理论联系实践

自己动手写操作系统

于渊 编著
尤晋元 审校

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

Broadview
www.broadview.com.cn

技术凝聚实力 · 专业创新出版

自己动手写操作系统

本书特色：

- 通过实例，一步一步，由浅入深地引领读者用汇编及C语言编写出一个操作系统框架。
- 在深入讲解代码细节与编程思想的同时，详尽而透彻地解析了操作系统的内核原理与运行机制，使读者知其然，更知其所以然。
- 介绍了大量的编程技巧和实践经验，让读者在实际开发中少走弯路，一举中的。
- 非常适合作为操作系统课程的实践参考书，既提高动手能力，又加深理论认识。

光盘包含本书全部代码

图书分类：计算机>操作系统

ISBN 7-121-01577-3

9 787121 015779 >

网上订购：www.dearbook.com.cn
第二书店 · 第一服务

责任编辑：张 毅
责任美编：张子建

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。
ISBN 7-121-01577-3 定价：48.00元（含光盘1张）

TP316
346D

自己动手写操作系统

于 渊 编著
尤晋元 审校

北方工业大学图书馆
00592933

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

SB 386/02

内 容 简 介

本书在详细分析操作系统原理的基础上，用丰富的实例代码，一步一步地指导读者用 C 语言和汇编语言编写出一个具备操作系统基本功能的操作系统框架。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。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。全书共分 7 章。

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

自己动手写操作系统 / 于渊编著. —北京：电子工业 出版社，2005.8

ISBN 7-121-01577-3

Ⅰ. 自… Ⅱ. 于… Ⅲ. 操作系统 Ⅳ. TP316

中国版本图书馆 CIP 数据核字（2005）第 079890 号

责任编辑：张 毅 zhangyi@phei.com.cn
印 刷：北京智力达印刷有限公司
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
经 销：各地新华书店
开 本：787×980 1/16 印张：24.75 字数：511 千字
印 次：2005 年 8 月第 1 次印刷
印 数：5000 册 定价：48.00 元（含光盘 1 张）

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。联系电话：（010）68279077。质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dhqq@phei.com.cn。

序

一年多以前，电子工业出版社的张毅编辑告诉我说，有一位年轻的程序员，正在写一本《自己动手写操作系统》的书。知道这个消息，我既有点好奇，又有些担忧。如果是在十年前，这样题材的书将会是读者争相传阅的对象，毕竟20世纪90年代是软件的理想主义年代。但是在理想褪尽、实务未兴的尴尬的这两年，这样一本书在市场上究竟会遇到怎样的待遇，确实让人不敢乐观。不过，在阅读了样章之后，我深为作者清新的文笔、流畅的思路和扎实的技术功底所折服，于是请张毅为我引见了这位作者，即本书的作者于渊。

于渊非常年轻，却有着高人一筹的表达能力和技术视野，我觉得他是难得的技术写作人才，就鼓励他在《程序员》杂志开辟了一个技术专栏，专门剖析操作系统相关的技术。一年来这个专栏陆续发表了一系列文章，获得了不少读者的正面反馈。

然而，事实证明，我最初的担忧并不是没有道理的。一年多来，不断有人表达过他们对这样一个题材的不同看法。他们认为，相对于90年代中后期，现在的软件产业已经务实了很多，今天的程序员更关心的是如何尽可能快、尽可能简单地用软件解决实际问题，创造实际价值，在一个既定的秩序中寻找自己的生存空间，而不是异想天开地憧憬能成为Linus Torvalds式的旧秩序的“破坏者”。因此诸如软件过程、开发方法、系统集成、应用架构等“高级”的话题受到关注和欢迎，而诸如操作系统、编译原理之类的基础技术，已经是关心者寥寥了。他们非常怀疑，这样的一本书，对于一线的开发者是否有实际的意义？对于尚在寻找自己职业发展方向的初学者是否构成一种误导？这个问题相当尖锐，必须面对。我想这样一本书，至少在以下几个方面是具有重要的正面意义的。

首先，对于正在大学里学习计算机科学的学生来说，“操作系统原理”是重要的专业基础课。为了达到大学阶段教育的标准，这方面的知识应当认真学习。一些比较严肃的学校鼓励学生在学习这门课程的同时自己动手开发一个具体而微的操作系统。这种实习对于学生充分掌握书本知识、打下扎实的基本功有非常大的好处。在我认识的比较有成就的开发者中，有不少人自己动手写过小的操作系统，他们认为编写操作系统的实践使他们最终消除了对编写软件系统的心理障碍，实在地消化和理解了书本上的知识，学会了解决问题的思路，收获非常巨大。可惜的是，大部分的学生都没有进行过这样的实践，这主要是因为目前的课本偏重操作系统理论，把大量的笔墨放在对操作系统运行机制的剖析或者现成源代码的分析上，对于那些想自己动手写一个操作系统的同学来说，从课本上反而得不到实际的指导。即使是一些世界级的名著，在“How”上也是语焉不详。

在这方面，我相信于渊的这本书在国内算是填补了一个空白。这本书最大的特点是明白、实在，将学习编写操作系统的每一个步骤都清清楚楚地交代出来，丝毫没有含糊其辞之处。可以说，只要读者能够耐心阅读学习，按照书上交代的步骤一步步来，就肯定能够进入操作系统的大门，把书本上的知识与实践紧密联系起来。毕竟写自己的操作系统是一个让所有程序员心动的事情。如果当年我学习操作系统知识的时候能够有这样一本书，那该有多好！

其次，对于那些希望通过分析 Linux 源代码学习与研究操作系统，进而在开源软件天地里有所作为的研究者和开发者来说，这本书是非常好的入门阶梯。目前研究 Linux 内核的图书，一般局限在对现有内核源代码的分析上，不但理解起来很困难，而且没有给读者以自己实践的机会。有人想到去分析 Linux 早期的版本，降低了读者理解的难度，但是总的来说还是纸上谈兵。本书的风格截然不同，不但行文活泼清新，叙理简明清晰，而且完全着眼于动手，以一种夹叙夹议的方式，对于编写操作系统过程中可能遇到的各种问题“逢山开路，遇水架桥”，读者可以在实际的语境中理解问题，解决问题。通过这种方式学习操作系统的实现技术，无疑要比其他方式更为有效。而且，于渊在这本书中构造的这个微型的操作系统，跟 Linux 有微妙的相关，读者细心品味便知。

另外，虽然目前国内软件产业的主流是做下游的生产性集成，但是对于程序员个体来说，也有不少从事系统级软件开发的机会。有幸从事系统级软件开发的朋友，更是可以直接地从本书中学到不少实用的知识和技能。特别是作者在解决一个又一个问题的过程中所体现出来的思路和方法，可能是更值得大家学习的东西。

众所周知，操作系统是计算机软件领域中核心的工程性技术，尽管它的理论相对成熟，但是在工程实施和维护上，仍然是体现一个国家软件技术水平的“两弹、一星、大飞机”级的标志性核心技术。世界上凡是在软件产业方面存有雄心壮志的国家，无不非常重视操作系统技术的研究和积累。比如法国在他们的一个国家级实验室中，自己研发了包括操作系统和编译器在内的全套基础软件，并由国家投入资金不断维护和发展。德国拥有大批 Linux 黑客，其政府因势利导，通过一系列的大型工程将自己的 Linux 软件人才组织起来，希望依托 Linux 重建自己的软件核心技术力量。20 世纪 80 年代中期，日本在美国的压力下而放弃了自己的“BTRON”操作系统，此后软件产业的发展让日本追悔莫及。痛定思痛之后，日本希望牢牢地把握自己在消费电子产品上的优势，一方面继续发展国产的 ITRON OS，另一方面把握住 Linux 的机会，希望在未来占据消费类嵌入式操作系统的制高点。我国在这个方面走过一些弯路，但是现在已经认识到了掌握核心软件技术的重要性，并且有了一定的投入，相信今后国家在这方面的支持力度会越来越强。我本人见过国内的一些操作系统方面的专家，切实地感到，就个体而言，国内的技术专家在理论和实践上都达到了很高的水平，但是由于缺乏一个质量高、并且有一定规模的

团队和社群，他们基本处于单打独斗或者小组作战的状态，不仅个人的技术不能够得到充分地发挥，而且也不能形成有规模的成果，无法从根本上扭转我国在软件核心技术领域上的劣势。

我认为，只有在中国出现一大批关心操作系统、熟悉操作系统的程序员，才有可能逐渐缩小我们与世界先进水平的差异。

作为中国软件产业中的普通一员，我非常希望看到这本书能够在这个过程中发挥一点作用。

孟　岩

《程序员》杂志技术主编

2005 年 7 月

审校者的话

这是一本编程爱好者编写的别具一格、颇有特色的操作系统原理与实现的书。该书作者对操作系统具有特殊爱好，在大量实践和反复钻研下积累了丰富而可贵的经验，为了与广大读者分享这些经验写成了此书。

本书对一般的操作系统原理教材不很重视的部分，例如，系统初启、保护模式、控制权如何转入OS Kernal等都写得具体详细，对操作系统的爱好者以及涉足于操作系统设计、实现和应用的读者有很好的参考价值。

本书的文字生动活泼，富有个性，可望提高青年学子的阅读兴趣。

尤晋元

尤晋元老师简介：

上海交通大学教授，博士生导师，国内操作系统领域著名权威专家，代表著作（含译著）有：《Windows操作系统原理》，《操作系统：设计与实现》，《莱昂氏UNIX源代码分析》等。

作 者 自 序

你是否有过这样的经历，有一天你兴致勃勃买来一堆菜谱想学厨艺，翻开之后却发现自己根本没见过那些材料的名字，也不知道什么叫文火什么叫武火，什么叫上浆什么叫勾芡。而菜谱里根本没告诉你！你扔掉菜谱，垂头丧气，从此对厨艺失去兴趣。

你也可能会有这样的经历，当你在计算机课上学完了一堆 C 语言语法，想要大展身手实践一番的时候，突然发现你居然不知道源代码应该敲到哪里，是 Word 还是 NotePad？

很多计算机自学者可能有过这样的经历，由于不知道如何跟踪调试，在辛辛苦苦编写的程序得不出正确的结果时，要么束手无策，要么用打印语句输出很多东西，费时费力，而教科书根本没教你这些操作的细节。

有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，约定俗成是由学习者本人去摸索出来的。但是实际情况恰恰是，这些书中忽略掉的内容可能占去了一个初学者大部分的时间，甚至因此影响了学习的热情。

学 C 语言是很容易找到老师的，你会被详细地告知 IDE 是什么，以及如何使用。但是学习操作系统呢？你会发现绝大多数操作系统书籍都只讲原理，讲各种各样的算法和策略。如果是为了考试，你将内容背下来，最后可能得一个高分；如果是出于兴趣，怕是读了没几页就感到索然无味了。你或许的确能找到极少数的书籍告诉你怎样去写，比如 Andrew S. Tanenbaum 和 Albert S. Woodhull 的《操作系统：设计与实现》，但是两位先生还是没能告诉你从哪里开始。你还是不得不在一开始的时候在浩瀚的因特网上搜索一个 BootSector 的写法。

你或许听说过张五常，他为了研究经济学问题亲自跑到大街上去卖橘子，后来写成了著名的《卖橘者言》，成为了实证经济学的典范。他没有仅仅躲在房子里研究，因为他相信通过实践得来的经验才最可靠、最深刻的。我想他真的是一个喜欢追根究底的人。

你可能也喜欢探求问题的本质，想了解事情的各个细节。面对神奇的计算机世界，很想知道为什么打开电源，电脑屏幕上就能出现这样色彩斑斓的图像，很想知道操作系统理论书籍中讲到的进程管理到底怎样实现，很想知道 DOS 和 Windows

到底有什么本质上的区别，想知道怎样才能像那些伟大的黑客一样参与修改 Linux 的源代码。是的，我就是这样一个喜欢探求本质的人，对这一切怀有极大的兴趣，于是我想写一个自己的操作系统，因为我知道只有通过自己动手，才能对它有真正深刻的了解。经过一段时间的努力，我终于完成了一个雏形。回头想想，很庆幸自己能克服困难走了过来。同时，我在网上了解到还有很多朋友也在写自己的操作系统，也遇到了许多困难。为了跟大家分享其中的经验，让后来者不至于走同样的弯路，我把自己的开发过程记录下来，希望能为初学者们做参考。

本书有幸得到国内操作系统方面的研究权威、上海交通大学尤晋元教授的审校和指点，在此我要表示最真心的敬意和最诚挚的感谢！

在本书的写作过程中，我也有幸得到了许多人的帮助。在我遇到一些未曾预料的困难和变故时，得到了电子工业出版社的张毅编辑以及《程序员》杂志社的孟岩先生的理解、宽容和支持，在这里，我要对你们表示衷心的感谢！

我同样想把感谢献给朱风明、郝慎水、包立光、李雷、郭洪桥和张璐。在整个过程中，你们都在方法上、精神上和物质上给予我巨大而无私的帮助。没有你们的付出，我不可能完成这项工作。

我还要感谢我的家人，爷爷、爸爸和妈妈，你们的爱和关怀是本书得以完成的保障。

要列出所有帮助过我的人的名字是不可能的，因为有些困难是通过因特网解决的，我甚至不知道他们的名字。在此，谨向他们一并表示感谢！

最后，以下面四句诗与读者共勉：

在你立足处深挖下去，

就会有泉水涌出！

别管蒙昧者们叫嚷：

“下面永远是地狱！”

——尼采

于　渊

本书导读

纸上得来终觉浅，绝知此事要躬行。

——陆游

本书适合谁读

本书是一本操作系统开发实践的技术书籍，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，都可以在本书中了解到实践所需的知识和思路。

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，都进行讨论，从开发环境的搭建到保护模式，再到 IBM PC 中有关芯片的知识，最后到操作系统本身的设计实现，都能在本书中找到相应介绍。所以，如果你也想亲身实践的话，本书可以使你的学习过程事半功倍。在读完本书后，你不但对于操作系统有一个初步的感性认识，并且对 IBM PC 的接口、IA 架构的保护模式，以及操作系统整体框架都会有一定程度的了解。

当你读完本书之后，再读那些纯理论性的操作系统书籍，体验将会完全不同，因为那些对你而言已不再是海市蜃楼了！

对于想阅读 Linux 源代码的操作系统爱好者来说，本书可以提供阅读前所必需的知识储备，这些知识也是很多 Linux 书籍没有提到的。

特别要说明的是，对于想通过 Andrew S. Tanenbaum 和 Albert S. Woodhull 所著的《操作系统：设计与实现》来学习操作系统的读者，本书尤其适合作为引路指南，因为它详细地介绍了初学者入门时所必需的知识，而这些知识在《操作系统：设计与实现》一书中是没有涉及的。笔者本人把该书作为写操作系统的主要参考书籍之一，所以在写作本书时对它多有借鉴。

你需要具备的基础

在本书中所用到的计算机编程语言只有两种：汇编语言和 C 语言。所以，只要你具备汇编语言和 C 语言的知识，就可以阅读本书。除对操作系统有常识性的了解（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，就会发现本书是针对理论知识的补充，它是从实践的角度为你展现一幅操作系统画面。

书中涉及到 Intel CPU 保护模式、Linux 命令等内容。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

• 有效而愉快地学习

你大概依然记得，在亲自敲出第一个“Hello world”程序并运行成功时的喜悦，那样的成就感点燃了你对编写程序的浓厚兴趣。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，又学到新知识，并获得新的成就感。

可现在请你设想一下，假如课程不是这样安排的，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？我想惟一的感受就是索然无味。

原因何在？因为你体会不到通过不断实践而带来的一次一次的成就感。而成就感是学习过程中快乐的源泉。没有了成就感，学习效率将大打折扣。

每个人都希望有效且愉快地学习，可不幸的是，我们见到的操作系统课程十之八九都是在喋喋不休地讲述着进程管理、存储管理、I/O 控制、调度算法，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。笔者厌烦这样的学习过程，也绝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，本书完全可以称做一本回忆录，记载了笔者从一开始（不知道保护模式为何物）到最后（形成一个小小的操作系统）的全过程。回忆录性质保证了章节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近在眼前。步步为营是我们的战术，不断获取成就感是我们的宗旨。

我们将从 20 行代码开始，使最最简单的操作系统婴儿慢慢长大，长成一位翩翩少年。而其中的每一步，都可以在书的指导下自己完成。不仅仅是可以看到，而且是自己能够做到！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的提升过程**

在你没有登过泰山之前，无论书中怎样描写它的样子，你都无法想像出它的真实面目，即便配有插图，你对它的了解也仍是支离破碎的。毫无疑问，一千本描述泰山的书都比不上你一次登山的经历。文学家的描述是华丽而优美的，可这样的描述最终产生的效果是促使你非去亲自登泰山不可。反过来呢？假如你已 经登过泰山，这样的经历所产生的效果会使你想读尽天下所有描述泰山的书吗？恰恰相反，你可能再也不想去看那些文字描述了。

是啊，再好的讲述也比不上亲身的体验。人们的认知规律本来如此，有了感性的认识，才能上升为理性的思考。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，引领你进入其中。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。而随着本书去亲身体验，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。

要强调一点，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轻松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地理解的时候，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完全不同。

对于操作系统这样复杂的事物来说，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产生。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，一定会让你备感愉悦。

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中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，精妙的背后一定隐藏着很多中间产 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，但即便不足以做到，只要能引起读者们的一点思索，也是莫大的欣慰。

- **挡住前路的，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什么是困难。就好像你买了一台功能很全

的微波炉回家，研究完整本说明书，踌躇满志地想要烹饪的时候，却突然发现家里的油盐已经用完。而当时已经是晚上 11 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 10 块钱 1 瓶的油，以及更加微不足道的 1 块钱 1 袋的盐。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，本书详述了这一大堆可能遇到却想不到的问题。所以，惟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，只是出于自己的兴趣，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。让我们马上投入实践，遇到问题再寻觅解决的办法。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

由于我们知道我们为什么学习，所以才会非常投入；由于我们知道我们的目标是解决什么问题，所以才会非常专注；由于我们在实践中学习，所以才会非常高效。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了更多的东西，并且格外扎实。

• 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 3 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读完的时候，我想起当初的畏惧，算来时间其实并没有过去多久。

所有的道理都是相通的，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过，所以我们更应无所畏惧。更何况不仅有书店，而且有因特网，动动手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，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掠过，

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一下，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

- **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。因为笔者自己在读书的时候有一个体验，就是有时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”。可能该书的作者认为它足够简单以至于可以跳过不提，但读者未必一下子就能弄清楚。所以，本书在很多地方将尽量地细节阐述得很清楚，以节省、读者理解的时间。

在本书后面的章节中，如果涉及的细节是前面章节提到过的，就会有意地略过。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，就不再提及了。

- **努力做到平易近人**

笔者更喜欢把本书称做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。如果哪怕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。

- **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，暂时并不考虑用做实际用途；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也不迟。

光盘说明

书后所附光盘中有本书用到的所有源代码。源代码并不是只有一份，而是每增加一部分模块就有一份，后一部分所附代码是在前一部分基础上完成的，所以读者在阅读本书的时候可以做到彻底的步步为营，你不必从一大堆不必要的代码中找出自己想要的部分。

在书的正文引用的代码中会标注出出自哪个文件。以在 Windows 下为例，如果光盘的盘符是“F:”，则“\chapter5\b\bar.c”表示文件的绝对路径为“F:\Tinix\ chapter5\b\bar.c”。

另外，光盘中还有一些必要的小工具，以供读者方便地使用。

光盘内容摘要请见下表。

位置	内容	说明	
\Tinix	书中涉及的全部源代码	书中的章节和代码对照表请见附录	
	其中很多目录中除了包含源代码（*.asm、*.inc、*.c、*.h）外，还有其他类型的一些文件	boot.bin	引导扇区（Boot Sector），可通过 FloppyWriter 写入软盘（或软盘映像）
		loader.bin	Loader，直接复制至软盘（或软盘映像）根目录
		kernel.bin	内核（Kernel），直接复制至软盘（或软盘映像）根目录
		bochsrc.bxrc	Bochs 配置文件，如果系统中安装了 Bochs 2.1.1 可直接双击运行它，其他细节请见 2.7 节
		godbg.bat	调试时可使用此批处理文件，它假设 Bochs 2.1.1 安装在 D:\Program Files\Bochs-2.1.1\中
		TINIX.IMG	软盘映像，可直接通过 Bochs 或者 Virtual PC 运行
		*.com	可以在 DOS（必须为纯 DOS）下运行的文件
\Tools	一些小工具（在 VC 6 下编译通过）	DescParser	描述符分析器，输入描述符的值，可以得出其基址、界限、属性等信息
		ELFParser	ELF 文件分析器，可以列出一个 ELF 文件的 ELF Header、Program Header、Section Header 等信息
		FloppyWriter	用以写引导扇区，支持软盘和软盘映像
		KrnlChecker	用以检查一个 Tinix 内核加载后位置是否正确

读者与作者技术交流，可上书友论坛：http://forum.broadview.com.cn。

意见反馈请发邮件至：editor@broadview.com.cn 或 jsj@phei.com.cn。

目　录

第 1 章　马上动手写一个最小的“操作系统” ……1

1.1　准备工作 ……1

1.2　10 分钟完成的操作系统 ……1

1.3　Boot Sector ……3

1.4　代码解释 ……3

1.5　水面下的冰山 ……5

1.6　回顾 ……6

第 2 章　搭建你的工作环境 ……7

2.1　虚拟计算机（Virtual PC） ……7

2.1.1　Virtual PC 初体验 ……8

2.1.2　创建你的第一个 Virtual PC ……9

2.1.3　虚拟软盘研究 ……12

2.1.4　虚拟软盘实战 ……14

2.2　编译器（NASM & GCC） ……18

2.3　安装虚拟 Linux ……19

2.4　在虚拟 Linux 上访问 Windows 文件夹 ……26

2.5　安装虚拟 PCDOS ……26

2.6　其他要素 ……29

2.7　Bochs ……29

2.7.1　Bochs vs. Virtual PC vs. VMware ……30

2.7.2　Bochs 的使用方法 ……31

2.7.3　用 Bochs 进行调试 ……33

2.7.4　在 Linux 上开发 ……34

2.8　总结与回顾 ……36

第 3 章　保护模式（Protect Mode） ……37

3.1　认识保护模式 ……37

3.1.1　GDT(Global Descriptor Table) ……42

3.1.2　实模式到保护模式，不一般的 jmp ……45

3.1.3 描述符属性……47
3.2 保护模式进阶……50
3.2.1 海阔凭鱼跃……50
3.2.2 LDT（Local Descriptor Table）……58
3.2.3 特权级……62
3.3 页式存储……82
3.3.1 分页机制概述……83
3.3.2 编写代码启动分页机制……84
3.3.3 PDE 和 PTE……85
3.3.4 cr3……88
3.3.5 回头看代码……88
3.3.6 克勤克俭用内存……90
3.3.7 进一步体会分页机制……100
3.4 中断和异常……107
3.4.1 中断和异常机制……109
3.4.2 外部中断……111
3.4.3 编程操作 8259A……113
3.4.4 建立 IDT……116
3.4.5 实现一个中断……117
3.4.6 时钟中断试验……119
3.4.7 几点额外说明……121
3.5 保护模式下的 I/O……122
3.5.1 IOPL……122
3.5.2 I/O 许可位图（I/O Permission Bitmap）……123
3.6 保护模式小结……123
第 4 章 让操作系统走进保护模式……125
4.1 突破 512 字节的限制……125
4.1.1 FAT12……126
4.1.2 DOS 可以识别的引导盘……131
4.1.3 一个最简单的 Loader……132
4.1.4 加载 Loader 入内存……133
4.1.5 向 Loader 交出控制权……142
4.1.6 整理 boot.asm……142
4.2 保护模式下的“操作系统”……144

第 5 章　内核雏形……146
5.1　用 NASM 在 Linux 下写 Hello World……146
5.2　再进一步，汇编和 C 同步使用……148
5.3　ELF（Executable and Linkable Format）……150
5.4　从 Loader 到内核……155
5.4.1　用 Loader 加载 ELF……155
5.4.2　跳入保护模式……161
5.4.3　重新放置内核……170
5.4.4　向内核交出控制权……175
5.4.5　操作系统的调试方法……176
5.5　扩充内核……184
5.5.1　切换堆栈和 GDT……184
5.5.2　整理我们的文件夹……191
5.5.3　Makefile……191
5.5.4　添加中断处理……200
5.5.5　两点说明……218
5.6　小结……219
第 6 章　进程……221
6.1　迟到的进程……221
6.2　概述……222
6.2.1　进程介绍……222
6.2.2　未雨绸缪——形成进程的必要考虑……222
6.2.3　参考的代码……224
6.3　最简单的进程……224
6.3.1　简单进程的关键技术预测……225
6.3.2　第一步——ring0→ring1……227
6.3.3　第二步——丰富中断处理程序……243
6.3.4　进程体设计技巧……254
6.4　多进程……256
6.4.1　添加一个进程体……256
6.4.2　相关的变量和宏……257
6.4.3　进程表初始化代码扩充……258
6.4.4　LDT……260
6.4.5　修改中断处理程序……261

6.4.6 添加一个任务的步骤总结……263
6.4.7 号外：Minix 的中断处理……265
6.4.8 代码回顾与整理……269
6.5 系统调用……280
6.5.1 实现一个简单的系统调用……280
6.5.2 get_ticks 的应用……286
6.6 进程调度……292
6.6.1 避免对称——进程的节奏感……292
6.6.2 优先级调度总结……300
第 7 章 输入/输出系统……302
7.1 键盘……302
7.1.1 从中断开始——键盘初体验……302
7.1.2 AT、PS/2 键盘……304
7.1.3 键盘敲击的过程……304
7.1.4 解析扫描码……309
7.2 显示器……325
7.2.1 初识 TTY……325
7.2.2 基本概念……326
7.2.3 寄存器……328
7.3 TTY 任务……332
7.3.1 TTY 任务框架的搭建……334
7.3.2 多控制台……340
7.3.3 完善键盘处理……346
7.3.4 TTY 任务总结……354
7.4 区分任务和用户进程……354
7.5 printf……357
7.5.1 为进程指定 TTY……357
7.5.2 printf()的实现……358
7.5.3 系统调用 write()……361
7.5.4 使用 printf()……363
后记……366
参考文献……369
附录 书中的章节和代码对照表……370

第1章

马上动手写一个最小的“操作系统”

勿以善小而不为。

——刘备

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（Windows 操作系统）
- 一张空白软盘

2. 软件

- 汇编编译器 NASM。最新版本可以在此链接处获得：http://sourceforge.net/projects/nasm。（此刻你可能会有疑问：为什么是 NASM，而不是 MASM 或者 TASM？对于这一点本书后面会有解释。）
- 软盘绝对扇区读写工具。比如本书附赠光盘中的 FloppyWriter.exe。

1.2 10 分钟完成的操作系统

你相不相信，一个“操作系统”可以只有 20 行代码？请看：

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, 0i301h      ; 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 512B 的 boot.bin，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。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 Virtual PC 的话（下文中将会有关于 Virtual PC 的详细介绍），你应该能看到图 1-1 所示的画面。

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！

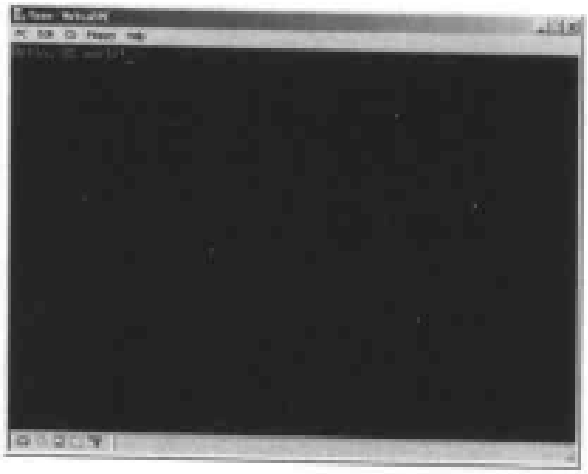

图 1-1　最小的“操作系统”

1.3　Boot Sector

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 OS，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（假如我们把此扇区看做一个字符数组 sector[]的话，那么此结束标志相当于 sector[510]=0x55，且 sector[511]=0xAA）结束，则 BIOS 认为它是一个引导扇区，也就是我们说的 Boot Sector。当然，一个正确的 Boot Sector 除了以 0xAA55 结束之外，还应该包含一段少于 512B 的执行码。

好了，一旦 BIOS 发现了 Boot Sector，就会将这 512B 的内容装载到内存的 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 7c00 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有 5 行（从第 2 行到第 6 行），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。

可能大部分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）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 = 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。这好像不太好理解，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$$表示什么呢？它表示一个节（section）的开始处被汇编后的地址。在这里，我们的程序只有 1 个节，所以，$$实际上就表示程序被编译后的开始地址，也就是 0x7c00。

> **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。**

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 510B 为止。

这样，加上结束标志 0xAA55 占用的 2B，恰好是 512B。

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 Boot Sector，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，以 Boot Sector 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，我们只要把“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 Boot Sector 就制作完毕了。调试.COM 文件可能让你仿佛一下子回到了 20 世纪，没关系，怀旧一下感觉还是蛮不错的。

Turbo Debugger 是一个不错的调试工具，“图形化”界面，全屏操作，用起来感觉和一个“严重”精简后的 Windows 差不多☺。

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 节自\chapter1\b\boot.asm

```
;%define    _BOOT_DEBUG_     ; 做 Boot Sector 时一定将此行注释掉!
                            ; 将此行打开后用 nasm Boot.asm -o Boot.com
                            ; 做成一个.COM 文件易于调试
%ifdef  _BOOT_DEBUG_
    org  0100h        ; 调试状态，做成 .COM 文件，可调试
%else
    org  07c00h       ; Boot 状态，BIOS 将把 Boot Sector 加载到 0:7C00
                      ; 处并开始执行
%endif
```

这样一来，如果我们想要调试，就让第一行有效，想要做 Boot Sector 时，将它注释掉就可以了。

这里的预编译命令跟 C 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，更具备了扩充它的条件——知道如何排错，如何调试。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，你无所不能。

如果说开始我们还是在黑暗中摸索，到现在为止，我们至少已经看到了光亮，我们有信心将 Boot Sector 进行不断扩充，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！

第2章

搭建你的工作环境

工欲善其事，必先利其器。

——孔子

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 Boot Sector 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，我知道此刻你跃跃满志，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 Virtual PC。

在介绍 Virtual PC 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 Virtual PC 开始介绍。

2.1 虚拟计算机（Virtual PC）

即便没有听说过虚拟计算机，你至少应该听说过磁盘映像。在 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 Virtual PC 初体验

先看几个屏幕截图。

图 2-1 为 Virtual PC 中的 Windows 2000 启动界面。

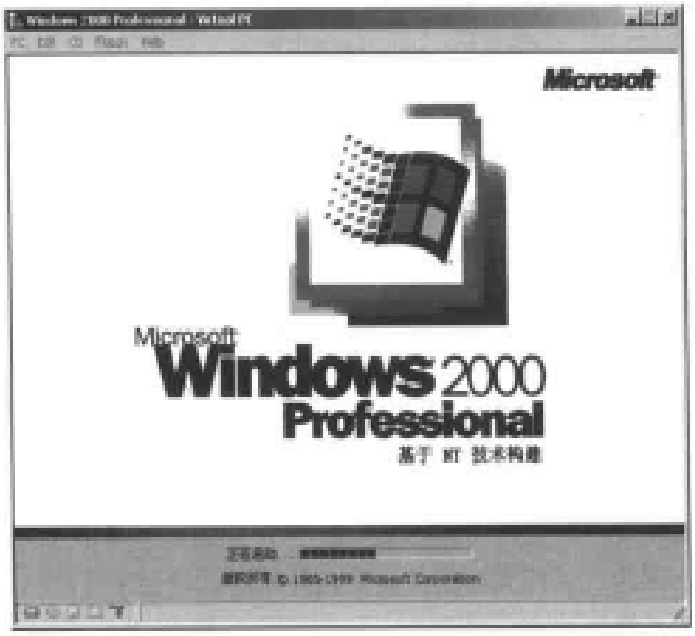

图 2-1 Virtual PC 中的 Windows 2000 启动界面

启动完毕的界面如图 2-2 所示。

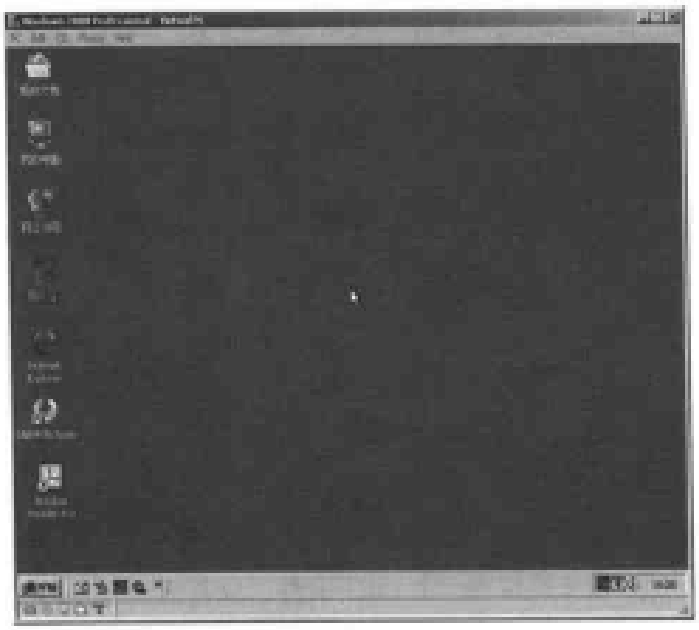

图 2-2 Virtual PC 中的 Windows 2000 启动完毕界面

读者一定要看清楚，眼前的画面不是一个显示器，而仅仅是一个窗口而已。Virtual PC 的意思就是“一个好像 PC 一样的窗口”。是的，在这个窗口中，你可以像对待真的 PC 一样进行各种操作，甚至在网络的其他计算机上，你也可以像访问一个真实的 PC 一样访问它，根本不知道它其实只是一个 Virtual PC。

真的很神奇，不是吗？更令人激动的是，这正是我们开发操作系统所需要的工具！有了它，我们不再需要频繁地重启计算机，即使程序有严重的问题，也丝毫伤害不到你的爱机。你可能迫不及待地要先试为快了，好的，我们现在就以 Virtual PC 5.0 为例，来简单介绍一下它的用法。

2.1.2　创建你的第一个 Virtual PC

安装完成，第一次运行，你将会看到如图 2-3 所示的窗口。

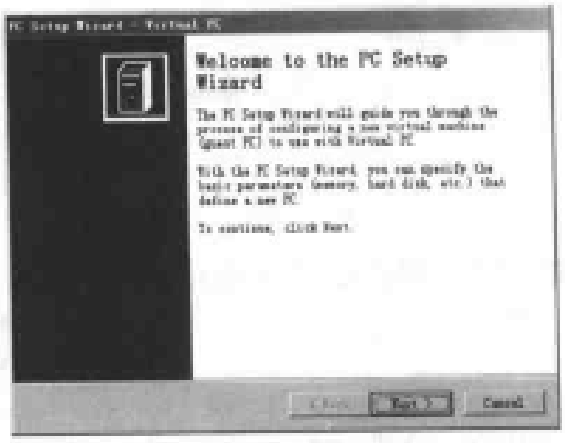

图 2-3　PC 设置向导

使用这个向导可以创建你的第一个虚拟计算机，也就是 Virtual Machine 或者 Guest PC。

（1）单击“Next”按钮，界面如图 2-4 所示。

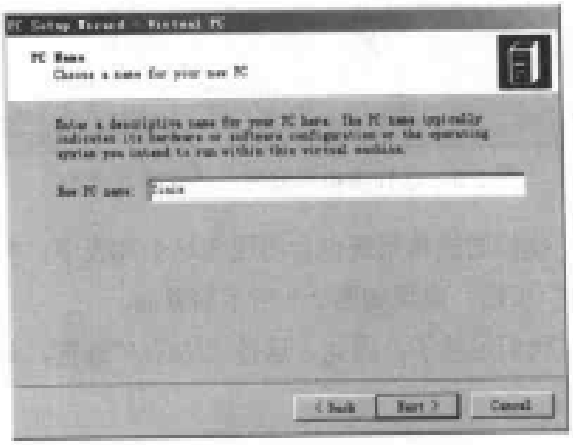

图 2-4　填写 PC 名称

（2）将“New PC name”命名为“Tinix”，然后单击“Next”按钮，出现如图 2-5 所示的界面。

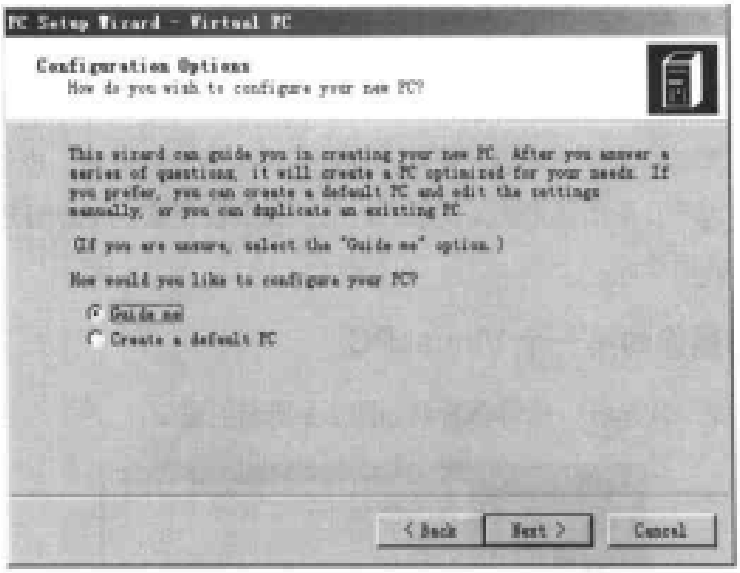

图 2-5　配置选项

（3）选择“Guide me”，单击“Next”按钮，出现如图 2-6 所示的界面。

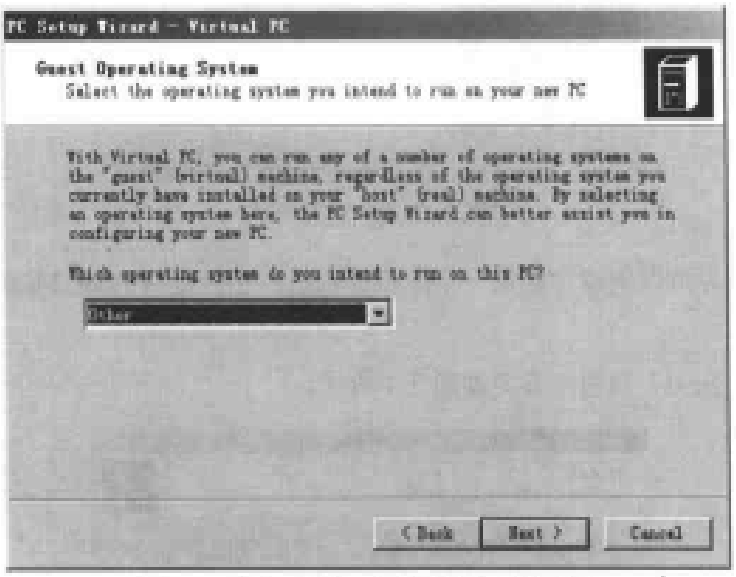

图 2-6　选择虚拟机安装的操作系统类型

（4）或许有一天，我们的操作系统也会出现在这个列表中，但现在，我们只能选择“Other”，单击“Next”按钮，出现如图 2-7 所示的界面。

（5）32MB 内存对我们足够了，因此，选择“No”单选框，单击“Next”按钮，出现如图 2-8 所示的界面。

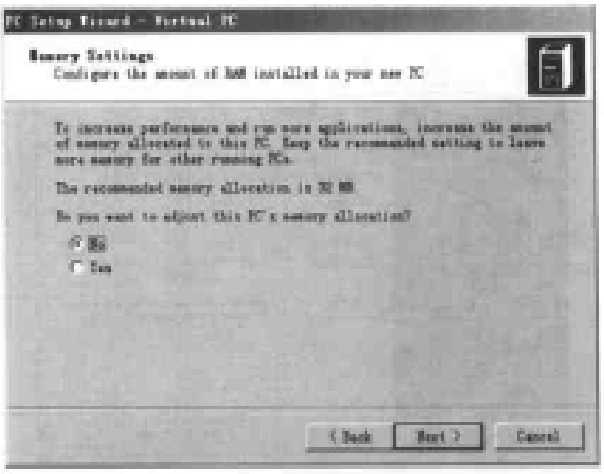

图 2-7　配置内存大小

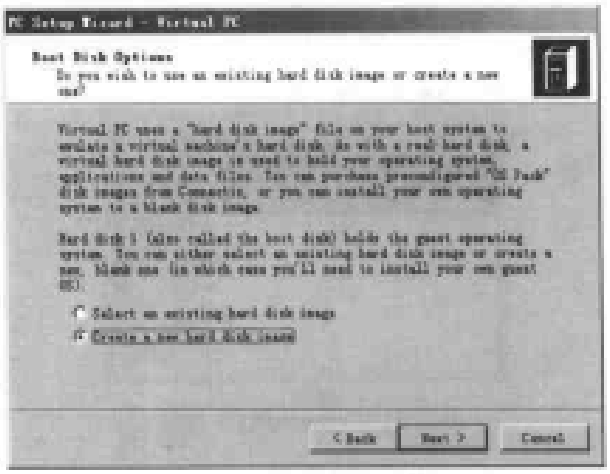

图 2-8　启动盘选项

（6）默认需要建一个虚拟硬盘，虽然开始我们并不需要它，不过建一个也无妨。然后单击“Next”按钮，出现如图 2-9 所示的界面。

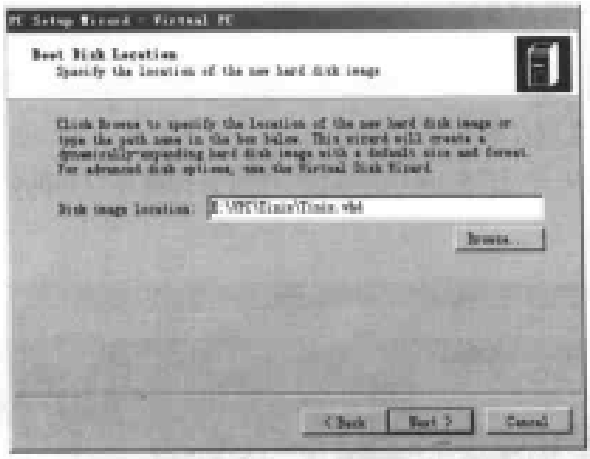

图 2-9　启动盘位置

（7）选择一个存储位置，然后单击“Next”按钮，出现如图 2-10 所示的界面。

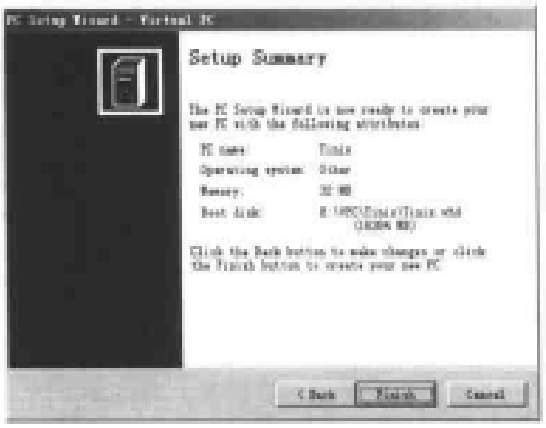

图 2-10　总结信息

（8）单击“Finish”按钮，出现如图 2-11 所示界面。

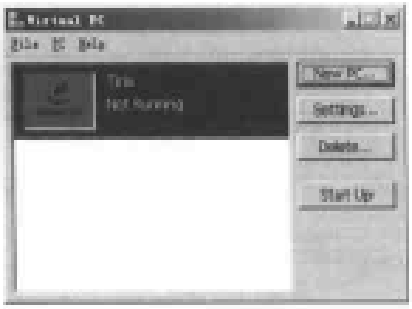

图 2-11　Virtual PC 主界面

到此为止，你的第一个 Virtual PC 就已经建立完毕了，那么，现在是不是已经可以单击“Start Up”了呢？先不要着急，现在它还只是一台“裸机”，现在就用我们的 Tinix 启动盘来试试看。

2.1.3　虚拟软盘研究

Virtual PC 是很灵活的，它允许你使用“真的”和“虚拟的”两种软盘。现在已经可以将先前做好的启动盘放在软驱中，用它启动。

将软盘放入软驱中，单击“Start Up”按钮，然后选择菜单“Floppy → Capture Host Drive A:”。如图 2-12 所示。

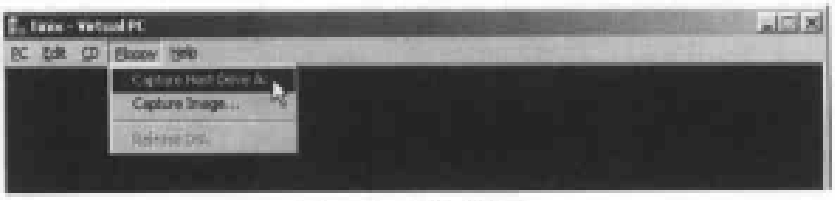

图 2-12　从软盘启动

然后，虚拟机会从软盘启动。最后，会看到如图 2-13 所示的界面。

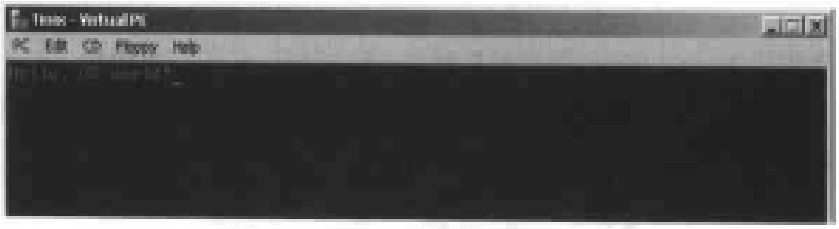

图 2-13　Tinix 引导扇区运行结果

虽然这是意料中的事，不过仍然很奇妙，不是吗？“像真的一样”，你会说。是的，像真的一样。有了它，你就好像有了许多台计算机，而且，你可以用同一个键盘和同一只鼠标来操作它们。

软盘的读写速度显然是太慢了，说不定，当你拿出一张勾起你若干回忆的尘封的软盘时，发现它再也不像当初那么好用了，咔咔若干声之后，一个错误提示出现在你的眼前，告诉你这个磁盘已经损坏。于是你知道，时间能改变的，除了爱情，还有软盘！

所以我们需要将这台 PC 彻底地虚拟化，丢弃让人伤心的软盘，使用映像。

有一种简单的方法，就是使用一种工具将我们已有的软盘做成.IMG 文件，比如使用 DOS 时代的 HD-COPY。可是，在以后每一次更新的时候，我们不能总是先操作软盘，然后再把它做成.IMG，所以，我们需要研究一下.IMG 的格式，以便在以后可以方便地操作它。

我们先用 HD-COPY 把启动盘制作成一个映像，假设是 A.IMG。然后用一个二进制查看器打开它，如图 2-14 所示。

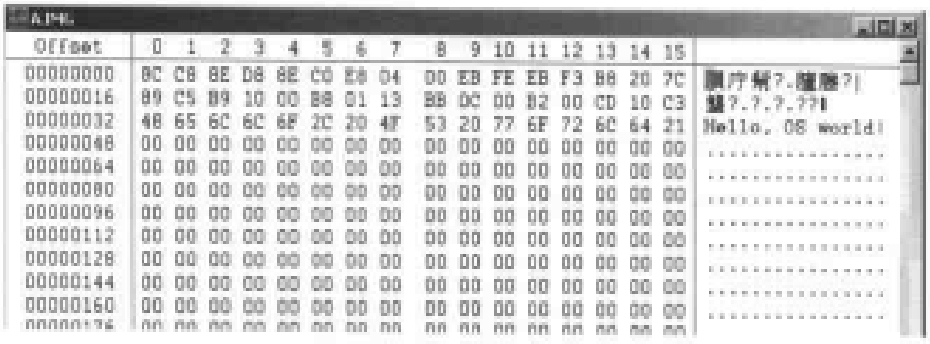

图 2-14　用二进制查看器观察软盘映像

打开之后，我们恍然大悟，原来映像文件的开始处就是软盘的 0 道 0 面 1 扇区。这下你心里有底了，使用简单的文件操作，我们就可以对软盘映像进行绝对扇区读写。

笔者自己编写了一个小工具，可以在本书的附带光盘中找到，名字叫做 FloppyWriter，用它可以将 Boot Sector 写进实际软盘或者软盘映像。源代码也在光盘中，代码非常简单，而且为自己的需要专门定制，使用起来极为方便。

2.1.4 虚拟软盘实战

现在，我们用 Virtual PC 和这个工具制作一张虚拟启动盘，步骤如下。

（1）首先，在 Virtual PC 主界面中选择“File→Virtual Disk Wizard”，弹出如图 2-15 所示的对话框。

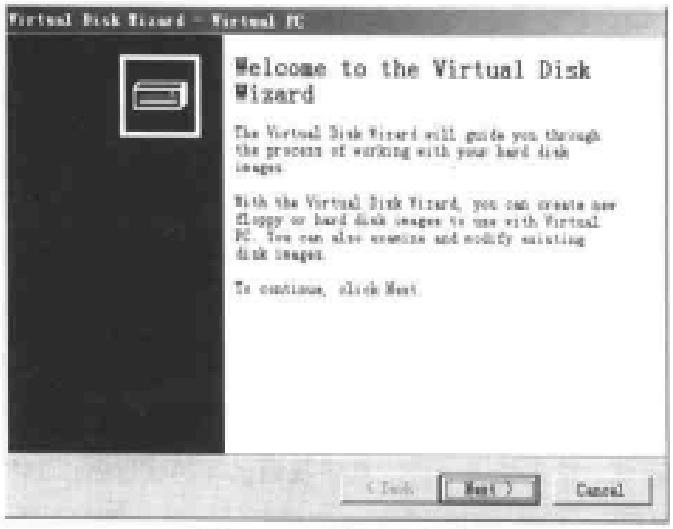

图 2-15 虚拟磁盘向导

（2）单击“Next”按钮，出现如图 2-16 所示的界面。

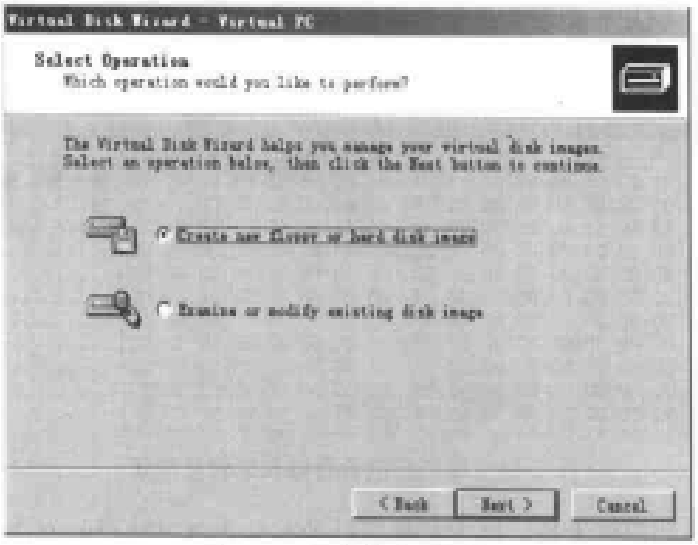

图 2-16 选择软盘或硬盘

（3）选择第一项“Create new floppy or hard disk image”，单击“Next”按钮，出现如图 2-17 所示的界面。

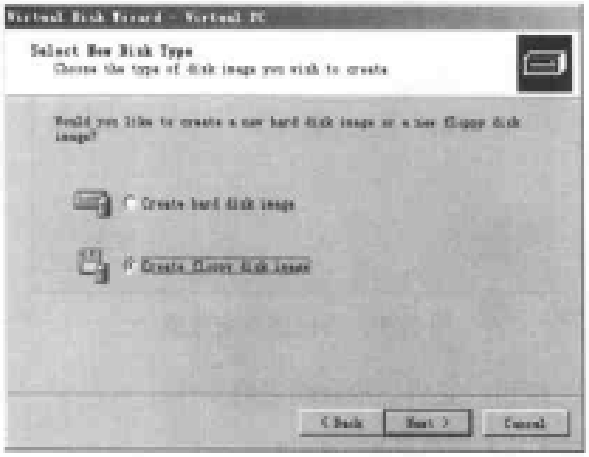

图 2-17　选择磁盘类型

（4）选择第二项“Create floppy disk image”，单击“Next”按钮，出现如图 2-18 所示的界面。

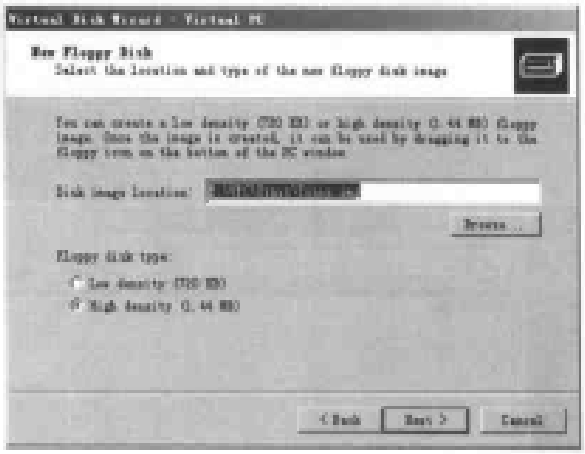

图 2-18　指定映像文件路径

（5）选择一个存储位置，单击“Next”按钮，出现如图 2-19 所示的界面。

图 2-19　结束界面

（6）单击“Finish”按钮，出现如图 2-20 所示的界面。

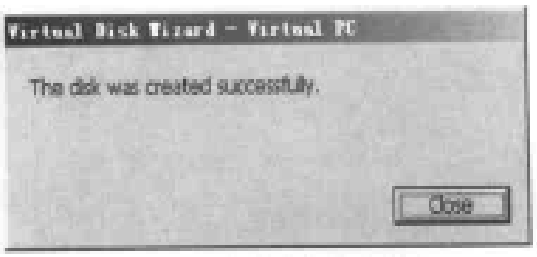

图 2-20　虚拟磁盘创建成功

（7）单击“Close”按钮，完成虚拟启动盘的制作。

下面我们用 FloppyWriter 写入 Boot Sector。

（1）在主界面上单击“Write File to Image”按钮，如图 2-21 所示。

图 2-21　用 FloppyWriter 往软盘映像中写入引导扇区（选择引导扇区）

（2）打开编译好的 Boot Sector，然后，程序会打开一个窗口提示打开映像文件，如图 2-22 所示。

（3）单击“打开”按钮，程序会将 Boot Sector 写入软盘映像，如果没什么意外，你会看到操作成功的提示，如图 2-23 所示。

图 2-22　用 FloppyWriter 往软盘映像中写入引导扇区（选择软盘映像）

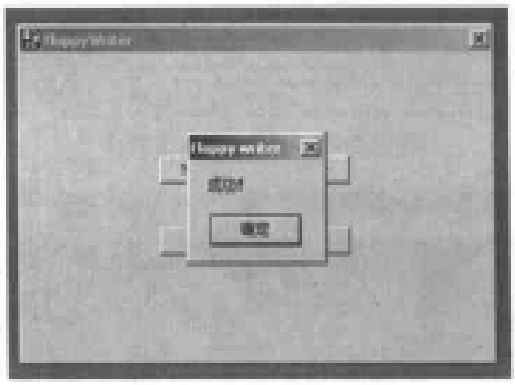

图 2-23　用 FloppyWriter 往软盘映像中写入引导扇区（成功提示）

现在，我们有了一张准备好的具有引导扇区的软盘映像，我们可以用它来启动了。

（1）首先启动 Tinix，然后选择菜单“Floppy→Release A:”，将刚刚捕获的 A 盘释放掉。如图 2-24 所示。

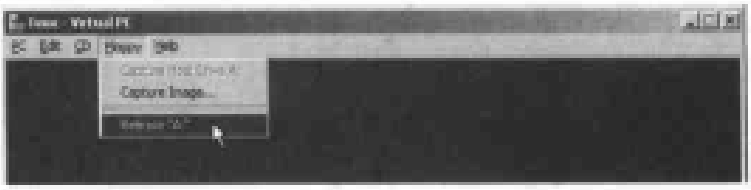

图 2-24　弹出挂载到 Virtual PC 上的软盘

（2）选择菜单“Floppy→Capture Image”，如图 2-25 所示。

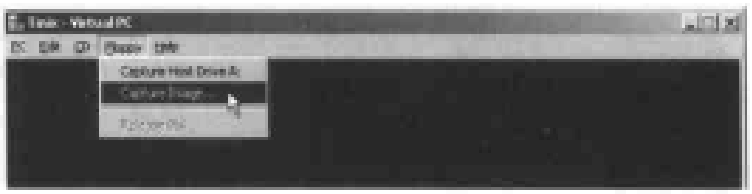

图 2-25　使用软盘映像

（3）在弹出的窗口中选择要捕获的映像文件，如图 2-26 所示。

（4）单击“打开”按钮，等待启动完成，界面如图 2-27 所示。

到目前为止，我们已经大体知道了虚拟机的用法。实际上，虚拟机软件不仅仅是 Virtual PC 一种，还有 VMware、Bochs 等。但是，相对于 VMware 来说，Virtual PC 的通用性更强，它声称“支持所有基于 x86 的操作系统”，而且从使用习惯上来讲，笔者本人更倾向于 Virtual PC。至于 Bochs，下文中会有相应介绍。

图 2-26　选择一个软盘映像

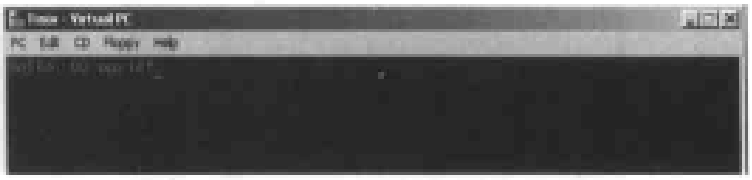

图 2-27　用刚挂载的软盘映像作为启动盘来启动

2.2　编译器（NASM & GCC）

现在，我们仅仅是一个开始，还没有涉及到开发平台的问题。用 NASM 编译一段可执行代码，在什么平台下完成都是一样的。但是我们终究要面对平台的问题，选择 Windows 还是 Linux？

在很多地方，我们参考了 Minix 和 Linux。Linux 在一定程度上就是参考 Minix 完成了最初的版本。由于 Linux 是学习操作系统最全面、最细致的教材，所以，今后我们的操作系统可以在很多方面跟 Linux 学习。基于这样的考虑，我们选择在 Linux 下使用 GCC 来编译代码。

我们只是想通过亲自编写代码的方式来学习操作系统的原理，而不是想做一个全新的操作系统，我们所应做的仅仅是学习罢了，可能有一天，我们在这个领域也有了自己的创新，但那一天我们一定已经站在了巨人的肩上，目前而言，我们还只能看到巨人的脚踝。

GCC 是一个庞大、复杂的工具，有着上百个参数，有一些参数可能一辈子也用不到，没关系，遵循我们的原则："用到什么再学什么"，等我们下文用到的时候再做必要的介绍。

C 编译器是必选的，我们再来回顾一下前面已经用过的汇编编译器 NASM。

如果在本书的第 1 章，你的疑问是"为什么是 NASM 而不是 MASM"，那么现在你的疑问可能变成了"为什么是 NASM 而不是 GAS"。因为 MASM 和 GAS 都太极端了，还是 NASM 比较中庸，它既可以在 Windows 平台下使用，又可以在 Linux 平台下使用。而且，如果没有接触过 AT&T 格式的汇编，GAS 看上去实在太奇怪了，入门时一定会让你感到晕头转向。

在这里，我们总结一下使用 NASM 的几大理由：

- NASM 和 MASM 接近，入门比较容易。
- NASM 在不同平台下均可使用，可以在 Windows 下编写并调试，然后拿到 Linux 下使用。
- 自带反汇编程序，方便取用。
- 文档丰富，自带的文档有 Word、PDF、CHM 三种版本，检索极为方便。
- 它是免费的。

NASM 安装起来也很简单，在 http://nasm.sourceforge.net/可以找到各个平台下的安装包和相关文档。

2.3　安装虚拟 Linux

编译器已经确定好了，我们需要一个地方来运行它们。

我们需要 Windows 版的 Virtual PC 来运行操作系统。所以，通常情况下，我们仍不会抛开 Windows 平台。如今，我们又用到了 Linux，这意味着我们需要另外一个装有 Linux 的机器，当然，可以是"真的"机器，也可以是"虚拟机"。

毫无疑问，虚拟的 Linux 更合算一些，而且更便于控制。安装它并不复杂，只是有几点需要注意，在这里，我们以 Red Hat 9 为例。

在虚拟机上安装 Red Hat 9 比在真正的机器上安装费时要稍多些，毕竟，虚拟机中间隔了一层，速度受了一定的影响。不过，我们完全可以只安装必不可少的组件，因为我

们仅仅是想用它来编译代码，除了编译器之外，只要它能跟 Windows 通信就足够了。

现在，我们又遇到了一个问题，就是我们的 Linux 如何与 Windows 通信。虽然运行在同一台计算机上，但不幸的是，虚拟机和宿主机之间的通信通常情况下跟两台计算机的情况是一样的。如果虚拟机安装 DOS 或者 Windows 的话会好一些，两者之间可以设置共享文件夹，但现在我们安装的是 Linux，要想跟宿主机通信的话，跟两台计算机的情况是完全一样的。

幸好我们有 Samba，通过它，Linux 和 Windows 之间可以共享文件夹，访问起来就好像 Windows 之间的互相访问一样，若把网络文件夹映射到本地，感觉更是像访问本机一样。

好了，现在我们知道，安装 Linux 时有两个组件是不可或缺的，一个是编译器 GCC 和 NASM，另一个是 Samba 相关组件。下面我们就开始安装。

（1）首先单击 Virtual PC 主界面中的“New PC”按钮，按照提示建立一个新的虚拟机。

（2）按如图 2-28 所示的方式启动它，开始安装 Red Hat 9。Red Hat 安装的欢迎界面如图 2-29 所示。

图 2-28 开始安装 Red Hat Linux

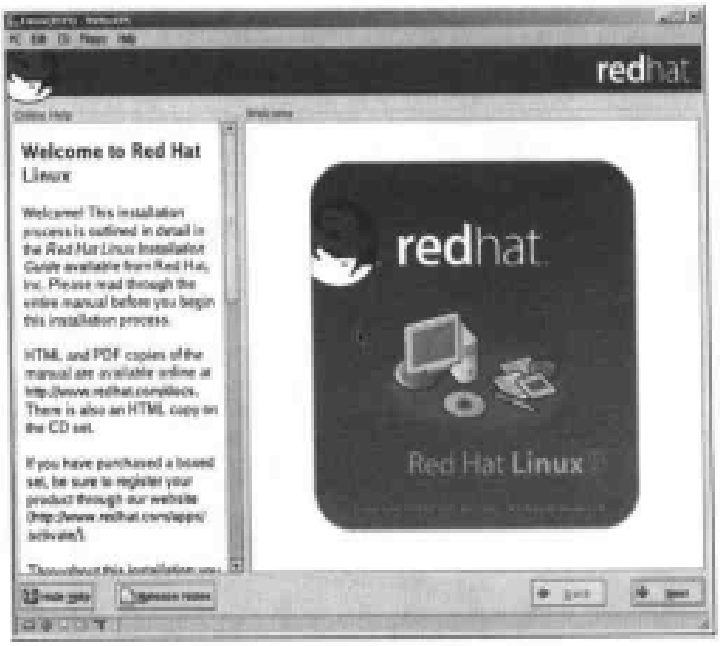

图 2-29　Red Hat 安装之欢迎界面

（3）单击“Next”按钮，出现如图 2-30 所示的界面。

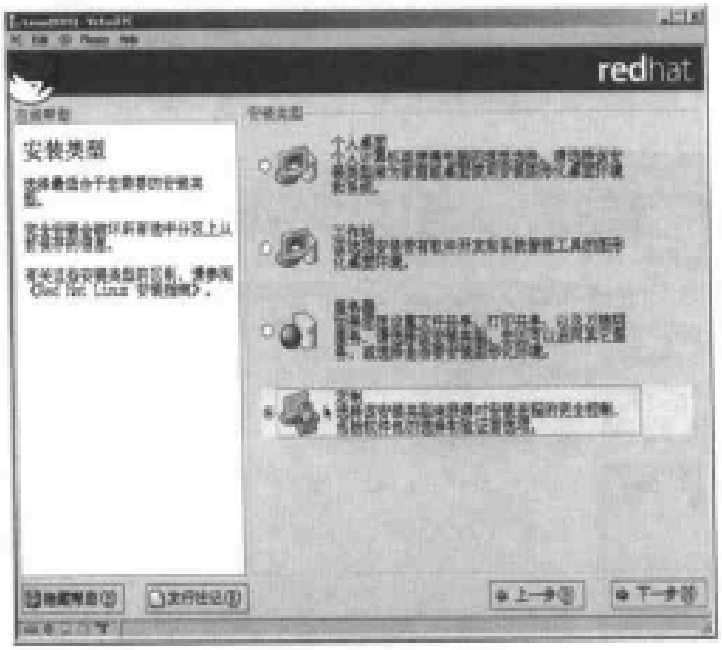

图 2-30　安装类型选择

> 注意：在选择安装类型时，记得选择“定制”。

（4）单击“下一步”按钮，出现如图 2-31 所示的“选择软件包组”界面。在这里，有几项是需要确定选中的。

选择“Windows 文件服务器”复选框，如图 2-31 所示。

图 2-31　选择 Windows 文件服务器

单击“细节”，可以看到，这里的“Windows 文件服务器”指的就是 Samba。如图 2-32 所示。

图 2-32　Windows 文件服务器细节

选择“开发工具”复选框，如图 2-33 所示。

图 2-33　选择开发工具

单击“细节”，出现如图 2-34 所示的界面，可以看到，“开发工具”中不但包含 GCC，而且有 NASM，记得要选中它。

图 2-34　开发工具细节（确认 NASM 被安装）

选择“系统工具”复选框，如图 2-35 所示。

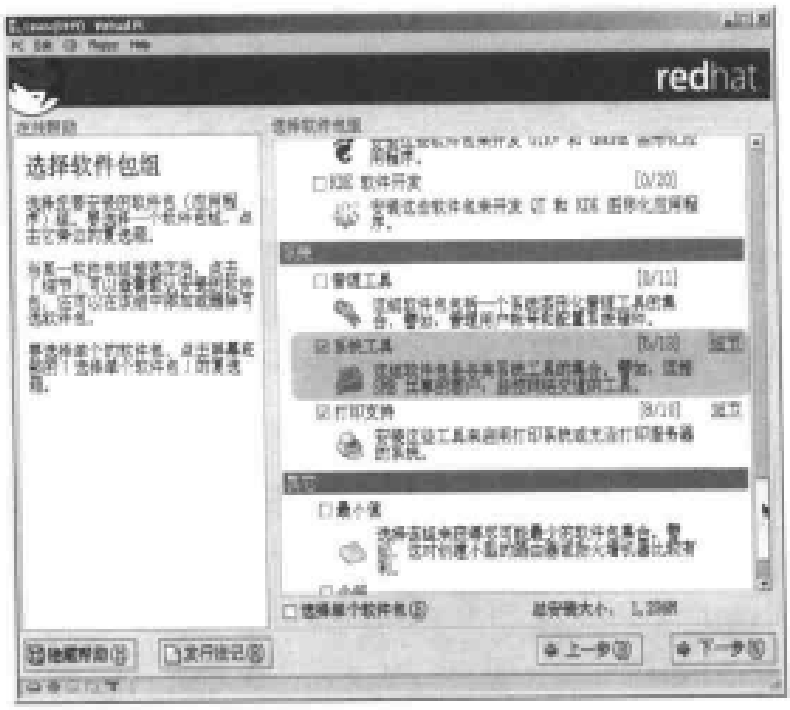

图 2-35　选择系统工具

单击“细节”，出现如图 2-36 所示的界面，可以看出，“系统工具”中包含 samba-client。

图 2-36　选择系统工具细节

对于“其他”各选项，尽可能不选，就我们用到的功能来说，这些选项已经足够了。

（5）单击“下一步”按钮，出现如图 2-37 所示的界面。

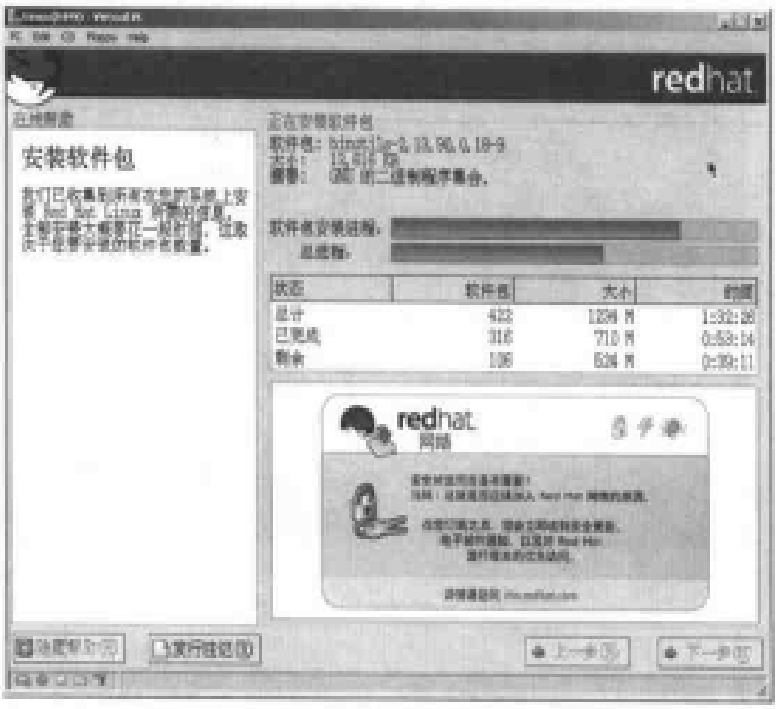

图 2-37　软件包安装进程

（6）安装进程结束后，单击“下一步”按钮，出现如图 2-38 所示的界面。

图 2-38　结束界面

（7）单击“退出”按钮，系统重启，我们就可以使用这个可爱的红帽子了。如图 2-39 所示。

图 2-39 Red Hat 的 login 画面

2.4 在虚拟 Linux 上访问 Windows 文件夹

从 Linux 访问 Windows 文件夹是非常简单的，只需键入下面的命令就可以了。

```
mount -t smbfs -o username=user,password=foobar,ip=192.168.**.**
//HOSTMACHINE_NAME/Tinix /mnt/smb/Tinix
```

其中，username 和 password 必须是宿主机上合法的用户名和密码，而且要有访问共享文件夹 Tinix 的权限。HOSTMACHINE_NAME 是宿主机的计算机名称。注意，逗号之后不能有空格。

我们并不需要配置 Samba 服务器，因为我们不想从 Windows 中访问 Linux，我们只需将 Windows 中的文件夹共享，这样宿主机和虚拟机都可以访问，我们的目的也就达到了。

现在，在/mnt/smb/Tinix 目录下已经能够看到 Windows 下的文件了。

2.5 安装虚拟 PCDOS

我们要写的是基于 x86 平台的 32 位操作系统，这需要我们对保护模式有非常好的了解。而学习保护模式时编写的代码需要依赖运行于实模式的 DOS，而不是 V86 模式下的 DOS。市面上的保护模式教材大多也用 DOS 下的代码作为例子讲解。另外，后面的章节会提到，为了方便调试，Tinix 可以在 DOS 下启动，这大大增加了我们做试验时的灵

活性。

有了以上使用经验，安装 DOS 应该非常容易了，而且，Virtual PC 提供了非常方便的插件，使得没有 Samba 的 DOS 也可以共享宿主机中的文件。

插件安装后，共享文件的方式如下：

（1）打开 Virtual PC 主界面，如图 2-40 所示。

图 2-40　Virtual PC 主界面和已启动的虚拟 DOS

（2）在 Virtual PC 主界面中单击“Settings”按钮，打开如图 2-41 所示的界面。

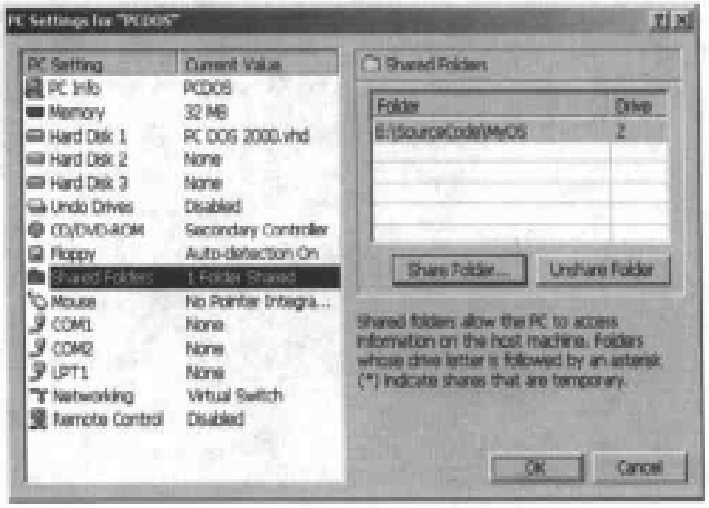

图 2-41　配置界面

（3）在图 2-41 左边列表中双击“Shared Folders”，在右侧单击“Share Folder”按钮，弹出如图 2-42 所示的“浏览文件夹”对话框。

图 2-42　配置 Virtual PC 的共享文件夹

（4）选择要共享的文件和映射到 DOS 中的盘符，这里，我们将“Drive letter”设成“Y”，并共享“C:\Inetpub”，单击“确定”按钮，再单击“确定”按钮，在 DOS 下按图 2-43 所示键入：

```
Y:（回车）
DIR（回车）
```

图 2-43　在 Virtual PC 的虚拟 DOS 中观察 Windows 文件夹内容

由图 2-43 可以看出，这些正是 C 盘相应目录中的那些文件。

有了这个功能，我们在必要时（比如学习保护模式时）可以在 Windows 下编写代码，

并把文件夹共享，然后就可以在虚拟 DOS 中运行了。即便程序崩溃的话，最多也只是虚拟机重启，非常方便。

2.6 其他要素

到目前为止，我们一直在谈论虚拟机，是不是我们所有的工作都是在虚拟机上进行的呢？答案恰恰相反，代码编写工作其实并不需要虚拟机。这也意味着，我们的大部分时间并不是花费在依赖虚拟机的工作中。

现实充满了悖论，需要费很大力气来搭建的，需要重点提及的，恰恰是使用时间最少的。这就好像每天吃饭，为解除饥饿感做出最大贡献的是碗里的米饭，可做熟米饭是如此容易，只需把米放进电饭煲，加上适量的水，按一下开关就可以了，而佐餐的菜肴呢，却要耗费你在厨房的大部分时间。当然，只有米饭的一餐是不太可能的，你也非常乐意花上几个小时来做几个可口的菜肴给自己和家人享用。

世界就是这样，用类似的各种各样的规律保持着那种特有的节奏感，有趣且耐人寻味。

正如馒头、米饭、面条、米线等可供选择，写代码的编辑器同样有许多选择，简单如 Notepad，复杂如 Visual Studio 这样功能强大的 IDE，均可用来编写代码。此外，还有 Edit Plus、Ultra Edit 、Source Insight 等丰富多彩的工具可供使用。相对而言，有人喜欢吃面食，而有人可能更偏爱米饭，同样，有人喜欢 Edit Plus，有人喜欢 Ultra Edit，而有些人则始终坚持使用 Notepad。就笔者本人而言，每一种工具都比较喜欢，它们各有所长，适合于不同的工作。使用什么编辑器完全是个人喜好的问题，不过有一个原则，就是一定要让自己感到得心应手，不会因为这些小问题耽误时间，增添不必要的麻烦。

既然它们各自适合编写不同的代码，我们也完全可以在不同情况下使用不同的工具。编写汇编程序的时候，笔者比较喜欢使用 Edit Plus，它界面简洁，支持关键字颜色，使用也比较灵活，可以自己配置用户工具，用起来像是随心所欲的 IDE。而在编写 C 代码的时候，可选择的工具就更多了，你甚至可以将 Visual Studio 仅仅当做一个编辑器来使用。即便不用它来编译，作为编辑器它也很好用、很强大，不是吗？

2.7 Bochs

我们上面提到的 Virtual PC 和 VMware 都是商业软件，而下面将要介绍的 Bochs 软件则是开源的和免费的。可以从 http://bochs.sourceforge.net/获取它的最新版本和相关信息。

2.7.1 Bochs vs. Virtual PC vs. VMware

不知道开源给你的感觉是什么，对笔者而言，它总是意味着亲切、清新，身上没有讨厌的商业气息。也许你也有同感，如果是的话，可能你已经对本书到现在才提到 Bochs 感到不满。实际上笔者对 Bochs 其实是充满好感的，尤其是它可以对操作系统进行调试，这样方便的特性不能不让我们对它青睐有加。理论上讲，Bochs 可以完全取代 Virtual PC 和 VMware，但以笔者个人的体会，从一个操作系统开发者的角度来看，Bochs 大致有这么几个缺点：

- Virtual PC 有共享文件夹功能，可以方便地读取 host 上的文件，而 Bochs 不可以。当我们学习保护模式时，免不了要用到 DOS，用 Virtual PC 可以在 Windows 下编写代码，通过共享文件夹的方式在 DOS 下执行，而 Bochs 就没这么方便。
- 跟 Virtual PC 和 VMware 比起来，Bochs 的速度要慢得多，笔者曾经试图在 Bochs 上安装 Red Hat 9，但随后就发现这是个愚蠢的决定，太慢了，不久笔者就把它中止掉了。
- Virtual PC 的图形化界面要比 Bochs 好用。

读者可能注意到了上文中“从一个操作系统开发者的角度来看”这一句，之所以这样说，是因为我们这样武断地将 Bochs 拿来跟 Virtual PC 或 Vmware 相比是不公平的。虽然它们模拟同一个操作系统时的输出可能差不多，但从实现机制上讲，它们有巨大的不同。

Bochs 是一个模拟器，它完全模拟 x86 的硬件以及一些外围设备。而 VMware 除了模拟一些特定的 I/O 之外，还可以用它的 x86 运行时引擎来完成其余内容的执行。意思是说，当客户 PC 试图执行一些动作时，VMware 并不是利用自己重造的机制来解释它，而是将它传递给实际的硬件。在这里，VMware 的工作更应称为“虚拟（virtualize）”而不是“模拟（emulate）”。Virtual PC 的原理介于两者之间，它的一些部分是模拟出来的，另一部分则是虚拟出来的。Virtual PC 能够实现共享文件夹也是因为同样的原因。

由于 Bochs 在模拟一台真正的机器的硬件，所以使得它比 VMware 和 Virtual PC 都慢得多。

笔者曾经将 Tinix 分别用 Bochs 2.1.1、Virtual PC 5.0、VMware 3.2.0 和真实的机器进行比较测试，发现在 Virtual PC 上运行的效果跟真实的机器居然差不多，而 Bochs 和 VMware 则慢得多，这可能跟它们实现机制的差异以及 Tinix 自身的特点有关。不过，Virtual PC 的确因为速度、操作方便性等特点具备很强的吸引力，以至于本书最终将它而不是开源免费的 Bochs 作为首选。

那么 Bochs 是不是就被我们抛弃了呢？当然不是，因为 Bochs 的调试功能实在太令人兴奋了。当我们的操作系统出现问题时，Bochs 就变得无法替代。下面，我们就来看一

下，Bochs 到底如何使用。

2.7.2　Bochs 的使用方法

我们先来一点感性认识，首先从 http://bochs.sourceforge.net/获取 Bochs 的安装程序。这里以 Bochs 2.1.1 为例，安装完成后，在菜单中可以找到快捷方式 Linux Demo in Bochs 2.1.1，双击它，几秒钟之后，如图 2-44 所示的画面出现了。

图 2-44　用 Bochs 启动 Linux

感觉怎么样？虽然你已经熟悉了虚拟机这种东西，看到这样的画面还是感到很有趣，不是吗？这是一个非常小的 Linux，输入 root，然后按回车键，就可以登录进去使用了。如图 2-45 所示。

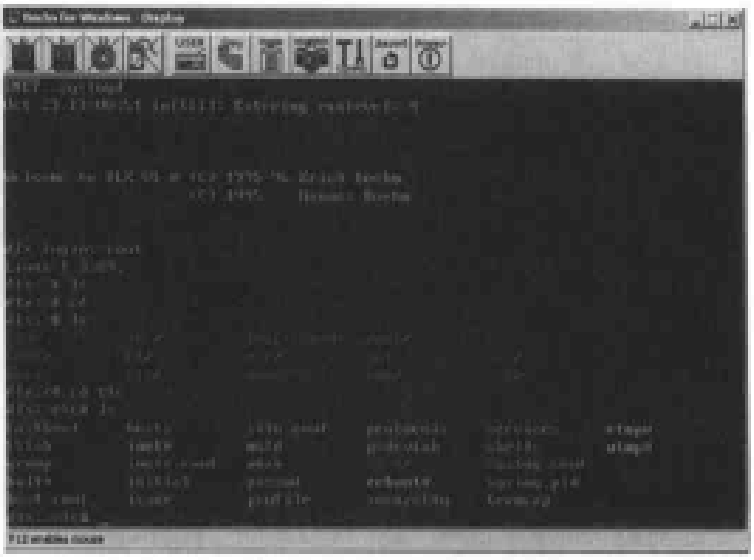

图 2-45　登录到 root

新奇之后，让我们来看看这奇妙的过程是怎么发生的。

首先在桌面上找到快捷方式 Linux Demo in Bochs 2.1.1，看看它的目的地：D:\Program Files\Bochs-2.1.1\dlxlinux\run.bat。现在，就让我们看看 run.bat 的内容：

```
cd "d:\Program Files\Bochs-2.1.1\dlxlinux"
..\bochs -q -f bochsrc.bxrc
```

原来 Bochs 通过 bochs -q -f bochsrc.bxrc 这样的命令行来启动，那么不用说，启动的各种参数一定就装在 bochsrc.bxrc 中，下面就让我们来看一下（为节省篇幅，省略了部分内容）：

代码 2-1 bochsrc.bxrc（请参考\chapter2\bochsrc.bxrc）

```
# how much memory the emulated machine will have
megs: 32

# filename of ROM images
romimage: file=$BXSHARE/BIOS-bochs-latest, address=0xf0000
vgaromimage: $BXSHARE/VGABIOS-elpin-2.40

# what disk images will be used
floppya: 1_44=floppya.img, status=inserted
floppyb: 1_44=floppyb.img, status=inserted

# hard disk
ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata0-master: type=disk, path="hd10meg.img", cylinders=306, heads=4,
spt=17

# choose the boot disk.
boot: c
......
# where do we send log messages?
log: bochsout.txt

# disable the mouse, since DLX is text only
mouse: enabled=0

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

内容看起来虽然不少，但是除去以#开头的注释之外，实际起作用的内容并不多，而且凭借字面意思，还是比较容易理解的。

你一定蠢蠢欲动地想把你的 Boot Sector 拿到这里试一下了，好的，我们就在 Bochs-2.1.1\目录下新建一个目录 Tinix\，然后把 Bochs-2.1.1\下面的内容全部复制过来，再把我们之前已经做好的 Tinix.img 复制到这里。好了，打开 bochsrc.bxrc，将上面代码中用阴影标出的两行改为：

```
floppya: 1_44=Tinix.img, status=inserted
```

以及

```
boot: a
```

然后修改一下 run.bat：

```
cd "d:\Program Files\Bochs-2.1.1\Tinix"
..\bochs -q -f bochsrc.bxrc
```

运行，结果如图 2-46 所示。

图 2-46　用 Bochs 运行引导扇区

画面有点乱，那是因为 Boot Sector 没有进行任何的清屏操作，不过毫无疑问，红色的“Hello, OS world”已经出现在眼前，这表明运行成功了，整个操作就是这么简单。在本书附送光盘中的文件夹“\chapter2”中有相应的 bochsrc.bxrc 和 run.bat 文件供读者参考。

当然，Bochs 的使用不止这么简单，就让我们在实际应用过程中慢慢学习吧。

2.7.3　用 Bochs 进行调试

用 Bochs 可以对操作系统进行调试，对于这一点，我们将在 5.4.5.2 节中再具体介绍。

2.7.4 在 Linux 上开发

或许你是个不折不扣的开源爱好者，对 Linux 无比热爱，并且决不肯对任何人妥协，没问题，运用 Bochs 使得在 Linux 上的开发变得与刚才介绍的情况非常类似。而且可以想像，在真正的 Linux 上编译我们的代码一定是非常快的。

为简单起见，我们从 http://bochs.sourceforge.net/处直接获得 Bochs 的 RPM 包，然后安装，如图 2-47 所示。

图 2-47 在 Linux 中安装 Bochs 2.1.1

然后修改文件 bochsrc.bxrc，涉及到的主要是相关路径：

代码 2-2 在 Linux 下使用的 bochsrc.bxrc

```
megs: 32

romimage:  file=/usr/local/share/bochs/BIOS-bochs-latest,  address
=0xf0000
vgaromimage: /usr/local/share/bochs/VGABIOS-elpin-2.40

floppya: 1_44=tinix.img, status=inserted
floppyb: 1_44=floppyb.img, status=inserted

ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata0-master: type=disk, path="hd_dos.img", cylinders=101, heads=16,
spt=63
```

```
boot: a

log: bochsout.txt

mouse: enabled=0

keyboard_mapping:   enabled=1,   map=/usr/local/share/bochs/keymaps/
x11-pc-us.map
```

这样就可以运行了，运行结果如图 2-48 所示。

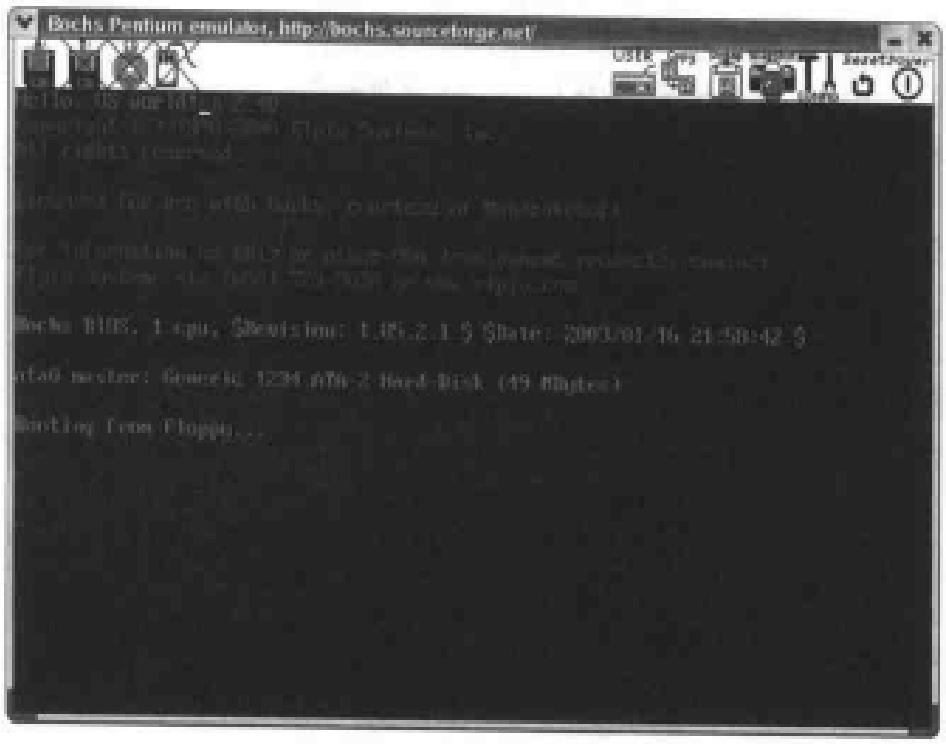

图 2-48　用 Bochs 运行引导扇区（在 Linux 平台下）

可以看到，与在 Windows 下的运行几乎是一摸一样的，不是吗？

另外，在 Linux 中使用软盘映像也非常方便，通过类似这样的命令就可以了：

```
mount /home/TINIX.IMG /mnt/floppy -o loop
```

总结一下，在 Linux 上开发可以通过这样的方式：

- GCC 编译生成内核。
- 用 mount 命令将.IMG 文件挂载到文件系统。
- 将内核复制到.IMG 文件中。
- 用 Bochs 运行 Tinix（可能需要先从文件系统中卸载.IMG 文件）。

2.8 总结与回顾

我们已经介绍了这么多，从 Virtual PC 到 Bochs，从 Windows 到 DOS，再到 Linux，其实，简言之，在最简陋的条件下，我们只要有一个 Linux 平台和一张软盘就够了，其他什么工具也不用，只是调试起来比较麻烦罢了。介绍这些工具的目的只不过是让开发环境用起来比较顺手，使我们的精力能够专注到问题的重点上，而不至于让一些操作上的细节占用太多的时间。无论哪一种工具和方法，只要能让你感到得心应手就够了。

对笔者而言，比较习惯于这样进行操作系统的开发：

- 在 Windows 下编写代码，使用 Edit Plus 等编辑器。
- 使用 Linux 的虚拟机编译内核和生成操作软盘映像。
- 必要时使用安装了 DOS 的虚拟机来调试程序以及操作软盘映像。
- 安装一个 Bochs，必要的时候用它来调试。
- 使用其他必要的工具，比如版本控制工具（VSS 或者 CVS）。

有了这些好用的工具，我们就真正做到了“武装到牙齿”，从此尽可奋勇杀敌，一切无忧！那么，我们是不是马上就可以冲锋陷阵了呢？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储备。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。

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。

第 3 章

保护模式（Protect Mode）

学而不思则惘，思而不学则怠。

——孔子

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，在“学”和“思”中不断前行。

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。

代码是这样的：

代码 3-1 （节自\chapter3\a\pmtest1.asm）

```
; ==========================================
; pmtest1.asm
```

```
; 编译方法: nasm pmtest1.asm -o pmtest1.com
; ==========================================
%include    "pm.inc"    ; 常量、宏，以及一些说明
org 0100h
    jmp LABEL_BEGIN

[SECTION .gdt]
; GDT
LABEL_GDT:          Descriptor 0, 0, 0                  ; 空描述符
LABEL_DESC_CODE32:  Descriptor 0, SegCode32Len - 1, DA_C + DA_32
                                                        ; 代码段，32 位
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          ; 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 做准备
    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

    ; 真正进入保护方式
    jmp dword SelectorCode32:0      ; 执行这一句会把 SelectorCode32
                                    ; 装入 cs，并跳转到
                                    ; SelectorCode32:0 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 * 10 + 0) * 2  ; 屏幕第 10 行，第 0 列。
    mov ah, 0Ch                 ; 0000: 黑底    1100: 红字
    mov al, 'P'
    mov [gs:edi], ax

    ; 到此停止
```

```
    jmp $

SegCode32Len    equ $ - LABEL_SEG_CODE32
; 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，然后很快地翻页，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。

你可能注意到了，在最开头处的注释中，这个程序被编译成了.COM 文件，最简单的二进制文件。这意味着，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，首先说说[SECTION .gdt]这个段，其中的 Descriptor 是在 pm.inc 中定义的宏：

代码 3-2 （节自\chapter3\a\pm.inc）

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

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。

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 的长度，后 4 字节是 GDT 的基地址。

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：

```
jmp SelectorCode32:0
```

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。

首先按照注释中所说的办法编译：

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

然后在 Virtual PC 中按照本书第 2 章 2.5 节的方法将包含 pmtest1.com 的文件夹共享，并且运行它，结果如图 3-1 所示。

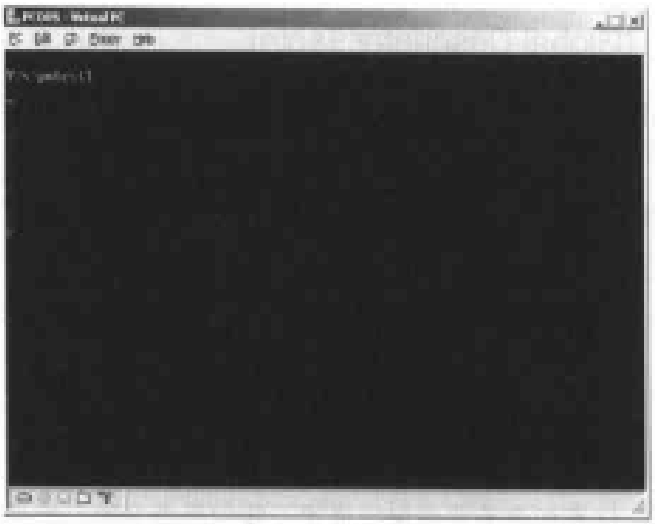

图 3-1　pmtest1.com 执行结果

可以看到，程序打印了一个红色的字符 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 2 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 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“SEG:OFFSET”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，我们来看一下如图 3-2 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。

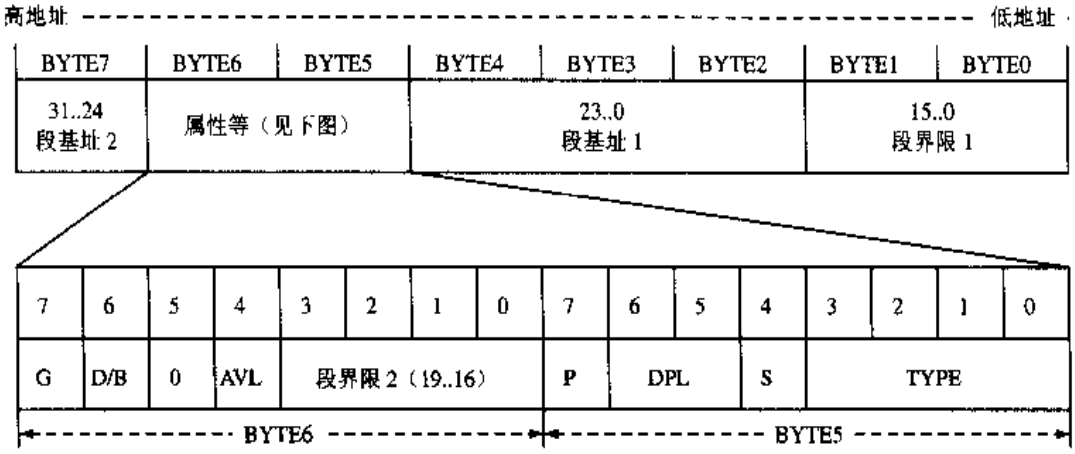

图 3-2　代码段和数据段描述符

好了，我们回头再来看看代码，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。现在，我们来看看 DESC_VIDEO，它的段基址是 0B8000h。顾名思义，这个描述符指向的正是显存。

现在我们已经知道了，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，在段[SECTION.s32]中有两句代码是这样的：

```
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：

```
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 3-3 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
描述符索引													TI	RPL	

图 3-3 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。

在程序快结束的位置，你可以找到这一行：

```
mov [gs:edi], ax
```

看到这里，读者肯定已经明白了，gs 的值是 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，整个的寻址方式可以用如图 3-4 所示的示意图来表示。

这是个示意图，真实的描述符中段基址以及段偏移等内容在描述符中的位置不是像图中这样安排的。

可能你注意到了，上图中“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。

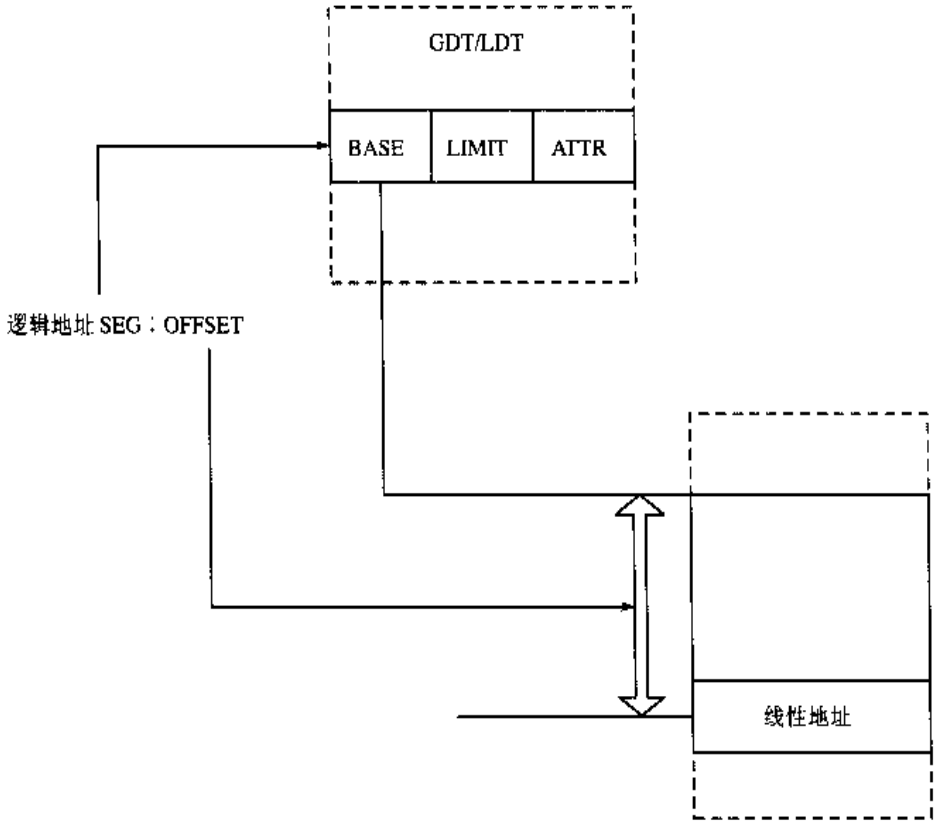

图 3-4　段式寻址示意图

3.1.2　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，然后执行了的指令：

```
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 3-5 所示。

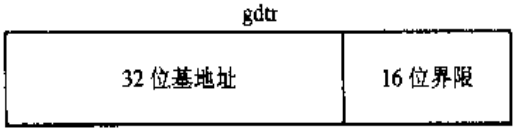

图 3-5　gdtr 结构示意图

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 是用 SEG:OFFSET 这样的模式分段的，所以，它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，我决定只使用通过操作端口 92h 来实现这一种方式，正如代码 3-1 中那样，其他方式略过不提。你需要知道的是，这不是惟一的方法，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 3-6 所示。

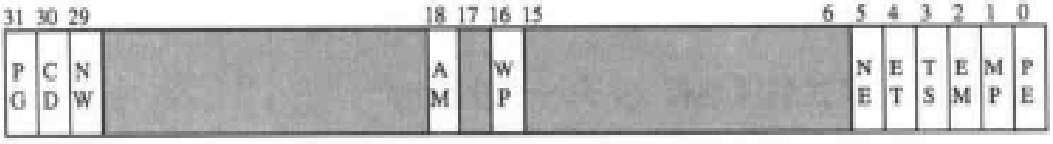

图 3-6 cr0（灰色表示保留位）

它的第 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 jmp 指令：

```
jmp dword SelectorCode32:0
```

根据上面解释过的寻址机制，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!
```

因为偏移地址应该是 32 位的，这样编译出来的只是 16 位的代码。假设目标地址的偏移不是 0，而是一个 32 位的值，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 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，回过头看看刚刚的程序，是不是觉得非常简单呢？

3.1.3　描述符属性

在刚才的程序中，涉及到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。

在读表 3-1 时，建议参考图 3-2。

表 3-1　描述符属性

属　性	概括说明	具体说明		
P	存在（Present）位	P=1	段在内存中存在	
		P=0	段在内存中不存在	
DPL（2 bits）	表示描述符特权级（Descriptor Privilege Level）	它规定了段的特权级 特权级范围：0~3		
S	说明描述符的类型	S=1	数据段和代码段描述符	
		S=0	系统段描述符和门描述符	
TYPE	描述符类型	描述符类型	TYPE 值	说明
		数据段描述符 S=1	0	只读

续表

属 性	概括说明	具体说明		
TYPE	描述符类型		1	只读、已访问
			2	读/写
			3	读/写、已访问
			4	只读、向下扩展
			5	只读、向下扩展、已访问
			6	读/写、向下扩展
			7	读/写、向下扩展、已访问
		代码段描述符 S=1	8	只执行
			9	只执行、已访问
			A	执行/读
			B	执行/读、已访问
			C	只执行、一致码段
			D	只执行、一致码段、已访问
			E	执行/读、一致码段
			F	执行/读、一致码段、已访问
		系统段描述符和门描述符 S=0	0	<未定义>
			1	可用 286TSS
			2	LDT
			3	忙的 286TSS
			4	286 调用门
			5	任务门
			6	286 中断门
			7	286 陷阱门
			8	<未定义>
			9	可用 386TSS
			A	<未定义>
			B	忙的 386TSS
			C	386 调用门
			D	<未定义>
			E	386 中断门
			F	386 陷阱门
G	段界限粒度（Granularity）位	G=0	段界限粒度为字节	
		G=1	段界限粒度为 4KB	
		在可执行代码段描述符中，这一位叫做 D 位	D=1	默认情况下，指令使用 32 位地址及 32 位或 8 位操作数
			D=0	默认情况下，使用 16 位地址及 16 位或 8 位操作数

续表

属　性	概括说明	具体说明		
D/B	这一位比较复杂，分三种情况	在向下扩展数据段的描述符中，这一位叫做 B 位	D=1	表示段的上部界限为 4GB
			D=0	表示段的上部界限为 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	D=1	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	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	保留并可用位	保留位，可以被系统软件使用。		

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。

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

上面的属性的字面意思大部分都容易理解，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，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 表明这个段是只执行的段。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。总之，这个段是存在的只执行的 32 位代码段，DPL 为 0。

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 的内存的访问能力。

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，这显然太粗暴了。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，所以，这样暂时的粗暴是值得的。下面就让程序优雅起来，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（请参考附书光盘中的 pmtest2.asm），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。增加的这两个段对应的描述符如下：

```
LABEL_DESC_DATA:    Descriptor          0,    DataLen - 1, DA_DRW
LABEL_DESC_TEST:    Descriptor  0500000h,         0ffffh, DA_DRW
```

选择子也没什么离奇：

```
SelectorData equ    LABEL_DESC_DATA - LABEL_GDT
SelectorTest equ    LABEL_DESC_TEST - LABEL_GDT
```

在段[SECTION .s32]的开头，初始化 ds、es 和 gs，让 ds 指向新增的数据段，es 指向新增的指向 5MB 内存的段，gs 指向显存。

代码 3-3　（节自\chapter3\b\pmtest2.asm）

```
mov ax, SelectorData
mov ds, ax              ; 数据段选择子
mov ax, SelectorTest
mov es, ax              ; 测试段选择子
mov ax, SelectorVideo
mov gs, ax              ; 视频段选择子
```

在显示一行字符串之后，开始读写大地址内存：

```
call    TestRead
call    TestWrite
call    TestRead
```

由于要读两次相同的内存，我们把它写进一个函数 TestRead，写内存的内容也写进函数 TestWrite，它们的内容是这样的：

代码 3-4　（节自\chapter3\b\pmtest2.asm）

```
; ------------------------------------------------------------
TestRead:
    xor     esi, esi
    mov     ecx, 8
.loop
```

```
    mov     al, [es:esi]
    call    DispAL
    inc     esi
    loop    .loop

    call    DispReturn
    ret
; TestRead 结束-----------------------------------------------

; ---------------------------------------------------------------
TestWrite:
    push    esi
    push    edi
    xor     esi, esi
    xor     edi, edi
    mov     esi, OffsetPMMessage    ; 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

    ret
; DispAL 结束-------------------------------------------------

; ----------------------------------------------------------
DispReturn:
    push    eax
    push    ebx
    mov     eax, edi
    mov     bl, 160
    div     bl
    and     eax, 0FFh
    inc     eax
    mov     bl, 160
    mul     bl
    mov     edi, eax
    pop     ebx
    pop     eax
```

```
    ret
; DispReturn 结束----------------------------------------------
```

可以看到，在 TestRead 中还调用了 DispAL 和 DispReturn 这两个函数，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。

在 TestWrite 中用到一个常量 OffsetPMMessage，下面是定义它的数据段：

代码 3-5 （节自\chapter3\b\pmtest2.asm）

```
[SECTION .data1]    ; 数据段
LABEL_DATA:
SPValueInRealMode   dw  0
; 字符串
PMMessage:      db  "In Protect Mode now. ^-^", 0   ; 进入保护模式后
                                                    ; 显示此字符串
OffsetPMMessage equ PMMessage - $$
StrTest:        db  "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 0
OffsetStrTest   equ StrTest - $$
DataLen         equ $ - LABEL_DATA
; END of [SECTION .data1]
```

我们用到这个字符串的时候并没有用标号 PMMessage，而是又定义了一个 Offset PMMessage，它等于 PMMessage - $$。还记得我们在第 1 章中提到过的$$的含义吗？它代表当前节(section)的开始处的地址。也就是说，PMMessage – $$表示字符串 PMMessage 相对于本节的开始处（即 LABEL_DATA 处）的偏移，再联系我们定义的段基址：

代码 3-6 （节自\chapter3\b\pmtest2.asm）

```
; 初始化数据段描述符
xor eax, eax
mov ax,  ds
shl eax, 4
add eax, LABEL_DATA
mov word [LABEL_DESC_DATA + 2], ax
shr eax, 16
mov byte [LABEL_DESC_DATA + 4], al
mov byte [LABEL_DESC_DATA + 7], ah
```

从这里我们看出，数据段的基址便是 LABEL_DATA 的物理地址。于是 Offset PMMessage 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 PMMessage - $$中用到的$$。当然，它不是没有替代品，而是这样做思路会比较清晰。OffsetStrTest 的情形与此类似。

另外，由于在保护模式下用到了堆栈，我们还需要建立一个堆栈段：

代码 3-7　（节自\chapter3\b\pmtest2.asm）

```
[SECTION .gs]
ALIGN   32
[BITS   32]
LABEL_STACK:
    times 512 db 0
TopOfStack  equ $ - LABEL_STACK
```

在 GDT 中增加一个描述符：

```
LABEL_DESC_STACK: Descriptor 0, TopOfStack, DA_DRWA + DA_32
```

注意：在这里加上了 DA_32，表明它是 32 位的堆栈段。

在[SECTION .s32]中加入改变 ss 和 esp 的代码：

```
mov ax, SelectorStack
mov ss, ax          ; 堆栈段选择子
mov esp, TopOfStack
```

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：

```
LABEL_DESC_NORMAL: Descriptor 0, 0ffffh, DA_DRW
```

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：

代码 3-8 （节自\chapter3\b\pmtest2.asm）

```
[SECTION .s16code]
ALIGN   32
[BITS   16]
LABEL_SEG_CODE16:
    mov ax, SelectorNormal
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax

    mov eax, cr0
    and al,  11111110b
    mov cr0, eax

LABEL_GO_BACK_TO_REAL:
    jmp 0:LABEL_REAL_ENTRY

Code16Len   equ $ - LABEL_SEG_CODE16
```

这个段是通过[SECTION .s32]中的 jmp SelectorCode16:0 跳进来的，这句跳转不必多讲。让我们来看这个段，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 3-9 （节自\chapter3\b\pmtest2.asm）

```
mov ax, cs
mov ds, ax
mov es, ax
mov ss, ax
mov sp, 0100h

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 3-7 所示。

BYTE1	BYTE2	BYTE3	BYTE4	BYTE5
OEAh	Offset		Segment	

图 3-7　实模式下长跳转指令图示

上图告诉我们，LABEL_GO_BACK_TO_REAL+3 恰好就是 Segment 的地址，而 mov [LABEL_GO_BACK_TO_REAL+3], ax 这一语句执行之前 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

```
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：

代码 3-10　（节自\chapter3\b\pmtest2.asm）

```
LABEL_REAL_ENTRY:
    mov ax, cs
    mov ds, ax
    mov es, ax
    mov ss, ax

    mov sp, [SPValueInRealMode]

    in  al, 92h          ; ┐
    and al, 11111101b    ; ├ 关闭 A20 地址线
    out 92h, al          ; ┘

    sti                  ; 开中断

    mov ax, 4c00h        ; ┐
    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 3-8 所示。

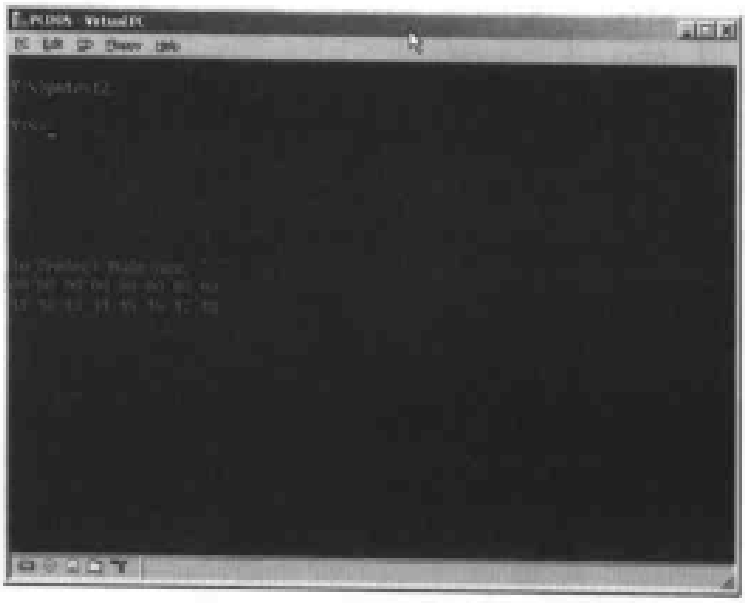

图 3-8 pmtest2.com 的执行结果

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。而且，十六进制数的 41、42、43、…、48 也恰好就是 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，无须重启计算机就可以方便地进行其他操作。

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：

代码 3-11 （节自\chapter3\c\pmtest3.asm）

```
[SECTION .gdt]
```

```
.............
LABEL_DESC_LDT:     Descriptor    0,   LDTLen - 1, DA_LDT    ; LDT
.............
; GDT 结束

.............

; GDT 选择子
.............
SelectorLDT     equ LABEL_DESC_LDT       - LABEL_GDT
.............
; END of [SECTION .gdt]

.............

[SECTION .s16]
.............
    ; 初始化 LDT 在 GDT 中的描述符
    xor eax, eax
    mov ax,  ds
    shl eax, 4
    add eax, LABEL_LDT
    mov word [LABEL_DESC_LDT + 2], ax
    shr eax, 16
    mov byte [LABEL_DESC_LDT + 4], al
    mov byte [LABEL_DESC_LDT + 7], ah

    ; 初始化 LDT 中的描述符
    xor eax, eax
    mov ax, ds
    shl eax, 4
    add eax, LABEL_CODE_A
    mov word [LABEL_LDT_DESC_CODEA + 2], ax
    shr eax, 16
    mov byte [LABEL_LDT_DESC_CODEA + 4], al
    mov byte [LABEL_LDT_DESC_CODEA + 7], ah

    .............
    ; 下面显示一个字符串
    .............
.2: ; 显示完毕
```

```
    call    DispReturn
    ; Load LDT
    mov     ax, SelectorLDT
    lldt    ax

    jmp     SelectorLDTCodeA:0  ; 跳入局部任务

    ...........
SegCode32Len    equ $ - LABEL_SEG_CODE32
; END of [SECTION .s32]

    ...........

; LDT
[SECTION .ldt]
ALIGN   32
LABEL_LDT:
LABEL_LDT_DESC_CODEA:  Descriptor  0, CodeALen - 1, DA_C + DA_32
LDTLen      equ $ - LABEL_LDT
; LDT 选择子
SelectorLDTCodeA  equ   LABEL_LDT_DESC_CODEA  - LABEL_LDT + SA_TIL
; END of [SECTION .ldt]


; CodeA (LDT, 32 位代码段)
[SECTION .la]
ALIGN   32
[BITS   32]
LABEL_CODE_A:
    mov ax, SelectorVideo
    mov gs, ax                      ; 视频段选择子(目的)

    mov edi, (80 * 12 + 0) * 2  ; 屏幕第 10 行, 第 0 列
    mov ah, 0Ch                     ; 0000: 黑底    1100: 红字
    mov al, 'L'
    mov [gs:edi], ax

    ; 准备经由 16 位代码段跳回实模式
    jmp SelectorCode16:0
CodeALen    equ $ - LABEL_CODE_A
; 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[SECTION .s32]中的这几句：

```
mov     ax, SelectorLDT
lldt    ax
jmp     SelectorLDTCodeA:0  ; 跳入局部任务
```

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 3-3 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 3-11 中用到 SelectorLDTCodeA 时，系统将会从 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 3-9 所示。

图 3-9　pmtest3.com 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

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 3-10 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）。

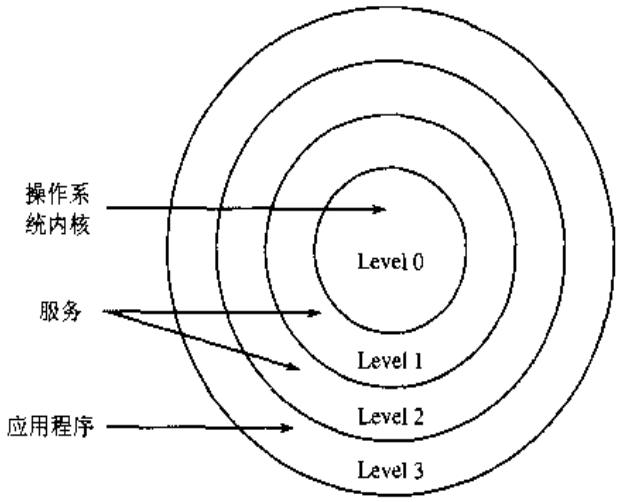

图 3-10　特权级

这里，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

处理器通过识别下面 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。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），而不是更低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？

就像图 3-11 所示的这样，Virtual PC 弹出一个对话框，系统崩溃了。原因就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。

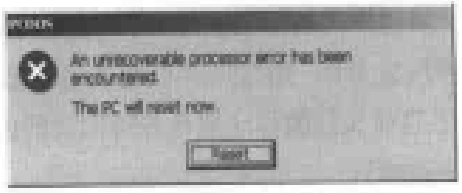

图 3-11　系统崩溃后 Virtual PC 弹出的对话框

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、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：

- 目标操作数包含目标代码段的段选择子。
- 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。
- 目标操作数指向一个包含目标代码段选择子的 TSS。
- 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。

1．通过 jmp 或 call 进行直接转移

我们在 3.1.3 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。

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 3-12 所示。

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 3.1.3 节里讲到的是一致的。

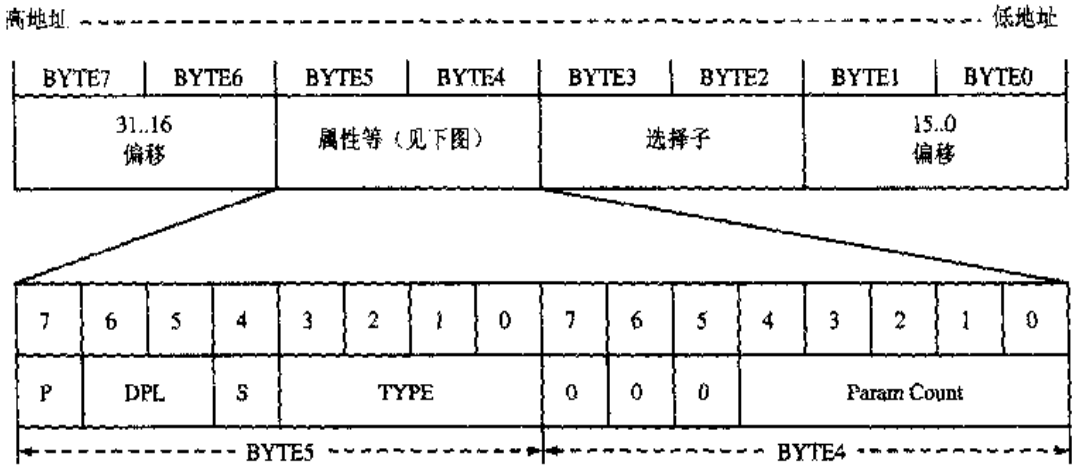

图 3-12 门描述符（Gate Descriptor）

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（请参考附书光盘中的 pmtest4.asm）：

代码 3-12 （节自\chapter3\d\pmtest4.asm）

```
[SECTION .sdest]; 调用门目标段
[BITS   32]

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

    mov edi, (80 * 13 + 0) * 2  ; 屏幕第 13 行，第 0 列
    mov ah, 0Ch                 ; 0000: 黑底    1100: 红字
    mov al, 'C'
```

```
    mov [gs:edi], ax

    retf

SegCodeDestLen  equ $ - LABEL_SEG_CODE_DEST
; 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：

```
LABEL_DESC_CODE_DEST: Descriptor 0,SegCodeDestLen - 1, DA_C + DA_32
```

同时加入初始化这个描述符的代码：

代码 3-13　（节自\chapter3\d\pmtest4.asm）

```
xor eax, eax
mov ax, cs
shl eax, 4
add eax, LABEL_SEG_CODE_DEST
mov word [LABEL_DESC_CODE_DEST + 2], ax
shr eax, 16
mov byte [LABEL_DESC_CODE_DEST + 4], al
mov byte [LABEL_DESC_CODE_DEST + 7], ah
```

这段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，到时为节省篇幅，类似代码将略过不提。

目标代码段的描述符如下：

```
SelectorCodeDest    equ     LABEL_DESC_CODE_DEST - LABEL_GDT
```

好了，现在添加调用门：

```
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　（节自\chapter3\d\pm.inc）

```
%macro Gate 4
    dw  (%2 & 0FFFFh)                       ; 偏移 1      (2 字节)
    dw  %1                                  ; 选择子      (2 字节)
    dw  (%3 & 1Fh) | ((%4 << 8) & 0FF00h)   ; 属性        (2 字节)
```

```
    dw  ((%2 >> 16) & 0FFFFh)                ; 偏移 2      (2 字节)
%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，现在我们的调用门准备就绪了，它指向的位置是 SelectorCodeDest:0，即标号 LABEL_SEG_CODE_DEST 处的代码。

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。

代码 3-15 （节自\chapter3\d\pmtest4.asm）

```
; Load LDT
mov     ax, SelectorLDT
lldt    ax

call    SelectorCallGateTest:0

jmp     SelectorLDTCodeA:0  ; 跳入局部任务
```

它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，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 3-13 所示。

图 3-13 pmtest4.com 的执行结果

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。其实，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
```

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. 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 3-14 所示。

图 3-14 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，它相当于代码 3-16 的执行：

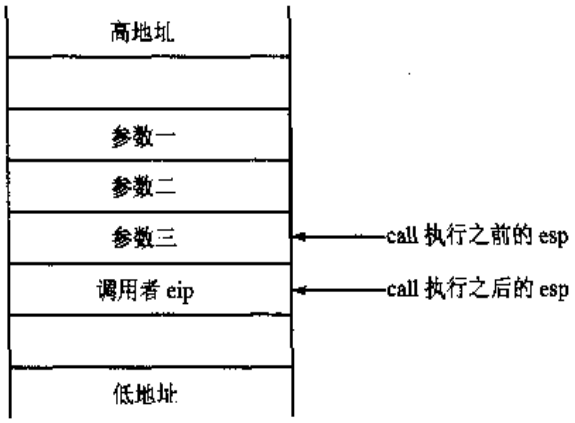

图 3-14 短调用时堆栈示意

代码 3-16 与图 3-14 对应的代码

```
push    param1
push    param2
push    param3
```

```
        call        foo
        nop
        ......
    foo:
        ......
        ret     3
```

图 3-14 中的调用者 eip 对应这里的 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 3-15 所示。

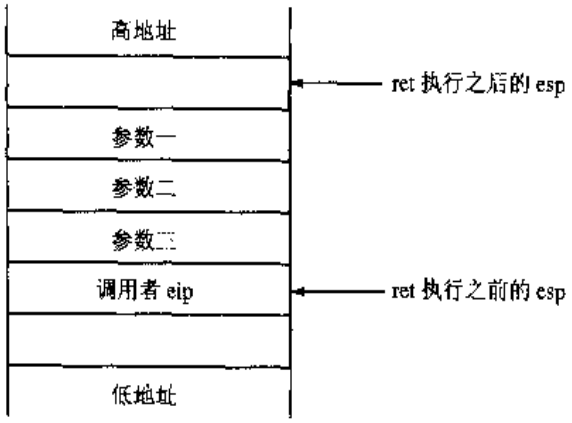

图 3-15　短调用返回时堆栈示意（通过带参数的 ret 指令）

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 3-16 所示。

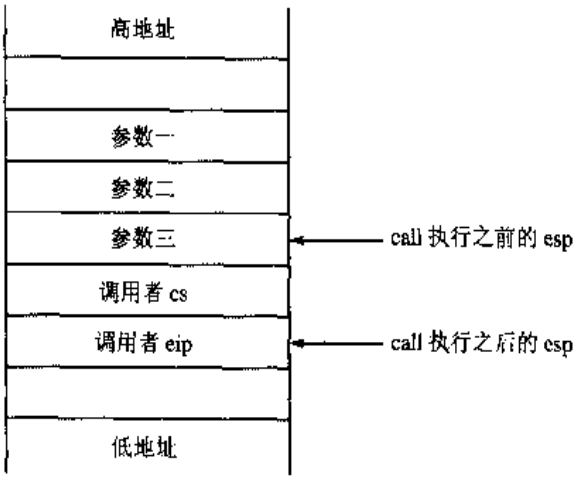

图 3-16　长调用时堆栈示意

相应地，带参数的 ret 指令执行前后的情形如图 3-17 所示。

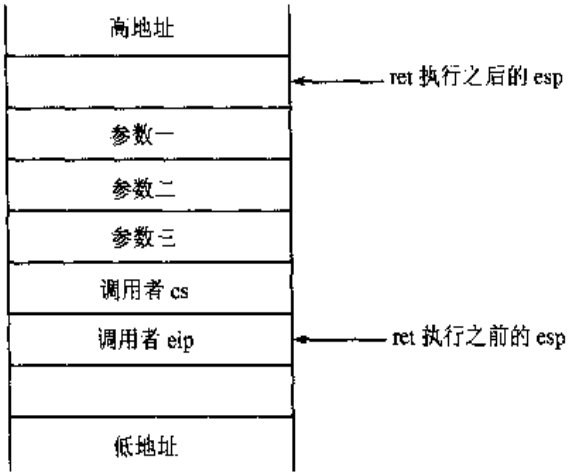

图 3-17 长调用返回时堆栈示意（通过带参数的 ret 指令）

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 3-18 所示。

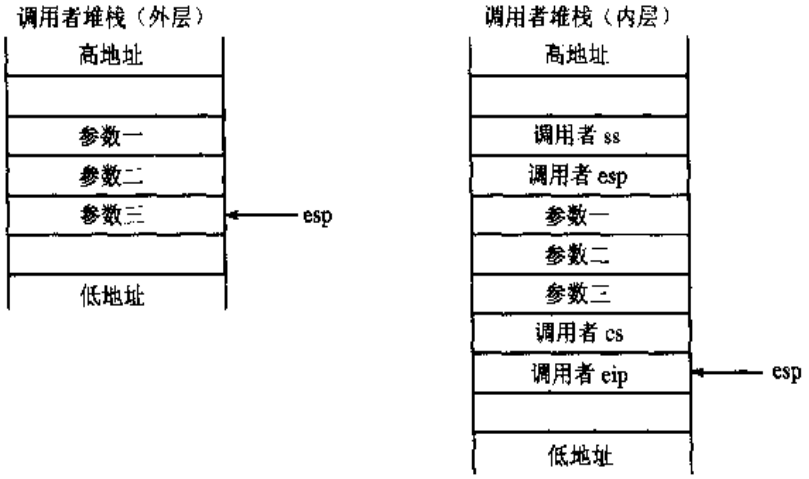

图 3-18 有特权级变换的转移时堆栈变化

这里，我们涉及到两个堆栈。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，这里涉及到一样新事物 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 3-19 所示。

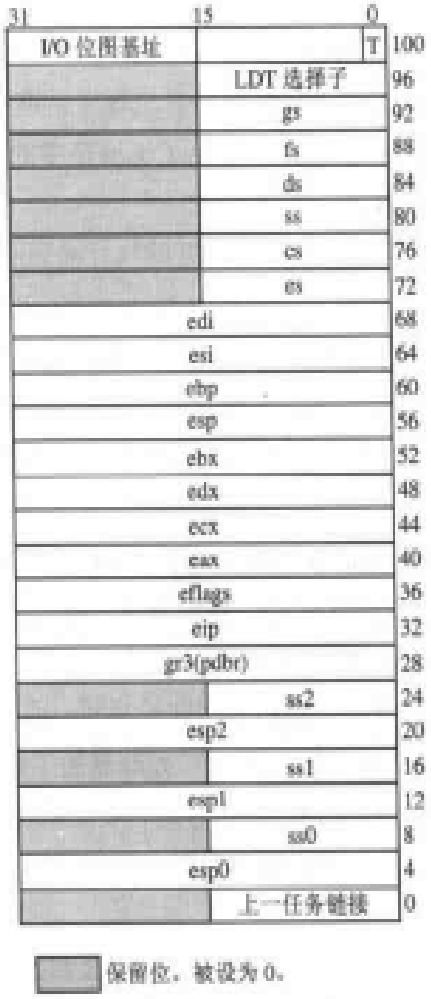

图 3-19　32 位 TSS（Task-State Segment）

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 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 3-19 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 3-14 和图 3-15、图 3-16 和图 3-17 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 3-20 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。

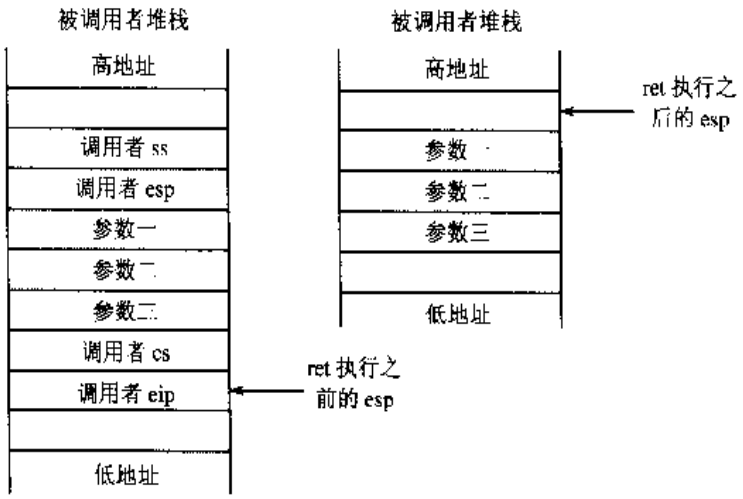

图 3-20　有特权级变换的长调用返回时堆栈示意（通过带参的 ret 指令）

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 3-21 所示。

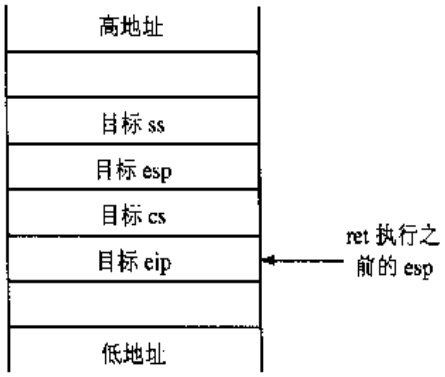

图 3-21　ret 指令执行前的堆栈

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（请参考附书光盘中的 pmtest5.asm）。如上图所示，我们至少要添加一个 ring3 的代码段和一个 ring3 的堆栈段。首先添加一个代码段：

代码 3-17　（节自\chapter3\e\pmtest5.asm）

```
; CodeRing3
[SECTION .ring3]
ALIGN   32
[BITS   32]
LABEL_CODE_RING3:
    mov ax, SelectorVideo
```

```
    mov gs, ax                      ; 视频段选择子（目的）

    mov edi, (80 * 14 + 0) * 2  ; 屏幕第 14 行，第 0 列
    mov ah, 0Ch                     ; 0000: 黑底    1100: 红字
    mov al, '3'
    mov [gs:edi], ax

    jmp $
SegCodeRing3Len equ $ - LABEL_CODE_RING3
; END of [SECTION .ring3]
```

这个代码段非常简单，仍然跟[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，我们需要把 VIDEO 段的 DPL 修改为 3：

```
LABEL_DESC_VIDEO: Descriptor 0B8000h, 0ffffh, DA_DRW + DA_DPL3
```

可以看到，在显示完数字 3 之后，执行了一句 jmp $，从而程序不再继续执行。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。

然后添加新段对应的描述符：

```
LABEL_DESC_CODE_RING3: Descriptor 0, SegCodeRing3Len - 1, DA_C + DA_32
+ DA_DPL3
```

注意：这里的属性加上了 DA_DPL3，让它的 DPL 变成了 3。

相应地，选择子是这样的：

```
SelectorCodeRing3 equ LABEL_DESC_CODE_RING3 - LABEL_GDT + SA_RPL3
```

我们用 SA_RPL3 将 RPL 也设成了 3。

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。

然后添加一个堆栈段：

代码 3-18 （节自\chapter3\e\pmtest5.asm）

```
; 堆栈段 ring3
[SECTION .s3]
ALIGN   32
[BITS   32]
LABEL_STACK3:
```

```
    times 512 db 0
TopOfStack3 equ $ - LABEL_STACK3
; END of [SECTION .s3]
```

它的描述符是这样的：

```
LABEL_DESC_STACK3: Descriptor 0, TopOfStack3, DA_DRWA + DA_32 + DA_DPL3
```

选择子是这样的：

```
SelectorStack3 equ LABEL_DESC_STACK3 - LABEL_GDT + SA_RPL3
```

至此，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 3-19　（节自\chapter3\e\pmtest5.asm）

```
.2: ; 显示完毕

    call    DispReturn

    push    SelectorStack3
    push    TopOfStack3
    push    SelectorCodeRing3
    push    0
    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 3-22 所示。

图 3-22　pmtest5.com 的执行结果（1）

成功了！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！

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[SECTION .ring3]的代码稍做修改：

代码 3-20 （节自\chapter3\e\pmtest5.asm）

```
LABEL_CODE_RING3:
    mov ax, SelectorVideo
    mov gs, ax                  ; 视频段选择子（目的）

    mov edi, (80 * 14 + 0) * 2  ; 屏幕第 14 行，第 0 列
    mov ah, 0Ch                 ; 0000: 黑底    1100: 红字
    mov al, '3'
    mov [gs:edi], ax

    call    SelectorCallGateTest:0  ; 测试调用门（无特权级变换），
                                    ; 将打印字母 'C'
    jmp $
SegCodeRing3Len equ $ - LABEL_CODE_RING3
; END of [SECTION .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，可是，为了满足 CPL 和 RPL 都小于等于调用门 DPL 的条件，我们必须同时修改调用门：

```
LABEL_CALL_GATE_TEST: Gate SelectorCodeDest, 0, 0, DA_386CGate + DA_
DPL0 + DA_DPL3
```

编译，运行。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 3-21 （节自\chapter3\e\pmtest5.asm）

```
[SECTION .tss]
ALIGN   32
[BITS   32]
LABEL_TSS:
        DD  0                   ; Back
        DD  TopOfStack          ; 0 级堆栈
        DD  SelectorStack       ;
        DD  0                   ; 1 级堆栈
```

```
        DD  0                       ;
        DD  0                       ; 2 级堆栈
        DD  0                       ;
        DD  0                       ; cr3
        DD  0                       ; eip
        DD  0                       ; eflags
        DD  0                       ; eax
        DD  0                       ; ecx
        DD  0                       ; edx
        DD  0                       ; ebx
        DD  0                       ; esp
        DD  0                       ; ebp
        DD  0                       ; esi
        DD  0                       ; edi
        DD  0                       ; es
        DD  0                       ; cs
        DD  0                       ; ss
        DD  0                       ; ds
        DD  0                       ; fs
        DD  0                       ; gs
        DD  0                       ; LDT
        DW  0                       ; 调试陷阱标志
        DW  $ - LABEL_TSS + 2       ; I/O 位图基址
        DW  0ffh                    ; I/O 位图结束标志
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。

对应 TSS 的描述符如下：

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

选择子：

```
SelectorTSS     equ     LABEL_DESC_TSS - LABEL_GDT
```

另外，添加初始化 TSS 描述符的代码之后，TSS 就准备好了，我们需要在特权级变换之前加载它：

代码 3-22　（节自\chapter3\e\pmtest5.asm）

```
.2: ; 显示完毕
```

```
call    DispReturn

; Load TSS
mov     ax, SelectorTSS
ltr     ax

push    SelectorStack3
push    TopOfStack3
push    SelectorCodeRing3
push    0
retf
```

再运行，好，成功了！运行结果如图 3-23 所示。

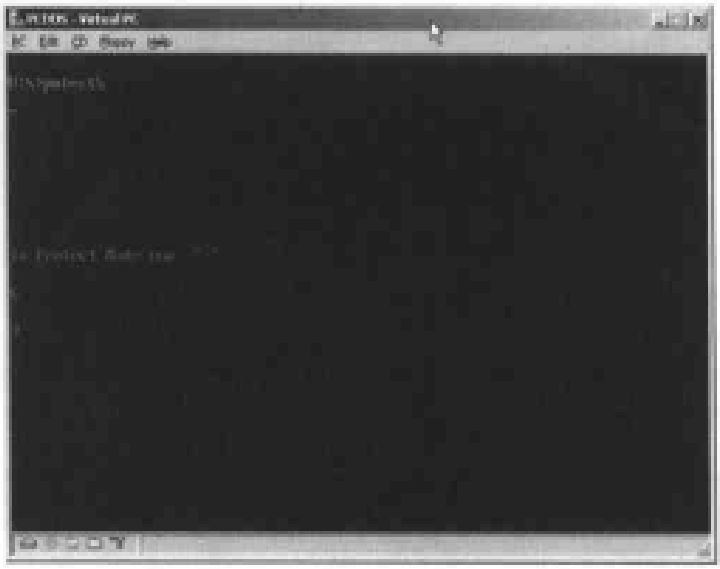

图 3-23　pmtest5.com 的执行结果（2）

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 DOS，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 DOS：

代码 3-23　（节自\chapter3\e\pmtest5.asm）

```
LABEL_SEG_CODE_DEST:
    ;jmp    $
    mov     ax, SelectorVideo
    mov     gs, ax                  ; 视频段选择子（目的）

    mov     edi, (80 * 12 + 0) * 2  ; 屏幕第 12 行，第 0 列
    mov     ah, 0Ch                 ; 0000: 黑底    1100: 红字
    mov     al, 'C'
    mov     [gs:edi], ax

    ; Load LDT
    mov     ax, SelectorLDT
    lldt    ax

    jmp     SelectorLDTCodeA:0  ; 跳入局部任务，将打印字母 'L'

    ;retf
SegCodeDestLen  equ $ - LABEL_SEG_CODE_DEST
; END of [SECTION .sdest]
```

编译，运行，结果如图 3-24 所示。

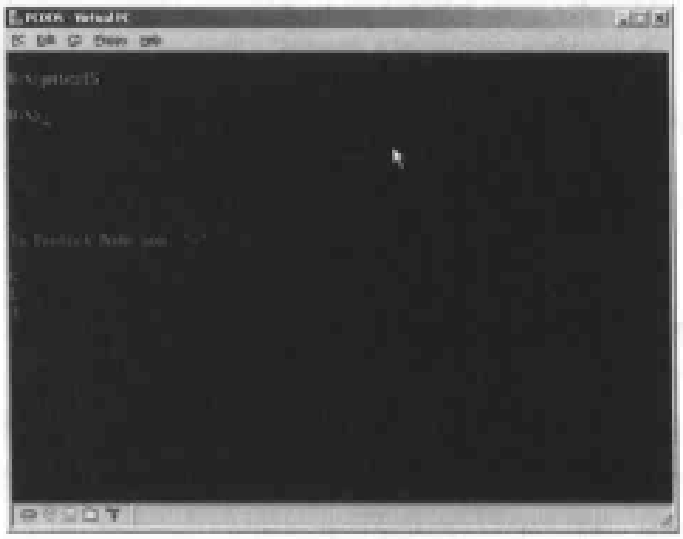

图 3-24　pmtest5.com 的执行结果（3）

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。

3.2.3.4　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。

在涉及到特权级的每一步中，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. 逻辑地址、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 3-4 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 3-25 来说明。

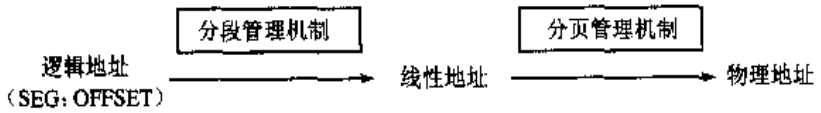

图 3-25 分页开启时的地址转换

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 3-25 中我们知道，分页机制就像一个函数：

物理地址=F（线性地址）

我们通过图 3-26 来看一下这个 F 是怎样的。

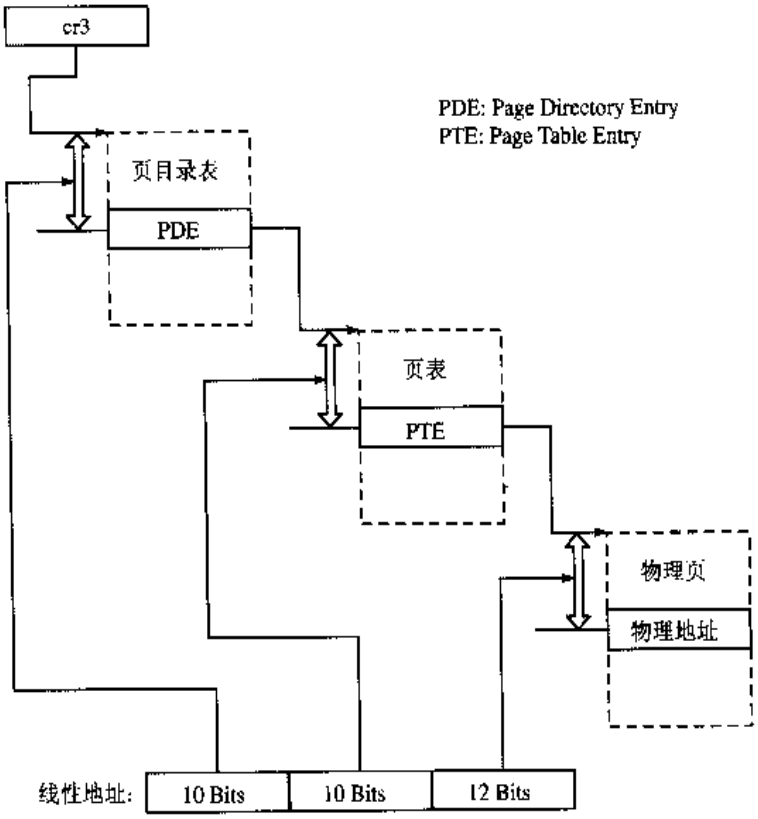

图 3-26　分页机制示意

如图 3-26 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 12~21 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 3-6）。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。

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（详见 pmtest6.asm）：

代码 3-24 （节自\chapter3\f\pmtest6.asm）

```
SetupPaging:    ; 为简化处理，暂时让所有线性地址对应相等的物理地址
    ; 首先初始化页目录
    mov     ax,  SelectorPageDir
    mov     es,  ax
    mov     ecx, 1024; 共 1024 个表项
    xor     edi, edi
    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（1024 个，4MB 内存空间）
    mov     ax,  SelectorPageTbl
    mov     es,  ax
    mov     ecx, 1024 * 1024    ; 共 1024² 个页表项，即有 1024² 个页
    xor     edi, edi
    xor     eax, eax
    mov     eax, PG_P  | PG_USU | PG_RWW
.2:
    stosd
    add     eax, 4096       ; 每一页指向 4KB 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
```

在这段代码中，相关符号的定义如下：

代码 3-25　（节自\chapter3\f\pmtest6.asm）

```
    PageDirBase     equ 200000h ; 页目录开始地址：2MB
    PageTblBase     equ 201000h ; 页表开始地址：       2MB+4KB
    ......
    LABEL_DESC_PAGE_DIR:    Descriptor PageDirBase, 4096, DA_DRW
    LABEL_DESC_PAGE_TBL:    Descriptor   PageTblBase,   1024,   DA_DRW   |
DA_LIMIT_4K
    ......
    SelectorPageDir equ LABEL_DESC_PAGE_DIR - LABEL_GDT
    SelectorPageTbl equ LABEL_DESC_PAGE_TBL - LABEL_GDT
```

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 3-27 和图 3-28 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，各位的名称使用的是英文原文，随后有更为详细的解释。

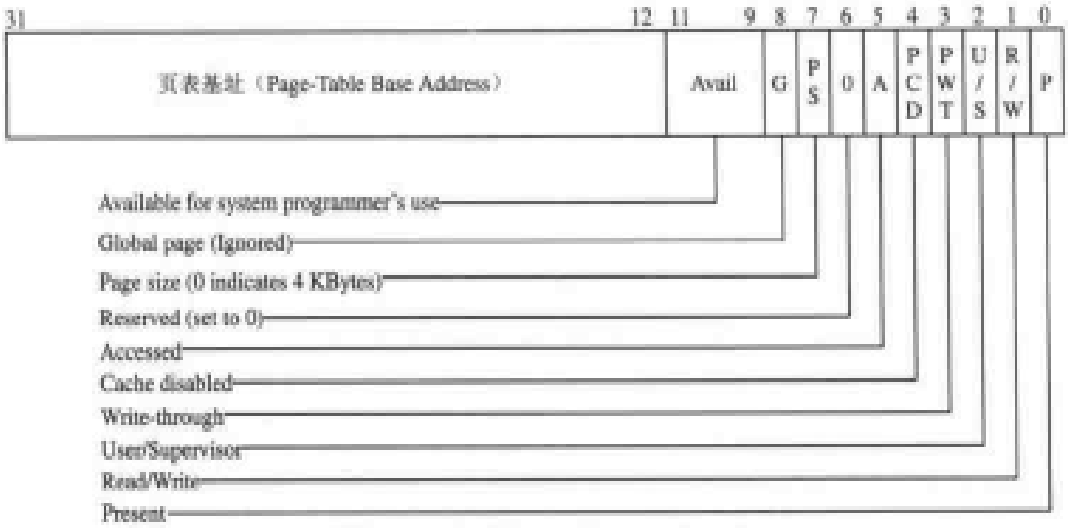

图 3-27 PDE（Page-Directory Entry）

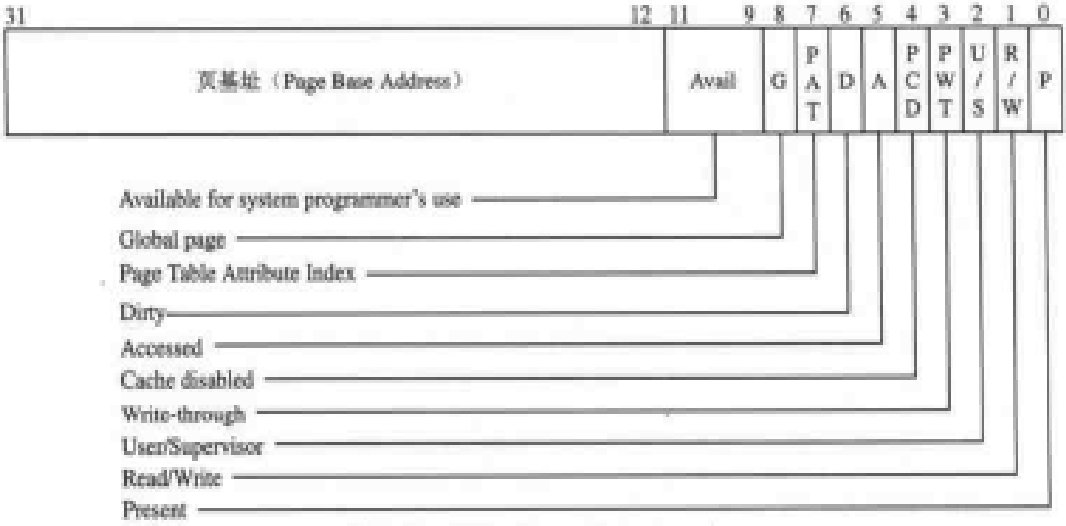

图 3-28 PTE（Page-Table Entry）

PDE 和 PTE 中各位的解释如表 3-5 所示。

表 3-5 PDE 和 PTE 详解

位	概括说明	具体说明		备 注
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	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	页在内存中	

续表

位	概括说明	具体说明		备　注
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	只读	如果 WP 为 0，那么即使用户级(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
		R/W=1	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	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	用户级别(User Privilege Level)，如果 CPL 为 3，那么它便是在此级别	
PWT	用于控制对单个页或者页表的 Write-through 或 Write-back 缓冲策略	PWT=0	Write-back 缓冲策略被使用	当 cr0 寄存器的 CD（Cache-Disable）位被设置时会被忽略
		PWT=1	Write-through 缓冲策略被使用	
PCD	用于控制对单个页或者页表的缓冲	PCD=0	页或页表可以被缓冲	
		PCD=1	页或页表不可以被缓冲	
A	指示页或页表是否被访问	A=0	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	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
		A=1	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	D=0	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	
		D=1	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	
PS	决定页大小	PS=0	页大小为 4KB，PDE 指向页表	
		PS=1	在此不予讨论	
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		

处理器会将最近经常用到的页目录和页表项保存在 TLB（Translation Lookaside Buffer）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 3-29 所示。

图 3-29 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

现在再来看我们先前写的那段代码，我想你差不多应该明白了（请同时参考图 3-30）。

开头的两个语句 mov ax, SelectorPageDir 和 mov es, ax 将 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。标号“.1”处的指令 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。

那么来看这个 PDE 是什么值。PageTblBase | PG_P | PG_USU | PG_RWW 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，而指令 add eax, 4096 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（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。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 $0\leqslant x\leqslant 0FFFh$。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。图 3-30 所示即为 pmtest6.asm 中的分页图示。

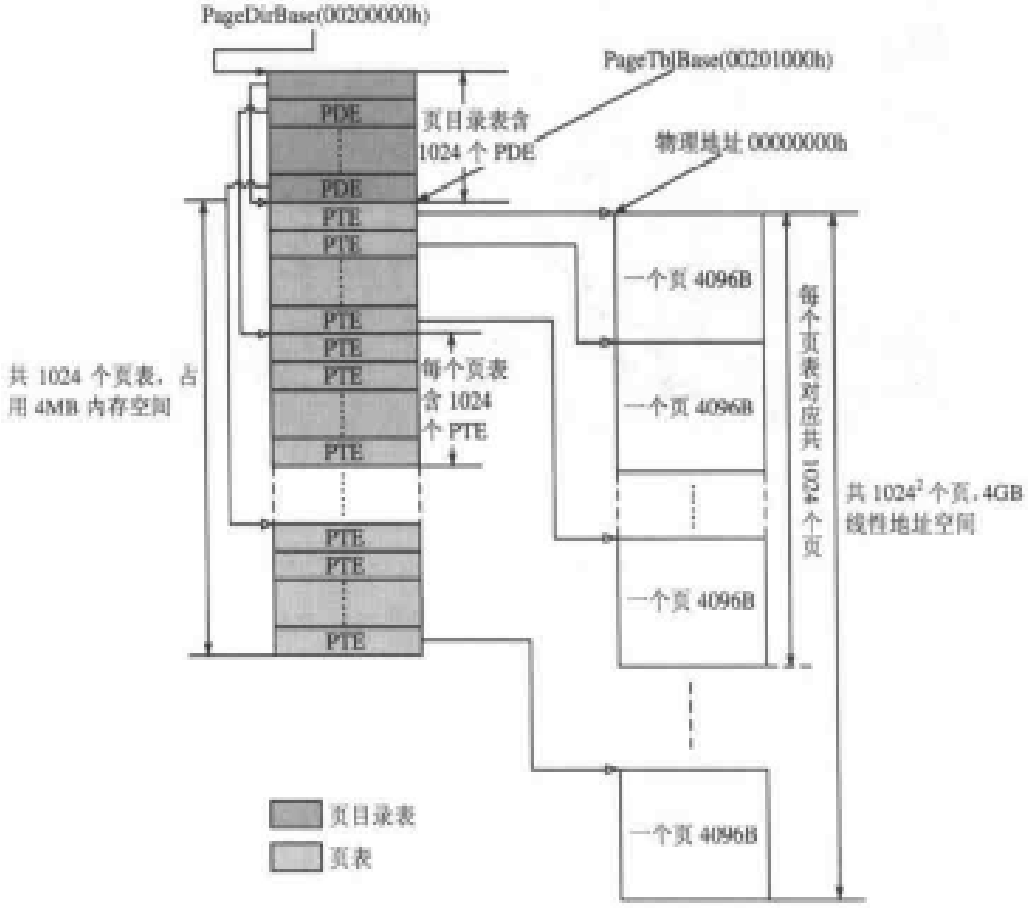

图 3-30　pmtest6.asm 中分页图示

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：

```
mov eax, PageDirBase
mov cr3, eax
```

然后设置 cr0 的 PG：

```
mov eax, cr0
or  eax, 80000000h
mov cr0, eax
```

这样，分页机制就启动完成了。

执行结果如图 3-31 所示。

图 3-31　pmtest6.com 的执行结果

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。

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。

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，具体用法如表 3-6 和表 3-7 所示。

表 3-6　int 15h 用法之输入

寄存器	描　述
eax	0000E820h
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（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

表 3-7　int 15h 用法之输出

寄存器	描　述
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

地址范围描述符结构（Address Range Descriptor Structure）如表 3-8 所示。

表 3-8　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 3-9 所示。

表 3-9　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，eax=0000E820h 的 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，必须先有一块缓冲区。我们在数据段中这样定义（详见 pmtest7.asm）：

```
_MemChkBuf: times   256 db  0
```

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 256B 的缓冲区，它最多可以存放 12 个 20B 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。

得到内存描述信息的代码如下所示：

代码 3-26 （节自\chapter3\g\pmtest7.asm）

```
LABEL_BEGIN:
    mov ax, cs
    mov ds, ax
    mov es, ax
    mov ss, ax
    mov sp, 0100h

    mov [LABEL_GO_BACK_TO_REAL+3], ax
    mov [_wSPValueInRealMode], sp

    ; 得到内存数
    mov ebx, 0
    mov di, _MemChkBuf
.loop:
    mov eax, 0E820h
    mov ecx, 20
    mov edx, 0534D4150h
    int 15h
    jc  LABEL_MEM_CHK_FAIL
    add di, 20
    inc dword [_dwMCRNumber]
    cmp ebx, 0
    jne .loop
    jmp LABEL_MEM_CHK_OK
LABEL_MEM_CHK_FAIL:
```

```
    mov dword [_dwMCRNumber], 0
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，每次循环我们让变量_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，添加这样一个过程：

代码 3-27　（节自\chapter3\g\pmtest7.asm）

```
DispMemSize:
    pushesi
    pushedi
    pushecx

    mov esi, MemChkBuf
    mov ecx, [dwMCRNumber]  ;for(int i=0;i<[MCRNumber];i++)
.loop:                      ;{
    mov edx, 5              ;   for(int j=0;j<5;j++)
    mov edi, ARDStruct      ;   {
.1:                         ;
    push dword [esi]        ;
    call DispInt            ;   DispInt(MemChkBuf[j*4]); /* 显示一个
                            ;                               成员 */
    pop eax                 ;
    stosd                   ;   ARDStruct[j*4] = MemChkBuf[j*4];
    add esi, 4              ;
    dec edx                 ;
    cmp edx, 0              ;
    jnz .1                  ;   }
    call DispReturn         ;   printf("\n");
    cmp dword [dwType], 1   ;   if(Type == AddressRangeMemory)
    jne .2                  ;   {
    mov eax,[dwBaseAddrLow] ;
    add eax,[dwLengthLow]   ;
    cmp eax,[dwMemSize]     ;   if(BaseAddrLow+LengthLow > MemSize)
    jb .2                   ;
    mov [dwMemSize], eax    ;   MemSize = BaseAddrLow + LengthLow;
```

```
.2:                         ;   }
    loop .loop              ;}
                            ;
    call DispReturn         ;printf("\n");
    push szRAMSize          ;
    call DispStr            ;printf("RAM size:");
    add esp, 4              ;
                            ;
    push dword [dwMemSize]  ;
    call DispInt            ;DispInt(MemSize);
    add esp, 4              ;
    pop ecx
    pop edi
    pop esi
    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 如下所示：

代码 3-28 （节自\chapter3\g\lib.inc）

```
.....

; ------------------------------------------------------------
```

```
; 显示一个整形数
; ----------------------------------------------------------------
DispInt:
    mov     eax, [esp + 4]
    shr     eax, 24
    call    DispAL

    mov     eax, [esp + 4]
    shr     eax, 16
    call    DispAL

    mov     eax, [esp + 4]
    shr     eax, 8
    call    DispAL

    mov     eax, [esp + 4]
    call    DispAL

    mov     ah, 07h         ; 0000b: 黑底    0111b: 灰字
    mov     al, 'h'
    push    edi
    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]即为已经压入栈的参数，函数通过 4 次调用 DispAL 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  - $$
```

在实模式下应该使用_dwMemSize，而在保护模式下应该使用 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 3.2.1 节。

我们添加了函数 DispMemSize，调用它的代码如下：

代码 3-29　（节自\chapter3\g\pmtest7.asm）

```
        push        szMemChkTitle
        call        DispStr
        add         esp, 4

        call        DispMemSize
```

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 3-32 所示。

从图 3-32 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 3-10 所示。

图 3-32　pmtest7.com 的执行结果

表 3-10　图 3-32 中列出的内存情况解释

内存段	属　性	是否可被 OS 使用
00000000h~0009FBFFh	AddressRangeMemory	可
0009FC00h~0009FFFFh	AddressRangeReserved	不可
000E0000h~000FFFFFh	AddressRangeReserved	不可
00100000h~01FFFFFFh	AddressRangeMemory	可
FFFC0000h~FFFFFFFFh	AddressRangeReserved	不可

从这里我们可以直观地看到，操作系统所能使用的最大的内存地址为 01FFFFFFh，所以此机器拥有 32MB 的内存。而且，幸运地，我们指定的 256B 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 3-30　（节自\chapter3\g\pmtest7.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=4MB=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，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, SelectorPageDir     ; 此段首地址为 PageDirBase
    mov     es, ax
    xor     edi, edi
    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（1024 个，4MB 内存空间）
    mov     ax, SelectorPageTbl ; 此段首地址为 PageTblBase
    mov     es, ax
    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
    xor     edi, edi
    xor     eax, eax
    mov     eax, PG_P | PG_USU | PG_RWW
.2:
    stosd
    add     eax, 4096       ; 每一页指向 4KB 的空间
    loop    .2

    ......
```

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, 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，在 Windows 下做开发时，如果写两个一样的程序，然后同时打开两个 VC，同时进行调试，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 3-33 所示。

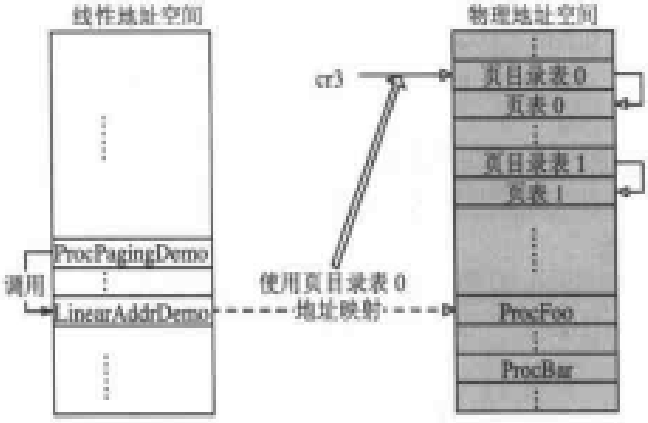

图 3-33　开始时的内存映射关系

开始，我们让过程 ProcPagingDemo 中的代码实现向线形地址 Linear AddrDemo 处的转移，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 3-34 所示的情形。

页目录表和页表的切换让 LinearAddrDemo 映射到物理地址空间中的 ProcBar 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。

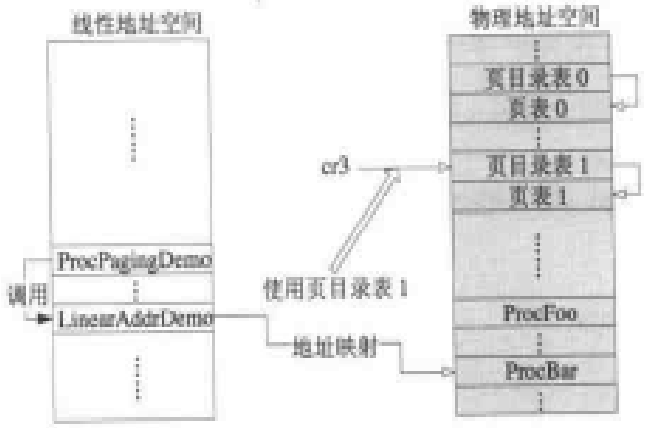

图 3-34　后来的内存映射关系

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。此段的定义如下：

```
LABEL_DESC_FLAT_C:  Descriptor 0, 0fffffh, DA_CR|DA_32|DA_LIMIT_4K
LABEL_DESC_FLAT_RW: Descriptor 0, 0fffffh, DA_DRW|DA_LIMIT_4K
SelectorFlatC       equ     LABEL_DESC_FLAT_C   - LABEL_GDT
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 3-31　（节自\chapter3\h\pmtest8.asm）

```
SetupPaging:
    ; 根据内存大小计算应初始化多少 PDE 以及多少页表
    ......

.no_remainder:
    mov [PageTableNumber], 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, PageDirBase0   ; 此段首地址为 PageDirBase
    xor eax, eax
    mov eax, PageTblBase0 | 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
    mov eax, [PageTableNumber]  ; 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, PageTblBase0   ; 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 4KB 的空间
    loop    .2

    mov eax, PageDirBase0
    mov cr3, eax
    ......
```

我们原来并没有把页表个数保存起来，而现在，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 PageTable Number，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。

比如页目录在物理地址 PageDirBase0 处，初始化 PDE 时将 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 3-33 和图 3-34，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。因为程序开始时 LinearAddrDemo 是指向 ProcFoo 的，而开始时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 中是要调用 LinearAddrDemo 的，而上面我们提到，LinearAddrDemo 是在 FLAT 段中的，所以，如果不想使用段间转移的话，我们就需要把 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，ProcPagingDemo、LinearAddrDemo、ProcFoo 和 ProcBar 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 3-32　（节自\chapter3\h\pmtest8.asm）

```
LinearAddrDemo  equ     00401000h
ProcFoo         equ     00401000h
ProcBar         equ     00501000h
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 3-33　（节自\chapter3\h\pmtest8.asm）

```
PagingDemo:
    mov     ax, cs
    mov     ds, ax
    mov     ax, SelectorFlatRW
    mov     es, ax

    push    LenFoo
    push    OffsetFoo
    push    ProcFoo
    call    MemCpy
    add     esp, 12

    push    LenBar
    push    OffsetBar
    push    ProcBar
```

```
    call    MemCpy
    add     esp, 12

    push    LenPagingDemoAll
    push    OffsetPagingDemoProc
    push    ProcPagingDemo
    call    MemCpy
    add     esp, 12

    mov     ax, SelectorData
    mov     ds, ax          ; 数据段选择子
    mov     es, ax

    call    SetupPaging     ; 启动分页
    call    SelectorFlatC:ProcPagingDemo
    call    PSwitch         ; 切换页目录，改变地址映射关系
    call    SelectorFlatC:ProcPagingDemo

    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下：

代码 3-34 物理地址 ProcPagingDemo 处的代码（节自\chapter3\h\pmtest8.asm）

```
PagingDemoProc:
OffsetPagingDemoProc    equ PagingDemoProc - $$
    mov     eax, LinearAddrDemo
    call    eax
    retf

LenPagingDemoAll    equ $ - PagingDemoProc
```

代码 3-35 物理地址 ProcFoo 处的代码（节自\chapter3\h\pmtest8.asm）

```
foo:
OffsetFoo   equ foo - $$
    mov ah, 0Ch                 ; 0000: 黑底    1100: 红字
    mov al, 'F'
```

```
    mov [gs:((80 * 17 + 0) * 2)],ax     ;目的数据偏移。屏幕第 17 行,第 0 列
    mov al, 'o'
    mov [gs:((80 * 17 + 1) * 2)],ax     ;目的数据偏移。屏幕第 17 行,第 1 列
    mov [gs:((80 * 17 + 2) * 2)],ax     ;目的数据偏移。屏幕第 17 行,第 2 列
    ret
LenFoo  equ $ - foo
```

代码 3-36　物理地址 ProcBar 处的代码（节自\chapter3\h\pmtest8.asm）

```
bar:
OffsetBar   equ bar - $$
    mov ah, 0Ch                         ; 0000: 黑底    1100: 红字
    mov al, 'B'
    mov [gs:((80 * 18 + 0) * 2)],ax     ;目的数据偏移。屏幕第 18 行,第 0 列
    mov al, 'a'
    mov [gs:((80 * 18 + 1) * 2)],ax     ;目的数据偏移。屏幕第 18 行,第 1 列
    mov al, 'r'
    mov [gs:((80 * 18 + 2) * 2)],ax     ;目的数据偏移。屏幕第 18 行,第 2 列
    ret
LenBar  equ $ - bar
```

其实，代码 3-34 中只是一个短调用。代码 3-35 和代码 3-36 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。

我们回过头来看代码 3-33，其中的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 3-33 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 3-37　改变地址映射关系（节自\chapter3\h\pmtest8.asm）

```
PSwitch:
    ; 初始化页目录
    mov     ax, SelectorFlatRW
    mov     es, ax
    mov     edi, PageDirBase1   ; 此段首地址为 PageDirBase
    xor     eax, eax
    mov     eax, PageTblBase1 | PG_P  | PG_USU | PG_RWW
    mov     ecx, [PageTableNumber]
```

```
.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
    mov     eax, [PageTableNumber]  ; 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, PageTblBase1   ; 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 4KB 的空间
    loop    .2

    mov     eax, LinearAddrDemo
    shr     eax, 22
    mov     ebx, 4096
    mul     ebx
    mov     ecx, eax
    mov     eax, LinearAddrDemo
    shr     eax, 12
    and     eax, 03FFh  ; 1111111111b (10 bits)
    mov     ebx, 4
    mul     ebx
    add     eax, ecx
    add     eax, PageTblBase1
    mov     dword [es:eax], ProcBar | PG_P | PG_USU | PG_RWW

    mov     eax, PageDirBase1
    mov     cr3, eax
    jmp     short .3
.3:
    nop

    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 3-34 所示的情况。

在代码 3-37 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 3-35 所示。

图 3-35　pmtest8.com 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 Windows 下做开发时看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 3-36 所示）。

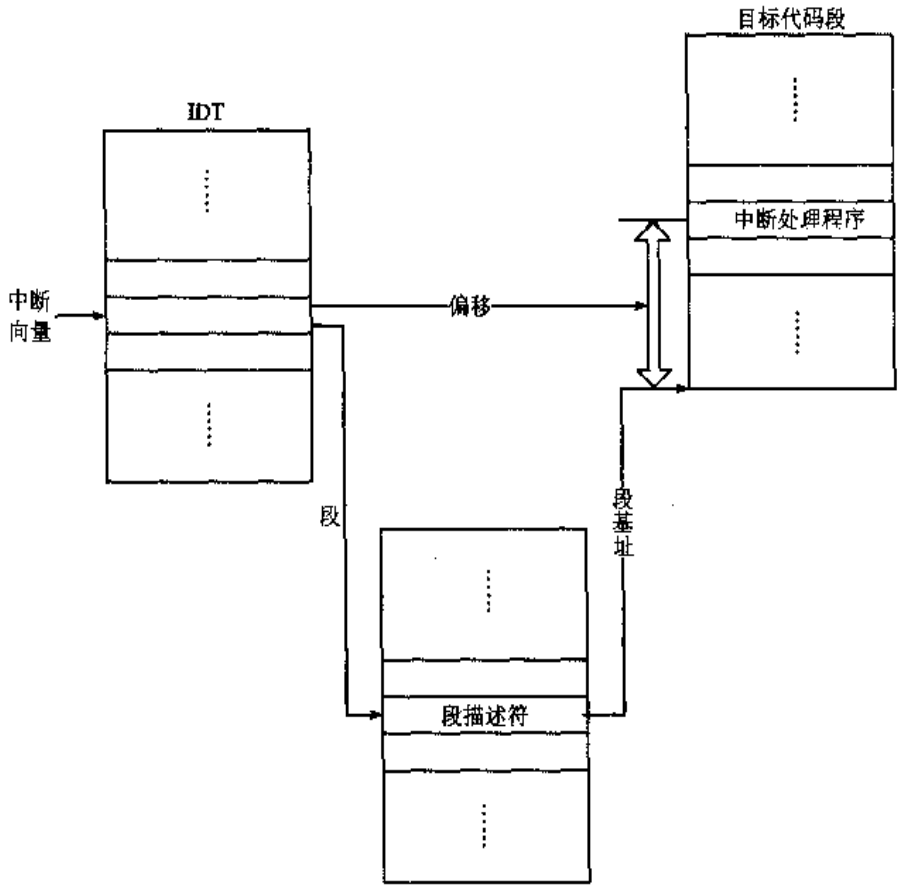

图 3-36 中断向量到中断处理程序的对应过程

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 3-37 所示。

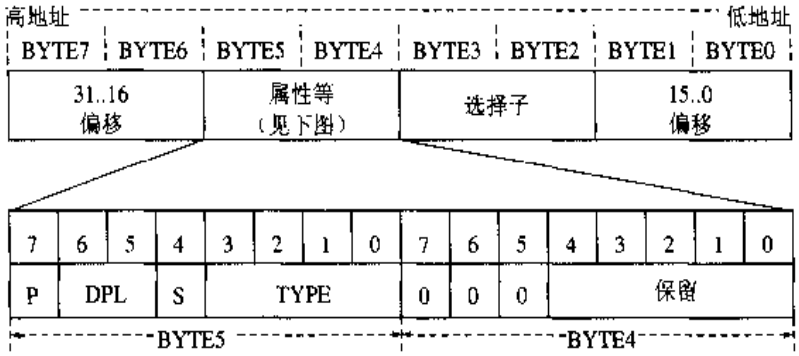

图 3-37　中断门和陷阱门

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，还涉及到处理器与硬件的联系等内容。所以，我们还是要对中断进行一下全面了解。

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。

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 3-36）。

那么，处理器到底能处理哪些中断或异常呢？表 3-11 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 3-11 保护模式中的中断和异常

向量号	助记符	描述	类型	有无出错码	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	浮点指令或 WAIT/FWAIT 指令
8	#DF	双重错误	Abort	有（零）	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（Zero）	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：

- **Faults** 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。
- **Traps** 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。
- **Aborts** 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 3-36 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 3-38 所示。

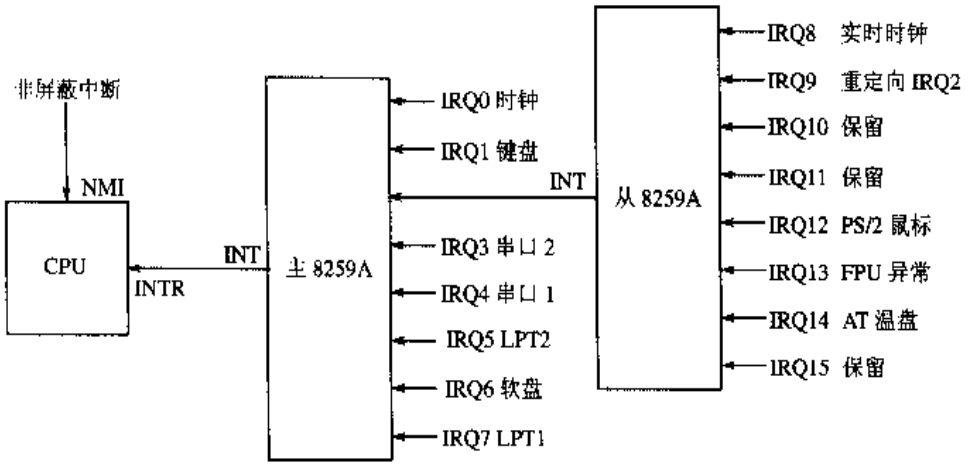

图 3-38　8259A

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 3-11 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 3-38 所示。

由图 3-38 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 3-11 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 20A 和 21A，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 3-39 所示的 ICW 的格式。

我们看到，在写入 ICW2 时涉及到了与中断向量号的对应，这便是窍门所在了。

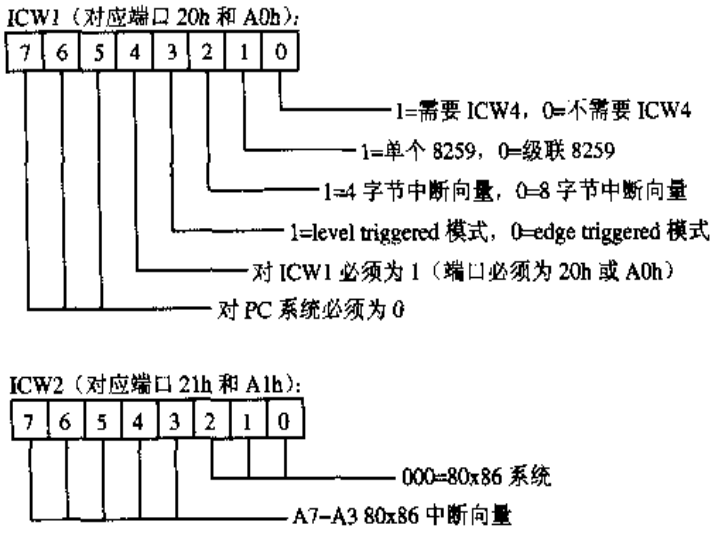

图 3-39　ICW 的格式

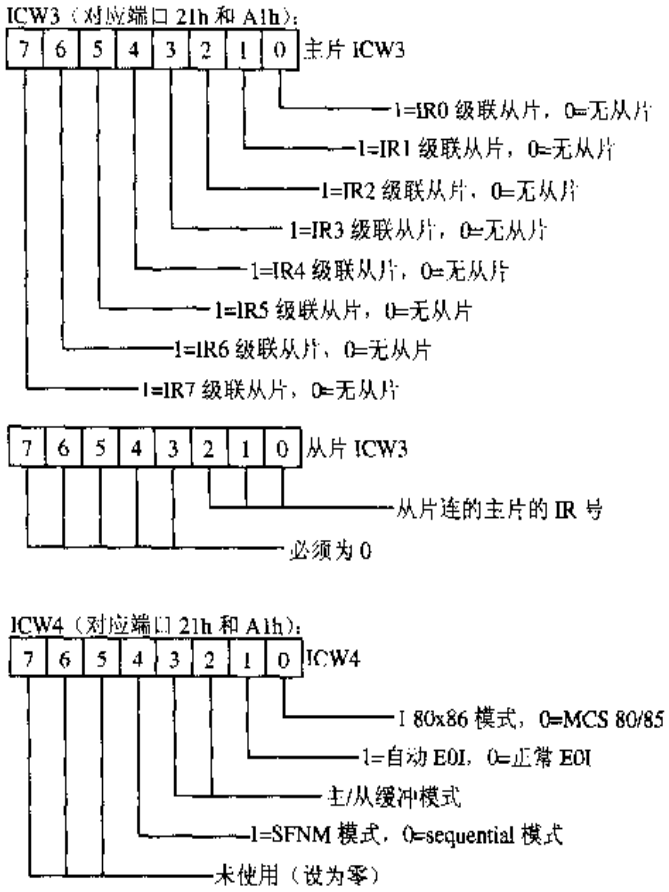

图 3-39　ICW 的格式（续图）

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 3-38　（节自\chapter3\i\pmtest9.asm）

```
Init8259A:
    mov     al, 011h
    out     020h, al    ; 主 8259, ICW1.
    call    io_delay

    out     0A0h, al    ; 从 8259, ICW1.
    call    io_delay
```

```
mov     al, 020h    ; IRQ0 对应中断向量 0x20
out     021h, al    ; 主 8259, ICW2.
call    io_delay

mov     al, 028h    ; IRQ8 对应中断向量 0x28
out     0A1h, al    ; 从 8259, ICW2.
call    io_delay

mov     al, 004h    ; IR2 对应从 8259
out     021h, al    ; 主 8259, ICW3.
call    io_delay

mov     al, 002h    ; 对应主 8259 的 IR2
out     0A1h, al    ; 从 8259, ICW3.
call    io_delay

mov     al, 001h
out     021h, al    ; 主 8259, ICW4.
call    io_delay

out     0A1h, al    ; 从 8259, ICW4.
call    io_delay

mov     al, 11111111b   ; 屏蔽主 8259 所有中断
out     021h, al        ; 主 8259, OCW1.
call    io_delay

mov     al, 11111111b   ; 屏蔽从 8259 所有中断
out     0A1h, al        ; 从 8259, OCW1.
call    io_delay

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 3-11 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 3-40 所示。

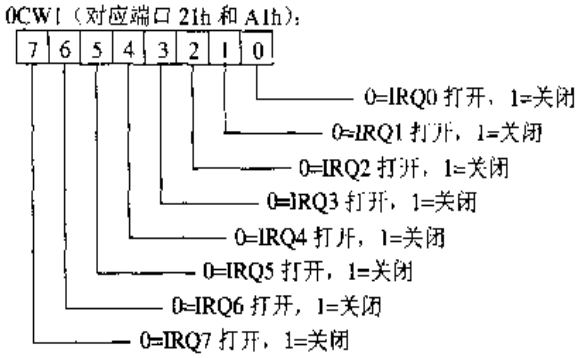

图 3-40　OCW1

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 3-41 所示。

OCW2（对应端口 21h 和 A1h）：

7	6	5	4	3	2	1	0
R	SL	EOI	0	0	L2	L1	L0

图 3-41　OCW2

发送 EOI 给 8259A 可以由如下的代码完成：

```
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 3-38，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 3-39　io_delay（节自\chapter3\i\pmtest9.asm）

```
io_delay:
    nop
```

```
    nop
    nop
    nop
    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 3-40 IDT（节自\chapter3\i\pmtest9.asm，后被修改）

```
[SECTION .idt]
LABEL_IDT:
%rep 255
            Gate SelectorCode32, SpuriousHandler, 0, DA_386IGate
%endrep

IdtLen      equ $ - LABEL_IDT
IdtPtr      dw  IdtLen  ; 段界限
            dd  0       ; 基地址
; 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 Selector Code32: 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 3-41 SpuriousHandler（节自\chapter3\i\pmtest9.asm）

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

加载 IDT 的代码与对 GDT 的处理非常类似：

代码 3-42　加载 IDTR（节自\chapter3\i\pmtest9.asm）

```
; 为加载 IDTR 做准备
xor     eax, eax
mov     ax, ds
shl     eax, 4
add     eax, LABEL_IDT              ; eax <- idt 基地址
mov     dword [IdtPtr + 2], eax     ; [IdtPtr + 2] <- idt 基地址

......

cli         ; 关中断

lidt    [IdtPtr]    ; 加载 IDTR
```

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 DOS。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 DOS 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 3-43　中断（节自\chapter3\i\pmtest9.asm）

```
......
call    Init8259A
int     080h
......
```

由于我们把 IDT 中所有的描述符都初始化成同一个样子，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。

先修改一下 IDT：

代码 3-44　IDT（节自\chapter3\i\pmtest9.asm）

```
LABEL_IDT:
%rep 128
            Gate SelectorCode32, SpuriousHandler, 0, DA_386IGate
%endrep
.080h:      Gate SelectorCode32,  UserIntHandler, 0, DA_386IGate
......
```

我们把第 80h 号中断单独列出来，并新增加一个函数 UserIntHandler 来处理它。函数 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 3-45　UserIntHandler（节自\chapter3\i\pmtest9.asm）

```
_UserIntHandler:
UserIntHandler  equ _UserIntHandler - $$
    mov ah, 0Ch                          ; 0000: 黑底    1100: 红字
    mov al, 'I'
    mov [gs:((80 * 0 + 70) * 2)], ax     ; 屏幕第 0 行，第 70 列。
    iretd
```

运行结果如图 3-42 所示，可以看到红色的字符 I 出现在屏幕右上方。

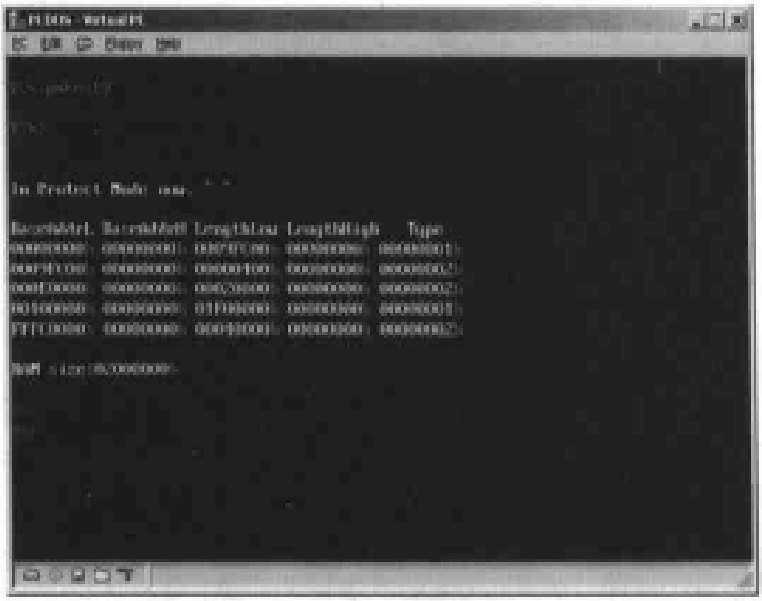

图 3-42　pmtest9.com 的执行结果（1）

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 3-46　ClockHandler（节自\chapter3\l\pmtest9.asm）

```
_ClockHandler:
ClockHandler    equ _ClockHandler - $$
    inc byte [gs:((80 * 0 + 70) * 2)]   ; 屏幕第 0 行，第 70 列
    mov al, 20h
    out 20h, al                         ; 发送 EOI
    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 3-47　打开时钟中断（节自\chapter3\l\pmtest9.asm）

```
mov     al, 11111110b   ; 仅仅开启定时器中断
out     021h, al        ; 主 8259, OCW1.
call    io_delay

mov     al, 11111111b   ; 屏蔽从 8259 所有中断
out     0A1h, al        ; 从 8259, OCW1
call    io_delay

ret
```

把 IDT 修改成代码 3-48 的样子。

代码 3-48　打开时钟中断（节自\chapter3\i\pmtest9.asm）

```
LABEL_IDT:
%rep 32
            Gate SelectorCode32, SpuriousHandler, 0, DA_386IGate
%endrep
.020h:      Gate SelectorCode32, ClockHandler, 0, DA_386IGate
%rep 95
            Gate SelectorCode32, SpuriousHandler, 0, DA_386IGate
%endrep
.080h:      Gate SelectorCode32,  UserIntHandler, 0, DA_386IGate
```

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 3-49　演示时钟中断（节自\chapter3\i\pmtest9.asm）

```
    int 080h
    sti
    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 3-43 是在一瞬间抓拍的图片，刚好显示到字符 d。

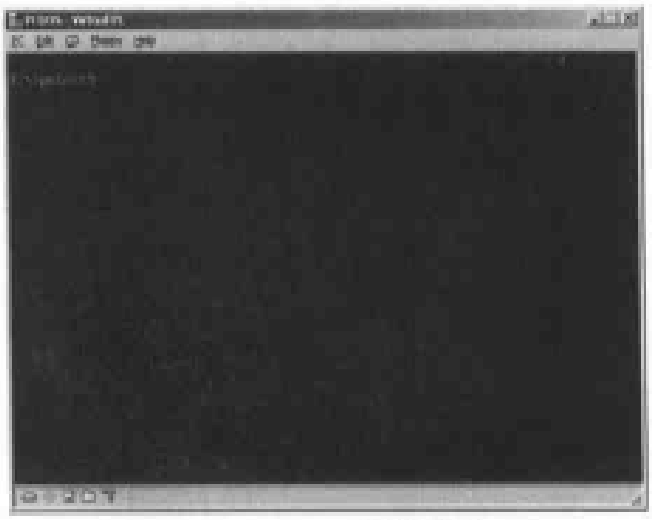

图 3-43　pmtest9.com 的执行结果（2）

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，涉及到的特权级变换的规则是完全一样的。读者可以参考 3.2.3.3 中调用门的相关内容。

3.4.7.2　中断或异常发生时的堆栈变化

如图 3-44 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。

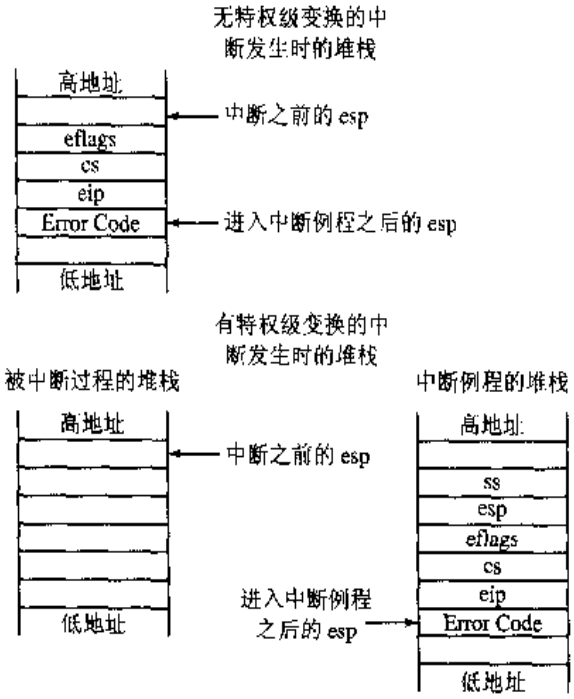

图 3-44　中断或异常发生时的堆栈

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 3-11。

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 3-45 所示。

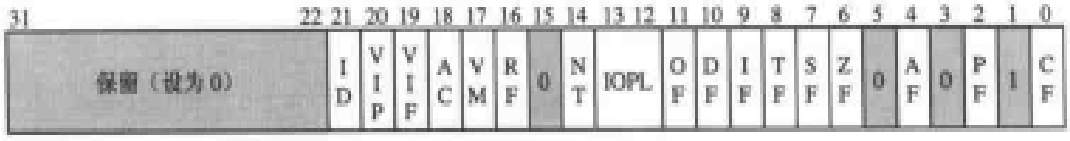

图 3-45　eflags

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 3-19 的话，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 3-50　TSS 举例

```
[SECTION .tss3]
LABEL_TSS3:
        ............
        DD  SelectorLDT3          ; LDT
        DW  0                     ; 调试陷阱标志
        DW  $ - LABEL_TSS3 + 2    ; 指向 I/O 许可位图
    times 12 DB 0FFh              ; 端口 00h--5fh
        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
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 3-21 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。

第4章

让操作系统走进保护模式

千里之行，始于足下。

——老子

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。

表 4-1 引导扇区的格式

名　称	开始字节	长　度	内　容	Tinix 引导盘的值
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（Bytes / Sector）	0x200
BPB_SecPerClus	13	1	每簇扇区数（Sectors / Cluster）	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

续表

名　称	开始字节	长　度	内　容	Tinix 引导盘的值
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	'Tinix0.01'
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 0xAA55	510	2	第 510 字节为 0x55，第 511 字节为 0xAA	0xAA55

紧接着引导扇区的是两个完全相同的 FAT 表，每个 FAT 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。

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。

图 4-1　软盘（1.44MB，FAT12）

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。

表 4-2　根目录区中的条目格式

名　称	偏移（字节）	长度（字节数）	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

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 FLOPPY.IMG，然后把它作为 Virtual PC 的 A 盘，格式化后就可以方便地往其中添加文件和目录了。这样，当我们想查看它的格式时，只需用二进制查看器打开 FLOPPY.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 Virtual PC 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 30 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打开 FLOPPY.IMG，定位到偏移 0x2600 处看一下，如图 4-2 所示。

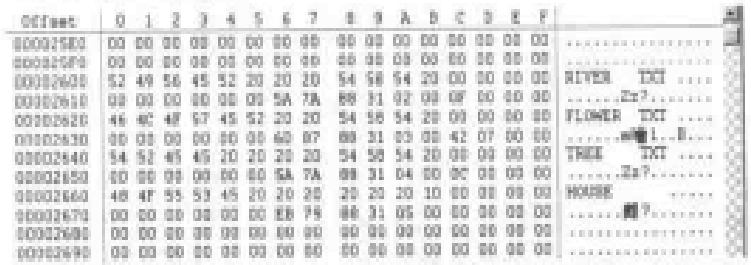

图 4-2　根目录区

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。

表 4-3　RIVER.TXT 的各项值

名　称	值
DIR_Name	RIVER　TXT
DIR_Attr	0x20
DIR_WrtTime	0x7A5A
DIR_WrtDate	0x3188
DIR_FstClus	0x0002
DIR_FileSize	0x0000000F

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 RootDir Sectors 个扇区，则有：

```
RootDirSectors = ((BPB_RootEntCnt * 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' = ((BPB_RootEntCnt * 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，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，如图 4-3 所示。

```
Offset   0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00004100 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00004110 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00004200 72 69 76 65 72 72 69 76  65 72 72 69 76 65 72 00  rivverrivverrivver.
00004210 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00004220 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

图 4-3　数据区

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），如图 4-4 所示。

```
Offset   0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000200 F0 FF FF FF 0F 00 FF FF  FF FF FF FF 09 A0 00 FF  ?       ?
00000210 0F 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000220 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

图 4-4　FAT

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，所以显得特别别扭。具体情况是这样的，假设连续 3 个字节分别如图 4-5 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。

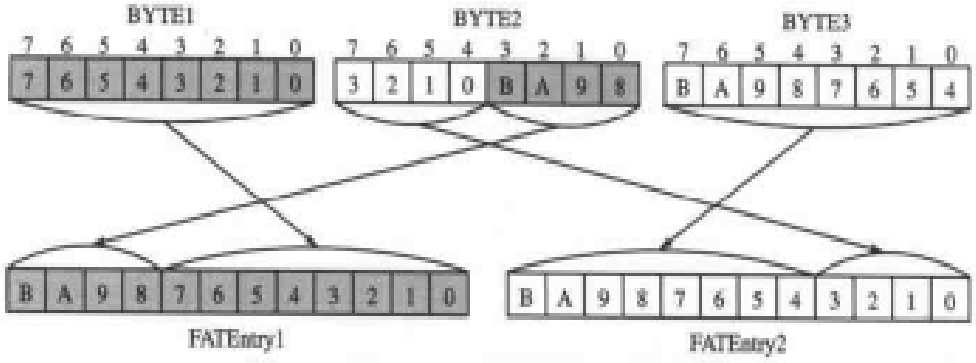

图 4-5　从 FAT 表中得到一个 FAT 项的值

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。结合图 4-4 和图 4-5 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　（节自\chapter4\a\boot.asm）

```
;%define _BOOT_DEBUG_

%ifdef _BOOT_DEBUG_
    org  0100h                  ; 调试状态，做成 .COM 文件，可调试
%else
    org  07c00h                 ; Boot 状态，BIOS 将把 BootSector
                                ; 加载到 0:7C00 处并开始执行
%endif
    jmp short LABEL_START       ; 从这里开始
    nop

    ; 下面是 FAT12 磁盘的头
    BS_OEMName      DB  'ForrestY'      ; OEM String，必须 8 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  'Tinix0.01 '    ; 卷标, 必须 11 字节
BS_FileSysType  DB  'FAT12 '        ; 文件系统类型, 必须 8 字节

LABEL_START:
    ......
```

把生成的 Boot.bin 用 FloppyWriter.exe 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 Windows 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 入内存了，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　（\chapter4\b\loader.asm）

```
org 0100h
    mov ax, 0B800h
    mov gs, ax
    mov ah, 0Fh                         ; 0000: 黑底    1111: 白字
    mov al, 'L'
    mov [gs:((80 * 0 + 39) * 2)], ax    ; 屏幕第 0 行, 第 39 列
    jmp $                               ; 到此停住
```

这段代码被编译成.COM 文件，可以直接在 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，我们用命令行“nasmw loader.asm -o loader.bin”

来编译它。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）	从磁盘读数据入 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：

2×80×18×512=1.44MB

于是，磁头号、柱面（磁道）号和起始扇区号可以用图 4-6 所示的方法来计算。

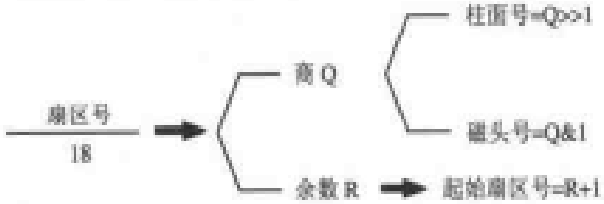

图 4-6　磁头号、柱面号、起始扇区号的计算方法图示

我们就先写一个读软盘扇区的函数吧：

代码 4-3　（节自\chapter4\b\boot.asm）

```
ReadSector:  ; 从第 ax 个 Sector 开始，将 cl 个 Sector 读入 es:bx 中
    push bp
    mov bp, sp
    sub esp, 2              ; 辟出 2 字节的堆栈区域保存要读的扇区数：
                            ; byte [bp-2]
```

```
    mov byte [bp-2], cl
    push bx                     ; 保存 bx
    mov bl, [BPB_SecPerTrk]; bl: 除数
    div bl                      ; y 在 al 中, z 在 ah 中
    inc ah                      ; z ++
    mov cl, ah                  ; cl <- 起始扇区号
    mov dh, al                  ; dh <- y
    shr al, 1                   ; y >> 1 (其实是 y/BPB_NumHeads, 这里
                                ; BPB_NumHeads=2)
    Mov ch, al                  ; ch <- 柱面号
    and dh, 1                   ; dh & 1 = 磁头号
    pop bx                      ; 恢复 bx
    ; 至此, "柱面号, 起始扇区, 磁头号" 全部得到
    mov dl, [BS_DrvNum]         ; 驱动器号 (0 表示 A 盘)
.GoOnReading:
    mov ah, 2                   ; 读
    mov al, byte [bp-2]         ; 读 al 个扇区
    int 13h
    jc .GoOnReading             ; 如果读取错误, CF 会被置为 1, 这时就不停地
                                ; 读, 直到正确为止

    add esp, 2
    pop bp

    ret
```

说到这里，我想提一句，我们写了这个函数，当然不是一蹴而就的，中间很可能会有错误，最好能够同时调试。所以，在开发过程中，我们就把"%define _BOOT_DEBUG_"这个定义打开，编译成.COM 文件，这样可以随时在 DOS 下进行调试。

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 （节自\chapter4\b\boot.asm）

```
    ......
%ifdef _BOOT_DEBUG_
    BaseOfStack equ 0100h   ; 调试状态下堆栈基地址(栈底, 从这个位置向
                            ; 低地址生长)
%else
    BaseOfStack equ 07c00h ; Boot 状态下堆栈基地址(栈底, 从这个位置向
                            ; 低地址生长)
%endif
    ......
```

```
mov ax, cs
    mov ds, ax
    mov es, ax
    mov ss, ax
    mov sp, BaseOfStack
    ......
```

好了，读扇区的函数写好了，下面我们就开始编写在软盘中寻找 Loader.bin 的代码。

代码 4-5　（节自\chapter4\b\boot.asm）

```
  ......
  mov sp, BaseOfStack

  xor ah, ah    ; ┐
  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    ; ┐
  jz LABEL_NO_LOADERBIN                ; ├ 判断根目录区是否已经读完，如果
  dec word [wRootDirSizeForLoop]       ; ┘ 读完表示没有找到 Loader.bin
  mov ax, BaseOfLoader
  mov es, ax                ; es <- BaseOfLoader
  mov bx, OffsetOfLoader    ; bx <- OffsetOfLoader 于是，
                            ; es:bx = BaseOfLoader: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 =
                            ; BaseOfLoader*10h+100
  cld
  mov dx, 10h

LABEL_SEARCH_FOR_LOADERBIN:
  cmp dx, 0                                  ; ┐ 循环次数控制，如果
  jz LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR      ; ├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。其中用到的变量和字符串的定义请参见代码 4-6。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 4-6　（节自\chapter4\b\boot.asm）

```
……

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
RootDirSectors              equ 14      ; 根目录占用空间
SectorNoOfRootDirectory     equ 19      ; Root Directory 的第一个扇区号

……

;变量
wRootDirSizeForLoop dw  RootDirSectors  ; Root Dir 占用的扇区数，
                                        ; 在循环中会递减至零.
wSectorNo           dw  0               ; 要读取的扇区号
bOdd                db  0               ; 奇数还是偶数

;字符串
LoaderFileName      db  "LOADER  BIN", 0    ; Loader.bin 之文件名
; 为简化代码，下面每个字符串的长度均为 MessageLength
MessageLength       equ 9
BootMessage:        db  "Booting□□" ; 9 字节，不够则用空格补齐. 序号 0
Message1            db  "Ready.□□□" ; 9 字节，不够则用空格补齐. 序号 1
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 4-7），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 4-7　（节自\chapter4\b\boot.asm）

```
DispStr:
    mov ax, MessageLength
    mul dh
    add ax, BootMessage
```

```
mov bp, ax              ; ┐
mov ax, ds              ; ├ es: bp = 串地址
mov es, ax              ; ┘
mov cx, MessageLength   ; cx = 串长度
mov ax, 01301h          ; ah = 13,  al = 01h
mov bx, 0007h           ; 页号为 0(bh = 0) 黑底白字(bl = 07h)
mov dl, 0
int 10h                 ; int 10h
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.com。

如今虽然 Boot.com 有了，但我们还没有软盘以及其中的 Loader。这很容易，用 Virtual PC 捕获某个.IMG 文件，将它格式化，然后把 Loader.bin 复制到虚拟软盘（*.IMG）上。来到 Boot.com 所在的文件夹就可以执行了。不过直接执行 Boot.com 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 Turbo Debugger 调试一下，在标号 LABEL_FILENAME_FOUND 后面的 jmp $语句设置断点，执行情况如图 4-7 所示。

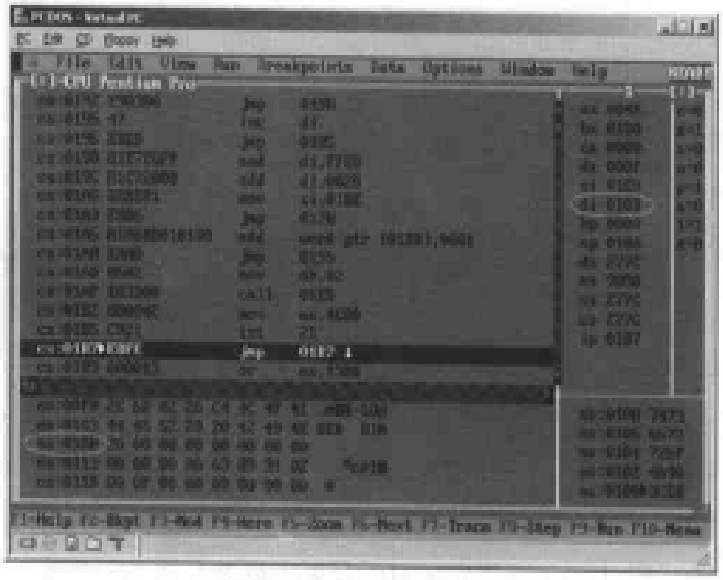

图 4-7　用 TD 进行调试

椭圆形标出来的地方是寄存器 di 的值以及 es:di 指向的内存情况。可以看出，此时 es:di 指向 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。

等确信找到文件这一模块是正确的后，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 4-8）。

代码 4-8　（节自\chapter4\c\boot.asm）

```
GetFATEntry:
    push    es
    push    bx
    push    ax

    mov     ax, BaseOfLoader    ; ┐
    sub     ax, 0100h           ; ├ 在 BaseOfLoader 后面留出 4KB 空
    mov     es, ax              ; ┘ 间用于存放 FAT

    pop     ax
    mov     byte [bOdd], 0
    mov     bx, 3
    mul     bx                  ; dx:ax = ax * 3
    mov     bx, 2
    div     bx                  ; dx:ax / 2  ==>  ax <- 商, dx <- 余数
    cmp     dx, 0
    jz      LABEL_EVEN
    mov     byte [bOdd], 1
LABEL_EVEN:                     ; 偶数
    xor     dx, dx              ; 现在 ax 中是 FATEntry 在 FAT 中的偏移量.
    mov     bx, [BPB_BytsPerSec]
    div     bx                  ; dx:ax / BPB_BytsPerSec  ==> ax <- 商
    push    dx
    mov     bx, 0               ; bx <- 0 于是, es:bx =
```

```
                                ; (BaseOfLoader -100):00
    add     ax, SectorNoOfFAT1  ; 此句执行之后的 ax 就是 FATEntry
                                ; 所在的扇区号
    mov     cl, 2
    call    ReadSector          ; 读取 FATEntry 所在的扇区，一次读两个
    pop     dx
    add     bx, dx
    mov     ax, [es:bx]
    cmp     byte [bOdd], 1
    jnz     LABEL_EVEN_2
    shr     ax, 4
LABEL_EVEN_2:
    and     ax, 0FFFh
LABEL_GET_FAT_ENRY_OK:
    pop     bx
    pop     es
    ret
```

其中，SectorNoOfFAT1 的定义如下：

```
SectorNoOfFAT1  equ 1    ; FAT1 的第一个扇区号（= BPB_RsvdSecCnt）
```

连同前面的 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 项的这段代码中惟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 4-5 以及相应的文字。

在 4.1.1 节中我们还提到，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 4-9 （节自\chapter4\c\boot.asm）

```
    ......
LABEL_FILENAME_FOUND:           ; 找到 Loader.bin 后便来到这里继续
    mov     ax, RootDirSectors
    and     di, 0FFE0h              ; di -> 当前条目的开始
    add     di, 01Ah                ; di -> 首 Sector
    mov     cx, word [es:di]
    push    cx                      ; 保存此 Sector 在 FAT 中的序号
    add     cx, ax
    add     cx, DeltaSectorNo       ; 这句完成时 cl 里面变成 Loader.bin
```

```
                                        ; 的起始扇区号
    mov     ax, BaseOfLoader
    mov     es, ax                      ; es <- BaseOfLoader
    mov     bx, OffsetOfLoader          ; bx <- OffsetOfLoader 于是, es:bx
                                        ; = BaseOfLoader:OffsetOfLoader
    mov     ax, cx                      ; ax <  Sector 号
LABEL_GOON_LOADING_FILE:
    push    ax              ; ┐
    push    bx              ; │
    mov     ah, 0Eh         ; │每读一个扇区就在"Booting  "后面打一个点,
                            ; │形成这样的效果:
    mov     al, '.'         ; │Booting ......
    mov     bl, 0Fh         ; │
    int     10h             ; │
    pop     bx              ; │
    pop     ax              ; ┘
    mov     cl, 1
    call    ReadSector
    pop     ax              ; 取出此 Sector 在 FAT 中的序号
    call    GetFATEntry
    cmp     ax, 0FFFh
    jz      LABEL_FILE_LOADED
    push    ax              ; 保存 Sector 在 FAT 中的序号
    mov     dx, RootDirSectors
    add     ax, dx
    add     ax, DeltaSectorNo
    add     bx, [BPB_BytsPerSec]
    jmp     LABEL_GOON_LOADING_FILE
LABEL_FILE_LOADED:
    ......
```

在代码 4-9 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 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，根目录区占用的空间是 RootDirSectors，即 14 个扇区，虽然根目录区的开始扇区号是 19，但我们却要用“X+RootDirSectors+19–2”算出来的扇区号才是“33”这个正确的值。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 4-10　（节自\chapter4\c\boot.asm）

```
......
LABEL_FILE_LOADED:
    jmp BaseOfLoader:OffsetOfLoader
......
```

好了，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 4-8 所示。

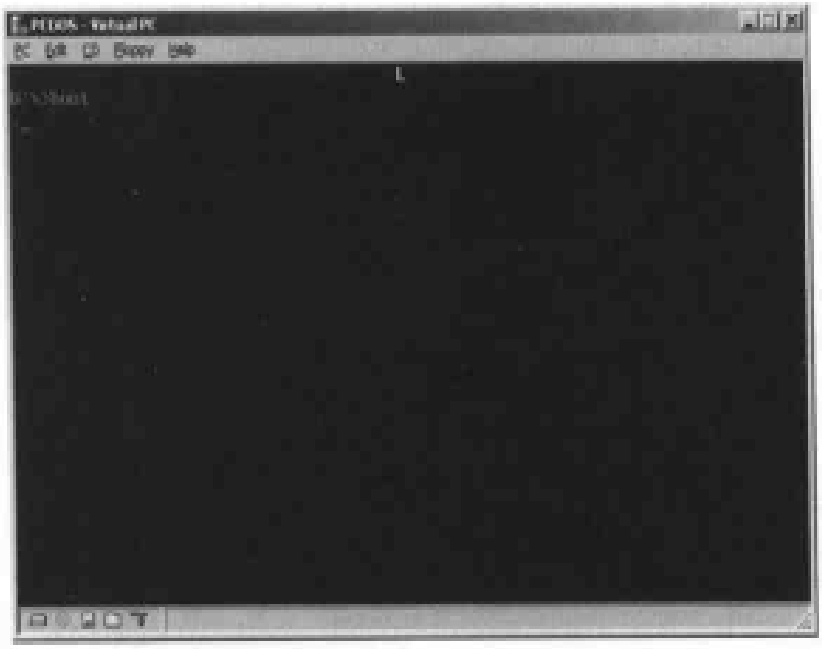

图 4-8　Loader 开始执行（在 DOS 下加载）

果然，如我们所预料的那样，看到了字符 L。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（别忘了编译前注释掉_BOOT_DEBUG_宏）。运行结果如图 4-9 所示。

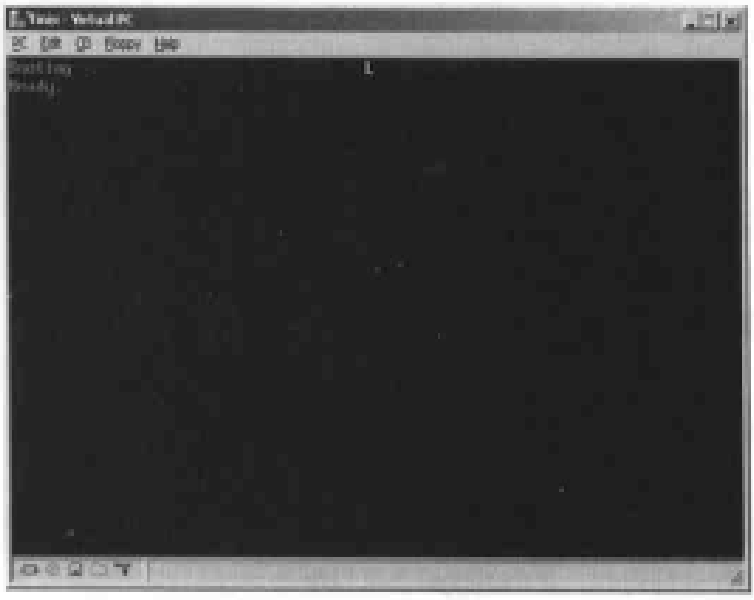

图 4-9　Loader 开始执行

代码 4-11 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。

代码 4-11　（节自\chapter4\c\boot.asm）

```
......
mov ss, ax
mov sp, BaseOfStack

; 清屏
mov ax, 0600h          ; ah = 6,  al = 0h
mov bx, 0700h          ; 黑底白字(BL = 07h)
mov cx, 0              ; 左上角: (0, 0)
mov dx, 0184fh         ; 右下角: (80, 50)
int 10h                ; int 10h

mov dh, 0              ; "Booting  "
call DispStr           ; 显示字符串
......
```

代码 4-12　（节自\chapter4\c\boot.asm）

```
......
LABEL_FILE_LOADED:
    mov dh, 1              ; "Ready."
    call DispStr           ; 显示字符串
```

```
    jmp BaseOfLoader:OffsetOfLoader
……
```

代码 4-12 实现的是在全部加载完毕准备跳入 Loader 之前打印字符串“Ready.”。

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 章中形成的代码 pmtest9.asm 改名为 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 4-10 所示。

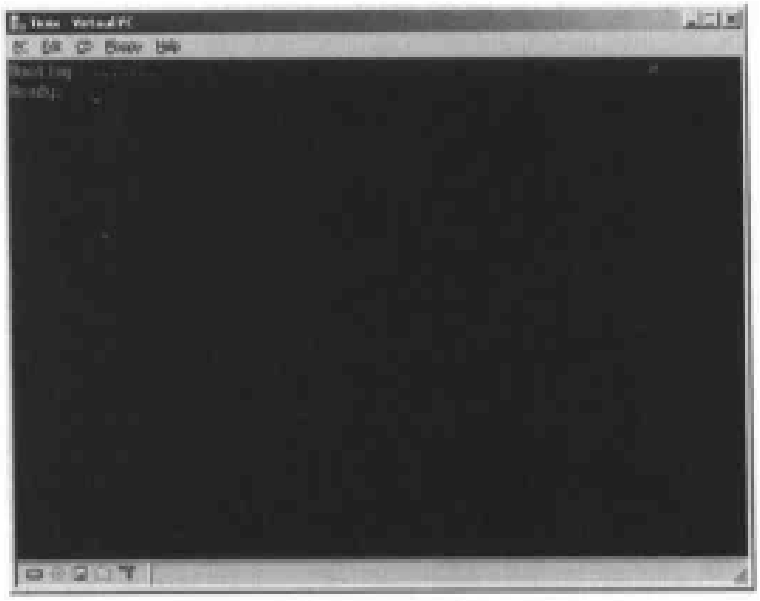

图 4-10　操作系统在保护模式下运行

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 4-10 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。

第5章

内核雏形

文武之道，一张一弛。

——礼记

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，而且将来要在Linux编译链接代码，我们也还没做过相应的工作，甚至还不知道用什么命令、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。现在就让我们来到Linux，写一个尽可能小但足以做加载试验的"内核"。

5.1 用NASM在Linux下写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Linux下我们仍然离不开汇编。还好，我们选择了NASM这一在各种平台都能使用的工具。下面就来体验一下用NASM在Linux下编程的感觉。

代码 5-1 （\chapter5\a\hello.asm）

```
[section .data]          ; 数据在此
strHello    db      "Hello, world!", 0Ah
STRLEN      equ     $ - strHello
[section .text]          ; 代码在此
global _start            ; 我们必须导出 _start 这个入口，以便让链接器识别
_start:
     mov edx, STRLEN
     mov ecx, strHello
     mov ebx, 1
     mov eax, 4          ; sys_write
     int 0x80            ; 系统调用
     mov ebx, 0
     mov eax, 1          ; sys_exit
     int 0x80            ; 系统调用
```

在 Linux 下编译链接及执行的情况如图 5-1 所示（每执行一步都用 ls 命令查看文件的增加情况）。

图 5-1 在 Linux 下编译、链接并执行 hello.asm

正如图 5-1 中所显示的，编译时使用命令：

```
[root@XXX XXX]# nasm -f elf hello.asm -o hello.o
```

选项“-f elf”指定了输出文件的格式为 ELF 格式。

链接用的命令是这样的：

```
[root@XXX XXX]# ld -s hello.o -o hello
```

选项“-s”中字母 s 是 strip 的简写，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。

接下来就是执行生成的可执行代码了，结果当然是成功。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（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 488 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 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 5-2 所示。

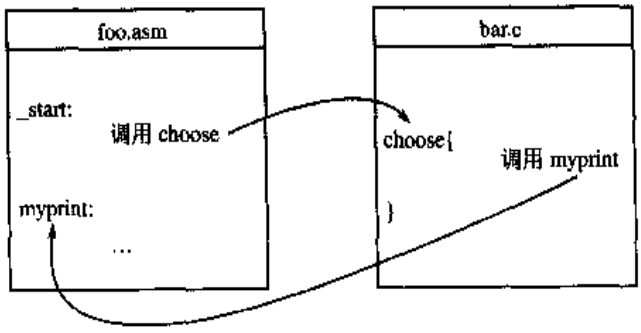

图 5-2 foo.asm 和 bar.c 之间的调用关系

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）

```
extern choose           ; int choose(int a, int b);
[section .data]         ; 数据在此
num1st dd 3
num2nd dd 4
[section .text]         ; 代码在此
global _start           ; 我们必须导出 _start 这个入口，以便让链接器识别
global myprint          ; 导出这个函数让 bar.c 使用
_start:                 ; 入口
    push num2nd         ; ┐
    push num1st         ; │
    call choose         ; ├ choose(num1st, num2nd);
    add esp, 4          ; ┘
    mov ebx, 0
    mov eax, 1          ; sys_exit
    int 0x80            ; 系统调用

myprint:                ; void myprint(char* msg, int len)
    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()，所以这里要用关键字global将其导出。

（2）由于用到本文件外定义的函数choose()，所以这里要用关键字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如图 5-3 所示。

图 5-3 编译链接并运行第一个汇编和 C 混合编写的程序

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 5-4 所示。

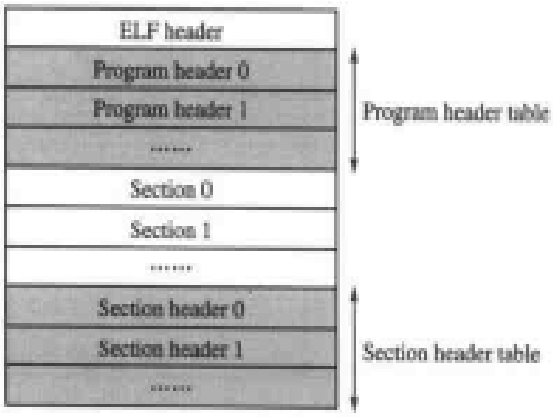

图 5-4 ELF 文件概览

从图5-4中可以看出，ELF文件由4部分组成：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 5-4 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 的格式如图 5-5 所示。其中各类型的说明见表 5-1。

```
#define EI_NIDENT          16
typedef struct{
        unsigned char      e_ident[EI_NIDENT];
        Elf32_Half         e_type;
        Elf32_Half         e_machine;
        Elf32_word         e_version;
        Elf32_Addr         e_entry;
        Elf32_Off          e_phoff;
        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;
```

图 5-5 ELF header 的格式

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 的开头如图 5-6 所示。

```
Offset   0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000000 7F 45 4C 46 01 01 01 00  00 00 00 00 00 00 00 00  .ELF............
00000010 02 00 03 00 01 00 00 00  80 80 04 08 34 00 00 00  ........II..4...
00000020 98 01 00 00 00 00 00 00  34 00 20 00 02 00 28 00  ?......4. ...(..
00000030 07 00 06 00 01 00 00 00  00 00 00 00 00 80 04 08  ................
00000040 00 80 04 08 00 01 00 00  00 01 00 00 05 00 00 00  .I..............
```

图 5-6　foobar（00h~4Fh）

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 0x8048080。
- 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 0x198。
- 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，这里有两个。
- 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 7 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 6，表示第 6 个节包含节名称。

在这里我们看到，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。我们首先看一下如图 5-7 所示的 Program header 数据结构。

```
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;
```

图 5-7 Program header

实际上，Program header 描述的是系统准备程序运行而需要的一个段（Segment）或其他信息。这样说有点抽象，让我们来看看 foobar 的情况（见图 5-8）。程序头表中共有两项（e_phnum=2），偏移分别是 0x34~0x53 和 0x54~0x73。

```
Offset   0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000030 07 00 06 00 01 00 00 00  00 00 00 00 00 80 04 08  ..............I..
00000040 00 80 04 08 00 01 00 00  00 01 00 00 05 00 00 00  .I..............
00000050 00 10 00 00 01 00 00 00  10 01 00 00 10 91 04 08  .............?..
00000060 10 91 04 08 08 00 00 00  08 00 00 00 06 00 00 00  .?..............
00000070 00 10 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000080 68 14 91 04 08 68 10 91  04 08 E8 29 00 00 00 81  h.?.h.?.?...I...
```

图 5-8 foobar（30h~8Fh）

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 中共有两个 Program header，其取值如表 5-2 所示。

表 5-2　foobar 中的 Program header

名　称	Program header 0	Program header 1
p_type	0x1	0x1
p_offset	0x0	0x110
p_vaddr	0x8048000	0x8049110
p_paddr	0x8048000	0x8049110
p_filesz	0x10D	0x8
p_memsz	0x10D	0x8
p_flags	0x5	0x6
p_align	0x1000	0x1000

根据这些信息，我们很容易知道 foobar（文件大小为 2B0h 字节）在加载进内存之后的情形，正如图 5-9 所描绘的那样。

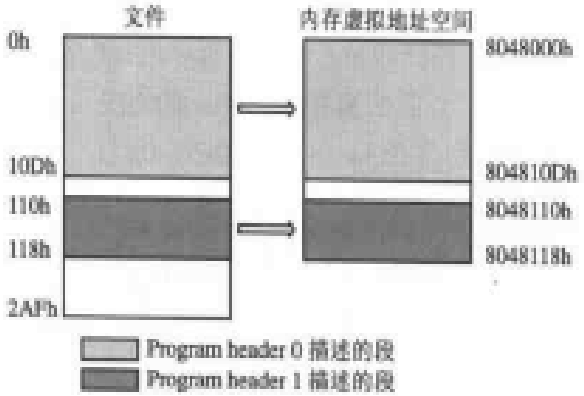

图 5-9　foobar 加载情况

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 5-4），在两个文件的开头相应的位置分别包含进去。

代码 5-4 （\chapter5\c\fat12hdr.inc）

```
; ------------------------------------------------------------
; FAT12 磁盘的头
; ------------------------------------------------------------
BS_OEMName      DB  'ForrestY'      ; OEM String, 必须 8 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  'Tinix0.01 '        ; 卷标, 必须 11 字节
BS_FileSysType  DB  'FAT12 '            ; 文件系统类型, 必须 8 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
; ------------------------------------------------------------
FATSz                   equ 9   ; BPB_FATSz16
RootDirSectors          equ 14  ; 根目录占用空间:
SectorNoOfRootDirectory equ 19  ; Root Directory 的第一个扇区号
SectorNoOfFAT1          equ 1   ; FAT1 的第一个扇区号
DeltaSectorNo           equ 17
```

这样一来，boot.asm 开头部分的代码就应该变成代码 5-5 所示的样子。

代码 5-5 （节自\chapter5\c\boot.asm）

```
    jmp short LABEL_START       ; Start to boot.
    nop                         ; 这个 nop 不可少
%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 5-6）。

代码 5-6 （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

%include    "fat12hdr.inc"
```

```
LABEL_START:                    ; <--- 从这里开始 *************
    mov     ax, cs
    mov     ds, ax
    mov     es, ax
    mov     ss, ax
    mov     sp, BaseOfStack

    mov     dh, 0               ; "Loading  "
    call    DispStr             ; 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
    dec     word [wRootDirSizeForLoop]          ; ┘ 是否已经读完

    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  ; es:di -> BaseOfKernelFile
    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
    dec     dx                                  ; ┘
    mov     cx, 11
LABEL_CMP_FILENAME:
    cmp     cx, 0                               ; ┐
    jz      LABEL_FILENAME_FOUND                ; ├ 循环次数控制
    dec     cx                                  ; ┘
    lodsb                                       ; ds:si -> al
    cmp     al, byte [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                  ; di &= e0 为了让它是
                                        ; 20h 的倍数
    add     di, 20h
    mov     si, KernelFileName
    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
    jmp     $                           ; 没有找到 Kernel.bin, 死循环
                                        ; 在这里

LABEL_FILENAME_FOUND:                   ; 找到 Kernel.bin 后到这里继续
    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

    add     di, 01Ah                    ; di -> 首 Sector
    mov     cx, word [es:di]
    push    cx                          ; 保存此 Sector 在 FAT 中的序号
    add     cx, ax
    add     cx, DeltaSectorNo           ; 这时 cl 里面是 Loader.bin 的
                                        ; 起始扇区号
    mov     ax, BaseOfKernelFile
    mov     es, ax                      ; es <- BaseOfKernelFile
    mov     bx, OffsetOfKernelFile      ; bx <- OffsetOfKernelFile
    mov     ax, cx                      ; ax <- Sector 号
```

```
LABEL_GOON_LOADING_FILE:
    push    ax          ; ┐
    push    bx          ; │
    mov     ah, 0Eh     ; │ 每读一个扇区就在"Loading  "后面打一个点,
                        ; │ 形成这样的效果:
    mov     al, '.'     ; │ Loading ......
    mov     bl, 0Fh     ; │
    int     10h         ; │
    pop     bx          ; │
    pop     ax          ; ┘

    mov     cl, 1
    call    ReadSector
    pop     ax                  ; 取出此 Sector 在 FAT 中的序号
    call    GetFATEntry
    cmp     ax, 0FFFh
    jz      LABEL_FILE_LOADED
    push    ax                  ; 保存 Sector 在 FAT 中的序号
    mov     dx, RootDirSectors
    add     ax, dx
    add     ax, DeltaSectorNo
    add     bx, [BPB_BytsPerSec]
    jmp     LABEL_GOON_LOADING_FILE
LABEL_FILE_LOADED:
    call    KillMotor           ; 关闭软驱马达
    mov     dh, 1               ; "Ready."
    call    DispStr             ; 显示字符串

    jmp     $
```

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（见代码 5-7），用来关闭软驱马达，不然软驱的灯会一直亮着。

代码 5-7 关闭软驱马达（节自\chapter5\c\loader.asm）

```
; 关闭软驱马达
KillMotor:
    push    dx
    mov     dx, 03F2h
    mov     al, 0
```

```
    out     dx, al
    pop     dx
    ret
```

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 5-10 所示的情况，“No KERNEL”字样会被显示出来。

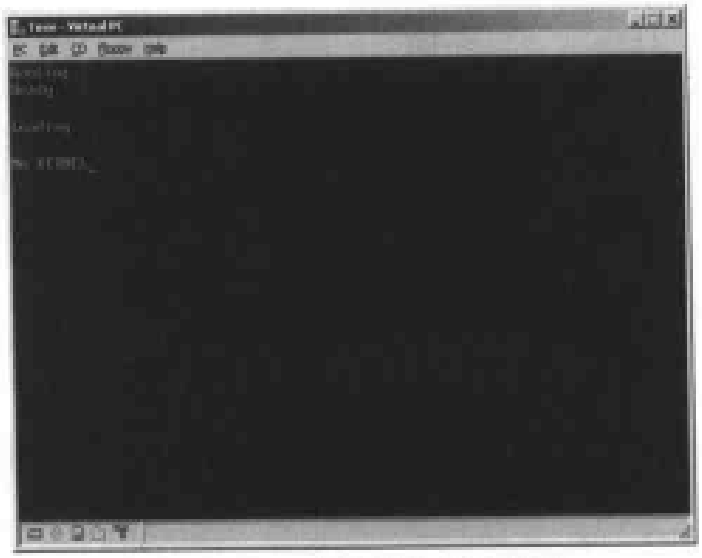

图 5-10 Loader 运行情况（无内核时）

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 5-8）。

显示字符时涉及到内存操作，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：

```
[root@XXX XXX]# nasm -f elf -o kernel.o kernel.asm
[root@XXX XXX]# ld -s -o kernel.bin kernel.o
```

这样将会生成可执行代码 Kernel.bin。我们在 Virtual PC 中用虚拟 DOS 将它复制到虚拟软盘中。再运行一遍，就不再是“No KERNEL”了，结果如图 5-11 所示。

代码 5-8 还算不上内核的内核雏形（\chapter5\c\kernel.asm）

```
[section .text]

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 * 22 + 39) * 2)], ax ; 屏幕第 22 行，第 39 列
jmp $
```

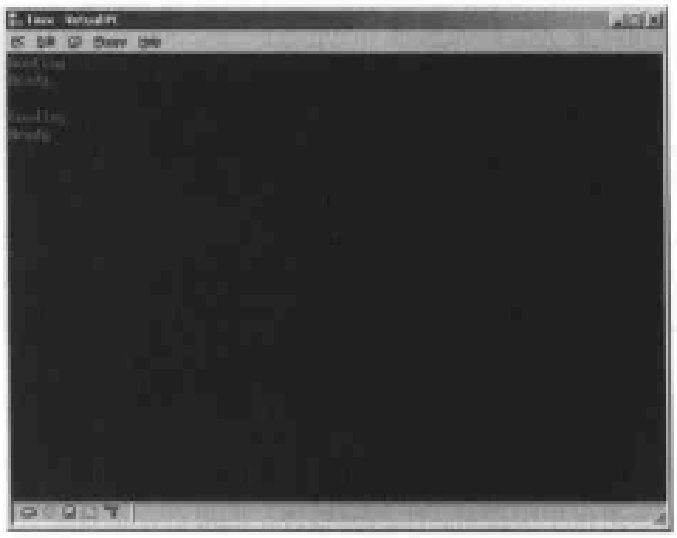

图 5-11 Loader 运行情况（有内核时）

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（见代码 5-9）。

代码 5-9 GDT（节自\chapter5\d\loader.asm）

```
.....
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"

%include "pm.inc"

LABEL_GDT: Descriptor 0, 0, 0   ; 空描述符
; 0 ~ 4G 的可执行段：
LABEL_DESC_FLAT_C:  Descriptor 0, 0fffffh, DA_CR|DA_32|DA_LIMIT_4K
; 0 ~ 4G 的可读写段：
LABEL_DESC_FLAT_RW: Descriptor 0, 0fffffh,DA_DRW|DA_ 32|DA_LIMIT_4K
```

```
; 指向显存的段:
LABEL_DESC_VIDEO:  Descriptor  0B8000h, 0ffffh, DA_DRW | DA_DPL3

GdtLen      equ $ - LABEL_GDT
GdtPtr      dw  GdtLen                                  ; 段界限
            dd  BaseOfLoaderPhyAddr + LABEL_GDT         ; 基地址

; 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

LABEL_START:
    ......
```

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 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：

标号（变量）的物理地址 = BaseOfLoader × 10h + 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 5-10）。

代码 5-10　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

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
```

同时，在 boot.asm 和 loader.asm 中分别用一句 %include "load.inc" 将其包含。

在代码 5-10 中可以看到，我们定义了一个宏 BaseOfLoaderPhyAddr 用以代替 BaseOfLoader×10h，它在代码 5-9 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（见代码 5-11）。

代码 5-11 32 位代码段（节自\chapter5\d\loader.asm）

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

LABEL_PM_START:
    mov ah, 0Fh                          ; 0000: 黑底    1111: 白字
    mov al, 'P'
    mov [gs:((80 * 0 + 39) * 2)], ax     ; 屏幕第 0 行, 第 39 列
    jmp $
```

进入保护模式的代码是再熟悉不过了，我们已经重复了许多次（见代码 5-12）。

代码 5-12 进入保护模式（节自\chapter5\d\loader.asm）

```
    ......
LABEL_FILE_LOADED:
    call    KillMotor       ; 关闭软驱马达
    mov     dh, 1           ; "Ready."
    call    DispStr         ; 显示字符串

; 下面准备跳入保护模式 -------------------------------------------
; 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 SelectorFlatC:(BaseOfLoaderPhyAddr+LABEL_PM_START)
    ......
```

运行，结果如图 5-12 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：

图 5-12　Loader 进入保护模式

代码 5-13　初始化各个寄存器（节自\chapter5\d\loader.asm）

```
LABEL_PM_START:
    mov ax, SelectorVideo
    mov gs, ax
    mov ax, SelectorFlatRW
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov ss, ax
    mov esp, TopOfStack
    ......
```

其中，TopOfStack 的定义是这样的：

代码 5-14　堆栈定义（节自\chapter5\d\loader.asm）

```
[SECTION .data1]
ALIGN 32
LABEL_DATA:
StackSpace: times 1024 db 0
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 的开头再增加一些代码：

代码 5-15 得到内存数的代码（节自\chapter5\d\loader.asm）

```
    ; 得到内存数
    mov     ebx, 0          ; ebx = 后续值，开始时需为 0
    mov     di, _MemChkBuf  ; es:di 指向一个地址范围描述符结构
                            ; (Address Range Descriptor Structure)
.MemChkLoop:
    mov     eax, 0E820h     ; eax = 0000E820h
    mov     ecx, 20         ; ecx = 地址范围描述符结构的大小
    mov     edx, 0534D4150h ; edx = 'SMAP'
    int     15h             ; int 15h
    jc      .MemChkFail
    add     di, 20
    inc     dword [_dwMCRAcount]    ; dwMCRAcount = ARDS 的个数
    cmp     ebx, 0
    jne     .MemChkLoop
    jmp     .MemChkOK
.MemChkFail:
    mov     dword [_dwMCRAcount], 0
.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 5-16 显示内存信息的函数 DispMemInfo（节自\chapter5\d\loader.asm）

```
; 显示内存信息 ---------------------------------------------------------
DispMemInfo:
    push    esi
    push    edi
    push    ecx

    mov     esi, MemChkBuf
    mov     ecx, [dwMCRNumber]  ; for(int i=0;i<{MCRNumber];i++) /* 每
                                ; 次得到一个 ARDS 结构 */
.loop:                          ; {
    mov     edx, 5              ; for(int j=0;j<5;j++)  /* 每次得到一
                                ; 个 ARDS 中的成员，共 5 个 */
    mov     edi, ARDStruct      ; {
.1:                             ;
```

```
        push    dword [esi]         ;
        call    DispInt             ; DispInt(MemChkBuf[j*4]); /* 显示一个
                                    ; 成员 */
        pop     eax                 ;
        stosd                       ; ARDStruct[j*4] = MemChkBuf[j*4];
        add     esi, 4              ;
        dec     edx                 ;
        cmp     edx, 0              ;
        jnz     .1                  ; }

        call    DispReturn          ; printf("\n");

        cmp     dword [dwType], 1   ; if(Type == AddressRangeMemory) /*
                                    ; AddressRangeMemory    : 1
                                    ; AddressRangeReserved  : 2    */
        jne     .2                  ; {
        mov     eax, [dwBaseAddrLow];
        add     eax, [dwLengthLow]  ;
        cmp     eax, [dwMemSize]    ; if(BaseAddrLow+LengthLow>MemSize)
        jb      .2                  ;
        mov     [dwMemSize], eax    ; MemSize = BaseAddrLow + LengthLow;
.2:                                 ; }
        loop    .loop               ; }
                                    ;
        call    DispReturn          ; printf("\n");
        push    szRAMSize           ;
        call    DispStr             ; printf("RAM size:");
        add     esp, 4              ;
                                    ;
        push    dword [dwMemSize]   ;
        call    DispInt             ; DispInt(MemSize);
        add     esp, 4              ;

        pop     ecx
        pop     edi
        pop     esi
        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了一个 DispStr，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）。页目录和页表的位置这样定义：

代码 5-17 页目录和页表的位置（节自\chapter5\d\load.inc）

```
......
PageDirBase equ 100000h     ; 页目录开始地址: 1MB
PageTblBase equ 101000h     ; 页表开始地址:   1MB + 4KB
......
```

代码 5-18 用来启动分页的函数 SetupPaging（节自\chapter5\d\loader.asm）

```
SetupPaging:
    xor     edx, edx
    mov     eax, [dwMemSize]
    mov     ebx, 400000h    ; 400000h = 4MB = 4096 * 1024B, 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，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 4KB 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
```

在以上代码中使用的字符串和变量是这样定义的：

代码 5-19 符号定义（节自\chapter5\d\loader.asm）

```
    LABEL_DATA:
    ; 实模式下使用这些符号
    ; 字符串
    _szMemChkTitle: db  "BaseAddrL BaseAddrH LengthLow LengthHigh Type",
0Ah, 0
    _szRAMSize:         db  "RAM size:", 0
    _szReturn:          db  0Ah, 0
    ;; 变量
    _dwMCRNumber:       dd  0                   ; Memory Check Result
    _dwDispPos:         dd  (80 * 6 + 0) * 2    ; 屏幕第 6 行，第 0 列。
    _dwMemSize:         dd  0
    _ARDStruct:     ; Address Range Descriptor Structure
        _dwBaseAddrLow:     dd  0
        _dwBaseAddrHigh:    dd  0
        _dwLengthLow:       dd  0
        _dwLengthHigh:      dd  0
        _dwType:            dd  0
    _MemChkBuf: times   256 db  0
    ;
    ;; 保护模式下使用这些符号
    szMemChkTitle       equ BaseOfLoaderPhyAddr + _szMemChkTitle
    szRAMSize           equ BaseOfLoaderPhyAddr + _szRAMSize
```

```
szReturn            equ BaseOfLoaderPhyAddr + _szReturn
dwDispPos           equ BaseOfLoaderPhyAddr + _dwDispPos
dwMemSize           equ BaseOfLoaderPhyAddr + _dwMemSize
dwMCRNumber         equ BaseOfLoaderPhyAddr + _dwMCRNumber
ARDStruct           equ BaseOfLoaderPhyAddr + _ARDStruct
    dwBaseAddrLow   equ BaseOfLoaderPhyAddr + _dwBaseAddrLow
    dwBaseAddrHigh  equ BaseOfLoaderPhyAddr + _dwBaseAddrHigh
    dwLengthLow     equ BaseOfLoaderPhyAddr + _dwLengthLow
    dwLengthHigh    equ BaseOfLoaderPhyAddr + _dwLengthHigh
    dwType          equ BaseOfLoaderPhyAddr + _dwType
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 5-20 显示内存信息并启动分页（节自\chapter5\d\loader.asm）

```
push    szMemChkTitle
call    DispStr
add     esp, 4
call    DispMemInfo
call    SetupPaging
```

代码 5-20 中除了有调用 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 5-13 所示。

图 5-13 Loader 列出内存情况

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。

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：

```
[root@XXX XXX]# nasm -f elf -o kernel.o kernel.asm
[root@XXX XXX]# 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。

表 5-3 Kernel.bin 的 ELF Header

ELF Header		
项　目	值	说　明
e_ident	…	
e_type	2 H	可执行文件
e_machine	3 H	80386
e_version	1 H	

续表

ELF Header		
项　目	值	说　明
e_entry	30400 H	入口地址
e_phoff	34 H	Program header table 在文件中的偏移量
e_shoff	44C H	Section header table 在文件中的偏移量
e_flags	0 H	
e_ehsize	34 H	ELF header 大小
e_phentsize	20 H	每一个 Program header 大小 20H 字节
e_phnum	1 H	Program header table 中只有 1 个条目
e_shentsize	28 H	每一个 Section header 大小 28H 字节
e_shnum	5 H	Section header table 中有 5 个条目
e_shstrndx	4 H	包含节名称的字符串表是第 4 个节

表 5-4　Kernel.bin 的 Program Header

Program Header（只此一个）		
项　目	值	说　明
p_type	1 H	PT_LOAD
p_offset	0 H	段的第一个字节在文件中的偏移
p_vaddr	30000 H	段的第一个字节在内存中的虚拟地址
p_paddr	30000 H	
p_filesz	40D H	段在文件中的长度
p_memsz	40D H	段在内存中的长度
p_flags	5 H	
p_align	1000 H	

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);
```

也就是说，我们应该把文件从开头开始共 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。我们来看一下如图 5-14 所示的 Kernel.bin 的内容。

图 5-14 就是 Kernel.bin，其中被省略号省去的部分都是 0。从中可以看出，从 400h 到 40Dh 是仅有的代码，看一下代码 5-8，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。

下面是调用 InitKernel 的语句：

代码 5-21　（节自\chapter5\e\loader.asm）

```
......
mov al, 'P'
mov [gs:((80 * 0 + 39) * 2)], ax    ; 屏幕第 0 行，第 39 列。

call InitKernel
......
```

```
Offset   | 0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F |
00000000 | 7F 45 4C 46 01 01 01 00  00 00 00 00 00 00 00 00 | .ELF............
00000010 | 02 00 03 00 01 00 00 00  00 04 03 00 34 00 00 00 | ............4...
00000020 | 4C 04 00 00 00 00 00 00  34 00 20 00 01 00 28 00 | L.......4. ...(.
00000030 | 05 00 04 00 01 00 00 00  00 00 00 00 00 00 00 03 | ................
00000040 | 00 00 03 00 00 04 00 00  00 04 00 00 05 00 00 00 | ................
00000050 | 00 10 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
00000060 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
00000070 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
00000080 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
00000090 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000000A0 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
         |          ⋮                                       |          ⋮
00000380 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000003C0 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000003D0 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000003E0 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000003F0 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
00000400 | B4 0F B0 4B 65 66 A3 EE  00 00 00 EB FE 00 54 68 | ?嘛ef?n ...隆.Th
00000410 | 65 20 4E 65 74 77 69 64  65 20 41 73 73 65 6D 62 | e Netwide Assemb
00000420 | 6C 65 72 20 30 2E 39 38  2E 33 35 00 00 2E 73 68 | ler 0.98.35...sh
00000430 | 73 74 72 74 61 62 00 2E  74 65 78 74 00 2E 62 73 | strtab..text..bs
00000440 | 73 00 2E 63 6F 6D 6D 65  6E 74 00 00 00 00 00 00 | s..comment......
00000450 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
00000460 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
00000470 | 00 00 00 00 0B 00 00 00  01 00 00 00 06 00 00 00 | ................
00000480 | 00 04 03 00 00 04 00 00  00 00 00 00 00 00 00 00 | ................
00000490 | 00 00 00 00 10 00 00 00  00 00 00 00 11 00 00 00 | ................
000004A0 | 01 00 00 00 01 00 00 00  10 14 03 00 00 04 00 00 | ................
000004B0 | 00 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00 | ................
000004C0 | 00 00 00 00 16 00 00 00  01 00 00 00 00 00 00 00 | ................
000004D0 | 00 00 00 00 00 04 00 00  1F 00 00 00 00 00 00 00 | ................
000004E0 | 00 00 00 00 01 00 00 00  00 00 00 00 01 00 00 00 | ................
000004F0 | 03 00 00 00 00 00 00 00  00 00 00 00 2C 04 00 00 | ................
00000500 | 1F 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00 | ................
00000510 | 00 00 00 00                                      | ....
```

图 5-14　Kernel.bin

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 Kernel.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。

代码 5-22　InitKernel（节自\chapter5\e\loader.asm）

```
InitKernel:
```

```
        xor     esi, esi
        ; ecx <- pELFHdr->e_phnum:
        mov     cx, word [BaseOfKernelFilePhyAddr + 2Ch]
        movzx   ecx, cx
        ; esi <- pELFHdr->e_phoff:
        mov     esi, [BaseOfKernelFilePhyAddr + 1Ch]
        ; esi <- OffsetOfKernel + pELFHdr->e_phoff:
        add     esi, BaseOfKernelFilePhyAddr
.Begin:
        mov     eax, [esi + 0]
        cmp     eax, 0                      ; PT_NULL
        jz      .NoAction
        ; memcpy(  (void*)(pPHdr->p_vaddr), uchCode + pPHdr->p_offset,
        ;         pPHdr->p_filesz;
        push    dword [esi + 010h]          ; size
        mov     eax, [esi + 04h]
        add     eax, BaseOfKernelFilePhyAddr
        push    eax                         ; src
        push    dword [esi + 08h]           ; dst
        call    MemCpy                      ;
        add     esp, 12                     ;

.NoAction:
        add     esi, 020h
        dec     ecx
        jnz     .Begin

        ret
```

图 5-15 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 3-32 和表 3-10 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。之前说过，通过对代码中宏定义的一点修改，我们可以让引导程序运行在 DOS 下，如果有错误可以很方便地进行调试。如果你亲自实践的话，这种调试一定是少不了的。（关于 Loader 的调试，详情请见 5.4.5 节。）

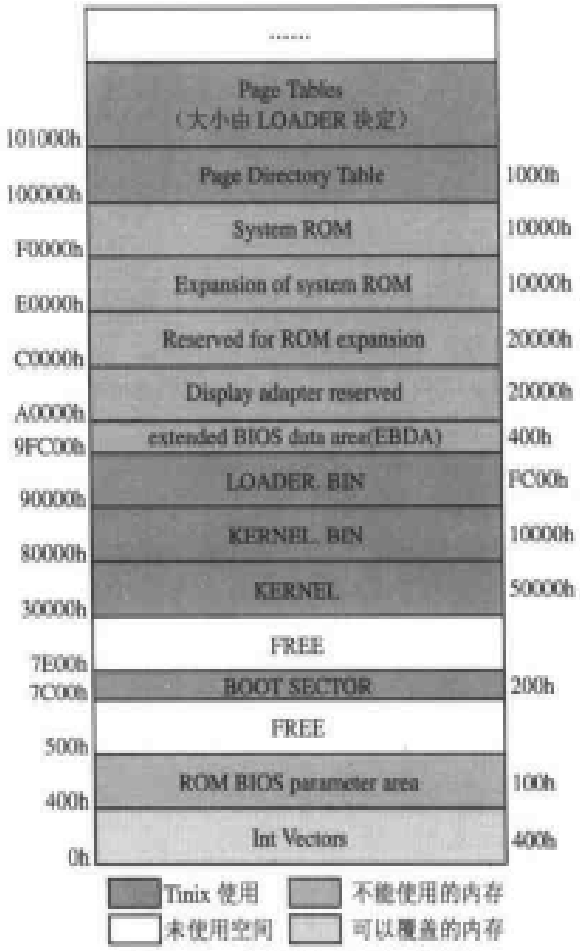

图 5-15　内存使用分布示意图

那么，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。

而且，我可以事先告诉你，内核文件超过 64KB 也不是一件非常容易的事。

好了，现在内存各部分的使用情况相信你已经很明了了。Tinix 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 Tinix 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 向内核跳转（节自\chapter5\e\loader.asm）

```
call    InitKernel
jmp     SelectorFlatC:KernelEntryPointPhyAddr
```

其中，KernelEntryPointPhyAddr 定义在 load.inc 中，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 5-16 所示。

图 5-16 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（如图 5-17 所示），在内核中我们需要这些信息。

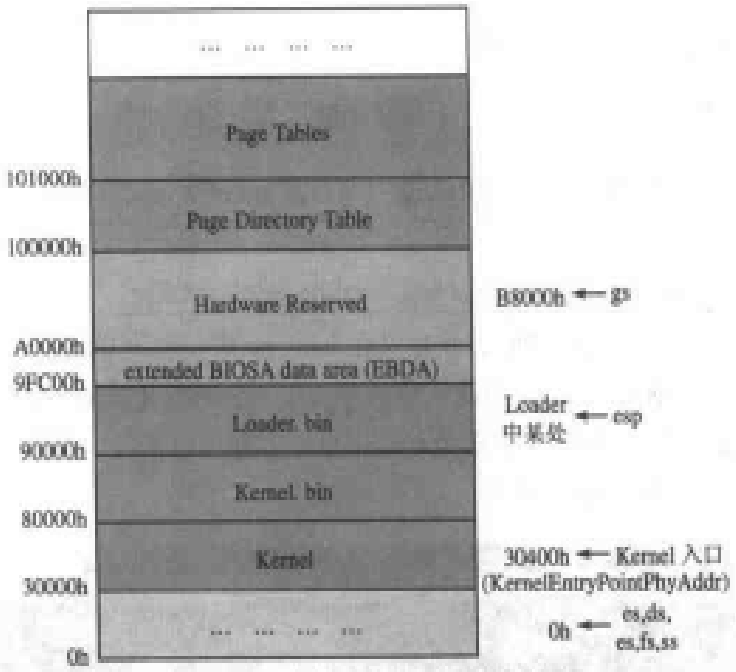

图 5-17　进入内核时寄存器情况示意图

如图 5-17 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 5-13）。

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。

5.4.5　操作系统的调试方法

编写操作系统和普通应用程序的最大区别在于操作系统更难于调试。这就大大增加了工作的难度，即便是一个很小的错误，如果无法进行有效的调试，可能发现起来也非常困难。不过还好，难于调试并非无法调试，运用 Bochs 的调试功能以及古董级的 DOS 下的调试工具，你会发现，调试本身也是一件有趣的事情。好，我们先来看一下，假如 Loader 出了问题该如何解决。

5.4.5.1 Turbo Debugger

在上文中提到，编写 Loader 的时候很可能需要调试，其实很简单，只需要把 boot.asm 中“%define _BOOT_DEBUG_”这个定义打开，编译成.COM 文件，就可以随时在 DOS 下进行调试了。

执行命令：

```
X:\>td boot.com
```

进入 Turbo Debugger 的调试界面，如图 5-18 所示。

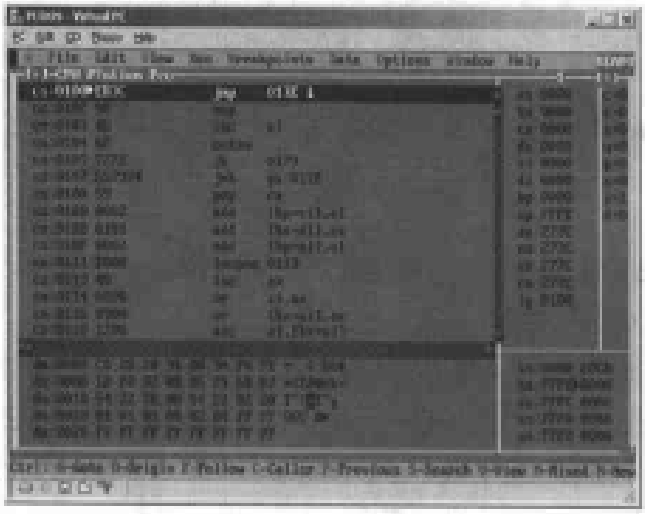

图 5-18 调试界面（1）

我们在 4.1.4 节中已经介绍过调试了（参见图 4-7），同样，让我们找到向 Loader 跳转的语句，让程序执行到这里，如图 5-19 所示。

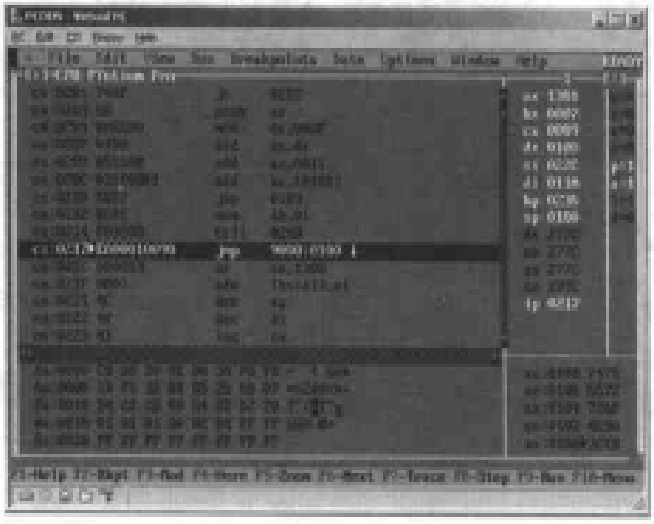

图 5-19 调试界面（2）

下一步就将跳转到 Loader 中，按一下 F8 键，呈现图 5-20 所示的界面。

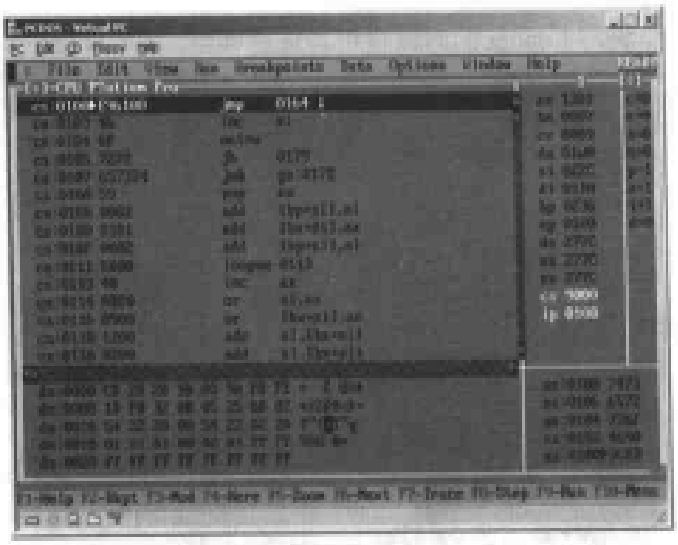

图 5-20　调试界面（3）

从图 5-20 中我们看到，cs 和 ip 的值分别变成了 9000h 和 0100h，这说明当前执行的指令位于 9000:0100 处，没错，这正是 Loader 的入口。

下面的事情就很简单了，单步执行按 F8 键，设置断点按 F2 键，惟一麻烦的一点是，调试不是源代码级的。虽然源代码也是汇编，但从调试窗口中看上去毕竟有一点不同，有时候可能找起来不那么一目了然。还好，细心一点，这不是什么大问题。

比如，如果发现得到内存信息的模块工作有问题，我们在调用 15h 号中断的位置设置断点，如图 5-21 所示。

图 5-21　调试界面（4）

TD 的缺点是无法进行保护模式下的调试，不过，Loader 中在保护模式下所做的工作

大部分是之前做过的，轻车熟路，基本没什么问题。

你一定觉得这个说法不令人满意，可能会问，一旦有问题怎么办？不要担心，我们有另外的工具，那就是 Bochs。

5.4.5.2 Bochsdbg

我们在第 2 章中曾经提到，Bochs 可以对操作系统进行调试，现在我们就来演示一下。

假设我们已经有了一个虚拟软盘 TINIX.IMG，它的引导扇区已经写好，Loader.bin 也已经放在软盘的根目录下，我们在 TINIX.IMG 所在目录下写一个 bochsrc.bxrc，如下所示：

代码 5-24 （\chapter5\f\bochsrc.bxrc）

```
# how much memory the emulated machine will have
megs: 32

# filename of ROM images
romimage: file=$BXSHARE/BIOS-bochs-latest, address=0xf0000
vgaromimage: $BXSHARE/VGABIOS-elpin-2.40

# what disk images will be used
floppya: 1_44=TINIX.IMG, status=inserted

# choose the boot disk.
boot: a

# where do we send log messages?
log: bochsout.txt

# disable the mouse, since Tinix is text only
mouse: enabled=0

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

这个文件再简单不过，勿需解释。我们再写一个批处理文件 godbg.bat（参见\chapter5\f\godbg.bat），内容只有一行：

```
D:\Progra~1\Bochs-~1.1\bochsdbg.exe -q -f bochsrc.bxrc
```

好了，就这么简单的两步，我们已经可以开始调试了。

举个最简单的例子，假设我们发现虚拟机的屏幕上只打印了字符 P 而没有 K，这说

明在向内核跳转的时候出现了问题，为了查出错误到底在何处，我们模拟一下调试。

运行 godbg.bat，屏幕出现两个窗口，如图 5-22 所示。

图 5-22　用 Bochs 调试 Tinix（1）

在上方的窗口的提示符后可以输入调试命令，由于我们怀疑在向内核跳转时程序出现问题，首先在地址 0x30400 处设置断点，设置断点、运行以及查看寄存器值等命令如表 5-5 所示。

表 5-5　部分 Bochs 的调试命令

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
查看寄存器信息	dump_cpu	dump_cpu
查看内存内容	xp /nuf addr	xp /40bx 0x9013e
反汇编一段内存	disassemble start end	disassemble 0x30400 0x30400
反汇编执行的每一条指令	trace-on	trace-on

输入指令设置断点：

```
<bochs:1> b 0x30400
```

用 info break 指令显示一下当前断点情况：

```
<bochs:2> info break
```

```
Num Type           Disp Enb Address
  1 pbreakpoint    keep y   0x00030400
```

可以看出，断点设置成功。

下面就用 c 指令执行程序直到 0x30400 处：

```
<bochs:3> c
(0) Breakpoint 1, 0x30400 in ?? ()
Next at t=2500464
(0) [0x00030400] 0008:00030400 (unk. ctxt): mov ah, 0xf       ; b40f
```

对照代码 5-8 我们知道，mov ah, 0xf 正是内核入口处第一条指令。接下来的工作你一定知道该怎么做了，根据实际情况需要，通过对内存、CPU 等内容的检查，以及单步执行、断点等方法，无论什么错误都难不倒我们。

下面的内容便是部分指令执行情况节选（输入的指令使用了黑体字），<bochs:1>在 0x30400 处设置了一个断点；<bochs:2>将断点信息打印出来；<bochs:3>用 c 指令让程序一直执行，直到我们设置的断点处（0x30400）；<bochs:4>将 CPU 寄存器信息打印出来；<bochs:5>则根据 gdtr 的值将整个 GDT 打印出来；<bochs:6>则是把 0x30400 到 0x3040D 这一部分二进制反汇编；<bochs:7>用了 s 指令前进一步。

```
========================================================================
                    Bochs x86 Emulator 2.1.1
                       February 08, 2004
========================================================================
00000000000i[     ] reading configuration from bochsrc.bxrc
00000000000i[     ] installing win32 module as the Bochs GUI
00000000000i[     ] Warning: no rc file specified.
00000000000i[     ] using log file bochsout.txt
Next at t=0
(0) context not implemented because BX_HAVE_HASH_MAP=0
[0x000ffff0] f000:fff0 (unk. ctxt): jmp f000:e05b     ; ea5be000f0
<bochs:1> b 0x30400
<bochs:2> info break
Num Type           Disp Enb Address
  1 pbreakpoint    keep y   0x00030400
<bochs:3> c
(0) Breakpoint 1, 0x30400 in ?? ()
Next at t=2500456
(0) [0x00030400] 0008:00030400 (unk. ctxt): mov ah, 0xf   ; b40f
<bochs:4> dump_cpu
eax:0x30000
```

```
ebx:0x400
ecx:0x0
edx:0x0
ebp:0x2b7
esi:0x80054
edi:0x209000
esp:0x9175d
eflags:0x42
eip:0x30400
cs:s=0x8, dl=0xffff, dh=0xcf9a00, valid=1
ss:s=0x10, dl=0xffff, dh=0xcf9300, valid=7
ds:s=0x10, dl=0xffff, dh=0xcf9200, valid=7
es:s=0x10, dl=0xffff, dh=0xcf9300, valid=5
fs:s=0x10, dl=0xffff, dh=0xcf9300, valid=1
gs:s=0x1b, dl=0x8000ffff, dh=0xf20b, valid=5
ldtr:s=0x0, dl=0x0, dh=0x0, valid=0
tr:s=0x0, dl=0x0, dh=0x0, valid=0
gdtr:base=0x9013e, limit=0x20
idtr:base=0x0, limit=0x3ff
dr0:0x0
dr1:0x0
dr2:0x0
dr3:0x0
dr6:0xffff0ff0
dr7:0x400
tr3:0x0
tr4:0x0
tr5:0x0
tr6:0x0
tr7:0x0
cr0:0xe0000011
cr1:0x0
cr2:0x0
cr3:0x200000
cr4:0x0
inhibit_mask:0
done
<bochs:5> xp /32bx 0x9013e
[bochs]:
0x0009013e <bogus+  0>:  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x00
```

```
    0x00090146 <bogus+  8>:  0xff  0xff  0x00  0x00  0x00  0x9a  0xcf
0x00
    0x0009014e <bogus+ 16>:  0xff  0xff  0x00  0x00  0x00  0x93  0xcf
0x00
    0x00090156 <bogus+ 24>:  0xff  0xff  0x00  0x80  0x0b  0xf3  0x00
0x00
    <bochs:6> disassemble 0x30400 0x3040D
    00030400: (                ): mov ah, 0xf            ; b40f
    00030402: (                ): mov al, 0x4b           ; b04b
    00030404: (                ): gs opsize mov [gs:0xee], ax ;
6566a3ee000000
    0003040b: (                ): jmp 0x3040b            ; ebfe
    <bochs:7> s
    Next at t=2500457
    (0) [0x00030402] 0008:00030402 (unk. ctxt): mov al,
0x4b            ; b04b
    <bochs:8>
```

可以看出，Bochs 的调试功能非常强大，可以获得非常详尽的信息。

图 5-23 是由整个执行过程的抓图拼接而成。

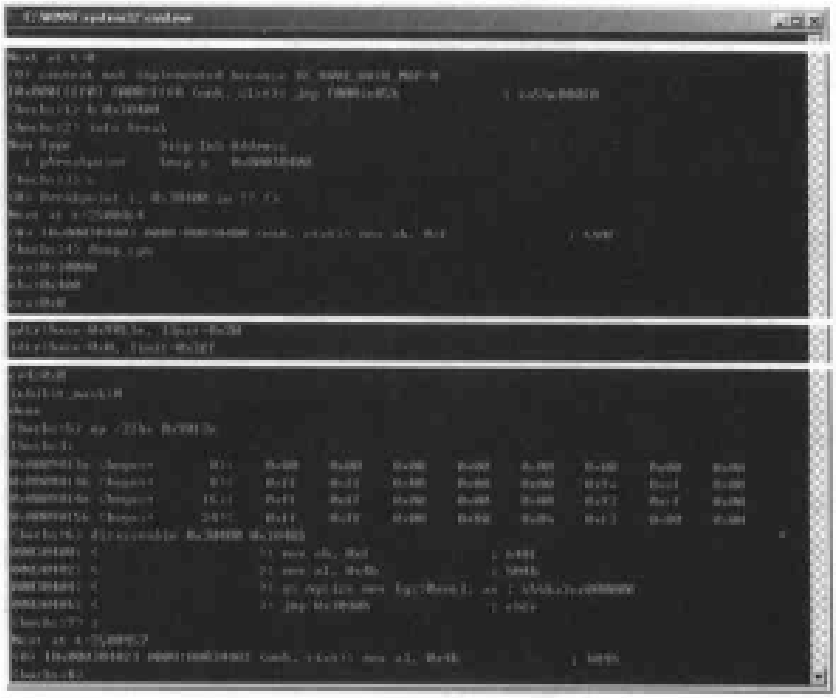

图 5-23 用 Bochs 调试 Tinix（2）

图 5-24 是 Bochs 执行到<bochs:8>时的主界面，由于显示字符 K 的指令（0x30404 处）还未执行，所以我们只看到字符 P。

图 5-24　用 Bochs 调试 Tinix（3）

从上面的例子看到，用 Bochs 可以非常方便地对我们的操作系统进行调试。而且，表 5-5 列出的指令只不过是一少部分，阅读 Bochs 的联机文档可以获得全部指令的使用介绍，运用它们，调试这一重大问题便轻松解决了。

5.5　扩充内核

不知道你现在有什么感觉，我觉得自己已经武装到牙齿，就准备冲锋陷阵了——连调试这个问题都解决了，我们还怕什么呢？

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，在此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，我们在 5.4.4 节中曾经提过，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 5-25。

代码 5-25 切换堆栈和 GDT（节自\chapter5\g\kernel.asm）

```
SELECTOR_KERNEL_CS equ 8

; 导入函数和全局变量
extern cstart
extern gdt_ptr
[SECTION .bss]
StackSpace      resb    2 * 1024
StackTop:               ; 栈顶
[section .text]         ; 代码在此
global _start           ; 导出 _start
_start:
     ; 把 esp 从 Loader 挪到 Kernel
     mov    esp, StackTop   ; 堆栈在 bss 段中
     sgdt   [gdt_ptr]       ; cstart() 中将会用到 gdt_ptr
     call   cstart          ; 在此函数中改变了 gdt_ptr，让它指向新的 GDT
     lgdt   [gdt_ref]       ; 使用新的 GDT
     jmp    SELECTOR_KERNEL_CS:csinit
csinit:                     ; 这个跳转指令强制使用刚刚初始化的结构
     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 5-26）。

代码 5-26 cstart（\chapter5\g\start.c）

```
#include "type.h"
#include "const.h"
#include "protect.h"

PUBLIC  void* memcpy(void* pDst, void* pSrc, int iSize);

PUBLIC  t_8             gdt_ptr[6];     // 0~15:Limit  16~47:Base
PUBLIC  DESCRIPTOR      gdt[GDT_SIZE];

PUBLIC void cstart()
{
    // 将 Loader 中的 GDT 复制到新的 GDT 中
    memcpy( &gdt,                       // New GDT
            (void*)(*((t_32*)(&gdt_ptr[2]))), // Base of Old GDT
```

```
                *((t_16*)(&gdt_ptr[0]))          // Limit of Old GDT
            );
    // gdt_ptr[6] 共 6 字节: 0~15:Limit  16~47:Base
    // 用做 sgdt 以及 lgdt 的参数
    t_16* p_gdt_limit   = (t_16*)(&gdt_ptr[0]);
    t_32* p_gdt_base    = (t_32*)(&gdt_ptr[2]);
    *p_gdt_limit     = GDT_SIZE * sizeof(DESCRIPTOR);
    *p_gdt_base      = (t_32)&gdt;
}
```

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，函数体我们已经很熟悉，这里就省去了。

代码 5-27 （节自\chapter5\g\string.asm）

```
[SECTION .text]

; 导出函数
global memcpy

memcpy:
    ......
```

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 5-28），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 5-28 （\chapter5\g\const.h）

```
#ifndef _TINIX_CONST_H_
#define _TINIX_CONST_H_

/* 函数类型 */
#define PUBLIC              /* PUBLIC is the opposite of PRIVATE */
#define PRIVATE     static  /* PRIVATE x limits the scope of x */

/* GDT 和 IDT 中描述符的个数   */
#define GDT_SIZE    128

#endif /* _TINIX_CONST_H_   */
```

GDT_SIZE 也定义在 cosnt.h 中。

t_8、t_16、t_32 等类型定义在 type.h 中（见代码 5-29），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 5-29 （\chapter5\g\type.h）

```
#ifndef _TINIX_TYPE_H_
#define _TINIX_TYPE_H_

typedef unsigned int        t_32;
typedef unsigned short      t_16;
typedef unsigned char       t_8;

#endif /* _TINIX_TYPE_H_ */
```

代码 5-30 （\chapter5\g\protect.h）

```
#ifndef _TINIX_PROTECT_H_
#define _TINIX_PROTECT_H_

/* 存储段描述符/系统段描述符    */
typedef struct s_descriptor/* 共 8 字节 */
{
    t_16    limit_low;          /* Limit */
    t_16    base_low;           /* Base */
    t_8     base_mid;           /* Base */
    t_8     attr1;              /* P(1) DPL(2) DT(1) TYPE(4) */
    t_8     limit_high_attr2;   /* G(1) D(1) 0(1) AVL(1) LimitHigh(4) */
    t_8     base_high;          /* Base */
}DESCRIPTOR;

#endif /* _TINIX_PROTECT_H_ */
```

Descriptor 用来表示描述符，它类似于在 pm.inc 中定义的宏 Descriptor（见代码 5-30）

介绍到这里，想必读者对代码 5-26 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 5-25 和代码 5-8 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：

```
[root@XXX XXX]# nasm -f elf -o kernel.o kernel.asm
[root@XXX XXX]# nasm -f elf -o string.o string.asm
[root@XXX XXX]# gcc -c -o start.o start.c
[root@XXX XXX]# ld -s -Ttext 0x30400 -o kernel.bin kernel.o string.o
start.o
```

运行，结果如图 5-25 所示。

图 5-25　切换堆栈和 GDT 后运行情况

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 klib.asm 中（见代码 5-31）。

像 memcpy 一样，只需要简单声明一下，在 C 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 5-32）。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 5-32 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 5-31　（\chapter5\g\klib.asm）

```
[SECTION .data]
disp_pos    dd  0
[SECTION .text]
```

```
; 导出函数
global disp_str

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
```

代码 5-32 （节自\chapter5\g\start.c）

```
......
PUBLIC void disp_str (char * pszInfo);
```

```
……
PUBLIC void cstart()
{
   disp_str   ("\n\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 klib.asm）：

```
[root@XXX XXX]# nasm -f elf -o kernel.o kernel.asm
[root@XXX XXX]# nasm -f elf -o string.o string.asm
[root@XXX XXX]# nasm -f elf -o klib.o klib.asm
[root@XXX XXX]# gcc -c -fno-builtin -o start.o start.c
[root@XXX XXX]# ld -s -Ttext 0x30400 -o kernel.bin kernel.o string.o
start.o klib.o
```

在编译 start.c 的时候，如果不加参数-fno-builtin，你会发现出现一个警告提示：

```
[root@TinixDev TinixForBook_chapter5\g]# 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 5-26 所示。

图 5-26　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，只几炷香的功夫，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.img、.bat，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；klib.asm 和 string.asm 放在\lib 中，作为库的形像出现；kernel.asm 和 start.c 放在\kernel 里面。这样一来，结构就清晰多了，目录树如图 5-27 所示。

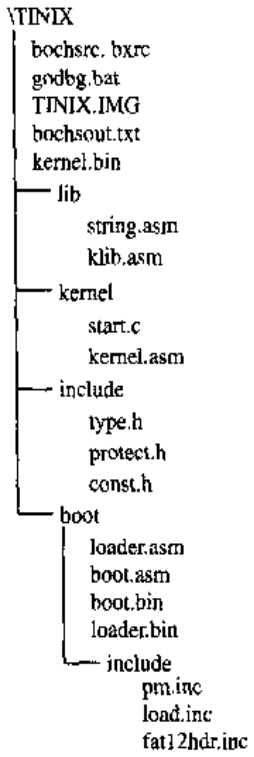

图 5-27 整理后的目录结构（请参考\chapter5\h）

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 5-26 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，如果全部说完的话篇幅可能会很长，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（见代码 5-33），把它放在目录\boot 下，可以用来编译 Boot.bin 和 Loader.bin。

代码 5-33 （节自\chapter5\h\boot\Makefile）

```
# Makefile for boot

# Programs, flags, etc.
ASM         = nasm
ASMFLAGS    =

# 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：

- 要想得到 target，需要执行命令 command。
- target 依赖于 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：

- 要想得到 Loader.bin，需要执行命令“$(ASM) $(ASMFLAGS) -o $@ $<”。
- Loader.bin 依赖于 loader.asm、./include/load.inc、./include/fat12hdr.inc、./include/pm.inc 这些文件，当这些文件中至少有一个比 Loader.bin 新时，command 被执行。

那么“$(ASM) $(ASMFLAGS) -o $@ $<”又是什么呢？其实$@和$<代表的便是 target 和 prerequisites 的第一个名字，联系前面我们说过的$(ASM)和$(ASMFLAGS)，这个命令行便等价于：

```
nasm -o loader.bin loader.asm
```

在文件中，除了 Boot.bin 和 Loader.bin 两个文件后面有冒号外，我们发现 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”，“rm -f $(TARGET)”这行命令将会被执行。根据$(TARGET)的定义，这行命令也相当于“rm -f boot.bin loader.bin”。

all 后面跟着的是 clean 和 everything，这表明如果我们执行“make all”，clean 和 everything 所表示的动作将分别被执行。图 5-28 就是 make all 执行的结果。

图 5-28 make all 的执行结果（请参考\chapter5\h\boot）

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，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。图 5-29 明白地表示了这一点。

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。在图 5-29 中可以看到，第二次运行 make 时出现这样的提示：

```
make: Nothing to be done for `everything'.
```

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。

图 5-29　make clean、make 和 make everything（请参考\chapter5\h\boot）

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 5-34）。

代码 5-34 （节自\chapter5\h\Makefile）

```
# Makefile for boot

# Programs, flags, etc.
ASM             = nasm
ASMBFLAGS       = -I boot/include

# This Program
TINIXBOOT       = boot/boot.bin boot/loader.bin

# All Phony Targets
.PHONY : everything all clean

# Default starting position
everything : $(TINIXBOOT)

all : clean everything

clean :
        rm -f $(TINIXBOOT)

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 $@ $<
```

跟代码 5-33 比较起来，代码 5-34 并没有大的改变，只是把其中的文件统统加上了路径“boot/”。另外，一个重要的细节是，ASMBFLAGS 此时变成了“-I boot/include”，而不再是原来的空值。这是因为，如果我们延用原来的空值，执行 make all 会出现以下错误提示：

```
rm -f boot/boot.bin boot/loader.bin
nasm  -o boot/boot.bin boot/boot.asm
boot/boot.asm:17: fatal: unable to open include file `./include/load.
inc'
make: *** [boot/boot.bin] Error 1
```

因为当前目录不再是“boot/”，所以它无法根据“./include/load.inc”找到“boot/include/

load.inc”。解决的办法是把 ASMBFLAGS 改成“-I boot/include”以便显式指定头文件目录，另外 boot.asm 和 loader.asm 中也要将包含头文件的代码改成代码 5-35 和代码 5-36 所示的样子。

代码 5-35　（节自\chapter5\h\boot\boot.asm）

```
......
%include    "load.inc"
......
%include    "fat12hdr.inc"
......
```

代码 5-36　（节自 chapter5\h\boot\loader.asm）

```
......
%include    "fat12hdr.inc"
%include    "load.inc"
%include    "pm.inc"
......
```

此时再运行 make，就一切正常了，结果如图 5-30 所示。

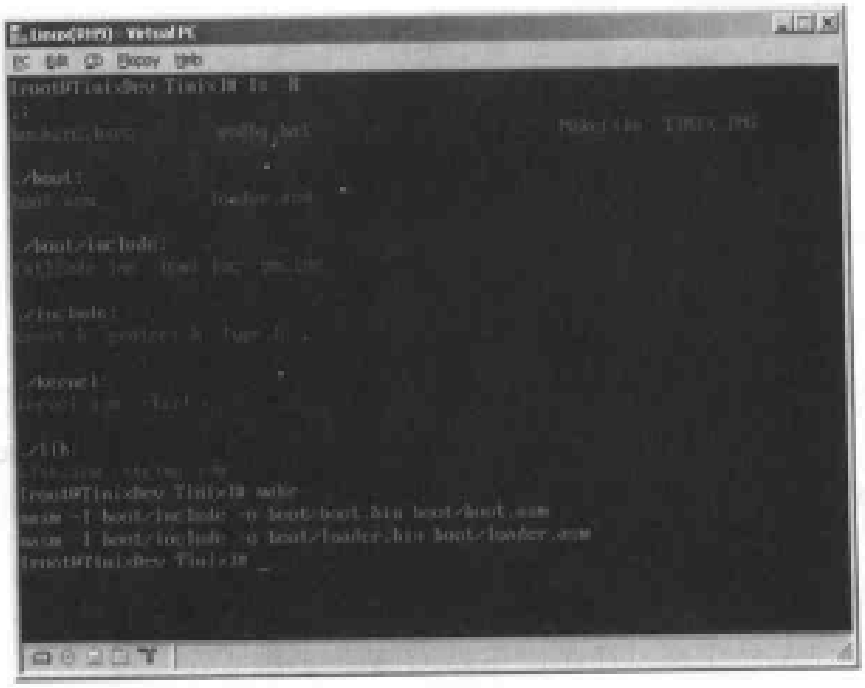

图 5-30　make 执行情况（请参考\chapter5\h\）

至此，再往下扩展 Makefile 就完全变成机械劳动了，代码 5-37 是完成后的成果，里面没有任何新鲜事物。

代码 5-37 （\chapter5\h\Makefile）

```
# Makefile for Tinix

# Entry point of Tinix
# It must be as same as '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
TINIXBOOT       = boot/boot.bin boot/loader.bin
TINIXKERNEL = kernel.bin
OBJS            = kernel/kernel.o kernel/start.o lib/klib.o \
                  lib/string.o
DASMOUTPUT      = kernel.bin.asm

# All Phony Targets
.PHONY : everything final image clean realclean disasm all buildimg

# Default starting position
everything : $(TINIXBOOT) $(TINIXKERNEL)

all : realclean everything

final : all clean

image : final buildimg
```

```
clean :
	rm -f $(OBJS)

realclean :
	rm -f $(OBJS) $(TINIXBOOT) $(TINIXKERNEL)

disasm :
	$(DASM) $(DASMFLAGS) $(TINIXKERNEL) > $(DASMOUTPUT)

# Write "boot.bin" & "loader.bin" into floppy image "TINIX.IMG"
# We assume that "TINIX.IMG" exists in current folder
buildimg :
	mount TINIX.IMG /mnt/floppy -o loop
	cp -f boot/loader.bin /mnt/floppy/
	cp -f kernel.bin /mnt/floppy
	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 $@ $<

$(TINIXKERNEL) : $(OBJS)
	$(LD) $(LDFLAGS) -o $(TINIXKERNEL) $(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/klib.o : lib/klib.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"，同时，我们将 start.c 中包含头文件的部分修改成代码 5-38 的样子。

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 Loader.bin 和 Kernel.bin 写入虚拟软盘而无需通过额外的 DOS 下的步骤进行。（把 Boot.bin 写入引导扇区的代码仍需专门的程序。）

既然如此神奇，我们就来试一下它的效果，输入 make image，执行情况如图 5-31 所示。

图 5-31 make image 的执行情况（请参考\chapter5\h\）

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，不要紧，我们试验一下就知道它的运行状况了。

代码 5-38 （节自\chapter5\h\kernel\start.c）

```
……
#include "type.h"
#include "const.h"
#include "protect.h"
……
```

来到 start.c，在 cstart()的结束处添加一行程序（见代码 5-39），如果我们运行时看到效果改变，就说明 make 运行正确。

代码 5-39 （节自\chapter5\h\kernel\start.c）

```
PUBLIC void cstart()
```

```
{
    ......
    *p_gdt_limit = GDT_SIZE * sizeof(DESCRIPTOR);
    *p_gdt_base  = (t_32)&gdt;

    disp_str("-----\"cstart\" finished-----\n");
}
```

添加这行程序的同时，你的心情是不是已经变得很激动呢？反正我觉得我自己都有点不想用 Virtual PC 的菜单加载.IMG 文件了，如果你跟我同样迫不及待，就让我们在重新 make image 之后双击 Bochs 的配置文件 bochsrc.bxrc 吧（这样可以更快看到效果）！几秒钟之后画面就出现了，如图 5-32 所示。

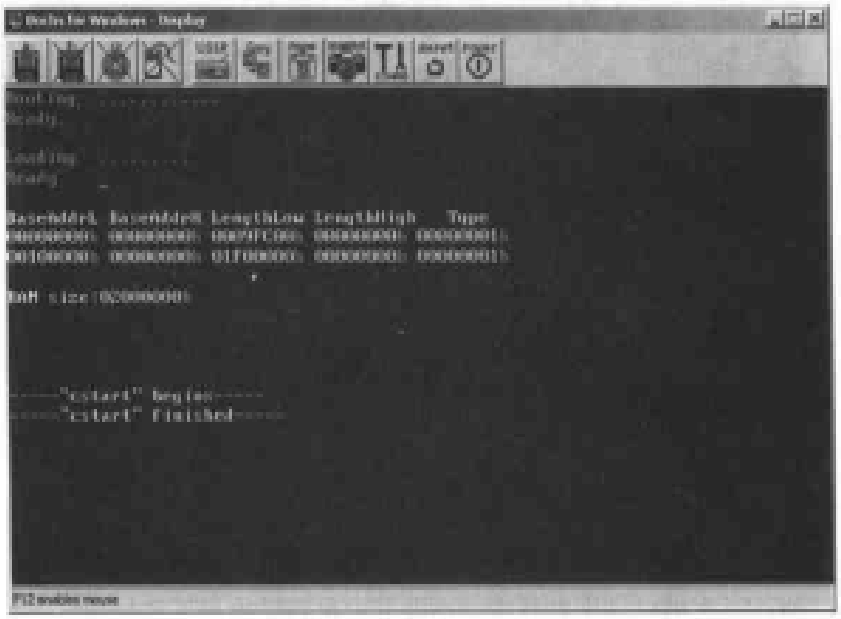

图 5-32　打印"cstart" finished 字符串（请参考\chapter5\h\）

怎么样？预期的字符串出现了！这说明我们的 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，而且不必每次都用 Virtual PC 将内核往虚拟软盘上复制了，make 已经完成了这项工作。

不过，我还是建议你在运行操作系统的时候使用 Virtual PC，因为相对来说它更接近于真正的 PC，在有些跟时间相关的程序运行过程中，这种差别能够看得到，在以后的章节中我们会遇到这种情况。

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 Tinix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5-40）。

代码5-40 （节自\chapter5\i\kernel\i8259.c）

```
#include "type.h"
#include "const.h"
#include "protect.h"
#include "proto.h"
PUBLIC void init_8259A()
{
    // Master 8259, ICW1:
    out_byte(INT_M_CTL, 0x11);
    // Slave 8259, ICW1:
    out_byte(INT_S_CTL, 0x11);
    // Master 8259, ICW2. 设置 '主8259'的中断入口地址为 0x20:
    out_byte(INT_M_CTLMASK, INT_VECTOR_IRQ0);
    // Slave 8259, ICW2. 设置 '从8259' 的中断入口地址为 0x28:
    out_byte(INT_S_CTLMASK, INT_VECTOR_IRQ8);
    // Master 8259, ICW3. IR2 对应 '从8259':
    out_byte(INT_M_CTLMASK, 0x4);
    // Slave  8259, ICW3. 对应 '主8259' 的 IR2:
    out_byte(INT_S_CTLMASK, 0x2);
    // Master 8259, ICW4:
    out_byte(INT_M_CTLMASK, 0x1);
    // Slave  8259, ICW4:
    out_byte(INT_S_CTLMASK, 0x1);
```

```
    // Master 8259, OCW1. 关闭所有中断:
    out_byte(INT_M_CTLMASK, 0xFF);
    // Slave  8259, OCW1. 屏蔽 '从 8259' 所有中断:
    out_byte(INT_S_CTLMASK, 0xFF);
}
```

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 3-38 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 5-41 和代码 5-42。

代码 5-41 （节自\chapter5\i\include\const.h）

```
/* 8259A interrupt controller ports. */
/* I/O port for interrupt controller        <Master> */
#define INT_M_CTL       0x20
/* setting bits in this port disables ints   <Master> */
#define INT_M_CTLMASK   0x21
/* I/O port for second interrupt controller  <Slave>  */
#define INT_S_CTL       0xA0
/* setting bits in this port disables ints   <Slave>  */
#define INT_S_CTLMASK   0xA1
```

代码 5-42 （节自\chapter5\i\include\protect.h）

```
/* 中断向量 */
#define INT_VECTOR_IRQ0             0x20
#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 klib.asm 中（请见代码 5-43）。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 5-43 （节自\chapter5\i\lib\klib.asm）

```
......
global out_byte
global in_byte
......
out_byte:
    mov     edx, [esp + 4]      ; port
    mov     al, [esp + 4 + 4]   ; value
    out     dx, al
```

```
    nop     ; 一点延迟
    nop
    ret

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 5-44）。可以看到，start.c 中函数 disp_str 的声明也被挪到了里面。

代码 5-44 （节自\chapter5\i\include\proto.h）

```
PUBLIC void     out_byte(t_port port, t_8 value);
PUBLIC t_8      in_byte(t_port port);
PUBLIC void     disp_str (char * pszInfo);
```

其中又使用了一个新的类型 t_port，它的定义在 include\proto.h 中（见代码 5-45）。

代码 5-45 （节自\chapter5\i\include\type.h）

```
typedef unsigned int t_port;
```

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 5-46）。

代码 5-46 Makefile 的修改（节自\chapter5\i\Makefile）

```
......
OBJS = kernel/kernel.o kernel/start.o kernel/i8259.o lib/klib.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：

```
[root@XXX XXX]# 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
[root@XXX XXX]#
```

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 5-47）。

代码 5-47　start.c（节自\chapter5\i\kernel\start.c）

```
……
#include "global.h"
……
    // idt_ptr[6] 共 6 字节：0~15:Limit  16~47:Base
    // 用做 sidt 以及 lidt 的参数
    t_16* p_idt_limit  = (t_16*)(&idt_ptr[0]);
    t_32* p_idt_base   = (t_32*)(&idt_ptr[2]);
    *p_idt_limit   = IDT_SIZE * sizeof(GATE);
    *p_idt_base    = (t_32)&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 5-48）。

代码 5-48 全局变量的声明（\chapter5\i\include\global.h）

```
/* EXTERN is defined as extern except in global.c */
#ifdef  GLOBAL_VARIABLES_HERE
#undef  EXTERN
#define EXTERN
#endif

EXTERN      t_8             gdt_ptr[6]; // 0~15:Limit  16~47:Base
EXTERN      DESCRIPTOR      gdt[GDT_SIZE];
EXTERN      t_8             idt_ptr[6]; // 0~15:Limit  16~47:Base
EXTERN      GATE            idt[IDT_SIZE];
```

EXTERN 的定义位于 const.h 中（参见代码 5-50），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 5-49）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 5-49 全局变量（\chapter5\i\kernel\global.c）

```
#define GLOBAL_VARIABLES_HERE

#include "type.h"
#include "const.h"
#include "protect.h"
#include "proto.h"
#include "global.h"
```

代码 5-50 EXTERN（节自\chapter5\i\include\const.h）

```
......
/* EXTERN is defined as extern except in global.c */
#define EXTERN extern
......
#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 5-51 GATE（节自\chapter5\i\include\protect.h）

```
/* 门描述符 */
typedef struct s_gate
{
    t_16    offset_low;     /* Offset Low */
    t_16    selector;       /* Selector */
    t_8     dcount;         /* 该字段只在调用门描述符中有效。*/
    t_8     attr;           /* P(1) DPL(2) DT(1) TYPE(4) */
    t_16    offset_high;    /* Offset High */
}GATE;
```

好了，start.c 修改完之后，我们在 kernel.asm 中添加如下两句：

代码 5-52 加载 IDT 的语句（节自\chapter5\i\kernel\kernel.asm）

```
......
extern idt_ptr
......
    lidt [idt_ptr]
......
```

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 3-11 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 5-53）。

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 3-44。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5-53 中断和异常（节自\chapter5\i\kernel\kernel.asm）

```
......
global  divide_error
global  single_step_exception
global  nmi
global  breakpoint_exception
global  overflow
global  bounds_check
global  inval_opcode
global  copr_not_available
global  double_fault
global  copr_seg_overrun
global  inval_tss
global  segment_not_present
global  stack_exception
global  general_protection
global  page_fault
global  copr_error

......

divide_error:
     push       0xFFFFFFFF  ; no err code
     push       0           ; vector_no = 0
     jmp        exception
single_step_exception:
     push       0xFFFFFFFF  ; no err code
     push       1           ; vector_no = 1
     jmp        exception
nmi:
     push       0xFFFFFFFF  ; no err code
     push       2           ; vector_no = 2
     jmp        exception
breakpoint_exception:
     push       0xFFFFFFFF  ; no err code
     push       3           ; vector_no = 3
     jmp        exception
overflow:
```

```
        push        0xFFFFFFFF  ; no err code
        push        4           ; vector_no = 4
        jmp         exception
bounds_check:
        push        0xFFFFFFFF  ; no err code
        push        5           ; vector_no = 5
        jmp         exception
inval_opcode:
        push        0xFFFFFFFF  ; no err code
        push        6           ; vector_no = 6
        jmp         exception
copr_not_available:
        push        0xFFFFFFFF  ; no err code
        push        7           ; vector_no = 7
        jmp         exception
double_fault:
        push        8           ; vector_no = 8
        jmp         exception
copr_seg_overrun:
        push        0xFFFFFFFF  ; no err code
        push        9           ; vector_no = 9
        jmp         exception
inval_tss:
        push        10          ; vector_no = A
        jmp         exception
segment_not_present:
        push        11          ; vector_no = B
        jmp         exception
stack_exception:
        push        12          ; vector_no = C
        jmp         exception
general_protection:
        push        13          ; vector_no = D
        jmp         exception
page_fault:
        push        14          ; vector_no = E
        jmp         exception
copr_error:
vpush               0xFFFFFFFF  ; no err code
        push        16          ; vector_no = 10h
        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，在这段代码的最后，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 5-54），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。

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。

代码 5-54 异常处理函数 exception_handler（节自\chapter5\i\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_description[][64] = {
            "#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 notuse.)",
                    "#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 5 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_description[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);
    }
}
```

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 5-55。

代码 5-55 函数 disp_color_str（节自\chapter5\i\lib\klib.asm）

```
disp_color_str:
    push    ebp
    mov     ebp, esp
```

```
        mov     esi, [ebp + 8]  ; pszInfo
        mov     edi, [disp_pos]
        mov     ah, [ebp + 12]  ; color
.1:
        ......
```

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 5-56。

代码 5-56 函数 disp_int（节自\chapter5\i\lib\klib.c）

```
PUBLIC void disp_int(int input)
{
    char output[16];
    itoa(output, input);
    disp_str(output);
}
```

函数 disp_int 很简单，用函数 itoa()将整数转换成字符串后显示出来。itoa()也定义在 klib.c 中，见代码 5-57。我们的 itoa()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。

代码 5-57 函数 itoa（节自\chapter5\i\lib\klib.c）

```
PUBLIC char * itoa(char * str, int num)
{
    char *  p = str;
    char ch;
    int  i;
    t_bool  flag = FALSE;
    *p++ = '0';
    *p++ = 'x';
    if(num == 0)
        *p++ = '0';
    else
        for(i=28;i>=0;i-=4){
            ch = (num >> i) & 0xF;
            if(flag || (ch > 0)){
                flag = TRUE;
                ch += '0';
                if(ch > '9'){
```

```
                        ch += 7;
                    }
                    *p++ = ch;
                }
            }
        *p = 0;
        return str;
    }
```

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 5-59），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()（见代码 5-58），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 (*t_pf_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 5-60）。

代码 5-58 函数 init_idt_desc（节自\chapter5\i\kernel\protect.c）

```
PUBLIC void init_idt_desc(unsigned char vector, t_8 desc_type,
                          t_pf_int_handler handler, unsigned
                          char privilege)
{
    GATE *   p_gate          = &idt[vector];
    t_32 base                = (t_32)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_开头的宏表示的是中断向量，表示中断门的 DA_386I Gate 定义在 protect.h 中，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 5-59 函数 init_prot()（节自\chapter5\i\kernel\protect.c）

```
PUBLIC void init_prot()
{
    init_8259A();
```

```
    // 全部初始化成中断门(没有陷阱门)
    init_idt_desc(INT_VECTOR_DIVIDE, DA_386IGate,
                  divide_error, PRIVILEGE_KRNL);
    init_idt_desc(INT_VECTOR_DEBUG, DA_386IGate,
                  single_step_exception, PRIVILEGE_KRNL);
    init_idt_desc(INT_VECTOR_NMI, DA_386IGate,
                  nmi, PRIVILEGE_KRNL);
    init_idt_desc(INT_VECTOR_BREAKPOINT, DA_386IGate,
                  breakpoint_exception, PRIVILEGE_USER);
    init_idt_desc(INT_VECTOR_OVERFLOW, DA_386IGate,
                  overflow, PRIVILEGE_USER);
    init_idt_desc(INT_VECTOR_BOUNDS, DA_386IGate,
                  bounds_check, PRIVILEGE_KRNL);
    init_idt_desc(INT_VECTOR_INVAL_OP, DA_386IGate,
                  inval_opcode, PRIVILEGE_KRNL);
    init_idt_desc(INT_VECTOR_COPROC_NOT, DA_386IGate,
                  copr_not_available, PRIVILEGE_KRNL);
    init_idt_desc(INT_VECTOR_DOUBLE_FAULT, DA_386IGate,
                  double_fault, PRIVILEGE_KRNL);
    init_idt_desc(INT_VECTOR_COPROC_SEG, DA_386IGate,
                  copr_seg_overrun, PRIVILEGE_KRNL);
    init_idt_desc(INT_VECTOR_INVAL_TSS, DA_386IGate,
                  inval_tss, PRIVILEGE_KRNL);
    init_idt_desc(INT_VECTOR_SEG_NOT, DA_386IGate,
                  segment_not_present, PRIVILEGE_KRNL);
    init_idt_desc(INT_VECTOR_STACK_FAULT, DA_386IGate,
                  stack_exception, PRIVILEGE_KRNL);
    init_idt_desc(INT_VECTOR_PROTECTION, DA_386IGate,
                  general_protection, PRIVILEGE_KRNL);
    init_idt_desc(INT_VECTOR_PAGE_FAULT, DA_386IGate,
                  page_fault, PRIVILEGE_KRNL);
    init_idt_desc(INT_VECTOR_COPROC_ERR, DA_386IGate,
                  copr_error, PRIVILEGE_KRNL);
}
```

代码 5-60 异常处理函数的声明（节自\chapter5\i\kernel\protect.c）

```
void divide_error();
void single_step_exception();
void nmi();
void breakpoint_exception();
void overflow();
```

```
void bounds_check();
void inval_opcode();
void copr_not_available();
void double_fault();
void copr_seg_overrun();
void inval_tss();
void segment_not_present();
void stack_exception();
void general_protection();
void page_fault();
void copr_error();
```

至此，设置 IDT 的代码总算添加得差不多了，我们在 start.c 中添加调用 init_prot() 的代码：

代码 5-61　调用 init_prot()（节自\chapter5\i\kernel\start.c）

```
    ......
    init_prot();
    disp_str("-----\"cstart\" finished-----\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 5-62　ud2（节自\chapter5\i\kernel\kernel.asm）

```
    ......
    csinit:
        ud2
        hlt
```

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 5-33 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 5-34 所示。

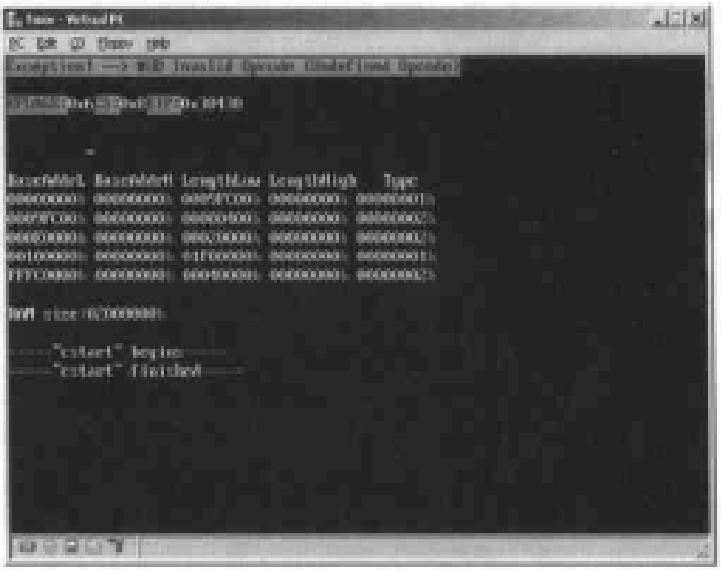

图 5-33 #UD 演示（请参考\chapter5\i\）

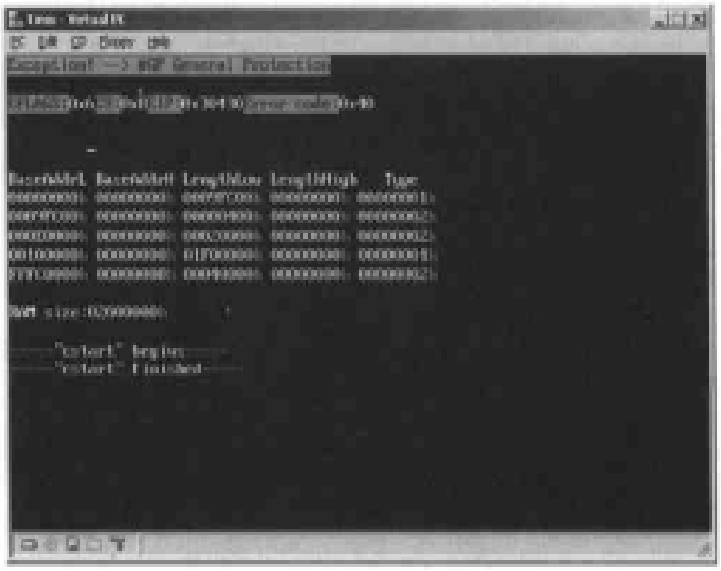

图 5-34 #GP 演示（请参考\chapter5\i\）

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 3-38，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 5-63 就是主 8259A 的中断例程，从片与此类似，就不多加叙述了。

代码 5-63 对应主 8259A 的中断例程（节自\chapter5\i\kernel\kernel.asm）

```
%macro hwint_master 1
    push    %1
    call    spurious_irq
    add     esp, 4
    hlt
%endmacro

ALIGN 16
hwint00:        ; Interrupt routine for irq 0 (the clock).
    hwint_master 0
ALIGN 16
hwint01:        ; Interrupt routine for irq 1 (keyboard)
    hwint_master 1
ALIGN 16
hwint02:        ; Interrupt routine for irq 2 (cascade!)
    hwint_master 2
ALIGN 16
hwint03:        ; Interrupt routine for irq 3 (second serial)
    hwint_master 3
ALIGN 16
hwint04:        ; Interrupt routine for irq 4 (first serial)
    hwint_master 4
ALIGN 16
hwint05:        ; Interrupt routine for irq 5 (XT winchester)
    hwint_master 5
ALIGN 16
hwint06:        ; Interrupt routine for irq 6 (floppy)
    hwint_master 6
ALIGN 16
hwint07:        ; Interrupt routine for irq 7 (printer)
    hwint_master 7
```

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 5-64 所示。

代码 5-64 函数 spurious_irq（节自\chapter5\i\kernel\i8259.c）

```
PUBLIC void spurious_irq(int irq)
{
```

```
    disp_str("spurious_irq: ");
    disp_int(irq);
    disp_str("\n");
}
```

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 5-65 设置 IDT（节自\chapter5\i\kernel\protect.c）

```
init_idt_desc(INT_VECTOR_IRQ0 + 0, DA_386IGate, hwint00,
              PRIVILEGE_KRNL);
init_idt_desc(INT_VECTOR_IRQ0 + 1, DA_386IGate, hwint01,
              PRIVILEGE_KRNL);
init_idt_desc(INT_VECTOR_IRQ0 + 2, DA_386IGate, hwint02,
              PRIVILEGE_KRNL);
init_idt_desc(INT_VECTOR_IRQ0 + 3, DA_386IGate, hwint03,
              PRIVILEGE_KRNL);
init_idt_desc(INT_VECTOR_IRQ0 + 4, DA_386IGate, hwint04,
              PRIVILEGE_KRNL);
init_idt_desc(INT_VECTOR_IRQ0 + 5, DA_386IGate, hwint05,
              PRIVILEGE_KRNL);
init_idt_desc(INT_VECTOR_IRQ0 + 6, DA_386IGate, hwint06,
              PRIVILEGE_KRNL);
init_idt_desc(INT_VECTOR_IRQ0 + 7, DA_386IGate, hwint07,
              PRIVILEGE_KRNL);
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 处做这样的修改：

代码 5-66　打开键盘中断（节自\chapter5\i\kernel\i8259.c）

```
out_byte(INT_M_CTLMASK, 0xFD); // Master 8259, OCW1.
out_byte(INT_S_CTLMASK, 0xFF); // Slave  8259, OCW1.
```

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 5-67　置 IF 位（节自\chapter5\i\kernel\kernel.asm）

```
csinit:
        ;ud2
        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 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 5-35 所示。

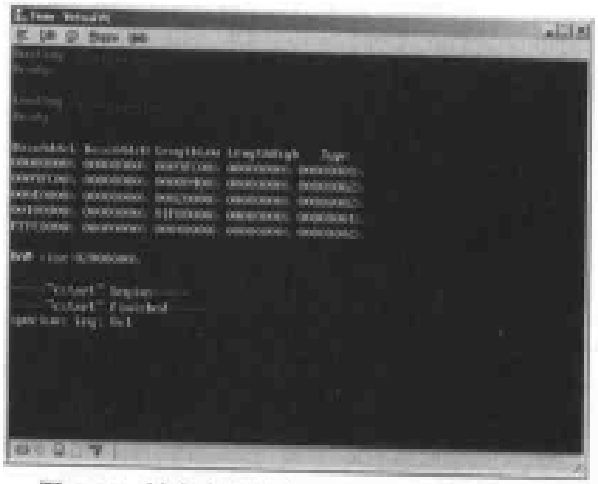

图 5-35　键盘中断发生（请参考\chapter5\i\）

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，有一些非常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。我猜你看了图 5-36 之后可能会有这样的感觉。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。

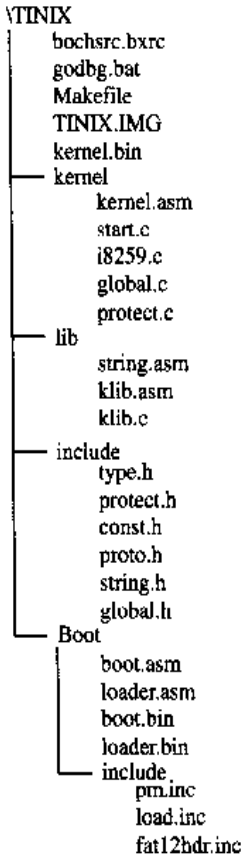

图 5-36　当前目录结构（请参考\chapter5\i\）

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。

第6章

进　　程

博学之，审问之，慎思之，明辨之，笃行之。

——礼记

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，我们先对进程有一个大概的认识。

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。

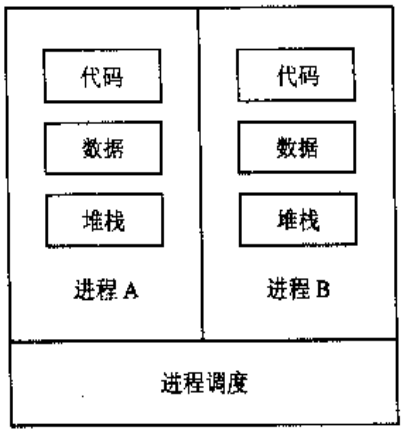

图 6-1 进程示意

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大部分情况下只有一个，哪怕我们可以有多个CPU，我们也不能每增加一个进程就增加一个CPU。这就决定了，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）。

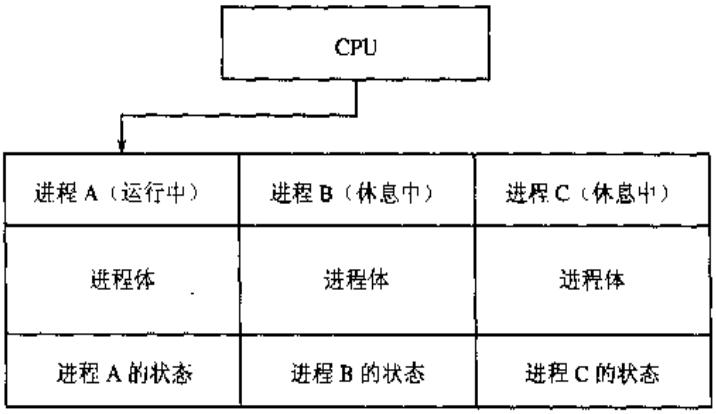

图6-2 需要一个数据结构记录进程的状态

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。

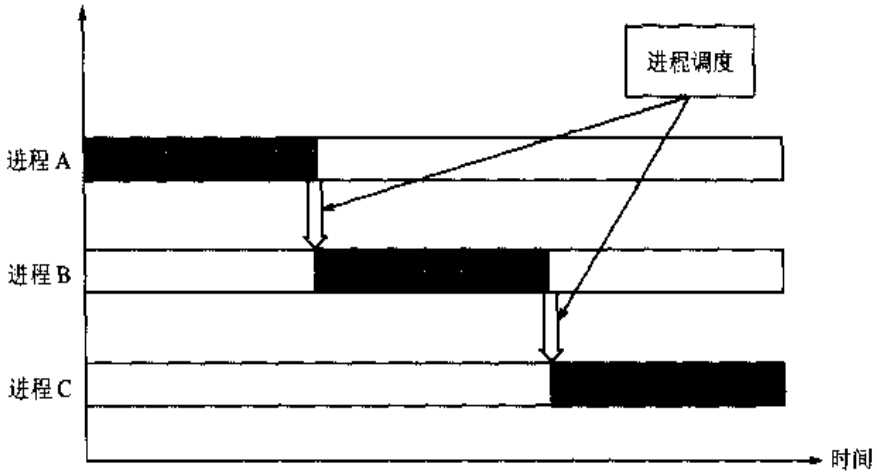

图6-3 进程切换

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

我们正在写的操作系统的名字叫做 Tinix，但前文中还一直未曾提起过这个名字的来历。实际上，笔者的初衷是试着弄清楚 Minix，于是 Try 的第 1 个字母和 Minix 的后 4 个字母。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，Tinix 的进程实现要简单得多，自成体系，而且本书给出了所有细节上的说明，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：

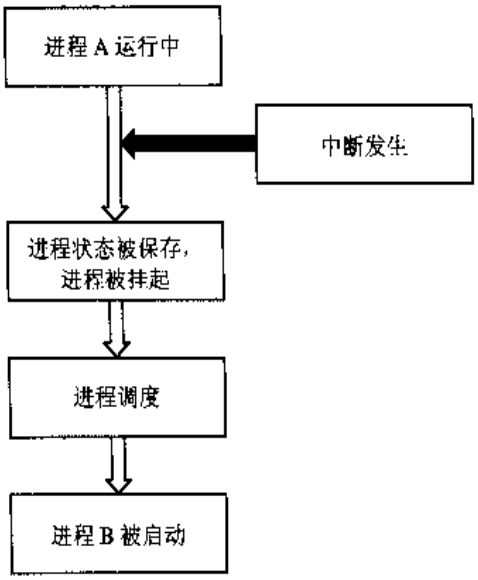

图 6-4　进程切换

（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。

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”。

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

图 6-5 进程表数组

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。

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）：

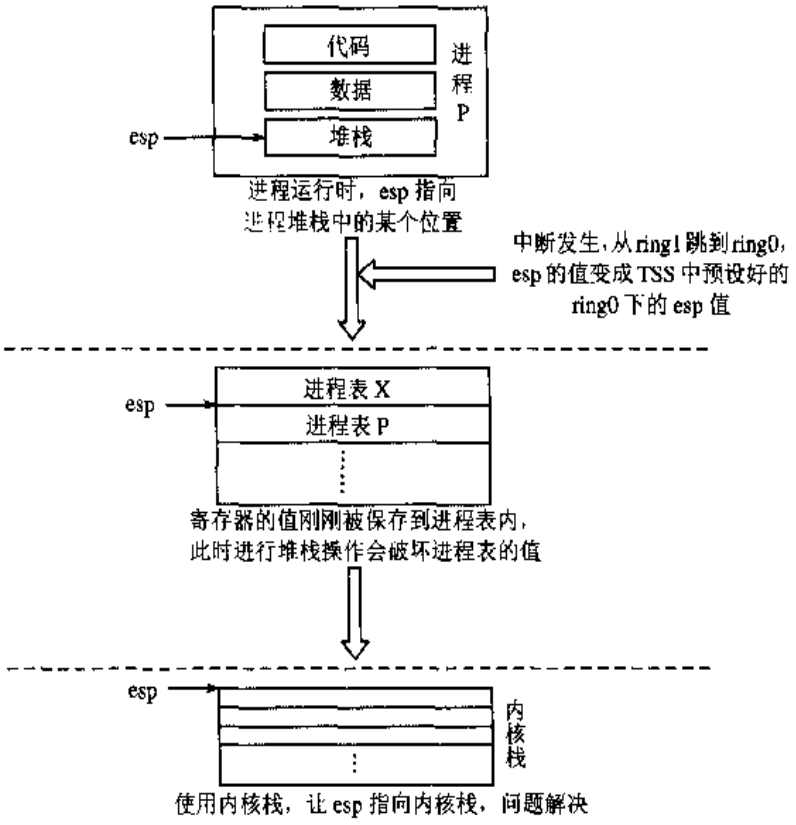

图 6-6 内核栈

- 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。
- 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\i）中 kernel.asm 的一小部分：

代码 6-1 （节自\chapter6\i\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
restart_krnl_int:
    dec     dword [k_reenter]
    pop     gs
```

```
        pop     fs
        pop     es
        pop     ds
        popad
        add     esp, 4
        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。

在文件夹\kernel 中，你会发现多了一个 main.c，里面有一个函数 tinix_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。

第二行 mov esp, [p_proc_ready]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。

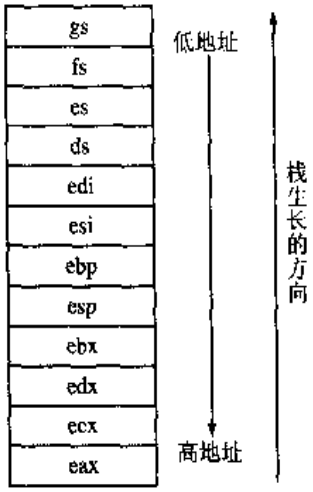

图 6-7 进程表推想

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 global.h 中，找到 p_proc_ready 的类型是 struct s_proc *，一个结构类型指针。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 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。

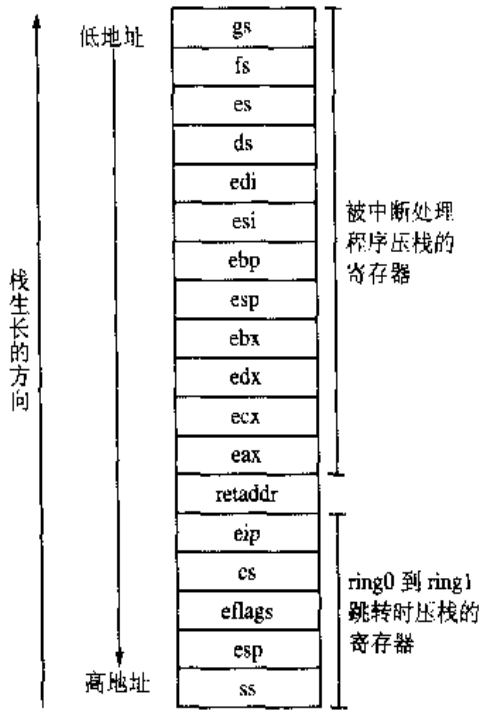

图 6-8 进程表的开始位置结构图示

我们再来看第 3 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 的值和结构体 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 4 行、第 5 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 3-44），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 （节自\chapter6\a\kernel\kernel.asm）

```
ALIGN   16
hwint00:        ; Interrupt routine for irq 0 (the clock).
    iretd
```

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。

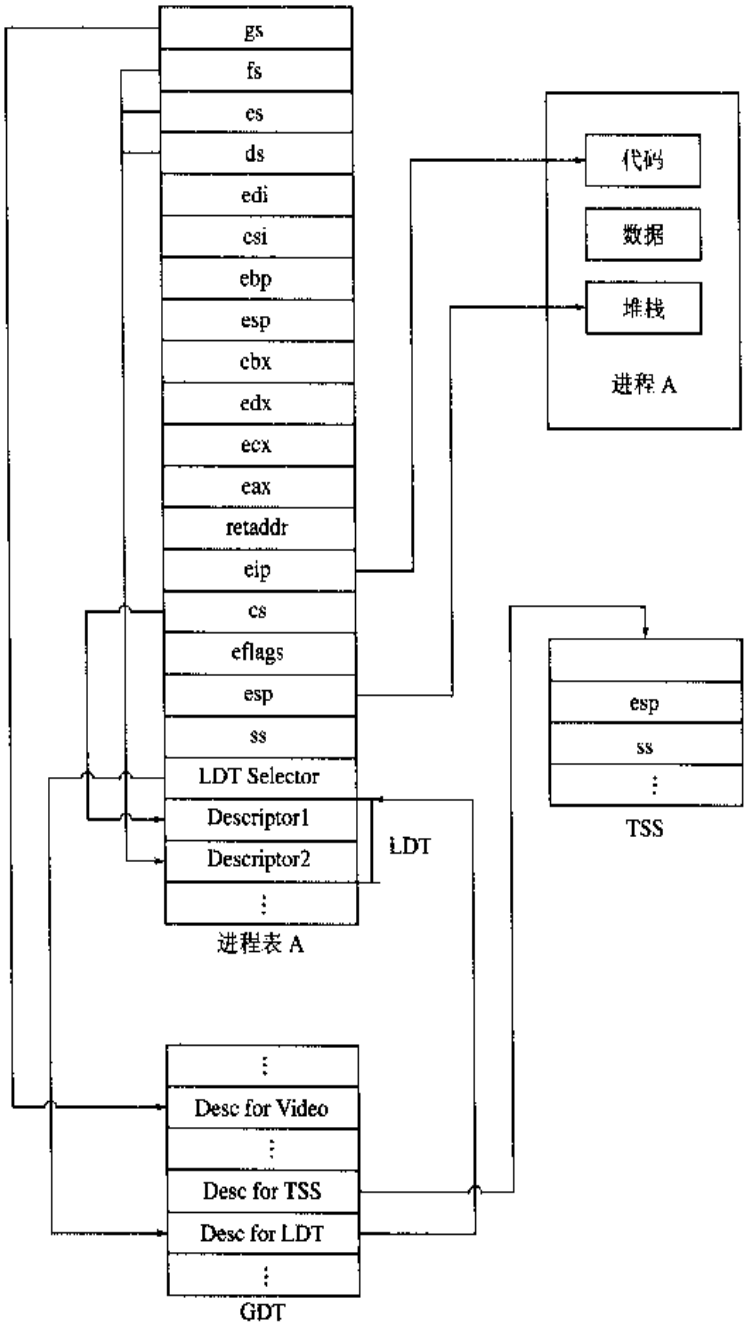

图 6-9 进程表及相关数据结构对应关系示意图

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：

- 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。
- 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。

- 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。

1. 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 （节自\chapter6\a\kernel\main.c）

```
void TestA()
{
    int i = 0;
    while(1){
        disp_str("A");
        disp_int(i++);
        disp_str(".");
        delay(1);
    }
}
```

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 tinix_main()这个函数中，见代码 6-5，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 tinix_main（节自\chapter6\a\kernel\main.c）

```
PUBLIC int tinix_main()
{
```

```
    disp_str("-----\"tinix_main\" begins-----\n");
    while(1){}
}
```

在 kernel.asm 的最后，我们跳转到 tinix_main()中：

代码 6-5 跳转到 tinix_main（节自\chapter6\a\kernel\kernel.asm）

```
extern  tinix_main
......
csinit:
    ;ud2
    ;sti
    jmp tinix_main
hlt
```

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 函数 delay（节自\chapter6\a\kernel\klib.c）

```
PUBLIC void delay(int time)
{
    int i, j, k;
    for(k=0;k<time;k++){
        for(i=0;i<10000;i++){
            for(j=0;j<10000;j++){}
        }
    }
}
```

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。

2. 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 结构体 STACK_FRAME（节自\chapter6\a\include\proc.h）

```
typedef struct s_stackframe {
    t_32        gs;
    t_32        fs;
    t_32        es;
    t_32        ds;
    t_32        edi;
```

```
    t_32        esi;
    t_32        ebp;
    t_32        kernel_esp;
    t_32        ebx;
    t_32        edx;
    t_32        ecx;
    t_32        eax;
    t_32        retaddr;
    t_32        eip;
    t_32        cs;
    t_32        eflags;
    t_32        esp;
    t_32        ss;
}STACK_FRAME;
```

如果一开始自己编写这部分代码的话，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 结构体 PROCESS（节自\chapter6\a\include\proc.h）

```
typedef struct s_proc {
    STACK_FRAME         regs;           /* 寄存器 */
    t_16                ldt_sel;        /* LDT 选择子 */
    DESCRIPTOR          ldts[LDT_SIZE]; /* LDTs */
    t_32                pid;            /* 进程号 */
    char                p_name[16];     /* 名字 */
}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 tinix_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（节自\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));
// change the DPL
p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5;
memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS >> 3],
        sizeof (DESCRIPTOR));
// change the DPL
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    = (t_32)TestA;
p_proc->regs.esp    = (t_32) task_stack + STACK_SIZE_TOTAL;
p_proc->regs.eflags = 0x1200;
```

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 3-45 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 6-5 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（节自\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
```

代码 6-10 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 填充 GDT 中进程的 LDT 的描述符（节自\chapter6\a\kernel\protect.c）

```
init_descriptor(&gdt[INDEX_LDT_FIRST],
        vir2phys(seg2phys(SELECTOR_KERNEL_DS), proc_table[0].ldts),
        LDT_SIZE * sizeof(DESCRIPTOR),
        DA_LDT);
```

这一小段代码放在 init_prot()中。init_descriptor 和 init_idt_desc 有些类似：

代码 6-12 函数 init_descriptor（节自\chapter6\a\kernel\protect.c）

```
PRIVATE void init_descriptor(DESCRIPTOR * p_desc, t_32 base,
                             t_32 limit, t_16 attribute)
{
    p_desc->limit_low   = limit&0x0FFFF;        // 段界限 1(2 字节)
    p_desc->base_low    = base&0x0FFFF;         // 段基址 1(2 字节)
    p_desc->base_mid    = (base >> 16) & 0x0FF; // 段基址 2(1 字节)
    p_desc->attr1       = attribute & 0xFF;     // 属性 1
    p_desc->limit_high_attr2    = ((limit >> 16) & 0x0F) |
                                (attribute >> 8) & 0xF0;//界限 2+属性 2
    p_desc->base_high   = (base>>24)&0x0FF; //段基址 3(1 字节)
}
```

seg2phys 是这样定义的：

代码 6-13 由段名求绝对地址（节自\chapter6\a\kernel\protect.c）

```
PUBLIC t_32 seg2phys(t_16 seg)
{
    DESCRIPTOR* p_dest = &gdt[seg >> 3];
    return (p_dest->base_high << 24)    |
           (p_dest->base_mid << 16)     |
           (p_dest->base_low);
}
```

vir2phys 是一个宏，定义在 protect.h 中：

```
#definevir2phys(seg_base,vir)(t_32)(((t_32)seg_base)+(t_32)(vir))
```

3. 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 6-16）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 6-14 TSS 相关（节自\chapter6\a\kernel\protect.c）

```
memset(&tss, 0, sizeof(tss));
tss.ss0 = SELECTOR_KERNEL_DS;
init_descriptor(&gdt[INDEX_TSS],
                vir2phys(seg2phys(SELECTOR_KERNEL_DS), &tss),
                sizeof(tss),/*0x69,*/
                DA_386TSS);
```

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，在 kernel.asm 中添加几行代码就够了：

代码 6-15 加载 tr（节自\chapter6\a\kernel\kernel.asm）

```
xor eax, eax
mov ax, SELECTOR_TSS
ltr ax
jmp tinix_main
```

代码 6-16 结构体 TSS（节自\chapter6\a\include\protect.h）

```
typedef struct s_tss {
    t_32    backlink;
    t_32    esp0;   /* stack pointer to use during interrupt */
    t_32    ss0;    /*  *   segment  *  *     *       *     */
    t_32    esp1;
    t_32    bss1;
    t_32    eesp2;
    t_32    sss2;
    t_32    ecr3;
    t_32    eip;
    t_32    flags;
    t_32    eax;
    t_32    ecx;
    t_32    edx;
    t_32    ebx;
    t_32    esp;
    t_32    ebp;
    t_32    esi;
    t_32    edi;
    t_32    es;
    t_32    cs;
    t_32    ss;
```

```
    t_32    ds;
    t_32    fs;
    t_32    gs;
    t_32    ldt;
    t_16    trap;
    t_16    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 6-17 这样就足够了。

代码 6-17 restart（节自\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 以及代码 6-15 中用到的 SELECTOR_TSS 都定义在新建立的文件 sconst.inc 中。一定要注意，这里的选择子值必须与 protect.h 中的值保持一致。

代码 6-18 \chapter6\a\include\sconst.inc

```
P_STACKBASE equ 0
GSREG           equ P_STACKBASE
FSREG           equ GSREG           + 4
ESREG           equ FSREG           + 4
```

```
DSREG           equ ESREG           + 4
EDIREG          equ DSREG           + 4
ESIREG          equ EDIREG          + 4
EBPREG          equ ESIREG          + 4
KERNELESPREG    equ EBPREG          + 4
EBXREG          equ KERNELESPREG    + 4
EDXREG          equ EBXREG          + 4
ECXREG          equ EDXREG          + 4
EAXREG          equ ECXREG          + 4
RETADR          equ EAXREG          + 4
EIPREG          equ RETADR          + 4
CSREG           equ EIPREG          + 4
EFLAGSREG       equ CSREG           + 4
ESPREG          equ EFLAGSREG       + 4
SSREG           equ ESPREG          + 4
P_STACKTOP      equ SSREG           + 4
P_LDT_SEL       equ P_STACKTOP
P_LDT           equ P_LDT_SEL       + 4

; 以下选择子值必须与 protect.h 中保持一致
SELECTOR_FLAT_C         equ     0x08
SELECTOR_TSS            equ     0x20
SELECTOR_KERNEL_CS      equ     SELECTOR_FLAT_C
```

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 tinix_main()，添加以下代码：

代码 6-19 扳机（节自\chapter6\a\kernel\main.c）

```
    p_proc_ready    = proc_table;
    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 (refer to code: TinixForBook_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（请参考\chapter6\a）

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。

在这里我要提及一点 Virtual PC 和 Bochs 的不同，如果把现在的.IMG 文件拿到 Bochs 上运行会发现，进程打印出一个字母 A 和一个数字之后就停住不动了，如图 6-11 所示。

图 6-11 在 Bochs 下运行出现问题（请参考\chapter6\a）

这是为什么呢？还记得在 2.7.1 节中我们提到，Bochs 的速度相对来说要慢得多，而在我们的进程体中用到的 delay()是用一个循环来耗费时间，虽然在 Virtual PC 中这个函数很快就执行完了，但在 Bochs 中却不一样，delay()需要更长的时间，所以“A0x0.”显示完之后，由于 Bochs 一直在执行 delay()中的循环而没有输出，所以看起来像是停止了。

所以，如果想在 Bochs 上运行，需要把 delay()稍微修改一下，减少循环的次数，然后就看到效果了，如图 6-12 所示。

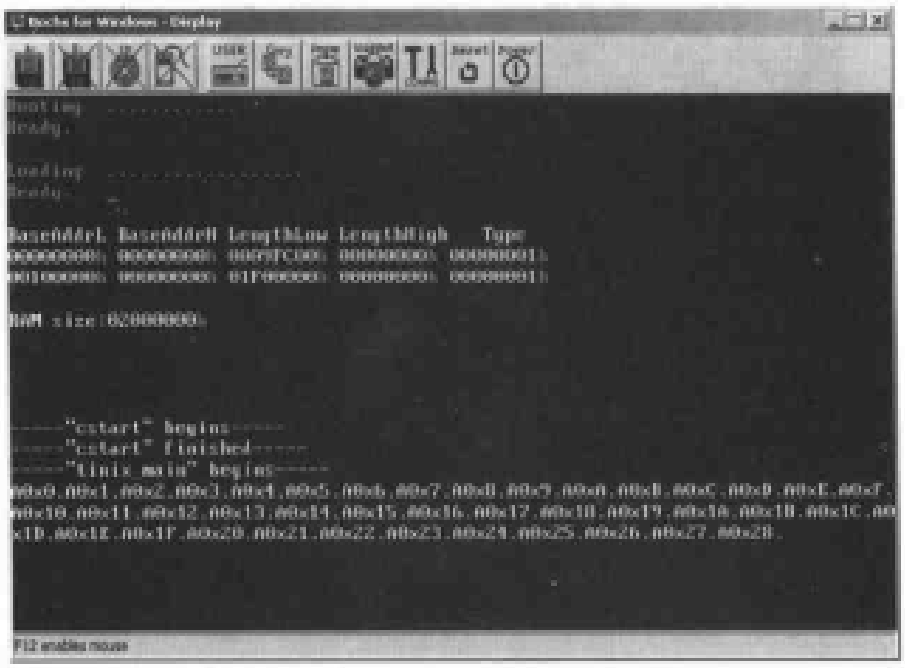

图 6-12　在 Bochs 下正常运行（请参考\chapter6\a）

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 6-13 所示。

图 6-13 有助于让我们更清晰地了解程序的执行过程。其中，进程体 TestA()是静态的，程序运行时所遵循的就是箭头所指的顺序。从图中还可以清楚地看到，6.3.2.2 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。

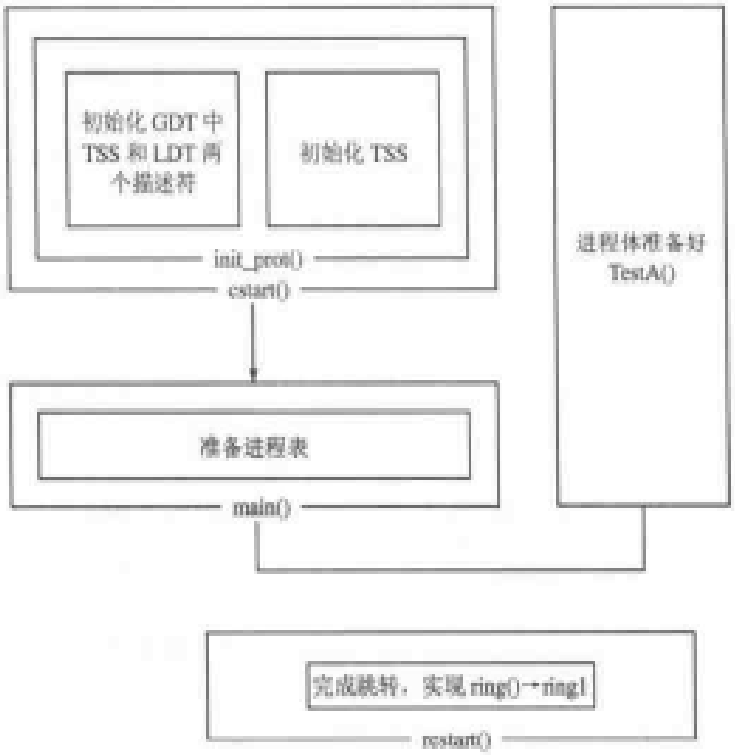

图 6-13 第一个进程的启动过程示意图

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 6-3 和图 6-4 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。

```
out_byte(INT_M_CTLMASK, 0xFE);
```

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 6-20 设置 EOI（节自\chapter6\b\kernel\kernel.asm）

```
ALIGN   16
hwint00:        ; Interrupt routine for irq 0 (the clock).
    mov  al, EOI          ; ┐reenable master 8259
    out  INT_M_CTL, al    ; ┘
```

```
    iretd
```

EOI 和 INT_M_CTL 定义在 sconst.inc 中：

代码 6-21　EOI 和 INT_M_CTL（节自\chapter6\b\include\sconst.inc）

```
INT_M_CTL       equ     0x20
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 6-22　时钟中断处理程序（节自\chapter6\b\kernel\kernel.asm）

```
ALIGN   16
hwint00:        ; Interrupt routine for irq 0 (the clock).
    inc byte [gs:0]     ; 改变屏幕第 0 行，第 0 列的字符
    mov al, EOI         ; ┐reenable master 8259
    out INT_M_CTL, al   ; ┘
    iretd
```

很明显，代码 6-22 参照了代码 3-46 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 6-14 所示。

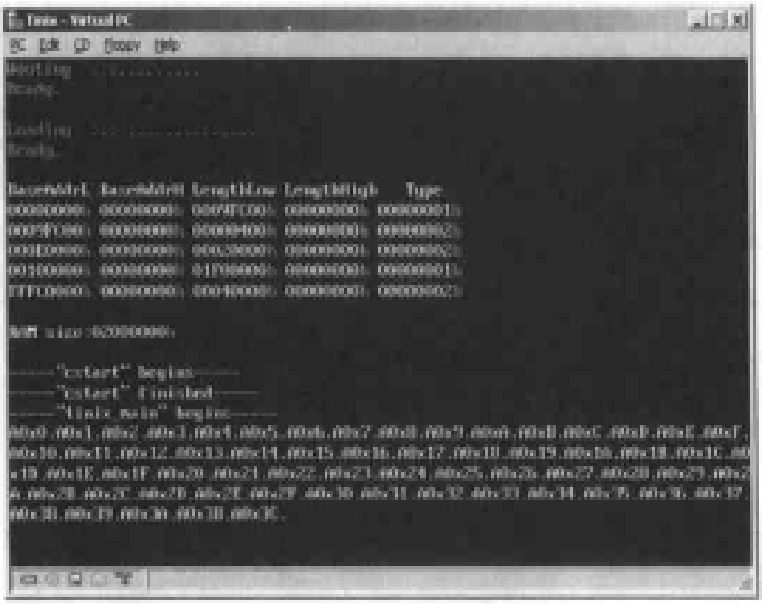

图 6-14　左上角的字符变化说明中断模块运行正常

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。图 6-14 正好显示到了字母 W。

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 6-23 的样子。

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。

代码 6-23 修改时钟中断处理程序（节自\chapter6\b\kernel\kernel.asm）

```
ALIGN   16
hwint00:    ; Interrupt routine for irq 0 (the clock).
    pushad      ; ┐
    push    ds  ; │
    push    es  ; ├ 保存原寄存器值
    push    fs  ; │
    push    gs  ; ┘

    inc     byte [gs:0]     ; 改变屏幕第 0 行，第 0 列的字符
    mov     al, EOI         ; ┐ reenable master 8259
    out     INT_M_CTL, al   ; ┘

    pop     gs  ; ┐
    pop     fs  ; │
    pop     es  ; ├ 恢复原寄存器值
    pop     ds  ; │
    pop     ad  ; ┘

    iretd
```

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，在 protect.c 的 init_prot()中可以找到（参见代码 6-14）：

```
tss.ss0 = SELECTOR_KERNEL_DS;
```

那么，tss.esp0 应该在什么时候被赋值呢？其实我们在 6.3.2 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。

我们当时分析过，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 6-24 给出了实现方法。

代码 6-24 修改时钟中断处理程序（节自\chapter6\b\kernel\kernel.asm）

```
ALIGN   16
hwint00:    ; Interrupt routine for irq 0 (the clock).
    sub     esp,4
    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

inc     byte [gs:0]     ; 改变屏幕第 0 行, 第 0 列的字符

mov     al, EOI         ; ┐reenable master 8259
out     INT_M_CTL, al   ; ┘

lea     eax, [esp + P_STACKTOP]
mov     dword [tss + TSS3_S_SP0], eax

pop     gs  ; ┐
pop     fs  ; │
pop     es  ; ├ 恢复原寄存器值
pop     ds  ; │
popad       ; ┘
add     esp,4

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 6-25 修改时钟中断处理程序（节自\chapter6\b\kernel\kernel.asm）

```
ALIGN   16
hwint00:    ; Interrupt routine for irq 0 (the clock).
    sub     esp, 4

    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     esp, StackTop   ; 切到内核栈

    inc     byte [gs:0]     ; 改变屏幕第 0 行，第 0 列的字符

    mov     al, EOI         ; ┐ reenable master 8259
    out     INT_M_CTL, al   ; ┘

    mov     esp, [p_proc_ready] ; 离开内核栈

    lea     eax, [esp + P_STACKTOP]
    mov     dword [tss + TSS3_S_SP0], eax

    ......

    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 6-26 修改时钟中断处理程序（节自\chapter6\b\kernel\kernel.asm）

```
ALIGN   16
hwint00:    ; Interrupt routine for irq 0 (the clock).
    ......

    mov     esp, StackTop       ; 切到内核栈

    inc     byte [gs:0]         ; 改变屏幕第 0 行，第 0 列的字符

    mov     al, EOI             ; ┐reenable master 8259
    out     INT_M_CTL, al       ; ┘

    push    clock_int_msg
    call    disp_str
    add     esp, 4

    mov     esp, [p_proc_ready] ; 离开内核栈;

    ......

    iretd
```

其中，clock_int_msg 的定义如代码 6-27 所示，仅仅是个字符“^”而已：

代码 6-27 clock_int_msg（节自\chapter6\b\kernel\kernel.asm）

```
......
[SECTION .data]
clock_int_msg       db  "^", 0

[SECTION .bss]
......
```

此时，如果运行的话，就应该看到图 6-15 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。

图 6-15 时钟中断模块运行正常（对应\chapter6\b）

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 修改时钟中断处理程序（节自\chapter6\c\kernel\kernel.asm）

```
hwint00:    ; Interrupt routine for irq 0 (the clock).
    ......
    mov     esp, StackTop   ; 切到内核栈
    inc     byte [gs:0]     ; 改变屏幕第 0 行，第 0 列的字符
    mov     al, EOI         ; ┐reenable master 8259
    out     INT_M_CTL, al   ; ┘

    sti

    push    clock_int_msg
    call    disp_str
    add     esp, 4
```

```
push    1
call    delay
add     esp, 4

cli

mov     esp, [p_proc_ready]; 离开内核栈;
......
```

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 6-16 所示。

图 6-16 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。

首先声明全局变量 k_reenter，并在 tinix_main()中将它赋值为-1：

```
k_reenter = -1;
```

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 6-29　修改时钟中断处理程序（节自\chapter6\c\kernel\kernel.asm）

```
......
inc     byte [gs:0]       ; 改变屏幕第 0 行，第 0 列的字符

mov     al, EOI           ; ┐reenable master 8259
out     INT_M_CTL, al     ; ┘

inc     dword [k_reenter]
cmp     dword [k_reenter], 0
jne     .re_enter

mov     esp, StackTop     ; 切到内核栈
sti
push    clock_int_msg
call    disp_str
add     esp, 4
push    1
call    delay
add     esp, 4
cli
mov     esp, [p_proc_ready] ; 离开内核栈;
lea     eax, [esp + P_STACKTOP]
mov     dword [tss + TSS3_S_SP0], eax

.re_enter:  ; 如果(k_reenter != 0)，会跳转到这里
    dec     dword [k_reenter]   ; k_reenter--;
    pop     gs
    pop     fs
    ......
```

我们再 make 一下，运行，结果如图 6-17 所示。

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。

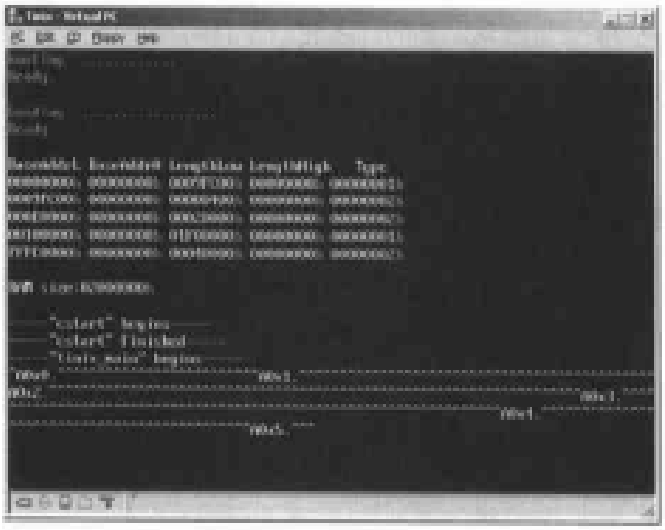

图 6-17 解决中断重入问题

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 6-30 修改时钟中断处理程序（节自\chapter6\c\kernel\kernel.asm）

```
......
;inc    byte [gs:0] ; 改变屏幕第 0 行，第 0 列的字符

;push   1
;call   delay
;add    esp, 4
......
```

再次运行，在 Virtual PC 中的运行结果如图 6-18 所示。

图 6-18 考虑中断重入之后的 Tinix（对应\chapter6\c）

你可能觉得图 6-18 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。

6.3.4　进程体设计技巧

你可能之前产生过这样的疑问，为什么我们在设计进程体的时候没有把它再简化，直接打印一连串的字符 A 而不打印数字？

我们不妨来做一个试验，假设进程体是这样的：

代码 6-31　不好的进程体

```
void TestA()
{
    while(1){
        disp_str("A");
        delay(1);
    }
}
```

在其他代码不变的情况下，我们看到的应该是 A 和 ^ 交替出现，如图 6-19 所示。

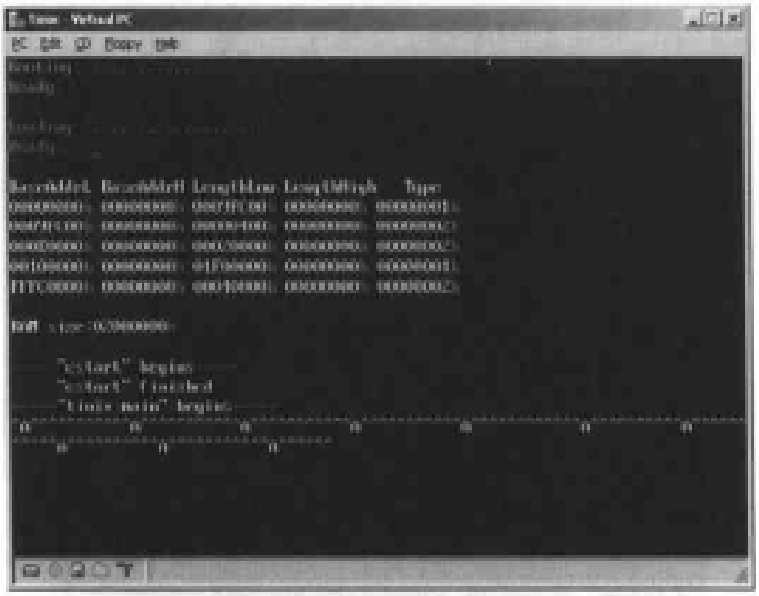

图 6-19　正确的进程实现在使用不好的进程体的情况下的输出

可是，是否只要看到 A 和 ^ 交替出现就表示我们的程序是正确的呢？下面我们来做一个试验，故意犯一个错误，把两个地方的

```
lea eax, [esp + P_STACKTOP]
```

修改成：

```
lea eax, [esp + 0x1000]
```

这是笔者故意设计的错误，实际情况中你可能不会犯这个错误（因为它太明显）。在这里，我们仅仅拿它做个例子。假设犯了这样的错误，那么结果会是怎样的呢？运行一下，结果如图 6-20 所示。

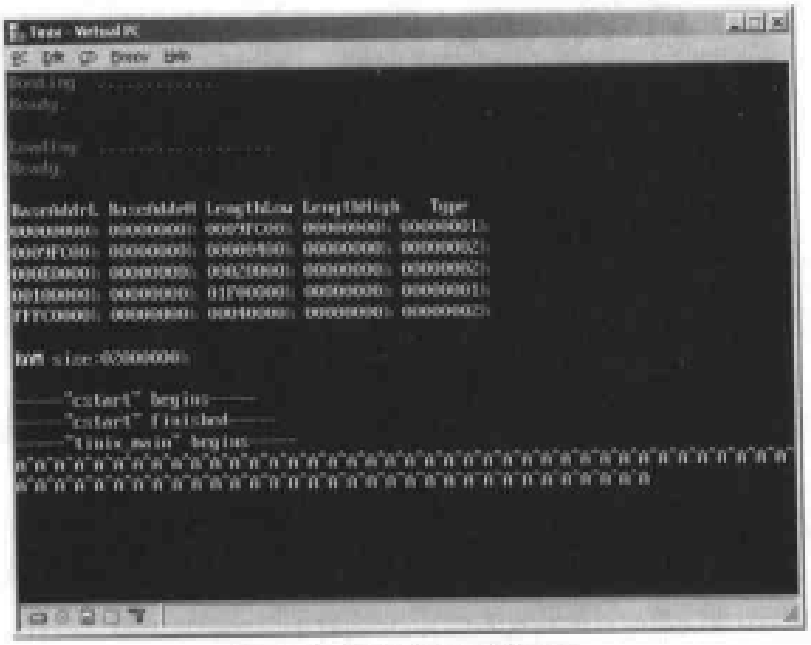

图 6-20 无法确定对错的输出

同样是 A 和 ^ 交替出现，而且均匀、美丽，看上去排列整齐，这个错误结果比正确结果看上去还要好。

想像一下，如果一开始你就得到了这个结果，你会认为是错的吗？不会，根本不会，你会非常兴奋地庆祝一番，庆祝第一个进程的运行成功。当然，早晚有一天，这个错误会导致某一个明显的错误结果，但是，到时候你可能不会想到，原来错误居然在这个时候就已经出现了。随着代码的增多，找到这个错误可能会越来越困难。

我们来看一下为什么仍然会有看上去正确的结果。显然，这个错误将导致 tss.esp0 里存放的是错误的值。所以当中断发生时，寄存器 esp 不再指向正确的进程表中的位置，而是指向了 0x1000 这个地址。于是，我们对所有寄存器的保存都告无效，因为没有保存到正确的位置，到下一次进程被恢复，进程又从开始的位置被执行，就好像第一次执行它时一样。

现在我们的进程体稍稍复杂了一点，变成字符后面跟一个不断增加的数字，这样，错误就很容易被发现。假设中断处理中我们的错误仍然存在，把进程体改成原来的样子，运行，结果如图 6-21 所示。

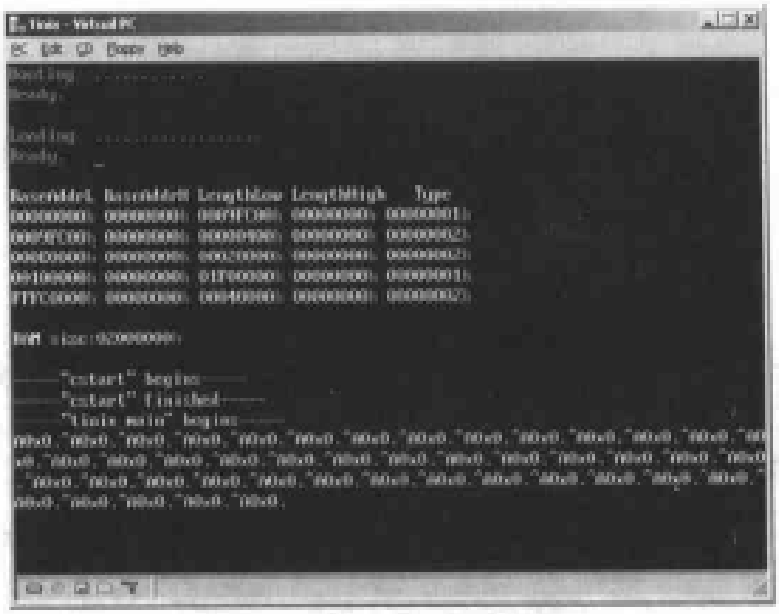

图 6-21　很明显是错误的输出

果然，每一次打印的数字都是 0，这表明进程都是从开始处开始执行。我们一下就意识到了错误。

这就是进程这样设计的原因。

说实话，刚才我们“假设”的那个错误其实不是个假设，而是笔者真的犯过一个类似的错误（跟这个错误类似，但略有不同），直到若干日之后才发现。而且，笔者费了很大的力气找到根源时才发现错误早就出现了。如果一开始进程就设计成打印字母和数字，错误就可以早些被发现了。

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 进程 B（节自\chapter6\d\kernel\main.c）

```
void TestB()
{
    int i = 0x1000;
    while(1){
        disp_str("B");
        disp_int(i++);
        disp_str(".");
        delay(1);
    }
}
```

在这里，除了打印的字母换成了 B 之外，i 的初始值也被设成了 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。回到 6.3 节，我们当时提到，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 tinix_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，将其中涉及到进程 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（参见 table.c）。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 s_task（节自\chapter6\d\include\proc.h）

```
typedef struct s_task {
    t_pf_task   initial_eip;
    int         stacksize;
    char        name[32];
}TASK;
```

其中，t_pf_task 是这样定义的（在 type.h 中）：

```
typedef void (*t_pf_task) ();
```

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 task_table（节自\chapter6\d\kernel\global.c）

```
PUBLIC TASK task_table[NR_TASKS]={
                                {TestA,STACK_SIZE_TESTA,"TestA"},
                                {TestB,STACK_SIZE_TESTB,"TestB"}
                                };
```

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：

```
#define NR_TASKS        2
```

在 task_table 的定义中，STACK_SIZE_TESTB 还没有定义。好的，到 proc.h 中找到宏 STACK_SIZE_TESTA，添加下面的定义：

代码 6-35 进程栈大小定义（节自\chapter6\c\include\proc.h）

```
#define STACK_SIZE_TESTA    0x8000
#define STACK_SIZE_TESTB    0x8000

#define STACK_SIZE_TOTAL    (STACK_SIZE_TESTA + \
                            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 （节自\chapter6\d\kernel\main.c）

```
TASK*       p_task          = task_table;
PROCESS*    p_proc          = proc_table;
char*       p_task_stack    = task_stack + STACK_SIZE_TOTAL;
t_16        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));
    // change the DPL:
    p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5;
    memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS>>3],
           sizeof(DESCRIPTOR));
    // change the DPL:
    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    = (t_32)p_task->initial_eip;
    p_proc->regs.esp    = (t_32)p_task_stack;
    p_proc->regs.eflags = 0x1202;   // IF=1, IOPL=1, bit 2 is always


    p_task_stack -= p_task->stacksize;
    p_proc++;
    p_task++;
    selector_ldt += 1 << 3;
}
k_reenter = -1;
```

```
    p_proc_ready    = proc_table;
```

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“惟一一个”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 （节自\chapter6\d\kernel\protect.c）

```
    int i;
    PROCESS* p_proc = proc_table;
    t_16 selector_ldt = INDEX_LDT_FIRST << 3;
    for(i=0;i<NR_TASKS;i++){
        init_descriptor(&gdt[selector_ldt>>3],
                vir2phys(seg2phys(SELECTOR_KERNEL_DS),
                        proc_ta ble[i].ldts),
                LDT_SIZE * sizeof(DESCRIPTOR),
                DA_LDT);
        p_proc++;
        selector_ldt += 1 << 3;
    }
```

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：

代码6-38 （节自\chapter6\d\kernel\kernel.asm）

```
……
mov     esp, [p_proc_ready] ; 离开内核栈;
lldt    [esp + P_LDT_SEL]
lea     eax, [esp + P_STACKTOP]
mov     dword [tss + TSS3_S_SP0], eax
……
```

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，因为涉及到进程调度等内容。一方面，这涉及到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6-39 （节自\chapter6\d\kernel\clock.c）

```
PUBLIC void clock_handler(int irq)
```

```
{
    disp_str("#");
}
```

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：

代码 6-40　（节自\chapter6\d\kernel\kernel.asm）

```
    ......
    mov     esp, StackTop       ; 切到内核栈
    sti
    push    0
    call    clock_handler
    add     esp, 4
    cli
    mov     esp, [p_proc_ready] ; 离开内核栈;
    ......
```

由于增加了一个文件，所以不要忘记修改 Makefile。make，运行，看到图 6-22 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 6-22　做好切换的准备但还未添加进程切换代码时的情形

添加进程切换代码：

代码 6-41　（节自\chapter6\d\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 6-41 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 6-23 所示。

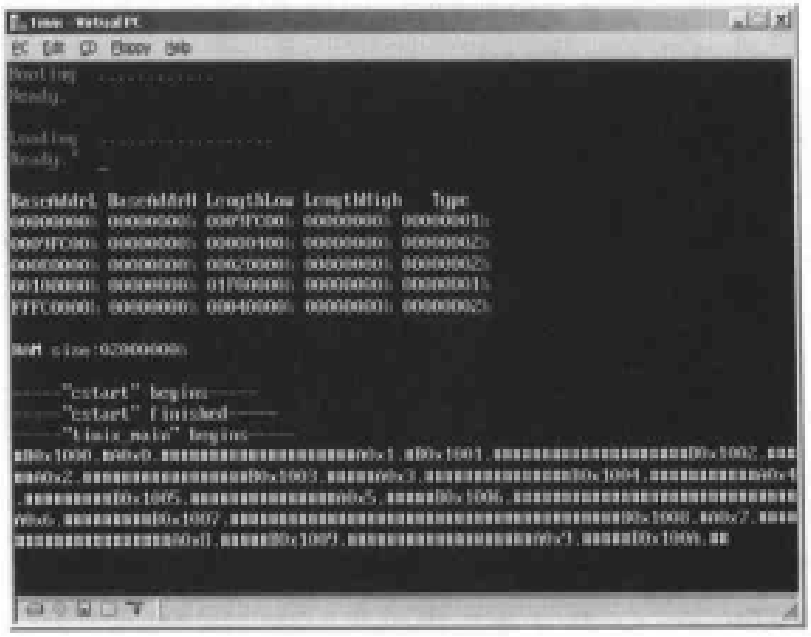

图 6-23 实现多进程

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。

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。

添加一个进程体：

代码 6-42 （节自\chapter6\d\kernel\main.c）

```
void TestC()
{
    int i = 0x2000;
    while(1){
        disp_str("C");
        disp_int(i++);
        disp_str(".");
        delay(1);
    }
}
```

在 task_table 中添加一项进程：

代码 6-43 （节自\chapter6\d\kernel\global.c）

```
PUBLIC TASK task_table [NR_TASKS] = {
                                    {TestA, STACK_SIZE_TESTA, "TestA"},
                                    {TestB, STACK_SIZE_TESTB, "TestB"},
                                    {TestB, STACK_SIZE_TESTC, "TestC"}
                                    };
```

然后在 proc.h 中修改 NR_TASKS 的值：

```
#define NR_TASKS        3
```

分配任务堆栈：

代码 6-44 （节自\chapter6\d\include\proc.h）

```
#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 6-24 所示。

图 6-24 三个进程同时运行（请参考\chapter6\d）

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 6-45 Minix 的中断处理（节自 src/kernel/mpx386.s）

```
#define hwint_master(irq) \
```

```
    call    save        /*save interrupted process state */    ;\
    inb INT_CTLMASK                                             ;\
    orb al, [1<<irq]                                            ;\
    outb INT_CTLMASK    /* disable the irq              */     ;\
    movb al, ENABLE                                             ;\
    outb INT_CTL        /* reenable master 8259         */     ;\
    sti                 /* enable interrupts            */     ;\
    push irq            /* irq                          */     ;\
    call (_irq_table + 4*irq) /* eax = (*irq_table[irq])(irq) */ ;\
    pop ecx                                                     ;\
    cli                 /* disable interrupts           */     ;\
    test    eax, eax    /* need to reenable irq?        */     ;\
    jz  0f                                                      ;\
    inb INT_CTLMASK                                             ;\
    andb    al, ~[1<<irq]                                       ;\
    outb    INT_CTLMASK /* enable the irq               */     ;\
0:  ret                 /* restart (another) process    */

! Each of these entry points is an expansion of the hwint_master macro
    .align  16
_hwint00:       ! Interrupt routine for irq 0 (the clock).
    hwint_master(0)

    .align  16
_hwint01:       ! Interrupt routine for irq 1 (keyboard)
    hwint_master(1)

    .align  16
_hwint02:       ! Interrupt routine for irq 2 (cascade!)
    hwint_master(2)

...........
```

代码后面部分的_hwint00、_hwint01、_hwint02 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 6-46 Minix 的中断处理（节自 src/kernel/mpx386.s）

```
save:
    cld                     ! set direction flag to a known value
    pushad                  ! save "general" registers
    o16 push    ds          ! save ds
    o16 push    es          ! save es
    o16 push    fs          ! save fs
    o16 push    gs          ! save gs
    mov dx, ss              ! ss is kernel data segment
    mov ds, dx              ! load rest of kernel segments
    mov es, dx              ! kernel does not use fs, gs
    mov eax, esp            ! prepare to return
    incb    (_k_reenter)    ! from -1 if not reentering
    jnz set_restart1        ! stack is already kernel stack
    mov esp, k_stktop
    push    _restart        ! build return address for int handler
    xor ebp, ebp            ! for stacktrace
    jmp RETADR-P_STACKBASE(eax)

    .align  4
set_restart1:
    push    restart1
    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？向前找一下语句：

```
mov eax, esp
```

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 6-47 Minix 的中断处理（节自 src/kernel/sconst.h）

```
P_STACKBASE =   0
GSREG       =   P_STACKBASE
FSREG       =   GSREG + 2   ! 386 introduces FS and GS segments
ESREG       =   FSREG + 2
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
```

我想你已经明白了，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 这条指令。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 6-48 Minix 的中断处理（src/kernel/mpx386.s）

```
_restart:

! Flush any held-up interrupts.
! This reenables interrupts, so the current interrupt handler may
```

```
! reenter. This does not matter, because the current handler is about
! to exit and no other handlers can reenter since flushing is only done
! when k_reenter == 0.

        cmp     (_held_head), 0         ! do fast test to usually
                                        ! avoid function call
        jz      over_call_unhold
        call    _unhold                 ! this is rare so overhead
                                        ! acceptable
over_call_unhold:
        mov     esp, (_proc_ptr)        ! will assume P_STACKBASE == 0
        lldt    P_LDT_SEL(esp)          ! enable segment descriptors
                                        ! for task
        lea     eax, P_STACKTOP(esp)    ! arrange for next interrupt
        mov     (_tss+TSS3_S_SP0), eax  ! to save state in process table

restart1:
        decb    (_k_reenter)
    o16 pop     gs
    o16 pop     fs
    o16 pop     es
    o16 pop     ds
        popad
        add     esp, 4                  ! skip return adr

        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 Tinix 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 6-49 修改时钟中断处理程序（节自\chapter6\e\kernel\kernel.asm）

```
hwint00:        ; Interrupt routine for irq 0 (the clock).
    ......
    inc     dword [k_reenter]
    cmp     dword [k_reenter], 0
    jne     .1  ; 重入时跳到.1，通常情况下顺序执行

    mov     esp, StackTop       ; 切到内核栈

    push    .restart_v2
    jmp     .2
.1: ; 中断重入
    push    .restart_reenter_v2
.2: ; 没有中断重入
    sti
    push    0
    call    clock_handler
    add     esp, 4
    cli

    ret     ; 重入时跳到.restart_reenter_v2，通常情况下到.restart_v2

.restart_v2:
    mov     esp, [p_proc_ready]     ; 离开内核栈
    lldt    [esp + P_LDT_SEL]
    lea     eax, [esp + P_STACKTOP]
    mov     dword [tss + TSS3_S_SP0], eax
.restart_reenter_v2:            ; 如果(k_reenter != 0)，会跳转到这里
    dec     dword [k_reenter]       ; k_reenter--;
    pop     gs  ; ┐
    pop     fs  ; │
    pop     es  ; ├ 恢复原寄存器值
    pop     ds  ; │
    popad       ; ┘
    add     esp, 4

    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 6-50 修改 clock_handler（节自\chapter6\e\kernel\clock.c）

```
PUBLIC void clock_handler(int irq)
{
    disp_str("#");

    if (k_reenter != 0) {
        disp_str("!");
        return;
    }

    p_proc_ready++;

    if (p_proc_ready >= proc_table + NR_TASKS) {
        p_proc_ready = proc_table;
    }
}
```

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，结果如图 6-24 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。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（main.c 中）：

```
k_reenter = 0;  /*原来为-1*/
```

代码 6-51 修改时钟中断处理程序（节自\chapter6\e\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
restart_reenter:
    dec     dword [k_reenter]
    pop     gs
```

```
        pop     fs
        pop     es
        pop     ds
        popad
        add     esp, 4
        iretd
```

现在如果对比代码 6-49 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 6-49 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（节自\chapter6\e\kernel\kernel.asm）

```
hwint00:        ; Interrupt routine for irq 0 (the clock).
    ............
    i
    mov     esp, StackTop       ; 切到内核栈

    push    restart
    jmp .2

.1: ; 中断重入
    push    restart_reenter

.2: ; 没有中断重入
    sti
    ............
```

现在，我们删除掉了 hwint00 中的最后一段代码，并修改了涉及到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 修改时钟中断处理程序（节自\chapter6\f\kernel\kernel.asm）

```
hwint00:                ; Interrupt routine for irq 0 (the clock).
```

```
    call    save
    mov     al, EOI          ; ┐reenable master 8259
    out     INT_M_CTL, al    ; ┘
    sti
    push    0
    call    clock_handler
    add     esp, 4
    cli
    ret ; 重入时跳到 restart_reenter，通常情况下到 restart

............

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 修改时钟中断处理程序（节自\chapter6\f\kernel\kernel.asm）

```
hwint00:        ; Interrupt routine for irq 0 (the clock).
    call    save

    in      al, INT_M_CTLMASK   ; ┐
    or      al, 1               ; ├ 不允许再发生时钟中断
    out     INT_M_CTLMASK, al   ; ┘

    mov     al, EOI             ; ┐ 置 EOI 位
    out     INT_M_CTL, al       ; ┘

    sti

    push    0
    call    clock_handler
    add     esp, 4

    cli

    in      al, INT_M_CTLMASK   ; ┐
    and     al, 0xFE            ; ├ 又允许时钟中断发生
    out     INT_M_CTLMASK, al   ; ┘

    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，并且修改中断例程如下：

代码 6-55 修改时钟中断处理程序（节自\chapter6\f\kernel\kernel.asm）

```
hwint00:        ; Interrupt routine for irq 0 (the clock).
```

```
    hwint_master    0
```

新的宏如下：

代码 6-56 宏 hwint_master（节自\chapter6\f\kernel\kernel.asm）

```
%macro  hwint_master    1
    call    save
    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  t_pf_irq_handler    irq_table[NR_IRQ];
```

其中，t_pf_irq_handler 在 type.h 中这样定义：

```
typedef void    (*t_pf_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 初始化 irq_table（节自\chapter6\f\kernel\8259.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 put_irq_handler（节自\chapter6\f\kernel\i8259.c）

```
PUBLIC void put_irq_handler(int irq, t_pf_irq_handler handler)
{
    disable_irq(irq);
    irq_table[iIRQ] = 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 和我们马上要用到的另一个函数 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 和代码 6-60）。

代码 6-59 disable_irq（节自\chapter6\f\lib\klib.asm）

```
disable_irq:
    mov     ecx, [esp + 4]      ; irq
    pushf
    cli
    mov     ah, 1
    rol     ah, cl              ; ah = (1 << (irq % 8))
    cmp     cl, 8
    jae     disable_8           ; disable irq >= 8 at the slave 8259
disable_0:
    in      al, INT_M_CTLMASK
    test    al, ah
    jnz     dis_already         ; already disabled?
    or      al, ah
    out     INT_M_CTLMASK, al   ; set bit at master 8259
    popf
    mov     eax, 1              ; disabled by this function
```

```
    ret
disable_8:
    in      al, INT_S_CTLMASK
    test    al, ah
    jnz     dis_already         ; already disabled?
    or      al, ah
    out     INT_S_CTLMASK, al   ; set bit at slave 8259
    popf
    mov     eax, 1              ; disabled by this function
    ret
dis_already:
    popf
    xor     eax, eax            ; already disabled
    ret
```

与 disable_irq 等价的一段 C 代码看上去应该是下面的样子：

```
if(irq < 8){
    out_byte(INT_M_CTLMASK, in_byte(INT_M_CTLMASK) | (1 << irq));
}
else{
    out_byte(INT_S_CTLMASK, in_byte(INT_S_CTLMASK) | (1 << irq));
}
```

代码 6-60 enable_irq（节自\chapter6\f\lib\klib.asm）

```
enable_irq:
        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
```

这两段代码逻辑都很清晰，与 enable_irq 等价的一段 C 代码如下：

```
if(irq < 8){
    out_byte(INT_M_CTLMASK, in_byte(INT_M_CTLMASK) & ~(1 << irq));
}
else{
    out_byte(INT_S_CTLMASK, in_byte(INT_S_CTLMASK) & ~(1 << irq));
}
```

现在在 tinix_main()中指定时钟中断处理程序：

代码 6-61　时钟中断相关（节自\chapter6\f\kernel\main.c）

```
/* 设定时钟中断处理程序 */
put_irq_handler(CLOCK_IRQ, clock_handler);
/* 让 8259A 可以接收时钟中断 */
enable_irq(CLOCK_IRQ);
```

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 6-25 所示。

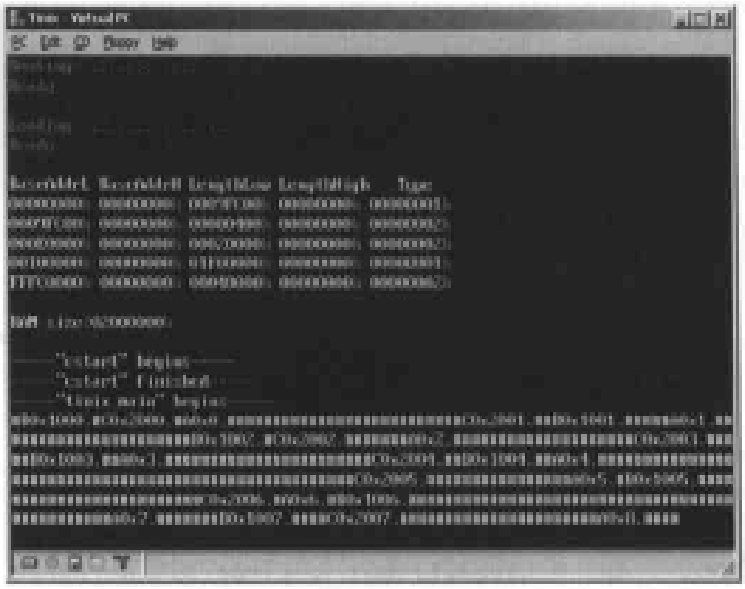

图 6-25　代码整理后的 Tinix（请参考\chapter6\f）

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 Tinix 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可爱的 Tinix 是怎样运转起来的，如图 6-26 所示。

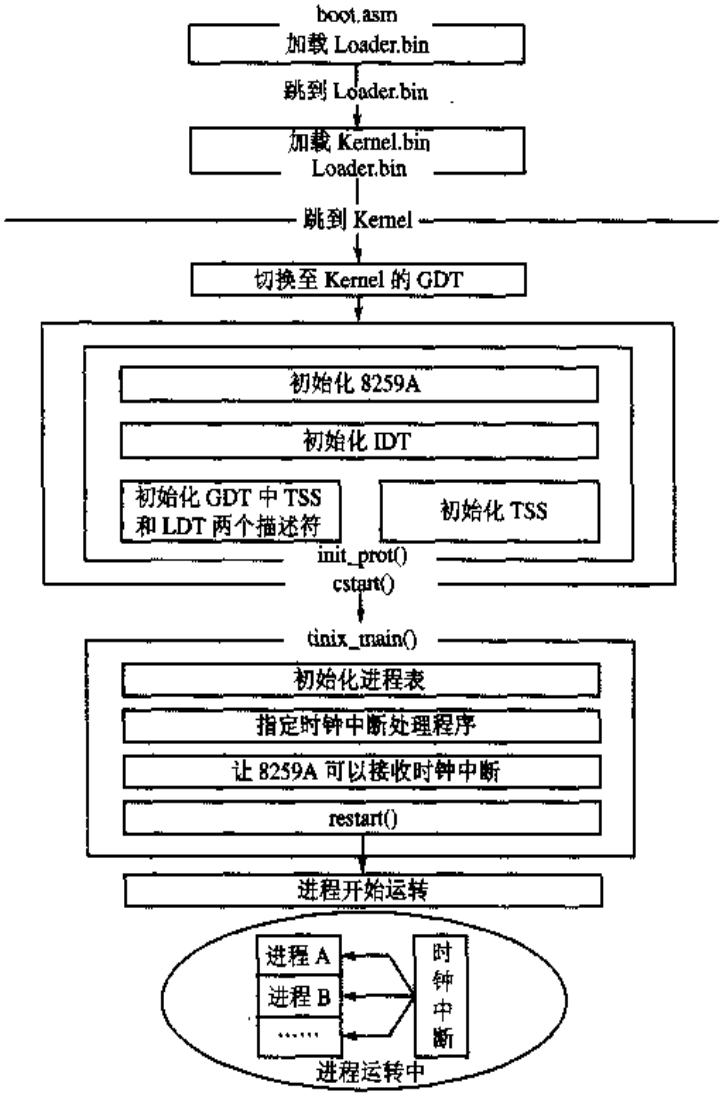

图 6-26 Tinix 的运转过程

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 6-26 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任何改动的，init_prot()中的init_8259A()看上去是比较稳定的代码，tinix_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

如果你没有听说过系统调用，那么你一定听说过 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 Tinix 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，问题就又涉及到了特权级变换。

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

既然这么简单，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。我们这样来写 get_ticks()：

代码 6-62 get_ticks（节自\chapter6\g\kernel\syscall.asm）

```
_NR_get_ticks                equ 0
INT_VECTOR_SYS_CALL          equ 0x90

get_ticks:
    mov     eax, _NR_ get_ticks
    int     INT_VECTOR_SYS_CALL
    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 protect.c 的 init_ prot()中，紧跟初始化其他中断门的语句：

```
init_idt_desc(INT_VECTOR_SYS_CALL, DA_386IGate, sys_call,
              PRIVILEGE_USER);
```

可以看到，第 INT_VECTOR_SYS_CALL 号中断对应的中断例程是 sys_call。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 修改后的 save（节自\chapter6\g\kernel\kernel.asm）

```
save:
    ......
    mov     esi, esp

    inc     dword [k_reenter]
    cmp     dword [k_reenter], 0
    jne     .1
    mov     esp, StackTop
    push    restart
    jmp     [esi + RETADR - P_STACKBASE]
.1:
    push    restart_reenter
    jmp     [esi + RETADR - P_STACKBASE]
```

现在可以写 sys_call 了，请看代码 6-64。它基本上是 hwint_master 宏的简化，甚至连对相应处理程序的调用都类似，那里是 call [irq_table + 4 * %1]（即调用了 irq_table[%1]），这里是 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（节自\chapter6\g\kernel\kernel.asm）

```
sys_call:
    call    save
    sti

    call    [sys_call_table + eax * 4]
    mov     [esi + EAXREG - P_STACKBASE], eax

    cli
    ret
```

sys_call_table 的定义在 global.c 中，实际上目前它只有一个成员：

```
PUBLIC t_sys_call sys_call_table[NR_SYS_CALL] = {sys_get_ticks};
```

其中，t_sys_call 是在 type.h 中定义的：

```
typedef void*   t_sys_call;
```

这样，无论系统调用何种函数，都不会有编译时错误。

前面我们已经把 eax 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]语句调用的便是 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 sys_get_ticks（节自\chapter6\g\kernel\proc.c）

```
PUBLIC int sys_ get_ticks ()
{
    disp_str("+");
    return 0;
}
```

继续看代码 6-64，如果你不明白 mov [esi + EAXREG - P_STACKBASE], eax 这一句的意义，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 中添加以下函数声明：

代码 6-66 函数声明（节自\chapter6\g\include\proto.h）

```
PUBLIC  int         sys_ get_ticks ();  /* t_sys_call */
PUBLIC  void        sys_call();         /* t_pf_int_handler */
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 函数声明（节自\chapter6\g\include\proto.h）

```
void TestA()
{
    int i = 0;
    while(1){
        get_ticks ();
        disp_str("A");
        disp_int(i++);
    ......
```

最后别忘了在 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 6-27 所示。

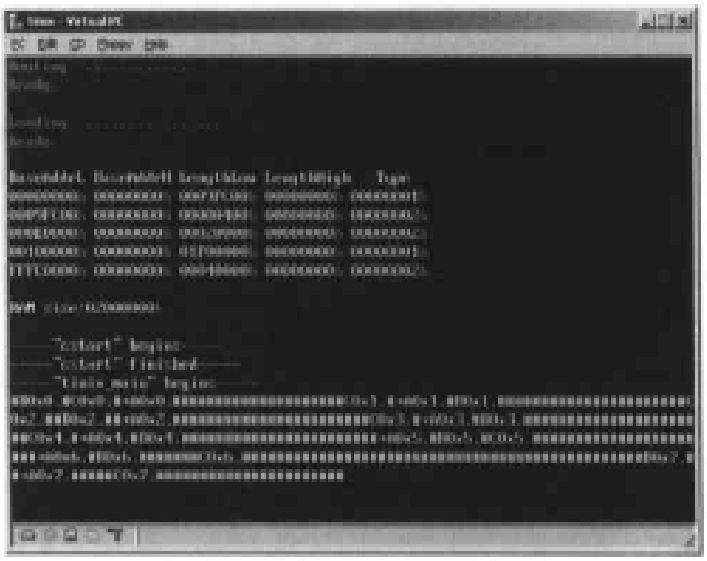

图 6-27　第一个系统调用开始运行

从图 6-27 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 clock_handler(int irq)中添加如下一句：

```
ticks++;
```

然后修改 sys_get_ticks()：

代码 6-68　修改后的 sys_get_ticks（节自\chapter6\g\ kernel \proc.c）

```
PUBLIC int sys_get_ticks()
{
    return ticks;
}
```

最后修改 TestA，我们不再打印递增的 i 了，改成打印当前 ticks：

代码 6-69　修改后的 TestA（节自\chapter6\g\kernel\main.c）

```
void TestA()
{
    while(1){
        disp_str("A");
        disp_int(get_ticks());
```

```
            disp_str(".");
            delay(1);
        }
    }
```

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，因为这样才像 Tinix，用 jiffies 就变成 Tinux 了。

好了，下面我们可以试一下了，结果如图 6-28 所示。

图 6-28 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 A0x18，如果你注意数一下，会发现两次打印之间的#恰好是 0x18（即 24 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：receive、send 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加此实现一个 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 Tinix 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，我们并不一定涉及到，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
Counter 0	输出到 IRQ0，以便每隔一段时间让系统产生一次时钟中断
Counter 1	通常被设为 18，以便大约每 15μs 做一次 RAM 刷新
Counter 2	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 1 193 180Hz。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 65 535，因此，默认的时钟中断的发生频率就是 1 193 180/65 536 ≈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 1 193 180/100≈11 931。

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。

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6-29所示。

表6-2 8253端口

端 口	描 述
40h	8253 Counter 0
41h	8253 Counter 1
42h	8253 Counter 2
43h	8253模式控制寄存器（Mode Control Register）

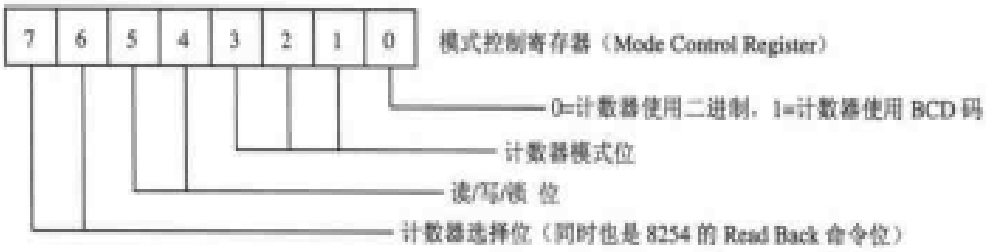

图6-29 8253模式控制寄存器

其中各部分的细节分列如下：

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：模式的选择不是惟一的，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 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 Read/Write/Latch 位

位		描 述
7	6	
0	0	选择 Counter 0
0	1	选择 Counter 1
1	0	选择 Counter 2
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 Counter 0，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。

下面来添加设置计数值的代码：

代码 6-70 设置 8253（节自\chapter6\h\kernel\main.c）

```
out_byte(TIMER_MODE, RATE_GENERATOR);
out_byte(TIMER0, (t_8) (TIMER_FREQ/HZ) );
out_byte(TIMER0, (t_8) ((TIMER_FREQ/HZ) >> 8));
```

其中各个宏的定义如下：

代码 6-71 有关 8253 的宏定义（节自\chapter6\h\include\const.h）

```
#define TIMER0              0x40
#define TIMER_MODE          0x43
#define RATE_GENERATOR      0x34
#define TIMER_FREQ          1193182L
#define HZ                  100
```

6.5.2.2 不太精确的延迟函数

通过代码 6-70，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，你会看到图 6-30 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。

现在我们就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：

代码6-72 精确到10ms的延迟函数（节自\chapter6\h\include\const.h）

```
PUBLIC void milli_delay(int milli_sec)
{
    int t = get_ticks();
    while(((get_ticks() - t) * 1000 / HZ) < milli_sec) {}
}
```

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。

图6-30 重新设置8253之后

接下来我们修改进程A的进程体：

代码6-73 修改进程A（节自\chapter6\h\ kernel\main.c）

```
void TestA()
{
    while(1){
```

```
            disp_str("A");
            disp_int(get_ticks());
            disp_str(".");
            milli_delay(1000);
        }
    }
```

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 6-31 所示的结果。

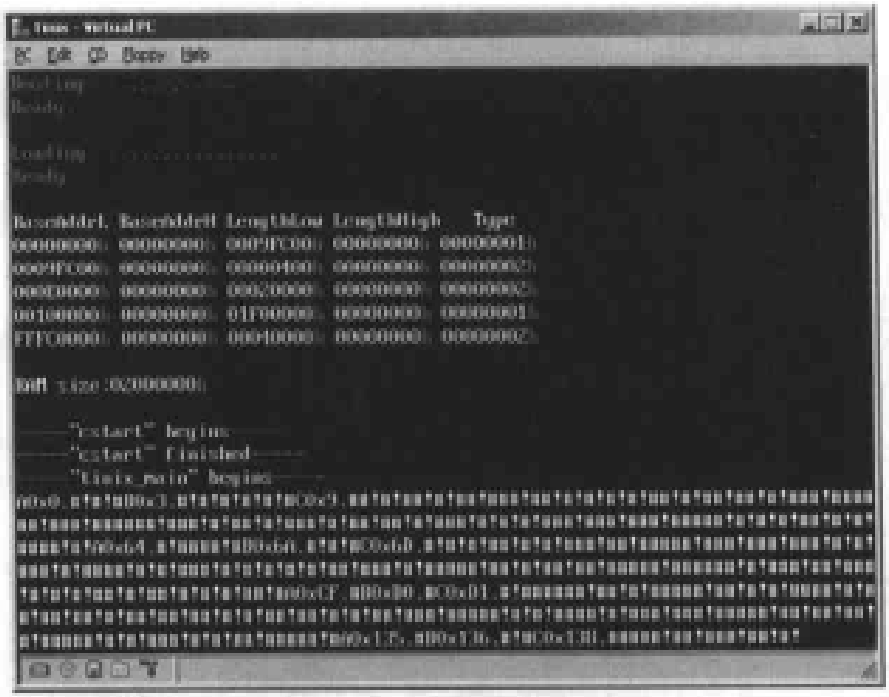

图 6-31　使用 milli_delay 的进程

从图 6-31 中看出，发生了很多次中断重入，由于现在进入内核态只有时钟中断和 get_ticks 系统调用两种方式，所以重入发生的惟一情况是调用 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。

但是细心的你一定已经发现，第 3 次和第 2 次打印 A 之间发生了 0x6B（0xCF-0x64），也就是 107 次中断，milli_delay 的误差这一次大约有 70ms。笔者另外做了若干次试验，最后发现有时候误差是小的，可能是 0ms 或者 10ms，有时候则比较大，比如刚才的 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（这可以通过修改 NR_TASKS 和 task_table[NR_TASKS]来实现），然后把中断例程中打印字符#和!的代码也去掉，再运行一次，会发现情况变得好多了（如图 6-32 所示），每一次的间隔都是 0x64，也就是 100 个 ticks。

图 6-32 只有一个进程运行时 milli_delay 的执行情况

可见多进程的确影响到了延迟的精度。

但是很明显，我们不能为了让这个延迟函数运行良好就只运行一个进程，因此，它的误差注定是大的。不过话说回来，虽然它的精度还很差，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。

而且根据目测你可以发现，两次打印的时间间隔大致是 1s，这与真实的机器上运行是基本一样的。如果你用 Bochs 运行一下会发现时间间隔偏差较大，这也是我们仍然使用 Virtual PC 而没完全转向 Bochs 的原因之一。Virtual PC 在很多方面更接近真实的计算机。

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 6-74 修改 3 个进程的延迟时间（节自\chapter6\i\kernel\main.c）

```
void TestA()
{
    while(1){
        disp_str("A.");
        milli_delay(300);
    }
}

void TestB()
{
    while(1){
        disp_str("B.");
        milli_delay(900);
    }
}

void TestC()
{
    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 6-33 所示。

数一数（用眼睛数不是个好主意，Virtual PC 的菜单“Edit→Copy”可以把输出复制成文本，写个小程序数一段文本中字母个数还是很容易的）可以知道，输出中共有 A 字母 117 个，B 字母 40 个，C 字母 25 个，所以 A 和 B 的个数之比是 2.925，A 和 C 的个

数之比是4.68，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）恰好是吻合的。

图6-33 让不同的进程休息不同的时间

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6-75 为进程表添加新的成员（节自\chapter6\\include\proc.h）

```
typedef struct s_proc {
```

```
    ......
    int             ticks;          /* remained ticks */
    int             priority;
    ......
```

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 最初赋值如下：

代码 6-76 ticks 和 priority 的初值（节自\chapter6\i\ kernel\main.c）

```
    proc_table[0].ticks = proc_table[0].priority = 150;
    proc_table[1].ticks = proc_table[1].priority =  50;
    proc_table[2].ticks = proc_table[2].priority =  30;
```

对于进程调度，我们可以单独写一个函数，叫做 schedule()，放在 proc.c 中：

代码 6-77 进程调度（节自\chapter6\i\ kernel\proc.c）

```
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) {
                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;
            }
        }
    }
}
```

同时修改时钟中断处理函数为下面的样子：

代码 6-78 修改后的时钟中断（节自\chapter6\i\ kernel\clock.c）

```
PUBLIC void clock_handler(int irq)
{
```

```
    ticks++;
    p_proc_ready->ticks--;

    if (k_reenter != 0) {
        return;
    }

    schedule();
}
```

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 6-34 所示的情形。

图 6-34　基于优先级的进程调度

从图 6-34 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，当前输出的 A、B、C 三个字母的个数之比是 59∶31∶23，大体相当于 2.57∶1.35∶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 6-79　修改后的进程（节自\chapter6\\ kernel\main.c）

```
void TestA()
{
```

```
        while(1){
            disp_color_str("A.", BRIGHT | MAKE_COLOR(BLACK, RED));
            disp_int(get_ticks());
            milli_delay(200);
        }
    }

    void TestB()
    {
        while(1){
            disp_color_str("B.", BRIGHT | MAKE_COLOR(BLACK, RED));
            disp_int(get_ticks());
            milli_delay(200);
        }
    }

    void TestC()
    {
        while(1){
            disp_color_str("C.", BRIGHT | MAKE_COLOR(BLACK, RED));
            disp_int(get_ticks());
            milli_delay(200);
        }
    }
```

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 6-80　修改后的 schedule（节自\chapter6\\ kernel\proc.c）

```
    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 tinix_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 6-81 修改后的 tinix_main（节自\chapter6\i\ kernel\main.c）

```
    disp_pos = 0;
    for (i=0; i<80*25; i++) {
        disp_str(" ");
    }
    disp_pos = 0;
```

再次运行，结果如图 6-35 所示。

图 6-35 打印出的进程调度情况

结合图 6-35 并稍加分析会发现，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

	进程中每次循环耗费的ticks	最初的100 ticks执行的循环数	后来的20*2 ticks执行的循环数	再后来的30*3 ticks执行的循环数	总共执行的循环数		结果的产生原因
进程A	20ticks	5	2	4	5+2+4	11	230/20 = 11.5
进程B	20ticks	由于优先级低而未被调度	2	4	2+4	6	130/20 = 6.5
进程C	20ticks	由于优先级低而未被调度	由于优先级低而未被调度	4	4	4	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 6-35 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、6、4 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 6-34 和图 6-35）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 11/6/4 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 6-82 修改后的 clock_handler（节自\chapter6\l\ kernel\clock.c）

```
if (p_proc_ready->ticks > 0) {
    return;
}
```

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 6-36 所示。

图 6-36 进程调度策略改进后的调度情况

从图 6-36 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。

下面，我们把现在的 schedule()重新改回原来的样子（代码 6-80 重新变回代码 6-77）。然后把 main.c 中进程体代码变回代码 6-79 的样子，并且把代码 6-81 的清屏代码去掉，make，运行，就显现图 6-37 所示的情形。可以明显看出，进程 A 在 150ticks 内执行 7~8 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 6-83 修改后的优先级（节自\chapter6\l\ kernel\main.c）

```
proc_table[0].ticks = proc_table[0].priority = 15;
proc_table[1].ticks = proc_table[1].priority = 5;
proc_table[2].ticks = proc_table[2].priority = 3;
```

然后把各个进程的延迟时间改成 10ms：

```
milli_delay(10);
```

图 6-37　进程调度改进后的执行情况

再运行一下，结果如图 6-38 所示。

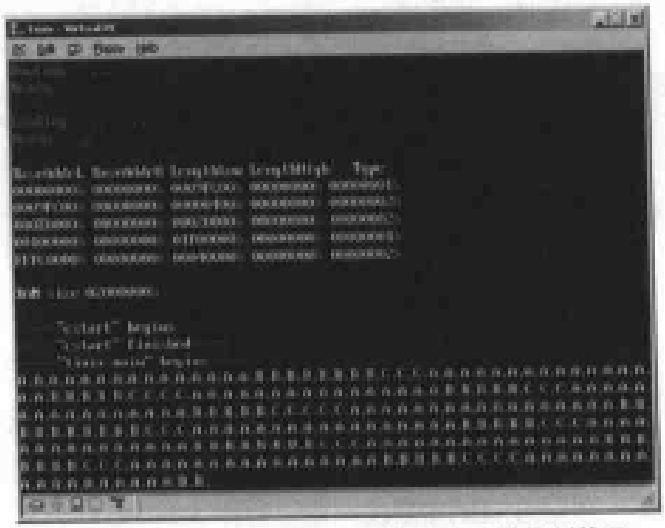

图 6-38　优先级和进程延迟时间改变后的执行情况

从这次结果可以看出，打印出的字符的个数之比精确地等于 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。

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。

第7章

输入/输出系统

欲穷千里目，更上一层楼。

——王之涣

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，Tinix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，你可能一下子就想起了8259A，其中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了（见代码5-66和图5-35）。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，添加如下函数：

代码 7-1　键盘中断处理程序（节自\chapter7\a\kernel\keyboard.c）

```
public void keyboard_handler(int irq)
{
    disp_str("*");
}
```

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。

最后添加指定中断处理程序和打开键盘中断的代码：

代码 7-2　打开键盘中断（节自\chapter7\a\kernel\keyboard.c）

```
/* 设定键盘中断处理程序 */
put_irq_handler(keyboard_irq, keyboard_handler);
/* 让 8259A 可以接收键盘中断 */
enable_irq(keyboard_irq);
```

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；如果是一个方的扁口，那么它很可能是个 USB 键盘。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”。

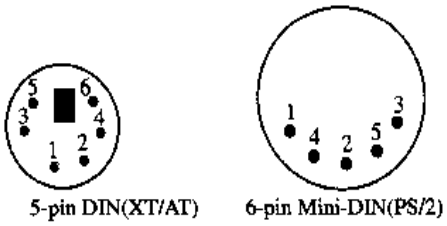

图 7-2 “5-pin DIN”和“6-pin Mini-DIN（XT/AT）”接口

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）。

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做扫描码（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。

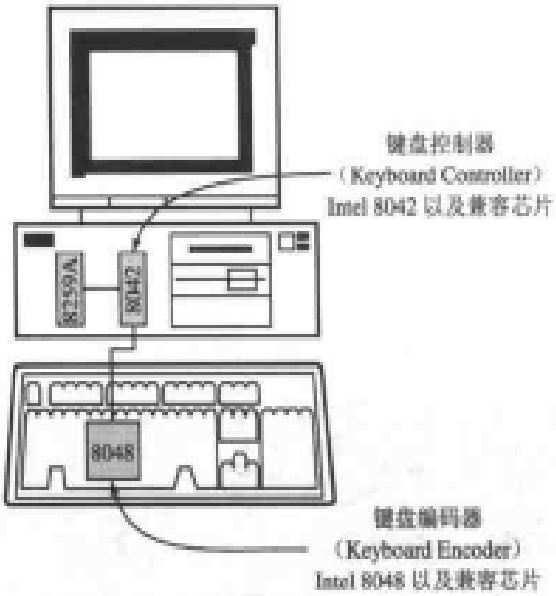

图 7-3　键盘和主机连接示意图

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 Tinix 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(KB_DATA);
```

其中 KB_DATA 被定义成 0x60。

运行，结果如图 7-4 所示。

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

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 7-3　修改键盘中断（节自\chapter7\a\kernel\keyboard.c）

```
t_8 scan_code = in_byte(KB_DATA);
disp_int(scan_code);
```

接下来在运行的时候按两个键：a 和 9，看出现什么？结果如图 7-5 所示。

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。

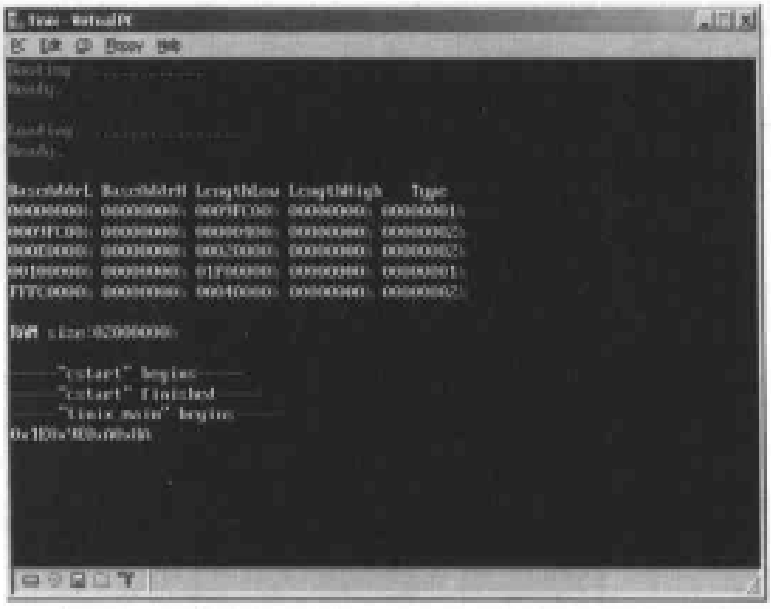

图 7-5　每发生一次键盘中断就读取一次 8042 输出缓冲区并打印读取的值

> **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：0x2A0x1E0x9 E0xAA，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？

表 7-2　Scan code set 1 扫描码

Key	Make	BReak	Key	Make	Break	Key	Make	Break
A	1E	9E	Tab	0F	8F	KP -	4A	CA

续表

Key	Make	Break	Key	Make	Break	Key	Make	Break
B	30	B0	C Lock	3A	BA	KP +	4E	CE
C	2E	AE	L Shift	2A	AA	KP EN	E0,1C	E0,9C
D	20	A0	L Ctrl	1D	9D	KP .	53	D3
E	12	92	L GUI	E0,5B	E0,DB	KP 0	52	D2
F	21	A1	L Alt	38	B8	KP 1	4F	CF
G	22	A2	R Shift	36	B6	KP 2	50	D0
H	23	A3	R Ctrl	E0,1D	E0,9D	KP 3	51	D1
I	17	97	R GUI	E0,5C	E0,DC	KP 4	4B	CB
J	24	A4	R Alt	E0,38	E0,B8	KP 5	4C	CC
K	25	A5	APPS	E0,5D	E0,DD	KP 6	4D	CD
L	26	A6	Enter	1C	9C	KP 7	47	C7
M	32	B2	Esc	1	81	KP 8	48	C8
N	31	B1	F1	3B	BB	KP 9	49	C9
O	18	98	F2	3C	BC	]	1B	9B
P	19	99	F3	3D	BD	;	27	A7
Q	10	19	F4	3E	BE	'	28	A8
R	13	93	F5	3F	BF	,	33	B3
S	1F	9F	F6	40	C0	.	34	B4
T	14	94	F7	41	C1	/	35	B5
U	16	96	F8	42	C2	Power	E0, 5E	E0, DE
V	2F	AF	F9	43	C3	Sleep	E0, 5F	E0, DF
W	11	91	F10	44	C4	Wake	E0, 63	E0, E3
X	2D	AD	F11	57	D7	Next Track*	E0, 19	E0, 99
Y	15	95	F12	58	D8	Previous Track*	E0, 10	E0, 90
Z	2C	AC	Print Screen	E0,2A, E0,37	E0,B7, E0,AA	Stop*	E0, 24	E0, A4
0	0B	8B	Scroll	46	C6	Play/Pause*	E0, 22	E0, A2
1	2	82	Pause	E1,1D,45, E1,9D,C5	-NONE-	Mute*	E0, 20	E0, A0
2	3	83	[	1A	9A	Volume Up*	E0, 30	E0, B0
3	4	84	Insert	E0,52	E0,D2	Volume Down*	E0, 2E	E0, AE
4	5	85	Home	E0,47	E0,C7	Media Select*	E0, 6D	E0, ED
5	6	86	PageUp	E0,49	E0,C9	E-mail*	E0, 6C	E0, EC

续表

Key	Make	Break	Key	Make	Break	Key	Make	Break
6	7	87*	Delete	E0,53	E0,D3	Calculator*	E0, 21	E0, A1
7	8	88	End	E0,4F	E0,CF	My Computer*	E0, 6B	E0, EB
8	9	89	PageDown	E0,51	E0,D1	WWW Search*	E0, 65	E0, E5
9	0A	8A	U ARROW	E0,48	E0,C8	WWW Home*	E0, 32	E0, B2
`	29	89	L ARROW	E0,4B	E0,CB	WWW Back*	E0, 6A	E0, EA
-	0C	8C	D ARROW	E0,50	E0,D0	WWW Forward*	E0, 69	E0, E9
=	0D	8D	R ARROW	E0,4D	E0,CD	WWW Stop*	E0, 68	E0, E8
\	2B	AB	Num	45	C5	WWW Refresh*	E0, 67	E0, E7
Backspace	0E	8E	KP /	E0,35	E0,B5	WWW Favorites*	E0, 66	E0, E6
Space	39	B9	KP *	37	B7			

*表示的是 Windows 多媒体键盘专有按钮

不过，我们马上就可以根据扫描码来处理键盘输入了，只要根据这个表格建立一种对应关系就够了。因此，忘掉这些不快吧。

7.1.4　解析扫描码

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 7-4 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 7-4　扫描码解析数组（节自\chapter7\a\include\keymap.h）

```
t_32 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,
......
/* 0x0F - TAB        */  TAB,            TAB,            0,
/* 0x10 - 'q'        */  'q',            'Q',            0,
/* 0x11 - 'w'        */  'w',            'W',            0,
/* 0x12 - 'e'        */  'e',            'E',            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,
......
/* 0x35 - '/'        */  '/',            '?',    PAD_SLASH,
/* 0x36 - r. SHIFT   */  SHIFT_R,        SHIFT_R,        0,
/* 0x37 - '*'        */  '*',            '*',            0,
/* 0x38 - ALT        */  ALT_L,          ALT_L,      ALT_R,
/* 0x39 - ' '        */  ' ',            ' ',            0,
/* 0x3A - CapsLock   */  CAPS_LOCK,      CAPS_LOCK,      0,
/* 0x3B - F1         */  F1,             F1,             0,
......
/* 0x44 - F10        */  F10,            F10,            0,
/* 0x45 - NumLock    */  NUM_LOCK,       NUM_LOCK,       0,
/* 0x46 - ScrLock    */  SCROLL_LOCK,    SCROLL_LOCK,    0,
/* 0x47 - Home       */  PAD_HOME,       '7',         HOME,
/* 0x48 - CurUp      */  PAD_UP,         '8',            UP,
......
/* 0x56 - ???        */  0,              0,              0,
/* 0x57 - F11        */  F11,            F11,            0,
/* 0x58 - F12        */  F12,            F12,            0,
......
/* 0x5B - ???        */  0,              0,          GUI_L,
/* 0x5C - ???        */  0,              0,          GUI_R,
/* 0x5D - ???        */  0,              0,           APPS,
......
};
```

举个例子，如果获得的扫描码是 0x1F，我们应该在代码 7-4 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 7-1 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。

7.1.4.1　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 7-5　键盘缓冲区（节自\chapter7\a\include\keyboard.h）

```
typedef struct s_kb {
    char*   p_head;                 /* 指向缓冲区中下一个空闲位置 */
    char*   p_tail;                 /* 指向键盘任务应处理的字节 */
    int     count;                  /* 缓冲区中共有多少字节 */
    char    buf[KB_IN_BYTES];       /* 缓冲区 */
}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。

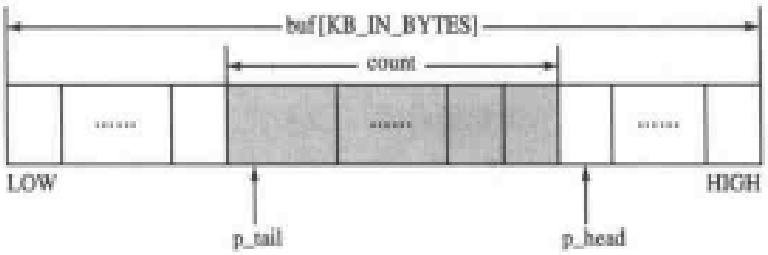

图 7-6 键盘缓冲区示意图

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 7-6 所示。

代码 7-6 修改后的 keyboard_handler（节自\chapter7\a\kernel\keyboard.c）

```
PUBLIC void keyboard_handler(int irq)
{
  t_8 scan_code = in_byte(KB_DATA);

  if (kb_in.count < KB_IN_BYTES) {
      *(kb_in.p_head) = scan_code;
      kb_in.p_head++;
      if (kb_in.p_head == kb_in.buf + KB_IN_BYTES) {
          kb_in.p_head = kb_in.buf;
      }
      kb_in.count++;
  }
}
```

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）：

```
PRIVATE KB_INPUT kb_in;
```

注意，kb_in 的成员需要初始化，可以写一个单独的函数 init_keyboard()来完成这件事。

代码 7-7 init_keyboard（节自\chapter7\a\kernel\keyboard.c）

```
PUBLIC void init_keyboard()
{
  kb_in.count = 0;
  kb_in.p_head = kb_in.p_tail = kb_in.buf;

  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 */
}
```

在 init_keyboard 中不但初始化了 kb_in，而且将设定和开启键盘中断的工作也挪了过来。

为了保持 tinix_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 7-8　init_clock（节自\chapter7\a\kernel\clock.c）

```
PUBLIC void init_clock()
{
    out_byte(TIMER_MODE, RATE_GENERATOR);
    out_byte(TIMER0, (t_8) (TIMER_FREQ/HZ) );
    out_byte(TIMER0, (t_8) ((TIMER_FREQ/HZ) >> 8));
    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 */
}
```

这样，在 tinix_main()中调用这两个函数就可以了（见代码 7-9）。

代码 7-9　初始化时钟和键盘（节自\chapter7\a\kernel\main.c）

```
init_clock();
init_keyboard();
```

7.1.4.2　添加任务

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　tty 任务（节自\chapter7\a\kernel\tty.c）

```
PUBLIC void task_tty()
{
```

```
    while (1) {
        keyboard_read();
    }
}
```

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()可以这样来定义：

代码 7-11 keyboard_read（节自\chapter7\a\kernel\keyboard.c）

```
PUBLIC void keyboard_read()
{
    t_8 scan_code;

    if(kb_in.count > 0){
        disable_int();
        scan_code = *(kb_in.p_tail);
        kb_in.p_tail++;
        if (kb_in.p_tail == kb_in.buf + KB_IN_BYTES) {
            kb_in.p_tail = kb_in.buf;
        }
        kb_in.count--;
        enable_int();

        disp_int(scan_code);
    }
}
```

其中，disable_int()和 enable_int()的定义很简单：

代码 7-12 disable_int 和 enable_int（节自\chapter7\a\lib\klib.asm）

```
; void disable_int();
disable_int:
    cli
    ret
; void enable_int();
enable_int:
    sti
    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。

7.1.4.3　解析过程

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。

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。

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 7-13 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（scan_code & 0x7F）这样一个“与”操作。一方面，如果当前扫描码是 Break Code，进行“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。

Tinix 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。

代码 7-13　解析扫描码（节自\chapter7\a\kernel\keyboard.c）

```
PUBLIC void keyboard_read()
```

```
{
    t_8     scan_code;
    char    output[2];
    t_bool  make;       /* TRUE : make  */
                        /* FALSE: break */

    memset(output, 0, 2);

    if(kb_in.count > 0){
        disable_int();

        scan_code = *(kb_in.p_tail);
        kb_in.p_tail++;
        if (kb_in.p_tail == kb_in.buf + KB_IN_BYTES) {
            kb_in.p_tail = kb_in.buf;
        }
        kb_in.count--;

        enable_int();

        /* 下面开始解析扫描码 */
        if (scan_code == 0xE1) {
            /* 暂时不做任何操作 */
        }
        else if (scan_code == 0xE0) {
            /* 暂时不做任何操作 */
        }
        else { /* 下面处理可打印字符 */

            /* 首先判断Make Code 还是 Break Code */
            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);
            }
        }
    }
}
```

图 7-7　可以显示键入的小写字母和数字（请参考\chapter7\a）

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让 Tinix 接收更复杂的输入，比如识别 Shift 什么的。现在按下 Shift 只能看到一个奇怪的字符。下面就来添加以下代码，使其能够响应这些功能键。

代码 7-14　解析扫描码（节自\chapter7\b\kernel\keyboard.c）

```
......
PRIVATE     t_bool      code_with_E0 = FALSE;
PRIVATE     t_bool      shift_l;        /* l shift state    */
PRIVATE     t_bool      shift_r;        /* r shift state    */
PRIVATE     t_bool      alt_l;          /* l alt state      */
PRIVATE     t_bool      alt_r;          /* r left state     */
PRIVATE     t_bool      ctrl_l;         /* l ctrl state     */
PRIVATE     t_bool      ctrl_r;         /* l ctrl state     */
PRIVATE     int         column = 0;     /* keyrow[column] 将是
                                            keymap 中某一个值 */
......
PUBLIC void keyboard_read()
{
    t_8 scan_code;
    char output[2];
    t_bool  make;       /* TRUE : make  */
                        /* FALSE: break */
    t_32    key = 0;    /* 用一个整型来表示一个键*/
```

```
t_32*    keyrow;     /* 指向 keymap[] 的某一行 */

if(kb_in.count > 0){
    scan_code = get_byte_from_kb_buf();
    /* 下面开始解析扫描码 */
    if (scan_code == 0xE1) {
        /* 暂时不做任何操作 */
    }
    else if (scan_code == 0xE0) {
        code_with_E0 = TRUE;
    }
    else {
        /* 首先判断是 Make Code 还是 Break Code */
        make = (scan_code & FLAG_BREAK ? FALSE : TRUE);

        /* 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 = FALSE;
        }

        key = keyrow[column];

        switch(key) {
        case SHIFT_L:
            shift_l = make;
            break;
        case SHIFT_R:
            shift_r = make;
            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:
            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，当我们按下右 Shift 键，shift_l 就变为 TRUE，如果它立即被释放，则 shift_l 又变回 FALSE。如果当右 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，如今的 Tinix 已经可以识别大写字母，以及!、@等字符了。

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，Tinix 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（节自\chapter7\b\kernel\keyboard.c）

```
PUBLIC void keyboard_read()
{
    ......
```

```
if(kb_in.count > 0){
    code_with_E0 = FALSE;
    scan_code = get_byte_from_kb_buf();
    /* 下面开始解析扫描码 */
    if (scan_code == 0xE1) {
        int i;
        t_B pausebreak_scan_code[] = {0xE1, 0x1D, 0x45,
        0xE1, 0x9D, 0xC5};
        t_bool is_pausebreak = TRUE;
        for(i=1;i<6;i++){
            if (get_byte_from_kb_buf() != pausebreak_
            scan_code[i]) {
                is_pausebreak = FALSE;
                break;
            }
        }
        if (is_pausebreak) {
            key = PAUSEBREAK;
        }
    }
    else if (scan_code == 0xE0) {
        scan_code = get_byte_from_kb_buf();

        /* PrintScreen 被按下 */
        if (scan_code == 0x2A) {
            if (get_byte_from_kb_buf() == 0xE0) {
                if (get_byte_from_kb_buf() == 0x37) {
                    key = PRINTSCREEN;
                    make = TRUE;
                }
            }
        }
        /* PrintScreen 被释放 */
        if (scan_code == 0xB7) {
            if (get_byte_from_kb_buf() == 0xE0) {
                if (get_byte_from_kb_buf() == 0xAA) {
                    key = PRINTSCREEN;
                    make = FALSE;
                }
            }
        }
```

```
                /* 不是 PrintScreen，此时 scan_code 为 0xE0 紧跟的那个
                值。 */
                if (key == 0) {
                    code_with_E0 = TRUE;
                }
            }
            if ((key != PAUSEBREAK) && (key != PRINTSCREEN)) {
            ......
    ......
    PRIVATE t_8 get_byte_from_kb_buf() /* 从键盘缓冲区中读取下一个字节 */
    {
        t_8 scan_code;

        while (kb_in.count <= 0) {} /* 等待下一个字节到来 */

        disable_int();
        scan_code = *(kb_in.p_tail);
        kb_in.p_tail++;
        if (kb_in.p_tail == kb_in.buf + KB_IN_BYTES) {
            kb_in.p_tail = kb_in.buf;
        }
        kb_in.count--;
        enable_int();

        return scan_code;
    }
```

这段代码有点长，但实际上除了把函数 get_byte_from_kb_buf()单独拿出来之外，只是单独处理了 Pause 和 Print Screen 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　解析扫描码（节自\chapter7\b\kernel\keyboard.c）

```
.....
if ((key != PAUSEBREAK) && (key != PRINTSCREEN)) {
        .....
        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　in_process（节自\chapter7\b\kernel\tty.c）

```
PUBLIC void in_process(t_32 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，可以看到如下情形：

代码 7-18 不可打印字符的定义（节自\chapter7\b\kernel\keyboard.c）

```
.....
/* 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       */
......
```

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（请参考\chapter7\b）

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 Tinix 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。

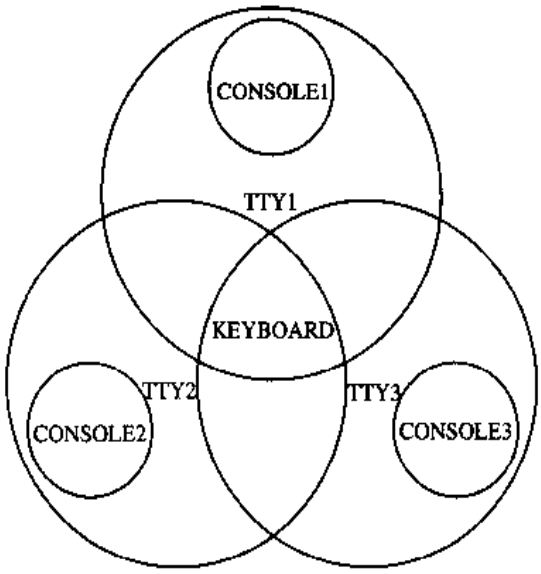

图 7-10　TTY

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共用同一块显存，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，显存的范围为0xB8000~0xBFFFF，共计32KB。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，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。

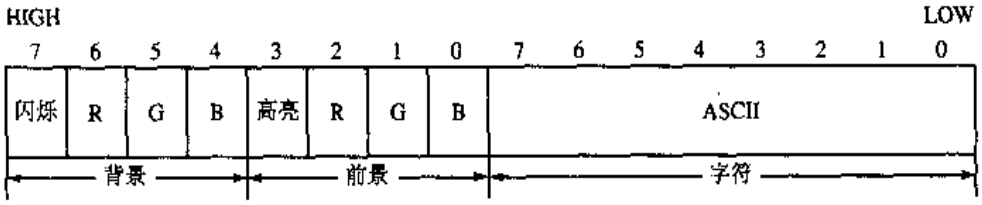

图7-11 屏幕上每一个字符对应的2字节

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	
BI	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：

80×25×2=4000 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		端口	
		R	W
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。

表 7-5　CRT Controller Data Registers

寄存器名称	索　引
Horizontal Total Register	00h
End Horizontal Display Register	01h
Start Horizontal Blanking Register	02h
End Horizontal Blanking Register	03h
Start Horizontal Retrace Register	04h
End Horizontal Retrace Register	05h
Vertical Total Register	06h
Overflow Register	07h
Preset Row Scan Register	08h
Maximum Scan Line Register	09h
Cursor Start Register	0Ah
Cursor End Register	0Bh
Start Address High Register	**0Ch**
Start Address Low Register	**0Dh**
Cursor Location High Register	**0Eh**
Cursor Location Low Register	**0Fh**
Vertical Retrace Start Register	10h
Vertical Retrace End Register	11h
Vertical Display End Register	12h
Offset Register	13h
Underline Location Register	14h
Start Vertical Blanking Register	15h
End Vertical Blanking	16h
CRTC Mode Control Register	17h
Line Compare Register	18h

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 看做一个数组，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 设置光标位置（节自\chapter7\c\kernel\tty.c）

```
PUBLIC void in_process(t_32 key)
{
    char    output[2] = {'\0', '\0'};

    if (!(key & FLAG_EXT)) {
            output[0] = key & 0xFF;
            disp_str(output);

            disable_int();
            out_byte(CRTC_ADDR_REG, CRTC_DATA_IDX_CURSOR_H);
            out_byte(CRTC_DATA_REG, ((disp_pos/2)>>8)&0xFF);
            out_byte(CRTC_ADDR_REG, CRTC_DATA_IDX_CURSOR_L);
            out_byte(CRTC_DATA_REG, (disp_pos/2)&0xFF);
            enable_int();
    }
}
```

其中，几个宏的定义如下（其中还定义了我们之后会用到的另两个寄存器的索引值）：

代码 7-20 CRTC 相关的宏（节自\chapter7\c\include\const.h）

```
#define CRTC_ADDR_REG                   0x3D4
#define CRTC_DATA_REG                   0x3D5
#define CRTC_DATA_IDX_START_ADDR_H      0xC
#define CRTC_DATA_IDX_START_ADDR_L      0xD
#define CRTC_DATA_IDX_CURSOR_H          0xE
#define CRTC_DATA_IDX_CURSOR_L          0xF
```

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。如果你不太明白，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

```
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)) {
                /* Shift + Up */
                        disable_int();
                        out_byte(CRTC_ADDR_REG,
                                CRTC_DATA_IDX_START_ADDR_H);
                        out_byte(CRTC_DATA_REG, ((80*15) >> 8) & 0xFF);
                        out_byte(CRTC_ADDR_REG,
                                CRTC_DATA_IDX_START_ADDR_L);
                        out_byte(CRTC_DATA_REG, (80*15) & 0xFF);
                        enable_int();
                }
```

```
            break;
        default:
            break;
        }
    }
```

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。

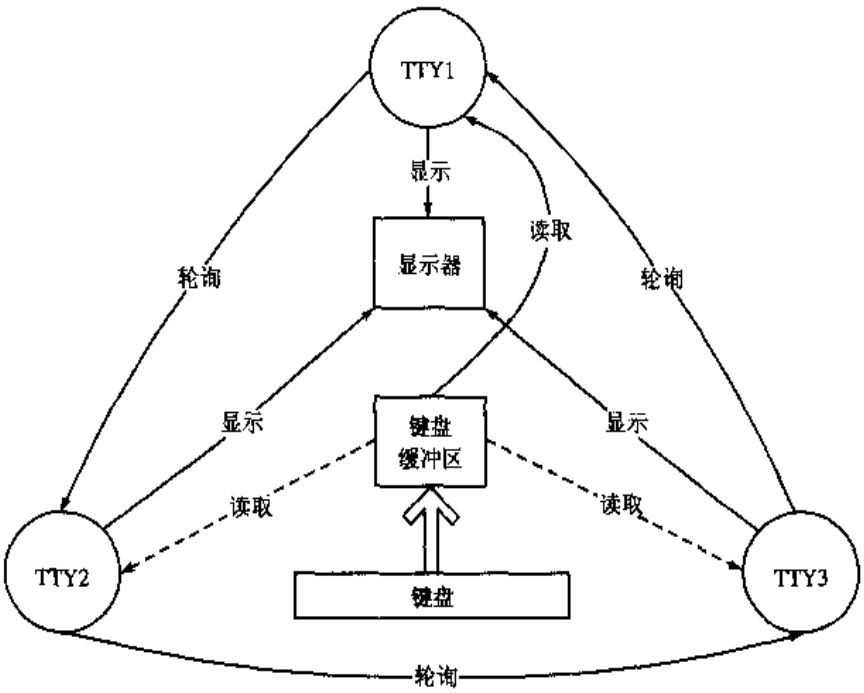

图 7-14　TTY 任务示意图

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）。

（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）对于每一个 TTY，都应该有自己的读和写的动作。所以，在调用 keyboard_read()时，必须让它知道是哪一个 TTY 在调用。我们通过为它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。

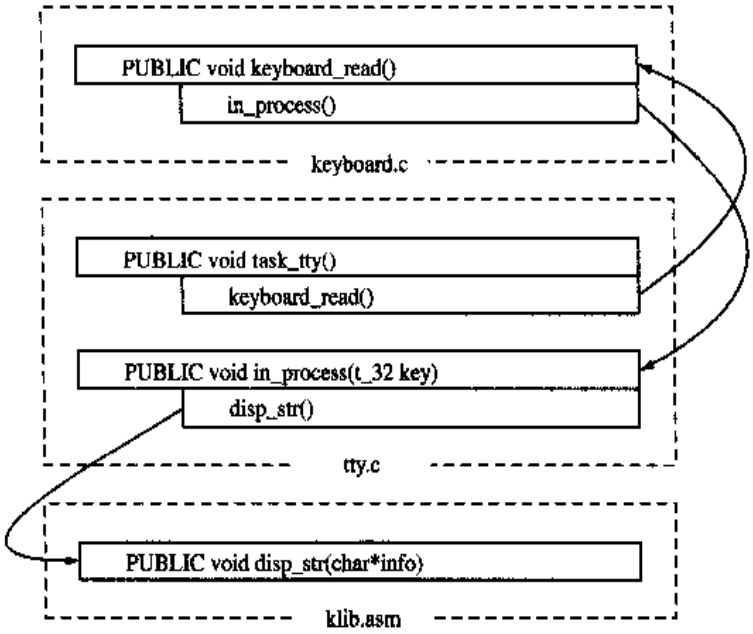

图 7-15 代码\chapter7\b 的 TTY 任务示意图

（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（节自\chapter7\c\include\tty.h）

```
typedef struct s_tty
{
    t_32    in_buf[TTY_IN_BYTES];   /* TTY 输入缓冲区 */
    t_32*   p_inbuf_head;           /* 指向缓冲区中下一个空闲位置 */
    t_32*   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（节自\chapter7\c\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。

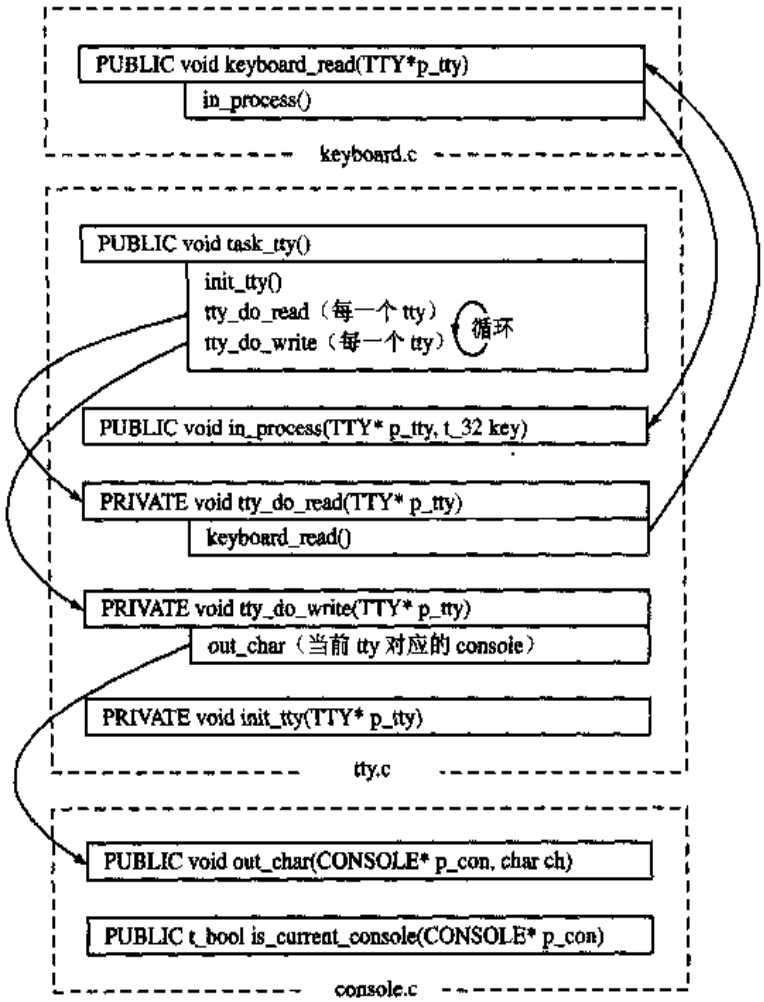

图 7-16　代码\chapter7\c 的 TTY 任务示意图

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（同时也是终端个数，节自\chapter7\c\include\const.h）

```
#define NR_CONSOLES     3
```

代码 7-25 TTY 和 CONSOLE（节自\chapter7\c\kernel\global.c）

```
PUBLIC TTY              tty_table[NR_CONSOLES];
PUBLIC CONSOLE          console_table[NR_CONSOLES];
```

下面来看一下框架性的 task_tty()。

代码 7-26 task_tty（节自\chapter7\c\kernel\tty.c）

```
......
#define TTY_FIRST   (tty_table)
#define TTY_END     (tty_table + NR_CONSOLES)
......
PUBLIC void task_tty()
{
    TTY* p_tty;

    init_keyboard();

    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。由于键盘应被看做是 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（节自\chapter7\c\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如下。源文件 console.c 是新建立的。

代码 7-28　判断是否为当前控制台（节自\chapter7\c\kernel\console.c）

```
PUBLIC t_bool 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　tty_do_read（节自\chapter7\c\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　in_process（节自\chapter7\c\kernel\tty.c）

```
PUBLIC void in_process(TTY* p_tty, t_32 key)
```

```
{
    if (!(key & FLAG_EXT)) {
        if (p_tty->inbuf_count < TTY_IN_BYTES) {
            *(p_tty->p_inbuf_head) = key;
            p_tty->p_inbuf_head++;
            if (p_tty->p_inbuf_head == p_tty->in_buf +
                                        TTY_IN_BYTES) {
                p_tty->p_inbuf_head = p_tty->in_buf;
            }
            p_tty->inbuf_count++;
        }
    }
    else {
    ......
```

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候再来处理。

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 tty_do_write（节自\chapter7\c\kernel\tty.c）

```
PRIVATE void tty_do_write(TTY* p_tty)
{
    if (p_tty->inbuf_count) {
        char ch = *(p_tty->p_inbuf_tail);
        p_tty->p_inbuf_tail++;
        if (p_tty->p_inbuf_tail == p_tty->in_buf + TTY_IN_BYTES) {
            p_tty->p_inbuf_tail = p_tty->in_buf;
        }
        p_tty->inbuf_count--;

        out_char(p_tty->p_console, ch);
    }
}
```

这段代码也很简单，从 TTY 缓冲区中取出键值，然后用 out_char 显示在对应的 CONSOLE 中。从缓冲区中取出一个值的代码类似函数 get_byte_from_ kb_buf()。

我们暂时这样实现 out_char()：

代码 7-32 往控制台输出字符（节自\chapter7\c\kernel\console.c）

```
PUBLIC void out_char(CONSOLE* p_con, char ch)
```

```
{
    t_8* p_vmem = (t_8*)(V_MEM_BASE + disp_pos);

    *p_vmem++ = ch;
    *p_vmem++ = DEFAULT_CHAR_COLOR;
    disp_pos += 2;

    set_cursor(disp_pos/2);
}

PRIVATE void set_cursor(unsigned int position)
{
    disable_int();
    out_byte(CRTC_ADDR_REG, CRTC_DATA_IDX_CURSOR_H);
    out_byte(CRTC_DATA_REG, (position >> 8) & 0xFF);
    out_byte(CRTC_ADDR_REG, CRTC_DATA_IDX_CURSOR_L);
    out_byte(CRTC_DATA_REG, position & 0xFF);
    enable_int();
}
```

V_MEM_BASE 在 const.h 中被定义成了 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。

在整个过程中，由于我们初始化了 nr_current_console 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。

一切良好！不过，框架虽然搭建起来了，我们仍然只是在使用一个 CONSOLE，下面我们就来实现多个 CONSOLE。

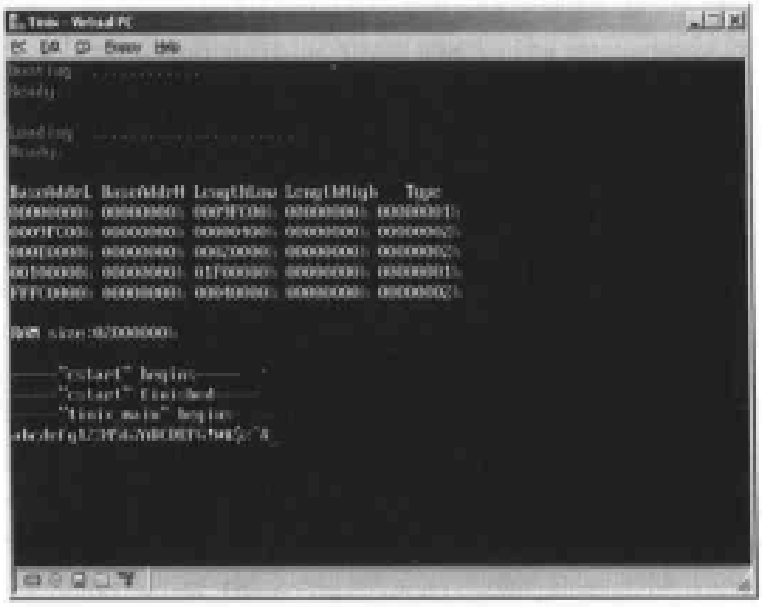

图 7-17　TTY 任务框架已经搭建好（请参考\chapter7\c）

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。

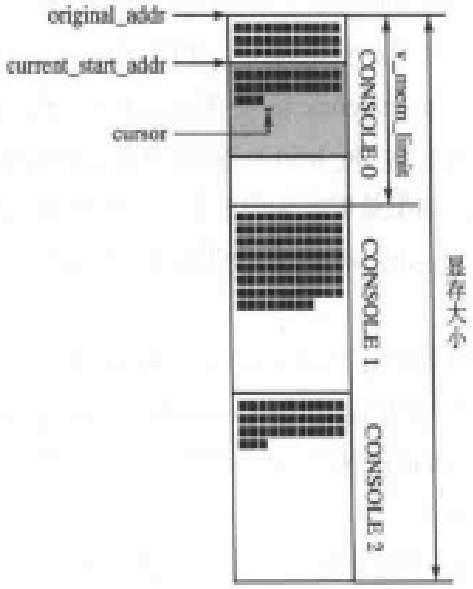

图 7-18　多控制台示意图

图 7-18 清楚地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　初始化控制台（节自\chapter7\d\kernel\console.c）

```
PUBLIC void init_screen(TTY* p_tty)
{
    int nr_tty          = p_tty - tty_table;
    p_tty->p_console    = console_table + nr_tty;

    int v_mem_size      = V_MEM_SIZE >> 1;  /* 显存总大小 (in WORD) */

    int con_v_mem_size  = v_mem_size / NR_CONSOLES;
    p_tty->p_console->original_addr = nr_tty * con_v_mem_size;

    p_tty->p_console->v_mem_limit = con_v_mem_size;
    p_tty->p_console->current_start_addr =
                            p_tty->p_console->original_addr;

    p_tty->p_console->cursor = p_tty->p_console->original_addr;
    if (nr_tty == 0) {
        p_tty->p_console->cursor = disp_pos / 2;   /* 第一个控制台延
                                                      用原来的光标位置 */
    }
    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）第一个控制台延用原来的光标位置，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()如下：

代码 7-34 修改后的初始化 TTY（节自\chapter7\d\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 修改后的 out_char（节自\chapter7\d\kernel\console.c）

```
PUBLIC void out_char(CONSOLE* p_con, char ch)
{
    t_8* p_vmem = (t_8*)(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，参见代码 7-36。

代码 7-36 切换控制台的代码（节自\chapter7\d\kernel\console.c）

```
PUBLIC void select_console(int nr_console) /* 0 - (NR_CONSOLES - 1) */
{
    if ((nr_console < 0) || (nr_console >= NR_CONSOLES)) {
        /*invalid console number */
        return;
    }

    nr_current_console = nr_console;

    set_cursor(console_table[nr_console].cursor);
    set_video_start_addr(
                console_table[nr_console].current_start_addr);
}
```

其中，函数 set_video_start_addr()我们已经很熟悉了：

代码 7-37　函数 set_video_start_addr()（节自\chapter7\d\kernel\console.c）

```
PRIVATE void set_video_start_addr(t_32 addr)
{
    disable_int();
    out_byte(CRTC_ADDR_REG, CRTC_DATA_IDX_START_ADDR_H);
    out_byte(CRTC_DATA_REG, (addr >> 8) & 0xFF);
    out_byte(CRTC_ADDR_REG, CRTC_DATA_IDX_START_ADDR_L);
    out_byte(CRTC_DATA_REG, addr & 0xFF);
    enable_int();
}
```

按照惯例，我们应该在按下 Alt+Fn 时做切换工作：

代码 7-38　处理 Alt+Fn（节自\chapter7\d\kernel\tty.c）

```
PUBLIC void in_process(TTY* p_tty, t_32 key)
{
    ......
    else {
        ......
        case F1:
        case F2:
        case F3:
        ......
        case F12:
            if ((key & FLAG_ALT_L) || (key & FLAG_ALT_R)) {
                /* Alt + F1~F12 */
                select_console(raw_code - F1);
            }
            break;
        default:
            ......
```

这样，我们就可以把原来 tty_task()中直接将 nr_current_console 赋值为 0 的语句换成对的调用了：

```
select_console(0);
```

这样，我们就可以运行一下看看了，结果如图 7-19 所示。

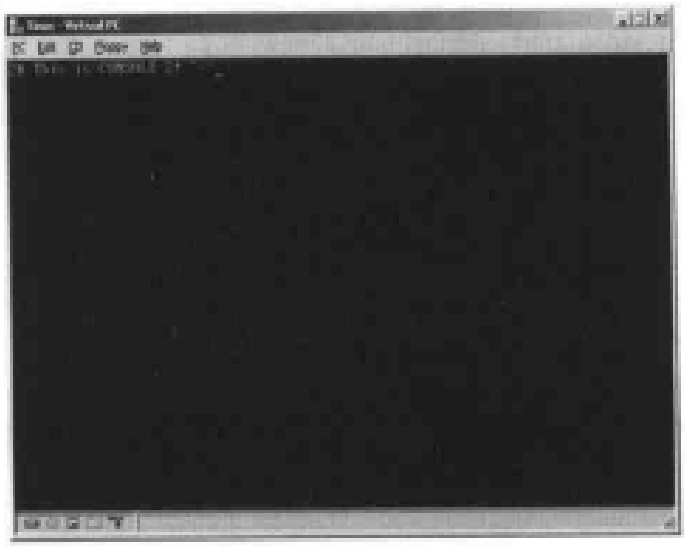

图 7-19　多控制台演示

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 7-39　scroll_screen（节自\chapter7\d\kernel\console.c）

```
PUBLIC void scroll_screen(CONSOLE* p_con, int direction)
{
    if (direction == SCROLL_SCREEN_UP) {
        if (p_con->current_start_addr > p_con->original_addr) {
            p_con->current_start_addr -= SCREEN_WIDTH;
        }
    }
    else if (direction == SCROLL_SCREEN_DOWN) {
        if (p_con->current_start_addr + SCREEN_SIZE <
                p_con->original_addr + p_con->v_mem_limit) {
            p_con->current_start_addr += SCREEN_WIDTH;
        }
    }
    else{
    }
    set_video_start_addr(p_con->current_start_addr);
    set_cursor(p_con->cursor);
}
```

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 7-40　响应 Shift+↑和 Shift+↓（节自\chapter7\d\kernel\tty.c）

```
PUBLIC void in_process(TTY* p_tty, t_32 key)
{
        ......
        case UP:
             if ((key & FLAG_SHIFT_L) || (key & FLAG_SHIFT_R))
                  /* Shift + Up */
                  scroll_screen(p_tty->p_console,
                                SCROLL_SCREEN_UP);
             break;
        case DOWN:
             if ((key & FLAG_SHIFT_L) || (key & FLAG_SHIFT_R))
                  /* Shift + Down */
                  scroll_screen(p_tty->p_console,
                                SCROLL_SCREEN_DOWN);
             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。

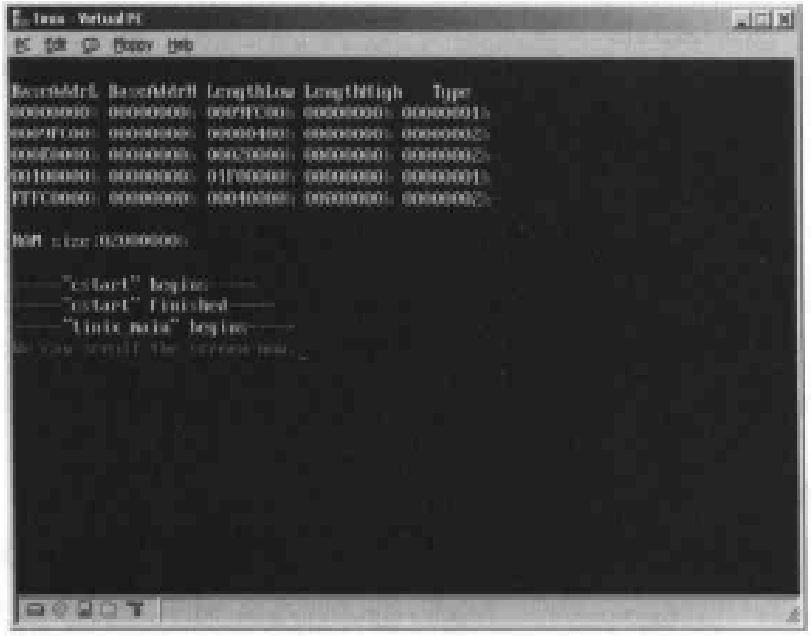

图 7-20　屏幕滚动演示

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 7-41。

代码 7-41 响应回车键和退格键（节自\chapter7\e\kernel\tty.c）

```
PUBLIC void in_process(TTY* p_tty, t_32 key)
{
    if (!(key & FLAG_EXT)) {
        put_key(p_tty, key);
    }
    else {
        int raw_code = key & MASK_RAW;
        switch(raw_code) {
        case ENTER:
            put_key(p_tty, 0x0A);
            break;
        case BACKSPACE:
            put_key(p_tty, '\b');
            break;
        ......
}

PRIVATE void put_key(TTY* p_tty, t_32 key)
{
    if (p_tty->inbuf_count < TTY_IN_BYTES) {
        *(p_tty->p_inbuf_head) = key;
        p_tty->p_inbuf_head++;
        if (p_tty->p_inbuf_head == p_tty->in_buf + TTY_IN_BYTES){
            p_tty->p_inbuf_head = p_tty->in_buf;
        }
        p_tty->inbuf_count++;
```

```
    }
}
```

然后修改 out_char()如下：

代码 7-42　修改 out_char()（节自\chapter7\e\kernel\tty.c）

```
PUBLIC void out_char(CONSOLE* p_con, char ch)
{
    t_8* p_vmem = (t_8*)(V_MEM_BASE + p_con->cursor * 2);

    switch(ch) {

    case '\n':
        if (p_con->cursor<p_con->original_addr +
                p_con->v_mem_limit-SCREEN_WIDTH){
            p_con->cursor = p_con->original_addr + SCREEN_WIDTH *
                            ((p_con->cursor - p_con->original_addr)
                              / SCREEN_WIDTH + 1);
        }
        break;
    case '\b':
        if (p_con->cursor > p_con->original_addr) {
            p_con->cursor--;
            *(p_vmem-2) = ' ';
            *(p_vmem-1) = DEFAULT_CHAR_COLOR;
        }
        break;
    default:
        if (p_con->cursor < p_con->original_addr +
                p_con->v_mem_limit - 1) {
            *p_vmem++ = ch;
            *p_vmem++ = DEFAULT_CHAR_COLOR;
            p_con->cursor++;
        }
        break;
    }

    while (p_con->cursor >= p_con->current_start_addr +
                SCREEN_SIZE) {
        scroll_screen(p_con, SCROLL_SCREEN_DOWN);
    }
```

```
    flush(p_con);
}

PRIVATE void flush(CONSOLE* p_con)
{
    set_cursor(p_con->cursor);
    set_video_start_addr(p_con-> current_start_addr);
}
```

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 Enter and BackSpace are available now.－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　回车键和换行键可用

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。

我们的目的是要往 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。

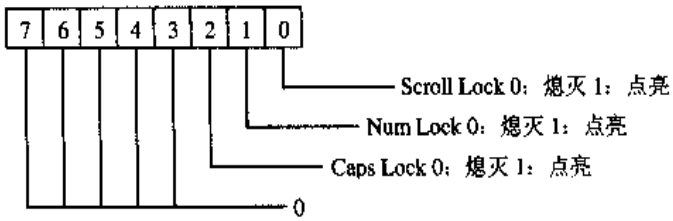

图 7-22　设置 LED 的参数字节

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 7-43　设置 LED（节自\chapter7\e\kernel\keyboard.c）

```
PRIVATE void kb_wait()  /* 等待 8042 的输入缓冲区空 */
{
    t_8 kb_stat;
    do {
        kb_stat = in_byte(KB_CMD);
    } while (kb_stat & 0x02);
}

PRIVATE void kb_ack()
{
    t_8 kb_read;
    do {
        kb_read = in_byte(KB_DATA);
    } while (kb_read =! KB_ACK);
```

```
}

PRIVATE void set_leds()
{
    t_8 leds = (caps_lock << 2) | (num_lock << 1) | scroll_lock;
    kb_wait();
    out_byte(KB_DATA, LED_CODE);
    kb_ack();
    kb_wait();
    out_byte(KB_DATA, leds);
    kb_ack();
}
```

代码中的 LED_CODE 和 KB_ACK 分别被定义成 0xED 和 0xFA，而 caps_lock、num_lock 和 scroll_lock 三个变量的声明和初始化如下：

代码 7-44 初始化 LEDs 的状态（节自\chapter7\e\kernel\keyboard.c）

```
......
PRIVATE t_bool      caps_lock;      /* Caps Lock        */
PRIVATE t_bool      num_lock;       /* Num Lock         */
PRIVATE t_bool      scroll_lock;    /* Scroll Lock      */
......
PUBLIC void init_keyboard()
{
    kb_in.count = 0;
    kb_in.p_head = kb_in.p_tail = kb_in.buf;

    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 Tinix 启动之后 NumLock 被点亮。

虽然灯亮了，但实际上还未起到作用，我们现在就来修改 keyboard_read()：

代码 7-45　使用 CapsLock、NumLock、小键盘（节自\chapter7\e\kernel\keyboard.c）

```
......
case CAPS_LOCK:
    if (make) {
        caps_lock   = !caps_lock;
        set_leds();
        }
        break;
    case NUM_LOCK:
        if (make) {
            num_lock    = !num_lock;
            set_leds();
        }
        break;
    case SCROLL_LOCK:
        if (make) {
            scroll_lock = !scroll_lock;
            set_leds();
        }
        break;
    default:
        break;
    }
}

if(make){ /* 忽略 Break Code */
    t_bool pad = FALSE;
    /* 首先处理小键盘 */
    if ((key >= PAD_SLASH) && (key <= PAD_9)) {
        pad = TRUE;
        switch(key) { /* '/', '*', '-', '+', and 'Enter' in num
        pad  */
        case PAD_SLASH:
            key = '/';
            break;
        case PAD_STAR:
            key = '*';
            break;
```

```
case PAD_MINUS:
        key = '-';
        break;
case PAD_PLUS:
        key = '+';
        break;
case PAD_ENTER:
        key = ENTER;
        break;
default: /* keys whose value depends on the NumLock */
        if (num_lock) { /* '0' ~ '9' and '.' in num pad */
                if ((key >= PAD_0) && (key <= PAD_9)) {
                        key = key - PAD_0 + '0';
                }
                else if (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 |= pad? FLAG_PAD : 0;
        in_process(p_tty, key);
    }
  }
}
```

代码虽长，但很容易理解。惟一要提的是，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 Caps Lock 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（请参考\chapter7e）

图中的字符是在点亮 Caps Lock 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。

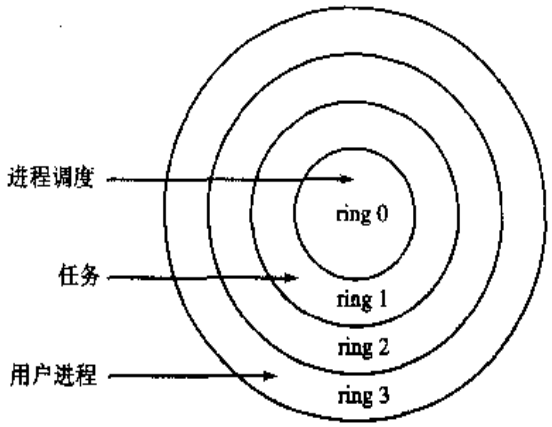

图 7-24 区分任务和用户进程

除了 ring2 未加使用之外，这个图和图 3-10 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 7-46 （节自\chapter7\f\include\proc.h）

```
#define NR_TASKS        1
```

```
#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 7-47　（节自\chapter7\f\kernel\global.c）

```
PUBLIC PROCESS proc_table[NR_TASKS + NR_PROCS];
PUBLIC TASK task_table[NR_TASKS] = {
                                {task_tty, STACK_SIZE_TTY, "tty"}
                                };
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 7-48　（节自\chapter7\f\kernel\main.c）

```
t_8 privilege;
t_8 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 */
    }
```

```
    /* name of the process*/
    strcpy(p_proc->name, p_task->name);

    p_proc->pid        = i;                /* pid */

    p_proc->ldt_sel    = selector_ldt;
    memcpy(&p_proc->ldts[0],&gdt[SELECTOR_KERNEL_CS>>3],
          sizeof(DESCRIPTOR));
    /* change the DPL */
    p_proc->ldts[0].attr1 = DA_C | privilege << 5;
    memcpy(&p_proc->ldts[1],&gdt[SELECTOR_KERNEL_DS>>3],
          sizeof(DESCRIPTOR));
    /* change the DPL */
    p_proc->ldts[1].attr1 = DA_DRW | privilege << 5;
    p_proc->regs.cs    = ((8 * 0) & SA_RPL_MASK & SA_TI_MASK) |
                         SA_TIL | rpl;
    p_proc->regs.ds    = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) |
                         SA_TIL | rpl;
    p_proc->regs.es    = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) |
                         SA_TIL | rpl;
    p_proc->regs.fs    = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) |
                         SA_TIL | rpl;
    p_proc->regs.ss    = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) |
                         SA_TIL | rpl;
    p_proc->regs.gs    = (SELECTOR_KERNEL_GS & SA_RPL_MASK) | rpl;
    p_proc->regs.eip   = (t_32)p_task->initial_eip;
    p_proc->regs.esp   = (t_32)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 7-49 和代码 7-50。

代码 7-49 （节自\chapter7\kernel\protect.c）

```
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),
                DA_LDT);
        p_proc++;
        selector_ldt += 1 << 3;
    }
```

代码 7-50　（节自\chapter7\f\kernel\proc.c）

```
    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\chapter7\e 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 Tinix 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 7-51 （节自\chapter7\g\include\proc.h）

```
typedef struct s_proc {
    ......
    int             nr_tty;
}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 7-52 （节自\chapter7\g\kernel\main.c）

```
for(i=0;i<NR_TASKS+NR_PROCS;i++){
        ......
        p_proc->nr_tty = 0;
        ......
}
......
proc_table[1].nr_tty = 0;
proc_table[2].nr_tty = 1;
proc_table[3].nr_tty = 1;
......
```

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 7-53 （节自\chapter7\g\kernel\printf.c）

```
int printf(const char *fmt, ...)
```

```
{
    int i;
    char buf[64];
    va_list arg = (va_list)((char*)(&fmt) + 4);    /* 4 是参数 fmt
                                                     所占堆栈中的大小 */
    i = vsprintf(buf, fmt, arg);
    write(buf, i);
    return i;
}
```

其中，vsprintf()的实现方法如下：

代码 7-54 （节自\chapter7\g\kernel\printf.c）

```
int vsprintf(char *buf, const char *fmt, va_list args)
{
    char* p;
    char tmp[64];
    va_list p_next_arg = args;

    for (p=buf;*fmt;fmt++) {
        if (*fmt != '%') {
            *p++ = *fmt;
            continue;
        }

        fmt++;

        switch (*fmt) {
        case 'x':
            itoa(tmp, *((int*)p_next_arg));
            strcpy(p, tmp);
            p_next_arg += 4;
            p += strlen(tmp);
            break;
        default:
            break;
        }
    }

    return (p - buf);
}
```

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 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 7-53 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 7-53 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。

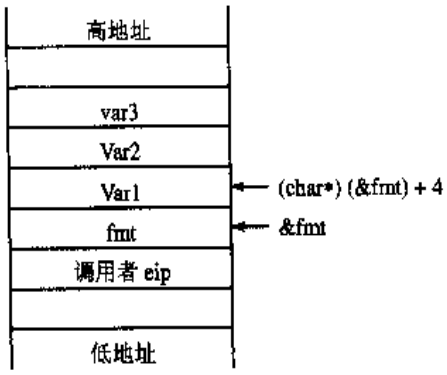

图 7-25 printf 调用后的堆栈情况

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。

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 7-54，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 7-53 和代码 7-54 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 7-53 中的 write()便是我们即将完成的系统调用了，它负责把 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 7-55　（节自\chapter7\g\include\proto.h）

```
PUBLIC int sys_write (char* buf, int len, PROCESS* p_proc);
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()：

代码 7-56　（节自\chapter7\g\kernel\syscall.asm）

```
write:
        mov eax, _NR_write
        mov ebx, [esp + 4]
        mov ecx, [esp + 8]
        int INT_VECTOR_SYS_CALL
        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()：

代码 7-57 （节自\chapter7\g\kernel\tty.asm）

```
PUBLIC void tty_write(TTY* p_tty, char* buf, int len)
{
    char* p = buf;
    int i = len;

    while (i) {
        out_char(p_tty->p_console, *p++);
        i--;
    }
}

PUBLIC int sys_write(char* buf, int len, PROCESS* p_proc)
{
    tty_write(&tty_table[p_proc->nr_tty], buf, len);
    return 0;
}
```

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 7-58 （节自\chapter7\g\kernel\kernel.asm）

```
sys_call:
    call    save
    push    dword [p_proc_ready]
    sti
    push    ecx
    push    ebx
    call    [sys_call_table + eax * 4]
    add     esp, 4 * 3
    mov     [esi + EAXREG - P_STACKBASE], eax
    cli
    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 个用户进程中添加调用它的代码：

代码 7-59　（节自\chapter7\g\kernel\main.c）

```
void TestA()
{
    while(1){
        printf("<Ticks:%x>", get_ticks());
        milli_delay(200);
    }
}
void TestB()
{
    while(1){
        printf("B");
        milli_delay(200);
    }
}
void TestC()
{
    while(1){
        printf("C");
        milli_delay(200);
    }
}
```

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。

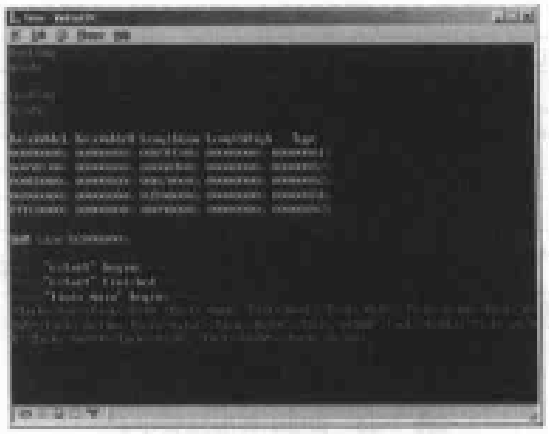

图 7-26　printf 打印出的字符和数字（控制台 0）

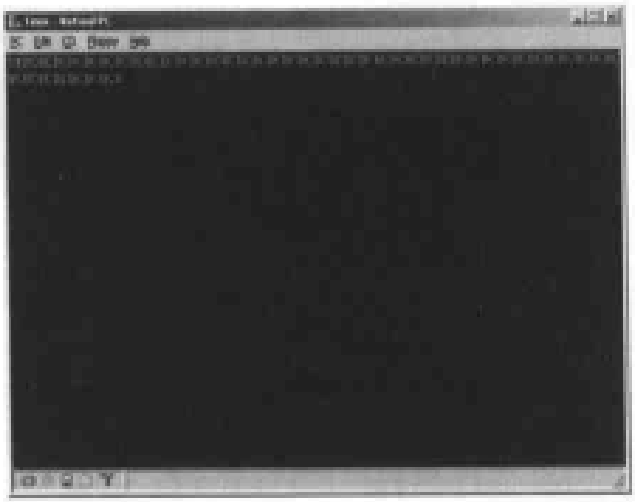

图 7-27　printf 打印出的字符（控制台 1）

太棒了，我们终于有了自己的 printf()，从此不但可以告别原来的 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。

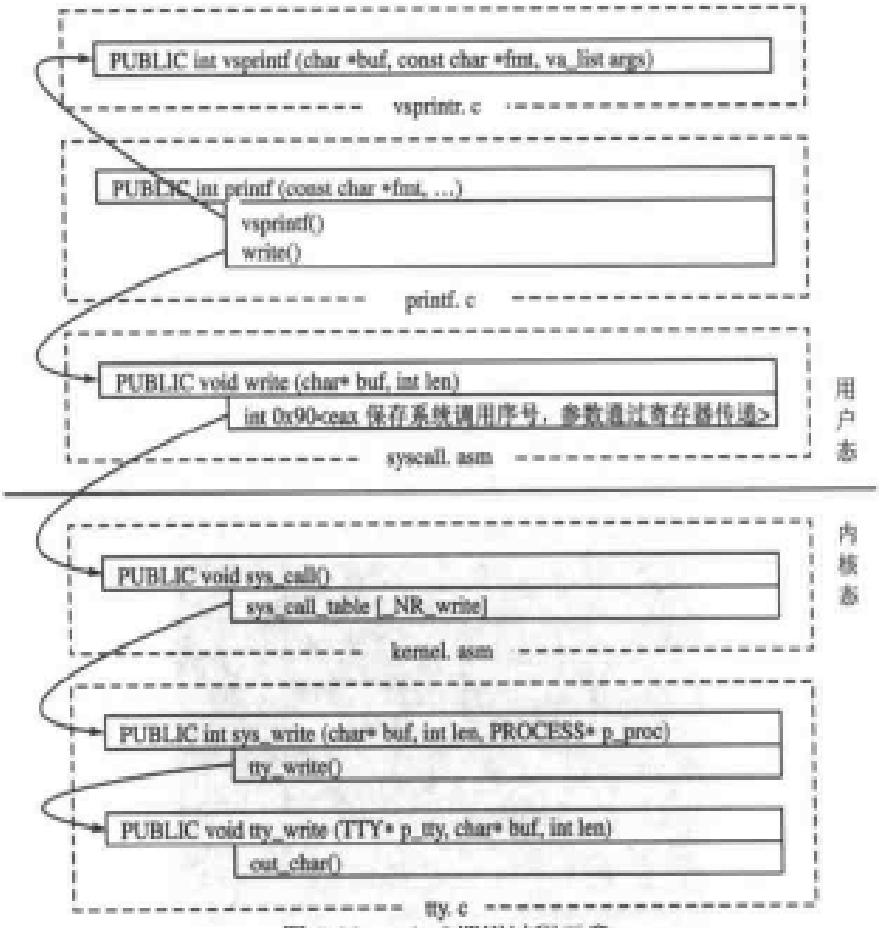

图 7-28　printf 调用过程示意

一个系统调用涉及到特权级的切换，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。

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。

后　记

路漫漫其修远兮，吾将上下而求索。

——屈原

本书已经到尾声了，但很明显，Tinix 作为一个操作系统，这仅仅才是个开头。然而正如前面导读中所说，操作系统这座大山，我们已经走进了山门，甚至走出了一段的距离，欣赏到了相当多的风景。而且本书不但从山门之外就开始讲解，还告诉了读者如何准备登山的装备以及如何热身。这样读者就不再是单纯地欣赏风景，更是参与其中，尽享探险的乐趣。

所以，本书最大的价值就在于，它让操作系统的实现这个问题变得具备了“可操作性”——仿佛随身携带了登山手册。从这个角度来说，纯理论性的书籍就好比从空中俯瞰，大山的结构和脉络尽收眼底，但如果把观者空投进山，则很可能迷路。但有了登山手册则不同，不但大山的各方面细节了然于胸，更是有了登山的方法和最基本的训练。不但不会迷路，而且会迷恋于山中的风景。更加难得的是，等到在山中探险之后再到空中俯瞰，感受就完全不同了。你一定会有全方位、多角度、立体、深刻的感悟。

操作系统无疑是复杂的，正如大山很大，路径很多，而且不同的山也有不同的风景一样，不同的操作系统实现起来也有不同。本书中的 Tinix 还很小，即便将来它变得很大，也不可能将天下山的风景都囊括在内。况且当前它还只是找到了一条进入山门的道路，但即便仅仅是一条并不长的进山之路，读者已经可以据此找到一个突破口，从这个突破口入手，阅读操作系统代码甚至亲手实践一个自己的操作系统都变得近在眼前，而不像从前那样遥不可及。

万事开头难，一点都不假。选择和购买装备、阅读登山手册以及热身的时间，未必比在山中走一段很长的路要少多少。在编写 Tinix 代码的过程中，我也着实有不少的体会，也借此机会跟读者分享一二，希望可以给读者一点启迪和借鉴：

（1）一知半解和透彻的理解之间相差甚远。在阅读一段代码时，读过一两遍之后可能感觉已经读懂了。但只有当亲手实践的时候才会发现有太多细节是之前所没有发现的。而且在动手实践之后，代码中的每个字都会有深刻的印象，并清晰地知道其来龙去脉，这是单纯“阅读”别人的代码所无法获得的体验。

（2）在一开始，我是想完全参考 Minix 来写 Tinix，Tinix 这个名字也是因此得来，不过在具体编写代码的过程中，我渐渐阅读了 Linux 的一部分代码以及其他一些书籍，并也有所借鉴，所以这个名字跟当初的初衷已经有了一些差距。不过这并不重要，有时我发现，拿 Minix 和 Linux 进行某些方面的比较是一件很有趣的事。而且，前面有越多的优秀代码，我们就有越多可供学习的素材。有时，当我遇到一个问题，已经习惯于去读不同操作系统的实现方式，并加以思考，最终选择自己认为适合的方法去做了。

（3）“学然后知不足”。只要不是浮光掠影式的学习，就一定会发现困难重重。在解决困难的时候，不但获得了成就感，而且获得了知识。“教然后知困”，当我在书中试图很清晰地表述一个问题时，也深刻地体会到其中的困难。正因为如此，在书中我把代码分成许多小块，而且配以很多的表格和图片，希望能够帮助读者理解书中的内容。

另外，对于 Tinix 这个操作系统来讲，我觉得其优点和缺点是兼而有之的，比如代码从未考虑过效率问题，这样做的好处是很容易理解，但坏处是效率方面的问题可能在以后表现出来。不过问题并不严重，因为目前我们的内核还很小，而且实现的这一部分也大多是从前辈处借鉴而来的，不存在大方向上的问题。

小和不完整是 Tinix 的另一缺点。它只有如此简单的进程管理和初级的输入/输出系统。但恰恰是这一点，可以使我们在阅读起来不必关心太多其他的细节，从而更容易看到进程本身。从这个方面讲这又变成了它的优点。

已经走进山门，这无疑是令人兴奋的，但前面的风景更加令我们激动。如果你读完本书之后依然保持了最开始时候的热情，迫不及待想要探寻山中更多的宝藏，我可能会比你更加开心，这说明本书做了合格的导游。

当然，在继续前行之前，我们将不得不学习更多的知识，阅读更多的别人的源代码。理论书籍对我们而言已经开始具备实际意义了，因为这些理论知识不但是可以理解的，而且变成了必需研究的课程。

当系统变得越来越庞大的，效率问题以及架构策略就变得越来越重要了。比如，我们已经到了选择宏内核还是微内核的时候，做出决定显然没有那么容易。因为我们看到，Linux 和 Windows NT 分别作为宏内核和微内核的代表，都获得了巨大的成功。相似的问题肯定还有很多，都需要我们去仔细地思考和研究。

但是，无论前面的路有多长，未知的困难有多少，我们至少已经走在满眼风景的路上。而在我们不知道引导扇区为何物的时候，甚至不知道路在哪里。

操作系统是我们正在攀登的大山，更是一座宝库，它不但让我们从中体会到寻根问

底的成就感，在其中更是包含了许许多多前人的智慧。当我们逐渐窥探到这座宝库的门径时，当我们逐渐看到宝库之内的闪闪光亮时，无论前面有多少困难，都无法阻止我们探寻它的步伐。

最后我想说的是，我愿意与所有读者共同学习，探讨操作系统的任何问题，我希望能与你们在讨论中相会。论坛的网址是：http://forum.broadview.com.cn。

参考文献

[1] IA-32 Intel® Architecture Software Developer's Manual – Volume 1: Basic Architecture.2004

[2] IA-32 Intel® Architecture Software Developer's Manual - Volume 2: Instruction Set Reference. 2004

[3] IA-32 Intel® Architecture Software Developer's Manual - Volume 3: System Programming Guide. 2004

[4] 操作系统：设计与实现（第二版）. [美]Andrew S. Tanenbaum，Albert S. Woodhull 著. 王鹏、尤晋元、朱鹏、敖青云译. 电子工业出版社，2001

[5] 操作系统——内核与设计原理（第四版）. [美]William Stallings 著. 魏迎梅、王涌等译. 电子工业出版社，2002

[6] 80x86 汇编语言程序设计教程. 杨季文等编著. 钱培德审. 清华大学出版社

[7] IBM-PC 汇编语言程序设计. 沈美明、温冬婵编著. 清华大学出版社，1996

[8] C 高级实用程序设计. 王士元编著. 清华大学出版社，1999

[9] Linux 内核完全注释. 赵炯. 机械工业出版社，2004

[10] Microsoft Extensible Firmware Initiative FAT 32 File System Specification, FAT: General Overview of On-Disk Format, published by Microsoft (version 1.03, December 6, 2000)

[11] Linux 操作系统分析教程. 骆耀祖主编. 李强主审. 清华大学出版社，北京交通大学出版社，2004

附录

书中的章节和代码对照表

第 1 章					
	1.1				
		1.1.1			
		1.1.2			
	1.2			chapter1\a\	最简单的引导扇区
	1.3				
	1.4				
	1.5			chapter1\b\	可以调试的引导扇区
	1.6				

第 3 章						
	3.1				chapter3\a\pmtest1.asm	进入保护模式
		3.1.1				
		3.1.2				
		3.1.3				
	3.2					
		3.2.1			chapter3\b\pmtest2.asm	访问 5MB 内存
		3.2.2			chapter3\c\pmtest3.asm	使用 LDT
		3.2.3				
			3.2.3.1			
			3.2.3.2			
			3.2.3.3			
				1.		

续

第 3 章						
				2.	chapter3\d\pmtest4.asm	使用调用门跳入一个段
				3.		
				4.		
				5.	chapter3\e\pmtest5.asm	Ring0 →ring3
				6.		在 ring3 使用调用门
			3.2.3.4			
	3.3					
		3.3.1				
		3.3.2			chapter3\f\pmtest6.asm	实现分页机制
		3.3.3				
		3.3.4				
		3.3.5				
		3.3.6			chapter3\g\pmtest7.asm	列出内存情况
		3.3.7			chapter3\h\pmtest8.asm	演示页表切换
	3.4					
		3.4.1				
		3.4.2				
		3.4.3			chapter3\i\pmtest9.asm	演示中断
		3.4.4				
		3.4.5				
		3.4.6				
		3.4.7				
			3.4.7.1			
			3.4.7.2			
			3.4.7.3			
	3.5					
		3.5.1				
		3.5.2				

第 4 章				
	4.1			
		4.1.1		
		4.1.2	chapter4\a\	Boot Sector 加入了软盘头

续

第 4 章				
		4.1.3	chapter4\b\	找到 loader.bin 这个文件名
		4.1.4		
		4.1.5	chapter4\c\	加载 loader.bin 入内存并跳入
		4.1.6		
	4.2			

第 5 章				
	5.1		chapter5\a\	用 NASM 在 Linux 下写 hello.asm
	5.2		chapter5\b\	Linux 下汇编和 C 混合编程
	5.3			
	5.4			
		5.4.1	chapter5\c\	把内核加载到内存
		5.4.2	chapter5\d\	跳入保护模式
		5.4.3	chapter5\e\	重新放置内核
		5.4.4		跳入内核
		5.4.5		
			5.4.5.1	
			5.4.5.2	chapter5\f\	如何用 Bochs 调试
	5.5			
		5.5.1	chapter5\g\	切换堆栈和 GDT（开始使用 C 代码）
		5.5.2	chapter5\h\	整理文件夹并编写 Makefile
		5.5.3		
		5.5.4	chapter5\i\	添加中断处理
		5.5.5		
	5.6			

第 6 章				
	6.1			
	6.2			
		6.2.1		
		6.2.2		
		6.2.3		
	6.3			

续

第6章						
		6.3.1				
			6.3.1.1			
			6.3.1.2			
			6.3.1.3			
			6.3.1.4			
			6.3.1.5			
			6.3.1.6			
			6.3.1.7			
		6.3.2				
			6.3.2.1		chapter6a\	启动第一个进程
			6.3.2.2			
				1.		
				2.		
				3.		
			6.3.2.3			
			6.3.2.4			
			6.3.2.5			
		6.3.3				
			6.3.3.1		chapter6b\	完善中断程序
			6.3.3.2			
			6.3.3.3			
			6.3.3.4			
			6.3.3.5		chapter6c\	考虑中断重入
		6.3.4				
	6.4					
		6.4.1			chapter6d\	
		6.4.2				
		6.4.3				
		6.4.4				
		6.4.5				
		6.4.6				
		6.4.7				
		6.4.8			chapter6e\	代码回顾与整理
					chapter6f\	
	6.5					
		6.5.1			chapter6g\	实现系统调用 get_ticks()

续

第 6 章						
		6.5.2				
			6.5.2.1		chapter6h\	实现 milli_delay()
			6.5.2.2			
	6.6					
		6.6.1			chapter6i\	改进调度算法
		6.6.2				

第 7 章						
	7.1					
		7.1.1			chapter7a\	显示键入的小写字母和数字
		7.1.2				
		7.1.3				
		7.1.4				
			7.1.4.1			
			7.1.4.2			
			7.1.4.3			
				1.		
				2.	chapter7b\	显示所有字符（大小写、数字、标点等）
				3.		
	7.2					
		7.2.1				
		7.2.2				
		7.2.3			chapter7c\	为实现多控制台做准备
	7.3					
		7.3.1				
		7.3.2			chapter7d\	实现多控制台
		7.3.3				
			7.3.3.1		chapter7e\	完善键盘处理
			7.3.3.2			
		7.3.4				
	7.4				chapter7f\	区分任务和用户进程
	7.5					
		7.5.1			chapter7g\	实现 printf()
		7.5.2				
		7.5.3				
		7.5.4				