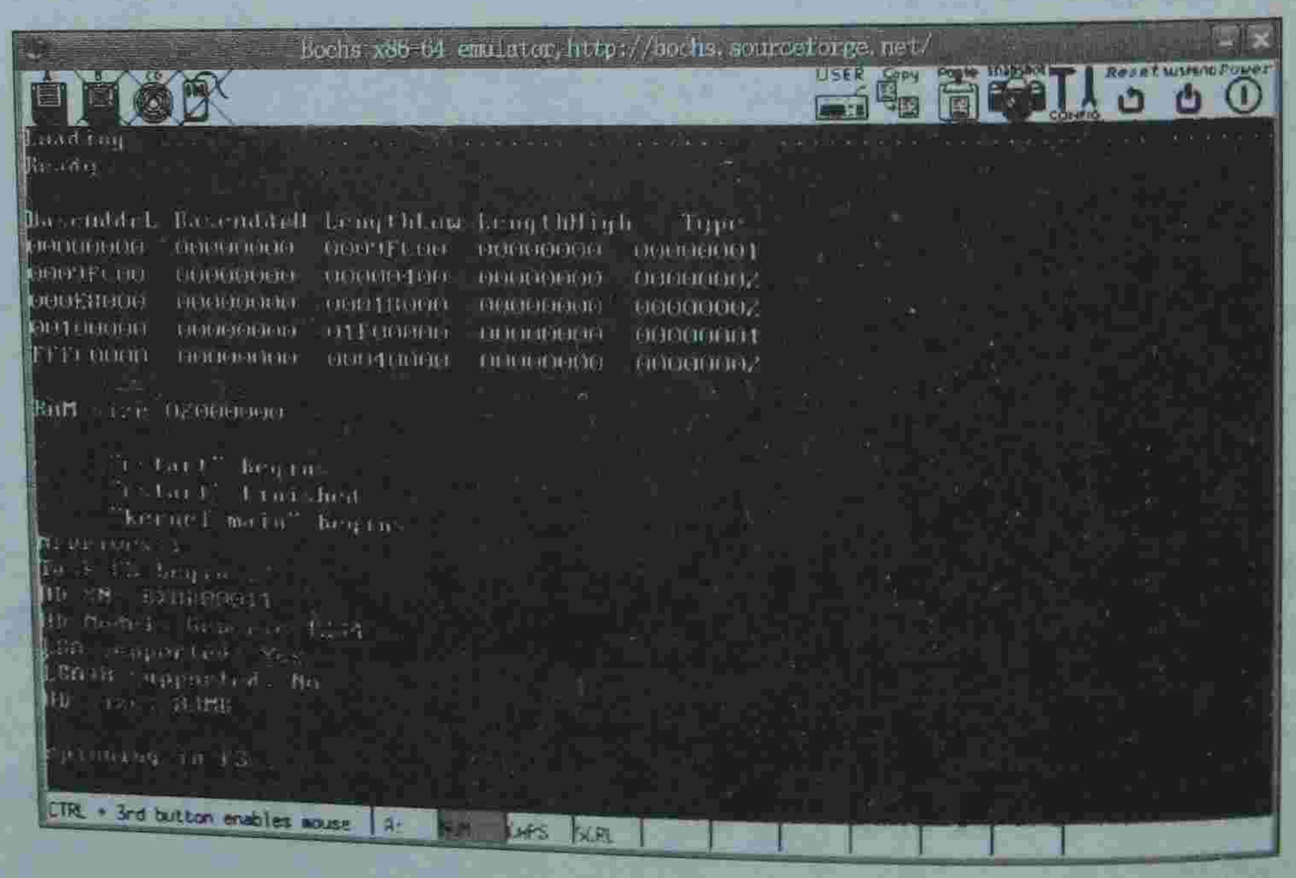


图9.5 用IDENTIFY命令得到硬盘参数

太酷了，我们的操作系统成功获取了硬盘参数：

```
HD SN: BXHD00011
HD Model: Generic 1234
LBA supported: Yes
LBA48 supported: No
HD size: 83MB
```

如果读者感兴趣，完全可以自己对照ATA文档，打印出更多硬盘参数。我至今记得当我第一次看到自己的操作系统打印出这些参数时是怎样的激动——小小的成果，都可以获得很大的满足感。

9.4 文件系统

激动之后，我们该静下心来仔细考虑如何构建一个文件系统了。这并不是我们第一次接触文件系统，我们在第 4 章的时候就研究过 FAT12。FAT12 算是很简单的文件系统了，既然我们已经比较熟悉它，就让我们结合它的结构来分析一下一个文件系统都需要哪些要素。

我们来参考图4.1，图分四个部分，分别是引导扇区、FAT 表、根目录区和数据区。其中引导扇区中不仅包含引导代码，而且包含 BPB（见表4.1），它包含诸如根目录文件数最大值之类的信息，可算是文件系统的 Metadata；FAT 表记录的是整个磁盘扇区的使用情况，有哪些扇区未被使用，以及每个文件占用哪些扇区等；根目录区则是文件的索引了，那里记录了文件的名称、属性等内容。

这么看来，一个简单的文件系统大致需要这么几个要素：

- 要有地方存放 Metadata；
- 要有地方记录扇区的使用情况；
- 要有地方来记录任一文件的信息，比如占用了哪些扇区等；
- 要有地方存放文件的索引。

这些要点不难理解，而且如果你分析其他文件系统的话，也基本是这些要素。与此同时，只要具备了这些要素，一个文件系统基本就可用了——至于好坏，那不是我们这样的初学者要考虑的问题。

好了，根据这些要素，同时参照 Minix 的文件系统，我们就把我们的文件系统设计成如图9.6所示的样子。

可以看到，总体上来看，它几乎是把前述的各要素一字排开：

- 要有地方存放 Metadata——占用整整一个扇区的 superblock；
- 要有地方记录扇区的使用情况——sector map；
- 要有地方来记录任一文件的信息，比如占用了哪些扇区等——inode map 以及被称作 inode_array 的 i-node 真正存放地；
- 要有地方存放文件的索引——root 数据区。

superblock 通常也叫超级块，关于文件系统的 Metadata 我们统统记在这里。sector map 是一个位图，它用来映射扇区的使用情况，用 1 表示扇区已被使用，0表示未使用。i-node 是 UNIX 世界各种文件系统的核心数据结构之一，我们把它借用过来。每个 i-node 对应一个文件，用于存放文件名、文件属性等内容，inode_array 就是把所有 i-node 都放在这里，形成一个较大的数组。而 inode map 就是用来映射 inode_array 这个数组使用情况的一个位图，用法跟 sector map 类似。root 数据区类似于 FAT12 的根目录区，但本质上它也是个普通文件，由于它是所有文件的索引，所以我们把它单独看待。为了简单起见，我们的文件系

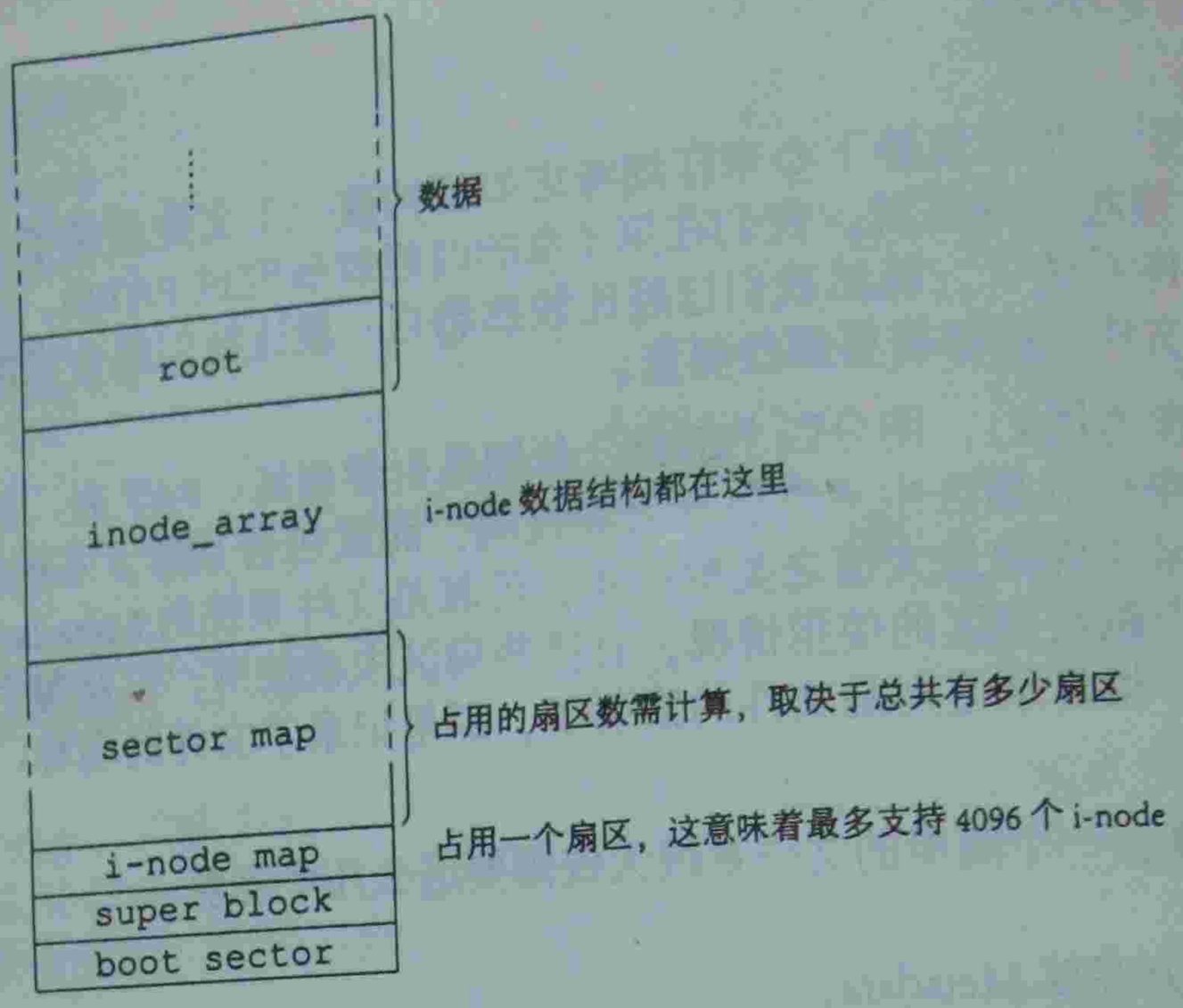


图 9.6 文件系统

统暂不支持文件夹，也就是说用来表示目录的特殊文件只有这么一个。这种不支持文件夹的文件系统，其实也不是我们的首创，历史上曾经有过，而且这种文件系统还有个名字，叫做扁平文件系统（Flat File System）。

至于引导扇区嘛，就让它纯粹用作引导吧，我们不打算学习 FAT12 把一些额外数据结构塞进去——512 字节已经够拥挤了，而如今的硬盘是如此的便宜。

轻轻松松，踩在前人的肩膀上，加上我们的偷工减料（刚开始当然要最简化），我们的文件系统就这样设计完成了。下面该是想想怎么将它放到硬盘上了。根据我们的经验，一个文件系统可以安装到硬盘的一个分区上，而且一块硬盘上可以有多个文件系统共存。那么，下面我们就来找个分区在它上面实现之，可是不忙，我们还不知道硬盘是怎么分区的呢。没关系，下面就来研究一下。

9.5 硬盘分区表

你可能比我还性急，会想为什么不把文件系统直接安装到整块硬盘上呢。我得承认那样做完全可以，而且简单易行。但是我有个想法，将来何不把我们辛苦实现的操作系统装到自己的计算机上呢？到时候稍微设置一下 Grub，实现多引导，让我们的操作系统跟 Linux，Windows 等并存，岂不美哉？所以在这里我们就多劳动一下，研究一下怎么来针对分区进行操作，要不然一下子用掉整块硬盘，又浪费，又不酷。

硬盘分区表其实是一个结构体数组，数组的每个成员是一个 16 字节的结构体，它的构成如表9.3所示。

这个数组位于引导扇区的1BEh处，共有四个成员——因为 IBM 当时觉得

表 9.3 分区表结构

偏移	长度	描述
0	1	状态（80h= 可引导，00h= 不可引导，其他 = 不合法）
1	1	起始磁头号
2	1	起始扇区号（仅用了低 6 位，高 2 位为起始柱面号的第 8,9 位）
3	1	起始柱面号的低 8 位
4	1	分区类型（System ID）
5	1	结束磁头号
6	1	结束扇区号（仅用了低 6 位，高 2 位为结束柱面号的第 8,9 位）
7	1	结束柱面号的低 8 位
8	4	起始扇区的 LBA
12	4	扇区数目

一台 PC 最多会装四个操作系统。现在我们的计算机中每块硬盘经常划分成不止四个分区，这是因为每个主分区可以进一步分成多个逻辑分区。具体的做法，我们还是需要一个示例来对照。虽然读者的 PC 可以作为现成的示例，但为了安全起见，我们还是操作映像，而不是真的硬盘。现在就让我们把刚才生成的硬盘分成几个区：

```
▷ /sbin/fdisk 80m.img
... ... ...
Command (m for help): x ↩ 进入 extra functionality 菜单

Expert command (m for help): c ↩ 设置柱面数
Number of cylinders (1-1048576): 162 ↩ 柱面数为 162

Expert command (m for help): h ↩ 设置磁头数
Number of heads (1-256, default 255): 16 ↩ 磁头数为 16

Expert command (m for help): r ↩ 回到主菜单

Command (m for help): n ↩ 新建分区
Command action
   e   extended
   p   primary partition (1-4)
p ↩ 主分区
Partition number (1-4): 1 ↩
First cylinder (1-162, default 1):↩
Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-162, default 162): 20 ↩

Command (m for help): n ↩ 新建分区
Command action
   e   extended
   p   primary partition (1-4)
e ↩ 扩展分区
Partition number (1-4): 2 ↩
First cylinder (21-162, default 21):↩
Using default value 21
Last cylinder or +size or +sizeM or +sizeK (21-162, default 162):↩
Using default value 162

Command (m for help): n ↩ 新建分区
Command action
   l   logical (5 or over)
   p   primary partition (1-4)
```

```
1 ↩ 逻辑分区
First cylinder (21-162, default 21):↩
Using default value 21
Last cylinder or +size or +sizeM or +sizeK (21-162, default 162): 60 ↩

Command (m for help): n ↩ 新建分区
Command action
   l   logical (5 or over)
   p   primary partition (1-4)
1 ↩ 逻辑分区
First cylinder (61-162, default 61):↩
Using default value 61
Last cylinder or +size or +sizeM or +sizeK (61-162, default 162): 90 ↩

Command (m for help): n ↩ 新建分区
Command action
   l   logical (5 or over)
   p   primary partition (1-4)
1 ↩ 逻辑分区
First cylinder (91-162, default 91):↩
Using default value 91
Last cylinder or +size or +sizeM or +sizeK (91-162, default 162): 132 ↩

Command (m for help): n ↩ 新建分区
Command action
   l   logical (5 or over)
   p   primary partition (1-4)
1 ↩ 逻辑分区
First cylinder (133-162, default 133):↩
Using default value 133
Last cylinder or +size or +sizeM or +sizeK (133-162, default 162): 160 ↩

Command (m for help): n ↩ 新建分区
Command action
   l   logical (5 or over)
   p   primary partition (1-4)
1 ↩ 逻辑分区
First cylinder (161-162, default 161):↩
Using default value 161
Last cylinder or +size or +sizeM or +sizeK (161-162, default 162):↩
Using default value 162

Command (m for help): p ↩ 打印分区表

Disk 80m.img: 0 MB, 0 bytes
16 heads, 63 sectors/track, 162 cylinders
Units = cylinders of 1008 * 512 = 516096 bytes

  Device Boot      Start         End      Blocks   Id  System
80m.img1               1          20       10048+  83  Linux
80m.img2              21         162       71568    5  Extended
⇒ 80m.img5            21          60       20128+  83  Linux
80m.img6              61          90       15088+  83  Linux
80m.img7              91         132       21136+  83  Linux
80m.img8             133         160       14080+  83  Linux
80m.img9             161         162         976+  83  Linux

Command (m for help): t ↩ 更改分区类型
Partition number (1-9): 5 ↩ 选 80m.img5
Hex code (type L to list codes): L ↩ 列出已知类型（受输出宽度影响，有些类型名被截断）

 0  Empty           1e  Hidden W95 FAT1 80  Old Minix       be  Solaris boot
 1  FAT12           24  NEC DOS         81  Minix / old Lin bf  Solaris
 2  XENIX root      39  Plan 9          82  Linux swap / So c1  DRDOS/sec
 3  XENIX usr       3c  PartitionMagic  83  Linux           c4  DRDOS/sec
 4  FAT16 <32M      40  Venix 80286     84  OS/2 hidden C:  c6  DRDOS/sec
 5  Extended        41  PPC PReP Boot   85  Linux extended  c7  Syrinx
```

```
 6  FAT16                42  SFS                 86  NTFS volume set da  Non-FS data
 7  HPFS/NTFS            4d  QNX4.x              87  NTFS volume set db  CP/M / CTOS
 8  AIX                  4e  QNX4.x 2nd part     88  Linux plaintext de  Dell Utility
 9  AIX bootable         4f  QNX4.x 3rd part     8e  Linux LVM           df  BootIt
 a  OS/2 Boot Manag      50  OnTrack DM          93  Amoeba              e1  DOS access
 b  W95 FAT32            51  OnTrack DM6 Aux     94  Amoeba BBT          e3  DOS R/O
 c  W95 FAT32 (LBA)      52  CP/M                9f  BSD/OS              e4  SpeedStor
 e  W95 FAT16 (LBA)      53  OnTrack DM6 Aux     a0  IBM Thinkpad hi    eb  BeOS fs
 f  W95 Ext'd (LBA)      54  OnTrackDM6          a5  FreeBSD             ee  EFI GPT
10  OPUS                 55  EZ-Drive            a6  OpenBSD             ef  EFI
11  Hidden FAT12         56  Golden Bow          a7  NeXTSTEP            f0  Linux/PA-RISC
12  Compaq diagnost      5c  Priam Edisk         a8  Darwin UFS          f1  SpeedStor
14  Hidden FAT16 <3      61  SpeedStor           a9  NetBSD              f4  SpeedStor
16  Hidden FAT16         63  GNU HURD or Sys     ab  Darwin boot         f2  DOS secondary
17  Hidden HPFS/NTF      64  Novell Netware      b7  BSDI fs             fd  Linux raid
18  AST SmartSleep       65  Novell Netware      b8  BSDI swap           fe  LANstep
1b  Hidden W95 FAT3      70  DiskSecure Mult     bb  Boot Wizard hid     ff  BBT
1c  Hidden W95 FAT3      75  PC/IX
Hex code (type L to list codes): 99 ↩ 将 99h 作为Orange'S FS的 system id
Changed system type of partition 5 to 99 (Unknown)

Command (m for help): a ↩ 设置可启动标志
Partition number (1-9): 5 ↩ 选 80m.img5

Command (m for help): p ↩ 再次打印分区表

Disk 80m.img: 0 MB, 0 bytes
16 heads, 63 sectors/track, 162 cylinders
Units = cylinders of 1008 * 512 = 516096 bytes

   Device Boot      Start         End      Blocks   Id  System
80m.img1               1          20       10048+  83  Linux
80m.img2              21         162       71568    5  Extended
⇒ 80m.img5   *        21          60       20128+  99  Unknown
80m.img6              61          90       15088+  83  Linux
80m.img7              91         132       21136+  83  Linux
80m.img8             133         160       14080+  83  Linux
80m.img9             161         162         976+  83  Linux

Command (m for help): w ↩ 写入“磁盘”
... ... ...
```

在这里我们把一个 80MB 的硬盘映像分成了一个主分区和一个扩展分区，扩展分区中又分成了五个逻辑分区。我们将来把Orange'S装在第一个逻辑分区上，也就是标为 80m.img5 的分区。我们先是把它的分区类型（System ID）改成99h，又为它设定了“可启动”标志。在设置分区类型时，我们先是列出了已知的类型，然后选定还未使用的99h作为我们文件系统的 System ID。

现在我们就来实际看一下分区表是什么样子的，用二进制查看器来看一下引导扇区：

```
▷ xxd -u -a -g 1 -c 16 -s 0 -l 512 80m.img
0000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00001b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01  ................
00001c0: 01 00 83 0F 3F 13 3F 00 00 00 81 4E 00 00 00 00  ....?.?....N....
00001d0: 01 14 05 0F 3F A1 C0 4E 00 00 20 2F 02 00 00 00  ....?..N.. /....
00001e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00001f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA  ..............U.
```

可以看到，我们的新硬盘前1BEh个字节都是零。第1BEh到第1FDh字节便是分区表的内容了，按照表9.3的说明，可知它们的意义如表9.4所示。

从表中可知，第一个分区始于第3Fh扇区，共有4E81h个扇区，第二个分

表9.4 硬盘映像主引导扇区的分区表

分区序号	状态	分区类型	起始扇区 LBA	扇区数目
0	不可引导	83	3F	4E81
1	不可引导	05	4EC0	22F20

区始于第4EC0h扇区，共有22F20h个扇区。然而显然这些信息是不够的，我们还有若干逻辑分区的信息没得到呢。没关系，一步一步来，我们现在就来看一下第二个分区——也就是扩展分区的第一个扇区是什么样子。扩展分区的开始字节为9D8000h（4EC0h×512），它的内容如下：

```
▷ xxd -u -a -g 1 -c 16 -s 0x9D8000 -l 512 80m.img
09d8000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
09d81b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 80 01  ................
09d81c0: 01 14 99 0F 3F 3B 3F 00 00 00 41 9D 00 00 00 00  ....?;?...A.....
09d81d0: 01 3C 05 0F 3F 59 80 9D 00 00 20 76 00 00 00 00  .<..?Y.... v....
09d81e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
09d81f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA  ..............U.
```

其主要项的意义如表9.5所示。

表9.5 硬盘映像扩展分区的分区表

分区序号	状态	分区类型	起始扇区 LBA	扇区数目
0	可引导	99	3F	9D41
1	不可引导	05	9D80	7620

前一个分区的起始扇区LBA是3Fh，这是个相对于扩展分区基地址的LBA，也就是说，它真正的LBA是4EC0h+3Fh=4EFFh。后一个分区，根据其分区类型05可知，它又是个扩展分区，起始扇区LBA为4EC0h+9D80h=EC40h，字节偏移为EC40h×512=1D88000h，我们继续看看其引导扇区：

```
▷ xxd -u -a -g 1 -c 16 -s 0x1D88000 -l 512 80m.img
1d88000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
1d881b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
1d881c0: 01 3C 83 0F 3F 59 3F 00 00 00 E1 75 00 00 00 00  .<..?Y?....u....
1d881d0: 01 5A 05 0F 3F 83 A0 13 01 00 60 A5 00 00 00 00  .Z..?.....`.....
1d881e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
1d881f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA  ..............U.
```

其意义如表9.6所示。

表9.6 硬盘映像扩展分区中的第二个分区表

分区序号	状态	分区类型	起始扇区 LBA	扇区数目
0	不可引导	83	3F	75E1
1	不可引导	05	113A0	A560

从分区类型值（System ID）可以看出，在这个分区中，又包含了一个“普

通的”分区和一个扩展分区，你可能一下子明白了，多个逻辑分区是由嵌套来实现的。一个扩展分区里包含一个普通分区的同时，又可以嵌套一个扩展分区，一层一层的。到目前为止，我们已经剥开了两层，如图9.7所示。

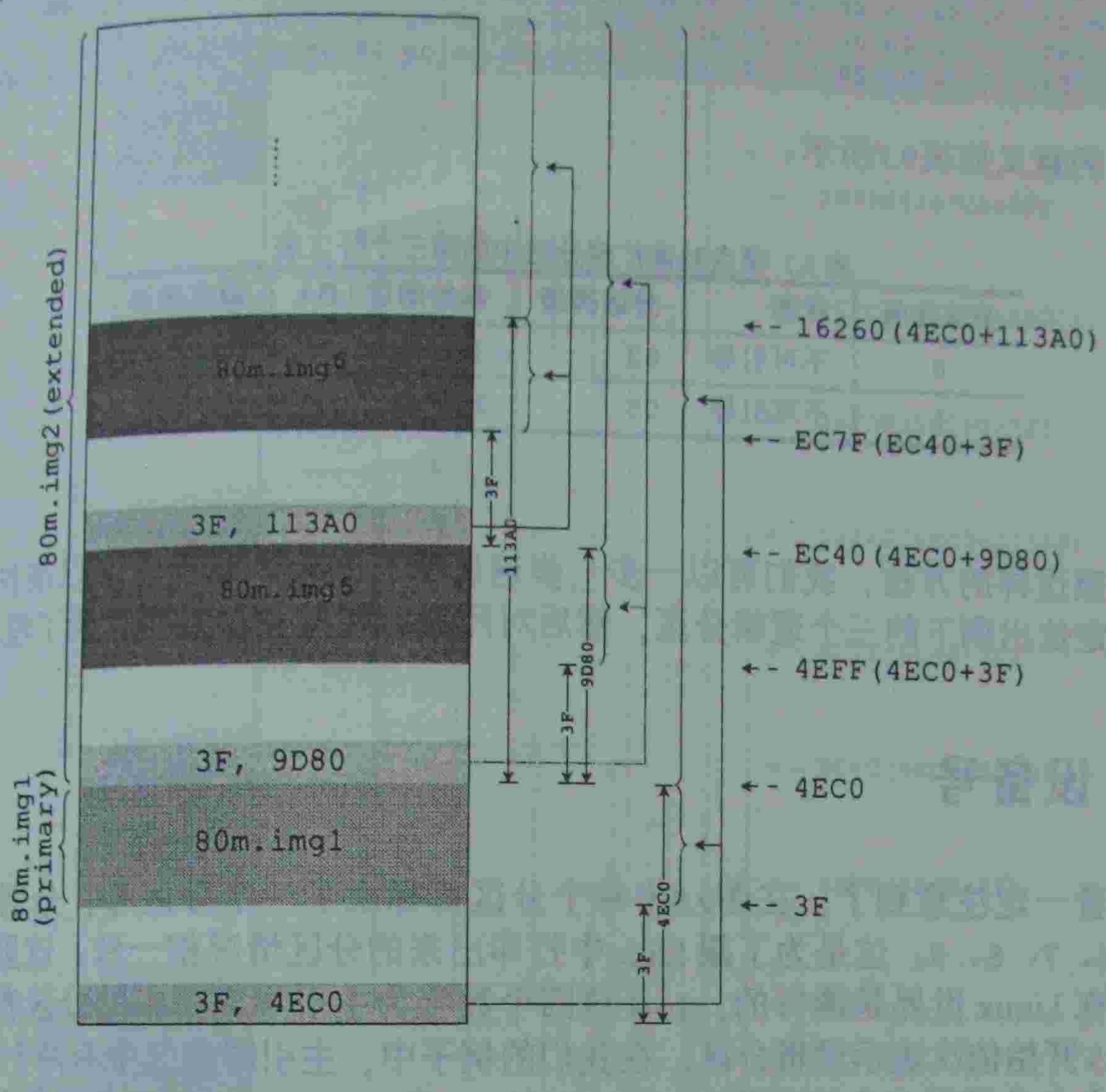


图 9.7 扩展分区和逻辑分区

其实这种层状结构，也可以看作是一个链表，链表的节点即为扩展分区的分区表，每个节点中有两个表项，前一个表项描述一个普通分区，后一个表项指向下一个节点。

需要留意的一点是，前一个表项中的起始扇区LBA是相对于当前扩展分区的，而后一个表项中的起始扇区——也就是下一个扩展分区的起始扇区——是相对于硬盘主引导扇区所指明的扩展分区的起始扇区的。这样说有点拗口，就本例来说，扇区EC40h（字节偏移为1D88000h）中的分区表有两个表项（见表9.6），前一项的起始扇区LBA为3Fh，它的实际LBA要将3Fh与EC40h相加，即EC7Fh；后一项的起始扇区LBA为113A0h，它的实际地址要与4EC0h相加，即16260h。

明白了这些，遍历所有逻辑扇区的工作所需要的就只剩下一点耐心和细心了。我们马上看一下扇区16260h，它的字节偏移为2C4C000h：

```
▷ xxd -u -a -g 1 -c 16 -s 0x2C4C000 -l 512 80m.img
```

```
2c4c000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
2c4c1b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01  ................
2c4c1c0: 01 5A 83 0F 3F 83 3F 00 00 00 21 A5 00 00 00 00  .Z..?.?...!.....
2c4c1d0: 01 84 05 0F 3F 9F 00 B9 01 00 40 6E 00 00 00 00  ....?.....@n....
2c4c1e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
2c4c1f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA  ..............U.
```

它的意义如表9.7所示。

表9.7 硬盘映像扩展分区中的第三个分区表

分区序号	状态	分区类型	起始扇区 LBA	扇区数目
0	不可引导	83	3F	A521
1	不可引导	05	1B900	6E40

按照这样的方法，我们可以一步一步遍历所有的分区，读者可以亲自试验一下，定位出剩下的三个逻辑分区，然后对照图9.8检验自己计算正确了没有。

9.6 设备号

读者一定注意到了，在图9.8中每个分区都标上了一个分区号，分别是1、2、5、6、7、8、9。这是为了跟 fdisk 中打印出来的分区情况相一致。这里的编号规则在 Linux 世界是通行的，1~4 这四个数字为主引导扇区中的分区表项所用，从 5 开始依次表示逻辑分区。在我们的例子中，主引导扇区中有两个表项，对应一个主分区和一个扩展分区，即 80m.img1 和 80m.img2，扩展分区中有 5 个逻辑分区，从 80m.img5 到 80m.img9。

其实 1、2、5~9 等这些数字有个名称，叫做次设备号。其作用是给每个设备（分区）起一个名字，这样驱动程序就能方便地管理它们。另外还有个我们没讲过的主设备号，它的作用是给每一类设备一个名字，以方便管理。举个例子，假设我们的计算机内有三块硬盘和两个软盘。对用户而言，操作硬盘和软盘上文件的区别可能仅在于路径不同，但对于操作系统，硬盘和软盘需要不同的驱动程序，所以不同类别的硬件需要区别对待，这就是主设备号存在的理由。同时，硬盘有多个，而且每个硬盘上可能有多个分区，对这些分区，又需要区别对待，于是又用到了次设备号。简单来说，主设备号告诉操作系统应该用哪个驱动程序来处理，次设备号告诉驱动程序这是具体哪个设备。如果读者无法一下子全明白，不要紧，等代码写出来，你肯定就全明白了。

在我们的系统中，我们也需要有主次设备号，但对硬盘而言，我们采用与 Linux 不同的编号规则，请看图9.9。

在这里我们还是只看主 IDE 通道上连接两块硬盘的情况。图中括号内的便是次设备号。主盘是 hd0，其次设备号为 0，它的主引导扇区分区表对应的四个分区分别是 hd1、hd2、hd3、hd4。每个扩展分区中最多有 16 个逻辑分区，以字母 a~p 表示，逻辑分区的次设备号是以 hd1a 为基准递增的。这种编号规则的好处是，给定一个次设备号，可以很容易地计算出它是主分区还是扩展分区，或者是哪个扩展分区的哪个逻辑分区。同时，给定一个分区的名称，我们也很容

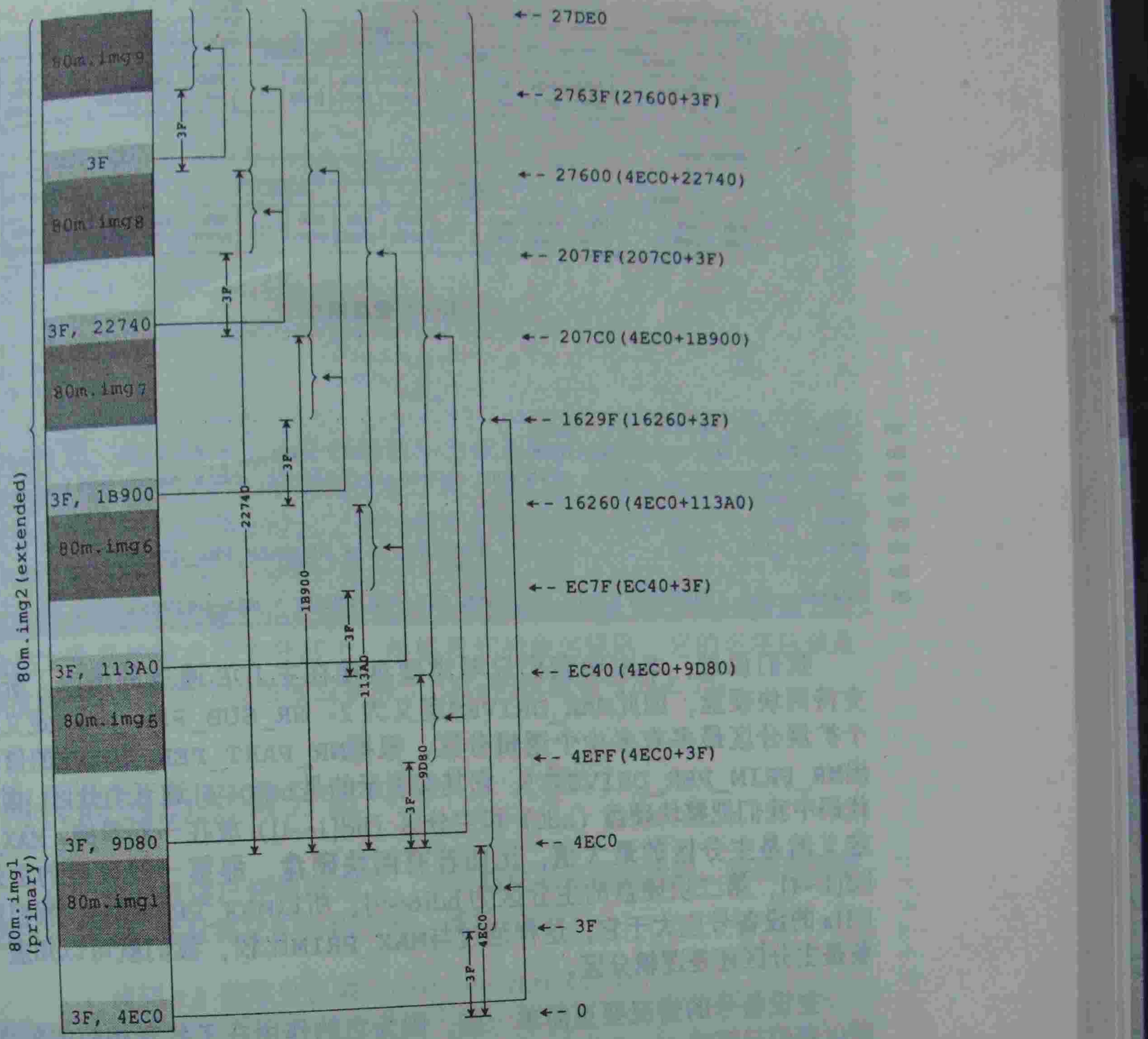


图 9.8 80m.img 的所有分区

易计算出其次设备号。

配合这套规则，我们定义了一些宏，见代码9.5。

代码 9.5 硬盘设备号相关的宏（chapter9/b/include/const.h）

```
#define MAX_DRIVES          2
#define NR_PART_PER_DRIVE   4
#define NR_SUB_PER_PART     16
#define NR_SUB_PER_DRIVE    (NR_SUB_PER_PART * NR_PART_PER_DRIVE)
#define NR_PRIM_PER_DRIVE   (NR_PART_PER_DRIVE + 1)

/**
```

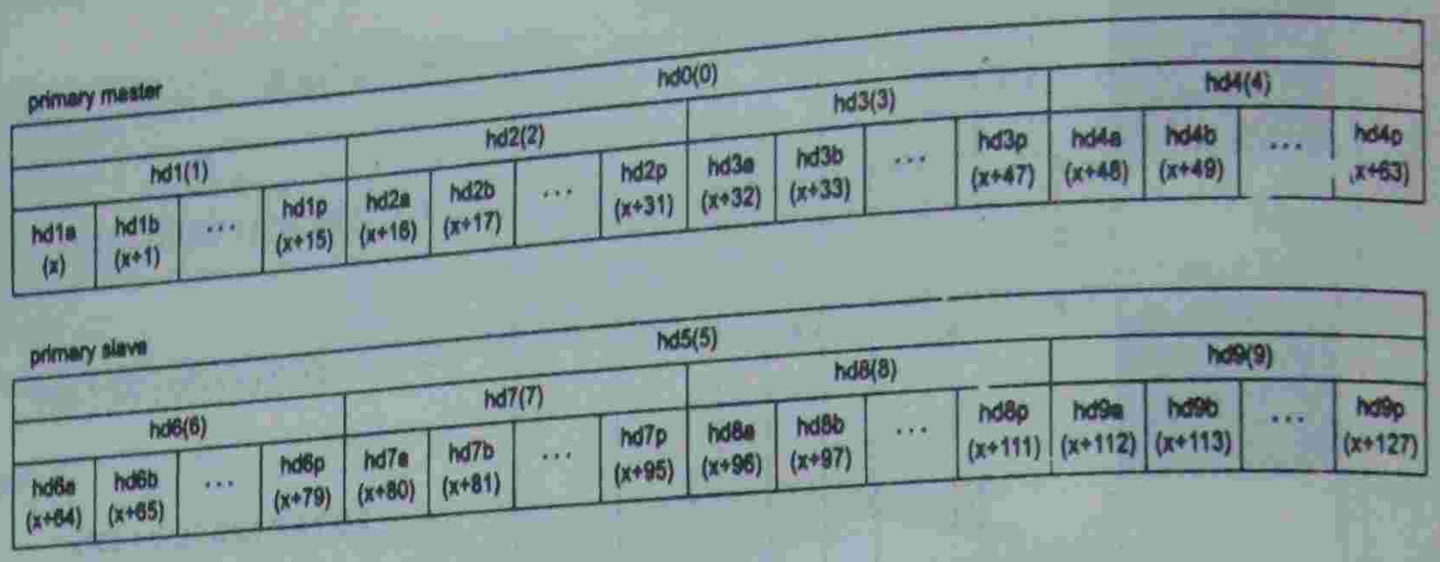


图 9.9 硬盘编号

```
224  * @def MAX_PRIM
225  * Defines the max minor number of the primary partitions.
226  * If there are 2 disks, prim_dev ranges in hd[0-9], this macro will
227  * equals 9.
228  */
229 #define MAX_PRIM                        (MAX_DRIVES * NR_PRIM_PER_DRIVE - 1)
230
231 #define MAX_SUBPARTITIONS               (NR_SUB_PER_DRIVE * MAX_DRIVES)
```

我们说过，本书中我们只考虑硬盘接在主 IDE 通道的情况，所以最多支持两块硬盘，因此MAX_DRIVES定义为 2。NR_SUB_PER_PART定义的是每个扩展分区最多有多少个逻辑分区。根据NR_PART_PER_DRIVE的值容易算出NR_PRIM_PER_DRIVE为5，它其实表示的是hd[0~4]这五个分区，因为有些代码中我们把整块硬盘（hd0）和主分区（hd[1~4]）放在一起看待。MAX_PRIM定义的是主分区的最大值，比如若有两块硬盘，那第一块硬盘的主分区为hd[1~4]，第二块硬盘的主分区为hd[6~9]，所以MAX_PRIM为 9，我们定义的hd1a 的设备号应大于它，这样通过与MAX_PRIM比较，我们就可以知道一个设备是主分区还是逻辑分区。

主设备号的情况要更简单一些，因为它的作用在于找到相应的驱动程序，所以我们只要建立一个以主设备号为下标、以驱动程序号（PID）为值的数组，就可以了，见代码9.6。

代码 9.6 dd_map（chapter9/b/kernel/global.c）

```
45 /*****************************************************************************/
46 /**
47  * For dd_map[k],
48  * 'k' is the device nr.\ dd_map[k].driver_nr is the driver nr.
49  *
50  * Remeber to modify include/const.h if the order is changed.
51  ****************************************************************************/
52 struct dev_drv_map dd_map[] = {
53         /* driver nr.                major device nr.
54            ----------                ----------------  */
55         {INVALID_DRIVER},         /**< 0 : Unused */
56         {INVALID_DRIVER},         /**< 1 : Reserved for floppy driver */
57         {INVALID_DRIVER},         /**< 2 : Reserved for cdrom driver */
58         {TASK_HD},                /**< 3 : Hard disk */
59         {TASK_TTY},               /**< 4 : TTY */
60         {INVALID_DRIVER}          /**< 5 : Reserved for scsi disk driver */
61 };
```

主设备号的定义见代码9.7。

代码 9.7 主设备号（chapter9/b/include/const.h）

```
198 /* major device numbers (corresponding to kernel/global.c::dd_map[]) */
199 #define NO_DEV              0
200 #define DEV_FLOPPY          1
201 #define DEV_CDROM           2
202 #define DEV_HD              3
203 #define DEV_CHAR_TTY        4
204 #define DEV_SCSI            5
205
206 /* make device number from major and minor numbers */
207 #define MAJOR_SHIFT         8
208 #define MAKE_DEV(a,b)       ((a << MAJOR_SHIFT) | b)
209
210 /* separate major and minor numbers from device number */
211 #define MAJOR(x)            ((x >> MAJOR_SHIFT) & 0xFF)
212 #define MINOR(x)            (x & 0xFF)
```

一定要注意，代码9.7中主设备号的值即为代码9.6中dd_map[]的下标，两者是相呼应的，若要改变的话要同时改变。将来我们每个设备号都有主设备号和次设备号组成（代码9.7第208行），通过简单的位运算即可得到主设备号（代码9.7第211行）及次设备号（代码9.7第212行）。

刚才我们在给磁盘映像 80m.img 分区时，指定 80m.img5 为Orange'S分区，我们将来会把文件系统建在这个分区上。根据我们的命名规则，它的名字应该是 hd2a（参考图9.9）。它的次设备号应该等于 hd1a 的次设备号加上 16。

9.7 用代码遍历所有分区

好了，分区表的原理已经很清楚了，下面我们就来添加代码，在硬盘驱动程序中找出所有分区并且将它们打印出来。请看代码9.8。

代码 9.8 读取分区表（chapter9/b/kernel/hd.c）

```
37 PRIVATE struct hd_info  hd_info[1];
38
39 #define DRV_OF_DEV(dev) (dev <= MAX_PRIM ? \
40                          dev / NR_PRIM_PER_DRIVE : \
41                          (dev - MINOR_hd1a) / NR_SUB_PER_DRIVE)
42
43 /*****************************************************************************
44  *                                task_hd
45  *****************************************************************************/
46 /**
47  * Main loop of HD driver.
48  *
49  *****************************************************************************/
50 PUBLIC void task_hd()
51 {
...
61
62         switch (msg.type) {
63         case DEV_OPEN:
64                 hd_open(msg.DEVICE);
65                 break;
...
70
...         }
74 }
75
76 /*****************************************************************************
```

```
77  *                                    init_hd
78  *****************************************************************************/
79 /**
80  * <Ring 1> Check hard drive, set IRQ handler, enable IRQ and initialize data
81  *          structures.
82  *****************************************************************************/
83 PRIVATE void init_hd()
84 {
85         int i;
...
95         for (i = 0; i < (sizeof(hd_info) / sizeof(hd_info[0])); i++)
96                 memset(&hd_info[i], 0, sizeof(hd_info[0]));
97         hd_info[0].open_cnt = 0;
98 }
99
100 /*****************************************************************************
101  *                                    hd_open
102  *****************************************************************************/
103 /**
104  * <Ring 1> This routine handles DEV_OPEN message. It identify the drive
105  * of the given device and read the partition table of the drive if it
106  * has not been read.
107  *
108  * @param device The device to be opened.
109  *****************************************************************************/
110 PRIVATE void hd_open(int device)
111 {
112         int drive = DRV_OF_DEV(device);
113         assert(drive == 0);     /* only one drive */
114
115         hd_identify(drive);
116
117         if (hd_info[drive].open_cnt++ == 0) {
118                 partition(drive * (NR_PART_PER_DRIVE + 1), P_PRIMARY);
119                 print_hdinfo(&hd_info[drive]);
120         }
121 }
122
123 /*****************************************************************************
124  *                                    get_part_table
125  *****************************************************************************/
126 /**
127  * <Ring 1> Get a partition table of a drive.
128  *
129  * @param drive   Drive nr (0 for the 1st disk, 1 for the 2nd, ...)n
130  * @param sect_nr The sector at which the partition table is located.
131  * @param entry   Ptr to part_ent struct.
132  *****************************************************************************/
133 PRIVATE void get_part_table(int drive, int sect_nr, struct part_ent * entry)
134 {
135         struct hd_cmd cmd;
136         cmd.features    = 0;
137         cmd.count       = 1;
138         cmd.lba_low     = sect_nr & 0xFF;
139         cmd.lba_mid     = (sect_nr >>  8) & 0xFF;
140         cmd.lba_high    = (sect_nr >> 16) & 0xFF;
141         cmd.device      = MAKE_DEVICE_REG(1, /* LBA mode*/
142                                           drive,
143                                           (sect_nr >> 24) & 0xF);
144         cmd.command     = ATA_READ;
145         hd_cmd_out(&cmd);
146         interrupt_wait();
147
148         port_read(REG_DATA, hdbuf, SECTOR_SIZE);
149         memcpy(entry,
150                hdbuf + PARTITION_TABLE_OFFSET,
151                sizeof(struct part_ent) * NR_PART_PER_DRIVE);
```

```
}

/*****************************************************************************
 *                                partition
 *****************************************************************************/
/**
 * <Ring 1> This routine is called when a device is opened. It reads the
 * partition table(s) and fills the hd_info struct.
 *
 * @param device Device nr.
 * @param style  P_PRIMARY or P_EXTENDED.
 *****************************************************************************/
PRIVATE void partition(int device, int style)
{
        int i;
        int drive = DRV_OF_DEV(device);
        struct hd_info * hdi = &hd_info[drive];

        struct part_ent part_tbl[NR_SUB_PER_DRIVE];

        if (style == P_PRIMARY) {
                get_part_table(drive, drive, part_tbl);

                int nr_prim_parts = 0;
                for (i = 0; i < NR_PART_PER_DRIVE; i++) { /* 0~3 */
                        if (part_tbl[i].sys_id == NO_PART)
                                continue;

                        nr_prim_parts++;
                        int dev_nr = i + 1;                       /* 1~4 */
                        hdi->primary[dev_nr].base = part_tbl[i].start_sect;
                        hdi->primary[dev_nr].size = part_tbl[i].nr_sects;

                        if (part_tbl[i].sys_id == EXT_PART) /* extended */
                                partition(device + dev_nr, P_EXTENDED);
                }
                assert(nr_prim_parts != 0);
        }
        else if (style == P_EXTENDED) {
                int j = device % NR_PRIM_PER_DRIVE; /* 1~4 */
                int ext_start_sect = hdi->primary[j].base;
                int s = ext_start_sect;
                int nr_1st_sub = (j - 1) * NR_SUB_PER_PART; /* 0/16/32/48 */

                for (i = 0; i < NR_SUB_PER_PART; i++) {
                        int dev_nr = nr_1st_sub + i;/* 0~15/16~31/32~47/48~63 */

                        get_part_table(drive, s, part_tbl);

                        hdi->logical[dev_nr].base = s + part_tbl[0].start_sect;
                        hdi->logical[dev_nr].size = part_tbl[0].nr_sects;

                        s = ext_start_sect + part_tbl[1].start_sect;

                        /* no more logical partitions
                           in this extended partition */
                        if (part_tbl[1].sys_id == NO_PART)
                                break;
                }
        }
        else {
                assert(0);
        }
}

/*****************************************************************************
 *                                print_hdinfo
```

```
219  /*****************************************************************************/
220  /**
221   * <Ring 1> Print disk info.
222   *
223   * @param hdi  Ptr to struct hd_info.
224   *****************************************************************************/
225  PRIVATE void print_hdinfo(struct hd_info * hdi)
226  {
227          int i;
228          for (i = 0; i < NR_PART_PER_DRIVE + 1; i++) {
229                  printl("%sPART_%d: base %d(0x%x), size %d(0x%x) (in sector)\n",
230                         i == 0 ? " " : "     ",
231                         i,
232                         hdi->primary[i].base,
233                         hdi->primary[i].base,
234                         hdi->primary[i].size);
235
236          }
237          for (i = 0; i < NR_SUB_PER_DRIVE; i++) {
238                  if (hdi->logical[i].size == 0)
239                          continue;
240                  printl("          "
241                         "%d: base %d(0x%x), size %d(0x%x) (in sector)\n",
242                         i,
243                         hdi->logical[i].base,
244                         hdi->logical[i].base,
245                         hdi->logical[i].size,
246                         hdi->logical[i].size);
247          }
248  }
249
250  /*****************************************************************************
251   *                                hd_identify
252   *****************************************************************************/
253  /**
254   * <Ring 1> Get the disk information.
255   *
256   * @param drive  Drive Nr.
257   *****************************************************************************/
258  PRIVATE void hd_identify(int drive)
259  {
260          struct hd_cmd cmd;
261          cmd.device  = MAKE_DEVICE_REG(0, drive, 0);
262          cmd.command = ATA_IDENTIFY;
263          hd_cmd_out(&cmd);
264          interrupt_wait();
265          port_read(REG_DATA, hdbuf, SECTOR_SIZE);
266
267          print_identify_info((u16*)hdbuf);
268
269          u16* hdinfo = (u16*)hdbuf;
270
271          hd_info[drive].primary[0].base = 0;
272          /* Total Nr of User Addressable Sectors */
273          hd_info[drive].primary[0].size = ((int)hdinfo[61] << 16) + hdinfo[60];
274  }
```

代码9.1中，驱动程序收到DEV_OPEN消息之后调用函数hd_identify()，在这里我们改成了调用函数hd_open()，这是新加的一个函数，它接受的参数即为设备的次设备号。在hd_open()中，我们首先由设备次设备号得到驱动器号，由于我们的 Bochs 只定义了一个硬盘，所以这里的驱动器号一定是 0。接下来便是调用hd_identify()了，这跟代码9.1是一样的。再往下是一个if语句，其中涉及我们新定义的一个结构体：hd_info。它的定义见代码9.9。

代码 9.9 hd_info（chapter9/b/include/hd.h）

```
struct part_info {
        u32     base;   /* # of start sector (NOT byte offset, but SECTOR) */
        u32     size;   /* how many sectors in this partition */
};

/* main drive struct, one entry per drive */
struct hd_info
{
        int                     open_cnt;
        struct part_info        primary[NR_PRIM_PER_DRIVE];
        struct part_info        logical[NR_SUB_PER_DRIVE];
};
```

与此同时我们声明了一个数组：hd_info[1]（代码9.8第37行），鉴于目前我们的虚拟机只装了一块硬盘，我们只给了它一个成员。hd_info的主要作用是记录硬盘的分区信息，每个硬盘应有一个hd_info结构。其中primary成员用来记录所有主分区的起始扇区和扇区数目，它们占用primary[1-4]，logical用来记录所有逻辑分区的起始扇区和扇区数目。注意这里整个硬盘的起始扇区和扇区数目记在了primary[0]中，见代码9.8第271行和第273行。

我们接着来看hd_open()，其中的if语句判断hd_info的open_cnt成员是否为零，并将其自加。由于在init_hd()中我们将结构体清零了，所以第一次执行到这里时if判断为真，于是调用partition()和print_hdinfo()。

函数partition()所做的便是获取硬盘分区表了，这个过程我们上一节已经很熟悉了，这里只不过是用C语言代码写出来而已。注意其中的读硬盘扇区的工作封装在了函数get_part_table()中，和执行IDENTIFY命令类似，执行READ命令时我们同样是先填充hd_cmd结构，然后交给hd_cmd_out()来写寄存器。

函数print_hdinfo()就更加简单了，将获取的分区信息打印出来而已。

跟代码9.1还有一点不同之处在于，代码9.8跟驱动程序的使用者之间的接口变了。过去FS发送DEV_OPEN消息时没有任何附加参数（代码9.4），现在hd_open()是带参数的了，所以FS的代码也要修改一下，见代码9.10。

代码 9.10 修改后的文件系统进程（chapter9/b/fs/main.c）

```
/*****************************************************************************
 *                                task_fs
 *****************************************************************************/
/**
 * <Ring 1> The main loop of TASK FS.
 *
 *****************************************************************************/
PUBLIC void task_fs()
{
        printl("Task␣FS␣begins.\n");

        /* open the device: hard disk */
        MESSAGE driver_msg;
        driver_msg.type = DEV_OPEN;
        driver_msg.DEVICE = MINOR(ROOT_DEV);
        assert(dd_map[MAJOR(ROOT_DEV)].driver_nr != INVALID_DRIVER);
        send_recv(BOTH, dd_map[MAJOR(ROOT_DEV)].driver_nr, &driver_msg);

        spin("FS");
}
```

这里我们不仅将ROOT_DEV的次设备号通过消息发给了驱动程序，而且使用哪个驱动程序也变成由dd_map来选择，这样一来，只要将ROOT_DEV定义好了，正确的消息便能发送给正确的驱动程序。ROOT_DEV的定义见代码9.11。

代码 9.11 ROOT_DEV（chapter9/b/include/const.h）

```
233 /* device numbers of hard disk */
234 #define MINOR_hd1a              0x10
235 #define MINOR_hd2a              (MINOR_hd1a+NR_SUB_PER_PART)
236
237 #define ROOT_DEV                MAKE_DEV(DEV_HD, MINOR_BOOT)
```

其中MINOR_BOOT被定义成MINOR_hd2a，放在了一个新的头文件config.h中，将来一些硬盘配置的宏定义将放在这个文件中。

好了，现在FS会把hd2a的次设备号发给dd_map[DEV_HD].driver_nr，即TASK_HD——我们的硬盘驱动，然后驱动程序将执行hd_open，从而获取硬盘的分区信息，让我们执行一下看看，见图9.10。

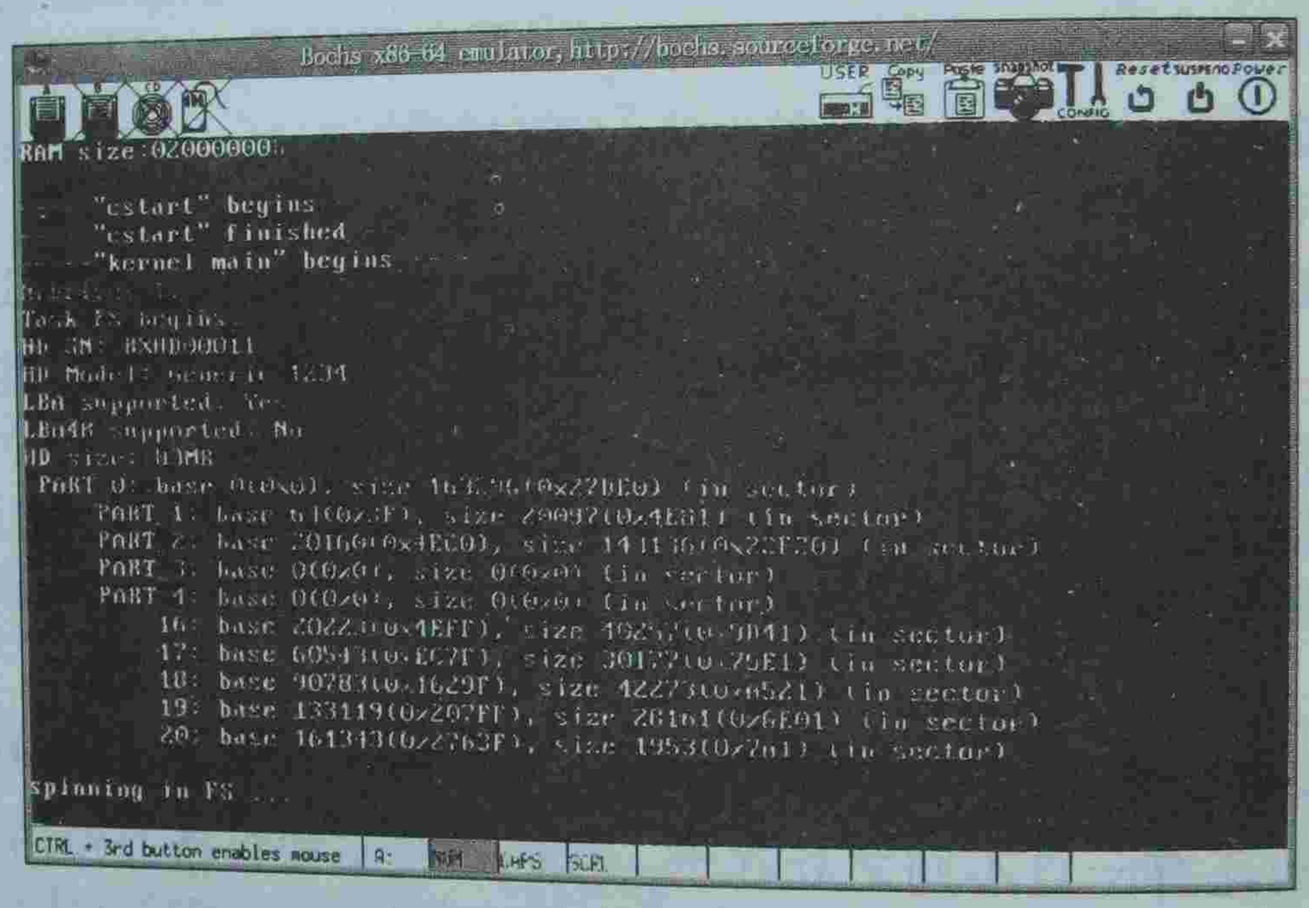


图 9.10 获取硬盘分区信息

Wow，我们之前费了很大力气才得到的分区信息，就这样一下子打印出来了，怎么样？很酷吧？

9.8 完善硬盘驱动程序

开心之余，让我们进一步完善一下我们的驱动程序。目前驱动程序只能处理DEV_OPEN消息，这显然是不够的，好歹也得有个DEV_READ和DEV_WRITE吧，要不然只“打开”它也用不起来啊。我们现在就来添加读写硬盘的代码。如果读者觉得不耐烦的话，可以跳过本节，等到将来用到这些消息时再回来看。如果你不着急，可以先浏览一下，因为代码总共也不多。

代码 9.12 让驱动程序处理更多消息（chapter9/c/kernel/hd.c）

```
46 /*****************************************************************************
47  *                                task_hd
```

```
/*****************************************************************************
 *                                task_hd
 *****************************************************************************/
/**
 * Main loop of HD driver.
 *
 *****************************************************************************/
PUBLIC void task_hd()
{
        MESSAGE msg;

        while (1) {
                send_recv(RECEIVE, ANY, &msg);

                switch (msg.type) {
                case DEV_OPEN:
                        hd_open(msg.DEVICE);
                        break;

                case DEV_CLOSE:
                        hd_close(msg.DEVICE);
                        break;

                case DEV_READ:
                case DEV_WRITE:
                        hd_rdwt(&msg);
                        break;

                case DEV_IOCTL:
                        hd_ioctl(&msg);
                        break;

                default:
                        dump_msg("HD driver::unknown msg", &msg);
                        spin("FS::main_loop (invalid msg.type)");
                        break;
                }

                send_recv(SEND, msg.source, &msg);
        }
}

/*****************************************************************************
 *                                hd_close
 *****************************************************************************/
/**
 * <Ring 1> This routine handles DEV_CLOSE message.
 *
 * @param device The device to be opened.
 *****************************************************************************/
PRIVATE void hd_close(int device)
{
        int drive = DRV_OF_DEV(device);
        assert(drive == 0);     /* only one drive */

        hd_info[drive].open_cnt--;
}


/*****************************************************************************
 *                                hd_rdwt
 *****************************************************************************/
/**
 * <Ring 1> This routine handles DEV_READ and DEV_WRITE message.
 *
 * @param p Message ptr.
 *****************************************************************************/
PRIVATE void hd_rdwt(MESSAGE * p)
{
        int drive = DRV_OF_DEV(p->DEVICE);

        u64 pos = p->POSITION;
        assert((pos >> SECTOR_SIZE_SHIFT) < (1 << 31));

        /**
         * We only allow to R/W from a SECTOR boundary:
         */
        assert((pos & 0x1FF) == 0);
```

```
177     u32 sect_nr = (u32)(pos >> SECTOR_SIZE_SHIFT); /* pos / SECTOR_SIZE */
178     int logidx = (p->DEVICE - MINOR_hd1a) % NR_SUB_PER_DRIVE;
179     sect_nr += p->DEVICE < MAX_PRIM ?
180             hd_info[drive].primary[p->DEVICE].base :
181             hd_info[drive].logical[logidx].base;
182
183     struct hd_cmd cmd;
184     cmd.features    = 0;
185     cmd.count       = (p->CNT + SECTOR_SIZE - 1) / SECTOR_SIZE;
186     cmd.lba_low     = sect_nr & 0xFF;
187     cmd.lba_mid     = (sect_nr >>  8) & 0xFF;
188     cmd.lba_high    = (sect_nr >> 16) & 0xFF;
189     cmd.device      = MAKE_DEVICE_REG(1, drive, (sect_nr >> 24) & 0xF);
190     cmd.command     = (p->type == DEV_READ) ? ATA_READ : ATA_WRITE;
191     hd_cmd_out(&cmd);
192
193     int bytes_left = p->CNT;
194     void * la = (void*)va2la(p->PROC_NR, p->BUF);
195
196     while (bytes_left) {
197             int bytes = min(SECTOR_SIZE, bytes_left);
198             if (p->type == DEV_READ) {
199                     interrupt_wait();
200                     port_read(REG_DATA, hdbuf, SECTOR_SIZE);
201                     phys_copy(la, (void*)va2la(TASK_HD, hdbuf), bytes);
202
203             }
204             else {
205                     if (!waitfor(STATUS_DRQ, STATUS_DRQ, HD_TIMEOUT))
206                             panic("hd writing error.");
207
208                     port_write(REG_DATA, la, bytes);
209                     interrupt_wait();
210             }
211             bytes_left -= SECTOR_SIZE;
212             la += SECTOR_SIZE;
213     }
214 }
215
216 /*****************************************************************************
217  *                                hd_ioctl
218  *****************************************************************************/
219 /**
220  * <Ring 1> This routine handles the DEV_IOCTL message.
221  *
222  * @param p  Ptr to the MESSAGE.
223  *****************************************************************************/
224 PRIVATE void hd_ioctl(MESSAGE * p)
225 {
226         int device = p->DEVICE;
227         int drive = DRV_OF_DEV(device);
228
229         struct hd_info * hdi = &hd_info[drive];
230
231         if (p->REQUEST == DIOCTL_GET_GEO) {
232                 void * dst = va2la(p->PROC_NR, p->BUF);
233                 void * src = va2la(TASK_HD,
234                                    device < MAX_PRIM ?
235                                    &hdi->primary[device] :
236                                    &hdi->logical[(device - MINOR_hd1a) %
237                                                  NR_SUB_PER_DRIVE]);
238
239                 phys_copy(dst, src, sizeof(struct part_info));
240         }
241         else {
242                 assert(0);
243         }
```

```
}
```

代码9.12中增加了对四种消息的处理，所以到目前为止我们的硬盘驱动总共支持五种消息：

- DEV_OPEN
- DEV_CLOSE
- DEV_READ
- DEV_WRITE
- DEV_IOCTL

其中DEV_READ和DEV_WRITE用同一个函数hd_rdwt()来处理，所以增加的函数共有三个，hd_close()非常简单。hd_ioctl()也容易理解，目前只支持一种消息类型——DIOCTL_GET_GEO，所做工作只是把请求的设备的起始扇区和扇区数目返回给调用者而已。剩下一个hd_rdwt()，这里我们没用任何的优化措施，没有缓冲区，也没有电梯算法，只是傻傻地让干什么就干什么，将来你会看到这样做的确效率不佳，但是管它呢，这样做至少让代码看上去很简洁，而且它已经够用了！等到我们无法忍受其速度时再来优化也不迟。记住Knuth教授的话：过早的优化是万恶之源。

9.9 在硬盘上制作一个文件系统

硬盘驱动写完，终于开始正式编写文件系统了。但在开始之前，我们需要了解，文件系统通常有两个含义：

- 用于存储和组织计算机文件数据的一套方法
- 存在于某介质上的具备某种格式的数据

当我们说要“编写一个文件系统”时，我们指的是前者。因为我们需要考虑如何利用空间，如何对文件进行添加、删除以及修改，还要考虑不同类型的文件如何并存于同一个文件系统，比如在*nix世界中，设备通常也是文件。所以我们考虑的其实是一套机制，包括静态的存储方法和动态的管理方法。

当我们说某个硬盘分区上是“某某文件系统”时，我们说的是后者。它是静态的，其格式表明这个分区是由某种策略和机制来管理。或者说，它是第一种含义的管理对象。

在第9.4节中，我们讨论的其实是第二种含义，这是我们对文件系统最直观的认识。接下来，我们仍然从这个直观认识入手，先在硬盘上划定格式，等有一个框架之后，再考虑文件的增删改等诸多事宜。有了图9.6以及我们刚刚完成的硬盘驱动程序，这一工作已经不困难了。

9.9.1 文件系统涉及的数据结构

首先，我们把第9.4节中提到的诸要素具体化成各种数据结构，如代码9.13所示。

代码9.13 super_block和inode（chapter9/d/include/fs.h）

```
20
21  /**
22   * @def    MAGIC_V1
23   * @brief Magic number of FS v1.0
24   */
25  #define MAGIC_V1        0x111
26
27  /**
28   * @struct super_block fs.h "include/fs.h"
29   * @brief  The 2nd sector of the FS
30   *
31   * Remember to change SUPER_BLOCK_SIZE if the members are changed.
32   */
33  struct super_block {
34          u32     magic;                  /**< Magic number */
35          u32     nr_inodes;              /**< How many inodes */
36          u32     nr_sects;               /**< How many sectors */
37          u32     nr_imap_sects;          /**< How many inode-map sectors */
38          u32     nr_smap_sects;          /**< How many sector-map sectors */
39          u32     n_1st_sect;             /**< Number of the 1st data sector */
40          u32     nr_inode_sects;         /**< How many inode sectors */
41          u32     root_inode;             /**< Inode nr of root directory */
42          u32     inode_size;             /**< INODE_SIZE */
43          u32     inode_isize_off;        /**< Offset of 'struct inode::i_size' */
44          u32     inode_start_off;        /**< Offset of 'struct inode::i_start_sect' */
45          u32     dir_ent_size;           /**< DIR_ENTRY_SIZE */
46          u32     dir_ent_inode_off;/**< Offset of 'struct dir_entry::inode_nr' */
47          u32     dir_ent_fname_off;/**< Offset of 'struct dir_entry::name' */
48
49          /*
50           * the following item(s) are only present in memory
51           */
52          int     sb_dev;                 /**< the super block's home device */
53  };
54
55  /**
56   * @def    SUPER_BLOCK_SIZE
57   * @brief The size of super block \b in \b the \b device.
58   *
59   * Note that this is the size of the struct in the device, \b NOT in memory.
60   * The size in memory is larger because of some more members.
61   */
62  #define SUPER_BLOCK_SIZE        56
63
64  /**
65   * @struct inode
66   * @brief  i-node
67   *
68   * The \c start_sect and\c nr_sects locate the file in the device,
69   * and the size show how many bytes is used.
70   * If <tt> size < (nr_sects * SECTOR_SIZE) </tt>, the rest bytes
71   * are wasted and reserved for later writing.
72   *
73   * \b NOTE: Remember to change INODE_SIZE if the members are changed
74   */
75  struct inode {
76          u32     i_mode;                 /**< Accsess mode */
77          u32     i_size;                 /**< File size */
78          u32     i_start_sect;           /**< The first sector of the data */
```

```
        u32     i_nr_sects;     /**< How many sectors the file occupies */
        u8      _unused[16];    /**< Stuff for alignment */

        /* the following items are only present in memory */
        int     i_dev;
        int     i_cnt;          /**< How many procs share this inode  */
        int     i_num;          /**< inode nr.  */
};

/**
 * @def   INODE_SIZE
 * @brief The size of i-node stored \b in \b the \b device.
 *
 * Note that this is the size of the struct in the device, \b NOT in memory.
 * The size in memory is larger because of some more members.
 */
#define INODE_SIZE      32

/**
 * @def   MAX_FILENAME_LEN
 * @brief Max len of a filename
 * @see   dir_entry
 */
#define MAX_FILENAME_LEN        12

/**
 * @struct dir_entry
 * @brief  Directory Entry
 */
struct dir_entry {
        int     inode_nr;               /**< inode nr. */
        char    name[MAX_FILENAME_LEN]; /**< Filename */
};

/**
 * @def   DIR_ENTRY_SIZE
 * @brief The size of directory entry in the device.
 *
 * It is as same as the size in memory.
 */
#define DIR_ENTRY_SIZE  sizeof(struct dir_entry)
```

代码9.13定义了三个结构体，分别代表超级块、i-node和目录项。

超级块主要关注以下内容：

- 文件系统的标识。这里用一个魔数（Magic Number）表明本文件系统是Orange'S FS v1.0。
- 文件系统最多允许有多少个i-node。
- inode_array占用多少扇区。
- 文件系统总共扇区数是多少。
- inode-map占用多少扇区。
- sector-map占用多少扇区。
- 第一个数据扇区的扇区号是多少。
- 根目录区的i-node号是多少。

此外，inode和dir_entry两个结构体的一些信息也记在了这里。超级块有512个字节，通常是用不完的，所以哪怕有些项可放可不放，为了编码的方便，我们也可以尽管放进来。

请注意super_block这个结构体有个特殊的成员sb_dev，它在硬盘上是不存在的，这也是我们将SUPER_BLOCK_SIZE定义为56的原因。sb_dev存在的理由是这样的，我们打开一个设备后，会将超级块读入内存，这时我们要记录这个超级块是从哪里来的，于是我们把这个超级块所在设备的设备号记在sb_dev里，这样我们随时都能知道这个超级块是属于哪个设备的，同时我们也可以通过辨识设备号来随时得到该设备的超级块而不必重新进行读取。

存在这样的特殊成员的缺点也很明显，那就是必须保持super_block这个结构体和SUPER_BLOCK_SIZE的一致，改变结构体时需要同时改变代表其大小的宏，读者编码时需要注意。

我们的inode结构体目前很简单，其中i_start_sect代表文件的起始扇区，i_nr_sects代表总扇区数，i_size代表文件大小。在这里我们使用一个很笨拙的方法来存储文件，那就是事先为文件预留出一定的扇区数(i_nr_sects)，以便让文件能够追加数据，i_nr_sects一旦确定就不再更改。这样做的优点很明显，那就是分配扇区的工作在建立文件时一次完成，从此再也不用额外分配或释放扇区。缺点也很明显，那就是文件大小范围在文件建立之后就无法改变了，i_size满足：$i_size \in [0, i_nr_sects \times 512]$。

笔者最终决定使用这种有明显缺陷的i-node，原因还是为了简单，即便它有千般不是，它实在是太简单了，以至于可以大大简化我们的代码，而且，它还是能用的。等到我们有很多文件，需要灵活性时，我们再推出一个v2.0，那时需要改进的怕也不仅仅是一个i-node而已，所以在做第一步时，就让它傻一点吧。

成员i_mode主要被用来区分文件类型。我们提到过，文件不仅仅可以是磁盘上的一块数据，也可以是一个设备。一个普通文件和一个特殊文件（比如设备文件）将首先从i_mode上区分开来。

inode结构体里也有几个只在内存中存在的成员，理由和super_block类似。具体作用我们将来编码时会了解到。

最后一个结构体是dir_entry，它是存在于根目录文件中的数据结构，用来索引一个文件。它只有两个成员：i-node和文件名。将来我们的根目录将会是一个dir_entry数组，用以索引文件系统中所有的文件。

9.9.2 编码建立文件系统

既然文件系统的结构和所需要的数据结构都已经齐备，下面我们就开始写硬盘了，过一会儿，我们的硬盘的hd2a分区就会变成如图9.6所示的样子了。

代码9.14 初始化文件系统（chapter9/d/fs/main.c）

```
27  /*****************************************************************************
28   *
29   *                                  task_fs
30  /**
31   * <Ring 1> The main loop of TASK FS.
32   *
```

```
/*****************************************************************************/
PUBLIC void task_fs()
{
        printl("Task␣FS␣begins.\n");
        init_fs();
        spin("FS");
}

/*****************************************************************************
                                    init_fs
 *****************************************************************************/
/**
 * <Ring 1> Do some preparation.
 *
 *****************************************************************************/
PRIVATE void init_fs()
{
        /* open the device: hard disk */
        MESSAGE driver_msg;
        driver_msg.type = DEV_OPEN;
        driver_msg.DEVICE = MINOR(ROOT_DEV);
        assert(dd_map[MAJOR(ROOT_DEV)].driver_nr != INVALID_DRIVER);
        send_recv(BOTH, dd_map[MAJOR(ROOT_DEV)].driver_nr, &driver_msg);

        mkfs();
}

/*****************************************************************************
                                    mkfs
 *****************************************************************************/
/**
 * <Ring 1> Make a available Orange'S FS in the disk. It will
 *         - Write a super block to sector 1.
 *         - Create three special files: dev_tty0, dev_tty1, dev_tty2
 *         - Create the inode map
 *         - Create the sector map
 *         - Create the inodes of the files
 *         - Create '/', the root directory
 *****************************************************************************/
PRIVATE void mkfs()
{
        MESSAGE driver_msg;
        int i, j;

        int bits_per_sect = SECTOR_SIZE * 8; /* 8 bits per byte */

        /* get the geometry of ROOTDEV */
        struct part_info geo;
        driver_msg.type         = DEV_IOCTL;
        driver_msg.DEVICE       = MINOR(ROOT_DEV);
        driver_msg.REQUEST      = DIOCTL_GET_GEO;
        driver_msg.BUF          = &geo;
        driver_msg.PROC_NR      = TASK_FS;
        assert(dd_map[MAJOR(ROOT_DEV)].driver_nr != INVALID_DRIVER);
        send_recv(BOTH, dd_map[MAJOR(ROOT_DEV)].driver_nr, &driver_msg);

        printl("dev␣size:␣0x%x␣sectors\n", geo.size);

        /**************************/
        /*      super block       */
        /**************************/
        struct super_block sb;
        sb.magic          = MAGIC_V1;
        sb.nr_inodes      = bits_per_sect;
        sb.nr_inode_sects = sb.nr_inodes * INODE_SIZE / SECTOR_SIZE;
        sb.nr_sects       = geo.size; /* partition size in sector */
        sb.nr_imap_sects  = 1;
```

```
100     sb.nr_smap_sects  = sb.nr_sects / bits_per_sect + 1;
101     sb.n_1st_sect     = 1 + 1 +   /* boot sector & super block */
102             sb.nr_imap_sects + sb.nr_smap_sects + sb.nr_inode_sects;
103     sb.root_inode     = ROOT_INODE;
104     sb.inode_size     = INODE_SIZE;
105     struct inode x;
106     sb.inode_isize_off= (int)&x.i_size - (int)&x;
107     sb.inode_start_off= (int)&x.i_start_sect - (int)&x;
108     sb.dir_ent_size   = DIR_ENTRY_SIZE;
109     struct dir_entry de;
110     sb.dir_ent_inode_off = (int)&de.inode_nr - (int)&de;
111     sb.dir_ent_fname_off = (int)&de.name - (int)&de;
112
113     memset(fsbuf, 0x90, SECTOR_SIZE);
114     memcpy(fsbuf, &sb, SUPER_BLOCK_SIZE);
115
116     /* write the super block */
117     WR_SECT(ROOT_DEV, 1);
118
119     printl("devbase:0x%x00,␣sb:0x%x00,␣imap:0x%x00,␣smap:0x%x00\n"
120            "␣␣␣␣␣␣␣␣inodes:0x%x00,␣1st_sector:0x%x00\n",
121            geo.base * 2,
122            (geo.base + 1) * 2,
123            (geo.base + 1 + 1) * 2,
124            (geo.base + 1 + 1 + sb.nr_imap_sects) * 2,
125            (geo.base + 1 + 1 + sb.nr_imap_sects + sb.nr_smap_sects) * 2,
126            (geo.base + sb.n_1st_sect) * 2);
127
128     /**************************/
129     /*       inode map        */
130     /**************************/
131     memset(fsbuf, 0, SECTOR_SIZE);
132     for (i = 0; i < (NR_CONSOLES + 2); i++)
133             fsbuf[0] |= 1 << i;
134
135     assert(fsbuf[0] == 0x1F);/* 0001 1111 :
136                               *     | ||||
137                               *     | |||'--- bit 0 : reserved
138                               *     | ||'---- bit 1 : the first inode,
139                               *     | ||                which indicates '/'
140                               *     | |'----- bit 2 : /dev_tty0
141                               *     | '------ bit 3 : /dev_tty1
142                               *     '-------- bit 4 : /dev_tty2
143                               */
144     WR_SECT(ROOT_DEV, 2);
145
146     /**************************/
147     /*       secter map       */
148     /**************************/
149     memset(fsbuf, 0, SECTOR_SIZE);
150     int nr_sects = NR_DEFAULT_FILE_SECTS + 1;
151     /*                  ~~~~~~~~~~~~~~~~~~~|~   |
152      *                                     |   '--- bit 0 is reserved
153      */                                    '------- for '/'
154     for (i = 0; i < nr_sects / 8; i++)
155             fsbuf[i] = 0xFF;
156
157     for (j = 0; j < nr_sects % 8; j++)
158             fsbuf[i] |= (1 << j);
159
160     WR_SECT(ROOT_DEV, 2 + sb.nr_imap_sects);
161
162     /* zeromemory the rest sector-map */
163     memset(fsbuf, 0, SECTOR_SIZE);
164     for (i = 1; i < sb.nr_smap_sects; i++)
165             WR_SECT(ROOT_DEV, 2 + sb.nr_imap_sects + i);
166
```

```
        /***************************/
        /*      inodes             */
        /***************************/
        /* inode of '/' */
        memset(fsbuf, 0, SECTOR_SIZE);
        struct inode * pi = (struct inode*)fsbuf;
        pi->i_mode = I_DIRECTORY;
        pi->i_size = DIR_ENTRY_SIZE * 4; /* 4 files:
                                          * '.',
                                          * 'dev_tty0', 'dev_tty1', 'dev_tty2',
                                          */
        pi->i_start_sect = sb.n_1st_sect;
        pi->i_nr_sects = NR_DEFAULT_FILE_SECTS;
        /* inode of '/dev_tty0~2' */
        for (i = 0; i < NR_CONSOLES; i++) {
                pi = (struct inode*)(fsbuf + (INODE_SIZE * (i + 1)));
                pi->i_mode = I_CHAR_SPECIAL;
                pi->i_size = 0;
                pi->i_start_sect = MAKE_DEV(DEV_CHAR_TTY, i);
                pi->i_nr_sects = 0;
        }
        WR_SECT(ROOT_DEV, 2 + sb.nr_imap_sects + sb.nr_smap_sects);

        /***************************/
        /*          '/'            */
        /***************************/
        memset(fsbuf, 0, SECTOR_SIZE);
        struct dir_entry * pde = (struct dir_entry *)fsbuf;

        pde->inode_nr = 1;
        strcpy(pde->name, ".");

        /* dir entries of '/dev_tty0~2' */
        for (i = 0; i < NR_CONSOLES; i++) {
                pde++;
                pde->inode_nr = i + 2; /* dev_tty0's inode_nr is 2 */
                sprintf(pde->name, "dev_tty%d", i);
        }
        WR_SECT(ROOT_DEV, sb.n_1st_sect);
}

/*****************************************************************************
 *                                rw_sector
 *****************************************************************************/
/**
 * <Ring 1> R/W a sector via messaging with the corresponding driver.
 *
 * @param io_type  DEV_READ or DEV_WRITE
 * @param dev      device nr
 * @param pos      Byte offset from/to where to r/w.
 * @param bytes    r/w count in bytes.
 * @param proc_nr  To whom the buffer belongs.
 * @param buf      r/w buffer.
 *
 * @return Zero if success.
 *****************************************************************************/
PUBLIC int rw_sector(int io_type, int dev, u64 pos, int bytes, int proc_nr,
                     void* buf)
{
        MESSAGE driver_msg;

        driver_msg.type         = io_type;
        driver_msg.DEVICE       = MINOR(dev);
        driver_msg.POSITION     = pos;
        driver_msg.BUF          = buf;
        driver_msg.CNT          = bytes;
```

```
234             driver_msg.PROC_NR      = proc_nr;
235             assert(dd_map[MAJOR(dev)].driver_nr != INVALID_DRIVER);
236             send_recv(BOTH, dd_map[MAJOR(dev)].driver_nr, &driver_msg);
237
238             return 0;
239     }
```

代码9.14中我们修改了task_fs()，让它调用函数init_fs()（第37行），而init_fs()在打开ROOT_DEV之后调用了mkfs()（第57行），这便是建立文件系统的函数了。

mkfs()有一点长，它分这么几个部分：

- 向硬盘驱动程序索取ROOT_DEV的起始扇区和大小；
- 建立超级块（Super Block）；
- 建立 inode-map；
- 建立 sector-map；
- 写入 inode_array；
- 建立根目录文件。

我们先来说说根目录文件。它很类似于 FAT12 中的根目录区，但跟那里不同，在这里本质上它也是个文件。它有自己的 i-node，于是在 inode-map 中理所应当地占有一个位置。同时它跟一个普通文件一样占有一些连续的扇区，以便存放所有文件的列表。我们为它预留了NR_DEFAULT_FILE_SECTS个扇区，当然这些扇区应该能容纳足够多个dir_entry，至少不少于系统内 i-node 数目最大值。根据惯例，我们也称根目录为 ROOT，写作“/”。由于我们的文件系统是扁平的，没有子目录，于是所有文件都能写成“/xxxx”的形式。

好了，我们看代码9.14。开头索取ROOT_DEV起始扇区和大小的工作是由向硬盘驱动程序发送DEV_IOCTL消息来完成的，读者可同时参考代码9.12。

接下来是建立超级块，这时需要做一个决定，那就是文件系统内 i-node 数目的上限是多少。我们决定最多允许有 4096 个 i-node，这样只需要一个扇区来做 inode-map 就够了（4096 为一个扇区内的 bit 数）。这个决定同时意味着我们的文件系统最多容纳 4096 个文件。

另外还有两项是我们定义的宏，一个是MAGIC_V1，一个是ROOT_INODE。魔数是个标识而已，读者可以根据个人偏好指定一个，只要自己的操作系统认识并且不会跟其他文件系统冲突就行。ROOT_INODE是根目录文件的 i-node，这里我们指定为 1（定义在include/const.h中），第 0 号 i-node 我们保留起来，用以表示“不正确的 i-node”，或者“没有 inode”。

超级块占用一个扇区，除我们使用的各项之外，其余各字节我们初始化成了0x90，这样可以顺便测试一下写扇区功能。

再接下来是建立 inode-map 这个位图，它的每一位都映射到 inode_array 中的一个 i-node。如果一个 i-node 被使用了，inode-map 中相应的位就应该是 1，反之则为 0。第 0 个 i-node 被保留，这里置为 1，第 1 个是ROOT_INODE，也置

为1，很快我们就要填充具体的i-node。另外，我们还添加了三个文件，分别是dev_tty0、dev_tty1和dev_tty2，它们三个是特殊文件，我们将来会用到，这里先占个位置，也置为1。所以整个的inode-map目前只使用了第一个字节，其值为0x1F。

再下面轮到sector-map了，它的每一位表示一个扇区的使用情况。如果第*i*个扇区被使用了，那么sector-map中的第*i*位应为1，这跟inode-map是类似的。sector-map中的第0位也被保留，由于第一个能用作数据区的扇区是第sb.n_1st_sect块，所以sector-map的第1位理应对应sb.n_1st_sect，这在以后的编码过程中需要注意。由于根目录占用NR_DEFAULT_FILE_SECTS个扇区，加上一个保留位，我们总共需要置（NR_DEFAULT_FILE_SECTS+1）个1。

下面到了写具体的i-node了——我们一直称这块数据为inode_array，因为它其实也是个大数组（array）。注意写i-node的顺序一定要跟inode-map中一致，先是ROOT_INODE，然后是dev_tty[0,1,2]。

ROOT_INODE的i_mode为I_DIRECTORY，因为它是目录文件。目录中目前存在四个条目，分别为“.”、“dev_tty0”、“dev_tty1”和“dev_tty2”，所以它的i_size是DIR_ENTRY_SIZE×4。根目录文件是数据区的第一个文件，所以它的i_start_sect值为sb.n_1st_sect，i_nr_sects如前所述被赋值为NR_DEFAULT_FILE_SECTS。

dev_tty[0,1,2]三个文件的i_mode为I_CHAR_SPECIAL，我们可把它们称为字符设备特殊文件。关于什么是字符设备，我们以后再说明，这里需要了解的是，i_start_sect一项因为它们“特殊文件”的身份而代表了与普通文件完全不同的意义。在普通文件中，它表示文件开始于哪个扇区，而在这里，它表示文件代表设备的设备号。与此同时，i_size和i_nr_sects都赋值为零，因为在磁盘上它们都不占有任何空间。

根目录文件的具体内容就比较好理解了，只是要注意，每个文件要与相应的i-node对应起来。

在mk_fs()中，所有写入磁盘的内容都是先放进fsbuf这个缓冲区的。与通常的做法不同，我们这次没有定义一个数组，而是定义了一个指针，让它指向0x600000：

代码9.15 FS缓冲区（chapter9/d/kernel/global.c）

```
/**
 * 6MB~7MB: buffer for FS
 */
PUBLIC  u8 *            fsbuf           = (u8*)0x600000;
PUBLIC  const int       FSBUF_SIZE      = 0x100000;
```

也就是说，我们指定内存地址6MB~7MB为文件系统的缓冲区，一定程度上，这大概算是一种低级形态的“内存管理”吧。

在整个建立FS的过程中，写扇区的函数都是由WR_SECT这个宏来完成的，它的定义见代码9.16。

代码9.16 读写扇区的宏（chapter9/d/include/fs.h）

```
/**
 * Since all invocations of 'rw_sector()' in FS look similar (most of the
```

```
 * params are the same), we use this macro to make code more readable.
 */
#define RD_SECT(dev,sect_nr) rw_sector(DEV_READ,  \
                                       dev,  \
                                       (sect_nr) * SECTOR_SIZE,  \
                                       SECTOR_SIZE, /* read one sector */ \
                                       \
                                       TASK_FS,  \
                                       fsbuf);  \
                                       \
#define WR_SECT(dev,sect_nr) rw_sector(DEV_WRITE,  \
                                       dev,  \
                                       (sect_nr) * SECTOR_SIZE,  \
                                       SECTOR_SIZE, /* write one sector */ \
                                       \
                                       TASK_FS,  \
                                       fsbuf);
```

为了通用性着想，rw_sector()这个函数参数很多，使用一个宏可以让代码整洁一些。

好了，mkfs()已经写好了，现在我们就来执行一下，如图9.11所示。

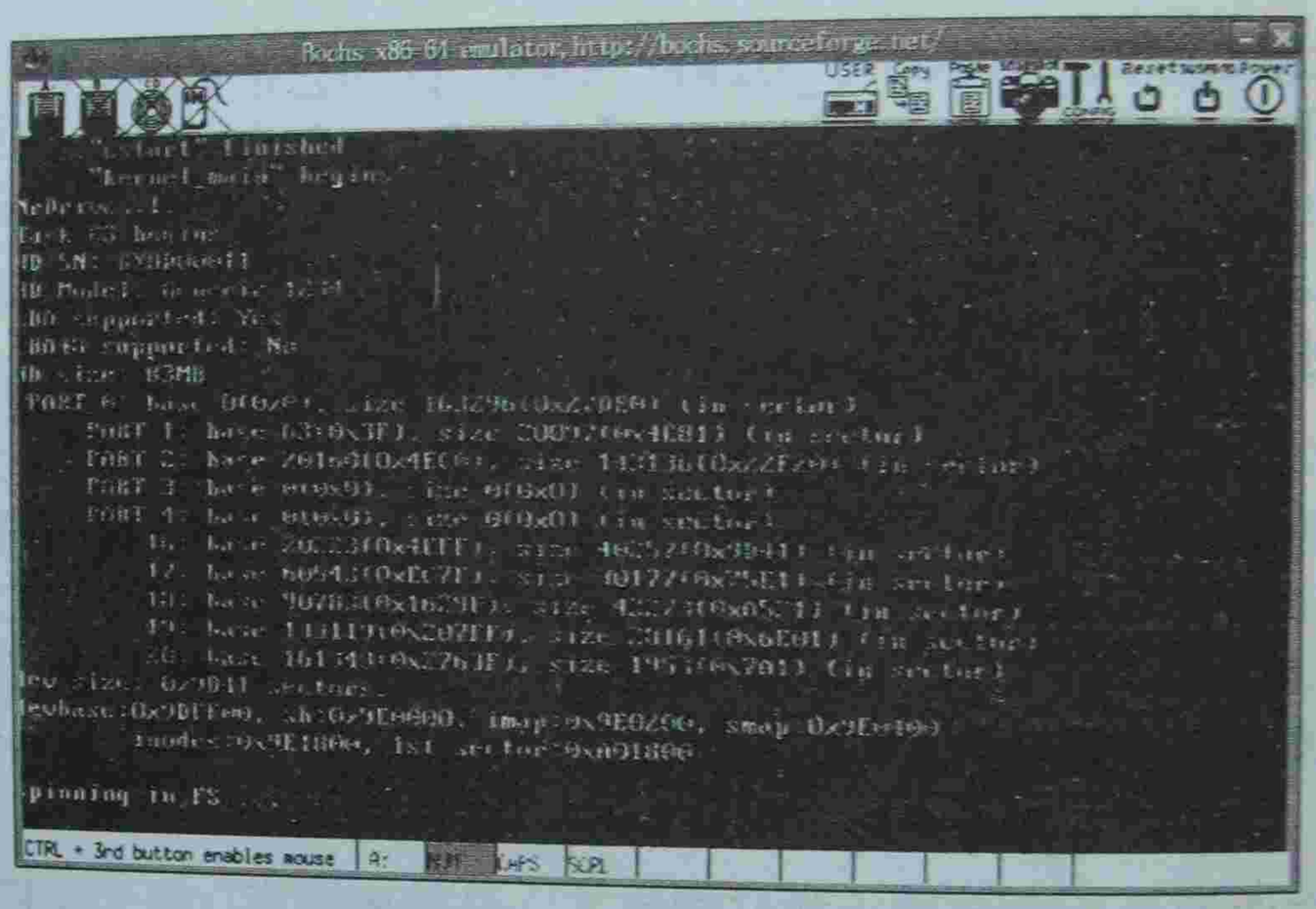


图 9.11 建立文件系统

根据运行的输出可知：

- 超级块开始于0x9E0000（字节偏移，下同）
- inode-map 开始于0x9E0200
- sector-map 开始于0x9E0400
- inode_array 开始于0x9E1800
- 根目录文件开始于0xA01800

下面我们就来看一看磁盘中的实际内容：

```
▷ xxd -u -a -g 1 -c 16 -s 0x9E0000 -l 512 80m.img
09e0000: 11 01 00 00 00 10 00 00 41 9D 00 00 01 00 00 00  ........A.......
09e0010: 0A 00 00 00 0D 01 00 00 00 01 00 00 01 00 00 00  ................
09e0020: 20 00 00 00 04 00 00 00 08 00 00 00 10 00 00 00  ................
09e0030: 00 00 00 00 04 00 00 00 90 90 90 90 90 90 90 90  ................
```

```
09e0040: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e0050: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e0060: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e0070: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e0080: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e0090: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e00a0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e00b0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e00c0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e00d0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e00e0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e00f0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e0100: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e0110: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e0120: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e0130: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e0140: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e0150: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e0160: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e0170: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e0180: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e0190: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e01a0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e01b0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e01c0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e01d0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e01e0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
09e01f0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
▷ xxd -u -a -g 1 -c 16 -s 0x9E0200 -l 512 80m.img
09e0200: 1F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
09e0210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
09e03f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
▷ xxd -u -a -g 1 -c 16 -s 0x9E0400 -l 512 80m.img
09e0400: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0410: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0420: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0430: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0440: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0450: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0460: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0470: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0480: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0490: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04a0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04b0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04c0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04d0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04e0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04f0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0500: 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
09e0510: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
09e05f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
▷ xxd -u -a -g 1 -c 16 -s 0x9E1800 -l 512 80m.img
09e1800: 00 40 00 00 40 00 00 00 0D 01 00 00 00 08 00 00  .@..@...........
09e1810: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
09e1820: 00 20 00 00 00 00 00 00 00 04 00 00 00 00 00 00  . ..............
09e1830: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
09e1840: 00 20 00 00 00 00 00 00 01 04 00 00 00 00 00 00  . ..............
09e1850: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
09e1860: 00 20 00 00 00 00 00 00 02 04 00 00 00 00 00 00  . ..............
09e1870: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
09e19f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
▷ xxd -u -a -g 1 -c 16 -s 0xA01800 -l 512 80m.img
0a01800: 01 00 00 00 2E 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
0a01810: 02 00 00 00 64 65 76 5F 74 74 79 30 00 00 00 00  ....dev_tty0....
0a01820: 03 00 00 00 64 65 76 5F 74 74 79 31 00 00 00 00  ....dev_tty1....
0a01830: 04 00 00 00 64 65 76 5F 74 74 79 32 00 00 00 00  ....dev_tty2....
0a01840: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
0a019f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

一下子，我们刚才写入磁盘的结果就呈现在眼前了，细心的读者可以自行对照每个结构体的成员，验证我们是不是写对了。

如果读者自己做试验的话，我倒建议你把这几个 xxd 的命令写成一个简单的脚本，这样在调试过程中，调用一个脚本，自己想看的磁盘内容就一下子打印出来，非常方便。

9.10 创建文件

一不小心，我们居然有了一个文件系统，真真切切地出现在我们的虚拟磁盘上。不过千万别庆祝得太早了，因为这仅仅是个静态的系统，我们还无法对文件进行添加、删除、修改等操作，而且现在磁盘上还没有一个真正的“普通文件”。不过不着急，有了这个初级形态的文件系统，我们只需要一步一步来就好了。

9.10.1 Linux 下的文件操作

要进行修改和删除操作，总要先有文件才行，所以第一步我们先写代码来创建文件。不过我们同时要考虑文件系统的对外接口，因为文件系统并不“主动”对文件进行操作，它总是根据用户进程的请求而做相应工作。我们不妨先回忆一下在 Linux 系统下用系统调用来进行文件操作的过程，请看代码9.17。

代码 9.17 Linux 下用系统调用读写文件（chapter9/e/tmp/f.c）

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/stat.h>
4  #include <fcntl.h>
5  #include <string.h>
6  #include <assert.h>
7
8  int main()
9  {
10         int fd;
11         int n;
12         const char filename[] = "blah";
13         const char bufw[] = "abcde";
14         const int rd_bytes = 3;
15         char bufr[rd_bytes];
16
17         assert(rd_bytes <= strlen(bufw));
18
19         /* create & write */
20         fd = open(filename, O_CREAT | O_RDWR | O_TRUNC, 0644);
21         if (fd == -1) {
22                 printf("failed to open %s\n", filename);
23                 return 1;
24         }
25
```

```
    n = write(fd, bufw, strlen(bufw));
    if (n != strlen(bufw)) {
            printf("failed to write to %s\n", filename);
            close(fd);
            return 2;
    }

    close(fd);

    /* open & read */
    fd = open(filename, O_RDWR);
    if (fd == -1) {
            printf("failed to open %s\n", filename);
            return 3;
    }

    n = read(fd, bufr, rd_bytes);
    if(n != rd_bytes) {
            printf("failed to read from %s\n", filename);
            close(fd);
            return 4;
    }
    bufr[n] = 0;
    printf("%d bytes read: %s\n", n, bufr);

    close(fd);

    return 0;
}
```

这是一段非常简单的对文件进行创建、读写以及关闭的代码，其中用到了open()、write()、read()、close()等几个系统调用。显然，如果我们的文件系统向用户进程提供服务的话，也要实现这些系统调用，所以我们最好模仿一下它们的行为。它们的声明如下：

```
int open(const char *pathname, int flags, mode_t mode);
ssize_t write(int fd, const void *buf, size_t count);
ssize_t read(int fd, void *buf, size_t count);
int close(int fd);
```

接下来，我们将逐步实现这些系统调用[1]，只是在功能上，我们将酌情进行简化。

9.10.2 文件描述符（file descriptor）

代码9.17中所有 Linux 系统调用都使用到了一个变量（或者返回值），那就是fd，即“file descriptor”，是它“代表”了一个文件，理所应当地充当了整个过程中最重要的角色。如果读者对它不甚了解的话，图9.12可以帮你建立一点初步的认识。

图9.12描述的是我们即将使用的文件操作方案。每个进程表中都将增加一个filp数组（代码9.18），其成员是指向 file descriptor（下文简称 fd）的指针。

[1]这里有两点需要说明一下，一是虽然在我们的系统里它们不是真正的系统调用，但根据习惯我们仍这样称呼它们；二是这里的write()跟第7章中用来打印字符的write()没有关系，那个write()函数已经在第8章中被我们弃用，并用printx()取而代之了。

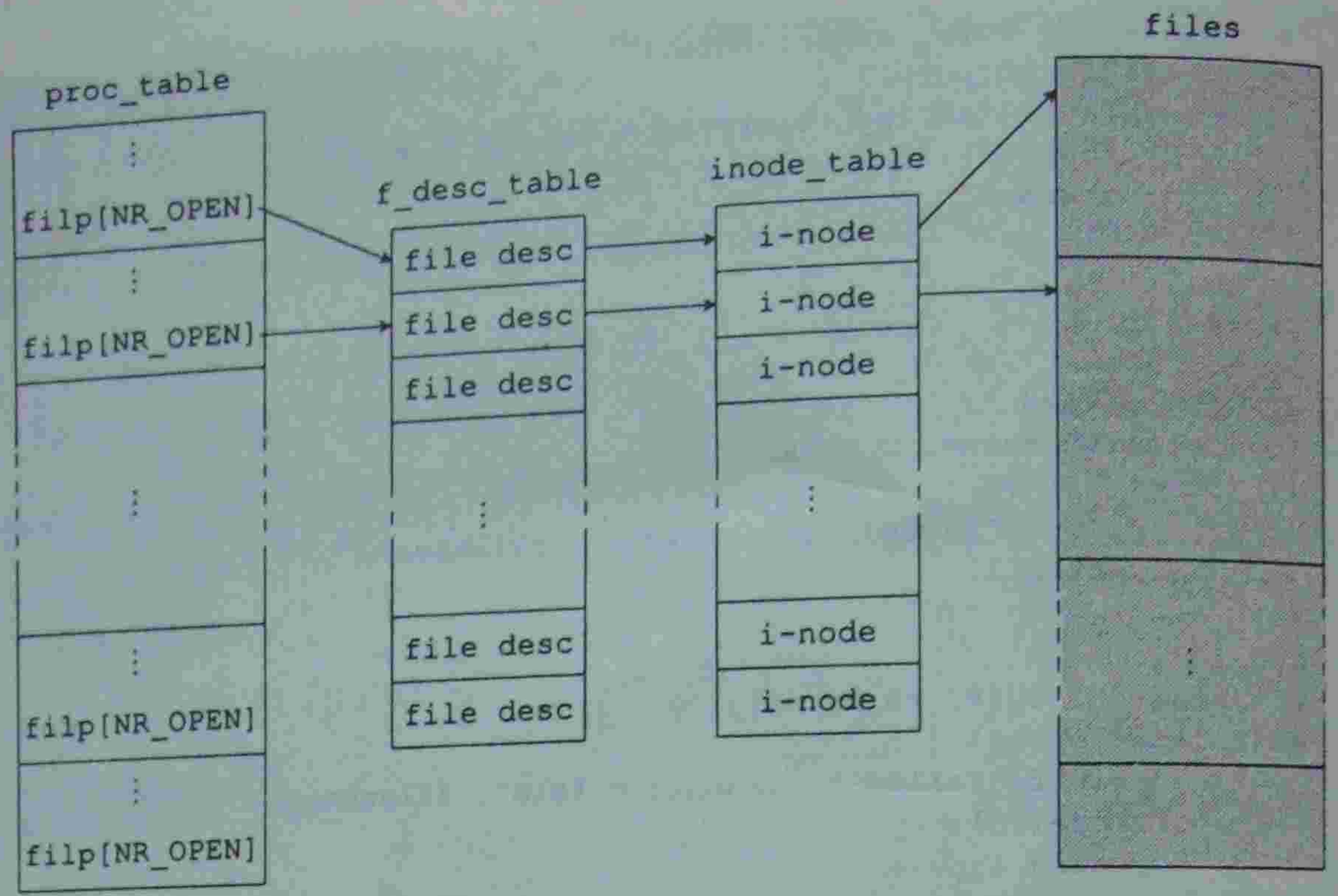


图 9.12 file descriptor

每一个使用中的 fd 都有一个指针指向一个inode结构体，而由这个inode结构体可以找到具体的文件。

代码 9.18 修改进程表（chapter9/e/include/sys/proc.h）

```
31  struct proc {
    ...
68          struct file_desc * filp[NR_FILES];
69  };
```

我们的 fd 的定义如代码9.19所示。

代码 9.19 file descriptor（chapter9/e/include/sys/fs.h）

```
121 /**
122  * @struct file_desc
123  * @brief  File Descriptor
124  */
125 struct file_desc {
126         int             fd_mode;        /**< R or W */
127         int             fd_pos;         /**< Current position for R/W. */
128         struct inode*   fd_inode;       /**< Ptr to the i-node */
129 };
```

这又是一个很简单的结构体，fd_mode用来记录这个 fd 是用来做什么操作的，比如是读、写，或是既读又写。fd_pos用来记录读写到了文件的什么位置。fd_inode便是指向inode的指针了。

每当一个进程打开一个文件——无论是打开一个已存在的还是创建一个新的，该进程的进程表的filp数组中就会分配一个位置——假设是k，用于存放打开文件的 fd 指针，而这个 k 就是返回给用户进程的open()函数的返回值了。这便是我们在代码9.17中看到的变量fd的真正含义——它其实是一个数组的下标，循着这个下标，系统可以找到用以描述文件的数据结构。

读者可能有疑问，既然三部分要连接起来，我们何不将它们合并起来？这个问题可以从两方面考虑。一方面f_desc_table[]和inode_table[]

不能合并，我们不能把fd_mode和fd_pos等值塞入i-node，因为不同的进程可以同时打开同一个文件，而且打开之后可能进行不同的操作——比如读取其不同的位置，所以合并后两个表是行不通的。另一方面，我们也不能将f_desc_table[]直接放入进程表，原因请参考第10.1.4节第424页。

9.10.3 open()

现在我们就可以着手编写open()了。我们先在用户进程中装模作样地创建一个文件：

代码 9.20 创建一个文件（chapter9/e/kernel/main.c）

```
130 void TestA()
131 {
132     int fd = open("/blah", O_CREAT);
133     printf("fd: %d\n", fd);
134     close(fd);
135     spin("TestA");
136 }
```

跟代码9.17中的open()系统调用不同，这里我们只用了两个参数，第三个参数我们干脆省了，访问权限等问题以后再考虑。

调用了open()，但我们压根儿还没这个函数呢，马上创建一个，见代码9.21。

代码 9.21 open()（chapter9/e/lib/open.c）

```
22 /*****************************************************************************
23  *                                open
24  *****************************************************************************/
25 /**
26  * open/create a file.
27  *
28  * @param pathname  The full path of the file to be opened/created.
29  * @param flags     O_CREAT, O_RDWR, etc.
30  *
31  * @return File descriptor if successful, otherwise -1.
32  *****************************************************************************/
33 PUBLIC int open(const char *pathname, int flags)
34 {
35     MESSAGE msg;
36
37     msg.type        = OPEN;
38
39     msg.PATHNAME    = (void*)pathname;
40     msg.FLAGS       = flags;
41     msg.NAME_LEN    = strlen(pathname);
42
43     send_recv(BOTH, TASK_FS, &msg);
44     assert(msg.type == SYSCALL_RET);
45
46     return msg.FD;
47 }
```

我们发了一个OPEN消息给文件系统，所以文件系统需要处理它：

代码 9.22 文件系统处理 OPEN 消息（chapter9/e/fs/main.c）

```
/*****************************************************************************
```

```
 *                                  task_fs
 *****************************************************************************/
/**
 * <Ring 1> The main loop of TASK FS.
 *
 *****************************************************************************/
PUBLIC void task_fs()
{
        printl("Task␣FS␣begins.\n");

        init_fs();

        while (1) {
                send_recv(RECEIVE, ANY, &fs_msg);

                int src = fs_msg.source;
                pcaller = &proc_table[src];

                switch (fs_msg.type) {
                case OPEN:
                        fs_msg.FD = do_open();
                        break;
                case CLOSE:
                        fs_msg.RETVAL = do_close();
                        break;

                }

                /* reply */
                fs_msg.type = SYSCALL_RET;
                send_recv(SEND, src, &fs_msg);
        }
}
```

这里我们用一个专门的函数do_open()来处理OPEN消息，见代码9.23。

代码9.23 do_open（chapter9/e/fs/open.c）

```
/*****************************************************************************
 *                                  do_open
 *****************************************************************************/
/**
 * Open a file and return the file descriptor.
 *
 * @return File descriptor if successful, otherwise a negative error code.
 *****************************************************************************/
PUBLIC int do_open()
{
        int fd = -1;            /* return value */

        char pathname[MAX_PATH];

        /* get parameters from the message */
        int flags = fs_msg.FLAGS;       /* access mode */
        int name_len = fs_msg.NAME_LEN; /* length of filename */
        int src = fs_msg.source;        /* caller proc nr. */
        assert(name_len < MAX_PATH);
        phys_copy((void*)va2la(TASK_FS, pathname),
                  (void*)va2la(src, fs_msg.PATHNAME),
                  name_len);
        pathname[name_len] = 0;

        /* find a free slot in PROCESS::filp[] */
        int i;
        for (i = 0; i < NR_FILES; i++) {
                if (pcaller->filp[i] == 0) {
                        fd = i;
```

```
                break;
            }

    }
    if ((fd < 0) || (fd >= NR_FILES))
            panic("filp[] is full (PID:%d)", proc2pid(pcaller));

    /* find a free slot in f_desc_table[] */
    for (i = 0; i < NR_FILE_DESC; i++)
            if (f_desc_table[i].fd_inode == 0)
                    break;
    if (i >= NR_FILE_DESC)
            panic("f_desc_table[] is full (PID:%d)", proc2pid(pcaller));

    int inode_nr = search_file(pathname);

    struct inode * pin = 0;
    if (flags & O_CREAT) {
            if (inode_nr) {
                    printl("file exists.\n");
                    return -1;
            }
            else {
                    pin = create_file(pathname, flags);
            }
    }
    else {
            assert(flags & O_RDWR);

            char filename[MAX_PATH];
            struct inode * dir_inode;
            if (strip_path(filename, pathname, &dir_inode) != 0)
                    return -1;
            pin = get_inode(dir_inode->i_dev, inode_nr);
    }

    if (pin) {
            /* connects proc with file_descriptor */
            pcaller->filp[fd] = &f_desc_table[i];

            /* connects file_descriptor with inode */
            f_desc_table[i].fd_inode = pin;

            f_desc_table[i].fd_mode = flags;
            /* f_desc_table[i].fd_cnt = 1; */
            f_desc_table[i].fd_pos = 0;

            int imode = pin->i_mode & I_TYPE_MASK;

            if (imode == I_CHAR_SPECIAL) {
                    MESSAGE driver_msg;
                    driver_msg.type = DEV_OPEN;
                    int dev = pin->i_start_sect;
                    driver_msg.DEVICE = MINOR(dev);
                    assert(MAJOR(dev) == 4);
                    assert(dd_map[MAJOR(dev)].driver_nr != INVALID_DRIVER);

                    send_recv(BOTH,
                              dd_map[MAJOR(dev)].driver_nr,
                              &driver_msg);

            }
            else if (imode == I_DIRECTORY) {
                    assert(pin->i_num == ROOT_INODE);
            }
            else {
                    assert(pin->i_mode == I_REGULAR);
            }
```

```
129             }
130             else {
131                     return -1;
132             }
133
134             return fd;
135     }
136
137     /*****************************************************************************
138      *                                create_file
139      *****************************************************************************/
140     /**
141      * Create a file and return it's inode ptr.
142      *
143      * @param[in] path   The full path of the new file
144      * @param[in] flags  Attribiutes of the new file
145      *
146      * @return           Ptr to i-node of the new file if successful, otherwise 0.
147      *
148      * @see open()
149      * @see do_open()
150      *****************************************************************************/
151     PRIVATE struct inode * create_file(char * path, int flags)
152     {
153             char filename[MAX_PATH];
154             struct inode * dir_inode;
155
156             if (strip_path(filename, path, &dir_inode) != 0)
157                     return 0;
158
159             int inode_nr = alloc_imap_bit(dir_inode->i_dev);
160
161             int free_sect_nr = alloc_smap_bit(dir_inode->i_dev,
162                                               NR_DEFAULT_FILE_SECTS);
163
164             struct inode *newino = new_inode(dir_inode->i_dev, inode_nr,
165                                              free_sect_nr);
166
167             new_dir_entry(dir_inode, newino->i_num, filename);
168
169             return newino;
170     }
```

do_open()首先是从消息内读出各项参数，其中需要格外注意的是文件名的读取。由于跨越了两个特权级，所以得到文件名需要付出额外的两份努力。一是需要事先记下文件名的长度，二是需要用phys_copy()来复制一份。

我们前面提到，open()要返回的是进程表filp[]内的一个索引，所以一开始我们就在filp[]内寻找一个空项，用来存放即将打开的文件的fd。由于filp[]内只保存指针，所以我们还要从f_desc_table[]中找一个空项。这两项工作做完之后，我们调用search_file()来看看要打开的文件是否已经存在。其中的具体细节我们暂时略过[2]，只需要知道这个函数将返回零，因为我们要创建的文件肯定不存在。

接下来是调用create_file()，这是真正用来创建文件的函数了。一个文件在文件系统中涉及的要素有五个：

- 文件内容（数据）所占用的扇区；
- i-node；

[2]或者读者可马上跳到第9.11.2节阅读其代码。

- i-node 在 inode-map 中占用的一位；
- 数据扇区在 sector-map 中占用的一位或多位；
- 文件在目录中占有的目录项（dir entry）。

相应地，我们创建一个文件，需要做以下几项工作：

- 为文件内容（数据）分配扇区；
- 在 inode_array 中分配一个 i-node；
- 在 inode-map 中分配一位；
- 在 sector-map 中分配一位或多位；
- 在相应目录中写入一个目录项（dir entry）。

在`create_file()`中，这几项工作分别是由四个函数来完成的：

- **`alloc_imap_bit()`** 在 inode-map 中分配一位，这也意味着新文件的 i-node 有了确定的位置。
- **`alloc_smap_bit()`** 在 sector-map 中分配多位，这也意味着为文件内容分配了扇区。
- **`new_inode()`** 在 inode_array 中分配一个 i-node，并写入内容。
- **`new_dir_entry()`** 在相应目录中写入一个目录项（dir entry）。

这四个函数的定义见代码9.24。

代码 9.24 创建文件需要的函数（chapter9/e/fs/open.c）

```
229 /*****************************************************************************
230  *                                alloc_imap_bit
231  *****************************************************************************/
232 /**
233  * Allocate a bit in inode-map.
234  *
235  * @param dev  In which device the inode-map is located.
236  *
237  * @return  I-node nr.
238  *****************************************************************************/
239 PRIVATE int alloc_imap_bit(int dev)
240 {
241         int inode_nr = 0;
242         int i, j, k;
243
244         int imap_blk0_nr = 1 + 1; /* 1 boot sector & 1 super block */
245         struct super_block * sb = get_super_block(dev);
246
247         for (i = 0; i < sb->nr_imap_sects; i++) {
248                 RD_SECT(dev, imap_blk0_nr + i);
249
250                 for (j = 0; j < SECTOR_SIZE; j++) {
251                         /* skip '11111111' bytes */
252                         if (fsbuf[j] == 0xFF)
253                                 continue;
```

```
                /* skip '1' bits */
                for (k = 0; ((fsbuf[j] >> k) & 1) != 0; k++) {}

                /* i: sector index; j: byte index; k: bit index */
                inode_nr = (i * SECTOR_SIZE + j) * 8 + k;
                fsbuf[j] |= (1 << k);

                /* write the bit to imap */
                WR_SECT(dev, imap_blk0_nr + i);
                break;
            }

            return inode_nr;
        }

        /* no free bit in imap */
        panic("inode-map is probably full.\n");

        return 0;
}

/*****************************************************************************
 *                                alloc_smap_bit
 *****************************************************************************/
/**
 * Allocate a bit in sector-map.
 *
 * @param dev  In which device the sector-map is located.
 * @param nr_sects_to_alloc  How many sectors are allocated.
 *
 * @return  The 1st sector nr allocated.
 *****************************************************************************/
PRIVATE int alloc_smap_bit(int dev, int nr_sects_to_alloc)
{
        /* int nr_sects_to_alloc = NR_DEFAULT_FILE_SECTS; */

        int i; /* sector index */
        int j; /* byte index */
        int k; /* bit index */

        struct super_block * sb = get_super_block(dev);

        int smap_blk0_nr = 1 + 1 + sb->nr_imap_sects;
        int free_sect_nr = 0;

        for (i = 0; i < sb->nr_smap_sects; i++) { /* smap_blk0_nr + i :
                                                     current sect nr. */
                RD_SECT(dev, smap_blk0_nr + i);

                /* byte offset in current sect */
                for (j = 0; j < SECTOR_SIZE && nr_sects_to_alloc > 0; j++) {
                        k = 0;
                        if (!free_sect_nr) {
                                /* loop until a free bit is found */
                                if (fsbuf[j] == 0xFF) continue;
                                for (; ((fsbuf[j] >> k) & 1) != 0; k++) {}
                                free_sect_nr = (i * SECTOR_SIZE + j) * 8 +
                                        k - 1 + sb->n_1st_sect;
                        }

                        for (; k < 8; k++) { /* repeat till enough bits are set */
                                assert(((fsbuf[j] >> k) & 1) == 0);
                                fsbuf[j] |= (1 << k);
                                if (--nr_sects_to_alloc == 0)
                                        break;
                        }
```

```
                }

                if (free_sect_nr) /* free bit found, write the bits to smap */
                        WR_SECT(dev, smap_blk0_nr + i);

                if (nr_sects_to_alloc == 0)
                        break;
        }

        assert(nr_sects_to_alloc == 0);

        return free_sect_nr;
}

/*****************************************************************************
 *                                new_inode
 *****************************************************************************/
/**
 * Generate a new i-node and write it to disk.
 *
 * @param dev  Home device of the i-node.
 * @param inode_nr  I-node nr.
 * @param start_sect  Start sector of the file pointed by the new i-node.
 *
 * @return  Ptr of the new i-node.
 *****************************************************************************/
PRIVATE struct inode * new_inode(int dev, int inode_nr, int start_sect)
{
        struct inode * new_inode = get_inode(dev, inode_nr);

        new_inode->i_mode = I_REGULAR;
        new_inode->i_size = 0;
        new_inode->i_start_sect = start_sect;
        new_inode->i_nr_sects = NR_DEFAULT_FILE_SECTS;

        new_inode->i_dev = dev;
        new_inode->i_cnt = 1;
        new_inode->i_num = inode_nr;

        /* write to the inode array */
        sync_inode(new_inode);

        return new_inode;
}

/*****************************************************************************
 *                                new_dir_entry
 *****************************************************************************/
/**
 * Write a new entry into the directory.
 *
 * @param dir_inode  I-node of the directory.
 * @param inode_nr   I-node nr of the new file.
 * @param filename   Filename of the new file.
 *****************************************************************************/
PRIVATE void new_dir_entry(struct inode *dir_inode,int inode_nr,char *filename)
{
        /* write the dir_entry */
        int dir_blk0_nr = dir_inode->i_start_sect;
        int nr_dir_blks = (dir_inode->i_size + SECTOR_SIZE) / SECTOR_SIZE;
        int nr_dir_entries =
                dir_inode->i_size / DIR_ENTRY_SIZE; /**
                                                     * including unused slots
                                                     * (the file has been
                                                     * deleted but the slot
                                                     * is still there)
                                                     */
```

```
388     int m = 0;
389     struct dir_entry * pde;
390     struct dir_entry * new_de = 0;
391
392     int i, j;
393     for (i = 0; i < nr_dir_blks; i++) {
394             RD_SECT(dir_inode->i_dev, dir_blk0_nr + i);
395
396             pde = (struct dir_entry *)fsbuf;
397             for (j = 0; j < SECTOR_SIZE / DIR_ENTRY_SIZE; j++,pde++) {
398                     if (++m > nr_dir_entries)
399                             break;
400
401                     if (pde->inode_nr == 0) { /* it's a free slot */
402                             new_de = pde;
403                             break;
404                     }
405             }
406             if (m > nr_dir_entries ||/* all entries have been iterated or */
407                 new_de)              /* free slot is found */
408                     break;
409     }
410     if (!new_de) { /* reached the end of the dir */
411             new_de = pde;
412             dir_inode->i_size += DIR_ENTRY_SIZE;
413     }
414     new_de->inode_nr = inode_nr;
415     strcpy(new_de->name, filename);
416
417     /* write dir block -- ROOT dir block */
418     WR_SECT(dir_inode->i_dev, dir_blk0_nr + i);
419
420     /* update dir inode */
421     sync_inode(dir_inode);
422 }
```

alloc_imap_bit()和alloc_smap_bit()都是对位图的操作，其实读者也可以根据自己的喜好把两个函数合二为一，因为它们之间有相似之处。值得注意的是在 sector-map 中分配扇区时我们没有考虑“空洞太小”的情况，或者说，我们一旦找到一个未使用的扇区，就认为以它为开头存在连续的nr_sects_to_alloc个扇区。我们这样假设是有道理的，一方面，参数nr_sects_to_alloc应该始终为NR_DEFAULT_FILE_SECTS或者它的倍数，另一方面，只要我们每次都这样分配磁盘，那么理论上，除非在一块磁盘的结尾处，我们不会遇到一处小于NR_DEFAULT_FILE_SECTS的连续扇区。

另外，我们对两个位图的操作中都是按位修改的，读者有兴趣可自行进行优化，比如可以以字节为单位或者以整数为单位进行操作。

new_inode()和new_dir_entry()相对简单一些，都是将相应的项赋值后马上写回磁盘，保持内存和磁盘的数据一致。

在do_open()中，调用create_file()之后文件其实就已经创建完毕了，之所以函数没有立刻结束，是因为创建完之后立即进行了“打开”操作，对照图9.12可知，代码9.23第98行到第106行是起到连接进程表、f_desc_table[]和inode_table[]的作用，三部分连接起来之后，文件就认为是被“打开”了。之后的第110行，我们对文件属性进行判断，如果是字符设备特殊文件的话就交给相应的驱动程序，至于交给哪个驱动程序是由文件的设备号决定的，它实际上应该是 TTY 进程。我们现在还没用过特殊文件，所以这段代码暂时不会执行，不过读者可以从这里看到主设备号的意义。

到这里，创建文件的主要过程已经很清楚了，最核心的函数其实就是由do_open()调用的create_file()。了解了这个主干，我们不如一鼓作气，看一看其中涉及的一些细枝末节。

9.11 创建文件所涉及的其他函数

9.11.1 strip_path()

首先看看create_file()中用到的strip_path()，见代码9.25。

代码 9.25 strip_path()（chapter9/e/fs/misc.c）

```
/*****************************************************************************
                                strip_path
 *****************************************************************************/
/**
 * Get the basename from the fullpath.
 *
 * In Orange'S FS v1.0, all files are stored in the root directory.
 * There is no sub-folder thing.
 *
 * This routine should be called at the very beginning of file operations
 * such as open(), read() and write(). It accepts the full path and returns
 * two things: the basename and a ptr of the root dir's i-node.
 *
 * e.g. After stip_path(filename, "/blah", ppinode) finishes, we get:
 *      - filename: "blah"
 *      - *ppinode: root_inode
 *      - ret val:  0 (successful)
 *
 * Currently an acceptable pathname should begin with at most one '/'
 * preceding a filename.
 *
 * Filenames may contain any character except '/' and '\\0'.
 *
 * @param[out] filename The string for the result.
 * @param[in]  pathname The full pathname.
 * @param[out] ppinode  The ptr of the dir's inode will be stored here.
 *
 * @return Zero if success, otherwise the pathname is not valid.
 *****************************************************************************/
PUBLIC int strip_path(char * filename, const char * pathname,
                      struct inode** ppinode)
{
        const char * s = pathname;
        char * t = filename;

        if (s == 0)
                return -1;

        if (*s == '/')
                s++;

        while (*s) {                    /* check each character */
                if (*s == '/')
                        return -1;
                *t++ = *s++;
                /* if filename is too long, just truncate it */
                if (t - filename >= MAX_FILENAME_LEN)
                        break;
        }
        *t = 0;
```

```
132         *ppinode = root_inode;
133
134         return 0;
135 }
136
```

虽然函数名有点晦涩，但代码做的事情很简单：把路径分成文件名和文件夹两部分。比如strip_path(filename, path, &dir_inode)中path是文件的路径，是输入，filename和dir_inode都是输出。函数调用成功之后，filename里面将包含“纯文件名”，即不含路径的文件名，dir_inode这个inode指针将指向文件所在文件夹的i-node。

一言蔽之，strip_path()的主要作用便是定位直接包含给定文件的文件夹，并得到给定文件在此文件夹中的名称。由于当前我们的文件系统是扁平的，所以这个函数返回之后dir_inode指向的将永远是根目录“/”的i-node。

9.11.2 search_file()

我们前面遇到这个函数时将它略过了，因为它其实是为下一步打开文件的操作而准备的，我们来看一看代码9.26。

代码 9.26 search_file()（chapter9/e/fs/misc.c）

```
27  /*****************************************************************************
28   *                                search_file
29   *****************************************************************************/
30  /**
31   * Search the file and return the inode nr.
32   *
33   * @param[in] path The full path of the file to search.
34   * @return         Ptr to the i-node of the file if successful, otherwise zero.
35   *
36   * @see open()
37   * @see do_open()
38   *****************************************************************************/
39  PUBLIC int search_file(char * path)
40  {
41          int i, j;
42
43          char filename[MAX_PATH];
44          memset(filename, 0, MAX_FILENAME_LEN);
45          struct inode * dir_inode;
46          if (strip_path(filename, path, &dir_inode) != 0)
47                  return 0;
48
49          if (filename[0] == 0)   /* path: "/" */
50                  return dir_inode->i_num;
51
52          /**
53           * Search the dir for the file.
54           */
55          int dir_blk0_nr = dir_inode->i_start_sect;
56          int nr_dir_blks = (dir_inode->i_size + SECTOR_SIZE - 1) / SECTOR_SIZE;
57          int nr_dir_entries =
58            dir_inode->i_size / DIR_ENTRY_SIZE; /**
59                                                 * including unused slots
60                                                 * (the file has been deleted
61                                                 * but the slot is still there)
62                                                 */
63          int m = 0;
64          struct dir_entry * pde;
```

```
        for (i = 0; i < nr_dir_blks; i++) {
                RD_SECT(dir_inode->i_dev, dir_blk0_nr + i);
                pde = (struct dir_entry *)fsbuf;
                for (j = 0; j < SECTOR_SIZE / DIR_ENTRY_SIZE; j++,pde++) {
                        if (memcmp(filename, pde->name, MAX_FILENAME_LEN) == 0)
                                return pde->inode_nr;
                        if (++m > nr_dir_entries)
                                break;
                }
                if (m > nr_dir_entries) /* all entries have been iterated */
                        break;
        }

        /* file not found */
        return 0;
}
```

我们还是通过strip_path()来得到文件所在目录的i-node，通过这个i-node来得到目录所在的扇区，然后读取这些扇区，查看里面是否有我们要找的文件，如果找到就返回文件的i-node，如果没找到就返回零。

9.11.3 get_inode()和sync_inode()

对于文件而言，i-node无疑是其灵魂，在对文件操作的过程中，从打开到关闭，我们始终需要面对i-node。在这里，我们用了一个缓冲区来存放系统中所有的i-node：inode_table[]。当我们需要用到一个inode时，我们就在inode_table[]中找一个位置将它放进去，所用到的函数便是这个get_inode()，请看代码9.27。

代码9.27 get_inode()（chapter9/e/fs/main.c）

```
/*****************************************************************************
 *                                get_inode
 *****************************************************************************/
/**
 * <Ring 1> Get the inode ptr of given inode nr. A cache -- inode_table[] -- is
 * maintained to make things faster. If the inode requested is already there,
 * just return it. Otherwise the inode will be read from the disk.
 *
 * @param dev Device nr.
 * @param num I-node nr.
 *
 * @return The inode ptr requested.
 *****************************************************************************/
PUBLIC struct inode * get_inode(int dev, int num)
{
        if (num == 0)
                return 0;

        struct inode * p;
        struct inode * q = 0;
        for (p = &inode_table[0]; p < &inode_table[NR_INODE]; p++) {
                if (p->i_cnt) { /* not a free slot */
                        if ((p->i_dev == dev) && (p->i_num == num)) {
                                /* this is the inode we want */
                                p->i_cnt++;
                                return p;
                        }
                }
                else {          /* a free slot */
                        if (!q) /* q hasn't been assigned yet */
```

```
                        q = p; /* q <- the 1st free slot */
                }
        }

        if (!q)
                panic("the inode table is full");

        q->i_dev = dev;
        q->i_num = num;
        q->i_cnt = 1;

        struct super_block * sb = get_super_block(dev);
        int blk_nr = 1 + 1 + sb->nr_imap_sects + sb->nr_smap_sects +
                ((num - 1) / (SECTOR_SIZE / INODE_SIZE));
        RD_SECT(dev, blk_nr);
        struct inode * pinode =
                (struct inode*)((u8*)fsbuf +
                                ((num - 1 ) % (SECTOR_SIZE / INODE_SIZE))
                                 * INODE_SIZE);

        q->i_mode = pinode->i_mode;
        q->i_size = pinode->i_size;
        q->i_start_sect = pinode->i_start_sect;
        q->i_nr_sects = pinode->i_nr_sects;
        return q;
}

/*****************************************************************************
 *                                put_inode
 *****************************************************************************/
/**
 * Decrease the reference nr of a slot in inode_table[]. When the nr reaches
 * zero, it means the inode is not used any more and can be overwritten by
 * a new inode.
 *
 * @param pinode I-node ptr.
 *****************************************************************************/
PUBLIC void put_inode(struct inode * pinode)
{
        assert(pinode->i_cnt > 0);
        pinode->i_cnt--;
}

/*****************************************************************************
 *                                sync_inode
 *****************************************************************************/
/**
 * <Ring 1> Write the inode back to the disk. Commonly invoked as soon as the
 *          inode is changed.
 *
 * @param p I-node ptr.
 *****************************************************************************/
PUBLIC void sync_inode(struct inode * p)
{
        struct inode * pinode;
        struct super_block * sb = get_super_block(p->i_dev);
        int blk_nr = 1 + 1 + sb->nr_imap_sects + sb->nr_smap_sects +
                ((p->i_num - 1) / (SECTOR_SIZE / INODE_SIZE));
        RD_SECT(p->i_dev, blk_nr);
        pinode = (struct inode*)((u8*)fsbuf +
                                (((p->i_num - 1) % (SECTOR_SIZE / INODE_SIZE))
                                 * INODE_SIZE));
        pinode->i_mode = p->i_mode;
        pinode->i_size = p->i_size;
        pinode->i_start_sect = p->i_start_sect;
        pinode->i_nr_sects = p->i_nr_sects;
        WR_SECT(p->i_dev, blk_nr);
}
```

如果一个inode已经被读入inode_table[]这个缓冲区了，那么下一次再需要它时，我们不需要再进行一次磁盘I/O，直接从缓冲区中读出来就可以了。在这里我们使用了比较原始的策略来保持磁盘和缓冲区的一致性：一旦内存中的值发生改变，则立即写入磁盘——这一过程由sync_inode()来完成。

对于缓冲区的管理是这样的：如果一个inode的i_cnt项为零，那么此项被认为是未使用，于是可以分配给新读入的i-node。一旦一个i-node读入，那么i_cnt自加。当i-node用完之后，使用者应调用一个put_inode()，这样i_cnt自减。当i_cnt自减至零时，说明不再有人继续使用这个i-node，它就又变成一个空项了。

9.11.4 init_fs()

init_fs()并不是一个新函数，不过其中增加了一些内容，见代码9.28。

代码 9.28 init_fs()（chapter9/e/fs/main.c）

```
89  /*****************************************************************************
90   *                                init_fs
91   *****************************************************************************/
92  /**
93   * <Ring 1> Do some preparation.
94   *
95   *****************************************************************************/
96  PRIVATE void init_fs()
97  {
98      int i;
99
100     /* f_desc_table[] */
101     for (i = 0; i < NR_FILE_DESC; i++)
102         memset(&f_desc_table[i], 0, sizeof(struct file_desc));
103
104     /* inode_table[] */
105     for (i = 0; i < NR_INODE; i++)
106         memset(&inode_table[i], 0, sizeof(struct inode));
107
108     /* super_block[] */
109     struct super_block * sb = super_block;
110     for (; sb < &super_block[NR_SUPER_BLOCK]; sb++)
111         sb->sb_dev = NO_DEV;
112
113     /* open the device: hard disk */
114     MESSAGE driver_msg;
115     driver_msg.type = DEV_OPEN;
116     driver_msg.DEVICE = MINOR(ROOT_DEV);
117     assert(dd_map[MAJOR(ROOT_DEV)].driver_nr != INVALID_DRIVER);
118     send_recv(BOTH, dd_map[MAJOR(ROOT_DEV)].driver_nr, &driver_msg);
119
120     /* make FS */
121     mkfs();
122
123     /* load super block of ROOT */
124     read_super_block(ROOT_DEV);
125
126     sb = get_super_block(ROOT_DEV);
127     assert(sb->magic == MAGIC_V1);
128
129     root_inode = get_inode(ROOT_DEV, ROOT_INODE);
130 }
```

函数开头初始化了三个缓冲区：f_desc_table[]、inode_table[]和super_block[]。前两者我们已经做了介绍，super_block[]使用方法类似，它用来存放超级块。每个分区都对应一个超级块，我们开始使用这个分区时（通常以“打开”设备为标志），会将其超级块读入内存，放入super_block[]这个缓冲区。之后我们可以由函数get_super_block()随时得到其指针。

中间部分是打开根设备和创建分区，这是前面已经做过的工作。然后第124行将根设备的超级块读入缓冲区，第129行将根目录文件的 i-node 读入缓冲区。对于一个扁平的文件系统，所有的文件都包含在根目录下面，所以根目录文件的 i-node 地位非常重要，这里我们用了一个全局变量root_inode来存放其指针。

9.11.5 read_super_block()和get_super_block()

前一节我们介绍了super_block[]，本节的两个函数便是操作它的接口，请看代码9.29。

代码 9.29 超级块缓冲区 ()（chapter9/e/fs/main.c）

```
314 /*****************************************************************************
315  *                                read_super_block
316  *****************************************************************************/
317 /**
318  * <Ring 1> Read super block from the given device then write it into a free
319  *          super_block[] slot.
320  *
321  * @param dev  From which device the super block comes.
322  *****************************************************************************/
323 PRIVATE void read_super_block(int dev)
324 {
325         int i;
326         MESSAGE driver_msg;
327 
328         driver_msg.type         = DEV_READ;
329         driver_msg.DEVICE       = MINOR(dev);
330         driver_msg.POSITION     = SECTOR_SIZE * 1;
331         driver_msg.BUF          = fsbuf;
332         driver_msg.CNT          = SECTOR_SIZE;
333         driver_msg.PROC_NR      = TASK_FS;
334         assert(dd_map[MAJOR(dev)].driver_nr != INVALID_DRIVER);
335         send_recv(BOTH, dd_map[MAJOR(dev)].driver_nr, &driver_msg);
336 
337         /* find a free slot in super_block[] */
338         for (i = 0; i < NR_SUPER_BLOCK; i++)
339                 if (super_block[i].sb_dev == NO_DEV)
340                         break;
341         if (i == NR_SUPER_BLOCK)
342                 panic("super_block slots used up");
343 
344         assert(i == 0); /* currently we use only the 1st slot */
345 
346         struct super_block * psb = (struct super_block *)fsbuf;
347 
348         super_block[i] = *psb;
349         super_block[i].sb_dev = dev;
350 }
351 
352 
353 /*****************************************************************************
```

```
/*****************************************************************************
 *                                get_super_block
 *****************************************************************************/
/**
 * <Ring 1> Get the super block from super_block[].
 *
 * @param dev Device nr.
 *
 * @return Super block ptr.
 *****************************************************************************/
PUBLIC struct super_block * get_super_block(int dev)
{
        struct super_block * sb = super_block;
        for (; sb < &super_block[NR_SUPER_BLOCK]; sb++)
                if (sb->sb_dev == dev)
                        return sb;

        panic("super block of devie %d not found.\n", dev);

        return 0;
}
```

代码9.29中，read_super_block()的作用是将一个设备的超级块读入缓存，get_super_block()可得到给定设备的超级块指针。在获取超级块时，我们采用了比较原始的遍历法来找到哪个是我们需要的。由于我们不可能有很多超级块同时存在于内存中，所以这样做其实并没有想像中那么笨拙。目前我们只有一个超级块，所以一次比较后函数就能返回。

9.12 关闭文件

跟创建文件比起来，关闭文件显得十分简单，由代码9.23可知，CLOSE消息是由do_close()来处理的，它的函数体如代码9.30所示。

代码 9.30 关闭文件（chapter9/e/fs/open.c）

```
/*****************************************************************************
 *                                do_close
 *****************************************************************************/
/**
 * Handle the message CLOSE.
 *
 * @return Zero if success.
 *****************************************************************************/
PUBLIC int do_close()
{
        int fd = fs_msg.FD;
        put_inode(pcaller->filp[fd]->fd_inode);
        pcaller->filp[fd]->fd_inode = 0;
        pcaller->filp[fd] = 0;

        return 0;
}
```

代码主要是三个语句：

- 调用put_inode()释放inode_table[]中的条目；
- 将filp[fd]->fd_inode清零释放f_desc_table[]中的条目；
- 将filp[fd]清零释放进程表中的fd条目。

经过这三个语句，之前为文件所分配的资源就全部释放了，文件也就可被认为是“关闭”了。

9.13 查看已创建的文件

终于完成了open()和close()两个系统调用，我们马上来运行一下，请看图9.13。

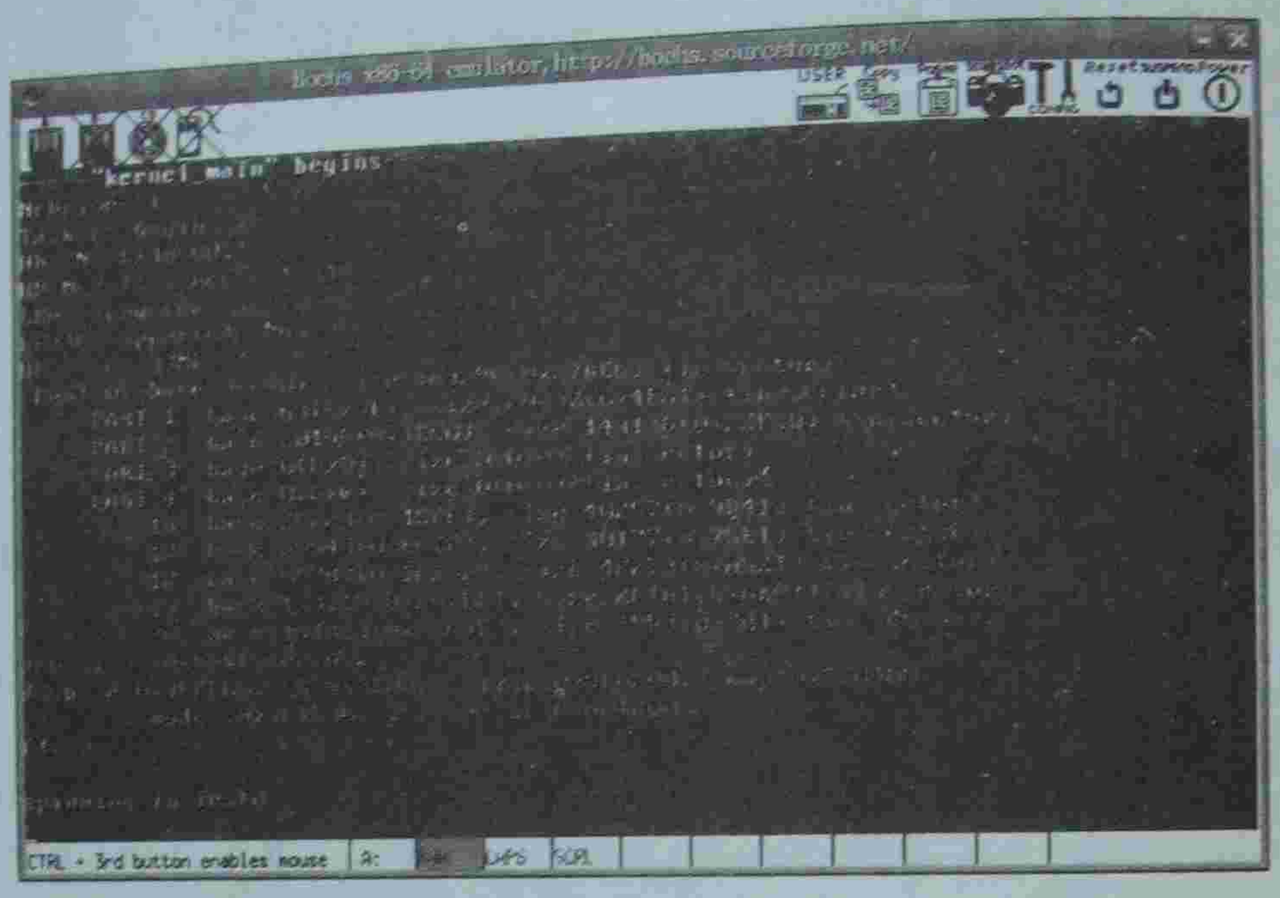


图 9.13 打开和关闭文件

可以看到，进程 TestA 打印出了新创建的文件的 fd：0。根据我们的代码，这个数字应该是正确的，不过文件到底有没有创建成功，我们还是要看看磁盘映像：

```
▷ xxd -u -a -g 1 -c 16 -s 0x9E0200 -l 512 80m.img
09e0200: 3F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ?...............
09e0210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
09e03f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
▷ xxd -u -a -g 1 -c 16 -s 0x9E0400 -l 1024 80m.img
09e0400: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0410: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0420: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0430: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0440: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0450: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0460: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0470: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0480: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0490: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04a0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04b0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04c0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04d0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04e0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04f0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0500: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0510: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0520: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0530: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
```

```
09e0540: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0550: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0560: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0570: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0580: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0590: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e05a0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e05b0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e05c0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e05d0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e05e0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e05f0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0600: 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
09e0610: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
09e07f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
▷ xxd -u -a -g 1 -c 16 -s 0x9E1800 -l 512 80m.img
09e1800: 00 40 00 00 50 00 00 00 0D 01 00 00 00 08 00 00  .@..P...........
09e1810: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
09e1820: 00 20 00 00 00 00 00 00 00 04 00 00 00 00 00 00  . ..............
09e1830: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
09e1840: 00 20 00 00 00 00 00 00 01 04 00 00 00 00 00 00  . ..............
09e1850: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
09e1860: 00 20 00 00 00 00 00 00 02 04 00 00 00 00 00 00  . ..............
09e1870: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
09e1880: 00 80 00 00 00 00 00 00 0D 09 00 00 00 08 00 00  ................
09e1890: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
09e19f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
▷ xxd -u -a -g 1 -c 16 -s 0xA01800 -l 512 80m.img
0a01800: 01 00 00 00 2E 00 00 00 00 00 00 00 00 00 00 00  ................
0a01810: 02 00 00 00 64 65 76 5F 74 74 79 30 00 00 00 00  ....dev_tty0....
0a01820: 03 00 00 00 64 65 76 5F 74 74 79 31 00 00 00 00  ....dev_tty1....
0a01830: 04 00 00 00 64 65 76 5F 74 74 79 32 00 00 00 00  ....dev_tty2....
0a01840: 05 00 00 00 62 6C 61 68 00 00 00 00 00 00 00 00  ....blah........
0a01850: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
0a019f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

可以看到，inode-map变成了二进制的“11 1111”（十六进制3Fh），文件系统创建之初这里是1Fh，这里第5位（从0开始数）由0变成了1。也就是说新增加了一个i-node，其编号为5。

我们马上看一下inode_array，从1开始数，数到5是9e1880h处，对照inode结构体的声明可知：

- i_mode为8000h（八进制0100000），这正是I_REGULAR的值。
- i_size为0，因为目前它还是个空文件。
- i_start_sect为90Dh。从超级块内容可知，sb.n_1st_sect为10Dh，由于第10Dh扇区占用sector-map中第1位（前文说过，第0位保留），所以第90Dh扇区占用第801h位。
- i_nr_sects为800h（十进制2048），正是NR_DEFAULT_FILE_SECTS的值，结合i_start_sect可知，文件在sector-map中占用第801h到第1000h位。

在文件系统刚刚建立起来的时候，sector-map中第0h位到第800h位被占，其中第0h位是保留位，第1h位到第800h位（共800h位）属于根目录文件，在

这里我们新建的文件“/blah”又占用了第801h位到第1000h位，换算出来，从第0字节到第1FFh字节（共计200h字节）都被占满，另外第200h字节被占用了1位。这跟我们得到的磁盘映像的信息是吻合的。

再来看一下根目录文件，很容易发现新增加的条目，其i-node号为5，这跟inode-map中是对应的，文件名是“blah”。所有信息都是正确的，这意味着我们的文件创建工作成功了！

读者想必也发现了，搞清楚i-node号和inode-map中的位置是件容易迷糊的事情，我们不妨再来理顺一下：

对根目录文件而言，i-node号为1（ROOT_INODE定义为1），在inode-map中占用第1位（从0开始数），具体i-node数据位于inode_array[0]。于是，第M号i-node在inode-map中占用第M位（从0开始数），具体i-node数据位于inode_array[M-1]。

我们还可以知道，inode_array中的第M项（从0开始数）对应第M+1号i-node以及inode-map中的第M+1位（从0开始数）。

与此类似，根目录区的开始扇区即为第sb.n_1st_sect扇区，占用sector-map中的第1位（从0开始数）。于是，第M扇区（以本分区的开始扇区为0扇区）对应sector-map中的第(M-super_block.n_1st_sect+1)位。同时setor-map中的第M位对应第(M-1+super_block.n_1st_sect)扇区。

9.14 打开文件

“创建”文件其实已经包含了“打开”操作，我们已经看到了。打开文件，其实就是根据文件名找到i-node，并且建立进程表、f_desc_table[]和inode_table[]这三个表之间的关联（参考图9.12）。

对于普通文件而言，“打开”操作有以下两种情况：

- 文件存在。这时我们获得文件的i-node号，读出i-node，建立前面所述三表的关联，并返回fd。
- 文件不存在。直接返回-1。

其实我们上一节说到的“创建”操作也是包含以下两种情况：

- 文件存在。返回-1。
- 文件不存在。创建文件，建立前面所述的关联，并返回fd。

这两种情况我们都已经说明了。下面，结合代码9.23，我们来看一看“打开”操作的两种情况。

在“打开”文件时，我们仍然是先调用search_file()来获取i-node号（代码9.23第75行）。文件存在与否便是由这个函数的返回值来决定了。如果search_file()返回值为0，说明文件不存在，否则应该返回一个大于0的i-node号。接下来我们通过i-node号使用get_inode()得到i-node指针（第94行），通过这一指针，我们就可以建立三表的关联了。

9.15 读写文件

由于我们使用“一次分配，终身使用”的扇区分配策略，所以文件读写变得非常容易，我们先来添加处理READ和WRITE消息的代码：

代码9.31 文件系统处理 READ 和 WRITE 消息（chapter9/f/fs/main.c）

```
/*****************************************************************************
                                task_fs
 *****************************************************************************/
/**
 * <Ring 1> The main loop of TASK FS.
 *
 *****************************************************************************/
PUBLIC void task_fs()
{
...
        switch (fs_msg.type) {
        case OPEN:
                fs_msg.FD = do_open();
                break;
        case CLOSE:
                fs_msg.RETVAL = do_close();
                break;
        case READ:
        case WRITE:
                fs_msg.CNT = do_rdwt();
                break;

...
}
```

读写两种消息由同一个函数do_rdwt()来处理，见代码9.32。

代码 9.32 do_rdwt（chapter9/f/fs/read_write.c）

```
/*****************************************************************************
                                do_rdwt
 *****************************************************************************/
/**
 * Read/Write file and return byte count read/written.
 *
 * Sector map is not needed to update, since the sectors for the file have been
 * allocated and the bits are set when the file was created.
 *
 * @return How many bytes have been read/written.
 *****************************************************************************/
PUBLIC int do_rdwt()
{
        int fd = fs_msg.FD;     /**< file descriptor. */
        void * buf = fs_msg.BUF;/**< r/w buffer */
        int len = fs_msg.CNT;   /**< r/w bytes */
                                                /* caller proc nr. */
        int src = fs_msg.source;

        assert((pcaller->filp[fd] >= &f_desc_table[0]) &&
               (pcaller->filp[fd] < &f_desc_table[NR_FILE_DESC]));

        if (!(pcaller->filp[fd]->fd_mode & O_RDWR))
                return -1;

        int pos = pcaller->filp[fd]->fd_pos;

        struct inode * pin = pcaller->filp[fd]->fd_inode;

        assert(pin >= &inode_table[0] && pin < &inode_table[NR_INODE]);
```

```
        int imode = pin->i_mode & I_TYPE_MASK;

        if (imode == I_CHAR_SPECIAL) {
                int t = fs_msg.type == READ ? DEV_READ : DEV_WRITE;
                fs_msg.type = t;

                int dev = pin->i_start_sect;
                assert(MAJOR(dev) == 4);

                fs_msg.DEVICE   = MINOR(dev);
                fs_msg.BUF      = buf;
                fs_msg.CNT      = len;
                fs_msg.PROC_NR  = src;
                assert(dd_map[MAJOR(dev)].driver_nr != INVALID_DRIVER);
                send_recv(BOTH, dd_map[MAJOR(dev)].driver_nr, &fs_msg);
                assert(fs_msg.CNT == len);

                return fs_msg.CNT;
        }
        else {
                assert(pin->i_mode == I_REGULAR || pin->i_mode == I_DIRECTORY);
                assert((fs_msg.type == READ) || (fs_msg.type == WRITE));

                int pos_end;
                if (fs_msg.type == READ)
                        pos_end = min(pos + len, pin->i_size);
                else                    /* WRITE */
                        pos_end = min(pos + len, pin->i_nr_sects * SECTOR_SIZE);

                int off = pos % SECTOR_SIZE;
                int rw_sect_min=pin->i_start_sect+(pos>>SECTOR_SIZE_SHIFT);
                int rw_sect_max=pin->i_start_sect+(pos_end>>SECTOR_SIZE_SHIFT);

                int chunk = min(rw_sect_max - rw_sect_min + 1,
                                FSBUF_SIZE >> SECTOR_SIZE_SHIFT);

                int bytes_rw = 0;
                int bytes_left = len;
                int i;
                for (i = rw_sect_min; i <= rw_sect_max; i += chunk) {
                        /* read/write this amount of bytes every time */
                        int bytes = min(bytes_left, chunk * SECTOR_SIZE - off);
                        rw_sector(DEV_READ,
                                  pin->i_dev,
                                  i * SECTOR_SIZE,
                                  chunk * SECTOR_SIZE,
                                  TASK_FS,
                                  fsbuf);

                        if (fs_msg.type == READ) {
                                phys_copy((void*)va2la(src, buf + bytes_rw),
                                          (void*)va2la(TASK_FS, fsbuf + off),
                                          bytes);
                        }
                        else {  /* WRITE */
                                phys_copy((void*)va2la(TASK_FS, fsbuf + off),
                                          (void*)va2la(src, buf + bytes_rw),
                                          bytes);

                                rw_sector(DEV_WRITE,
                                          pin->i_dev,
                                          i * SECTOR_SIZE,
                                          chunk * SECTOR_SIZE,
                                          TASK_FS,
                                          fsbuf);
                        }
```

```
            off = 0;
            bytes_rw += bytes;
            pcaller->filp[fd]->fd_pos += bytes;
            bytes_left -= bytes;
        }

        if (pcaller->filp[fd]->fd_pos > pin->i_size) {
            /* update inode::size */
            pin->i_size = pcaller->filp[fd]->fd_pos;

            /* write the updated i-node back to disk */
            sync_inode(pin);
        }

        return bytes_rw;
    }
}
```

在读写的过程中，我们仍然照顾到了字符设备特殊文件。跟前面一样，我们仍然是把它扔给相应的驱动程序——虽然驱动程序并未准备好处理，但发送一个消息只是举手之劳，我们不妨先把它添上。

读写普通文件时，file_desc结构体的成员悉数到场（可参考代码9.19）。首先是对fd_mode进行简单的判断（第46行），这其实是判断open()函数调用时是否传入了正确的flags参数，因为fd_mode是从那里得来的。我们对flags的可选值进行了简化，它的可选值只有两个：O_CREAT和O_RDWR，要想读写文件，调用open()时需要加上O_RDWR。

fd_pos的用途在于记录读写到文件的哪个位置，类似于一个书签，在文件刚打开时它被置为0。

fd_inode所指向的便是被操作文件的i-node了，我们通过它获得文件的开始扇区、文件类型，以及大小等信息。

真正的读写过程从第78行开始。变量pos表示开始读写的位置，pos_end表示结束读写的位置，读操作时pos_end不能越过文件已有的大小，写操作时pos_end不能越过为文件分配的最大空间。通过pos和pos_end，我们可以计算出读/写操作所涉及的扇区边界，这里用rw_sect_min和rw_sect_max表示。计算时用右移操作代替除法运算，右移SECTOR_SIZE_SHIFT位相当于被SECTOR_SIZE除。

对扇区的读写以chunk为单位，最大不能超过为fsbuf分配的空间。

需要注意，不仅读操作，写操作也需要先将目标扇区读出，因为读写都是以扇区为单位的，而写操作可以在文件的任意位置进行，所以以扇区为单位的上下文需要先行读出。写操作的另一特殊之处在于它可能改变文件大小，所以返回之前要检查这一点，如果文件大小被改变，则需更新i-node。

现在FS能处理READ和WRITE消息了，我们马上写两个函数：read()和write()，以便用户进程使用，见代码9.33和9.34。

代码9.33 read()（chapter9/f/lib/read.c）

```
/*****************************************************************************
 *                                read
 *****************************************************************************/
/**
 * Read from a file descriptor.
 *
```

```
28    * @param fd     File descriptor.
29    * @param buf    Buffer to accept the bytes read.
30    * @param count  How many bytes to read.
31    *
32    * @return  On success, the number of bytes read are returned.
33    *          On error, -1 is returned.
34    *****************************************************************************/
35   PUBLIC int read(int fd, void *buf, int count)
36   {
37           MESSAGE msg;
38           msg.type = READ;
39           msg.FD   = fd;
40           msg.BUF  = buf;
41           msg.CNT  = count;
42
43           send_recv(BOTH, TASK_FS, &msg);
44
45           return msg.CNT;
46   }
```

代码 9.34 write()（chapter9/f/lib/write.c）

```
22   /*****************************************************************************
23    *                                write
24    *****************************************************************************/
25   /**
26    * Write to a file descriptor.
27    *
28    * @param fd     File descriptor.
29    * @param buf    Buffer including the bytes to write.
30    * @param count  How many bytes to write.
31    *
32    * @return  On success, the number of bytes written are returned.
33    *          On error, -1 is returned.
34    *****************************************************************************/
35   PUBLIC int write(int fd, const void *buf, int count)
36   {
37           MESSAGE msg;
38           msg.type = WRITE;
39           msg.FD   = fd;
40           msg.BUF  = (void*)buf;
41           msg.CNT  = count;
42
43           send_recv(BOTH, TASK_FS, &msg);
44
45           return msg.CNT;
46   }
```

9.16 测试文件读写

好了，如今可以读写文件了，我们马上来测试一下，修改 TestA（见代码9.35）。

代码 9.35 读写文件（chapter9/f/kernel/main.c）

```
130  void TestA()
131  {
132          int fd;
133          int n;
134          const char filename[] = "blah";
135          const char bufw[] = "abcde";
```

```
    const int rd_bytes = 3;
    char bufr[rd_bytes];

    assert(rd_bytes <= strlen(bufw));

    /* create */
    fd = open(filename, O_CREAT | O_RDWR);
    assert(fd != -1);
    printf("File␣created.␣fd:␣%d\n", fd);

    /* write */
    n = write(fd, bufw, strlen(bufw));
    assert(n == strlen(bufw));

    /* close */
    close(fd);

    /* open */
    fd = open(filename, O_RDWR);
    assert(fd != -1);
    printf("File␣opened.␣fd:␣%d\n", fd);

    /* read */
    n = read(fd, bufr, rd_bytes);
    assert(n == rd_bytes);
    bufr[n] = 0;
    printf("%d␣bytes␣read:␣%s\n", n, bufr);

    /* close */
    close(fd);

    spin("TestA");
}
```

除了出错处理的方式不同，现在的 TestA 和代码9.17基本是一样的，我们来编译运行一下看看（图9.14）。

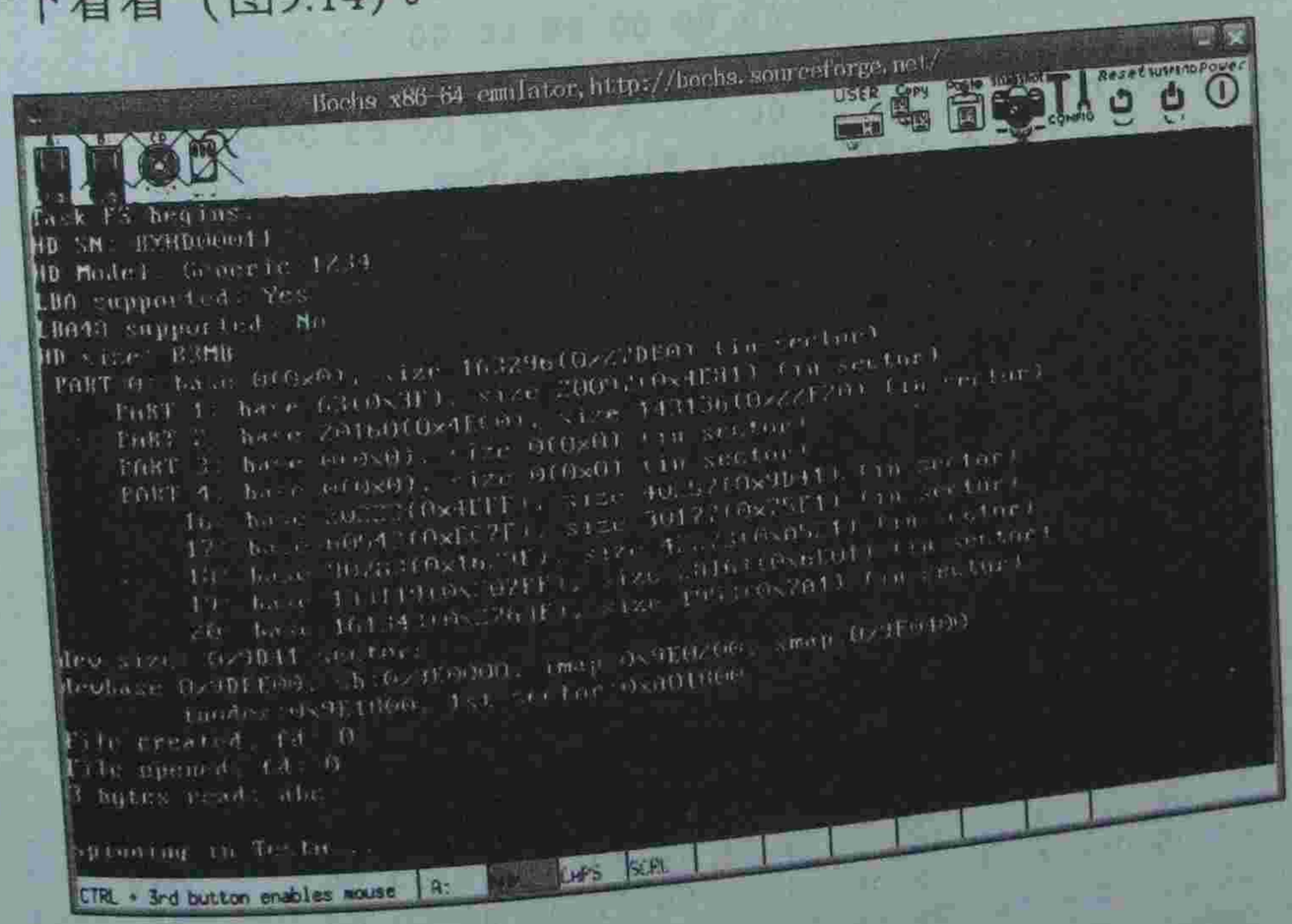


图 9.14 读写文件

从运行结果来看，文件读操作成功了，“abc”三个字符被读出，为保险起见，我们还是深入磁盘映像，看一下如今的“/blah”文件变成什么样了：

```
▷ xxd -u -a -g 1 -c 16 -s 0x9E0200 -l 512 80m.img
09e0200: 3F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ?...............
```

```
09e0210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
09e03f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
> xxd -u -a -g 1 -c 16 -s 0x9E0400 -l 1024 80m.img
09e0400: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0410: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0420: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0430: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0440: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0450: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0460: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0470: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0480: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0490: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04a0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04b0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04c0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04d0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04e0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e04f0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0500: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0510: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0520: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0530: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0540: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0550: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0560: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0570: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0580: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0590: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e05a0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e05b0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e05c0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e05d0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e05e0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e05f0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ................
09e0600: 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
09e0610: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
09e07f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
> xxd -u -a -g 1 -c 16 -s 0x9E1800 -l 512 80m.img
09e1800: 00 40 00 00 50 00 00 00 0D 01 00 00 00 08 00 00  .@..P...........
09e1810: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
09e1820: 00 20 00 00 00 00 00 00 00 04 00 00 00 00 00 00  . ..............
09e1830: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
09e1840: 00 20 00 00 00 00 00 00 01 04 00 00 00 00 00 00  . ..............
09e1850: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
09e1860: 00 20 00 00 00 00 00 00 02 04 00 00 00 00 00 00  . ..............
09e1870: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
09e1880: 00 80 00 00 05 00 00 00 0D 09 00 00 00 08 00 00  ................
09e1890: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
09e19f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
> xxd -u -a -g 1 -c 16 -s 0xA01800 -l 512 80m.img
0a01800: 01 00 00 00 2E 00 00 00 00 00 00 00 00 00 00 00  ................
0a01810: 02 00 00 00 64 65 76 5F 74 74 79 30 00 00 00 00  ....dev_tty0....
0a01820: 03 00 00 00 64 65 76 5F 74 74 79 31 00 00 00 00  ....dev_tty1....
0a01830: 04 00 00 00 64 65 76 5F 74 74 79 32 00 00 00 00  ....dev_tty2....
0a01840: 05 00 00 00 62 6C 61 68 00 00 00 00 00 00 00 00  ....blah........
0a01850: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
0a019f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

理所应当地，inode-map、sector-map以及根目录区没有任何改变，只是“/blah”的i-node变了，i_size变成了5，这是正确的，因为我们写入了5个字节。

接下来还要看一看“/blah”实际占用的扇区中数据的情况。从 i-node 中可知，文件的开始扇区号是90Dh，将它与分区的开始扇区号4EFF相加，得到“/blah”占用的首扇区的 LBA：580Ch，将它乘以200h（512 的十六进制），得到B01800h，这便是“/blah”数据扇区的字节偏移了，让我们来看看里面的内容：

```
▷ xxd -u -a -g 1 -c 16 -s 0xB01800 -l 512 80m.img
0b01800: 61 62 63 64 65 00 00 00 00 00 00 00 00 00 00 00  abcde...........
0b01810: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
0b019f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

我们看到了“abcde”！写入操作也成功了！

虽然目前的测试还远远不够充分，但我想我们有足够的理由庆贺一番了，因为有了创建和读写功能，我们的文件系统就算是具备雏形了。

9.17 文件系统调试

在通常情况下，随着代码越来越多，调试也会越来越难，但也不尽然，因为调试的手段也在同步增加。比如现在，我们有了硬盘驱动和文件系统，带来的直接便利是我们可以开始写 log 了。由于我们的文件系统还比较初级，所以 log 可以直接通过硬盘驱动写入某个扇区，之后回到 Linux 下用工具将它取出来就可以了。

这种方法产生的效果甚至可能超出你的想像，不信请看看图9.15。

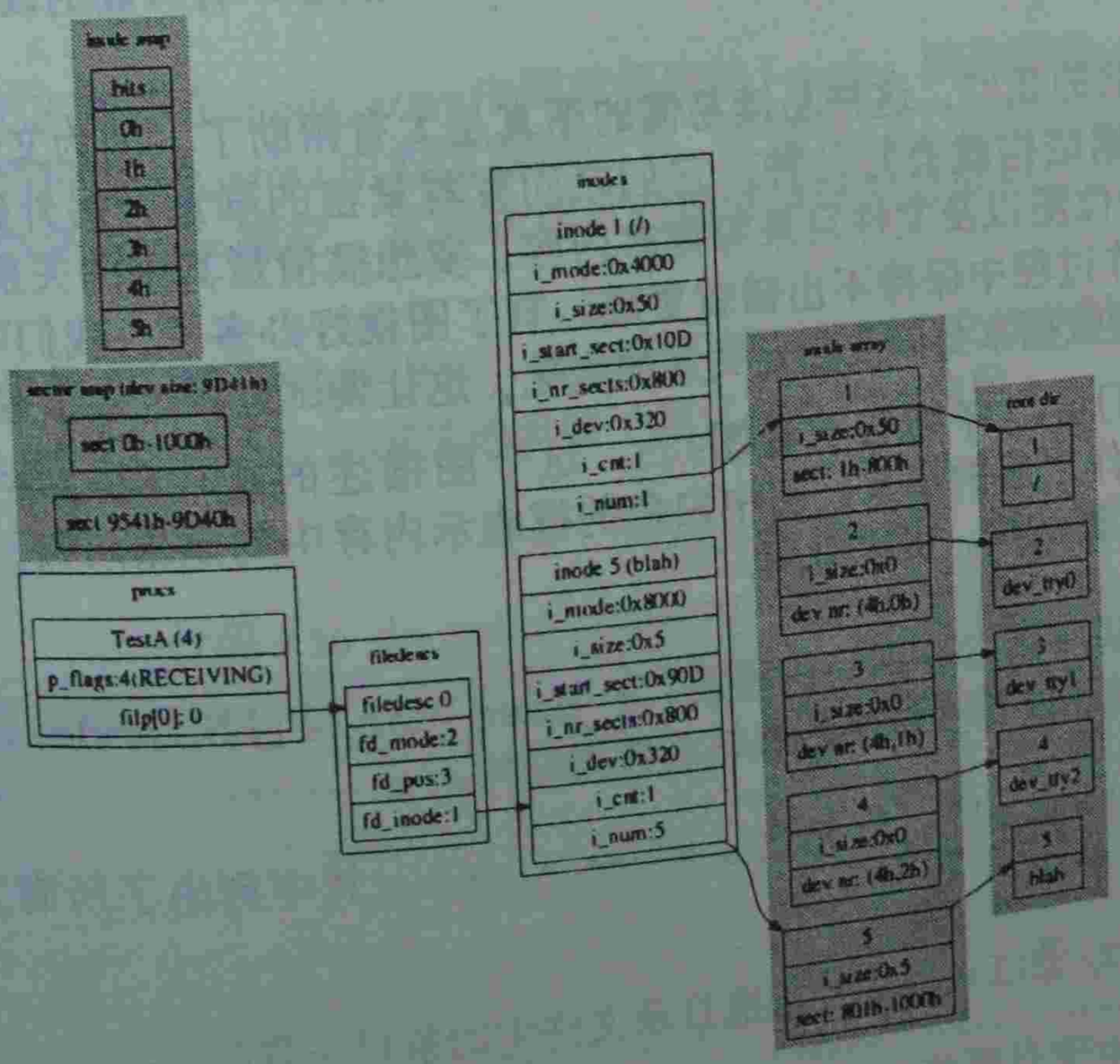


图 9.15 文件系统快照

千万不要误会，这个图不是我用 GIMP 或是什么工具画的，它是我们的 log。没错，一分钟之前你可能都想不到，我们的 log 可以酷到这种程度！

为什么我们的 log 会是一个图呢？这图当然不是我们自己写程序做出来的。其中的奥秘在于，我们可以将 log 写成某种特定的格式——比如这里存成 DOT 源文件，关闭虚拟机之后，我们进行以下操作：

1. 用dd命令将磁盘映像中我们写入的 log 扇区抽取出来，存成本地的文件，假设为x.log。

2. 用 bash 脚本或现成的工具将x.log中的 dot 部分抽取出来，存成一个或多个文件，比如a.dot、b.dot、c.dot……

3. 用 graphviz 包里面的工具将这些 dot 文件转成你喜欢的可视化文件格式，比如 png。

4. 查看图片——如果你不断地单击图像查看器的“下一幅”按钮，甚至可以看到动画效果。

如果你把这些步骤写成脚本[3]，只要一个命令，图像就呈现在眼前了。

其中用到的 DOT，可算得上一种语言，它是 Graphviz 的一部分。Graphviz 是 AT&T 实验室开发的一个软件包[4]，可将 DOT 源文件转化成图像。DOT 的语法非常简单，读者可以在十分钟之内掌握基本的画图方法，而且对我们而言，画画方框和箭头就足够了。

我们的画图代码位于chapter9/g/fs/disklog.c之中，代码比较烦琐，但没有任何难点，往磁盘上写字符串而已，这我们已经很熟悉。有兴趣的读者可以自行查看。

对于当前的我们，这种易读易懂的图真是太有帮助了，因为文件系统中涉及的主要数据结构就有好几种，它们之间有着紧密的联系。另外磁盘上也有 i-node、两种位图以及文件目录等诸多内容，要想搞清楚其中的关系已是不易，要在文件操作过程中保持不出错就更难。有了图就好办多了，我们可以多画几张，随时掌握数据本身以及相互关系的变化，这让调试过程也多了些趣味。

比如我们可以仔细看一看图9.15。这个图描述的是代码9.35中刚刚执行完read()操作之后的情景。图中白色的框表示内存中的数据结构，灰色框表示硬盘上的数据结构。可以看到：

- 白色框从左到右分别为进程表、f_desc_table[]和inode_table[]，它们构成了文件系统中的最核心数据结构。
- inode_table[]对应磁盘上的 inode_array，其中列出了所有文件的 i-node。
- inode_array 通过 i-node 号与根目录文件中的条目关联起来。
- inode-map 中有哪些位被占用。

[3]读者可参考chapter9/g/genlog，这是个简单的脚本示例。
[4]主页位于 http://www.graphviz.org/。

- sector-map 中哪些区域被占用，其中9541h~9D40h便是 log 所在的区域了，我们在 sector-map 中将这些位置 1 以免被普通文件误用。
- 根目录文件的 i-node 在inode_table[]中占了一项，因为在init_fs()中我们调用get_inode()给 root_inode 赋了值（详见第9.11.4节）。

这个图是read()执行完之后系统内数据的快照，一下子我们就能看出，PID 为 4 的进程 TestA 已经打开了文件“/blah”，并且读到了位置 3（fd_pos为 3）。

由于我们的 log 是将各数据结构的值直接写入磁盘，它只依赖于硬盘驱动程序，独立于文件系统逻辑，所以它是可信赖的，可帮助我们调试文件系统。

9.18 删除文件

有了新的调试手段，写起代码来就更加自信了。我们下面就来添加代码来删除文件。

删除是添加的反过程，所以要删除文件，我们需要做以下工作（读者可同时参考第373页中创建文件的过程）：

- 释放 inode-map 中的相应位。
- 释放 sector-map 中的相应位。
- 删除根目录中的目录项。

注意我们不需要在 inode_array 中释放相应的 i-node，因为释放 inode-map 中的相应位已经将 inode_array 中的位置标记为“未使用”了，不过在接下来的代码中我们仍然将 i-node 清空，这样在调试时能够清晰地看到相应的 i-node 不见了。

另外从最简单的删除功能出发，我们并不需要释放为文件分配的扇区[5]，因为 sector-map 中的 1 和 0 已经清楚地表明了扇区是否可以使用。

我们来看看代码9.36。

代码 9.36 do_unlink（chapter9/h/fs/link.c）

```
24 /*****************************************************************************
25  *                                do_unlink
26  *****************************************************************************/
27 /**
28  * Remove a file.
29  *
30  * @note We clear the i-node in inode_array[] although it is not really needed.
31  *       We don't clear the data bytes so the file is recoverable.
32  *
33  * @return On success, zero is returned.  On error, -1 is returned.
34  *****************************************************************************/
35 PUBLIC int do_unlink()
```

[5]严格来讲，留着原来的扇区可能造成安全隐患，因为数据还在那里，虽然主人的意图是完全消灭它们，不过目前我们先不考虑这些复杂问题。

```
{
        char pathname[MAX_PATH];

        /* get parameters from the message */
        int name_len = fs_msg.NAME_LEN; /* length of filename */
        int src = fs_msg.source;        /* caller proc nr. */
        assert(name_len < MAX_PATH);
        phys_copy((void*)va2la(TASK_FS, pathname),
                  (void*)va2la(src, fs_msg.PATHNAME),
                  name_len);
        pathname[name_len] = 0;

        if (strcmp(pathname , "/") == 0) {
                printl("FS:do_unlink():: cannot unlink the root\n");
                return -1;
        }

        int inode_nr = search_file(pathname);
        if (inode_nr == INVALID_INODE) {        /* file not found */
                printl("FS::do_unlink():: search_file() returns "
                       "invalid inode: %s\n", pathname);
                return -1;
        }

        char filename[MAX_PATH];
        struct inode * dir_inode;
        if (strip_path(filename, pathname, &dir_inode) != 0)
                return -1;

        struct inode * pin = get_inode(dir_inode->i_dev, inode_nr);

        if (pin->i_mode != I_REGULAR) { /* can only remove regular files */
                printl("cannot remove file %s, because "
                       "it is not a regular file.\n",
                       pathname);
                return -1;
        }

        if (pin->i_cnt > 1) {   /* the file was opened */
                printl("cannot remove file %s, because pin->i_cnt is %d.\n",
                       pathname, pin->i_cnt);
                return -1;
        }

        struct super_block * sb = get_super_block(pin->i_dev);

        /****************************/
        /* free the bit in i-map */
        /****************************/
        int byte_idx = inode_nr / 8;
        int bit_idx = inode_nr % 8;
        assert(byte_idx < SECTOR_SIZE); /* we have only one i-map sector */
        /* read sector 2 (skip bootsect and superblk): */
        RD_SECT(pin->i_dev, 2);
        assert(fsbuf[byte_idx % SECTOR_SIZE] & (1 << bit_idx));
        fsbuf[byte_idx % SECTOR_SIZE] &= ~(1 << bit_idx);
        WR_SECT(pin->i_dev, 2);

        /****************************/
        /* free the bits in s-map */
        /****************************/
        /*
         *                    bit_idx: bit idx in the entire i-map
         *              ... ____|____
         *                              \
         *                               \        .-- byte_cnt: how many bytes between
         *                                \       |             the first and last byte
         *        +-+-+-+-+-+-+-+-+-+-+-+-+ V +-+-+-+-+-+-+-+-+-+-+-+-+
```

```
 *      ... | | | | | |*|*|*|...|*|*|*|*| | | | |
 *          +-+-+-+-+-+-+-+-+   +-+-+-+-+-+-+-+-+
 *           0 1 2 3 4 5 6 7     0 1 2 3 4 5 6 7
 *                 ___/
 *      byte_idx: byte idx in the entire i-map
 */
int bit_idx  = pin->i_start_sect - sb->n_1st_sect + 1;
int byte_idx = bit_idx / 8;
int bits_left = pin->i_nr_sects;
int byte_cnt = (bits_left - (8 - (bit_idx % 8))) / 8;

/* current sector nr. */
int s = 2  /* 2: bootsect + superblk */
        + sb->nr_imap_sects + byte_idx / SECTOR_SIZE;

RD_SECT(pin->i_dev, s);

int i;
/* clear the first byte */
for (i = bit_idx % 8; (i < 8) && bits_left; i++,bits_left--) {
        assert((fsbuf[byte_idx % SECTOR_SIZE] >> i & 1) == 1);
        fsbuf[byte_idx % SECTOR_SIZE] &= ~(1 << i);
}

/* clear bytes from the second byte to the second to last */
int k;
i = (byte_idx % SECTOR_SIZE) + 1;               /* the second byte */
for (k = 0; k < byte_cnt; k++,i++,bits_left-=8) {
        if (i == SECTOR_SIZE) {
                i = 0;
                WR_SECT(pin->i_dev, s);
                RD_SECT(pin->i_dev, ++s);
        }
        assert(fsbuf[i] == 0xFF);
        fsbuf[i] = 0;
}

/* clear the last byte */
if (i == SECTOR_SIZE) {
        i = 0;
        WR_SECT(pin->i_dev, s);
        RD_SECT(pin->i_dev, ++s);
}
unsigned char mask = ~((unsigned char)(~0) << bits_left);
assert((fsbuf[i] & mask) == mask);
fsbuf[i] &= (~0) << bits_left;
WR_SECT(pin->i_dev, s);

/***************************/
/* clear the i-node itself */
/***************************/
pin->i_mode = 0;
pin->i_size = 0;
pin->i_start_sect = 0;
pin->i_nr_sects = 0;
sync_inode(pin);
/* release slot in inode_table[] */
put_inode(pin);

/*****************************************************/
/* set the inode-nr to 0 in the directory entry */
/*****************************************************/
int dir_blk0_nr = dir_inode->i_start_sect;
int nr_dir_blks = (dir_inode->i_size + SECTOR_SIZE) / SECTOR_SIZE;
int nr_dir_entries =
        dir_inode->i_size / DIR_ENTRY_SIZE; /* including unused slots
                                             * (the file has been
```

```
170                                                     * deleted but the slot
171                                                     * is still there)
172                                                     */
173         int m = 0;
174         struct dir_entry * pde = 0;
175         int flg = 0;
176         int dir_size = 0;
177
178         for (i = 0; i < nr_dir_blks; i++) {
179                 RD_SECT(dir_inode->i_dev, dir_blk0_nr + i);
180
181                 pde = (struct dir_entry *)fsbuf;
182                 int j;
183                 for (j = 0; j < SECTOR_SIZE / DIR_ENTRY_SIZE; j++,pde++) {
184                         if (++m > nr_dir_entries)
185                                 break;
186
187                         if (pde->inode_nr == inode_nr) {
188                                 /* pde->inode_nr = 0; */
189                                 memset(pde, 0, DIR_ENTRY_SIZE);
190                                 WR_SECT(dir_inode->i_dev, dir_blk0_nr + i);
191                                 flg = 1;
192                                 break;
193                         }
194
195                         if (pde->inode_nr != INVALID_INODE)
196                                 dir_size += DIR_ENTRY_SIZE;
197                 }
198
199                 if (m > nr_dir_entries || /* all entries have been iterated OR */
200                     flg) /* file is found */
201                         break;
202         }
203         assert(flg);
204         if (m == nr_dir_entries) { /* the file is the last one in the dir */
205                 dir_inode->i_size = dir_size;
206                 sync_inode(dir_inode);
207         }
208
209         return 0;
210 }
```

代码分四部分：

1. 释放 inode-map 中的相应位；
2. 释放 sector-map 中的相应位；
3. 将 inode_array 中的 i-node 清零；
4. 删除根目录中的目录项。

第74行我们对pin->i_cnt进行了判断，只有当i_cnt等于 1 时我们才能继续下去。也就是说，此时的 i-node 应该只被引用了一次。只要i_cnt大于 1，函数就直接返回代表不成功的 -1，因为此时一定还有别的文件描述符（file descriptor）引用了 i-node。这时文件还在使用中，我们当然不能就这样删除它。

由于代码第65行调用了get_inode()，所以一定不要忘记在使用完pin之后调用put_inode()来释放i_cnt（第160行），不然inode_table[]中的相应项就永不会被覆盖，这就造成内存泄漏了。

在删除根目录中的目录项时，我们面临一个选择：是让删除后的目录项空洞留在那里还是重新布置根目录。我们选择了前者，因为这样比较省事。这样做的坏处在于，一旦系统中有过文件删除操作，那么根目录文件的i_size就可能无法准确反映根目录中文件的多少。比如图9.16的情况，虽然根目录中连表示自身的“.”都算上也只有7个文件，但根目录文件的i_size却是(7+*N*)×DIR_ENTRY_SIZE，因为中间有*N*个之前存在的文件被删掉，此刻留下了大小为*N*×DIR_ENTRY_SIZE的空洞。

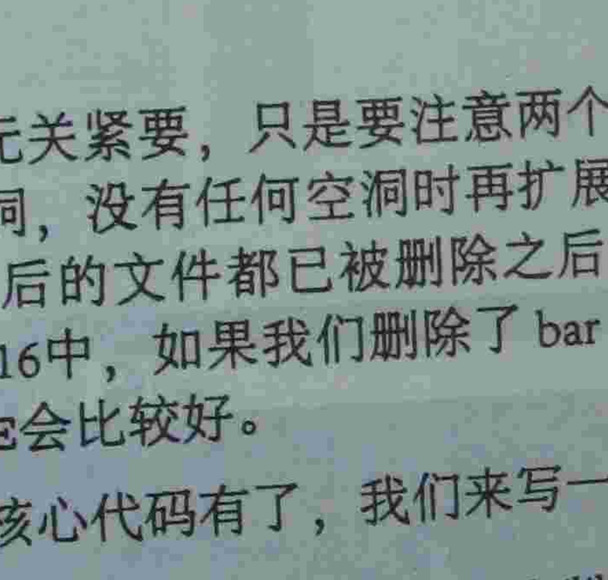


图9.16 根目录空洞

不过这个坏处其实无关紧要，只是要注意两个地方。一是当我们创建文件时，有空洞就先利用空洞，没有任何空洞时再扩展根目录文件的大小（参考代码9.24）；二是当空洞之后的文件都已被删除之后，我们最好改变根目录文件的i_size。比如在图9.16中，如果我们删除了bar，那么i_size的大小被更新成6×DIR_ENTRY_SIZE会比较好。

好了，删除文件的核心代码有了，我们来写一个对用户的接口函数：

代码9.37 unlink()（chapter9/h/lib/unlink.c）

```
/*****************************************************************************
 *                                unlink
 *****************************************************************************/
/**
 * Delete a file.
 *
 * @param pathname  The full path of the file to delete.
 *
 * @return Zero if successful, otherwise -1.
 *****************************************************************************/
PUBLIC int unlink(const char * pathname)
{
	MESSAGE msg;
	msg.type	= UNLINK;

	msg.PATHNAME	= (void*)pathname;
	msg.NAME_LEN	= strlen(pathname);

	send_recv(BOTH, TASK_FS, &msg);

	return msg.RETVAL;
}
```

然后就可以在用户进程中调用它了，见代码9.38。

代码9.38 删除文件（chapter9/h/kernel/main.c）

```
130 void TestA()
131 {
...
168         char * filenames[] = {"/foo", "/bar", "/baz"};
169
170         /* create files */
171         for (i = 0; i < sizeof(filenames) / sizeof(filenames[0]); i++) {
172                 fd = open(filenames[i], O_CREAT | O_RDWR);
173                 assert(fd != -1);
174                 printf("File␣created:␣%s␣(fd␣%d)\n", filenames[i], fd);
175                 close(fd);
176         }
177
178         char * rfilenames[] = {"/bar", "/foo", "/baz", "/dev_tty0"};
179
180         /* remove files */
181         for (i = 0; i < sizeof(rfilenames) / sizeof(rfilenames[0]); i++) {
182                 if (unlink(rfilenames[i]) == 0)
183                         printf("File␣removed:␣%s\n", rfilenames[i]);
184                 else
185                         printf("Failed␣to␣remove␣file:␣%s\n", rfilenames[i]);
186         }
187
188         spin("TestA");
189 }
```

我们运行一下，请看图9.17。

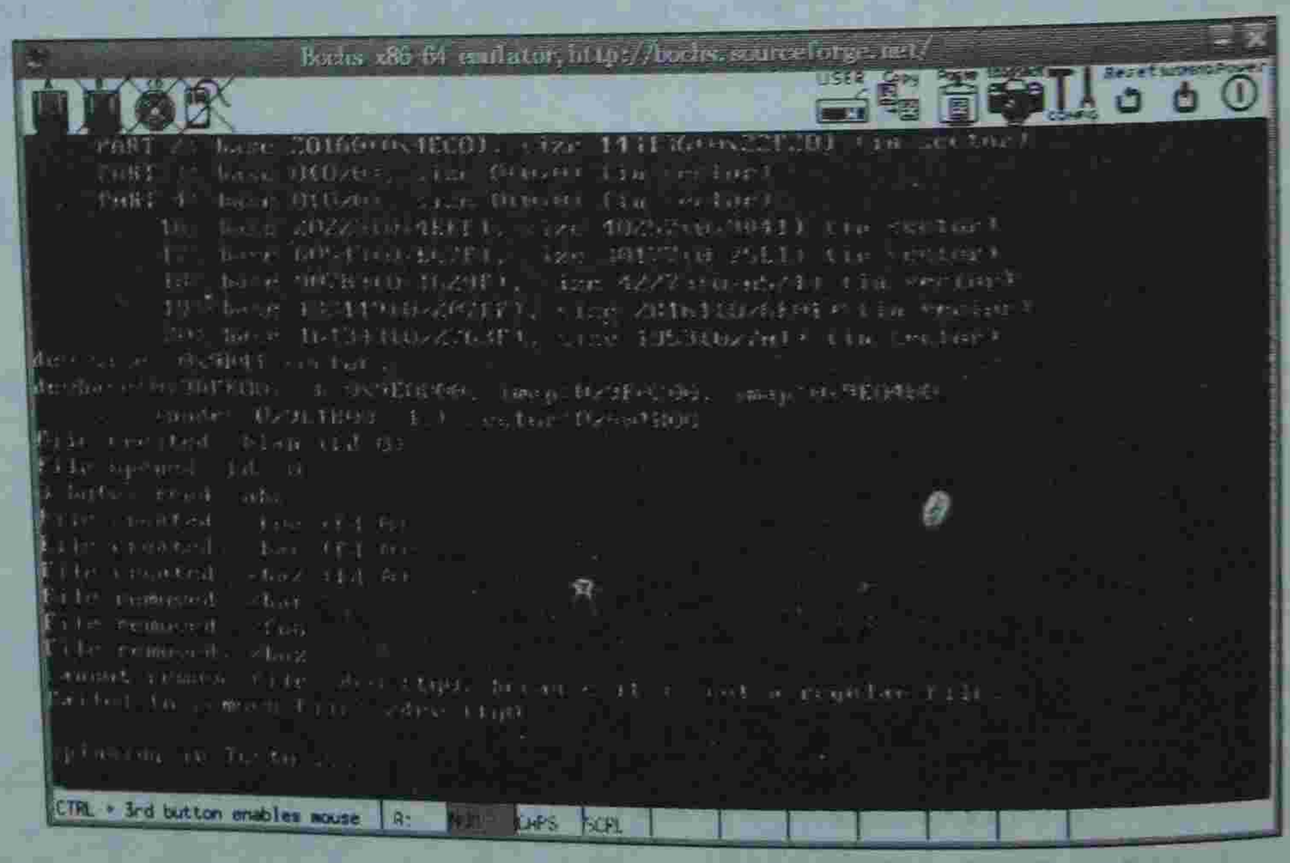


图 9.17 删除文件

我们创建了“foo”、“bar”和“baz”三个文件，然后又以不同的顺序删除它们。另外我们还试图删除特殊文件“/dev_tty0”，这也被及时地“制止”了。

如果读者用我们上一节介绍过的写log方法来画图的话，可以“动态地”观察到文件创建及删除的过程，比如图9.18中的左图便是刚刚创建了文件“/baz”的情况，右图是刚刚删除了文件“/bar”的情况。

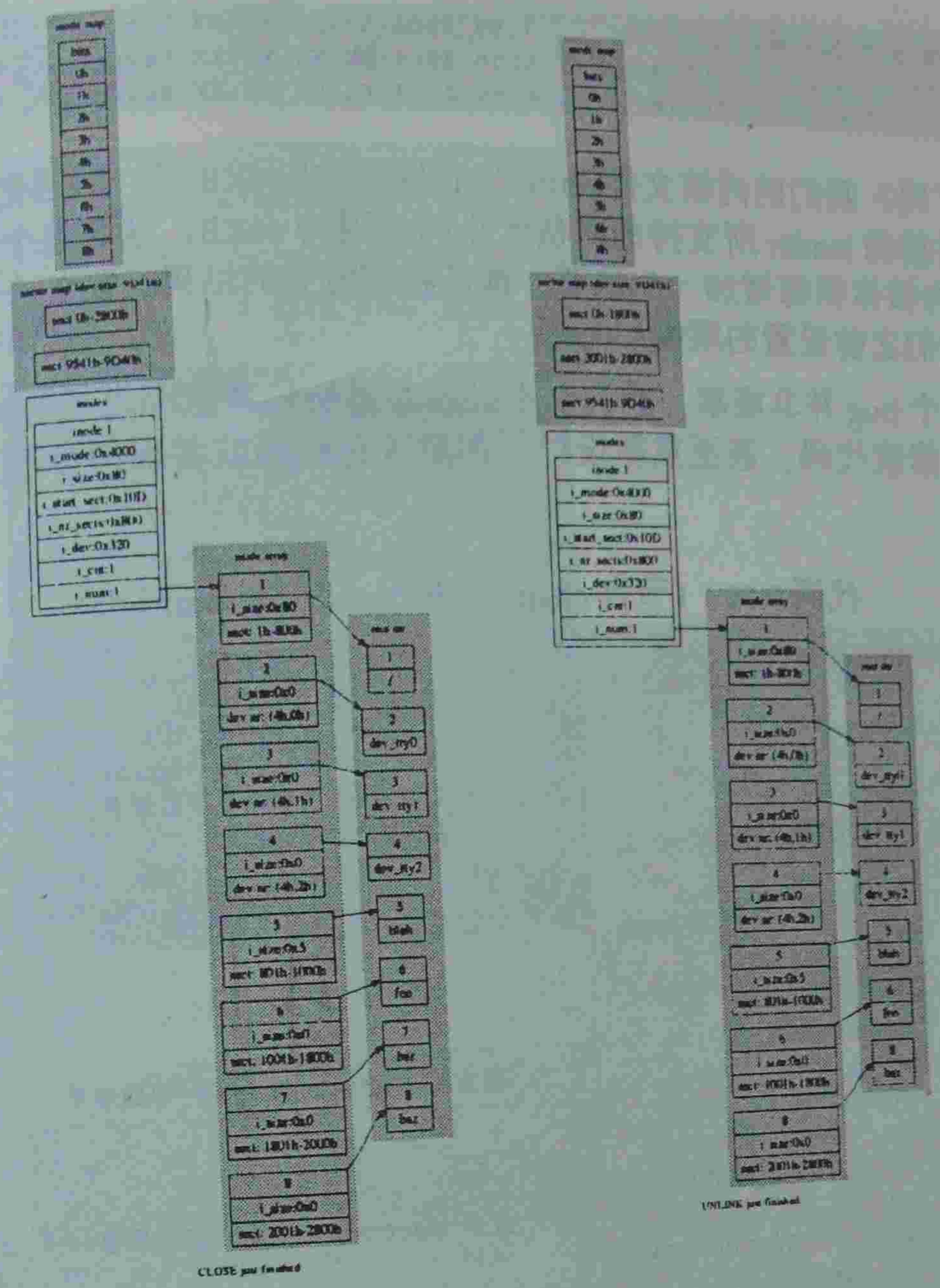


图 9.18 文件系统快照

9.19 插曲：奇怪的异常

就在我们完成unlink()系统调用的过程中，一个意外发生了，就在我们添加了几行很简单的代码之后，突然发现系统无法启动了：每次启动都很快崩溃。遇到这种情况千万不要惊慌，谁都会犯错误，再说我们有这么多调试手段，不愁找不到错误。

经过调试，错误终于被发现。其实如果此刻我使用一个简单的ls命令列出我们代码文件夹中的文件，你大概也猜到错误的原因了：

```
▷ ls -lF
total 1328
-rw-r--r-- 1 forrest forrest 83607552 2008-XX-XX XX:XX 80m.img
-rw-r--r-- 1 forrest forrest  1474560 2008-XX-XX XX:XX a.img
-rw-r--r-- 1 forrest forrest     1017 2008-XX-XX XX:XX bochsrc
drwxr-xr-x 4 forrest forrest     4096 2008-XX-XX XX:XX boot/
drwxr-xr-x 3 forrest forrest     4096 2008-XX-XX XX:XX fs/
drwxr-xr-x 4 forrest forrest     4096 2008-XX-XX XX:XX include/
drwxr-xr-x 3 forrest forrest     4096 2008-XX-XX XX:XX kernel/
-rwxr-xr-x 1 forrest forrest    66763 2008-XX-XX XX:XX kernel.bin*
-rw-r--r-- 1 forrest forrest    29556 2008-XX-XX XX:XX krnl.map
```

```
drwxr-xr-x 3 forrest forrest     4096 2008-XX-XX XX:XX lib/
-rw-r--r-- 1 forrest forrest     4100 2008-XX-XX XX:XX Makefile
drwxr-xr-x 3 forrest forrest     4096 2008-XX-XX XX:XX scripts/
```

看到了吗？我们的内核文件kernel.bin已经超过64KB了！而回忆第5章，我们知道，目前的loader所支持的内核大小的上限为64KB。这真是个让人开心的bug，因为内核体积每增加一个字节，都是我们努力的结果，而如今，它居然已经超出了我们之前设置的限制！

修复这个bug并非难事，我们也不必因为当初只考虑小内核的决定而气恼，因为每一次修改代码，甚至重新设计，都意味着代码或者我们自己又迈上一个新的台阶。

代码9.39 （节自chapter9/h/boot/loader.asm）

```
150 LABEL_GOON_LOADING_FILE:                    ; .
151         push    ax                          ; |
152         push    bx                          ; |  每读一个扇区就在 "Loading  " 后面
153         mov     ah, 0Eh                     ; |  打一个点，形成这样的效果：
154         mov     al, '.'                     ; |  Loading .....
155         mov     bl, 0Fh                     ; |
156         int     10h                         ; |
157         pop     bx                          ; |
158         pop     ax                          ; /
159
160         mov     cl, 1
161         call    ReadSector
162         pop     ax                          ; 取出此 Sector 在 FAT 中的序号
163         call    GetFATEntry
164         cmp     ax, 0FFFh
165         jz      LABEL_FILE_LOADED
166         push    ax                          ; 保存 Sector 在 FAT 中的序号
167         mov     dx, RootDirSectors
168         add     ax, dx
169         add     ax, DeltaSectorNo
170         add     bx, [BPB_BytsPerSec]
⇒ 171       jc      .1                          ; 如果 bx 重新变成 0，说明内核大于 64KB
172         jmp     .2
173 .1:
174         push    ax                          ; es += 0x1000  ← es 指向下一个段
175         mov     ax, es
176         add     ax, 1000h
177         mov     es, ax
178         pop     ax
179 .2:
180         jmp     LABEL_GOON_LOADING_FILE
181 LABEL_FILE_LOADED:
```

代码9.39是修复这个bug的关键部分，不外乎是增加一个判断而已。此外，我们还修改了load.inc中几个宏定义，使得运行时内存的使用情况如图9.19所示。

现在我们使用70000h~8FFFFh计128KB的空间来存放内核文件kernel.bin，使用1000h~6FFFFh共计约450KB的空间来存放重新布置后可执行的内核。7C00h~7DFFh本来是引导扇区在内存中的位置，由于引导结束之后它就不再使用，我们用内核将它覆盖掉了。

有了这次的经验，读者可能会想，一旦内核又太大盛不下怎么办呢？或者一旦内核重新布置之后超过了70000h的界限会不会又出现麻烦呢？避免再次出错的方法有很多，最容易的，便是写一个脚本，并修改Makefile，让脚本在每

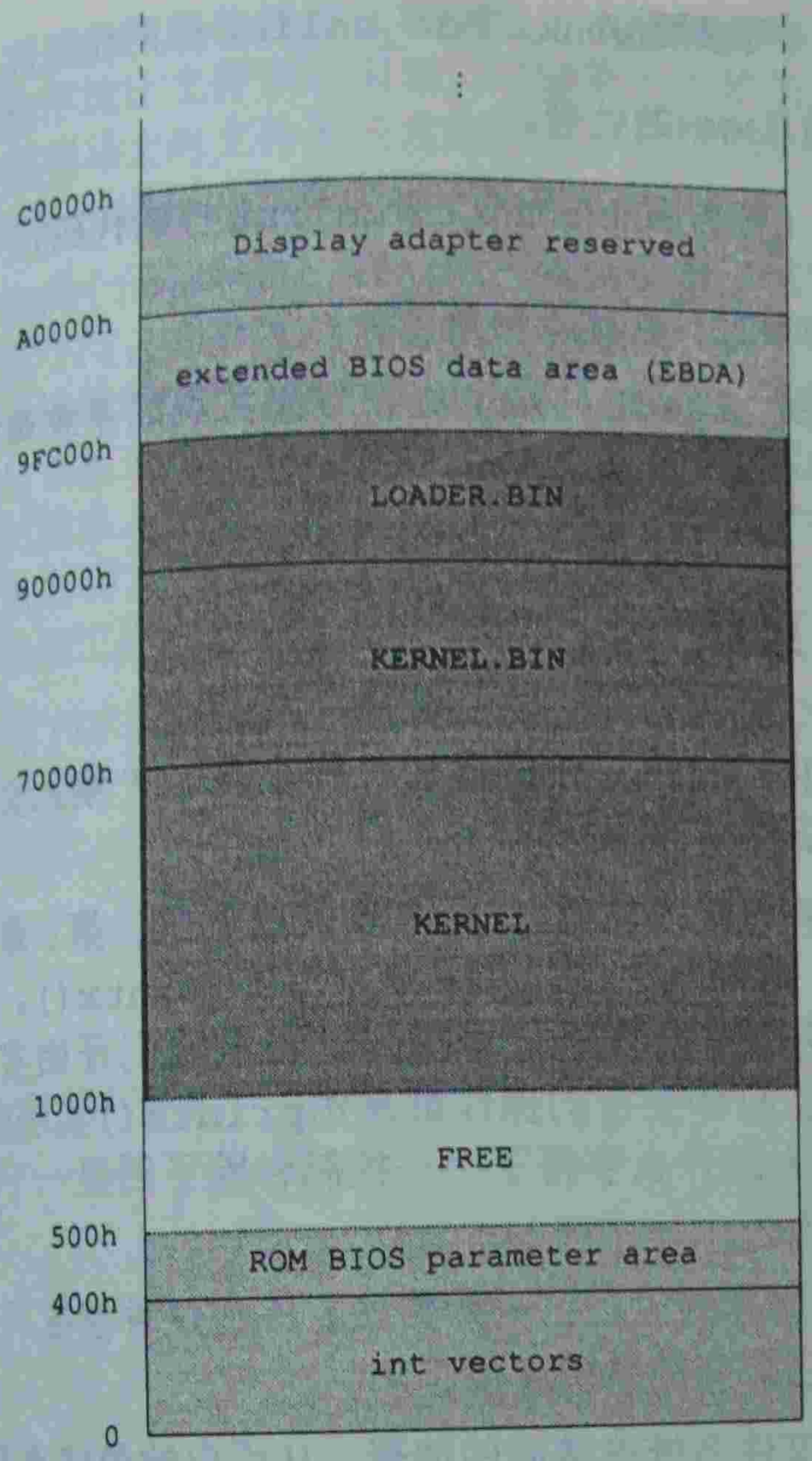


图 9.19 如今的内存的使用情况

次 make 之后执行一下。脚本的任务有二：一是比较kernel.bin的大小和我们预留出来的空间——此处是20000h；二是用 readelf 之类的工具获取内核在内存中的最终布局，并判断它是否越过了1000h~6FFFFh这个界限。我们当然也可以修改 LOADER 来完成这项工作，但比起写汇编代码，完成一个脚本显然更容易一些，而且绝对不会为操作系统引入新的 bug。有兴趣的读者可以自己试着做一做。

9.20 为文件系统添加系统调用的步骤

如今我们已有了open()、close()、read()、write()以及unlink()等系统调用，这当然还不够，不过更多的系统调用也不会有太多的新意了，读者可以自行进行添加。为文件系统添加一个系统调用xxx()的步骤大致上有这些：

1. 定义一种消息，比如MMM（可参照incluse/sys/const.h中UNLINK的定义方法）。

2. 写一个函数来处理MMM消息（可参照fs/link.c中do_unlink()的代码）。

3. 修改task_fs()，增加对消息MMM的处理。

4. 写一个用户接口函数xxx()（可参照lib/unlink.c中unlink()的代码）。

另外诸如增加函数声明的琐碎事宜，在此不再赘述。

如果读者现在就跃跃欲试的话，lseek()是个比较容易实现的系统调用，参考这些步骤，很快就能写就。您不妨试一下。

9.21 将TTY纳入文件系统

你早就猜到有这么一天，我们会像Linux中的做法一样，将TTY纳入文件系统的管辖范围，完成统一大业。现在是时候了。

让我们来简单地看一下TTY的现状。在第367页中我们提到过，第7章中的write()系统调用已经被弃用，取而代之的是第8章中写的printx()。而且write()也有了新的用途，它用来对文件进行写操作。自从我们开始实现IPC和文件系统，几乎所有在屏幕上打印字符的操作都是用printx()实现的。同时我们也很久没有在控制台上输入过什么字符了——控制台除了回显一下之外再无作为，我们干嘛要敲键盘呢？

所以如今的TTY如同一座几乎废弃的工厂，一片萧条。不过这种情况马上要改变了，我们马上要做的就是将它跟文件系统连接起来。

我们不妨先来想像一下TTY和文件系统连接后的情形。从用户的角度来看，读写TTY和读写普通文件将不会有差别，都是通过调用read()和write()来实现。普通文件和特殊文件的分辨是由文件系统来完成的，而且我们在实现文件系统时已经编写了一点代码了。在代码9.23第110行和代码9.32第57行我们都判断了文件的属性，并将字符设备特殊文件交给其驱动程序——TTY——来处理。

写入TTY跟写入普通文件是很类似的，不同之处只是在于TTY不需要进行任何端口操作，只是写入显存就可以了。而对于读出操作，TTY和普通文件则有着很大不同。TTY收到进程读请求的消息之后并不能马上返回数据，因为此时还没有任何输入呢，这时候需要用户输入字符，等输入结束之后，TTY才可以返回给进程。这个过程使我们面临两个问题。一是怎样才算是“输入结束”，是每一次键盘敲击之后都算结束呢，还是等敲回车才算结束，或者其他；二是是否要让文件系统等待输入过程结束。

对于第一个问题，前人已经给了我们答案。面向字符和面向行的需求是分别存在的，所以通常对TTY的操作有两种模式，分别叫生模式（Raw Mode）和熟模式（Cooked Mode），有时也被称作非规范模式（Uncanonical Mode）和规范模式（Canonical Mode）。在生模式下，字符被原原本本地向上传递，而在熟模式下，TTY负责一定的解释工作，比如将Backspace解释为删除前一字符、将回车解释为输入完毕等。

完全实现两种模式是一件烦琐的事情，按照一贯的偷懒原则，我们这里只实现熟模式，也就是说，每次等回车出现，才算完成一次输入。

第二个问题的答案比较明显。由于键盘输入可能耗时很久，这段时间内我们当然不能让文件系统闲着，可能有一大堆的进程想要读写磁盘文件呢，所以TTY需要马上向文件系统发送消息以示返回。而文件系统则完全应该阻塞想要得到输入的进程，一直到输入结束。

这两个问题解决，我们就大致上了解了读写 TTY 的过程了。假设进程 P 要求读取 TTY，它会发送消息给文件系统，文件系统将消息传递给 TTY，TTY 记下发出请求的进程号等信息之后立即返回，而文件系统这时并不对 P 接触阻塞，因为结果还未准备好。在接下来的过程中，文件系统像往常一样等待来自任何进程的请求。而 TTY 则会将键盘输入复制进 P 传入的内存地址，一旦遇到回车，TTY 就告诉文件系统，P 的请求已被满足，文件系统会解除对 P 的阻塞，于是整个读取工作结束。

写TTY的过程则很简单。P发消息给文件系统，文件系统传递给TTY，TTY收到消息后立即将字符写入显存（保持P和FS进程的阻塞），完成后发消息给文件系统，文件系统再发消息给P，整个过程结束。

好了，现在我们就来修改 TTY 任务，见代码9.40。

代码 9.40 TTY（chapter9/i/kernel/tty.c）

```
/*****************************************************************************
 *                                task_tty
 *****************************************************************************/
/**
 * <Ring 1> Main loop of task TTY.
 *****************************************************************************/
PUBLIC void task_tty()
{
	TTY *	tty;
	MESSAGE msg;

	init_keyboard();

	for (tty = TTY_FIRST; tty < TTY_END; tty++)
		init_tty(tty);

	select_console(0);

	while (1) {
		for (tty = TTY_FIRST; tty < TTY_END; tty++) {
			do {
				tty_dev_read(tty);
				tty_dev_write(tty);
			} while (tty->ibuf_cnt);
		}

		send_recv(RECEIVE, ANY, &msg);

		int src = msg.source;
		assert(src != TASK_TTY);

		TTY* ptty = &tty_table[msg.DEVICE];

		switch (msg.type) {
		case DEV_OPEN:
			reset_msg(&msg);
			msg.type = SYSCALL_RET;
			send_recv(SEND, src, &msg);
			break;
		case DEV_READ:
			tty_do_read(ptty, &msg);
```

```
                break;
        case DEV_WRITE:
                tty_do_write(ptty, &msg);
                break;
        case HARD_INT:
                /**
                 * waked up by clock_handler -- a key was just pressed
                 * @see clock_handler() inform_int()
                 */
                key_pressed = 0;
                continue;
        default:
                dump_msg("TTY::unknown msg", &msg);
                break;
        }
    }
}

/*****************************************************************************
 *                                tty_dev_read
 *****************************************************************************/
/**
 * Get chars from the keyboard buffer if the TTY::console is the 'current'
 * console.
 *
 * @see keyboard_read()
 *
 * @param tty  Ptr to TTY.
 *****************************************************************************/
PRIVATE void tty_dev_read(TTY* tty)
{
        if (is_current_console(tty->console))
                keyboard_read(tty);
}


/*****************************************************************************
 *                                tty_dev_write
 *****************************************************************************/
/**
 * Echo the char just pressed and transfer it to the waiting process.
 *
 * @param tty   Ptr to a TTY struct.
 *****************************************************************************/
PRIVATE void tty_dev_write(TTY* tty)
{
        while (tty->ibuf_cnt) {
                char ch = *(tty->ibuf_tail);
                tty->ibuf_tail++;
                if (tty->ibuf_tail == tty->ibuf + TTY_IN_BYTES)
                        tty->ibuf_tail = tty->ibuf;
                tty->ibuf_cnt--;

                if (tty->tty_left_cnt) {
                        if (ch >= ' ' && ch <= '~') { /* printable */
                                out_char(tty->console, ch);
                                void * p = tty->tty_req_buf +
                                           tty->tty_trans_cnt;
                                phys_copy(p, (void *)va2la(TASK_TTY, &ch), 1);
                                tty->tty_trans_cnt++;
                                tty->tty_left_cnt--;
                        }
                        else if (ch == '\b' && tty->tty_trans_cnt) {
                                out_char(tty->console, ch);
                                tty->tty_trans_cnt--;
                                tty->tty_left_cnt++;
                        }
```

```
                if (ch == '\n' || tty->tty_left_cnt == 0) {
                        out_char(tty->console, '\n');
                        MESSAGE msg;
                        msg.type = RESUME_PROC;
                        msg.PROC_NR = tty->tty_procnr;
                        msg.CNT = tty->tty_trans_cnt;
                        send_recv(SEND, tty->tty_caller, &msg);
                        tty->tty_left_cnt = 0;
                }
        }
    }
}


/*****************************************************************************
                                tty_do_read
 *****************************************************************************/
/**
 * Invoked when task TTY receives DEV_READ message.
 *
 * @note The routine will return immediately after setting some members of
 * TTY struct, telling FS to suspend the proc who wants to read. The real
 * transfer (tty buffer -> proc buffer) is not done here.
 *
 * @param tty  From which TTY the caller proc wants to read.
 * @param msg  The MESSAGE just received.
 *****************************************************************************/
PRIVATE void tty_do_read(TTY* tty, MESSAGE* msg)
{
        /* tell the tty: */
        tty->tty_caller   = msg->source;  /* who called, usually FS */
        tty->tty_procnr   = msg->PROC_NR; /* who wants the chars */
        tty->tty_req_buf  = va2la(tty->tty_procnr,
                                  msg->BUF);/* where the chars should be put */
        tty->tty_left_cnt = msg->CNT; /* how many chars are requested */
        tty->tty_trans_cnt= 0; /* how many chars have been transferred */

        msg->type = SUSPEND_PROC;
        msg->CNT = tty->tty_left_cnt;
        send_recv(SEND, tty->tty_caller, msg);
}


/*****************************************************************************
                                tty_do_write
 *****************************************************************************/
/**
 * Invoked when task TTY receives DEV_WRITE message.
 *
 * @param tty  To which TTY the calller proc is bound.
 * @param msg  The MESSAGE.
 *****************************************************************************/
PRIVATE void tty_do_write(TTY* tty, MESSAGE* msg)
{
        char buf[TTY_OUT_BUF_LEN];
        char * p = (char*)va2la(msg->PROC_NR, msg->BUF);
        int i = msg->CNT;
        int j;

        while (i) {
                int bytes = min(TTY_OUT_BUF_LEN, i);
                phys_copy(va2la(TASK_TTY, buf), (void*)p, bytes);
                for (j = 0; j < bytes; j++)
                        out_char(tty->console, buf[j]);
                i -= bytes;
                p += bytes;
```

```
342             }
343
344             msg->type = SYSCALL_RET;
345             send_recv(SEND, msg->source, msg);
346     }
```

跟第7章中的TTY任务相比，代码9.40复杂了许多，原因在于现在的输入和输出都是面向进程的，而原先的TTY自顾自地接受输入并马上输出。

作为驱动程序，TTY接收并处理DEV_OPEN、DEV_READ和DEV_WRITE消息，这跟硬盘驱动是类似的。

DEV_OPEN基本上是个摆设，收到此消息我们直接返回，因为实在没有什么可OPEN的。DEV_READ和DEV_WRITE分别由对应的函数tty_do_read()和tty_do_write()来处理[6]。从tty_do_read()的内容可以看出，结构体TTY的成员增加了若干，因为我们要保存发送读请求的进程的一些信息。新的TTY结构如代码9.41所示。

代码9.41 TTY结构（chapter9/i/include/sys/tty.h）

```
19  typedef struct s_tty
20  {
21          u32     ibuf[TTY_IN_BYTES];     /* TTY input buffer */
22          u32*    ibuf_head;              /* the next free slot */
23          u32*    ibuf_tail;              /* the val to be processed by TTY */
24          int     ibuf_cnt;               /* how many */
25
26          int     tty_caller;
27          int     tty_procnr;
28          void*   tty_req_buf;
29          int     tty_left_cnt;
30          int     tty_trans_cnt;
31
32          struct s_console *      console;
33  }TTY;
```

tty_caller用来保存向TTY发送消息的进程（通常这个进程应该是FS）的进程号。tty_procnr用来保存请求数据的进程（下文中我们称它为进程P）的进程号。tty_req_buf保存进程P用来存放读入字符的缓冲区的线性地址。tty_left_cnt保存P想读入多少字符。tty_trans_cnt保存TTY已经向P传送了多少字符。

在tty_do_read()将结构体的这些成员赋值之后，马上向文件系统发送了一个SUSPEND_PROC消息，文件系统需要处理它：

代码9.42 文件系统处理SUSPEND_PROC消息（chapter9/i/fs/main.c）

```
29  /*****************************************************************************
30   *
31   *                                  task_fs
32  /**
33   * <Ring 1> The main loop of TASK FS.
34   *
35   *****************************************************************************/
36  PUBLIC void task_fs()
37  {
    ...
49                  switch (msgtype) {
```

[6]也就是说，这两个函数跟第7章中的同名函数已有不同。

```
        case OPEN:
                fs_msg.FD = do_open();
                break;
        case CLOSE:
                fs_msg.RETVAL = do_close();
                break;
        case READ:
        case WRITE:
                fs_msg.CNT = do_rdwt();
                break;
        case UNLINK:
                fs_msg.RETVAL = do_unlink();
                break;
        case RESUME_PROC:
                src = fs_msg.PROC_NR;
                break;

...             /* reply */
                if (fs_msg.type != SUSPEND_PROC) {
                        fs_msg.type = SYSCALL_RET;
                        send_recv(SEND, src, &fs_msg);
                }
        }
}
```

文件系统在收到SUSPEND_PROC消息之后，并不像处理完READ或WRITE消息之后那样向进程 P 发送消息，而是不理不睬，径自开始下一个消息处理的循环，留下 P 独自等待。一直到 TTY 发送了RESUME_PROC消息，文件系统才会通知 P，让它继续运行。这一过程我们已经解释过。

我们回头接着来看task_tty()。TTY 处理完DEV_READ消息之后，来到第84行继续下一个循环，这里tty_dev_read()将从键盘缓冲区将字符读入，在接下来的tty_dev_write()中，这些字符将被送入进程 P 的缓冲区，直到读入一个回车（\n）或者已经传输了足够多的字符（由tty_left_cnt指定）。

注意，传输字符的过程并非在一个循环中就可以完成，因为人的手很慢而机器处理的速度很快。所以很可能发生的事情是，人通过键盘输入了一个字符“a”，被tty_dev_read()读出并由tty_dev_write()传给 P，由于tty->ibuf_cnt这时变成 0，所以tty_dev_write()返回，程序就来到了task_fs()的第91行继续接收消息了。而所有这些完成之后，人的手才慢吞吞地（以机器的视角来看，这应该是过了“很久”）输入了下一个字符“b”。

那么问题随之出现了，如果这时没有人给 TTY 发消息，TTY 就会在第91行永远等下去，即便我们敲了再多字符，TTY 也不加理会了，这可不行。所以我们还是用 TASK_HD 中使用的方法，借助从不停歇的时钟中断来唤醒 TTY：

代码 9.43 时钟中断唤醒 TTY（chapter9/i/kernel/clock.c）

```
PUBLIC void clock_handler(int irq)
{
...
        if (key_pressed)
                inform_int(TASK_TTY);
...
}
```

每次时钟中断发生时，系统都判断key_pressed这一变量，它是在键盘中断处理程序中指定的：

代码9.44 键盘中断（chapter9/i/kernel/keyboard.c）

```
54  PUBLIC void keyboard_handler(int irq)
55  {
    ...
66          key_pressed = 1;
67  }
```

也就是说，每次键盘敲击都会通过key_pressed这一变量反映出来，随后的时钟中断根据它来唤醒TTY，这样TTY就又来到代码9.40第84行进行下一次循环。

跟读取TTY的过程想比，写入的过程相对简单，一个tty_do_write()就解决了。

到这里TTY就改写完毕了，下面就来改造一个用户进程，让它使用新的机制来读写控制台，见代码9.45。

代码9.45 TestB（chapter9/i/kernel/main.c）

```
194 void TestB()
195 {
196         char tty_name[] = "/dev_tty1";
197
198         int fd_stdin  = open(tty_name, O_RDWR);
199         assert(fd_stdin  == 0);
200         int fd_stdout = open(tty_name, O_RDWR);
201         assert(fd_stdout == 1);
202
203         char rdbuf[128];
204
205         while (1) {
206                 write(fd_stdout, "$ ", 2);
207                 int r = read(fd_stdin, rdbuf, 70);
208                 rdbuf[r] = 0;
209
210                 if (strcmp(rdbuf, "hello") == 0) {
211                         write(fd_stdout, "hello world!\n", 13);
212                 }
213                 else {
214                         if (rdbuf[0]) {
215                                 write(fd_stdout, "{", 1);
216                                 write(fd_stdout, rdbuf, r);
217                                 write(fd_stdout, "}\n", 2);
218                         }
219                 }
220         }
221
222         assert(0); /* never arrive here */
223 }
```

原先进程跟TTY之间的对应关系是由进程表中的nr_tty成员来实现的，如今我们只需要打开文件就可以了，所以nr_tty再无用处，我们可以删去。

好了，现在可以编译运行了，见图9.20。

怎么样，还挺像模像样的是吧？几乎可以被人误认为是个shell了。不过这个假shell太傻了，除了能应对一个“hello”之外，它干脆就是一个回音壁，输入什么随即输出什么。读者肯定在想，如果我输入个“ls”，能得到所有文件的列表，岂不是酷毙了？没错，的确会酷得不行了。那么怎么才能做到呢？我们当然可以改造一下TestB，像处理“hello”一样处理“ls”，但这不能从根本上让

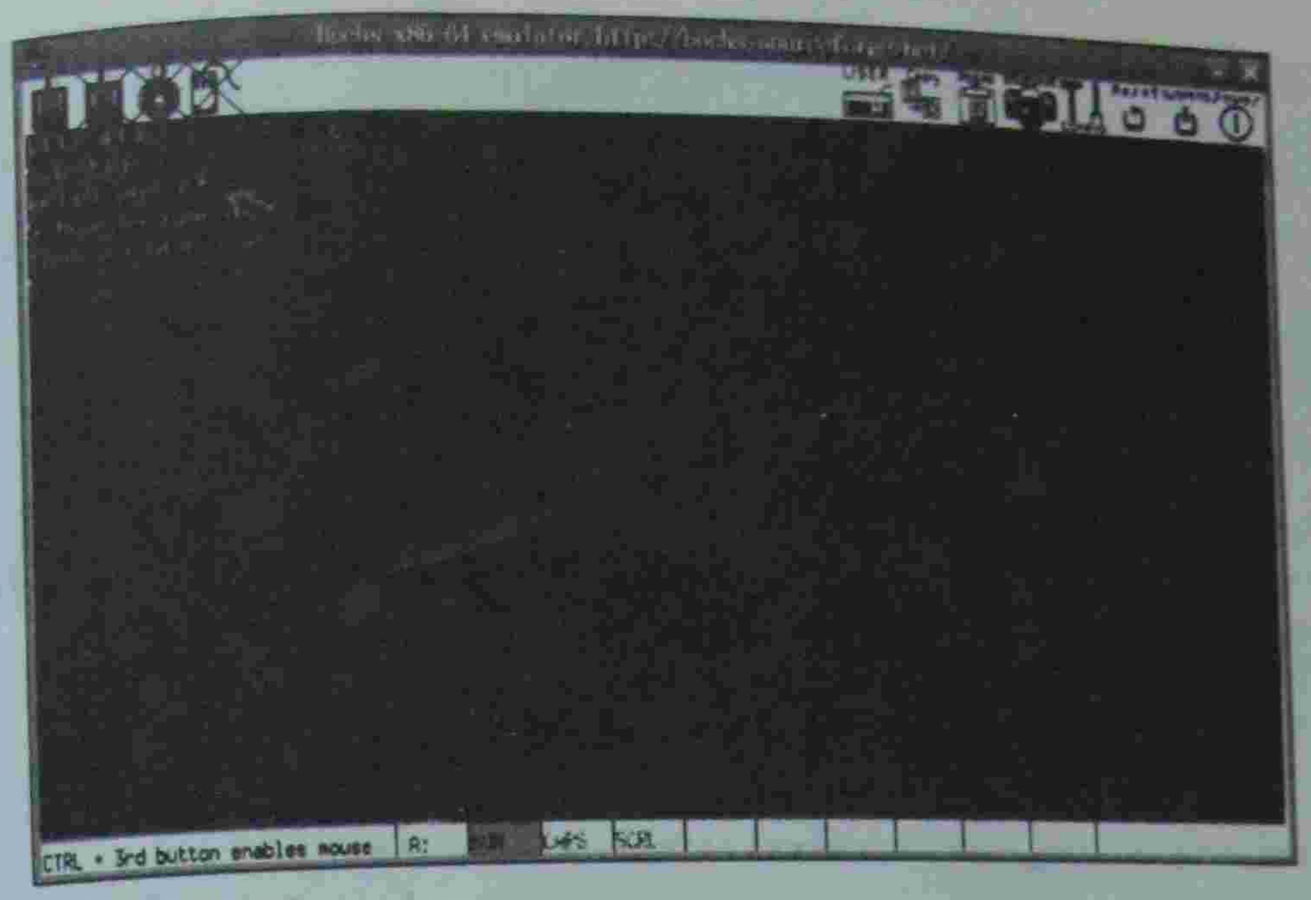


图 9.20 文件系统和 TTY 连接起来

假 shell 变聪明。我们不如现在就来写一个真的 shell，哪怕是个极其简单的 shell，让它能够从磁盘上读取可执行文件并执行它，那才叫真的酷。

其实做一个真的 shell 并不难，很快，在下一章中，我们就会讲到。

9.22 改造printf

代码9.45已经可以工作了，但里面对write()的调用多少有点丑陋，输出一个“{”居然需要三个参数。不用我说，你也猜到下面是时候改造printf了，其实这个工作很简单：

代码 9.46 printf（chapter9/j/lib/printf.c）

```
/*****************************************************************************
 *                                printf
 *****************************************************************************/
/**
 * The most famous one.
 *
 * @param fmt  The format string
 *
 * @return  The number of chars printed.
 *****************************************************************************/
PUBLIC int printf(const char *fmt, ...)
{
        int i;
        char buf[STR_DEFAULT_LEN];

        va_list arg = (va_list)((char*)(&fmt) + 4); /**
                                                     * 4: size of 'fmt' in
                                                     *    the stack
                                                     */


        i = vsprintf(buf, fmt, arg);
        int c = write(1, buf, i);

        assert(c == i);

        return i;
```

```
85  }
86
87  /*****************************************************************************
88   *                                printl
89   *****************************************************************************/
90  /**
91   * low level print
92   *
93   * @param fmt  The format string
94   *
95   * @return  The number of chars printed.
96   *****************************************************************************/
97  PUBLIC int printl(const char *fmt, ...)
98  {
99          int i;
100         char buf[STR_DEFAULT_LEN];
101
102         va_list arg = (va_list)((char*)(&fmt) + 4); /**
103                                                      * 4: size of 'fmt' in
104                                                      *    the stack
105                                                      */
106
107         i = vsprintf(buf, fmt, arg);
108         printx(buf);
109
110         return i;
111 }
```

我们把原来的printf()改名作printl()，然后将printf()用write()系统调用重写了一遍。没有把直接使用系统调用的printl()删掉，这意味着之前的printl()调用没有变化。今后在内核以及任务中我们仍然会使用printl()，只有在用户进程中，我们才使用依赖于文件系统的printf()。现在我们就把 TestB 修改一下，见代码9.47。

代码 9.47 TestB（chapter9/j/kernel/main.c）

```
194 void TestB()
195 {
196         char tty_name[] = "/dev_tty1";
197
198         int fd_stdin  = open(tty_name, O_RDWR);
199         assert(fd_stdin  == 0);
200         int fd_stdout = open(tty_name, O_RDWR);
201         assert(fd_stdout == 1);
202
203         char rdbuf[128];
204
205         while (1) {
⇒ 206                 printf("$ ");
207                 int r = read(fd_stdin, rdbuf, 70);
208                 rdbuf[r] = 0;
209
210                 if (strcmp(rdbuf, "hello") == 0)
⇒ 211                         printf("hello world!\n");
212                 else
213                         if (rdbuf[0])
⇒ 214                                 printf("{%s}\n", rdbuf);
215         }
216
217         assert(0); /* never arrive here */
218 }
```

改造后的函数好看多了，只是一定要注意，printf()默认调用它的进程已经打开了控制台文件，并且fd为1，在使用时一定要保证这一点成立。

9.23 总结

这真是冗长的一章，希望读者没有失去耐心。实际上本章讲的不仅仅是文件系统，还有驱动程序、硬盘操作等[7]。现在再来看代码9.17，我想你一定觉得很有成就感了，我们的文件系统也可以做这些事情了。而且不仅如此，连控制台都也已经纳入了文件系统的控制之中。由此我们不仅熟悉了如何读写普通文件，也熟悉了特殊文件的操作方法。将来我们当然应该在这基础上扩充文件系统，但那已经是相对容易的事情。

现在，让我们先暂时放下文件系统，进入下一章——内存管理。

[7]在最后一节的代码中笔者甚至偷偷修改了console.c，使它能够无限滚屏，它跟文件系统没有很大干系，而且只是雕虫小技，故在此略过不提，读者可自行查看。

Life is long. There is always time for Plan B. But don't begin with it.

—— *Drew G. Faust*

10 内存管理

我们在学习保护模式的时候已经初窥过内存管理的门径。不管是分段还是分页，都是内存管理的基本概念。然而在本章中，我们并不把事情搞得太复杂。跟上章的扁平文件系统类似，我们将以系统调用为驱动，逐步实现一种很简单的内存管理模式。

10.1 fork

上一章的结尾处我们提到要做一个 shell，这是个好主意，但是我们得先搞清楚 shell 做事的原理。它的原理说来简单，其实就是用一个子进程来执行一个命令。子进程对我们而言是个陌生的概念，目前我们的系统中所有的进程都是编译时确定好的，每个进程的入口地址、堆栈等都是在global.c中指定的，而且除了权限有不同，所有进程是并列的，没有层次关系。根据第6.4.6节中我们对添加任务步骤的总结，加上图6.9，我们可以想像得到，一个新的进程需要的要素有：

- 自己的代码，数据和堆栈；
- 在`proc_table[]`中占用一个位置；
- 在 GDT 中占用一个位置，用以存放进程对应的 LDT 描述符。

后两项工作比较容易完成，那第一项呢，代码、数据和堆栈从哪里来呢？传统上，生成一个新的进程时，这些都是直接从某个已有的进程那里继承或者复制。这也正解释了子进程这一概念：如果新的进程 C 的代码、数据和堆栈是从已有的进程 P 而来，那么 P 被称为父进程（parent），C 被称为子进程（child）。

10.1.1 认识fork

生成一个子进程的系统调用被称为`fork()`，操作系统接到一个`fork`请求后，会将调用者复制一份[1]，这时就会有两个一模一样的进程同时运行。很好玩

[1]为了提高效率，有时并不真的复制，而是采取 copy-on-write 策略，在此不展开讨论。

不是吗？我们现在就来实现这一好玩的过程。

第一个问题是拿谁作为父进程来执行fork()。你可能知道，在Linux中，所有进程都有同一个祖先，那就是init进程，其实那是跟Minix学的，Minix中也有个INIT进程，作为用户进程的祖先。既然如此，我们也来模仿一下，写个Init进程，如代码10.1所示。

代码 10.1 Init（chapter10/a/kernel/main.c）

```
/*****************************************************************************
                                Init
 *****************************************************************************/
/**
 * The hen.
 *
 *****************************************************************************/
void Init()
{
        int fd_stdin  = open("/dev_tty0", O_RDWR);
        assert(fd_stdin  == 0);
        int fd_stdout = open("/dev_tty0", O_RDWR);
        assert(fd_stdout == 1);

        printf("Init() is running ...\n");

        int pid = fork();
        if (pid != 0) { /* parent process */
                printf("parent is running, child pid:%d\n", pid);
                spin("parent");
        }
        else {  /* child process */
                printf("child is running, pid:%d\n", getpid());
                spin("child");
        }
}
```

Init()中调用了我们即将实现的fork()，而且判断了返回值。如果你看过施瓦辛格主演的电影《第六日》，你一定可以马上理解这个返回值的意义：被克隆者自己并不知道自己是被克隆的，只有克隆他的人告诉他们，他们才明白真相。在这里，对于父进程P和子进程C而言，判断操作系统返回给进程的fork()函数返回值，可以让进程知道自己是本人，还是被克隆者。

其实还有一个方法，可以让进程得知自己是父进程还是子进程，那就是通过调用getpid()。每个进程有唯一的pid，如果调用fork()前后pid一致，说明进程为父进程，反之为子进程。有趣的是，在《第六日》中，一个人也有一个方法来获知自己被克隆了几次，那就是观察自己的眼睑内侧[2]。好了，亲爱的读者，下面我们不研究操作系统了，拿出镜子，观察一下你的眼睑内侧，那里记录了你的pid……

哈哈，不要害怕，我猜你不是个克隆人。那么让我们继续。fork()的返回值有两种：零或非零。如果返回零，表明自己是个子进程；如果返回非零，不仅表明自己是父进程，而且返回值即子进程的pid。

在这里我们不光要增加一个Init进程，还要增加一个MM进程——本章说的就是内存管理，这个进程无疑是最重要的。MM将负责从用户进程接收消息，完成fork()等操作。增加进程的方法读者已经很熟悉了，在此不再赘述。

[2]如果读者没看过这部电影，千万别骂我剧透，我不是故意的☺。

不过事情还不仅如此，增加了一个 Init，同时还意味着增加了 Init 的子子孙孙。我们来看图10.1（它跟图6.9有点类似），它分三部分，分别是 GDT、进程表和进程体。GDT 和进程表之间有两种联系，一是进程表的ldt_sel指向 GDT 中的某一项（②），而这一项反过来指向进程表内的两个 LDT 描述符（①）。两个 LDT 描述符所描述的内存范围，即为进程在内存中所占的空间，这用③来表示。

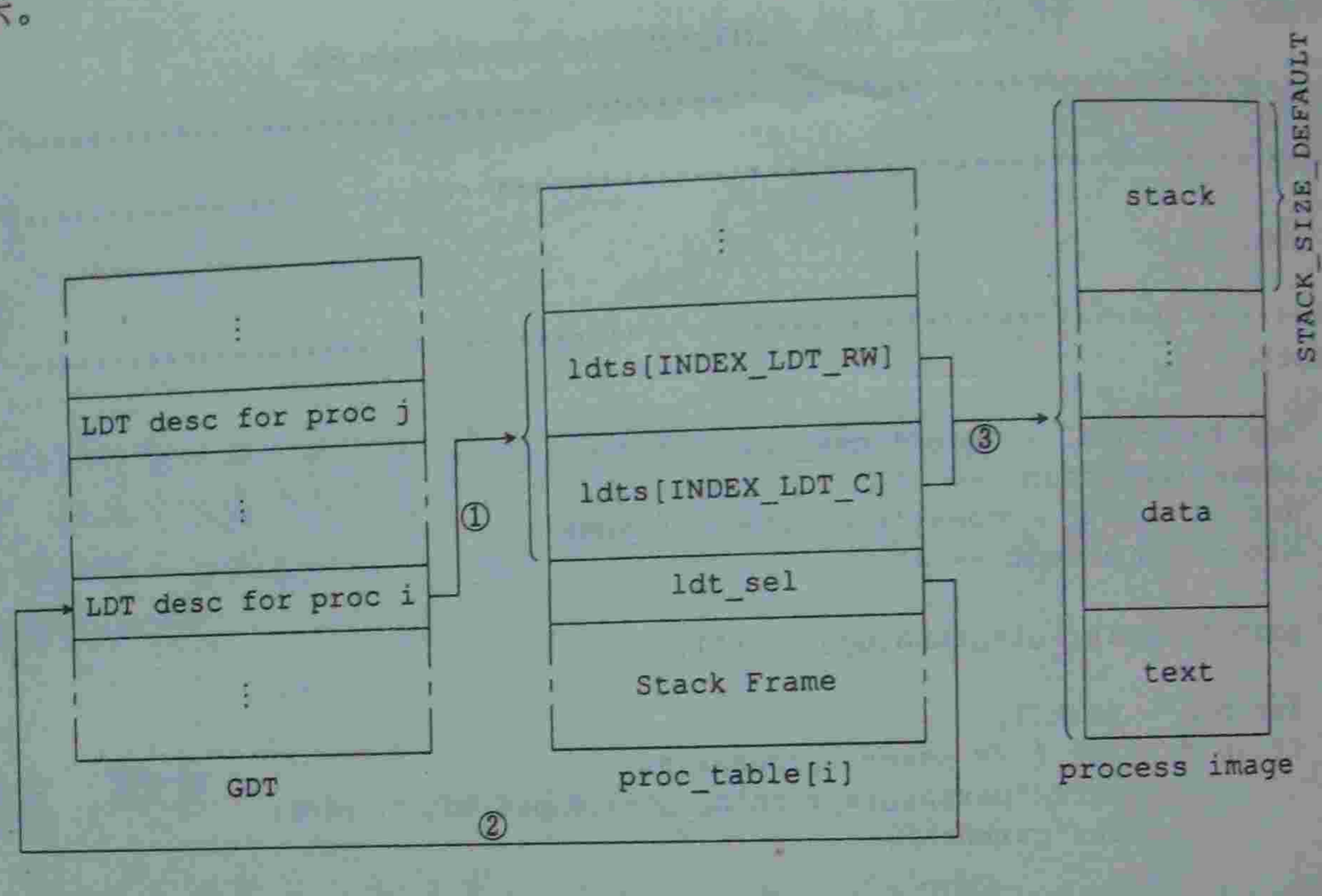


图 10.1 一个进程涉及的所有数据结构及相互联系

之前所有的进程都是我们“手工”添加的，进程的入口地址和堆栈长度等信息首先是记在global.c中，然后关系①在init_prot()中确定，关系②和关系③在kernel_main()中确定。将来我们肯定没办法把由 Init 进程fork出来的子进程的入口地址放在global.c中，也不可能事先分配好某个未知进程的内存，所以有些内容肯定是要动态决定的。不过也不是所有工作都需要fork时才做，有些工作仍然是可以提前做好的。总结一下的话，以下工作我们可以提前做好：

- 在proc_table[]中预留出一些空项，供新进程使用；
- 将proc_table[]中的每一个进程表项中的ldt_sel项都设定好（关系②）；
- 将进程所需的 GDT 表项都初始化好（关系①）。

而新进程需要的内存空间，以及关系③，则都需要 MM 来完成，这是过会儿我们要重点关注的代码。

了解了图10.1中各个元素之间的关系，我们就可以写代码了，首先是可以提前做好的工作，然后是需要 MM 来动态完成的工作。

10.1.2 fork前要做的工作（为fork所做的准备）

首先在proc_table[]中预留出一些空项，这需要通过改变NR_PROCS来实现：

代码 10.2 NR_PROCS（chapter10/a/include/sys/proc.h）

```
#define NR_TASKS        5
#define NR_PROCS        32
#define NR_NATIVE_PROCS 4
```

这里我们不仅改变了NR_PROCS，还增加了一个宏：NR_NATIVE_PROCS，它表示系统初启时共有多少个用户进程。因为我们有 TestA、TestB、TestC 和 Init，所以它应该是 4。在表示进程数目的宏发生改变后，我们还应该修改kernel_main()，见代码10.3。

代码 10.3 修改后的 kernel_main()（chapter10/a/kernel/main.c）

```
/*****************************************************************************
 *                                kernel_main
 *****************************************************************************/
/**
 * jmp from kernel.asm::_start.
 *
 *****************************************************************************/
PUBLIC int kernel_main()
{
        disp_str("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n"
                 "-----------------------------------\n");

        int i, j, eflags, prio;
        u8  rpl;
        u8  priv; /* privilege */

        struct task * t;
        struct proc * p = proc_table;

        char * stk = task_stack + STACK_SIZE_TOTAL;

        for (i = 0; i < NR_TASKS + NR_PROCS; i++,p++,t++) {
                if (i >= NR_TASKS + NR_NATIVE_PROCS) {
                        p->p_flags = FREE_SLOT;
                        continue;
                }

                if (i < NR_TASKS) {     /* TASK */
                        t       = task_table + i;
                        priv    = PRIVILEGE_TASK;
                        rpl     = RPL_TASK;
                        eflags  = 0x1202;/* IF=1, IOPL=1, bit 2 is always 1 */
                        prio    = 15;
                }
                else {                  /* USER PROC */
                        t       = user_proc_table + (i - NR_TASKS);
                        priv    = PRIVILEGE_USER;
                        rpl     = RPL_USER;
                        eflags  = 0x202;        /* IF=1, bit 2 is always 1 */
                        prio    = 5;
                }

                strcpy(p->name, t->name);       /* name of the process */
                p->p_parent = NO_TASK;

                if (strcmp(t->name, "INIT") != 0) {
```

```
67              p->ldts[INDEX_LDT_C]  = gdt[SELECTOR_KERNEL_CS >> 3];
68              p->ldts[INDEX_LDT_RW] = gdt[SELECTOR_KERNEL_DS >> 3];
69
70              /* change the DPLs */
71              p->ldts[INDEX_LDT_C].attr1  = DA_C   | priv << 5;
72              p->ldts[INDEX_LDT_RW].attr1 = DA_DRW | priv << 5;
73      }
74      else {          /* INIT process */
75              unsigned int k_base;
76              unsigned int k_limit;
77              int ret = get_kernel_map(&k_base, &k_limit);
78              assert(ret == 0);
79              init_desc(&p->ldts[INDEX_LDT_C],
80                        0, /* bytes before the entry point
81                            * are useless (wasted) for the
82                            * INIT process, doesn't matter
83                            */
84                        (k_base + k_limit) >> LIMIT_4K_SHIFT,
85                        DA_32 | DA_LIMIT_4K | DA_C | priv << 5);
86
87              init_desc(&p->ldts[INDEX_LDT_RW],
88                        0, /* bytes before the entry point
89                            * are useless (wasted) for the
90                            * INIT process, doesn't matter
91                            */
92                        (k_base + k_limit) >> LIMIT_4K_SHIFT,
93                        DA_32 | DA_LIMIT_4K | DA_DRW | priv << 5);
94      }
95
96      p->regs.cs = INDEX_LDT_C << 3 | SA_TIL | rpl;
97      p->regs.ds =
98              p->regs.es =
99              p->regs.fs =
100             p->regs.ss = INDEX_LDT_RW << 3 | SA_TIL | rpl;
101     p->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK) | rpl;
102     p->regs.eip     = (u32)t->initial_eip;
103     p->regs.esp     = (u32)stk;
104     p->regs.eflags  = eflags;
105
106     p->ticks = p->priority = prio;
107
108     p->p_flags = 0;
109     p->p_msg = 0;
110     p->p_recvfrom = NO_TASK;
111     p->p_sendto = NO_TASK;
112     p->has_int_msg = 0;
113     p->q_sending = 0;
114     p->next_sending = 0;
115
116     for (j = 0; j < NR_FILES; j++)
117             p->filp[j] = 0;
118
119             stk -= t->stacksize;
120     }
121
122     k_reenter = 0;
123     ticks = 0;
124
125     p_proc_ready    = proc_table;
126
127     init_clock();
128     init_keyboard();
129
130     restart();
131
132     while(1){}
133 }
```

跟之前相比，代码做了一些调整，主要是两点。一是将暂时没有用到的proc_table[]表项的p_flags成员赋值为FREE_SLOT，将来MM会根据此项值来判断一个表项是否为空；二是将Init进程单独区分开来，其LDT单独赋值。这样做的原因很简单，Init将来会fork出子进程，那么Init的所有内存范围都将被复制到新的位置，并在那里运行。如果Init用与其他进程相同的处理方法，使用0~4GB的扁平空间作为LDT描述符，那么我们在内存中将找不到另一块空间来存放它——在32位系统中，4GB是全部内存空间。

那么Init进程的内存应该被限制在怎样的范围之内呢？最简单的思路就是，内核有多大，Init进程的内存空间就有多大，这样不但大大缩小了进程占用的内存，而且可以保证对read()、write()等所有库函数的调用都是合法的。你可能会说，Init进程还可以更小，没错，因为内核中的大部分代码对Init是不会直接调用的，但更小的Init将会增加代码的复杂程度，所以我们还是偷个懒，用整个内核占用的内存来当做Init的内存范围。

下面的问题就是如何得到内核的内存范围，由于内核是由LOADER加载的，所以需要问LOADER。具体方法是在LOADER中将内核中需要的数据写到内存的某个位置，然后在内核中读出来。根据图9.19可知，500h~FFFh之间的区域都是空闲的，读者可随意使用。

loader.asm中增加的内容如代码10.4所示。

代码10.4 （节自chapter10/a/boot/loader.asm）

```
428         ;; fill in BootParam[]
429         mov     dword [BOOT_PARAM_ADDR], BOOT_PARAM_MAGIC ; Magic Number
430         mov     eax, [dwMemSize]
431         mov     [BOOT_PARAM_ADDR + 4], eax ; memory size
432         mov     eax, KERNEL_FILE_SEG
433         shl     eax, 4
434         add     eax, KERNEL_FILE_OFF
435         mov     [BOOT_PARAM_ADDR + 8], eax ; phy-addr of kernel.bin
```

我们在地址BOOT_PARAM_ADDR（在load.inc中定义）处写入三个数字。第一个数字是个魔数，用做简单的标识。第二个数字是内存的大小。第三个数字是kernel.bin在内存中的物理地址。这样在内核中，我们只需要简单地读取就可以了，见代码10.5。

代码10.5 获取LOADER预先保存的参数（chapter10/a/lib/klib.c）

```
27  /*****************************************************************************
28   *                                get_boot_params
29   *****************************************************************************/
30  /**
31   * <Ring 0~1> The boot parameters have been saved by LOADER.
32   *            We just read them out.
33   *
34   * @param pbp  Ptr to the boot params structure
35   *****************************************************************************/
36  PUBLIC void get_boot_params(struct boot_params * pbp)
37  {
38          /**
39           * Boot params should have been saved at BOOT_PARAM_ADDR.
40           */
41          int * p = (int*)BOOT_PARAM_ADDR;
42          assert(p[BI_MAG] == BOOT_PARAM_MAGIC);
43
44          pbp->mem_size = p[BI_MEM_SIZE];
```

```
45          pbp->kernel_file = (unsigned char *)(p[BI_KERNEL_FILE]);
46
47          /**
48           * the kernel file should be a ELF executable,
49           * check it's magic number
50           */
51          assert(memcmp(pbp->kernel_file, ELFMAG, SELFMAG) == 0);
52  }
53
54  /*****************************************************************************
55                                      get_kernel_map
56   *
57   *****************************************************************************/
58  /**
59   * <Ring 0~1> Parse the kernel file, get the memory range of the kernel image.
60   *
61   * - The meaning of 'base':     base => first_valid_byte
62   * - The meaning of 'limit':    base + limit => last_valid_byte
63   *
64   * @param b   Memory base of kernel.
65   * @param l   Memory limit of kernel.
66   *****************************************************************************/
67  PUBLIC int get_kernel_map(unsigned int * b, unsigned int * l)
68  {
69          struct boot_params bp;
70          get_boot_params(&bp);
71
72          Elf32_Ehdr* elf_header = (Elf32_Ehdr*)(bp.kernel_file);
73
74          /* the kernel file should be in ELF format */
75          if (memcmp(elf_header->e_ident, ELFMAG, SELFMAG) != 0)
76                  return -1;
77
78          *b = ~0;
79          unsigned int t = 0;
80          int i;
81          for (i = 0; i < elf_header->e_shnum; i++) {
82                  Elf32_Shdr* section_header =
83                          (Elf32_Shdr*)(bp.kernel_file +
84                                        elf_header->e_shoff +
85                                        i * elf_header->e_shentsize);
86
87                  if (section_header->sh_flags & SHF_ALLOC) {
88                          int bottom = section_header->sh_addr;
89                          int top = section_header->sh_addr +
90                                    section_header->sh_size;
91
92                          if (*b > bottom)
93                                  *b = bottom;
94                          if (t < top)
95                                  t = top;
96                  }
97          }
98          assert(*b < t);
99          *l = t - *b - 1;
100
101         return 0;
102 }
```

说了这么多，都是为了代码10.3第77行使用函数get_kernel_map()来得到内核用到的内存范围。之前提到过，这个范围用现成的工具（如 readelf）可以很容易地得到。在本节中，内核占用0x1000～0x3BFA8的内存空间，大约240KB。

接下来我们处理图10.1中的三个关系，首先是①和②。原先它们在两个地方处理，现在我们把它们合在一处，见代码10.6。

代码 10.6 关系①和② (chapter10/a/kernel/protect.c)

```
179     int i;
180     for (i = 0; i < NR_TASKS + NR_PROCS; i++) {
181             memset(&proc_table[i], 0, sizeof(struct proc));
182
183             proc_table[i].ldt_sel = SELECTOR_LDT_FIRST + (i << 3);
184             assert(INDEX_LDT_FIRST + i < GDT_SIZE);
185             init_desc(&gdt[INDEX_LDT_FIRST + i],
186                       makelinear(SELECTOR_KERNEL_DS, proc_table[i].ldts),
187                       LDT_SIZE * sizeof(struct descriptor) - 1,
188                       DA_LDT);
189     }
```

到这里，看上去图10.1我们已经用代码实现了大半了，剩下的内容需要MM上场了。由于MM将接收用户的消息，所以我们不妨先完成用户进程和MM的桥梁部分，即fork()函数的界面。

10.1.3 fork()库函数

根据代码10.1中fork()的用法，我们很容易写一个库函数，见代码10.7。

代码 10.7 fork() (chapter10/a/lib/fork.c)

```
23  /*****************************************************************************
24   *                                fork
25   *****************************************************************************/
26  /**
27   * Create a child process, which is actually a copy of the caller.
28   *
29   * @return  On success, the PID of the child process is returned in the
30   *          parent's thread of execution, and a 0 is returned in the child's
31   *          thread of execution.
32   *          On failure, a -1 will be returned in the parent's context, no
33   *          child process will be created.
34   *****************************************************************************/
35  PUBLIC int fork()
36  {
37          MESSAGE msg;
38          msg.type = FORK;
39
40          send_recv(BOTH, TASK_MM, &msg);
41          assert(msg.type == SYSCALL_RET);
42          assert(msg.RETVAL == 0);
43
44          return msg.PID;
45  }
```

也就是说，我们调用fork()时，MM将收到一个FORK消息，这跟FS处理各种消息是类似的。

10.1.4 MM

跟FS一样，MM要有一个主消息循环，见代码10.8。

代码 10.8 MM 主消息循环（chapter10/a/mm/main.c）

```
28  /*****************************************************************************
29   *                                task_mm
30   *****************************************************************************/
31  /**
32   * <Ring 1> The main loop of TASK MM.
33   *
34   *****************************************************************************/
35  PUBLIC void task_mm()
36  {
37          init_mm();
38
39          while (1) {
40                  send_recv(RECEIVE, ANY, &mm_msg);
41                  int src = mm_msg.source;
42                  int reply = 1;
43
44                  int msgtype = mm_msg.type;
45
46                  switch (msgtype) {
47                  case FORK:
48                          mm_msg.RETVAL = do_fork();
49                          break;
...
61                  default:
62                          dump_msg("MM::unknown_msg", &mm_msg);
63                          assert(0);
64                          break;
65                  }
66
67                  if (reply) {
68                          mm_msg.type = SYSCALL_RET;
69                          send_recv(SEND, src, &mm_msg);
70                  }
71          }
72  }
73
74  /*****************************************************************************
75   *                                init_mm
76   *****************************************************************************/
77  /**
78   * Do some initialization work.
79   *
80   *****************************************************************************/
81  PRIVATE void init_mm()
82  {
83          struct boot_params bp;
84          get_boot_params(&bp);
85
86          memory_size = bp.mem_size;
87
88          /* print memory size */
89          printl("{MM} memsize:%dMB\n", memory_size / (1024 * 1024));
90  }
```

当 MM 接到FORK消息后，调用do_fork()来处理，见代码10.9。

代码 10.9 do_fork()（chapter10/a/mm/forkexit.c）

```
26  /*****************************************************************************
27   *                                do_fork
28   *****************************************************************************/
29  /**
30   * Perform the fork() syscall.
31   *
32   * @return  Zero if success, otherwise -1.
```

```
 *****************************************************************************/
PUBLIC int do_fork()
{
        /* find a free slot in proc_table */
        struct proc* p = proc_table;
        int i;
        for (i = 0; i < NR_TASKS + NR_PROCS; i++,p++)
                if (p->p_flags == FREE_SLOT)
                        break;

        int child_pid = i;
        assert(p == &proc_table[child_pid]);
        assert(child_pid >= NR_TASKS + NR_NATIVE_PROCS);
        if (i == NR_TASKS + NR_PROCS) /* no free slot */
                return -1;
        assert(i < NR_TASKS + NR_PROCS);

        /* duplicate the process table */
        int pid = mm_msg.source;
        u16 child_ldt_sel = p->ldt_sel;
        *p = proc_table[pid];
        p->ldt_sel = child_ldt_sel;
        p->p_parent = pid;
        sprintf(p->name, "%s_%d", proc_table[pid].name, child_pid);

        /* duplicate the process: T, D & S */
        struct descriptor * ppd;

        /* Text segment */
        ppd = &proc_table[pid].ldts[INDEX_LDT_C];
        /* base of T-seg, in bytes */
        int caller_T_base  = reassembly(ppd->base_high, 24,
                                        ppd->base_mid,  16,
                                        ppd->base_low);
        /* limit of T-seg, in 1 or 4096 bytes,
           depending on the G bit of descriptor */
        int caller_T_limit = reassembly(0, 0,
                                        (ppd->limit_high_attr2 & 0xF), 16,
                                        ppd->limit_low);
        /* size of T-seg, in bytes */
        int caller_T_size  = ((caller_T_limit + 1) *
                              ((ppd->limit_high_attr2 & (DA_LIMIT_4K >> 8)) ?
                               4096 : 1));

        /* Data & Stack segments */
        ppd = &proc_table[pid].ldts[INDEX_LDT_RW];
        /* base of D&S-seg, in bytes */
        int caller_D_S_base  = reassembly(ppd->base_high, 24,
                                          ppd->base_mid,  16,
                                          ppd->base_low);
        /* limit of D&S-seg, in 1 or 4096 bytes,
           depending on the G bit of descriptor */
        int caller_D_S_limit = reassembly((ppd->limit_high_attr2 & 0xF), 16,
                                          0, 0,
                                          ppd->limit_low);
        /* size of D&S-seg, in bytes */
        int caller_D_S_size  = ((caller_T_limit + 1) *
                                ((ppd->limit_high_attr2 & (DA_LIMIT_4K >> 8)) ?
                                 4096 : 1));

        /* we don't separate T, D & S segments, so we have: */
        assert((caller_T_base  == caller_D_S_base ) &&
               (caller_T_limit == caller_D_S_limit) &&
               (caller_T_size  == caller_D_S_size ));

        /* base of child proc, T, D & S segments share the same space,
           so we allocate memory just once */
```

```
100         int child_base = alloc_mem(child_pid, caller_T_size);
101         /* int child_limit = caller_T_limit; */
102         printl("{MM} 0x%x <- 0x%x (0x%x bytes)\n",
103                child_base, caller_T_base, caller_T_size);
104         /* child is a copy of the parent */
105         phys_copy((void*)child_base, (void*)caller_T_base, caller_T_size);
106
107         /* child's LDT */
108         init_desc(&p->ldts[INDEX_LDT_C],
109                   child_base,
110                   (PROC_IMAGE_SIZE_DEFAULT - 1) >> LIMIT_4K_SHIFT,
111                   DA_LIMIT_4K | DA_32 | DA_C | PRIVILEGE_USER << 5);
112         init_desc(&p->ldts[INDEX_LDT_RW],
113                   child_base,
114                   (PROC_IMAGE_SIZE_DEFAULT - 1) >> LIMIT_4K_SHIFT,
115                   DA_LIMIT_4K | DA_32 | DA_DRW | PRIVILEGE_USER << 5);
116
117         /* tell FS, see fs_fork() */
118         MESSAGE msg2fs;
119         msg2fs.type = FORK;
120         msg2fs.PID = child_pid;
121         send_recv(BOTH, TASK_FS, &msg2fs);
122
123         /* child PID will be returned to the parent proc */
124         mm_msg.PID = child_pid;
125
126         /* birth of the child */
127         MESSAGE m;
128         m.type = SYSCALL_RET;
129         m.RETVAL = 0;
130         m.PID = 0;
131         send_recv(SEND, child_pid, &m);
132
133         return 0;
134 }
```

代码总体上分为五个部分，第一部分是分配进程表。第36行到第48行从数组proc_table[]中寻找一个空项，用于存放子进程的进程表。接下来的第50行到第56行将父进程的进程表原原本本地赋给子进程。

第二部分是分配内存。由于子进程是父进程的副本，所以首先需要得到父进程的内存占用情况，这由读取 LDT 来完成（第58行到第91行）。注意其中以“_limit”结尾的变量表示的是段的“界限”，它可能是以字节为单位，也可能是以 4096 字节为单位，这取决于描述符的粒度位（详见第3.1.4节）。经过计算，段的以字节为单位的实际大小放在以“_size”结尾的变量中。我们不仔细区分代码、数据和堆栈，也就是说，三个段指向相同的地址空间，在第94行我们用了个assert来保证这一点。

有了父进程的内存占用情况，下面就可以分配内存了。分配内存的函数为alloc_mem()（第100行），我们过会儿再介绍它。接下来的第105行将父进程的内存空间完整地复制了一份给新分配的空间。由于内存空间不同，所以子进程的 LDT 需要更新一下，这是第107行到第115行所做的工作。

事情到这里并没有结束，而是生出枝节。我们知道，进程生出子进程的时候，可能已经打开了文件，我们需要在父进程和子进程之间共享文件[3]，所以我们需要通知 FS，由它来完成相应的工作（第117行到第121行）。

[3]只有共享了文件，父进程和子进程才不至于在写文件时发生相互覆盖的情况。而且正是由于需要共享文件，所以f_desc_table[]才不能被并入进程表中（见第9.10.2节）。

所有这些做完之后，函数就可以返回了（第123行到第131行）。返回之前别忘了，子进程由于使用了跟父进程几乎一样的进程表，所以目前也是挂起状态，我们需要给它也发一个消息，这样不但解除其阻塞，而且将零作为返回值传递给子进程，以便让它知道自己的身份。对于父进程，do_fork()正常返回后会由MM的主消息循环发送消息给它，我们只需要在do_fork()中给mm_msg.PID赋值就好了。

do_fork()的整体过程就是这样了，我们接下来看一看其中的两个旁枝。

10.1.4.1 内存分配

一个成熟的内存分配机制是比较复杂的，但这并不意味着简单的机制就完全行不通。在这里我们就使用了一种十分老土的方案，那就是划定格子，每个格子大小固定。有新的进程需要内存，就给它一个格子，而且这个进程在整个生命周期中，只能使用这个格子的内存。

我们把格子的大小定为1MB，这个数字在一定程度内可随意选取，但至少应该大于内核的大小240KB，这样才能盛得下Init。

与此同时，并不是所有的内存空间都能被Init的子进程使用。0~1MB的空间被内核使用，显然不能用。另外别忘了，FS等进程还使用了一些内存作为缓冲区，我们也要避开。

综合各种因素，我们定义了下面这几个宏：

代码10.10 内存分配相关的宏（chapter10/a/include/sys/proc.h）

```
88  /**
89   * All forked proc will use memory above PROCS_BASE.
90   *
91   * @attention make sure PROCS_BASE is higher than any buffers, such as
92   *            fsbuf, mmbuf, etc
93   * @see global.c
94   * @see global.h
95   */
96  #define PROCS_BASE                0xA00000 /* 10 MB */
97  #define PROC_IMAGE_SIZE_DEFAULT 0x100000 /*  1 MB */
98  #define PROC_ORIGIN_STACK         0x400    /*  1 KB */
```

我们将PROCS_BASE定义成0xA00000，也就是说，将10MB以上的空间给用户进程使用。由于我们总共有32MB的内存，所以同时最多可以有12个用户进程同时运行。在一个装有1GB内存的真实机器中，用户进程数最多可达近一千个，这听起来还不错。

PROC_ORIGIN_STACK的用途我们下文中再说。

好了，分配方案一经确定，alloc_mem()也就容易写了，请看代码10.11。

代码10.11 alloc_mem()（chapter10/a/mm/main.c）

```
/*****************************************************************************
 *                                alloc_mem
 *****************************************************************************/
/**
 * Allocate a memory block for a proc.
 *
 * @param pid  Which proc the memory is for.
 * @param memsize  How many bytes is needed.
```

```
 *
 * @return  The base of the memory just allocated.
 *****************************************************************************/
PUBLIC int alloc_mem(int pid, int memsize)
{
        assert(pid >= (NR_TASKS + NR_NATIVE_PROCS));
        if (memsize > PROC_IMAGE_SIZE_DEFAULT) {
                panic("unsupported memory request: %d. "
                      "(should be less than %d)",
                      memsize,
                      PROC_IMAGE_SIZE_DEFAULT);
        }

        int base = PROCS_BASE +
                   (pid - (NR_TASKS + NR_NATIVE_PROCS)) * PROC_IMAGE_SIZE_DEFAULT;

        if (base + memsize >= memory_size)
                panic("memory allocation failed. pid:%d", pid);

        return base;
}
```

我们这种分配方案，其实就是建立了 PID 和进程内存空间之间的映射关系，或者说，内存空间是 PID 的一个函数。这一方案的缺点非常明显，对于小一点的程序，1MB 的内存太浪费，而对于大一点的程序，1MB 又可能会不够。不过我们先不管那么多，内核我们写了这么久，在内存中才只占约四分之一个 1MB，可见 1MB 也没有那么小；至于浪费嘛，那是属于优化的范畴，我们以后再说。总之，跟设计一个简单的 FS 思路相同，我们先不求好，只求先有个能用的版本。

10.1.4.2 对文件描述符的处理

上文我们说到，fork一个子进程还需要 FS 来协助，其实 FS 主要是要增加两个计数器，请看代码10.12。

代码 10.12 fs_fork()（chapter10/a/fs/main.c）

```
/*****************************************************************************
 *                                  fs_fork
 *****************************************************************************/
/**
 * Perform the aspects of fork() that relate to files.
 *
 * @return Zero if success, otherwise a negative integer.
 *****************************************************************************/
PRIVATE int fs_fork()
{
        int i;
        struct proc* child = &proc_table[fs_msg.PID];
        for (i = 0; i < NR_FILES; i++) {
                if (child->filp[i]) {
                        child->filp[i]->fd_cnt++;
                        child->filp[i]->fd_inode->i_cnt++;
                }
        }

        return 0;
}
```

隶属于inode结构的i_cnt这个计数器我们已经很熟悉了，只要有进程使用这个inode，其i_cnt就应该加1。属于file_desc结构的fd_cnt是我们新增加的成员，它的道理很类似。假设进程P生成了进程C，那么P和C共享使用f_desc_table[]中的同一个file_desc结构，这时fd_cnt为2，表明有两个进程在使用这一结构。等进程P或C退出时，fd_cnt自减。若P和C都已退出，fd_cnt自减为零，这时系统应将fd_inode赋值为零，这样这个f_desc_tabel[]条目就又可被使用了（见代码10.16）。

10.1.5 运行

好了，fork()系统调用的前后因果我们已经全都了解了，现在编译运行一下看看，见图10.2。

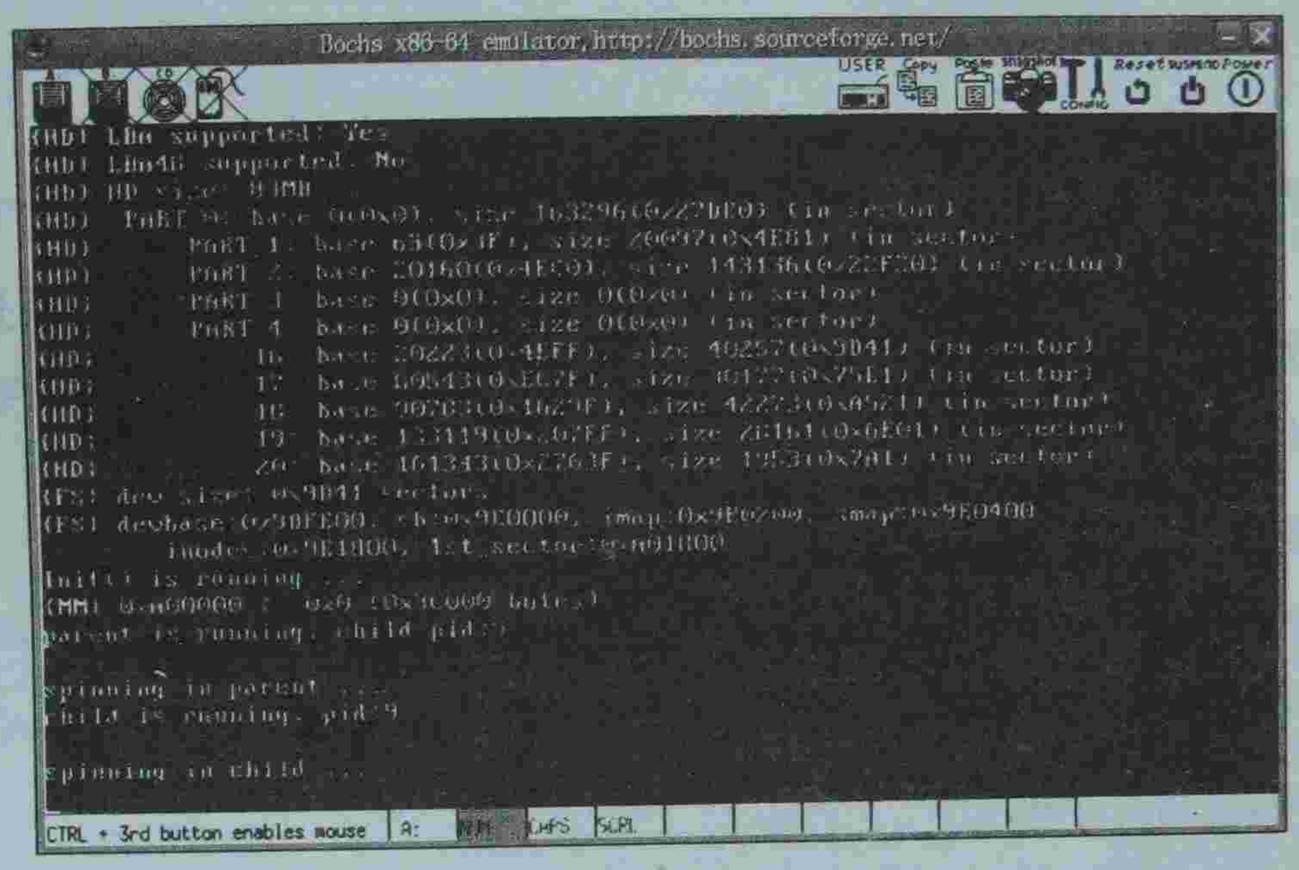


图 10.2 fork

不出所料，我们看到了父进程打印出的“parent is running”和子进程打印出的“child is running”，这说明我们的fork()成功了！

10.2 exit和wait

生成子进程的最重要的fork()我们已经有了，但这还不够，因为进程有出生就有死亡。不要看到“死亡”就觉得很沉重，进程比人好一点，它可以随时“重生”，我们重新fork一下，它就又活过来了[4]。

让进程死亡的系统调用叫做exit()，直译作“退出”，其实它叫“自杀”更贴切，因为exit()通常是由要走的进程自己调用的，而且调用之后这个进程不是“退出”了，而是干脆消失了。

那么wait()是干什么的呢？如果你写过 shell 脚本的话，就很容易理解它的作用。我们执行一个程序之后，有时需要判断其返回值，这个返回值通常是

[4]或许上帝也可以让人随时重生，这超出本书的讨论范畴。

通过$?得到。而你获取$?时，你执行的程序显然已经退出了（所以它才有返回值）。容易理解，这个返回值是我们执行的进程返回给 shell 的。换言之，是子进程返回给父进程的。父进程得到返回值的方法，就是执行一个wait()挂起，等子进程退出时，wait()调用方结束，并且父进程因此得到返回值。

如果用代码表示的话，那么应该是下面这个样子：

代码 10.13 Init（chapter10/b/kernel/main.c）

```
165     int pid = fork();
166     if (pid != 0) { /* parent process */
167             printf("parent is running, child pid:%d\n", pid);
168             int s;
169             int child = wait(&s);
170             printf("child (%d) exited with status: %d.\n", child, s);
171     }
172     else {  /* child process */
173             printf("child is running, pid:%d\n", getpid());
174             exit(123);
175     }
```

如果一切正常，子进程将会退出，而父进程将会得到返回值并打印出来。

跟fork()类似，exit()和wait()这两个函数同样是发送消息给 MM，它们发送的消息分别是EXIT和WAIT。在 MM 中，这两个消息由do_exit()和do_wait()两个函数来处理（见代码10.14）。

代码 10.14 do_exit() 和 do_wait()（chapter10/b/mm/forkexit.c）

```
136 /*****************************************************************************
137  *                                do_exit
138  *****************************************************************************/
139 /**
140  * Perform the exit() syscall.
141  *
142  * If proc A calls exit(), then MM will do the following in this routine:
143  *      <1> inform FS so that the fd-related things will be cleaned up
144  *      <2> free A's memory
145  *      <3> set A.exit_status, which is for the parent
146  *      <4> depends on parent's status. if parent (say P) is:
147  *              (1) WAITING
148  *                      - clean P's WAITING bit, and
149  *                      - send P a message to unblock it
150  *                      - release A's proc_table[] slot
151  *              (2) not WAITING
152  *                      - set A's HANGING bit
153  *      <5> iterate proc_table[], if proc B is found as A's child, then:
154  *              (1) make INIT the new parent of B, and
155  *              (2) if INIT is WAITING and B is HANGING, then:
156  *                      - clean INIT's WAITING bit, and
157  *                      - send INIT a message to unblock it
158  *                      - release B's proc_table[] slot
159  *                  else
160  *                      if INIT is WAITING but B is not HANGING, then
161  *                              - B will call exit()
162  *                      if B is HANGING but INIT is not WAITING, then
163  *                              - INIT will call wait()
164  *
165  * TERMs:
166  *      - HANGING: everything except the proc_table entry has been cleaned up.
167  *      - WAITING: a proc has at least one child, and it is waiting for the
168  *                 child(ren) to exit()
169  *      - zombie: say P has a child A, A will become a zombie if
170  *              - A exit(), and
```

```
 *              - P does not wait(), neither does it exit(). that is to say, P just
 *                keeps running without terminating itself or its child
 *
 * @param status  Exiting status for parent.
 *
 *****************************************************************************/
PUBLIC void do_exit(int status)
{
        int i;
        int pid = mm_msg.source; /* PID of caller */
        int parent_pid = proc_table[pid].p_parent;
        struct proc * p = &proc_table[pid];

        /* tell FS, see fs_exit() */
        MESSAGE msg2fs;
        msg2fs.type = EXIT;
        msg2fs.PID = pid;
        send_recv(BOTH, TASK_FS, &msg2fs);

        free_mem(pid);

        p->exit_status = status;

        if (proc_table[parent_pid].p_flags & WAITING) { /* parent is waiting */
                proc_table[parent_pid].p_flags &= ~WAITING;
                cleanup(&proc_table[pid]);
        }
        else { /* parent is not waiting */
                proc_table[pid].p_flags |= HANGING;
        }

        /* if the proc has any child, make INIT the new parent */
        for (i = 0; i < NR_TASKS + NR_PROCS; i++) {
                if (proc_table[i].p_parent == pid) { /* is a child */
                        proc_table[i].p_parent = INIT;
                        if ((proc_table[INIT].p_flags & WAITING) &&
                            (proc_table[i].p_flags & HANGING)) {
                                proc_table[INIT].p_flags &= ~WAITING;
                                cleanup(&proc_table[i]);
                        }
                }
        }
}

/*****************************************************************************
 *                                cleanup
 *****************************************************************************/
/**
 * Do the last jobs to clean up a proc thoroughly:
 *      - Send proc's parent a message to unblock it, and
 *      - release proc's proc_table[] entry
 *
 * @param proc  Process to clean up.
 *****************************************************************************/
PRIVATE void cleanup(struct proc * proc)
{
        MESSAGE msg2parent;
        msg2parent.type = SYSCALL_RET;
        msg2parent.PID = proc2pid(proc);
        msg2parent.STATUS = proc->exit_status;
        send_recv(SEND, proc->p_parent, &msg2parent);

        proc->p_flags = FREE_SLOT;
}

/*****************************************************************************
 *                                do_wait
```

```
238  *****************************************************************************/
239  /**
240   * Perform the wait() syscall.
241   *
242   * If proc P calls wait(), then MM will do the following in this routine:
243   *     <1> iterate proc_table[],
244   *         if proc A is found as P's child and it is HANGING
245   *             - reply to P (cleanup() will send P a messageto unblock it)
246   *             - release A's proc_table[] entry
247   *             - return (MM will go on with the next message loop)
248   *     <2> if no child of P is HANGING
249   *             - set P's WAITING bit
250   *     <3> if P has no child at all
251   *             - reply to P with error
252   *     <4> return (MM will go on with the next message loop)
253   *
254   *****************************************************************************/
255  PUBLIC void do_wait()
256  {
257          int pid = mm_msg.source;
258
259          int i;
260          int children = 0;
261          struct proc* p_proc = proc_table;
262          for (i = 0; i < NR_TASKS + NR_PROCS; i++,p_proc++) {
263                  if (p_proc->p_parent == pid) {
264                          children++;
265                          if (p_proc->p_flags & HANGING) {
266                                  cleanup(p_proc);
267                                  return;
268                          }
269                  }
270          }
271
272          if (children) {
273                  /* has children, but no child is HANGING */
274                  proc_table[pid].p_flags |= WAITING;
275          }
276          else {
277                  /* no child at all */
278                  MESSAGE msg;
279                  msg.type = SYSCALL_RET;
280                  msg.PID = NO_TASK;
281                  send_recv(SEND, pid, &msg);
282          }
283  }
```

想像得出，do_exit/do_wait跟msg_send/msg_receive这两对函数是有点类似的，它们最终都是实现一次“握手”。

假设进程 P 有子进程 A。而 A 调用exit()，那么 MM 将会：

1. 告诉 FS：A 退出，请做相应处理
2. 释放 A 占用的内存
3. 判断 P 是否正在WAITING
    - 如果是：
        - 清除 P 的WAITING位
        - 向 P 发送消息以解除阻塞（到此 P 的wait()函数结束）
        - 释放 A 的进程表项（到此 A 的exit()函数结束）

- 如果否:
  - 设置A的HANGING位

4. 遍历proc_table[]，如果发现A有子进程B，那么
   - 将Init进程设置为B的父进程（换言之，将B过继给Init）
   - 判断是否满足Init正在WAITING且B正在HANGING
     - 如果是:
       - ★ 清除Init的WAITING位
       - ★ 向Init发送消息以解除阻塞（到此Init的wait()函数结束）
       - ★ 释放B的进程表项（到此B的exit()函数结束）
     - 如果否:
       - ★ 如果Init正在WAITING但B并没有HANGING，那么“握手”会在将来B调用exit()时发生
       - ★ 如果B正在HANGING但Init并没有WAITING，那么“握手”会在将来Init调用wait()时发生

如果P调用wait()，那么MM将会:

1. 遍历proc_tabel[]，如果发现A是P的子进程，并且它正在HANGING，那么
   - 向P发送消息以解除阻塞（到此P的wait()函数结束）
   - 释放A的进程表项（到此A的exit()函数结束）
2. 如果P的子进程没有一个在HANGING，则
   - 设P的WAITING位
3. 如果P压根儿没有子进程，则
   - 向P发送消息，消息携带一个表示出错的返回值（到此P的wait()函数结束）

为配合exit()和wait()，进程又多了两种状态：WAITING和HANGING。如果一个进程X被置了HANGING位，那么X的所有资源都已被释放，只剩一个进程表项还占着。为什么要占着进程表项不释放呢？因为这个进程表项里面有个新成员：exit_status，它记录了X的返回值。只有当X的父进程通过调用wait()取走了这个返回值，X的进程表项才被释放。

如果一个进程Y被置了WAITING位，意味着Y至少有一个子进程，并且正在等待某个子进程退出。

读者可能会想，如果一个子进程Z试图退出，但它的父进程却没有调用wait()，那Z的进程表项岂不一直占着得不到释放吗？事情的确如此，而且有个名称专门用来称呼像Z这样的进程，叫做“僵尸进程”(zombie)。别害怕，它只是个进程而已，不会咬人。

如果一个进程 Q 有子进程，但它没有wait()就自己先exit()了，那么 Q 的子进程不会变成 zombie，因为 MM 会把它们过继给 Init，变成 Init 的子进程。也就是说，Init 应该被设计成不停地调用wait()，以便让 Q 的子进程们退出并释放进程表项。

既然如此，让我们先改造一下 Init，在结尾处添加不停wait()的代码：

代码 10.15 Init（chapter10/b/kernel/main.c）

```
177         while (1) {
178                 int s;
179                 int child = wait(&s);
180                 printf("child (%d) exited with status: %d.\n", child, s);
181         }
```

这样do_exit()和do_wait()做的事情我们就了解清楚了，不过事情还没完，因为 FS 接到 MM 的进程退出消息之后还要做些工作，见代码10.16。

代码 10.16 fs_exit()（chapter10/b/fs/main.c）

```
535 /*****************************************************************************
536  *                                fs_exit
537  *****************************************************************************/
538 /**
539  * Perform the aspects of exit() that relate to files.
540  *
541  * @return Zero if success.
542  *****************************************************************************/
543 PRIVATE int fs_exit()
544 {
545         int i;
546         struct proc* p = &proc_table[fs_msg.PID];
547         for (i = 0; i < NR_FILES; i++) {
548                 if (p->filp[i]) {
549                         /* release the inode */
550                         p->filp[i]->fd_inode->i_cnt--;
551                         /* release the file desc slot */
552                         if (--p->filp[i]->fd_cnt == 0)
553                                 p->filp[i]->fd_inode = 0;
554                         p->filp[i] = 0;
555                 }
556         }
557         return 0;
558 }
```

完成了exit()和wait()，子进程的产生和消亡这个过程就都有了，我们下面就来编译运行一下看看，见图10.3。

非常好，我们看到了“child (9) exited with status: 123.”这一行输出，这是由父进程 Init 打印的，看上去一切良好。

由于exit()和wait()的调用顺序可能不同，加上进程“过继”等诸多可能性，所以要想保证退出和等待两个过程在所有情况下运行良好还是需要经过多一点的测试，这里不多说，读者可以自行测试。

10.3 exec

我们说了fork()，还说了exit()和wait()，该回头想想我们之前设定的目标了。我们的目标是要实现一个可以执行命令的 shell，可无论 Init 进

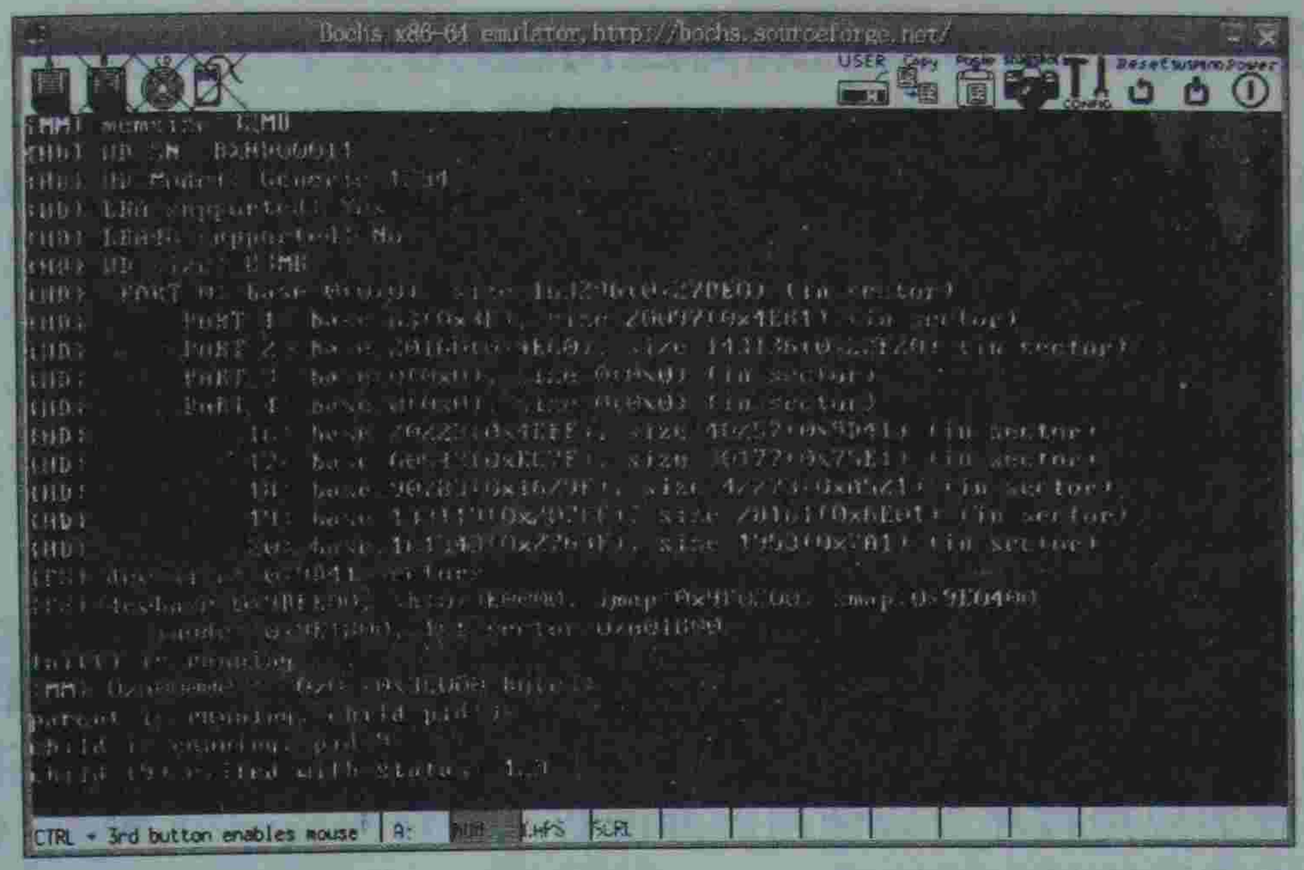


图 10.3 exit 和 wait

程fork出多少进程，它也都只是 Init 而已。所以我们还需要一个系统调用，它就是exec()。

10.3.1 认识exec

exec的语义很简单，它将当前的进程映像替换成另一个。也就是说，我们可以从硬盘上读取另一个可执行的文件，用它替换掉刚刚被fork出来的子进程，于是被替换的子进程摇身一边，就成了彻头彻尾的新鲜进程了。

以 shell 中常见的 echo 命令为例。我们输入“echo hello world”，shell 就会fork出一个子进程 A，这时 A 跟 shell 一模一样，fork结束后父进程和子进程分别判断自己的fork()返回值，如果是 0 则表明当前进程为子进程 A，这时 A 马上执行一个exec()，于是进程 A 的内存映像被 echo 替换，它就变成 echo 了。这一过程可用代码10.17表示。

代码 10.17 exec() 的执行过程（chapter10/c/kernel/main.c）

```
247         int pid = fork();
248         if (pid != 0) { /* parent process */
249                 printf("parent is running, child pid:%d\n", pid);
250                 int s;
251                 int child = wait(&s);
252                 printf("child (%d) exited with status: %d.\n", child, s);
253         }
254         else {  /* child process */
255                 execl("/echo", "echo", "hello", "world", 0);
256         }
```

上述代码中用到的execl()是exec的一种形式，我们下文中细说。无论如何，这一过程看上去很简单，但是我们面临两个问题，一是要实现exec()，二是我们还没有一个可执行的程序 echo 呢。在这两个问题中，要顺利执行exec()依赖于可执行程序的存在。所以我们先需要写一个 echo，下面就来做这项工作。

10.3.2 为自己的操作系统编写应用程序

echo 将以操作系统中普通应用程序的身份出现，它跟操作系统的接口是系统调用。其实本质上，一个应用程序只能调用两种东西：属于自己的函数，以及中断（系统调用其实就是软中断）。可是根据我们写程序的经验，一个应用程序通常都会调用一些现成的函数，很少见写程序时里面满是中断调用的。这是因为编译器偷偷地为我们链接了 C 运行时库（CRT），库里面有已经编译好的库函数代码。这样两者链接起来，应用程序就能正确运行了。

假如我们要写一个 echo，最笨的办法就是将`send_recv()`、`printf()`、`write()`等所有用到的系统调用的代码都复制到源文件中，然后编译一下。这肯定是能成功的，但更优雅的做法是制作一个类似 C 运行时库的东西。我们把之前已经写就的应用程序可以使用的库函数单独链接成一个文件，每次写应用程序的时候直接链接起来就好了。

到目前为止，可以被用来链接成库的文件及其包含的主要函数有这些：

- 两个真正的系统调用：`sendrec`和`printx`
  - lib/syscall.asm
- 字符串操作：`memcpy`、`memset`、`strcpy`、`strlen`
  - lib/string.asm
- FS 的接口
  - lib/open.c
  - lib/read.c
  - lib/write.c
  - lib/close.c
  - lib/unlink.c
- MM 的接口
  - lib/fork.c
  - lib/exit.c
  - lib/wait.c
- SYS 的接口
  - lib/getpid.c
- 其他
  - lib/misc.c
  - lib/vsprintf.c
  - lib/printf.c

我们就把这些函数单独链接成一个库，把它起名为orangescrt.a，表明这是我们的C运行时库。做这样一个库的方法非常简单：

```
▷ ar rcs lib/orangescrt.a lib/syscall.o lib/printf.o lib/vsprintf.o \
lib/string.o lib/misc.o lib/open.o lib/read.o lib/write.o lib/close.o \
lib/unlink.o lib/getpid.o lib/fork.o lib/exit.o lib/wait.o
```

有了库，我们就可以放心地写一个应用程序了，先写一个最简单的echo，见代码10.18。

代码 10.18 chapter10/c/command/echo.c

```
1  #include "stdio.h"
2
3  int main(int argc, char * argv[])
4  {
5          int i;
6          for (i = 1; i < argc; i++)
7                  printf("%s%s", i == 1 ? "" : "␣", argv[i]);
8          printf("\n");
9
10         return 0;
11 }
```

没有考虑任何异常情况，这个程序写得极为简单。我们编译链接之：

```
▷ gcc -I ../include/ -c -fno-builtin -Wall -o echo.o echo.c
▷ ld -Ttext 0x1000 -o echo echo.o ../lib/orangescrt.a
ld: warning: cannot find entry symbol _start; defaulting to 0000000000001000
```

哦？怎么了？居然出错？啊，原来是找不到符号_start。你可能想起来了，链接器需要找到_start作为入口，没关系，我们马上写一个，见代码10.19。

代码 10.19 chapter10/c/command/start.asm

```
1  ;; start.asm
2
3  extern  main
4  extern  exit
5
6  bits 32
7
8  [section .text]
9
10 global _start
11
12 _start:
13         push    eax
14         push    ecx
15         call    main
16         ;; need not clean up the stack here
17
18         push    eax
19         call    exit
20
21         hlt     ; should never arrive here
```

千万不要小看_start，虽然只有寥寥几行，但它肩负三项使命：

- 为main()函数准备参数：argc和argv
- 调用main()

- 将main()函数的返回值通过exit()传递给父进程

怎么样？了不起吧，五行代码就做三件事情。之所以小小的start.asm要肩负如此多且重大的使命，原因就在于main()函数本质上只不过是个普通函数，在编译器的眼里，它跟其他任何函数没什么两样[5]。既然是个普通函数，那么自然需要别人为它准备参数，调用它，以及做清理工作。这个角色便是由_start来扮演了。

好了，下面再来链接一下，这次别忘了加上start.o：

```
▷ gcc -I ../include/ -c -fno-builtin -Wall -o echo.o echo.c
▷ nasm -I ../include/ -f elf -o start.o start.asm
▷ ld -Ttext 0x1000 -o echo echo.o start.o ../lib/orangescrt.a
```

成功了！我们有了自己的应用程序！可是高兴之后，又一个问题摆在面前：如何将这个程序放进我们的操作系统中呢？

10.3.3 “安装”应用程序

千万别心焦，既然编译成功了，还愁我们的操作系统用不上吗？不过是多些 effort 罢了。

我们还记得，在 FS 进程启动时，会调用mkfs()创建一个简单的文件系统(第9.9.2节)，里面创建了四个特殊文件：“.”以及“dev_tty[012]”。既然可以创建特殊文件，那么普通文件也是可以创建的，所以我们完全可以在mkfs()中多加几行代码，创建一个普通文件，然后在 Linux 中用 dd 命令将文件内容写入磁盘映像文件，一切就大功告成了。

不过这样一来有个明显的缺点，就是以后我们每写一个应用程序，就得改造mkfs()，很费事，而且容易出错。所以我们可以将这个方法稍作改进，将所有的应用程序文件打成一个 tar 包，做成一个文件，然后放进去，在操作系统启动时将这个包解开，问题就解决了。这个改进方法需要我们额外付出的努力，就是需要写一小段程序来解开 tar 包，不过过会儿你会发现，这项工作非常简单。

总结一下，要想“安装”一些应用程序到我们的文件系统中，需要做如下工作：

- 编写应用程序，并编译链接。
- 将链接好的应用程序打成一个 tar 包：inst.tar。
- 将inst.tar用工具 dd 写入磁盘（映像）的某段特定扇区（假设这一段的首扇区的扇区号为 X）。
- 启动系统，这时mkfs()会在文件系统中建立一个新文件cmd.tar，它的inode中的i_start_sect成员会被设为 X。
- 在某个进程中——比如 Init——将cmd.tar解包，将其中包含的文件存入文件系统。

[5]您若不信，过会儿编译成功之后，试着将start.asm中的两个main（第3行和第15行）以及echo.c中的main都改成“niam”，看看是不是还能编译成功。

我得承认这个方法算不上优雅，如果可以通过软盘或光盘来安装应用程序肯定更酷，但那需要写相应的驱动程序，你也看到了，我已经迫不及待，所以使用这个空降兵硬塞的方法来“安装”应用程序。其实这个方法也没有那么差，好歹我们还可额外练习一下如何解开 tar 包。

我们首先来改造mkfs()，在其中增加一个文件，见代码10.20。

代码 10.20 在 mkfs() 中增加文件 cmd.tar（节自 chapter10/c/fs/main.c）

```
170 /*****************************************************************************
171                                 mkfs
172  *****************************************************************************/
173 /**
174  * <Ring 1> Make a available Orange'S FS in the disk. It will
175  *          - Write a super block to sector 1.
176  *          - Create three special files: dev_tty0, dev_tty1, dev_tty2
177  *          - Create a file cmd.tar
178  *          - Create the inode map
179  *          - Create the sector map
180  *          - Create the inodes of the files
181  *          - Create '/', the root directory
182  *****************************************************************************/
183 PRIVATE void mkfs()
184 {
    ...
239         /**************************/
240         /*      inode map         */
241         /**************************/
242         memset(fsbuf, 0, SECTOR_SIZE);
243         for (i = 0; i < (NR_CONSOLES + 3); i++)
244                 fsbuf[0] |= 1 << i;
245
⇒ 246         assert(fsbuf[0] == 0x3F);/* 0011 1111 :
247                                     *   || ||||
248                                     *   || |||'--- bit 0 : reserved
249                                     *   || ||'---- bit 1 : the first inode,
250                                     *   || ||              which indicates '/'
251                                     *   || |'----- bit 2 : /dev_tty0
252                                     *   || '------ bit 3 : /dev_tty1
253                                     *   |'-------- bit 4 : /dev_tty2
⇒ 254                                 *   '--------- bit 5 : /cmd.tar
255                                     */
256         WR_SECT(ROOT_DEV, 2);
257
258         /**************************/
259         /*      secter map        */
260         /**************************/
    ...
⇒ 280         /* cmd.tar */
281         /* make sure it'll not be overwritten by the disk log */
282         assert(INSTALL_START_SECT + INSTALL_NR_SECTS <
283                sb.nr_sects - NR_SECTS_FOR_LOG);
284         int bit_offset = INSTALL_START_SECT -
285                 sb.n_1st_sect + 1; /* sect M <-> bit (M - sb.n_1stsect + 1) */
286         int bit_off_in_sect = bit_offset % (SECTOR_SIZE * 8);
287         int bit_left = INSTALL_NR_SECTS;
288         int cur_sect = bit_offset / (SECTOR_SIZE * 8);
289         RD_SECT(ROOT_DEV, 2 + sb.nr_imap_sects + cur_sect);
290         while (bit_left) {
291                 int byte_off = bit_off_in_sect / 8;
292                 /* this line is ineffecient in a loop, but I don't care */
293                 fsbuf[byte_off] |= 1 << (bit_off_in_sect % 8);
294                 bit_left--;
295                 bit_off_in_sect++;
296                 if (bit_off_in_sect == (SECTOR_SIZE * 8)) {
297                         WR_SECT(ROOT_DEV, 2 + sb.nr_imap_sects + cur_sect);
```

```
298                     cur_sect++;
299                     RD_SECT(ROOT_DEV, 2 + sb.nr_imap_sects + cur_sect);
300                     bit_off_in_sect = 0;
301             }
302     }
303     WR_SECT(ROOT_DEV, 2 + sb.nr_imap_sects + cur_sect);
304
305     /**************************/
306     /*      inodes            */
307     /**************************/
308     /* inode of '/' */
309     memset(fsbuf, 0, SECTOR_SIZE);
310     struct inode * pi = (struct inode*)fsbuf;
311     pi->i_mode = I_DIRECTORY;
⇒ 312   pi->i_size = DIR_ENTRY_SIZE * 5; /* 5 files:
313                                       * '.',
314                                       * 'dev_tty0', 'dev_tty1', 'dev_tty2',
⇒ 315                                   * 'cmd.tar'
316                                       */
317     pi->i_start_sect = sb.n_1st_sect;
318     pi->i_nr_sects = NR_DEFAULT_FILE_SECTS;
...
⇒ 327   /* inode of '/cmd.tar' */
328     pi = (struct inode*)(fsbuf + (INODE_SIZE * (NR_CONSOLES + 1)));
329     pi->i_mode = I_REGULAR;
330     pi->i_size = INSTALL_NR_SECTS * SECTOR_SIZE;
331     pi->i_start_sect = INSTALL_START_SECT;
332     pi->i_nr_sects = INSTALL_NR_SECTS;
333     WR_SECT(ROOT_DEV, 2 + sb.nr_imap_sects + sb.nr_smap_sects);
334
335     /**************************/
336     /*          '/'           */
337     /**************************/
338     memset(fsbuf, 0, SECTOR_SIZE);
339     struct dir_entry * pde = (struct dir_entry *)fsbuf;
340
341     pde->inode_nr = 1;
342     strcpy(pde->name, ".");
343
344     /* dir entries of '/dev_tty0~2' */
345     for (i = 0; i < NR_CONSOLES; i++) {
346             pde++;
347             pde->inode_nr = i + 2; /* dev_tty0's inode_nr is 2 */
348             sprintf(pde->name, "dev_tty%d", i);
349     }
350     (++pde)->inode_nr = NR_CONSOLES + 2;
⇒ 351   strcpy(pde->name, "cmd.tar");
352     WR_SECT(ROOT_DEV, sb.n_1st_sect);
353 }
```

除了写 sector-map 的一段代码有点丑，其他都还好。注意其中我们引入的两个宏：

代码 10.21 两个宏（节自 chapter10/c/include/sys/config.h）

```
10  /**
11   * Some sector are reserved for us (the gods of the os) to copy a tar file
12   * there, which will be extracted and used by the OS.
13   */
14  #define INSTALL_START_SECT              0x8000
15  #define INSTALL_NR_SECTS                0x800
```

INSTALL_START_SECT便是cmd.tar的首扇区的扇区号，注意它不要越过分区的边界，也不要跟为日志预留的扇区冲突。INSTALL_NR_SECTS这里设置成了0x800，也就是说cmd.tar最多 1MB。

好了，下面是时候将应用程序打包并空降到磁盘映像了：

```
▷ tar vcf inst.tar kernel.bin echo pwd ↩
kernel.bin
echo
pwd
▷ dd if=inst.tar of=../80m.img seek=`echo "obase=10;ibase=16;(\`egrep␣-e
'^ROOT_BASE'␣../boot/include/load.inc␣|␣sed␣-e␣'s/.*0x//g'\`+\`egrep␣-e
'#define[[:space:]]*INSTALL_START_SECT'␣../include/sys/config.h␣|␣sed␣-e
's/.*0x//g'\`)*200" | bc` bs=1 count=`ls -l inst.tar | awk -F "␣" '{print
$5}'` conv=notrunc ↩
102400+0 records in
102400+0 records out
102400 bytes (102 kB) copied, 0.300071 seconds, 341 kB/s
```

打包的命令很简单，读者很可能以前也用过。我们除了将 echo 打入包中，还加入了内核文件kernel.bin和另一个简单的程序 pwd。之所以又加入两个文件，是因为这样可以更好地理解生成的 tar 文件结构。

写入磁盘的 dd 命令稍微复杂一点，其实它是在一个命令中嵌入了其他命令，所以显得很凌乱。其中

```
egrep -e '^ROOT_BASE' ../boot/include/load.inc | sed -e 's/.*0x//g'
```

的作用是从load.inc中找出ROOT_BASE的定义，并取出其十六进制的值，所以其输出为“4EFF”。这是根设备的开始扇区号。

这一部分：

```
egrep -e '#define[[:space:]]*INSTALL_START_SECT' ../include/sys/config.h |
sed -e 's/.*0x//g'
```

的作用是从config.h中找到INSTALL_START_SECT的定义，并取出其十六进制值，其输出为“8000”。这是cmd.tar的开始扇区号。

有了这两个值，命令通过工具 bc 计算cmd.tar的字节偏移：

```
echo "obase=10;ibase=16;(4EFF+8000)*200" | bc
```

这是cmd.tar相对于整个磁盘映像的字节偏移。

另外，这一部分：

```
ls -l inst.tar | awk -F "␣" '{print $5}'
```

得到的是 inst.tar 的文件大小（以字节为单位）。

有了这些，我们就可以用 dd 命令来写入了，所以最终上面那个复杂的命令到最后执行的是：

```
dd if=inst.tar of=../80m.img seek=27131392 bs=1 count=102400 conv=notrunc
```

通过这个复杂的命令行，读者也可以管窥 shell 的威力——命令可以组合起来，完成复杂功能，一个命令抵得上一个小程序。

好了，材料都已齐备，下面该下锅烹炒了。我们改造一下 Init，让它可以读取cmd.tar，并且将包解开，见代码10.22。

代码 10.22 解包（chapter10/c/kernel/main.c）

```
149 /**
150  * @struct posix_tar_header
```

```
151  * Borrowed from GNU 'tar'
152  */
153 struct posix_tar_header                 /* byte offset */
154 {
155         char name[100];                 /*   0 */
156         char mode[8];                   /* 100 */
157         char uid[8];                    /* 108 */
158         char gid[8];                    /* 116 */
159         char size[12];                  /* 124 */
160         char mtime[12];                 /* 136 */
161         char chksum[8];                 /* 148 */
162         char typeflag;                  /* 156 */
163         char linkname[100];             /* 157 */
164         char magic[6];                  /* 257 */
165         char version[2];                /* 263 */
166         char uname[32];                 /* 265 */
167         char gname[32];                 /* 297 */
168         char devmajor[8];               /* 329 */
169         char devminor[8];               /* 337 */
170         char prefix[155];               /* 345 */
171         /* 500 */
172 };
173
174 /*****************************************************************************
175  *                                  untar
176  *****************************************************************************/
177 /**
178  * Extract the tar file and store them.
179  *
180  * @param filename The tar file.
181  *****************************************************************************/
182 void untar(const char * filename)
183 {
184         printf("[extract␣'%s'\n", filename);
185         int fd = open(filename, O_RDWR);
186         assert(fd != -1);
187
188         char buf[SECTOR_SIZE * 16];
189         int chunk = sizeof(buf);
190
191         while (1) {
192                 read(fd, buf, SECTOR_SIZE);
193                 if (buf[0] == 0)
194                         break;
195
196                 struct posix_tar_header * phdr = (struct posix_tar_header *)buf;
197
198                 /* calculate the file size */
199                 char * p = phdr->size;
200                 int f_len = 0;
201                 while (*p)
202                         f_len = (f_len * 8) + (*p++ - '0'); /* octal */
203
204                 int bytes_left = f_len;
205                 int fdout = open(phdr->name, O_CREAT | O_RDWR);
206                 if (fdout == -1) {
207                         printf("␣␣␣␣failed␣to␣extract␣file:␣%s\n", phdr->name);
208                         printf("␣aborted]\n");
209                         return;
210                 }
211                 printf("␣␣␣␣%s␣(%d␣bytes)\n", phdr->name, f_len);
212                 while (bytes_left) {
213                         int iobytes = min(chunk, bytes_left);
214                         read(fd, buf,
215                              ((iobytes - 1) / SECTOR_SIZE + 1) * SECTOR_SIZE);
216                         write(fdout, buf, iobytes);
217                         bytes_left -= iobytes;
```

```
218                 }
219                 close(fdout);
220         }
221
222         close(fd);
223
224         printf("␣done]\n");
225 }
226
227 /*****************************************************************************
228  *                                Init
229  *****************************************************************************/
230 /**
231  * The hen.
232  *
233  *****************************************************************************/
234 void Init()
235 {
236         int fd_stdin  = open("/dev_tty0", O_RDWR);
237         assert(fd_stdin  == 0);
238         int fd_stdout = open("/dev_tty0", O_RDWR);
239         assert(fd_stdout == 1);
240
241         printf("Init()␣is␣running␣...\n");
242
243         /* extract 'cmd.tar' */
244         untar("/cmd.tar");
...
266 }
```

之所以我一直没有提到 tar 文件的格式，是因为一看代码10.22你就能明白。打成 tar 包的过程，其实就是在每个文件的前面加一个 512 字节的文件头，并把所有文件叠放在一起。所以解包的过程，就是读取文件头，根据文件头里记录的文件大小读出文件，然后是下一个文件头和下一个文件，如此循环。tar 文件的文件头定义是从 GNU tar 的源代码中借过来的，我们只用到其中两项：name 和size。需要注意的是size一项存放的不是个整数，而是个字符串，而且用的是八进制，这需要我们读出来之后转换一下。

好了，该出锅了，我们运行一下，见图10.4。

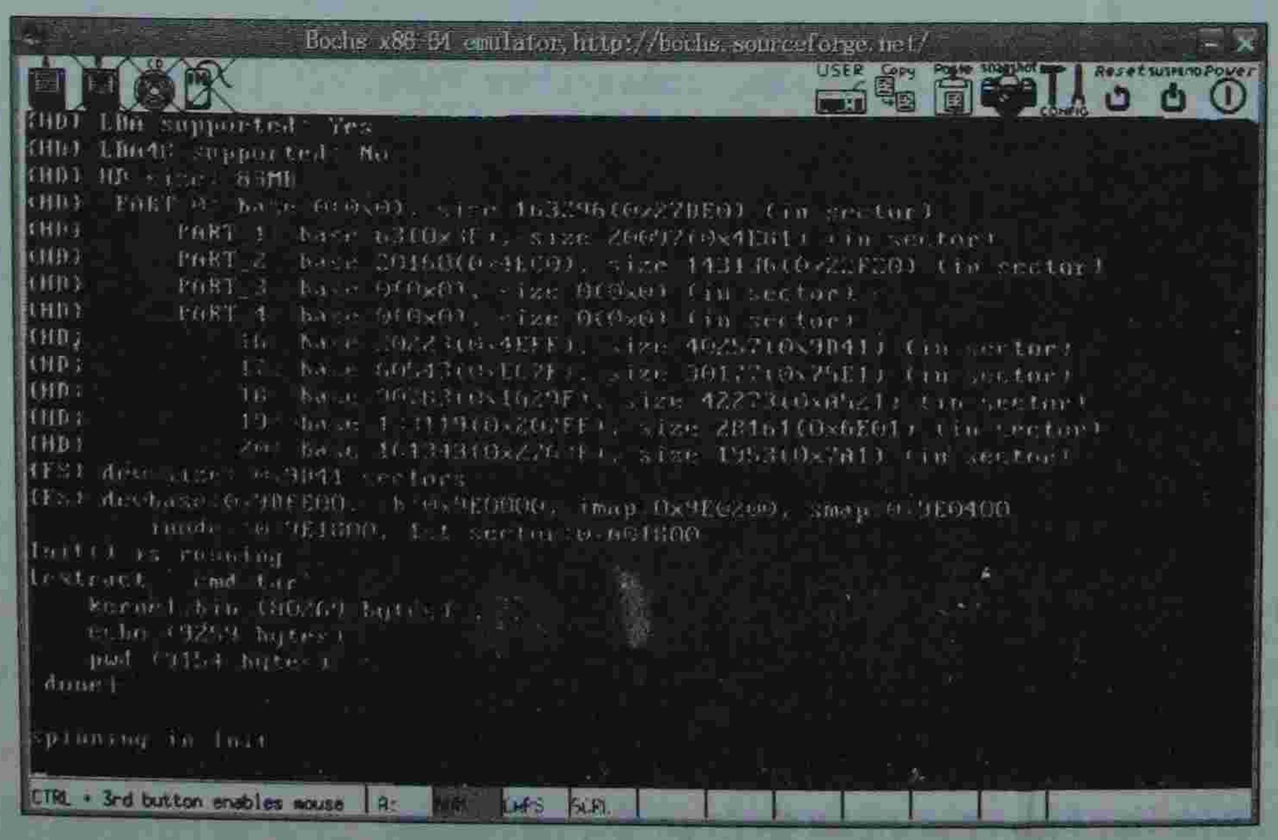


图 10.4 安装应用程序

我们得到了如下输出：

```
[extract '/cmd.tar'
    kernel.bin (80269 bytes)
    echo (9259 bytes)
    pwd (9154 bytes)
done]
```

看上去一切良好，通过直接观察磁盘映像，我们可以看到如下情景：

```
▷ xxd -u -a -g 1 -c 16 -s 0xA01800 -l 512 80m.img
0a01800: 01 00 00 00 2E 00 00 00 00 00 00 00 00 00 00 00  ................
0a01810: 02 00 00 00 64 65 76 5F 74 74 79 30 00 00 00 00  ....dev_tty0....
0a01820: 03 00 00 00 64 65 76 5F 74 74 79 31 00 00 00 00  ....dev_tty1....
0a01830: 04 00 00 00 64 65 76 5F 74 74 79 32 00 00 00 00  ....dev_tty2....
0a01840: 05 00 00 00 63 6D 64 2E 74 61 72 00 00 00 00 00  ....cmd.tar.....
0a01850: 06 00 00 00 6B 65 72 6E 65 6C 2E 62 69 6E 00 00  ....kernel.bin..
0a01860: 07 00 00 00 65 63 68 6F 00 00 00 00 00 00 00 00  ....echo........
0a01870: 08 00 00 00 70 77 64 00 00 00 00 00 00 00 00 00  ....pwd.........
0a01880: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
0a019f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

一眼就看到，我们的文件系统中多了三个文件。细心又耐心的读者可以通过inode等数据结构的值来手工验证文件写入的正确性，此处不再赘述。

10.3.4 实现exec

应用程序已经齐备，终于可以写一个exec()了，我们还是先完成库函数，但这次跟之前不同，因为exec()通常有若干变体，比如你查看exec的manpage，会看到这样的解释：

```
NAME
       execl, execlp, execle, execv, execvp - execute a file

SYNOPSIS
       #include <unistd.h>

       extern char **environ;

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

一阵眩晕，眼花缭乱。不要怕，这些命名是有规律的，其中：

- l表示函数直接接收来自命令行的参数（可变参数）
- p表示使用 PATH 环境变量来寻找要执行的文件
- e表示指向环境变量的指针直接传递给了子进程
- v表示以字符指针的方式传递来自命令行的参数

在这里我们不去实现所有的变体，只选其中的execl()和execv()来实现，见代码10.23。

代码 10.23 exec()（chapter10/d/lib/exec.c）

```
47  /*****************************************************************************
48   *                                execl
49   *****************************************************************************/
50  PUBLIC int execl(const char *path, const char *arg, ...)
51  {
52          va_list parg = (va_list)(&arg);
53          char **p = (char**)parg;
54          return execv(path, p);
55  }
56
57  /*****************************************************************************
58   *                                execv
59   *****************************************************************************/
60  PUBLIC int execv(const char *path, char * argv[])
61  {
62          char **p = argv;
63          char arg_stack[PROC_ORIGIN_STACK];
64          int stack_len = 0;
65
66          while(*p++) {
67                  assert(stack_len + 2 * sizeof(char*) < PROC_ORIGIN_STACK);
68                  stack_len += sizeof(char*);
69          }
70
71          *((int*)(&arg_stack[stack_len])) = 0;
72          stack_len += sizeof(char*);
73
74          char ** q = (char**)arg_stack;
75          for (p = argv; *p != 0; p++) {
76                  *q++ = &arg_stack[stack_len];
77
78                  assert(stack_len + strlen(*p) + 1 < PROC_ORIGIN_STACK);
79                  strcpy(&arg_stack[stack_len], *p);
80                  stack_len += strlen(*p);
81                  arg_stack[stack_len] = 0;
82                  stack_len++;
83          }
84
85          MESSAGE msg;
86          msg.type        = EXEC;
87          msg.PATHNAME    = (void*)path;
88          msg.NAME_LEN    = strlen(path);
89          msg.BUF         = (void*)arg_stack;
90          msg.BUF_LEN     = stack_len;
91
92          send_recv(BOTH, TASK_MM, &msg);
93          assert(msg.type == SYSCALL_RET);
94
95          return msg.RETVAL;
96  }
```

execl()最终调用execv()。而execv()所做的其实只是一件事，那就是向MM提供最终供调用exec的进程使用的堆栈。我们知道，main()函数接受两个参数：argc和argv，其中的argv看上去像个细绳，实际上另一端拴着一头牛呢。通过一个argv，我们可以得到用户输入的所有参数，我们来复习一下这个过程，请看图10.5。

这个图描述的是echo的main()函数执行前堆栈的情形（假设我们输入了命令“echo hello world”）。argv是个指向指针数组的指针，图中argv的值为0xC000，这是个地址。在0xC000处是个指针数组，也就是说，数组内放着另外一些指针。图中可以看到，数组内有0xC010，0xC015和0xC01B三

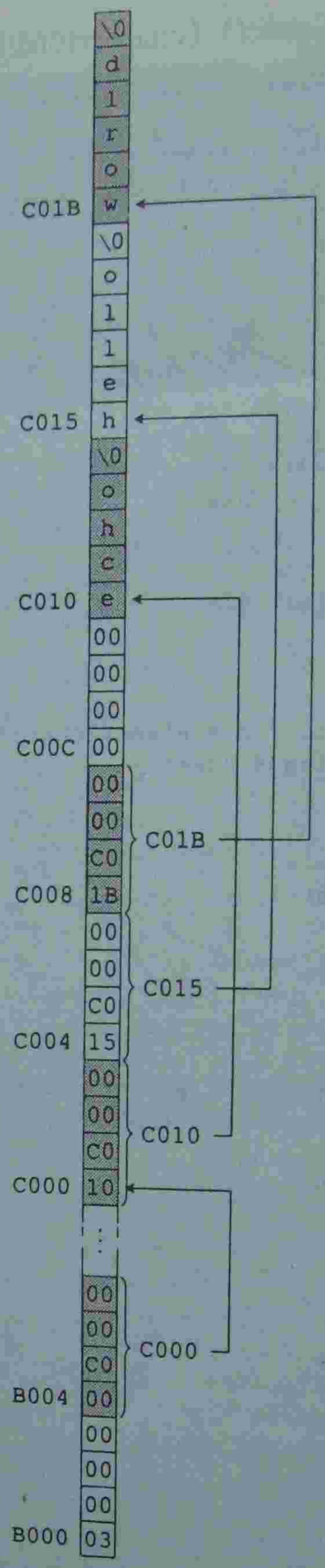


图 10.5 argv

个指针。这三个指针分别指向三个字符串：“echo”，“hello”和“world”。这就是 echo 的main()函数开始执行时内存应该有的样子。换句话说，在将控制权交给main()之前，MM 应该先将内存准备成如图10.5所示的样子。

execv()完成的就是这么一件事情，它先准备好一块内存arg_stack[]，然后完成以下工作：

- 遍历其调用者（比如execl()）传递给自己的参数，数一数参数的个数（第68行）

- 将指针数组的末尾赋零（第71行）
- 遍历所有字符串
  - 将字符串复制到arg_stack[]中（第79行）
  - 将每个字符串的地址写入指针数组的正确位置（第76行）

这项工作做完之后，它将arg_stack[]的首地址以及其中有效内容的长度等内容通过消息发送给MM，MM就可以进行实际的exec操作了。

我们下面在MM中用do_exec()来处理EXEC消息，见代码10.24。

代码10.24 do_exec()（chapter10/d/mm/exec.c）

```
25  /*****************************************************************************
26   *                                do_exec
27   *****************************************************************************/
28  /**
29   * Perform the exec() system call.
30   *
31   * @return  Zero if successful, otherwise -1.
32   *****************************************************************************/
33  PUBLIC int do_exec()
34  {
35          /* get parameters from the message */
36          int name_len = mm_msg.NAME_LEN; /* length of filename */
37          int src = mm_msg.source;        /* caller proc nr. */
38          assert(name_len < MAX_PATH);
39
40          char pathname[MAX_PATH];
41          phys_copy((void*)va2la(TASK_MM, pathname),
42                    (void*)va2la(src, mm_msg.PATHNAME),
43                    name_len);
44          pathname[name_len] = 0; /* terminate the string */
45
46          /* get the file size */
47          struct stat s;
48          int ret = stat(pathname, &s);
49          if (ret != 0) {
50                  printl("{MM} MM::do_exec()::stat() returns error. %s", pathname);
51                  return -1;
52          }
53
54          /* read the file */
55          int fd = open(pathname, O_RDWR);
56          if (fd == -1)
57                  return -1;
58          assert(s.st_size < MMBUF_SIZE);
59          read(fd, mmbuf, s.st_size);
60          close(fd);
61
62          /* overwrite the current proc image with the new one */
63          Elf32_Ehdr* elf_hdr = (Elf32_Ehdr*)(mmbuf);
64          int i;
65          for (i = 0; i < elf_hdr->e_phnum; i++) {
66                  Elf32_Phdr* prog_hdr = (Elf32_Phdr*)(mmbuf + elf_hdr->e_phoff +
67                                                       (i * elf_hdr->e_phentsize));
68                  if (prog_hdr->p_type == PT_LOAD) {
69                          assert(prog_hdr->p_vaddr + prog_hdr->p_memsz <
70                                 PROC_IMAGE_SIZE_DEFAULT);
71                          phys_copy((void*)va2la(src, (void*)prog_hdr->p_vaddr),
72                                    (void*)va2la(TASK_MM,
73                                                 mmbuf + prog_hdr->p_offset),
74                                    prog_hdr->p_filesz);
75                  }
```

```
76      }
77
78      /* setup the arg stack */
79      int orig_stack_len = mm_msg.BUF_LEN;
80      char stackcopy[PROC_ORIGIN_STACK];
81      phys_copy((void*)va2la(TASK_MM, stackcopy),
82                (void*)va2la(src, mm_msg.BUF),
83                orig_stack_len);
84
85      u8 * orig_stack = (u8*)(PROC_IMAGE_SIZE_DEFAULT - PROC_ORIGIN_STACK);
86
87      int delta = (int)orig_stack - (int)mm_msg.BUF;
88
89      int argc = 0;
90      if (orig_stack_len) {    /* has args */
91            char **q = (char**)stackcopy;
92            for (; *q != 0; q++,argc++)
93                  *q += delta;
94      }
95
96      phys_copy((void*)va2la(src, orig_stack),
97                (void*)va2la(TASK_MM, stackcopy),
98                orig_stack_len);
99
100     proc_table[src].regs.ecx = argc; /* argc */
101     proc_table[src].regs.eax = (u32)orig_stack; /* argv */
102
103     /* setup eip & esp */
104     proc_table[src].regs.eip = elf_hdr->e_entry; /* @see _start.asm */
105     proc_table[src].regs.esp = PROC_IMAGE_SIZE_DEFAULT - PROC_ORIGIN_STACK;
106
107     strcpy(proc_table[src].name, pathname);
108
109     return 0;
110 }
```

代码分为九部分：

1. 从消息体中获取各种参数。由于调用者和 MM 处在不同的地址空间，所以对于文件名这样的一段内存，需要通过获取其物理地址并进行物理地址复制。
2. 通过一个新的系统调用`stat()`[6]获取被执行文件的大小。
3. 将被执行文件全部读入 MM 自己的缓冲区（MM 的缓冲区有 1MB，我们姑且假设这个空间足够了。等有一天真的不够了，会触发一个`assert`，到时我们再做打算）。
4. 根据 ELF 文件的程序头（Program Header）信息，将被执行文件的各个段放置到合适的位置。
5. 建立参数栈——这个栈在`execv()`中已经准备好了，但由于内存空间发生了变化，所以里面所有的指针都需要重新定位，这个过程并不难，通过一个`delta`变量即可完成（第93行）。
6. 为被执行程序的`eax`和`ecx`赋值——还记得`_start`中我们将`eax`和`ecx`压栈吗？压的就是`argv`和`argc`。它们是在这里被赋值的（第100行和第 101 行）。

[6] `stat()`的作用是得到文件的信息，在这里我们主要关心其大小。这个系统调用实现起来非常容易，将传入的结构体填充好返回就行了。涉及的主要代码位于fs/misc.c中。

7. 为程序的 eip 赋值，这是程序的入口地址，即 _start 处。

8. 为程序的 esp 赋值。注意要闪出刚才我们准备好的堆栈的位置。

9. 最后是将进程的名字改成被执行程序的名字。

哇哈，不知道你感觉怎么样，反正我是激动得不行了，因为我们的第一个应用程序就要开始执行了，编译，运行！（见图10.6）

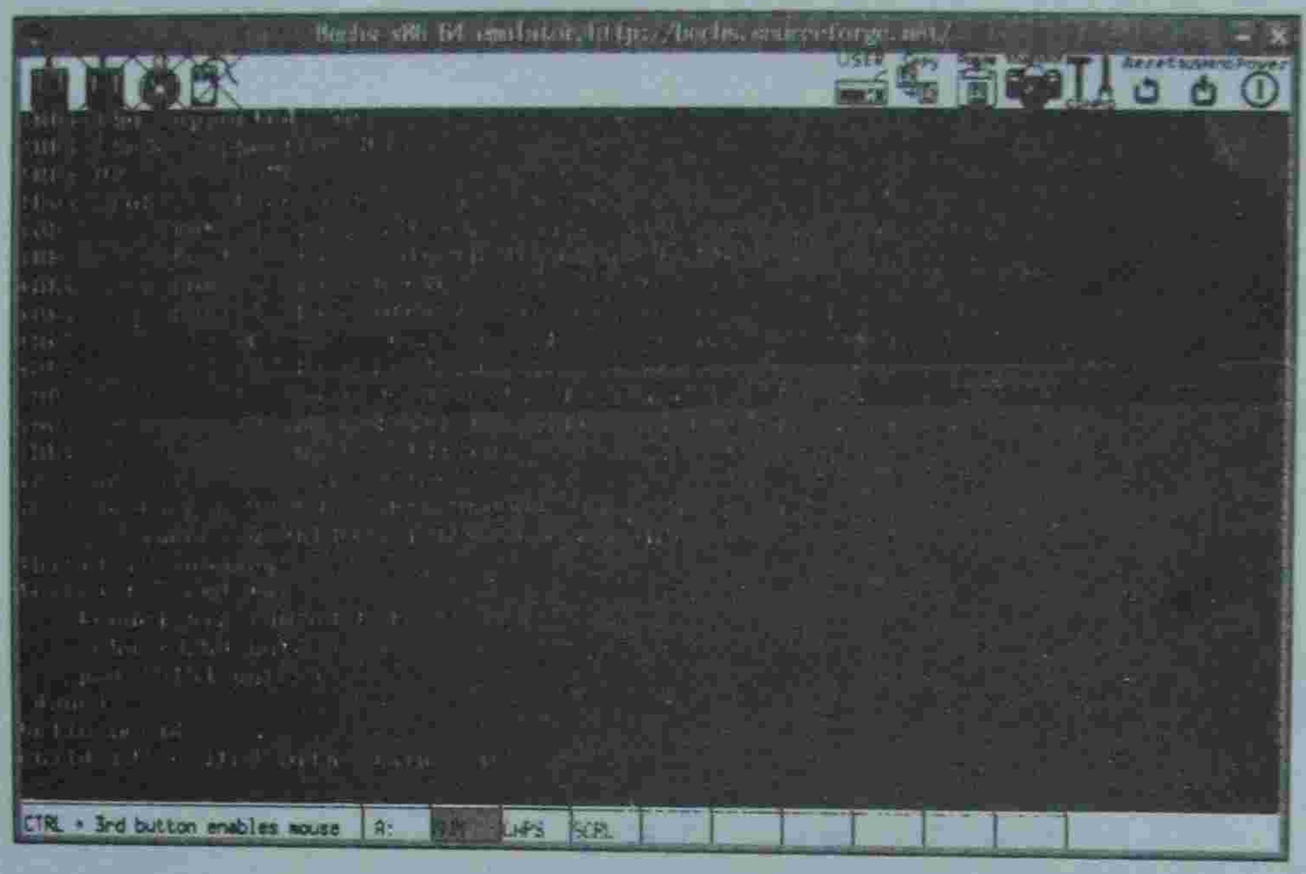


图 10.6 echo

“hello world”出现了！虽然你平生会写出无数个hello world，但有哪一次可以跟现在这个相媲美呢？这是我们在自己操作系统上用自己的应用程序打印的hello world！尽情享受这一时刻吧！

10.4 简单的 shell

有了fork()和exec()，我们的 shell 终于可以开始写了。shell 可以很复杂，但目前我们只实现一种功能，那就是读取命令并执行之（如果命令存在的话）。

这个简单的 shell 可以这么实现：

代码 10.25 shabby_shell（chapter10/e/kernel/main.c）

```
227 /*****************************************************************************
228  *                                shabby_shell
229  *****************************************************************************/
230 /**
231  * A very very simple shell.
232  *
233  * @param tty_name  TTY file name.
234  *****************************************************************************/
235 void shabby_shell(const char * tty_name)
236 {
237         int fd_stdin  = open(tty_name, O_RDWR);
238         assert(fd_stdin  == 0);
239         int fd_stdout = open(tty_name, O_RDWR);
240         assert(fd_stdout == 1);
```

```
        char rdbuf[128];

        while (1) {
                write(1, "$␣", 2);
                int r = read(0, rdbuf, 70);
                rdbuf[r] = 0;

                int argc = 0;
                char * argv[PROC_ORIGIN_STACK];
                char * p = rdbuf;
                char * s;
                int word = 0;
                char ch;
                do {
                        ch = *p;
                        if (*p != '␣' && *p != 0 && !word) {
                                s = p;
                                word = 1;
                        }
                        if ((*p == '␣' || *p == 0) && word) {
                                word = 0;
                                argv[argc++] = s;
                                *p = 0;
                        }
                        p++;
                } while(ch);
                argv[argc] = 0;

                int fd = open(argv[0], O_RDWR);
                if (fd == -1) {
                        if (rdbuf[0]) {
                                write(1, "{", 1);
                                write(1, rdbuf, r);
                                write(1, "}\n", 2);
                        }
                }
                else {
                        close(fd);
                        int pid = fork();
                        if (pid != 0) { /* parent */
                                int s;
                                wait(&s);
                        }
                        else {  /* child */
                                execv(argv[0], argv);
                        }
                }
        }

        close(1);
        close(0);
}

/*****************************************************************************
 *                                Init
 *****************************************************************************/
/**
 * The hen.
 *
 *****************************************************************************/
void Init()
{
        int fd_stdin  = open("/dev_tty0", O_RDWR);
        assert(fd_stdin  == 0);
        int fd_stdout = open("/dev_tty0", O_RDWR);
        assert(fd_stdout == 1);
```

```
308         printf("Init() is running ...\n");
309
310         /* extract 'cmd.tar' */
311         untar("/cmd.tar");
312
313
314         char * tty_list[] = {"/dev_tty1", "/dev_tty2"};
315
316         int i;
317         for (i = 0; i < sizeof(tty_list) / sizeof(tty_list[0]); i++) {
318                 int pid = fork();
319                 if (pid != 0) { /* parent process */
320                         printf("[parent is running, child pid:%d]\n", pid);
321                 }
322                 else {  /* child process */
323                         printf("[child is running, pid:%d]\n", getpid());
324                         close(fd_stdin);
325                         close(fd_stdout);
326
327                         shabby_shell(tty_list[i]);
328                         assert(0);
329                 }
330         }
331
332         while (1) {
333                 int s;
334                 int child = wait(&s);
335                 printf("child (%d) exited with status: %d.\n", child, s);
336         }
337
338         assert(0);
339 }
340
```

我们用 Init 进程启动两个 shell，分别运行在 TTY1 和 TTY2 上。两个 shell 都是 Init 进程的子进程，同时它们也将生成自己的子进程。由于它实在很简陋，我们给它起个贱名，叫shabby_shell。

shabby_shell用read()读取用户输入，然后fork出一个子进程，在子进程中将输入交给execv()来执行。如果用户的输入并不是一个合法的命令，那么shabby_shell只是将命令回显出来，不做其他任何处理。

让我们来尝试着用一下这个shabby_shell，请看图10.7和图10.8。

在图10.7中，我们使用了一个 echo 命令和一个不合法的命令。在图10.8中，我们还使用了 pwd 命令。在我们的系统中，pwd 可以永远打印“/”，因为我们的文件系统是扁平的。因此 pwd 是个比 echo 还要简单的程序。读者从现在开始可以为自己的操作系统写应用程序了，比如 ls、rm 等命令，都比较容易实现，您不妨现在就试试。

想想真不可思议，我们居然有了自己的 shell，可以执行自己的应用程序！眼看着我们的试验品已经越来越像是个能用的操作系统了！

10.5 总结

本章的名字叫做“内存管理”，但其实我们并没有涉及太多“管理”的事情，我们只是围绕如何通过实现fork()、exit()、wait()以及exec()等系

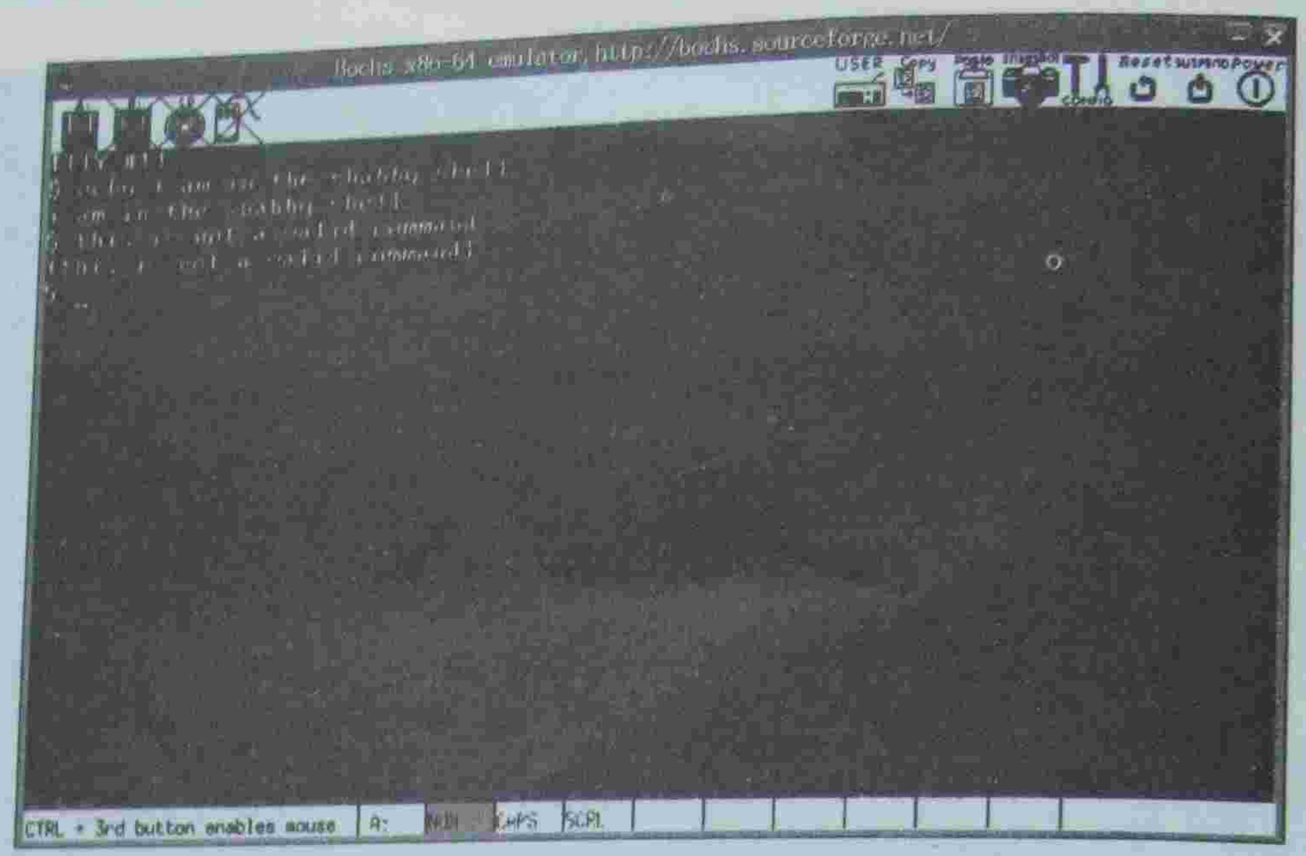


图 10.7 Shabby Shell

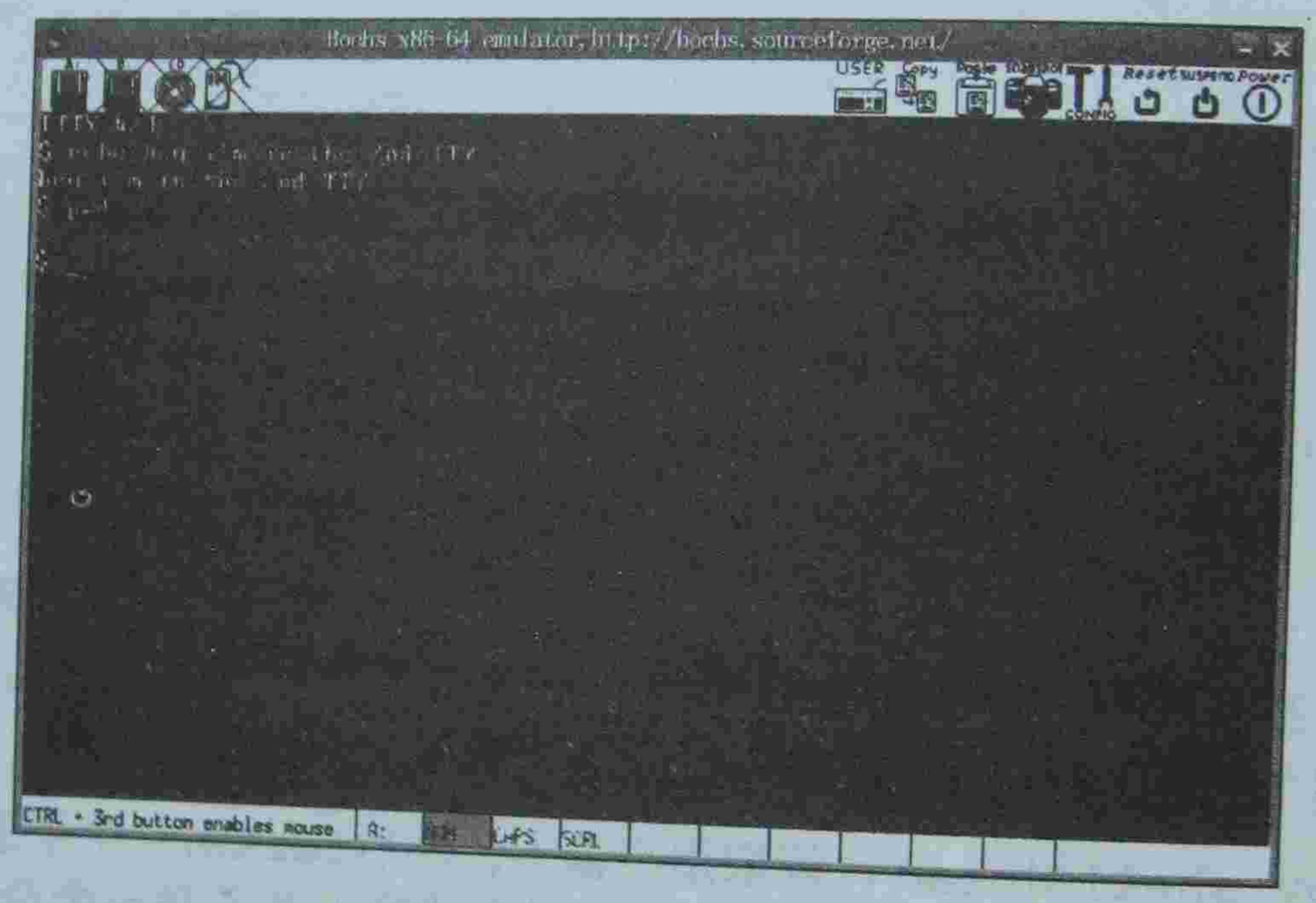


图 10.8 Shabby Shell

统调用来实现一个简单的 shell，并用它来执行自己的应用程序。不过尽管如此，内存管理的框架我们却已经建立起来了。不需要太多努力，我们就可以进一步实现诸如`brk()`这样的系统调用，从而进一步让用户进程可以使用`malloc()`，逐步地，内存管理就可以完善起来了。

从有到好，从简单到完善，从来都比从无到有要困难得多，而且这是个无止境的过程，所以本章到此就不再多说。我相信有兴趣的读者可以以本章所实现的 MM 为参考，写出漂亮得多的内存管理模块。如此本章的目的也就算达到了。

You often don't really understand the problem until after the first time you implement a solution. The second time, maybe you know enough to do it right. So if you want to get it right, be ready to start over at least once.

—— *Eric S. Raymond*

11 尾声

我相信读者能读这本书，并且读到这里的话，多半是出于一种热爱。换句话说，觉得操作系统“挺好玩的”。如果是这样，我希望你永远不失掉这份“have fun”之心。在本章中，就让我们停下赶路，本着这份好玩之心，鼓捣一点前面没来得及玩的东西。

11.1 让mkfs()只执行一次

目前我们的系统每次启动都是“全新”的，因为每一次`init_fs()`都会调用`mkfs()`刷新硬盘，一定程度上这比较利于我们调试——每次启动时可保证文件系统是一样的，但它也存在明显的坏处，那就是上次建立的文件到下一次启动时就不见了。我们下面就来改变这一现状，见代码11.1。

代码 11.1 让 mkfs() 只执行一次（chapter11/a/fs/main.c）

```
134  PRIVATE void init_fs()
135  {
      ...
151          /* open the device: hard disk */
152          MESSAGE driver_msg;
153          driver_msg.type = DEV_OPEN;
154          driver_msg.DEVICE = MINOR(ROOT_DEV);
155          assert(dd_map[MAJOR(ROOT_DEV)].driver_nr != INVALID_DRIVER);
156          send_recv(BOTH, dd_map[MAJOR(ROOT_DEV)].driver_nr, &driver_msg);
157
158          /* read the super block of ROOT DEVICE */
⇒ 159          RD_SECT(ROOT_DEV, 1);
160
161          sb = (struct super_block *)fsbuf;
⇒ 162          if (sb->magic != MAGIC_V1) {
163                  printl("{FS} mkfs\n");
164                  mkfs(); /* make FS */
165          }
      ...
174  }
```

很简单，只需要每次先读取超级块，如果发现了魔数（Magic Number），则认为分区已经“装上Orange'S了”，否则调用`mkfs()`。

接下来你会发现，系统启动时不会每次都mkfs()了，但每次还是会执行一次解开cmd.tar的操作，无论是不是上次启动时解开过。我们这就来改一下untar()，让它解包之后就在cmd.tar这文件中留个记号，下次看到记号，就不再傻傻地解包了。见代码11.2。

代码 11.2 修改 untar()（chapter11/a/kernel/main.c）

```
181 void untar(const char * filename)
182 {
183         printf("[extract␣'%s'\n", filename);
184         int fd = open(filename, O_RDWR);
185         assert(fd != -1);
186
187         char buf[SECTOR_SIZE * 16];
188         int chunk = sizeof(buf);
189         int i = 0;
190         int bytes = 0;
191
192         while (1) {
193                 bytes = read(fd, buf, SECTOR_SIZE);
194                 assert(bytes == SECTOR_SIZE); /* size of a TAR file
195                                                * must be multiple of 512
196                                                */
197
198                 if (buf[0] == 0) {
199                         if (i == 0)
200                                 printf("␣␣␣␣need␣not␣unpack␣the␣file.\n");
201                         break;
202                 }
203                 i++;
204
205                 struct posix_tar_header * phdr = (struct posix_tar_header *)buf;
206
207                 /* calculate the file size */
208                 char * p = phdr->size;
209                 int f_len = 0;
210                 while (*p)
211                         f_len = (f_len * 8) + (*p++ - '0'); /* octal */
212
213                 int bytes_left = f_len;
214                 int fdout = open(phdr->name, O_CREAT | O_RDWR | O_TRUNC);
215                 if (fdout == -1) {
216                         printf("␣␣␣␣failed␣to␣extract␣file:␣%s\n", phdr->name);
217                         printf("␣aborted]\n");
218                         close(fd);
219                         return;
220                 }
221                 printf("␣␣␣␣%s\n", phdr->name);
222                 while (bytes_left) {
223                         int iobytes = min(chunk, bytes_left);
224                         read(fd, buf,
225                              ((iobytes - 1) / SECTOR_SIZE + 1) * SECTOR_SIZE);
226                         bytes = write(fdout, buf, iobytes);
227                         assert(bytes == iobytes);
228                         bytes_left -= iobytes;
229                 }
230                 close(fdout);
231         }
232
233         if (i) {
234                 lseek(fd, 0, SEEK_SET);
235                 buf[0] = 0;
236                 bytes = write(fd, buf, 1);
237                 assert(bytes == 1);
238         }
239
         close(fd);
```

```
240         printf("␣done,␣%d␣files␣extracted]\n", i);
241
242 }
```

这里增加了一个变量i，表示从cmd.tar中总共解出来多少文件。每次成功的解包操作之后（这时i必大于0），我们将cmd.tar的第一个字节置为零（第234行），这样下一次untar()执行到第197行时会发现第一个字节为零，于是退出。

如果我们增加或者改写了应用程序，通常会使用dd重新将cmd.tar写入磁盘。由于TAR文件的开始处是包含其中的文件的文件名，所以第一个字节必不为零，于是再次启动时，untar()发现第一个字节非零，从而进入解包的步骤。

值得注意的一点是再一次解包时，很可能包内包含的文件已经在磁盘上存在了，所以我们需要将原来的文件内容清除，然后写入新内容，这需要引入O_TRUNC，加入到open()的参数中，见第213行。

引入O_TRUNC后，我们还要修改文件系统中的do_open()，见代码11.3[1]。

代码11.3 修改do_open()（chapter11/a/fs/open.c）

```
33  /*****************************************************************************
34   *                                do_open
35   *****************************************************************************/
36  /**
37   * Open a file and return the file descriptor.
38   *
39   * @return File descriptor if successful, otherwise a negative error code.
40   *****************************************************************************/
41  PUBLIC int do_open()
42  {
    ...
75          int inode_nr = search_file(pathname);
76
77          struct inode * pin = 0;
78
79          if (inode_nr == INVALID_INODE) { /* file not exists */
80                  if (flags & O_CREAT) {
81                          pin = create_file(pathname, flags);
82                  }
83                  else {
84                          printl("{FS}␣file␣not␣exists:␣%s\n", pathname);
85                          return -1;
86                  }
87          }
88          else if (flags & O_RDWR) { /* file exists */
89                  if ((flags & O_CREAT) && (!(flags & O_TRUNC))) {
90                          assert(flags == (O_RDWR | O_CREAT));
91                          printl("{FS}␣file␣exists:␣%s\n", pathname);
92                          return -1;
93                  }
94                  assert((flags ==  O_RDWR                          ) ||
95                         (flags == (O_RDWR | O_TRUNC                )) ||
96                         (flags == (O_RDWR | O_TRUNC | O_CREAT)));
97
98                  char filename[MAX_PATH];
99                  struct inode * dir_inode;
100                 if (strip_path(filename, pathname, &dir_inode) != 0)
101                         return -1;
```

[1]为了简化操作，这里采用了与Linux稍有不同的规则。读者如果有兴趣（或者不满意笔者的做法），完全可以改造do_open()使之符合POSIX标准。

```
102             pin = get_inode(dir_inode->i_dev, inode_nr);
103         }
104         else { /* file exists, no O_RDWR flag */
105             printl("{FS} file exists: %s\n", pathname);
106             return -1;
107         }
108
109         if (flags & O_TRUNC) {
110             assert(pin);
⇒ 111            pin->i_size = 0;
112             sync_inode(pin);
113         }
114
115         if (pin) {
...
148         }
149
150         return fd;
151 }
```

引入O_TRUNC之后do_open()的逻辑复杂了若干，大致可以描述为：

- 如果文件不存在，则只要有O_CREAT就成功，没有就失败。
- 如果文件存在，则判断是否有O_RDWR，若没有则失败，若有的话，则分以下情况：
  - 仅有O_RDWR，成功
  - 有O_CREAT但无O_TRUNC，失败
  - 有O_TRUNC（无论有没有O_CREAT），成功

这样再次运行时，mkfs()和解开cmd.tar的操作就省掉了，见图11.1。

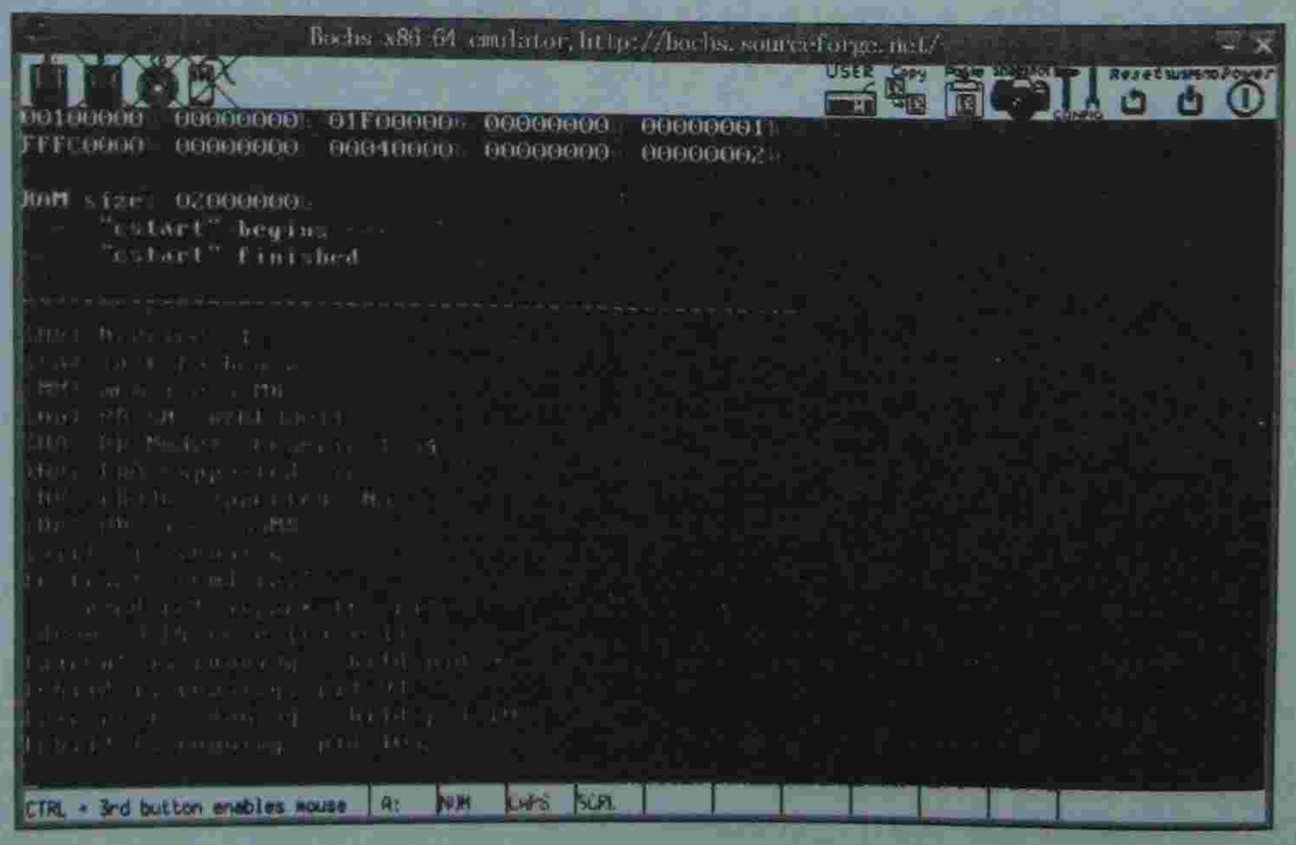


图 11.1 省掉 mkfs() 和 untar()

11.2 从硬盘引导

虽然我们的硬盘上已经有不少内容了，但到目前为止，我们的系统始终是从软盘启动的。本节中我们将把Orange'S安装到硬盘上，并实现硬盘启动。

我们先回忆一下从软盘启动的过程：

1. BIOS将引导扇区读入内存0000:7c00处；
2. 跳转到0000:7c00处开始执行引导代码；
3. 引导代码从软盘中找到loader.bin，并将其读入内存；
4. 跳转到loader.bin开始执行；
5. loader.bin从软盘中找到kernel.bin，并将其读入内存；
6. 跳转到kernel.bin开始执行，到此可认为启动过程结束；
7. 系统运行中。

在第1步中，BIOS到底读软盘还是硬盘是由CMOS设置决定的，通常你可以找到一个叫做“Boot Sequence”的选项，从中选择首选启动设备。在第3步和第5步中，对于软盘启动，代码将在软盘中寻找loader.bin和kernel.bin，对于硬盘启动，我们需要让引导扇区代码从硬盘中寻找loader.bin并让loader从硬盘中寻找kernel.bin。这便是软盘和硬盘启动的区别了。剩下的几步中，软盘和硬盘启动没有分别。

因此我们需要重写boot.asm和loader.asm，让它们读取硬盘而不是软盘。新的文件我们起名为hdboot.asm和hdldr.asm。

11.2.1 编写硬盘引导扇区和硬盘版loader

我们先来完成hdboot.asm。它跟boot.asm的区别将主要在两方面，一是读取软盘和硬盘扇区的方法有所不同；二是软盘的文件系统（FAT12）跟硬盘的文件系统（Orange'S FS）不同，所以寻找loader的方式肯定也不一样。

我们在第4章中讲到如何用int 13h读取软盘扇区（表4.4），实际上这个中断也可用来读取硬盘，见表11.1。

表11.1 用BIOS中断int 13h（ah=02h）读取硬盘

中断号	寄存器		作用
13h	ah: 02h	al: 要读扇区数	从磁盘读数据入es:bx指向的缓冲区中
	ch: 柱面号低八位	cl: 第0~5位:起始扇区号 第6~7位:柱面号高二位	
	dh: 磁头号	dl: 驱动器号（置第7位表示硬盘操作）	

根据这个表我们知道，以这种方式读取磁盘，允许的磁头数最大为 256（2^8），柱面数最大为 1024（2^{10}），扇区数最大为 64（2^6），所以容易算出所支持的硬盘最多有 16777216（256×1024×64）个扇区，合 8GB。这实在是个太严重的限制了，因为目前的硬盘很少小于 8GB，显然这个方法读取硬盘不太好。

好在我们并不是最先不满于这个限制的一批人（还好呵呵），West Digital 和 Phoenix Technologies 联合推出了 EDD 标准（BIOS Enhanced Disk Drive Services），它支持 64 位 LBA，啊，我们都松了一口气，这个好多了。它的具体做法是将原来放进寄存器中的参数放进内存中的数据结构，这个数据结构叫做 Disk Address Packet，见表11.2。

表 11.2 Disk Address Packet

偏移	位数	描述
0	8	Packet 有多少字节
1	8	保留
2	8	要传输的块数（最大值为 127；0 表示不传输数据）
3	8	保留
4	32	读操作的目的地址（段:偏移）
8	64	LBA 地址

当读磁盘扇区时，只需要将AH设为42h，DL设为驱动器号，DS:SI设为Disk Address Packet的地址，然后调用int 13h就可以了。

了解了读取硬盘的方法，我们就可以开始写hdboot.asm了，请看代码11.4。

代码 11.4 chapter11/b/boot/hdboot.asm

```
; ==========================================================================
;                                  hdboot.asm
; ==========================================================================
;                                                    Forrest Yu, 2008
; ==========================================================================

org  0x7c00                     ; bios always loads boot sector to 0000:7C00

      jmp     boot_start

%include                "load.inc"

STACK_BASE              equ     0x7C00  ; base address of stack when booting
TRANS_SECT_NR           equ     2
SECT_BUF_SIZE           equ     TRANS_SECT_NR * 512

disk_address_packet:    db      0x10                ; [ 0] Packet size in bytes.
                        db      0                   ; [ 1] Reserved, must be 0.
                        db      TRANS_SECT_NR       ; [ 2] Nr of blocks to transfer.
                        db      0                   ; [ 3] Reserved, must be 0.
                        dw      0                   ; [ 4] Addr of transfer - Offset
                        dw      SUPER_BLK_SEG       ; [ 6] buffer.          - Seg
                        dd      0                   ; [ 8] LBA. Low  32-bits.
                        dd      0                   ; [12] LBA. High 32-bits.

err:
        mov     dh, 3                       ; "Error 0  "
        call    disp_str                    ; display the string
        jmp     $
```

```
boot_start:
        mov     ax, cs
        mov     ds, ax
        mov     es, ax
        mov     ss, ax
        mov     sp, STACK_BASE

        call    clear_screen

        mov     dh, 0                   ; "Booting  "
        call    disp_str                ; display the string

        ;; read the super block to SUPER_BLK_SEG::0
        mov     dword [disk_address_packet +  8], ROOT_BASE + 1
        call    read_sector
        mov     ax, SUPER_BLK_SEG
        mov     fs, ax

        mov     dword [disk_address_packet +  4], LOADER_OFF
        mov     dword [disk_address_packet +  6], LOADER_SEG

        ;; get the sector nr of '/' (ROOT_INODE), it'll be stored in eax
        mov     eax, [fs:SB_ROOT_INODE]
        call    get_inode

        ;; read '/' into ex:bx
        mov     dword [disk_address_packet +  8], eax
        call    read_sector

        ;; let's search '/' for the loader
        mov     si, LoaderFileName
        push    bx                      ; <- save
.str_cmp:
        ;; before comparation:
        ;;      es:bx -> dir_entry @ disk
        ;;      ds:si -> filename we want
        add     bx, [fs:SB_DIR_ENT_FNAME_OFF]
.1:
        lodsb                           ; ds:si -> al
        cmp     al, byte [es:bx]
        jz      .2
        jmp     .different              ; oops
.2:                                     ; so far so good
        cmp     al, 0                   ; both arrive at a '\0', match
        jz      .found
        inc     bx                      ; next char @ disk
        jmp     .1                      ; on and on
.different:
        pop     bx                      ; -> restore
        add     bx, [fs:SB_DIR_ENT_SIZE]
        sub     ecx, [fs:SB_DIR_ENT_SIZE]
        jz      .not_found

        mov     dx, SECT_BUF_SIZE
        cmp     bx, dx
        jge     .not_found

        push    bx
        mov     si, LoaderFileName
        jmp     .str_cmp
.not_found:
        mov     dh, 2
        call    disp_str
        jmp     $
.found:
        pop     bx
```

```
98          add     bx, [fs:SB_DIR_ENT_INODE_OFF]
99          mov     eax, [es:bx]                    ; eax <- inode nr of loader
100         call    get_inode                       ; eax <- start sector nr of loader
101         mov     dword [disk_address_packet +  8], eax
102 load_loader:
103         call    read_sector
104         cmp     ecx, SECT_BUF_SIZE
105         jl      .done
106         sub     ecx, SECT_BUF_SIZE      ; bytes_left -= SECT_BUF_SIZE
107         add     word  [disk_address_packet + 4], SECT_BUF_SIZE ; transfer buffer
108         jc      err
109         add     dword [disk_address_packet + 8], TRANS_SECT_NR ; LBA
110         jmp     load_loader
111 .done:
112         mov     dh, 1
113         call    disp_str
114         jmp     LOADER_SEG:LOADER_OFF
115         jmp     $
116
117
118 ;=============================================================================
119 ;字符串
120 ;-----------------------------------------------------------------------------
121 LoaderFileName             db      "hdldr.bin", 0  ; LOADER 之文件名
122 ; 为简化代码, 下面每个字符串的长度均为 MessageLength
123 MessageLength              equ     9
124 BootMessage:               db      "Booting  "; 9字节, 不够则用空格补齐. 序号 0
125 Message1                   db      "HD  Boot  "; 9字节, 不够则用空格补齐. 序号 1
126 Message2                   db      "No  LOADER"; 9字节, 不够则用空格补齐. 序号 2
127 Message3                   db      "Error  0  "; 9字节, 不够则用空格补齐. 序号 3
128 ;=============================================================================
129
130 clear_screen:
131         mov     ax, 0x600               ; AH = 6,  AL = 0
132         mov     bx, 0x700               ; 黑底白字(BL = 0x7)
133         mov     cx, 0                   ; 左上角: (0, 0)
134         mov     dx, 0x184f              ; 右下角: (80, 50)
135         int     0x10                    ; int 0x10
136         ret
137
138 ;-----------------------------------------------------------------------------
139 ; 函数名: disp_str
140 ;-----------------------------------------------------------------------------
141 ; 作用:
142 ;       显示一个字符串, 函数开始时 dh 中应该是字符串序号(0-based)
143 disp_str:
144         mov     ax, MessageLength
145         mul     dh
146         add     ax, BootMessage
147         mov     bp, ax                  ; ┓
148         mov     ax, ds                  ; ┃ ES:BP = 串地址
149         mov     es, ax                  ; ┛
150         mov     cx, MessageLength       ; CX = 串长度
151         mov     ax, 0x1301              ; AH = 0x13,  AL = 0x1
152         mov     bx, 0x7                 ; 页号为0(BH = 0) 黑底白字(BL = 0x7)
153         mov     dl, 0
154         int     0x10                    ; int 0x10
155         ret
156
157 ;-----------------------------------------------------------------------------
158 ; read_sector
159 ;-----------------------------------------------------------------------------
160 ; Entry:
161 ;       - fields disk_address_packet should have been filled
162 ;         before invoking the routine
163 ; Exit:
164 ;       - es:bx -> data read
```

```
; registers changed:
;       - eax, ebx, dl, si, es
;
read_sector:
        xor     ebx, ebx

        mov     ah, 0x42
        mov     dl, 0x80
        mov     si, disk_address_packet
        int     0x13

        mov     ax, [disk_address_packet + 6]
        mov     es, ax
        mov     bx, [disk_address_packet + 4]

        ret

;-------------------------------------------------------------------------
; get_inode
;-------------------------------------------------------------------------
; Entry:
;       - eax     : inode nr.
; Exit:
;       - eax     : sector nr.
;       - ecx     : the_inode.i_size
;       - es:ebx  : inodes sector buffer
; registers changed:
;       - eax, ebx, ecx, edx
get_inode:
        dec     eax                             ; eax <-  inode_nr -1
        mov     bl, [fs:SB_INODE_SIZE]
        mul     bl                              ; eax <- (inode_nr - 1) * INODE_SIZE
        mov     edx, SECT_BUF_SIZE
        sub     edx, dword [fs:SB_INODE_SIZE]
        cmp     eax, edx
        jg      err
        push    eax

        mov     ebx, [fs:SB_NR_IMAP_SECTS]
        mov     edx, [fs:SB_NR_SMAP_SECTS]
        lea     eax, [ebx+edx+ROOT_BASE+2]
        mov     dword [disk_address_packet +  8], eax
        call    read_sector

        pop     eax                             ; [es:ebx+eax] -> the inode

        mov     edx, dword [fs:SB_INODE_ISIZE_OFF]
        add     edx, ebx
        add     edx, eax                        ; [es:edx] -> the_inode.i_size
        mov     ecx, [es:edx]                   ; ecx <- the_inode.i_size

        ; es:[ebx+eax] -> the_inode.i_start_sect
        add     ax, word [fs:SB_INODE_START_OFF]

        add     bx, ax
        mov     eax, [es:bx]
        add     eax, ROOT_BASE                  ; eax <- the_inode.i_start_sect
        ret


times   510-($-$$) db 0 ; 填充剩下的空间，使生成的二进制代码恰好为512字节
dw      0xaa55          ; 结束标志
```

代码的主干部分逻辑还是比较清晰的，所做工作依次是读取超级块，读取根目录，从根目录中寻找hdldr.bin，将hdldr.bin读入内存并将控制权交给它。在整个过程中有两个函数被调用多次：是read_sector和get_inode。

read_sector默认disk_address_packet已被填充完毕，所以它只是为AH和DL赋值，然后调用int 13h，这样指定扇区就被读入到 Disk Address Packet 指定的内存了。为使用方便，read_sector之后让es:bx指向刚刚读入的内存地址，这样调用者通过es:bx访问就可以了。

举例说明，在第45行之前，Disk Address Packet 中已经准备好了读操作的目的地址和要读取的扇区数，所以第45行我们给 LBA 赋值之后，就可以调用read_sector了（第46行），这样超级块就被读入到SUPER_BLK_SEG:0处。接下来第47行和第48行让fs段指向SUPER_BLK_SEG，这样我们就可以通过fs方便地访问超级块了。比如第54行就从超级块中读出根 inode 号，从而用get_inode得到根目录的首扇区号。

get_inode用来获取给定 i-node 对应的文件的首扇区号。它的输入是 i-node 号（保存在eax中），输出有二，分别是文件的首扇区号（保存在eax中）和文件长度（保存在ecx中）。

对于 Disk Address Packet 这个结构，在第22行我们将数据传输目的段地址设为SUPER_BLK_SEG，因为我们首先要读取超级块。SUPER_BLK_SEG在load.inc中被设为0x70，也就是说超级块将被读到0x700~0x8FF处。之后在第50行和第51行将数据传输的目的地址设成了LOADER_SEG:LOADER_OFF。从这之后除 LBA 之外，Disk Address Packet 结构的其余各项基本不再改变，也就是说，在调用read_sector之前，只需要关注 LBA 就可以了。

注意LOADER_SEG:LOADER_OFF这块内存被复用多次。在第55行和第100行调用get_inode时，这块内存用于保存inode_array中的扇区内容。而在第59行调用read_sector时，这块内存用于保存根目录内容。只有到了第103行，LOADER_SEG:LOADER_OFF这块内存才真正“名副其实”地用来保存hdldr.bin的文件内容。

此外请读者留意以下几个方面：

1. ROOT_BASE以宏的方式定义，所以如果更换磁盘或者分区，hdboot.asm需要重新编译。之所以这样做，是因为对于某个特定的系统，这个值是固定的，而且通过宏来“指定”它，而不是通过读取分区表来“计算”它，将大大简化hdboot.asm的代码。

2. 代码中用到了若干的宏，比如SB_INODE_SIZE、SB_NR_IMAP_SECTS等，它们需要和fs.h中super_block结构相一致。

3. read_sector中，寄存器DL总是被赋值为0x80，也就是说，这个引导扇区默认我们的操作系统安装在第一块硬盘上。

4. 并不是所有的 i-node 都能被get_inode支持。inode_array可能很长，但我们只读取最开始的一段，具体由TRANS_SECT_NR决定。也就是说，如果所请求的 i-node 位于inode_array的前TRANS_SECT_NR个扇区之外，get_inode将提示错误。这样做是有理由的，因为get_inode只用于得到“/”和“hdldr.bin”两个文件的 i-node，而“/”通常占用第一个 i-node，而hdldr.bin在系统被安装时就理应存在于文件系统之内，所以它的 i-node 也应该是很靠前的，因此get_inode不会出错。

5. 跟第4条的情况类似，对根目录的遍历也是有限的，但由于hdldr.bin理应位于目录的前部，所以只要hdldr.bin存在，它就应该能被找到。

以上便是硬盘引导扇区的情况了。接下来我们可以修改loader.asm，形成一个hdldr.asm是很容易的事情了。需要改的只是读取kernel.bin的部分，而这一部分跟hdboot.asm中读取hdldr.bin的代码基本一致，所以不再赘述。

11.2.2 “安装”hdboot.bin 和 hdldr.bin

下面我们将hdboot.asm和hdldr.asm编译成hdboot.bin和hdldr.bin，然后将它们“安装”到系统中。

安装hdboot.bin也就是将它放入硬盘引导扇区中，这可以使用 dd 命令：

```
▷ dd if=boot/hdboot.bin of=80m.img bs=1 count=446 conv=notrunc
▷ dd if=boot/hdboot.bin of=80m.img seek=510 skip=510 bs=1 count=2 conv=notrunc
```

请注意这里使用了两个 dd 命令，这是为了不至于覆盖掉原有的分区表——如果存在的话。

安装hdldr.bin看上去不如hdboot.bin那么直接，因为它将以普通文件的身份存在于Orange'S FS。不要担心，我们可以将它打进cmd.tar这个包，然后用软盘启动一下，它就老老实实地被解压到文件系统中了。

注意若想从硬盘启动，之前的一次软盘启动是免不了的。软盘启动主要做两件事情：

- 通过mkfs()将硬盘的相应分区做成Orange'S FS
- 将cmd.tar解开，这时 FS 中就有hdldr.bin和kernel.bin了（kernel.bin早在上一章就被打包了）

安装好了引导扇区和 loader，系统就可以启动了（别忘了先将bochsrc中的启动设备改成硬盘），见图11.2。

成功了！而且从打印的字符串可以明白地看到，这时我们的hdboot.bin和hdldr.bin在运行了。

11.2.3 grub

系统已经可以在硬盘上运行了，不过我们仍然可以改进一下，将引导扇区安装到Orange'S分区的引导扇区，而不是整块硬盘的引导扇区，这样Orange'S就可以跟硬盘上其他操作系统和平共处了。做到这一点其实很容易，只需要安装一个 grub 就可以了。

我们先将引导扇区装到Orange'S分区的引导扇区：

```
▷ dd if=boot/hdboot.bin of=80m.img seek=`echo "obase=10;ibase=16;\`egrep
␣-e␣'^ROOT_BASE'␣boot/include/load.inc␣|␣sed␣-e␣'s/.*0x//g'\`"200" | bc`
 bs=1 count=446 conv=notrunc
▷ dd if=boot/hdboot.bin of=80m.img seek=`echo "obase=10;ibase=16;\`egrep
␣-e␣'^ROOT_BASE'␣boot/include/load.inc␣|␣sed␣-e␣'s/.*0x//g'\`"200+1FE" | bc`
 skip=510 bs=1 count=2 conv=notrunc
```

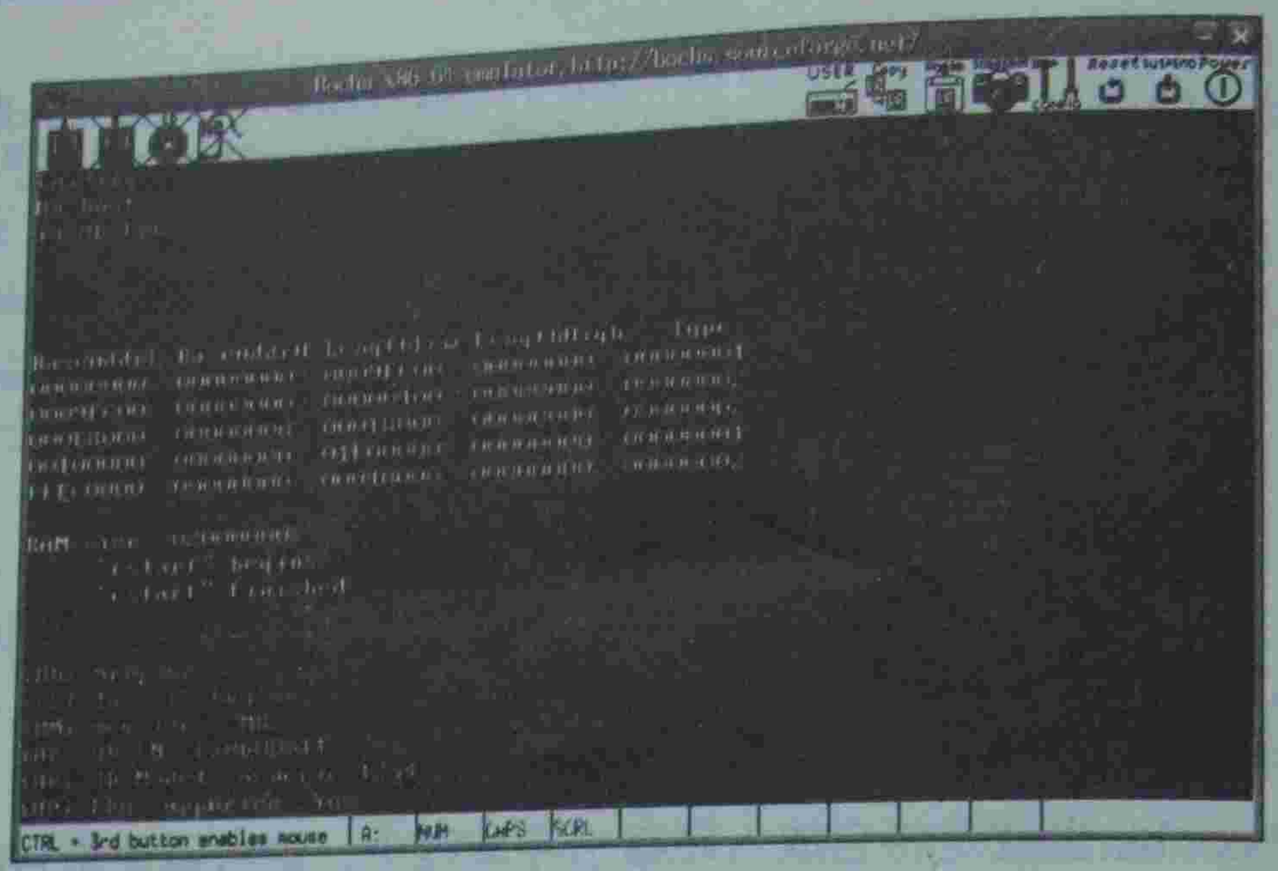


图 11.2 从硬盘启动

其中 dd 命令的 seek 参数的值仍然是用一个组合命令从load.inc中得到。

接下来该 grub 上场了。如果硬盘上已经有一个 Linux 发行版，那么在它当初安装时很可能已经装上 grub 了。如果是块“干净”硬盘，我们可以这么做：

- 从 grub 官方网站[2]下载 grub 的源代码（我们以 0.97 版本为例）
- 进入源代码目录，编译：

```
▷ ./configure
▷ make
```

- 取出其中的两个文件：grub-0.97/stage1/stage1 和 grub-0.97/stage2/stage2。
- 将 stage1 和 stage2 写入磁盘映像：

```
▷ dd if=stage1 of=80m.img bs=1 count=446 conv=notrunc
▷ dd if=stage2 of=80m.img bs=512 seek=1 conv=notrunc
```

也就是说，我们并没有完全安装 grub，只是使用它的 stage1 和 stage2，有了它们，多重引导就可以实现了，请看图11.3。

安装了 grub 的 stage1 和 stage2 之后，启动时会出现 grub 提示符，这时我们输入三个命令：

```
grub> rootnoverify (hd0,4)
grub> chainloader +1
grub> boot
```

boot之后敲一个回车，我们的 OS 就启动起来了，很容易不是吗？

下面简单介绍一下 grub 中输入的三个命令的意义。

“rootnoverify”意为将指定分区作为根分区，但不试图挂载（mount）它。我们知道，grub 可以用来启动 Linux，并且可以指定启动哪个内核，要做到

[2]http://www.gnu.org/software/grub/。

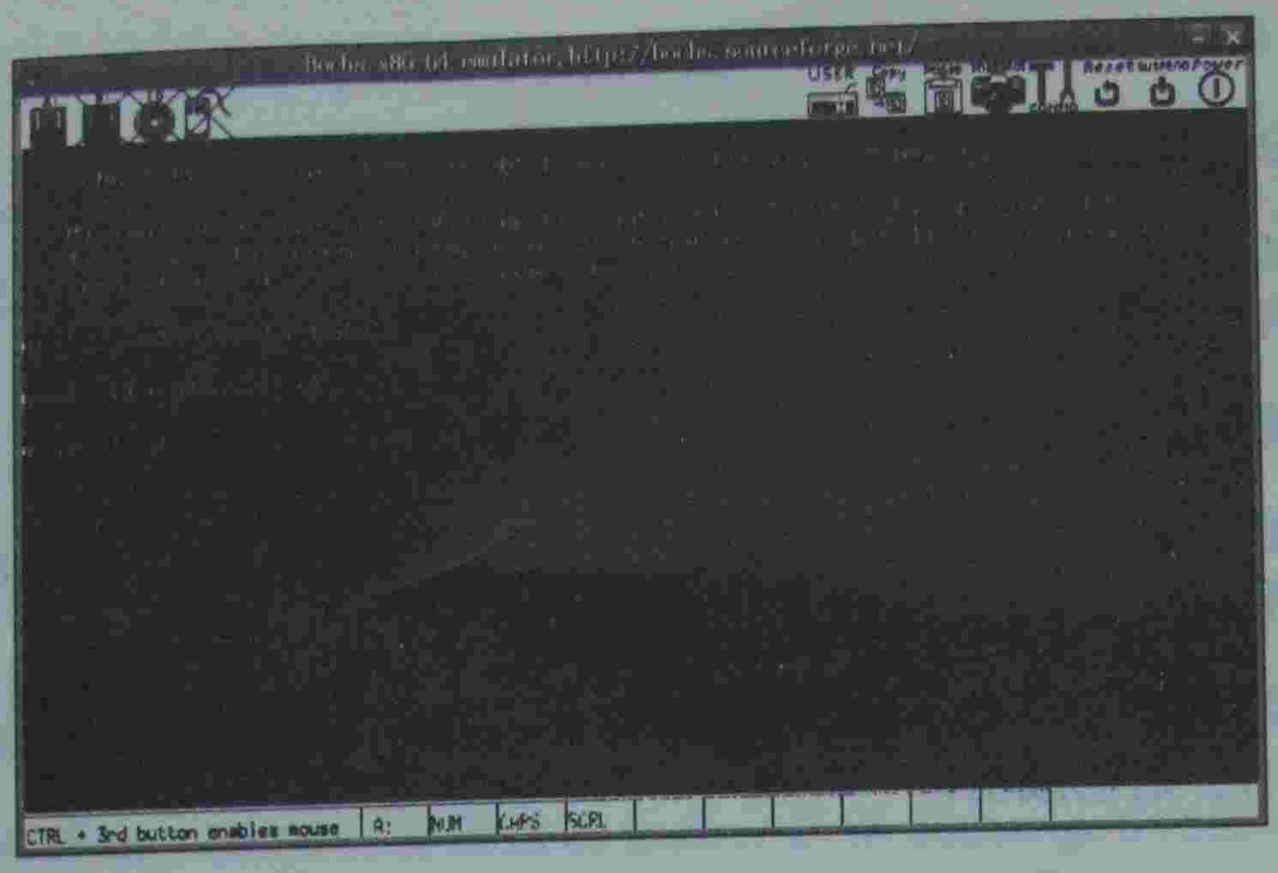


图 11.3 grub

这一点，grub 显然应该是可以识别存放内核文件的文件系统的。这就是root命令试图去挂载文件系统的原因。然而，grub 并不认识我们的文件系统——至少目前如此，所以我们用一个rootnoverify来告诉 grub，不要试图挂载它，只需要将分区作为根就好了。

需要注意一点，grub 对硬盘分区的编号跟 Linux 下的规则有所不同，它是从零开始编号的。hd[0,1,2,3] 表示四个主分区，hd4 表示第一个逻辑分区——这正是我们的根分区。

“chainloader +1”会把刚刚指定的根分区的引导扇区加载到0x7c00处，也就是说，这一命令完成了之前 BIOS 完成的工作。

“boot”的作用显而易见，它将控制权交给刚刚读入的引导扇区，于是系统就归我们管了。

上述命令的更详细解释读者可参考 grub 的官方文档。

11.2.4 小结

我们现在可以从硬盘引导自己的操作系统了，不过由于需要做的工作有点细碎，我们在这里总结一下，将我们的操作系统安装到某个新硬盘的某个新分区需要哪些步骤：

1. 使用 fdisk 等工具查看硬盘的分区情况，确定要装到哪个分区，记下分区的首扇区扇区号。
2. 根据我们在第 9 章中提到的规则，算出分区的次设备号。
3. 将boot/include/load.inc中的ROOT_BASE设置为Orange'S分区的首扇区的扇区号。
4. 将include/sys/config.h中的MINOR_BOOT设置为Orange'S分区的次设备号。
5. 如果使用 Bochs 的话，确认 bochsrc 中设置了正确的磁盘映像文件。

6. 用 dd 等工具将hdboot.bin写入引导扇区。
7. 将hdldr.bin和kernel.bin打包入cmd.tar。
8. 将cmd.tar用 dd 命令写入硬盘（注意位置一定要再三确认，以免破坏硬盘中已有的数据）。
9. 如果硬盘上没有 grub 的话，用 dd 命令写入 stage1 和 stage2。
10. 从软盘启动，这时`mkfs()`会将硬盘的相应分区做成Orange'S FS格式，并且会将cmd.tar解开，这时 FS 中就有hdldr.bin和kernel.bin了。
11. 从硬盘启动，待出现grub提示符时，输入命令（注意确认`rootnoverify`的参数），启动成功。

比如，我们可以新建一个 100MB 磁盘映像，起名为 100m.img，并用 fdisk 分区如下（Start 和 End 的单位均为扇区）：

```
   Device Boot      Start         End      Blocks   Id  System
100m.img1               63       20159      10048+  83  Linux
100m.img2           20160      204623      92232    5  Extended
100m.img5           20223       40896      10337   83  Linux
100m.img6           40960       61376      10208+  83  Linux
100m.img7           61440      163776      51168+  99  Unknown
100m.img8          163840      204623      20392   83  Linux
```

我们根据刚才的步骤来做一下：

1. 我们打算把Orange'S装到 100m.img7 这个分区中，其首扇区扇区号为61440，即`0xF000`。
2. 根据图9.9可知，分区 100m.img7 可表示为 hd2c，其次设备号为`0x23`。
3. 将boot/include/load.inc中的`ROOT_BASE`设置为`0xF000`。
4. 将include/sys/config.h中的`MINOR_BOOT`设置为`MINOR_hd2c`。
5. 将 bochsrc 中磁盘映像文件名改为 100m.img，并将 ata0-master 一行改为：

```
ata0-master: type=disk, path="100m.img", mode=flat, cylinders=203,
heads=16, spt=63
```

6. 用 dd 等工具将hdboot.bin写入引导扇区：

```
▷ dd if=boot/hdboot.bin of=100m.img seek=`echo "obase=10;ibase=16;\`
egrep␣-e␣'^ROOT_BASE'␣boot/include/load.inc␣|␣sed␣-e␣'s/.*0x//g'\`*200"
| bc` bs=1 count=446 conv=notrunc
▷ dd if=boot/hdboot.bin of=100m.img seek=`echo "obase=10;ibase=16;\`
egrep␣-e␣'^ROOT_BASE'␣boot/include/load.inc␣|␣sed␣-e␣'s/.*0x//g'\`
*200+1FE" | bc` skip=510 bs=1 count=2 conv=notrunc
```

7. 将hdldr.bin和kernel.bin打包入cmd.tar：

```
▷ tar vcf inst.tar kernel.bin echo pwd hdldr.bin
kernel.bin
echo
pwd
hdldr.bin
```

8. 将cmd.tar用 dd 命令写入硬盘：

```
▷ dd if=inst.tar of=../100m.img seek=`echo "obase=10;ibase=16;(\`
egrep␣-e␣'^ROOT_BASE'␣../boot/include/load.inc␣|␣sed␣-e␣'s/.*0x//g'\`+\`
egrep␣-e␣'#define[[:space:]]*INSTALL_START_SECT'␣../include/sys/config.h
|␣sed␣-e␣'s/.*0x//g'\`)*200" | bc` bs=1 count=`ls -l inst.tar | awk -F
"␣" '{print $5}'` conv=notrunc
122880+0 records in
122880+0 records out
122880 bytes (123 kB) copied, 0.378139 seconds, 325 kB/s
```

9. 由于硬盘上没有 grub，我们用 dd 命令写入 stage1 和 stage2：

```
▷ dd if=stage1 of=100m.img bs=1 count=446 conv=notrunc
▷ dd if=stage2 of=100m.img bs=512 seek=1 conv=notrunc
```

10. 从软盘启动，mkfs()将硬盘的相应分区做成Orange'S FS格式，并且会将cmd.tar解开，这时 FS 中就有hdldr.bin和kernel.bin了。

11. 从硬盘启动，待出现grub提示符时，输入命令（注意确认rootnoverify的参数）：

```
grub> rootnoverify (hd0,6)
grub> chainloader +1
grub> boot
```

启动成功。

由于 hd2c 这个分区比原来我们一直使用的分区大，所以我们可以将头文件include/config.h中的INSTALL_START_SECT修改为0x17000，让cmd.tar尽量靠近分区的末端。

11.3 将 OS 安装到真实的计算机

不知道你有没有这样的念头，将操作系统安装到真实的计算机。反正我是从写引导扇区那一天就在琢磨这档子事儿了。那么这一章中我们就来完成这一“壮举”！

其实安装到真实的硬盘和安装到硬盘映像操作起来是差不多的，因为在 Linux 下面硬盘也是文件。下面我们就来具体操作一下。

11.3.1 准备工作

在开始之前，请做好以下准备：

- 一台计算机。请再三确认这台计算机的硬盘内不包含重要数据，因为接下来的操作有可能对硬盘的数据造成破坏！
- 一张软盘。
- 一张 Linux 某个发行版的安装盘。
- 一颗认真仔细且强壮的心脏。

我们的整个安装过程将分三部分：

1. 安装 Linux（包含为硬盘分区）。
2. 在 Linux 中编译我们的源代码。
3. 将我们的操作系统安装到某个分区。

虽然读者可能平时就在使用 Linux，或者在硬盘中的某个分区安装了 Linux，但笔者还是不建议你用“正在使用的”硬盘来进行下面的操作，毕竟它是有风险的。总之请使用一块空白的硬盘，或者一块包含无关紧要数据的硬盘。

笔者自己试验时，用的是一块有点老旧的 10GB 的希捷硬盘。

11.3.2 安装 Linux

无论读者喜欢哪一个发行版，在进行本次安装时都可以选择尽量少的组件，只要能保证可以正常编译我们的源代码就可以了。

笔者自己习惯使用 Debian，所以在硬盘中安装了一个 Debian Etch，并安装了 gcc 等编译所需的工具。

11.3.3 编译源代码

这里的操作过程跟第463页中提到的步骤基本是一致的。笔者自己的操作过程是这样的：

1. 首先使用 fdisk 查看硬盘的分区情况：

```
Disk /dev/hda: 10.2 GB, 10242892800 bytes
255 heads, 63 sectors/track, 1245 cylinders, total 20005650 sectors
Units = sectors of 1 * 512 = 512 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/hda1   *          63    13671314     6835626   83  Linux
/dev/hda2       13671315    20000924     3164805    5  Extended
/dev/hda5   *   13671378    14651279      489951   99  Unknown
/dev/hda6       14651343    16209584      779121   83  Linux
/dev/hda7       16209648    18169514      979933+  83  Linux
/dev/hda8       18169578    20000924      915673+  82  Linux swap/Solaris
```

   我们把Orange'S安装到/dev/hda5中，它的开始扇区号是 13671378，即十六进制0xD09BD2。与此同时，根据第 9 章中提到的规则，可知分区的次设备号为0x21。

2. 将boot/include/load.inc中的ROOT_BASE设置为0xD09BD2。
3. 将include/sys/config.h中的MINOR_BOOT设置为MINOR_hd2a。

然后是编译，过程略。

11.3.4 开始安装

下面是正式的安装过程了，再次强调，一定要小心谨慎，不要破坏了重要的数据。笔者自己的操作过程如下，供读者参考：

1. 用 dd 将hdboot.bin写入 hda5 的引导扇区。

```
▷ sudo dd if=boot/hdboot.bin of=/dev/hda5 bs=1 count=446 conv=notrunc
446+0 records in
446+0 records out
446 bytes (446 B) copied, 0.083676 seconds, 5.3 kB/s
▷ sudo dd if=boot/hdboot.bin of=/dev/hda5 seek=510 skip=510 bs=1
count=2 conv=notrunc
2+0 records in
2+0 records out
2 bytes (2 B) copied, 0.0744931 seconds, 0.0 kB/s
```

2. 将hdldr.bin和kernel.bin打包入cmd.tar。

```
▷ tar vcf inst.tar kernel.bin ls touch echo pwd test01 hdldr.bin
kernel.bin
ls
touch
echo
pwd
test01
hdldr.bin
```

3. 将cmd.tar用 dd 命令写入硬盘。

```
▷ sudo dd if=inst.tar of=/dev/hda5 seek=`echo "obase=10;ibase=16;(\'
egrep␣-e␣'#define[[:space:]]*INSTALL_START_SECT'␣../include/sys/
config.h␣|␣sed␣-e␣'s/.*0x//g'\`)*200" | bc` bs=1 count=`ls -l
inst.tar | awk -F "␣" '{print $5}'` conv=notrunc
163840+0 records in
163840+0 records out
163840 bytes (164 kB) copied, 0.589778 seconds, 278 kB/s
```

4. 在安装 Debian 时，grub 已经装上了，所以不再需要安装 grub，不过我们可以修改/boot/grub/menu.lst，加入如下几行：

```
title           Orange'S
rootnoverify    (hd0,4)
chainloader     +1
```

这样启动时就不必输入命令了，直接选择Orange'S一行即可。

好了，安装完毕，开始启动。先从软盘启动，之后重启，就可以从硬盘启动了，待出现 grub 画面时，选择Orange'S，回车，系统就启动起来了。图11.4是在 TTY1 中使用 echo 命令的图片，图11.5是运行了一个山寨版的 ls。

真是太棒了！用真实的计算机运行自己的操作系统，是不是比用虚拟机还要令人激动呢？

11.4 总结

在本章中，我们基本上没有鼓捣什么新鲜玩意，不过是将原来在软盘上的东西拿到硬盘上而已。而且在此基础上，我们还把Orange'S安装到了真实的计算

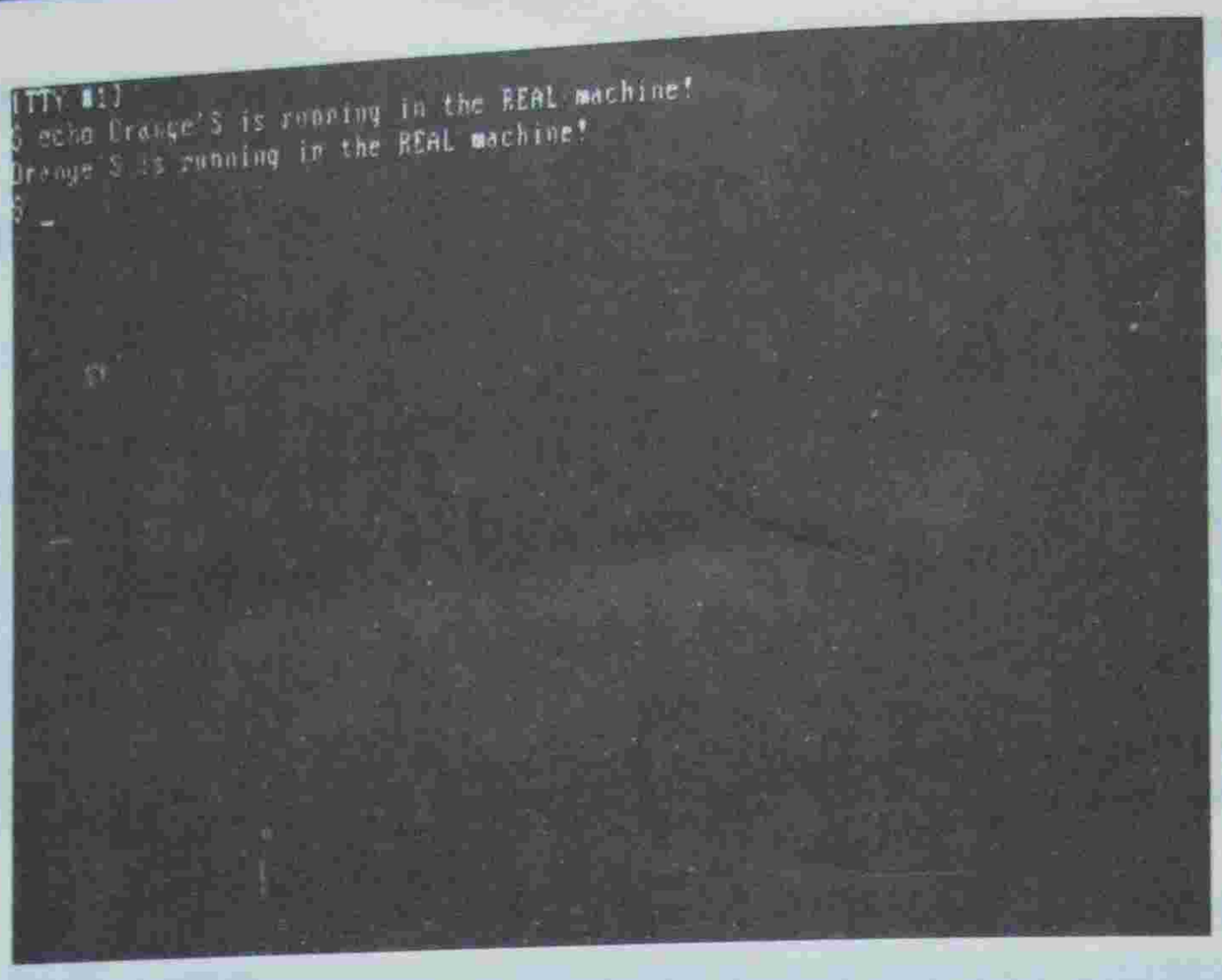


图 11.4 在真实的机器中使用 echo 命令

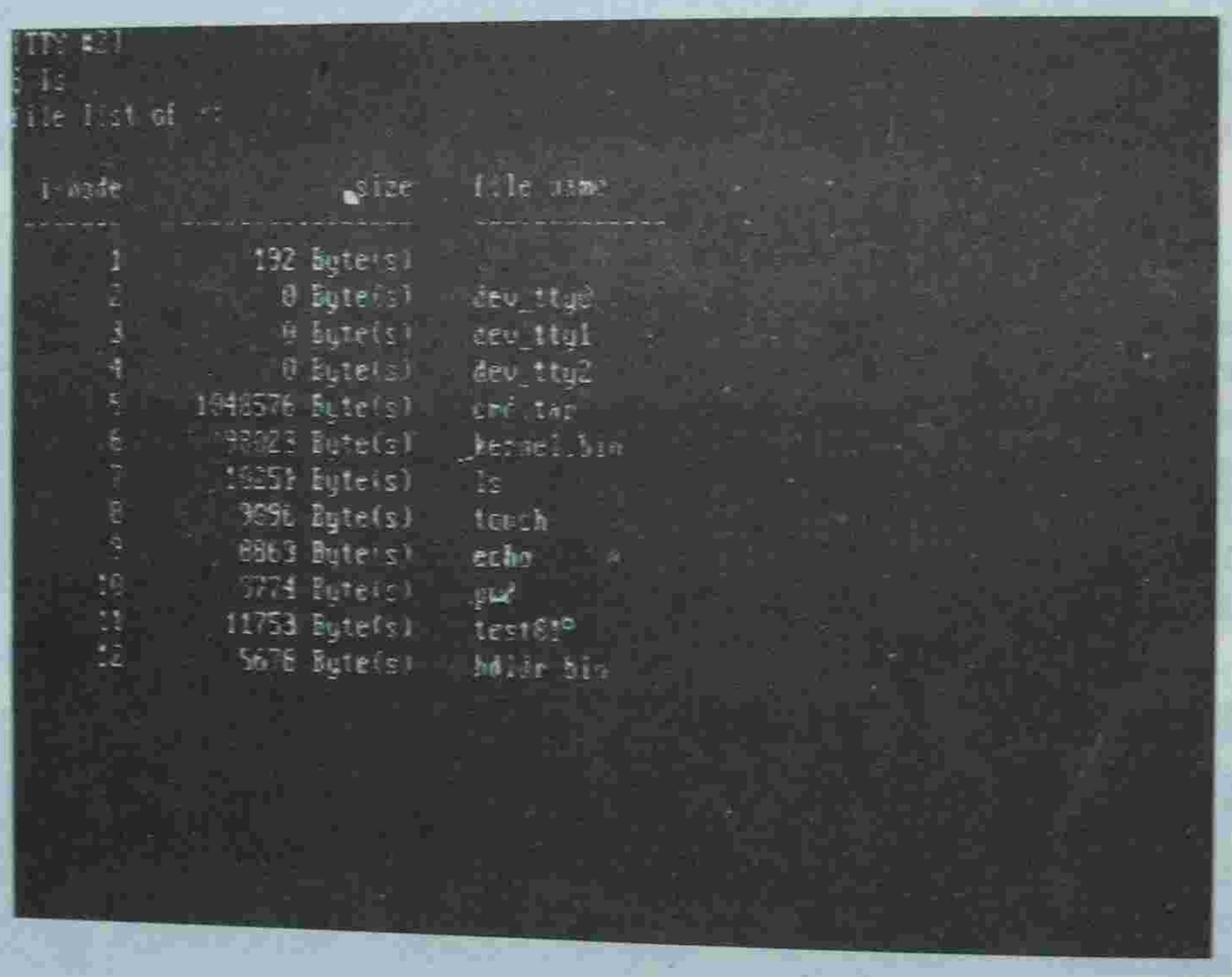


图 11.5 在真实的机器中使用 ls 命令

机上过了一把瘾。我们写自己的操作系统是出于一种好奇，或者说一种求知欲。我希望这样不停地“过把瘾”能让这种好奇不停地延续。

我们刚开始只不过是好奇为什么一按电源计算机就启动起来，出现一堆字符。后来我们了解了，然后又好奇进程如何运行。后来进程也了解了，我们又好奇如何进行键盘输入、如何操作硬盘、如何建立自己的文件系统……

如今，或许你又开始好奇，为什么 Linux 的磁盘 I/O 那么快，而我们的系统却那么慢？也或许，你在好奇，如何才能让我们的系统有一个图形界面呢？或者你在好奇，如何能让我们的 OS 联网呢？

如果你真的好奇，如果你真的还想继续探索，我真诚地希望你再次走进书

店，或者打开网络浏览器，去阅读，去寻找。我相信有这么一份好奇，加上一点点努力，我们一定能从一无所知走向“知道一点点”，然后是“知道一些”，这样慢慢地积累，说不定有一天你突然发现，原来我也可以站在巨人的肩膀上，原来我真的已经站在了巨人的肩膀上。到那时，我们一定已经开始新的旅程了，那将是真正具备创造性的工作。到那一天，请不要忘记，一切都是从好奇开始的，都是从那一个引导扇区开始的。

参考文献

[1] Intel Inc.: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture (2006).

[2] Intel Inc.: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference (2006).

[3] Intel Inc.: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide, Part 1&2 (2006).

[4] [美]Andrew S. Tanenbaum, Albert S. Woodhull 著，王鹏、尤晋元、朱鹏、敖青云译: 操作系统：设计与实现（第二版）. 电子工业出版社 (2001).

[5] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, A Silberschatz: Operating System Concepts, 6th Edition.

[6] [美]William Stallings 著，魏迎梅、王涌等译: 操作系统——内核与设计原理（第四版）. 电子工业出版社 (2002).

[7] T13: Information Technology - AT Attachment with Packet Interface - 6 (ATA/ATAPI-6) (2001).

[8] Microsoft: Microsoft Extensible Firmware Initiative FAT 32 File System Specification, FAT: General Overview of On-Disk Format (2000).

[9] 杨季文等编著: 80x86 汇编语言程序设计教程. 清华大学出版社.

[10] 沈美明、温冬婵编著: IBM-PC 汇编语言程序设计. 清华大学出版社 (1996).

[11] 王士元编著: C 高级实用程序设计. 清华大学出版社 (1999).

[12] 赵炯: Linux 内核完全注释. 机械工业出版社 (2004).

[13] 骆耀祖主编: Linux 操作系统分析教程. (2004).

驱网核心技术丛书　精品图书免费试读

《天书夜读——从汇编语言到 Windows 内核编程》
电子工业出版社【书号】9787121073397【出版日期】2008年10月
网络购买地址：http://www.china-pub.com/209258&ref=ps

【内容简介】

本书从基本的 Windows 程序与汇编指令出发，深入浅出地讲解了 Windows 内核的编程、调试、阅读，以及自行探索的方法。读者在使用 C/C++开发 Windows 程序的基础上，将熟练掌握汇编和 C 语言的应用，深入了解 Windows 底层，并掌握阅读 Windows 内核的基本方法，以及 Windows 内核的基本编程方法。

本书适合使用 C/C++在 Windows 上编程的读者，尤其适合希望加深自己技术功底的 Windows 应用程序员、计算机专业的有志于软件开发的大中院校学生；专业的 Windows 内核程序员，亦可从本书得到超越一般内核程序开发的启发。

【书　摘】

3.1.2　算法反汇编阅读技巧

要阅读上面那样的代码，首先把流程控制的代码与数值计算的代码分开是关键，因此前面的练习能起很大的帮助作用。得到数值计算的代码部分后，必须判断输入与输出（一般自然是被读的内部变量为输入，被写的内部变量为输出），然后把它还原成一个 C 语言的表达式。任何一段中间不加任何跳转，连续的 mov 和加减乘除的指令一般都可以还原为一个 C 表达式。当然，这可不是一个轻松的工作。

在这里顺便可以看到，二维数组 a[x][y]，处理等同于一个大小为 a[y]的结构长度为 x 的数组。所以，前面讲到的数组访问的代码非常有价值。基本的方法如下：

```
mov         eax,<我要取的数组元素的下标>
imul        eax,eax,<结构的大小>
mov         ecx,<结构数组开始的地址>
mov         eax,dword ptr [ecx+eax];取得数组元素的内容放到 eax 中
```

访问结构内部变量的时候，最后面的一个指令还会加上一个数字：

```
mov         eax,dword ptr[ecx+eax+0Ch]
```

看到这样的代码，我们应该联想到表达式中含有的数组或结构体。

精彩样章免费试读地址：
http://www.china-pub.com/ureader/product.asp?bookid=209258

《寒江独钓——Windows 内核安全编程》

电子工业出版社　预计出版日期：2009 年 6 月

【内容简介】

本书从 Windows 内核编程出发，全面介绍串口、键盘、磁盘、文件系统、网络等相关的 Windows 内核模块的编程技术，以及基于这些技术的密码保护、防毒引擎、文件加密、网络嗅探、网络防火墙的具体实现。对于驱动编程模型的选择，本书同时兼顾 WDM 与 WDF。

本书适合大专院校计算机系的学生、计算机编程爱好者、普通 Windows 程序员、Windows 内核程序员、信息安全行业的程序员使用。阅读本书，需要读者有 C 语言、数据结构、操作系统和计算机网络的基础知识。

【书　摘】

5.4.4　如何处理发往设备的请求

在设备被创建好了之后，如何处理所有可能发送给设备的请求是需要考虑的下一个问题。在以往的 WDM 开发中，常用的方式是设置这个设备各个请求的分发函数为自己实现的回调函数，并且将特殊的处理放置在这些函数中。例如在这个例子里，可以将所有的读/写请求都实现为去读写内存，这就是最简单的内存盘。上面的处理方式说起来很简单，但是实现时还是需要一些技巧的，一种常用的方式是建立一个或多个队列，将所有发送到这个设备的请求都插入队列中，由另一个线程去处理队列。这是一个典型的生产者—消费者模型，这样做的好处是有了一个小小的缓冲，同时还不用担心由于缓冲带来的同步问题，因为所有的请求都被队列排队了。无独有偶的是，在 WDF 驱动框架中，微软直接提供了这种处理队列，　样就不用开发人员自己去操心如何建立队列，如何设置同步事件，如何在正确的时间销毁队列，这可真是一个造福大众的做法。

为了实现为驱动制作一个处理队列这一目标，在 WDF 驱动框架中需要初始化一个队列配置变量 ioQueueConfig，这个变量会说明队列的各种属性。一种简单的初始化方法是将这个配置变量初始化为默认状态，之后再对一些具有特殊属性的请求注册回调函数，例如为读请求注册回调函数等。在这样的初始化之后再为指定设备建立这个队列，WDF 驱动框架会自动将所有发往这个指定设备的请求都放入这个队列中处理，同时当请求符合感　趣的属性（例如读写操作）时会调用之前注册过的处理函数去处理。对每个设备可以建　多　个队列，但是在本例中不会讨论多个处理队列的情况。另外，在队列中也具有和设备　关　的扩展，下面也会使用到。

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
Broadview
www.broadview.com.cn

《Orange'S：一个操作系统的实现》读者交流区

尊敬的读者：

感谢您选择我们出版的图书，您的支持与信任是我们持续上升的动力。为了使您能通过本书更透彻地了解相关领域，更深入的学习相关技术，我们将特别为您提供一系列后续的服务，包括：

1. 提供本书的修订和升级内容、相关配套资料；
2. 本书作者的见面会信息或网络视频的沟通活动；
3. 相关领域的培训优惠等。

请您抽出宝贵的时间将您的个人信息和需求反馈给我们，以便我们及时与您取得联系。

您可以任意选择以下三种方式与我们联系，我们都将记录和保存您的信息，并给您提供不定期的信息反馈。

1．短信

您只需编写如下短信：B08442+您的需求+您的建议

发送到1066 6666 789（本服务免费，短信资费按照相应电信运营商正常标准收取，无其他信息收费）

为保证我们对您的服务质量，如果您在发送短信24小时后，尚未收到我们的回复信息，请直接拨打电话（010）88254369。

2．电子邮件

您可以发邮件至jsj@phei.com.cn或editor@broadview.com.cn。

3．信件

您可以写信至如下地址：北京万寿路173信箱博文视点，邮编：100036。

如果您选择第2种或第3种方式，您还可以告诉我们更多有关您个人的情况，及您对本书的意见、评论等，内容可以包括：

（1）您的姓名、职业、您关注的领域、您的电话、E-mail地址或通信地址；

（2）您了解新书信息的途径、影响您购买图书的因素；

（3）您对本书的意见、您读过的同领域的图书、您还希望增加的图书、您希望参加的培训等。

如果您在后期想退出读者俱乐部，停止接收后续资讯，只需发送“B08442+退订”至10666666789即可，或者编写邮件“B08442+退订+手机号码+需退订的邮箱地址”发送至邮箱：market@broadview.com.cn 亦可取消该项服务。

同时，我们非常欢迎您为本书撰写书评，将您的切身感受变成文字与广大书友共享。我们将挑选特别优秀的作品转载在我们的网站（www.broadview.com.cn）上，或推荐至CSDN.NET等专业网站上发表，被发表的书评的作者将获得价值50元的博文视点图书奖励。

我们期待您的消息！

博文视点愿与所有爱书的人一起，共同学习，共同进步！

通信地址：北京万寿路 173 信箱　博文视点（100036）　　电话：010-51260888

E-mail：jsj@phei.com.cn，editor@broadview.com.cn

www.phei.com.cn
www.broadview.com.cn

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888

传　　真：（010）88254397

E-mail：　dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

　　电子工业出版社总编办公室

邮　　编：100036