第1章 GNU 和 Linux 简介

本章知识点：

- 自由的天地：GNU 和 Linux
- 在乐趣中获得成功
- GNU 的开发工具

GNU 的主要精神是软件源代码应该自由流通。只有开放了软件的源代码，才能让软件开发者自由地与其他使用者和开发人员充分交流，共同创造出优秀的软件。

Linux 所有的源代码，都可以被使用者轻易获得，并进行任意的研究和修改。自由的 Linux 为计算机爱好者提供了学习操作系统设计的最好教材。本书的特色是深入地分析 Linux 0.01 的源代码，同时借助 GNU 提供的开发工具指导读者进行操作系统设计实验，使读者快速掌握操作系统设计的基本原理和技术。

自由软件提供了大量的开发工具，如 GCC、make、nasm、ld 等，使用 GNU 丰富的开发工具，学习和编写操作系统是非常方便的。本章将简单介绍有关 GNU 开发工具的相关知识。

1.1 自由的天地：GNU 和 Linux

GNU 是由自由软件基金会（Free Software Foundation，FSF）的董事长 Richard M. Stallman（RMS）于 1984 年发起的，至今已经有 20 年的历史了。

GNU 是 GNU's Not Unix 的缩写。GNU 的创始人 Stallman 认为，Unix 虽然不是最好的操作系统，但是至少不会太差，而他自信有能力把 Unix 不足的地方加以改进，使它成为一个优秀的操作系统。Stallman 把这个更优秀的操作系统命名为 GNU，开发 GNU 的目的就是为了让所有计算机用户都可以自由地获得这个系统及其源代码，并且可以相互自由复制。因此，在使用 GNU 软件时可以理直气壮地说自己使用的是“正版软件”。

在 GNU Manifesto（GNU 宣言）中对 GNU 的精神进行了阐述：软件的源代码应该自由流通，软件开发者应该做的不是把源代码据为己有，赚取发行可执行文件的金钱，而是应该赚取整合与服务的费用。因为源代码自由流通的软件才能让软件的质量提高，让软件开发人员可以自由地与他人交换心得，不受知识产权的约束。

为了保证 GNU Manifesto 精神的实施，GNU 制定了 GPL（The GNU General Public License，GNU 通用公共准许证），即先依照著作权法获得 GNU 软件的版权，再通过 GPL 释放此权力给所有使用者；只要用户遵守 GPL，不把源代码以及自己对源代码所作的修

PDF-XCHANGE Click to buy NOW! www.docu-track.com

改据为己有，就拥有使用 GPL 软件的权力。

1.1.1 热爱和享受自由

自由软件发展至今已经成为计算机界一个重要的组成部分。通过不断的发展，“自由软件”越来越显示出它所具有的强大力量，已成为计算机技术发展的重要推动力。

使用自由软件可以获得前所未有的自由，包括：使用的自由（可以不受任何限制来使用软件）、研究的自由（可以研究软件运作方式并使其适合个人需要）、散布的自由（可以自由地复制此软件并散布给他人）、改良的自由（可以自行改良软件并散布改良后的版本以使全体用户受益）。

作为自由软件代表之一的 Linux，已经在世界无数的服务器上运行，为公司、个人提供强大的计算服务。Linux 现在几乎可以胜任从桌面计算到企业服务的各个领域，成为强大、自由、免费的操作系统。

“自由”对于人们来说是非常重要的，越来越多的人意识到了这一点。自由软件也吸引了越来越多的个人和公司投人到自由软件的开发和维护中来。

自由软件为什会有如此大的魅力呢？

自由软件的适应性比商业软件大得多。自由软件和传统商业软件之间最显著的差异在于：自由软件鼓励复制，允许研究、改良。只要有人需要某种功能，就有人把这种功能加入到程序里。自由软件也往往具有很强的可移植性，可以非常容易地在不同的软硬件平台上使用。而商业软件在功能性和可移植性上都无法和自由软件相比，缺乏源代码也直接导致了用户对商业软件难以进行维护和改进。

自由软件的全部或部分能够被随意地利用、改进、再次发行。自由软件的使用者可以修改发现的 bug 或者加入自己需要的功能，或者移植给其他系统。自由软件通常总是在世界范围内拥有顶级的高手进行维护。在软件的使用过程中，如果报告一个 bug，很快就有人解决这个问题，提出一个新的修改。通过互联网的交流和沟通，对一些他人发现的 bug，感兴趣的程序员也可以通过对源代码的分析来解决问题。借助于互联网的交流，联合全世界无数对自由软件热爱的开发人员，自由软件的升级和更新通常比商业软件快得多。自由软件虽然是自由的，但自由软件中发现的严重 bug 和缺陷在很多时候比商业软件要少得多。

对于自由软件来说，使用者能够自己改进程序后，再次发行，这样软件的功能就会越来越强大，衍生出来的软件会使更多的人受益。比如，对于 Linux 操作系统，中国的程序员完全可以在其内核中加入中文处理能力，将 Linux 改造成为“纯粹”的中文自由操作系统。商业软件没有这个优点，商业软件的升级版本可能需要付费，或者根本就不会有使用者所需要的那种升级版本诞生。

自由软件具有强大的生命力，这是由它的开放性决定的。一个自由软件，哪怕只剩一个人喜欢，它都可以自己来维护这个程序的生存，适应自己的需要，说不定以后还会有更多的人对这个程序产生兴趣。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

自由软件有良好的社会作用。它的一切工作原理都是公开的，这体现了尊重科学、不为名利、信息公开、共同进步的良好风尚，这对于科学研究工作是非常重要的。它能够被随意地复制给需要它的人们去用，这体现了人们互相帮助的美德，一个理解自由软件思想的人会更加关爱社会，乐于助人，对于改善整个社会风气都有很大的好处。

1.1.2 神奇的 Linux

1991 年 8 月，一位来自芬兰赫尔辛基大学的年轻人 Linus Benedict Torvalds（李纳斯·托沃兹），对外发布了一套全新的操作系统：Linux。

Linus 是一名大学生，为了实习使用著名计算机科学家 Andrew S. Tanenbaum 教授开发的 Minix（一套功能简单、易学易懂的 Unix 操作系统，可以在 Intel 8086 上运行，后来也支持 Intel 80386，在一些 PC 机平台上非常流行），Linus 购买了一台 486 微机，但是他发现 Minix 的功能还很不完善，于是他决心自己写一个保护模式下的操作系统，这就是 Linux 的原型。

最开始的 Linux 是用汇编语言编写的，主要工作是用来处理 Intel 80386 保护模式，按照 Linus 本人的描述，刚开始的时候是这样的：

“最开始的确是一次痛苦的航行，但是我终于可以拥有自己的一些设备驱动程序了，并且排错也变得更容易了，我开始使用 C 语言来开发程序，这大大加快了开发速度，我开始担心我发的誓言：作一个比 Minix 更好的 Minix，我梦想有一天能在 Minix 下重新编译 GCC…… ”

“我花了两个月来进行基本的设置工作，直到我拥有了一个磁盘驱动程序（有很多错误，但碰巧能在我的机器上工作）和一个小小的文件系统，这就是我的第 0.01 版（大约是 1991 年 8 月下旬的事情），它并不完善，连软盘驱动器的驱动程序都没有，什么事情也做不了，但是我已经被它吸引住了，除非我能放弃使用 Minix，不然我不会停止改进它。”

上述文字引自：《乐者为王》（英文名为 Just For Fun）——自由软件 Linux 之父李纳斯·托沃兹自述，中国青年出版社 2001 年 7 月出版。

1991 年 10 月 5 日，Linus 发布了 Linux 的第一个“正式”版本：0.02 版，现在 Linux 可以运行 bash（GNU 的一个 Unix shell 程序）、GCC（GNU 的 C 编译器），它几乎还是什么事情都做不了，但是它被设计成一个“黑客”的操作系统，主要的注意力被集中在系统核心的开发工作上了，没有人去注意用户支持、文档工作、版本发布等其他东西。

提示：黑客是指计算机技术上的行家或热衷于解决问题、克服限制的人。黑客乐意解决和发现新问题，并乐意互相帮助。他们相信计算机中充满各种各样有趣的问题，并且这些问题一旦被自己解决，他们愿意让解决方法被所有人共享。他们感觉发现问题，解决问题，共享答案的过程是有趣的，并着迷地去做。他们将解决问题视为乐趣，相信技术应该是自由的，应该被所有人共享。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

黑客是对拥有高深技术和崇高理念的孜孜不倦的计算机爱好者、工作者的尊称。

最开始的 Linux 版本被放置到一个 FTP 服务器上供大家自由下载，FTP 服务器的管理员认为这是 Linus 的 Minix，因而就建了一个 Linux 目录来存放这些文件，于是 Linux 这个名字就传开了，如今已经成了这一个伟大的操作系统的约定名称。

Linus 在 USENT 打讨论区 comp.os.minix 首先发布下面这条消息：

“大家可曾渴望 Minix-1.1 会有这样美好的一天：每个人可以自己编写驱动程序，但是可能我们没有发现这样一个美妙的计划——可以自己修改操作系统以适应自己的需要？你是否对所有东西都在 Minix 上运行这一点感到沮丧？你是否没有找到一个业余时间可以干的好题目？下面这篇文章也许正是你所需要的：”

“如同我在一个月以前所提到的那样，我正在开发一个类似于 Minix 的基于 80386 的操作系统，它现在已经可以工作了（当然得看你怎么想），现在我将公布它的源代码，它是第 0.02 版本，但是可以运行 bash、GCC、gnu-make、gnu-sed、compress 等。”。

该文引自 Linus 在 USENET 讨论区 comp.os.minix 中发表 Linux 核心 0.01 时的说明。

这以后，这个“娃娃”操作系统就以两个星期出一次修正版本的速度迅速成长，在版本 0.03 之后 Linus 将版本号迅速提高到 0.10，这时候更多的人开始在这个系统上工作。在几次修正之后 Linus 将版本号提高到 0.95，这表明他希望这个系统迅速成为一个“正式”的操作系统，这时候是 1992 年，但是直到一年半之后，Linux 的系统核心版本仍然是 0.99p114，已经非常接近 1.0 了。

Linux 终于在 1994 年的 3 月 14 日发布了它的第一个正式版本 1.0 版，而 Linux 的讨论区也从原来的 comp.os.minix 中独立成为 alt.os.linux，后来又更名为 comp.os.linux。

这是 USENET 上有名的投票表决之一，有好几万用户参加了投票。后来由于使用者越来越多，讨论区也越来越拥挤又不得不再细分成 comp.os.linux.*。如今，comp.os.1inux.*已经有十几个讨论组了，这还不把专门为 Redhat Linux 和 Debian Linux 设的讨论组计算在内。comp.os.1inux.*讨论组也是 USENET 上最热闹的讨论组之一，每天都有数以万计的文章发表。

目前 Linux 已经是一个功能强大的操作系统了，在它上面可以运行无数的自由软件，借助 Linux 可以完成几乎所有其他商用操作系统才能完成的工作，这就是自由的 Linux。

如图 1-1 所示就是 www.linux.org 网站（Linux 官方网站）上的 Linux 吉祥物，一只可爱的小企鹅（起因是因为 Linus 是芬兰人，因而挑选企鹅作为吉祥物）。

图 1-1　Linux 的吉祥物

Linux 操作系统是一个类 Unix 操作系统，具有以下的特色：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

- 遵循 OSI、POSIX 等规范，具有良好的开放性与互操作性。
- 多用户和多任务支持。
- 具有方便的命令行、用户界面和优秀的用户图形界面。
- 设备独立性，内核具有高度适应能力。
- 丰富的网络功能。
- 可靠的系统安全。
- 良好的可移植性。
- 丰富的应用软件。
- 良好的开放型，可以免费获得源代码。

正因为 Linux 具有如此多的优点，因此越来越多的计算机爱好者投入到了学习 Linux 的行列中来。对于软件工程师而言，Linux 作为一个功能强大、源码开发的操作系统，是学习操作系统设计的最好教材。

目前，Linux 内核的最新版本是 2.6.0。Linux 2.6.0 的内核是一个完善的操作系统，内核，具有丰富的功能。但是 Linux 2.6.0 内核的源代码数量也是惊人的，有数百万行代码之巨。

注意：Linux 核心源代码可以在下面的站点下载:

http://www.kernel.org/pub/linux

http://www.de.kernel.org/pub/linux

如表 1-1 所示是不同版本的 Linux 核心文件数量和源代码数量。如图 1-2 所示和如图 l-3 所示是不同版本的 Linux 核心源代码比较图。

表 1-1　不同版本的 Linux 核心的文件数量和源代码行数

Lnux 内核版本号	文件数量（个）	源代码行数（行）
0.01	76	8413
0.12	99	15486
0.97	187	38928
1.00	487	165165
2.0.1	1643	686201
2.4.22	10302	468534

从表 1-1 和图 1-2、图 1-3 中可以看出，Linux 核心源代码的文件数量和代码的行数随着版本的增长而飞速地增加。对于 Linux 0.01 而言，文件数量是 76 个，所有源代码只有 8413 行。而 Linux 2.4.22 核心有文件 1 万个，源代码有 468 万行之多。

对于有接近 500 万行源文件的 Linux 2.4.22 核心而言，要学习它是非常困难的。困为在这个核心中功能太多了，使学习者难以把握操作系统最主要的精髓。

而 Linux 0.01 虽然是 Linux 的第一个发行版本，但是却基本具备了操作系统中最重要的组成部分。同时，Linux 0.01 只有将近 1 万行左右的代码，对于初学者而言，学习起来就非常简单了。因此，本书以 Linux 0.01 核心为背景，来介绍操作系统的设计。

图 1-2　不同版本的 Linux 核心源代码的文件数量

图 1-3　不同版本的 Linux 核心源代码的源代码行数

1.2　在乐趣中获得成功

Linus 编写的 Linux 是最著名的自由软件之一，如今已经发展成为能够与微软的 Windows 和 Unix 抗衡的自由软件操作系统。现在不仅仅是计算机爱好者在研究和使用 Linux，全世界几乎所有著名的软件公司（包括微软）都在研究 Linux。Linux 正在成为全世界计算机界的明星，而 Linus 也成为我们这个自由软件时代计算机爱好者们新的偶像。

如果说是商业利益的驱动成就了比尔·盖茨，那么 Linus 却是因为追求电脑“乐趣”而无意中获得成功。

1997 年，Linus 从芬兰移居到硅谷，出任 Transmeta 公司首席软件科学家。Linus 在《乐者为王》一书中，自述了他的成功之路。

1.2.1　11 岁开始编程序

Linus 的外公是赫尔辛基大学一位统计学教授。1981 年 Linus 11 岁时，外公抱回来一台新的 Commodore VIC-20 计算机。这种计算机不具备开发商业程序的条件，它能做的惟一事情就是用 BASIC 语言编程序。Linus 的计算机生涯，就是从这台计算机开始的。

通过使用 Commodore VIC-20 计算机，Linus 把键盘玩得很顺，然后 Linus 开始阅读电脑操作手册，并尝试将里面的示范程序（example program）输入进去。手册里有一些简单游戏的示范程序，如屏幕上会出现一个人横穿走过的图像，还可以稍作修改，以显出各种不同的背景颜色，这些程序都大大地激发了 Linus 的兴趣。

然后 Linus 开始自己写程序。Linus 编写的第一个程序与其他人编的没什么不同，屏幕上显示出一行又一行的 HELLO，直到厌烦而中止它。

这是 Linus 的第一步，也是许多人的最后一步。

但是，兴趣让 Linus 迈出了第二步和后续的无数多步。

Linus 的妹妹萨拉让 Linus 对这个程序作了修改，从而产生了这个程序的第二个版本，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

屏幕上显示的不再是“ HELLO”的字样，而是无休无止的“ SARA IS THE BEST”（“萨拉是最棒的”）。

在兴趣的驱动下，Linus 没有停下来，而是每写完一个程序，就再去编写下一个程序。Linus 用零花钱购买电脑杂志，从中找到新的程序设计的兴趣。就这样，一点一点地，Linus 逐渐走进了计算机的世界。

3 年过去了，当其他孩子在外面踢足球的时候，Linus 却觉得电脑更加有意思，机器本身就是一个自由的世界。计算机并没有想象中的那么复杂，Linus 在还是孩子的时候，就经常打开电脑的盖子自己动手修理。或许，计算机对于孩子而言，是一种很快乐的玩具。

今天的小孩通常一般不再自己拆卸组装电脑，也很少书写程序，更多的时间是使用电脑玩游戏。这样，电脑不能帮助孩子们发展智力。Linus 认为游戏并没有什么不好，Linus 最早编写的一些程序也是游戏，但 Linus 主要是为了编程，才创造出新的游戏。电脑游戏是生活的一部分，但是也可以把编写程序当作电脑游戏来玩，并且这种游戏也很好玩。

1.2.2 一种操作系统的诞生

在兴趣的驱使下，Linus 上大学后第 1 年并没有选择相应的专业，计算机成了他的主修课。那时在整个赫尔辛基大学，连 Linus 在内，希望主修电脑的瑞典学生只有两个。

每个人都会有一本改变其一生的书籍，把 Linus 推向生命高峰的书是 Andrew S. Tanenbaum 写的《操作系统：设计和实现》。在这本书中，阿姆斯特丹的大学教授 Tanenbaum 讨论了 Minix 操作系统，那是他为 Unix 撰写的教学辅助软件。读完此书，了解到 Unix 背后的理念以及那个强大、简洁、漂亮的操作系统所能做到的事情后，Linus 便决定弄一台机器来“玩玩 ”Unix。

赫尔辛基大学第一次拥有 Unix 是在 1990 年秋季开学的时候。那个强大的操作系统是美国贝尔实验室 20 世纪 60 年代发明的，然而它的开发却在别的地方。Unix 的独到之处在于“优美”。可能很难理解使用优美来形容一个计算机软件，但是当理解了 Unix 之后，就会发现 Unix 的思想和理念是简洁和优美的，是以后无数操作系统设计的典范。

1991 年，Linus 在赫尔辛基理工大学参加一个学术报告，演讲者理查德 • 斯多曼是自由软件的鼓吹者。这是 Linus 第一次见到的典型“黑客”形象，留着长发，蓄着长胡子。这年，Linus 用圣诞节和生日得到的钱，买了一台价值不菲的“组装”计算机，4MB 内存，33MHz 80386 CPU、14 英寸显示器。

Linus 购买的计算机本来运行的是 DOS 操作系统，但是 Linus 却不准备使用 DOS。因为，DOS 是一个太简单的操作系统，缺乏现代操作系统的基本要素。例如 DOS 不支持多任务和多用户。，

安装 Minix 并不简单，Linus 用 16 张软盘把 Minix 装入计算机，并且使用了将近一个月时间，通过把需要的软件下载、安装、调试，“玩”了很久，才使这个系统完全变成

PDF-XCHANGE Click to buy NOW! www.docu-track.com

了 Linus 需要的系统。

撰写 Minix 的 Andrew S. Tanenbaum 希望 Minix 是一个教学工具，因此 Minix 在许多设计方面并不是非常有利于真正的使用。许多人对 Minix 进行了改进，最著名的一个改进出自一位叫布鲁斯·伊文斯的澳大利亚人，他使用的是 Minix386。他的改进使 Minix 在 386 上运行起来更方便。

在 Linus 购买这台电脑之前，Linus 就一直在网上跟踪 Minix 的消息，所以从一开始 Linus 就使用它的升级版。但 Minix 有一些性能令 Linus 很不满意，其中最大的失望是终端仿真（terminal emulation）。仿真很重要，因为 Linus 只能依赖这个程序，才能让 Linus 家里的电脑登录到大学的电脑中。每当 Linus 拨电话接通大学的电脑，使用强大的 Unix 工作或仅仅是上网时，都得使用终端仿真程序。

于是 Linus 决定自己开始做一个项目，制作自己的终端仿真程序。Linus 不想在 Minix 下做这个项目，而是想在硬件水平上完成它。为了不基于 Minix 来开发这个终端仿真程序，Linus 不得不从 BIOS 开始，BIOS 是计算机启动的早期 ROM 编码，它可读软盘和硬盘，所以这次 Linus 是在软盘上操作。

80386 在刚启动的时候，是工作在“常规模式”的，为了充分利用 80386 的 32 bit 能力，Linus 只得让 CPU 进入“保护模式”。

为了完成这个“项目”，Linus 设计了两条独立的线程。一条线程从调制解调器读出数据，然后在显示器上显示。另一条线程从键盘上读出数据，然后写入调制解调器。这样就会在两条线程上运行两条管道。Linus 最早的试验程序是使用一个线程将字母 A 写到显示器上，另一个线程写字母 B。LAnus 把此编入程序，让其在一秒钟之内出现若干次。在定时器的帮助下，使这个程序这样不停地运转：显示器上先出现一连串的字母 A，然后突然之间，转变成一连串的字母 B，这表明 Linus 所做的任务转换是可行的。最终 Linus 能改变由一连串 A 和一连串 B 组成的两个线程，从而使数据一个读自调制解调器，再写入显示器；另一个读自键盘，再写入调制解调器。这使 Linus 终于有了自己的终端仿真程序。

每当 Linus 想读新闻，Linus 就运行自己的程序，把自己的软盘插进，重新启动机器，这样就能从大学的计算机里读到所需的新闻。倘若想要改进这个终端仿真程序，就得启动 Minix，使用 Minix 来对终端仿真程序进行修改。

Linux 操作系统就是这样诞生的。

现在，Linux 已经变成了具有强大吸引力的操作平台，全球大约已有数以千万计的计算机正在运行 Linux 系统。

1.3 GNU 的开发工具

GNU 拥有丰富的开发工具，例如：

- Emacs：功能强大的编辑环境。
- GCC：性能优异的多平台的 C/C++、Fortran 编译器。
- Kdevelop：KDE 集成开发工具。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

- Cygwin：Windows 下的 GNU 开发环境；'
- Tcl/Tk：功能强大的脚本语言。

同时，GNU 还拥有强大编译自动化工具 make、汇编程序 nasm、连接器 ld 等等。使用 GNU 丰富的开发工具，完全可以完成开发一个操作系统的所有工作。

1.3.1 编译器家族 GCC。

GCC 是 GNU 最为著名的跨平台编译器，通过它可以在大量的硬件平台和操作系统上编译程序。GCC 拥有惊人的可移植性，这使得很多商业公司也采用 GCC 来开发软件，HP 公司、苹果公司都使用 GCC 来开发程序。在 PC 游戏中最好的 3D 射击游戏 QUAKE（由 ID Software 公司开发）也是用 GCC 的 DOS 移植版本 DJGPP 编写的。

GCC 是 C、C++和 Objective C 三者有机结合的编译器，它的编译原理同大多数编译器不一样。它先由一个前端处理程序将 C、C+十和 Objective C 的语句转换成为一个类似于 Lisp 的内部语言——RTL，再由一个后端处理程序将其优化后产生目的 CPU 可以执行的机器代码，因而对于每种新语言来说，只要写好一个新的前端处理程序就可以立刻将此语言移植到 GCC 已经支持的不同硬件平台上去，而且编译出来的就已经是经过优化的二进制代码。除了 C 系列以外，GCC 还有 Fortran 77、Ada9x、Pascal 的前端处理程序。GCC 的多平台的实现方法同 Java 的虚拟机技术不太一样，因而其运行速度远非 Java 可比。

当使用 GCC 来进行 C 语言开发的时候，可以使用内联汇编语句，这样比较适合系统程序的开发。GCC 支持多种目标文件的格式，如 coff、elf 等，可以将 C 源程序方便地编译成多种目标文件。在 Windows 环境下，默认情况下产生 coff 文件格式，而在 Linux 环境下，默认产生出 elf 格式的文件。

GCC 是 GNU 中主要使用的编译器软件，是 GNU 能够持续发展的重要保证。因为有了 GCC，就不再需要其他的 Non-Free Software（非自由软件）来编译 / 生成自己了。本书也将利用自由的 GNU 软件来学习编写操作系统的技术。

GCC 可以说是 Richard Stallman 所创立的 GNU 计划中最重要的作品之一，它提供了自由软件世界高品质的编译器（Compiler），实现了在自由软件平台上开发程序的梦想。如果没有 GCC，恐怕今日也不会有这么多形形色色的自由软件可用。

早期 GCC 的发展方向是以 C/C++和 Objective C 等语言为主，故“ GCC”的涵义即为 GNU C Compiler。但发展到现在，GCC 的内涵已不只是 C 与类似 C 的程序语言而已了，它同时还包含了许多其他语言的编译器，如 GNU Ada Translatorgnat、Java（gcj）、Fortran 77（g77）、Modula-2、Chill、Pascal（gpc）等，有些发展已接近成熟，有些尚在发展中。这些编译器全部共享同一组程序编译最佳化的引擎，使用相同的浮点数运算模块，同时也都享有 GCC 高移植性的特色。因此，GCC 事实上包含了一个很大的编译器家族，而 GCC 的涵义也随之转变为 GNU Compiler Collection。

GCC 一个很大的特色是高度可移植性，目前已知有超过 30 种硬件平台与操作系统可以执行 GCC，其中硬件平台包括：x86、ia64、alpha、hppa、m68k、Power PC、mips、IBM rs6000、sparc/sparc64 等，而操作系统则从 Microsoft 平台（DOS/Win32）到 IBM OS/2 到各家的 Unix。如此强大的可移植性正是 GCC 广为流传散布的主要原因。在许多商业版的 Unix 系统中，如果没有特别购买专用编译器的话，通常可以选择安装 GCC 来使用。而且，尽管 GCC 是自由软件计划开发出来的，但其所编译出来的程序品质并不亚于商业版的编译器，甚至在某些平台上所编译出来程序具有更好的执行效能。

除了各种编译器以外，GCC 还内含了其他的工具程序以及各程序语言所需的函数库，像 C++（g++）所需的 libstdc++，以及 Java（gcj）所需的 class 函数库 libgci 等。而其工具程序中，最重要的就是 cpp，它是程序编译时期的前置处理器。通常一个程序在编译时的步骤如下：

宏前置处理→程序代码编译与最佳化→组译→链接函数库→可执行文件

而 cpp 的工作就是负责整个流程的第一个步骤。通常在编写程序时会定义许多宏程序代码，同时也会嵌入许多“引用文件”（header files，扩展名为.h），这些用以方便程序撰写的手法在编译时期都会交由 cpp 处理，在源代码全部展开后，才交由编译器进行实际的编译工作。当然，这些在执行 GCC 进行编译时都会自动进行。

1.3.2 和操作系统开发有关的工具

Linux 系统中带有大量的开发工具，从而可以有效地支持操作系统的开发。本节将重点介绍 make、虚拟机、汇编程序、链接程序等操作系统开发工具。通过学习这些工具，可以使读者提高在 Linux 下开发程序的能力。

1. 编译自动化工具 make

当程序比较复杂，包含很多文件时，往往需要相当复杂的编译动作才能将程序整个编译完成，并且安装到正确的位置上使用。在这种情况下，仅仅依靠手动执行每一个编译命令显然其效率是很低的。同时，由于一个大型程序的编译往往需要花很长的时间，如果每次对程序进行修改后（不论修改的幅度有多少）都要对整个程序进行编译，其效率也是相当低的。

解决这些问题，就需要使用 make 程序。make 的功能是将整个程序编译的工作自动化。当 nlake 运行时，它会读入当前工作目录下的指令文件 Makefile，并根据 Makefile 的指令一步步地执行。因此，使用 make 工具，可以将整个程序编译的细节写在 Makefile 中，当程序需要编译时就执行 make 将它编译完成。

不仅如此，还可以在 Makefile 中指明各编译产物与其来源文件之间的依赖关系，例如，要编译生成一个静态的函数库，其最终产物是一个.a 的函数库文件，而该函数库文件是由许多的目标文件所组成的，而这些目标文件又是由各自的源程序代码编译而来，如此，就有了一连串的依赖关系了。当在 Makefile 中指明了这些依赖关系后，如果对其

中的任何一个源程序进行了相应的修改后，则当执行 make 时，它只会去编译与该源程序文件有依赖关系的目标文件，如此一来该目标文件就等于修改过了，由于最后的函数库文件是依赖于所有的目标文件，故 make 最后会再重新产生一个函数库文件，如此即可编译完成。

换句话说，make 依据 Makefile 所指定的依赖关系，来判断编译过程中各中间产物是否需要重新编译，而判断的标准就是该中间产物的源文件是否被修改过，也就是源文件的最后修改日期是否较该中间产物要晚？如果晚，就表示该源文件已修改过了，故该中间产物需要重新编译，反之成立。因此，每当原始程序有小幅的修改时，它就只会去重新编译真正修改过的部分，而不会每次都从头开始编译，如此效率就可以大幅度提升了。

GNU 中的 make 程序，除了具备上述的基本特性外，同时还包含了许多 GNU 的延伸功能，例如各种宏的定义，条件判断式等，而有些延伸功能是其他版本的 make 所没有的。因此，有许多大型的自由软件，都会特别指明要 GNU make（或者还包括 GCC 及其他的 GNU 编译工具）才能进行编译，因为它们的 Makefile 中已隐含了 GNU make 许多特有的功能。

一个操作系统所包含的源代码通常是非常巨大的，使用 make 可以有效地管理这些源代码，提高操作系统开发的效率。

2. 虚拟机软件 Bochs

操作系统是运行在一定的硬件平台上的，为了开发操作系统，一个最直观的需求是必须要有一台计算机。但是，当在一台真正的计算机上开发操作系统时，面临的难题却是令人头疼的。因为，操作系统是计算机上运行的最底层软件，为了测试编写的操作系统是否可以运行正常，通常情况下必须重新启动计算机来运行编写的新操作系统。如果发现了错误，又必须重新引导原来的操作系统，再次改写程序，然后编译连接，然后再次引导新的操作系统，这样的反复操作是非常令人烦恼的。

因此，为了避免在测试操作系统时反复地重新启动计算机，通常开发操作系统都需要一个模拟器软件。Bochs 正是满足这样需求的一个计算机模拟器软件。所谓计算机模拟器软件就是指通过软件来完整地模拟一台物理计算机，从而可以运行直接基于物理计算机平台的其他软件（例如操作系统）。

Bochs 是 GNU 的 x86 系统的模拟器，可以运行在 Windows、Unix 和 Linux 平台上，提供了强大的计算机硬件模拟功能，可以极大地提高操作系统的开发效率。Bochs 可以“同时”运行多个操作系统，这里的“同时”和平常在计算机上多个操作系统“共存”是不一样的。通常，计算机上都可以同时共存多个操作系统，例如 DOS、、Windows 9x、Windows2000/XP、Linux 等是“共存”，而不是“同时运行”。而 Bochs 的特点是可以在一台计算机上让多个操作系统“同时运行”，每一个操作系统就像在单独、完整的计算机上运行的效果完全一样。

在开发操作系统的过程中使用 Bochs 模拟器，可以大大地提高开发效率。Bochs 运行于现有的操作系统之上，可以将其他的操作系统内核读入，在其内部模拟 x86 系统，从而调试、开发操作系统的内核。通过 Bochs，就可以直接在开发计算机上进行操作系

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

统的运行和调试，而不需每次都通过重新启动计算机来加载开发的操作系统内核。图 1-4 演示了在 Bochs 中运行 Linux 的效果。对于 Bochs 的使用，本书将单独进行专门的介绍。

图 1-4　在 Bochs 中运行 Linux 的效果

3. 汇编器 nasm

在众多的汇编器中，GNU 的 nasm 是一种优秀的汇编器。nasm 在 Linux 和 Windows/DOS 下都可以运行，是一种跨平台的汇编器。

nasm 非常适合操作系统的开发。如果使用其他的汇编器，在从实模式到保护模式的过渡过程中，因为需要处理 16 位与 32 位指令的过渡，一般要直接写二进制代码来进行。jmp 的步骤。而在 nasm 中，直接用 jmp dword 就可以轻松做到。可以使用 nasm -f coff a.asm 来产生 coff 格式的目标文件，-f 开关用于选择要产生的目标文件类型。nasm 也支持 elf、coff 等多种目标文件格式。nasm 功能强大，使用简单，是开发操作系统的理想汇编器。

4. 链接器 ld

在使用编译器产生 C 和汇编目标文件后，下一步是用链接器将它们连接成一个可执行文件（在操作系统开发中，很多时候要生成一种特殊的可执行文件格式，这种格式能够被系统正确地引导），GNU 的 ld 链接器可以很好地完成这些工作。

ld 是跨平台可移植的标准链接程序，在 Linux 下和 Window/DOS 下都可以运行。本书介绍的操作系统也是使用 ld 来完成链接的工作。

5. 调试器（gdb）

编写的程序通常会存在 bug。在 GNU 中提供了专门的调试工具 gdb、ddd。gdb、ddd 作为通用的调试器，可以调试 C/C++、汇编等语言编写的代码，为操作系统的调试提供了很大的便利。

1.4　本章小结

GNU 是一个伟大的事业。因为 GNU 的出现，才能够有机会深入地阅读世界最优秀软件地核心和精髓。

Linux 作为 GNU 中最优秀的软件之一，它的诞生是乐趣驱动的。乐趣可以创造一切。本书的目的也是让读者在乐趣中，使用自由的软件，学习操作系统设计的思想和技术。

自由软件提供了大量的开发工具，使用这些免费的开发工具可以方便地进行操作系统的开发。

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

第2章 操作系统设计入门

本章知识点：

- 操作系统介绍
- 操作系统的基本功能编写
- 操作系统的建议
- x86 虚拟机 Bochs 使用简介
- 使用 Bochs 运行一个操作系统
- 操作系统设计的基本准则

操作系统设计是计算机软件领域最具有挑战性的工作之一。通过设计自己的操作系统，可以对计算机的软硬件有一个彻底的理解和认识。操作系统设计课程是每一名计算机技术学习者的必修课程。

操作系统是一种非常复杂的软件。由于它是运行在“裸机”上的第一层软件，因此操作系统和计算机硬件是紧密结合在一起的。编写操作系统软件，不仅需要软件设计方面的理论，还需要对计算机硬件方面的理论和知识充分了解。因此，计算机操作系统的设计是一门非常具有挑战性的课程。

本章是计算机操作系统设计的一个入门性章节。通过本章的学习，读者可以快速地对操作系统设计建立一个全局性的整体认识和概念。本章将向读者介绍开发操作系统可能涉及的开发语言、开发工具、开发方法等方面的知识。

2.1 操作系统介绍

计算机系统是由硬件系统和软件系统组成的。计算机的硬件系统主要包括：CPU、存储器（主存和辅存）、I/O 系统等。计算机软件系统包括：系统软件（操作系统、支撑软件）、应用软件等。

如果对软件进行细分，主要有如下的分类。

- 系统软件：管理计算机系统本身的软件，例如操作系统。
- 支撑软件：支持其他软件的开发和维护的软件，如软件开发工具。
- 应用软件：提供给用户解决具体问题的软件，如文字处理软件、企业管理软件、游戏软件等。

计算机系统在运行时，各种类型的软件通常构成一个层次结构，如图 2-1 所示。在计算机中，操作系统是其他所有软件运行的基础。从用户角度来看，操作系统是用户与

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

计算机硬件系统之间的接口。从资源管理角度来看，操作系统是计算机资源的管理者，承担着 CPU 管理、存储器管理、I/O 设备管理、文件管理等工作。操作系统必须使用方便，简单有效，具有可扩充、开放的特性。

图 2-1　计算机系统中软硬件的层次结构

在设计操作系统时，可以使用的软件模型包括：

- 整体式系统
- 层次式系统
- 客户 / 服务器式系统
- 虚拟机系统

随着计算机的出现，操作系统都在不停地发展。这主要是计算机用户需要不断提高计算机资源利用率、更加方便地使用计算机的需要，也是计算机硬件的不断更新换代、计算机体系结构不断发展的结果。

从 1946 年诞生第一台电子计算机以来，计算机的每一代进化都以减少成本、缩小体积、降低功耗、增大容量和提高性能为目标，随着计算机硬件的发展，同时也加速了操作系统的形成和发展。操作系统的发展经历了以下阶段。

2.1.1　早期的操作系统

最初的计算机并没有操作系统，人们通过各种操作按钮来控制计算机。后来出现了汇编语言，操作人员通过有孔的纸带将程序输入电脑进行编译。这个时期，计算机只能由操作人员自己编写程序来运行，不利于设备、程序的共用。为了解决这种问题，就出现了操作系统，这样就很好地实现了程序的共用，以及对计算机硬件资源的管理。

随着计算技术和大规模集成电路的发展，微型计算机迅速发展起来。从 20 世纪 70 年代中期开始出现了计算机操作系统。1976 年，美国 DIGITAL RESEARCH 软件公司研制出 8 位的 CP/M 操作系统。这个系统允许用户通过控制台的键盘对系统进行控制和管理，其主要功能是对文件信息进行管理，以实现硬盘文件或其他设备文件的自动存取。此后出现的一些 8 位操作系统多采用 CP/M 结构。

2.1.2　DOS 操作系统

计算机操作系统的发展经历了两个阶段。第一个阶段为单用户、单任务的操作系统，CP/M 系统是最早出现的单用户、单任务磁盘操作系统之一。继 CP/M 操作系统之后，还出现了 C-DOS、M-DOS、TRS-DOS、S-DOS 和 MS-DOS 等磁盘操作系统。

其中值得一提的是 MS-DOS，它是在 IBM PC 及其兼容机上运行的操作系统，它起

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

源于 SCP86-DOS，是 1980 年基于 8086 微处理器而设计的单用户操作系统。后来，微软公司获得了该操作系统的专利权，经过修改后配备在 IBM PC 机上，并命名为 PC-DOS。1981 年，微软的 MS-DOS 1.0 版与 IBM 的 PC 面世，这是第一个实际应用的 16 位操作系统。1987 年，微软发布 MS-DOS 3.3 版本。它已经是非常成熟可靠的 DOS 版本，微软通过 MS-DOS 取得了个人操作系统的霸主地位。

DOS 曾经占领了个人电脑操作系统领域的大部分，全球绝大多数电脑上都能看到它的身影。由于 DOS 系统并不需要十分强劲的硬件系统来支持，所以从商业用户到家庭用户都能使用。D0s 的主要优点包括：

- 文件管理方便

DOS 采用了 FAT（文件分配表）来管理文件。所谓 FAT（文件分配表），就是管理文件的连接指令表，它用链表的形式将表示文件在磁盘上实际位置的点连起来。同时，DOS 也引进了 Unix 系统的目录树管理结构，这样很利于文件的管理。

- 外设支持良好

DOS 系统对外部设备也有较好的支持。DOS 对外设采取模块化管理，设计了设备驱动程序表，用户可以在 config.sys 文件中加载系统需要的设备驱动程序。

- 小巧灵活

DOS 系统的体积很小，其核心一直只有几百 KB。通常，启动 DOS 系统只需要一张软盘即可，DOS 的系统启动文件有 io.sys、msdos.sys 和 command.com 三个。

- 应用程序众多

能在 DOS 下运行的软件很多，各类工具软件是应有尽有。现在许多 Windows 下运行的软件都是从 DOS 版本发展过去的，如 Word、WPS 等，一些编程软件如 FoxPro 等也是由 DOS 版本的 FoxBase 进化而成的。

虽然 DOS 有不少的优点，但同时它也具有一些不足。DOS 是一个单用户、单任务的操作系统，只支持一个用户使用，并且一次只能运行一个程序，这和 Windows、Linux 等支持多用户、多任务的操作系统相比就比较逊色了。

DOS 采用的是字符操作界面，用户对电脑的操作一般是通过键盘输入命令来完成的。所以想要操作 DOS 就必须学习相应的命令。另外它的操作也不如图形界面来得直观，对 DOS 的学习还是比较费力的，这对家庭用户多少造成了一些困难。

DOS 对多媒体的支持也不尽人意，纯 DOS 系统基本上不支持多媒体。从 1981 年问世至今，DOS 经历了 7 次大的版本升级，从 DOS 1.0 版到现在的 DOS 7.0 版，不断地改进和完善。但是，DOS 系统的单用户、单任务、字符界面和 16 位的大格局没有变化，因此它对于内存的管理也局限在 640KB 的范围内。DOS 操作系统的这些限制使它缺乏现代操作系统的一些基本特征，例如多任务处理能力和高级内存管理机制。因此，DOS 很难成为现代主流操作系统。实际上，DOS 已经逐渐退出了历史舞台。

2.1.3 Unix 和 windoWS 操作系统两大阵营

计算机操作系统发展的第二个阶段是多用户、多道作业和分时系统，其典型代表有，Unix 和 Windows 操作系统。Unix 代表了操作系统设计和技术的最先进思想，分时的多用户、多任务、树形结构的文件系统以及重定向和管道是 Unix 的三大特点。而 Windows 则较多地使用了图形界面，为普通用户使用计算机带来了很大的方便。

Unix 操作系统，具有先进、稳定的特性，其功能非常适合企业计算领域。在需要强大而稳定性的计算环境中，Unix 比 Windows 具有更大的优势。

Windows 是 Microsoft 公司在 1985 年 11 月发布的第一代窗口式多任务系统，它使 PC 开始进入了所谓的图形用户界面时代。Windows 1.x 版和 2.x 版都因为种种原因，并不十分成功。1990 年，Microsoft 公司推出了 Windows 3.0，它的功能进一步加强，具有强大的内存管理，且提供了数量相当多的 Windows 应用软件，因此成为 Intel 386，Intel 486 微机新的操作系统标准。

1995 年，Microsoft 公司推出了 Windows 95。在此之前的 Windows 都是由 DOS 引导的，也就是说它们还不是一个完全独立的系统，而 Windows 95 是一个完全独立的系统，并在很多方面做了进一步的改进，还集成了网络功能和即插即用功能，是一个全新的 32 位操作系统。

从微软 1985 年推出 Windows 1.0 以来，Windows 系统从最初运行在 DOS 下的 Windows 3.x，到现在的 Windows 9x/Me/2000/NT/XP/2003，Windows 取得了很大的成功，应用非常广泛。

2.1.4 自由时代的宠儿 Linux

Linux 是目前全球最大的一个自由软件，它是一个可与 Unix 和 Windows 相媲美的免费操作系统。Linux 最初由芬兰人 Linus Torvalds 开发，其源程序在 Internet 网上公布以后，引起了全球计算机爱好者的开发热情，许多人下载该源程序并按自己的意愿完善某一方面的功能，再发回到网上，Linux 也因此被雕琢成为一个全球最稳定的、最有发展前景的操作系统。

从操作系统的发展上看，虽然现在 Linux 还不能完全取代 Unix 和 Windows，但是 Linux 的确已经成为一个稳定性、灵活性和易用性都非常好的免费操作系统软件，已经在操作系统领域牢牢地占据了自己的一席之地。在未来的发展中，从桌面应用到企业计算，Linux 在各个领域都可能成为主流操作系统。

2.2 操作系统的基本功能

现代操作系统都具有：并发、共享、虚拟、异步的特性，同时具有如下的五个基本

功能：存储器管理、处理器管理、设备管理、文件管理、用户接口。

- 文件管理

文件管理的中心问题就是实现“按名存取”。通过文件管理，可以实现操作系统中文件资源的有效存取，使用户可以简单方便地管理自己的文件资源。

- 处理器的管理

CPU 是计算机最重要的资源之一。在现在操作系统中都支持多任务并发执行，因此，对 CPU 资源进行有效的管理是操作系统一个非常重要的问题。

- I/O 管理

计算机上通常配置了众多的 I/O 设备，通过 I/O 管理功能可以对这些设备进行有效的管理，使计算机的各种设备有序地工作。

- 存储管理

存储管理就是管理计算机的存储器，主要是指管理计算机的内存。计算机的内存是比较宝贵的资源，需要进行合理的管理，才能有效地发挥计算机的性能。

- 用户接口

用户接口是用户使用操作系统的界面。用户接口通常有命令行界面和图形界面两种，命令行界面通常功能强大，而图形界面通常使用方便。两种界面各有优点，一般需要结合使用。

2.3 编写操作系统的建议

操作系统设计是计算机软件领域最具有挑战性的工作之一。通过自己设计操作系统，可以对计算机的硬件、软件有一个彻底的理解和认识。

操作系统是一种非常复杂的软件。由于它是运行在“裸机”上的第一层软件，因此操作系统和计算机硬件是紧密结合在一起的。所以，编写计算机操作系统，不仅需要软件设计方面的理论，还需要计算机硬件方面的理论和知识。因此，计算机操作系统的设计是一件非常具有挑战性的事情。

操作系统设计是一件非常复杂的工作。面对这件复杂的工作，初学者最困难的就是如何入手。许多计算机书籍在介绍操作系统时，都是大量地介绍操作系统设计方面的理论知识，而无法让读者对操作系统设计有一个最初的直观认识。

本节不是一个操作系统设计的理论课程，而是一个快速入门。开发计算机操作系统的确是一件不简单的工作。因此，对于每一个操作系统的开发者而言，必须对一些基本的计算机知识有所了解，下面是一些最基本的知识：

- 可以较好地使用 GNU/Linux 操作系统。
- 熟悉 C 语言和汇编语言。
- 理解 x86 处理器的保护模式、段、描述表、逻辑地址转换。

如果读者对上述知识还不太熟悉，本书将在后面的章节中，详细地对上述知识进行介绍。

在本书介绍的操作系统开发中需要使用一些软件开发工具，这些基本工具包括：

- 用 GNU/Linux 操作系统。
- Bochs x86 模拟器。
- GCC、nasm、ld 等开发工具。

上述软件都是自由软件，可以自由地使用。在本书配套光盘中，提供了本书使用到的所有软件和源代码。

本书将逐渐地向读者介绍编写计算机操作系统的各个方面的知识。学习本书，不仅可以对操作系统设计有一个完整的认识，同时对计算机硬件和软件也是一次极好的学习。

2.3.1 编程语言选择

在计算机系统启动的时候，计算机会装载启动设备上的 0 扇区（512 字节）到内存中。0 扇区在一般情况下就是启动扇区，启动扇区里面通常包含引导代码。引导代码的作用：

- 将处理器设定到最初的工作状态。
- 同时将操作系统引导到计算机的内存中。

通常，编写一个操作系统的第一个步骤是编写一段引导代码（这段引导代码通常是使用汇编语言编写）。使用汇编语言编写引导代码是有一点难度的工作，如果完全从头开始编写这段引导代码，通常这个工作可能花去最少几周的时间。

引导代码的编译完成对于开发一个操作系统而言，仅仅是完成了“万里长征第一步”。如果完成了引导代码后，仍然使用汇编语言来编写操作系统的其他部分，那么即便是一个最简单的操作系统，对于个人而言，也是太庞大了。因为，使用汇编语言编写程序是很低效的（低效不是指代码的运行效率低，而是指代码编写速度慢，程序可移植性差）。

因此，实际上多数操作系统的大部分代码都是使用高级语言来编写。当然，并不是每一种高级语言都适合编写操作系统，最适合编写操作系统的高级语言毋庸置疑就是 C 语言。实际上，现在被广泛使用的操作系统就是使用 C 语言来编写的，包括 Windows，Unix 和 Linux。

C 语言伴随着 Unix 而诞生，但是从某种意义上说，C 语言比 Unix 更为成功。C 语言在设计的时候就是定位成一种“低级的”高级语言，也就是说 C 语言除了具有高级语言所有必要的特性以外，同时具有低级语言的特性，例如可以直接操作硬件。

C 语言的这种“高低结合”的特性，使 C 语言既具有高级语言的高编程效率，同时也具有低级语言的可以直接操作硬件的特性。因此，C 语言是最适合编写操作系统的高级语言。

使用 C 语言编写操作系统的最大优点是具有很高的可移植性。因为 C 语言是和机器无关的，而汇编语言是和具体的硬件环境相关的。所以，C 语言编写的 Unix 操作系统具有很好的可移植性，几乎已经移植到了所有的硬件平台上（使用 C 语言编写的 Linux 也具有这样的特性，几乎已经移植到所有的硬件平台上了）。

使用 C 语言编写的操作系统与使用汇编语言编写的操作系统相比较而言，运行的效

率可能会略微低一些，但是比起它的优点而言，这个缺点是微不足道的。

2.3.2 编译器和链接器

使用 C 语言来编写程序，必须具有两种主要的工具：编译器（Compiler）和链接器（Linker）。编译器负责将 C 语言的源代码编译为目标文件（OBJ file），目标文件包含了 C 语言源代码翻译而成的机器码和数据，但是还不是完整的可执行文件。

当编译完成后，.c 文件就被翻译成为了.obj 文件。然后可以使用链接器来将多个.c 文件翻译成的.obj 文件链接在一起生成可执行文件。

许多 C 语言的产品具有集成开发环境，可以通过单击一下鼠标完成编译、链接工作，但是仍然是先进行编译、然后再进行链接。

现在有很多 C 语言编译产品，例如在 DOS 下有著名的 Turbo C、Borland C/C++、Quick C/C++、Watcom C/C++，在 Windows 下有 Visual C/C++、C++ Builder 等。这些 C 语言编译器都是曾经普遍使用的产品，但是在本书中，重点推荐 GNU C/C++的编译器 GCC 和链接器 ld，这是因为：

- GNU C/C++是完全免费和开放源代码。
- ld 支持生成大多数平台下的可执行文件格式。
- GCC 可以支持几乎所有的处理器。

在大多数操作系统上都具有了 GNU C/C++，例如 DOS 系统中的 GNU C/++被称作 DJGPP，Windows 环境中的 GNU C/C++被称为 Cygwin。在所有的 Linux 版本中，也全部都包含了 GNU C/C++。

通常情况下，不同操作系统下的 CNU C/C++只能编译链接生成在本地操作系统下运行的可执行程序，例如 DJGPP 只能编译生成在 DOS 环境中运行的可执行程序，Cygwin 只能编译生成在 Windows 环境中的可执行程序。

在 GNU C/C++中，除了编译器和链接器以外，还包含了一些其他的常用工具。例如，objdump 就是一个非常有用的工具。objdump 容许程序开发人员查看可执行文件内部的结构，并且可以反汇编可执行文件，这些特性在编写操作系统的引导代码时非常有用。

使用 GNU C/C++中的 ld 链接程序的一个缺点是无法生成某些特殊格式的可执行文件。但是，生成特殊格式的可执行文件在实际中很少使用，因为目前在各种不同的操作系统中都有了通用的可执行文件格式，例如在 Unix 下的 elf 格式和 Windows 环境中的 pe 格式都是最通用的可执行文件格式。

在编写操作系统时，通常大多数设计者都在自己设计的操作系统上使用 elf 可执行文件格式，因为 elf 文件格式是非常简单的。而 pe 文件格式可以带来更多的功能，但是同样 pe 文件格式也是非常复杂的。

在编写操作系统时，还有一种更简单的可执行文件格式，就是“ flat binary forrnat（直接二进制可执行文件格式）”。直接二进制可执行文件格式其实根本没有任何格式，它就是直接将二进制的机器代码写入可执行文件中。这种文件格式和 DOS 下面的.com 文件

格式是一样的，但是 DOS 下的.com 文件不能大于 64IKB，而直接二进制可执行文件格式可以大于 64KB。

2.3.3 运行时函数库

在编写操作系统的过程中，一个主要的部分就是编写运行时函数库（Run-Time Library，RTL）。运行时函数库就是在程序运行的时候可能会调用到的函数，例如常见的 lib。

编译器提供的 RTL 通常是和操作系统相关的。例如，C 语言的 RTL 通常提供了大量的函数用来编写可移植的应用程序，但是这些函数的内部实现却是和操作系统紧密联系在一起的。实际上，生产编译器的厂商通常针对不同的编译模式或者运行模式提供不同的 RTL，例如在 Windows 中，Microsoft Visual C++针对调试、多线程、创建 DLL 等不同的使用场合提供了不同的 RTL。而在 DOS 下，Borland C/C++针对不同的内存使用模式（例如小模式、中模式、大模式等）提供了 6 种不同的 RTL。

本书的学习目的是重新编写一个新的操作系统，因此在这个新的操作系统上执行的 RTL 也必须重新编写，否则使用针对其他操作系统的 RTL 编译生成的可执行程序在读者编写的操作系统上是无法运行的。

在设计自己的 RTL 时，通常考虑在自己的操作系统上实现标准的 C 函数库，例如 ISO C。因为，通常大多数的 C 程序都是使用 ISO C 编写的，如果在读者编写的操作系统上实现了 ISO C，那么就可以很方便地将其他操作系统上使用 ISO C 编写的程序移植到新的操作系统上（移植是源代码级别的移植，二进制可执行代码是不能移植的。因此，在新操作系统上执行程序前需要通过重新编译源代码获得新的二进制可执行文件）。

假如在新的操作系统上实现的是很特殊的 RTL,，而不是标准的 RTL，那么所有其他的程序都必须使用新的 RTL 来重新书写。记住，在操作系统上实现的 RTL 越标准，那么越容易将其他操作系统上的应用程序移植过来。

在实现操作系统上的 RTL 时，如果可以获得和自己准备编写的新操作系统类似的操作系统上的 RTL 源代码，那么对实现新的 RTE 具有巨大帮助。

在标准的 RTL 中，有一些功能是和操作系统紧密联系在一起的，例如 malloc()、fork()等。在标准的 RTL 中，也有许多的函数功能是操作系统无关的，例如在 C 语言中，string.h 中定义的大多数字符串处理函数都是可以直接在任意的操作系统上移植的。

在通常情况下，许多编译器里提供的函数是只能在某种操作系统上使用的，例如在 DOS 下，Borland C/C++中提供的 biso.h 中的函数容许应用程序访问 PC 机的 BOIS。而在非 DOS 操作系统中，通常系统工作在保护模式下，除非在系统中实现了 VM86 虚拟模式，否则这些 BOIS 函数调用是无法直接使用的。

在 bios.h 中的函数通常是对标准 C 函数库的扩展，因此无法在各种操作系统上移植。这些扩展的函数通常都会在名字前面有一个下划线；例如_biosdisk()。

在新编写的操作系统上，不一定要移植这些函数。但是，在编写操作系统时，可以

定义自己的扩展函数调用，当然按照惯例这些函数的名字前面还是应该添加一个下划线。

在标准 C 函数库中，一些函数是依赖于操作系统的核心的，例如，stdio.h 中的函数通常都依赖于操作系统的。一部分，其中 printf()就依赖于操作系统核心提供的输出功能。许多开放源代码的 C 语言函数库都实现了一个通用的 printf 引擎，使用这个通用的 printf 引擎可以实现包括 printf 在内的一系列 printf 函数族，例如 fprintf()、sprintf()等。

在自己操作系统上实现 printf()时，可以借鉴 printf 引擎来实现。。从而可以使用这种引擎来接收格式化的字符串，然后把翻译后的结果输出到显示设备上（或者是输出到一个字符串缓冲区内。）

如果在自己的操作系统上已经实现了标准的 RTL，这将非常有助于快速移植其他应用程序到自己的新系统中。但是，实现全部的标准 RTL 还是具有一点难度的。

一种折衷的办法是在编写操作系统时，可以精选一些实用的 RTL 函数来首先实现，这样有助于快速地实现操作系统原型。这些最基本的 RTL 可以在操作系统的核心中实现。这些函数实现了最基本、最常用的功能，例如 strcpy()和 malloc()，不仅可以有利于编写、移植应用程序，还可以用来改进操作系统本身。

2.3.4 使用汇编语言完成底层操作

作为一种最常用的编写操作系统的语言，C 语言也不是完美无缺的，例如，C 语言没有一种通用的方法来访问处理器的特殊功能。因此，为了在 C 语言中实现一些和硬件紧密联系的功能，就必须使用嵌入式汇编语言，或者将某些模块使用汇编语言编写，然后和其他使用 C 语言编写的程序部分连接在一起（这就是通常所说的混合语言编程方法）。

嵌入式汇编允许程序设计者将某些需要访问硬件，或者需要最高执行效率的 C 函数使用汇编语言来实现，同时在嵌入的汇编语言代码中还能用字节访问 C 语言的变量。嵌入式汇编语言极大地方便了使用 C 语言和汇编语言混合编程。现在的主流 C 编译器，通常都支持嵌入式汇编语言。

在不同的编译器中，实现嵌入式汇编语言的方法是不同的。在 GCC 中使用的嵌入式汇编语言使用的是 AT&T 的汇编语言语法，而在 Vistlal C++和 Borland C++中使用的是 Intel 的汇编语言语法。虽然 Visual C++和 Borland C++使用的 Intel 的汇编语言语法更为简单和使用广泛，但是在 GCC 中的嵌入式汇编语言却具有更为强大的功能。只有在 GCC 中，才能在嵌入式汇编语言中完全没有限制地使用 C 中的表达式和变量，而这一点 Visual C++和 Borland C++是无法做到的。

2.3.5 关于 C++的使用

是否使用 C++来编写操作系统是一个引起很多争论的问题。Linux 的编写者们仅仅使用 C 语言就实现了强大的 Linux 操作系统内核，而一些其他的操作系统却完全是使用

C++语言编写的。

本书主要是使用 C 语言来编写操作系统内核的。当然，使用 C++来编写内核也是可以的。但是，要熟练和有效地使用 C++，对于初学者而言是困难的，并且 C++的设计思想也是需要相当的培养和训练才能够掌握。所以，本书以 C 语言为工具来实现一个操作系统内核。

2.4 x86 虚拟机 Bochs 使用简介

Bochs 是一个非常优秀的 x86 PC 模拟器（x86 PC emulator）。使用 Bochs 可以在一台计算机上同时运行两个或多个操作系统。注意，Bochs 可以“同时”运行多个操作系统，这里的“同时”和平常在计算机上多个操作系统“共存”是不一样的。通常，计算机上都可以同时共存多个操作系统，例如 DOS、Windows 9X、Windows 2000/XP、Linux 等是“共存”而不是“同时运行”。而 Bochs 的特点是可以在一台计算机上让多个操作系统“同时运行”，每一个操作系统就像在一台单独、完整的计算机上运行的效果完全一样。

使用 Bochs，可以在一台计算机先安装 Windows 系统，然后在 Bochs 模拟的虚拟计算机上运行 Linux 系统，两个系统可以有不同的 IP 地址，可以同时使用网络通信。

Bochs 还有一个显著的特点：Bochs 支持 Windows 平台和 Linux 平台，而且完全是免费的。

除了 Bochs 外，还有一些商用的模拟器软件，例如 VMWare 和 Virtual PC。Bochs 的功能很强大，完全可以满足开发操作系统的需要。使用 VMWare 和 Virtual PC 也可以来开发操作系统，但是这些软件通常是收费的。

2.4.1 Bochs 简介

在计算机性能越来越高的今天，出现了“模拟器”的概念，不同类型的模拟器功能是不一样的。

对于像 DOSEmu 和 Wine 等模拟器软件，是操作系统 API 级别的模拟器。如图 2-2 所示，这些软件执行本地代码（就像 x86 代码运行在 x86 处理器上），然后转换这些执行程序所希望的操作系统接口 API（例如 Windows API）到当前运行的实际操作系统接口 API（例如 Linux API），从而实现在模拟器中运行目标操作系统的可执行程序。

这种类型的模拟器必须运行在 x86 平台上，因为这些模拟器提供的是 API 接口类型的模拟，而不是处理器指令集级别上的模拟。这种 API 级别的模拟，比起处理器指令级别的模拟，通常具有更高的效率和性能，因此也是在一台机器上运行两种操作系统的重要选择。

Bochs 不同于 DOSEmu 或者 Wine，如图 2-3 所示，Bochs 是一种纯粹的 x86 指令级模拟器，可以模拟 x86 处理器的每一条硬件指令，同时还包括一台计算机运行所必需的 BOIS 和硬件设备。Bochs 的运行过程原理很简单。Bochs 在执行时是一个大的循环，不

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

停地从执行文件中取出执行代码，然后解码（Decode），然后将解码后的代码执行。CPU 的一些基本组成部分——例如寄存器（Register），在 Bochs 中被模拟为一个数据结构。计算机的内存（Main memory）被模拟为 C 语言中的内存数组。输入和输出设备，例如键盘（keyboard）、定时器（timers）、PIC 等都是非常模块化的设计，通过将这些设备的 IRQ、I/O 地址空间和中断服务例程设置正确来插入到计算机中。

图 2-2　Wine/DOSEmu 软件的运行结构

图 2-3　Bochs 软件的运行结构

Bochs 是完整的模拟计算机硬件，而不是模拟操作系统 API。在 Windows 中运行 Boobs 的时候，需要模拟计算机的图形显示卡，可以选择多种显示卡来模拟，例如单色的 monochrome Hercules Graphics Adapter（大力神单色显示卡，HGA）。这种显示卡是实现比较简单的显示卡。Bochs 使用一个 x 窗口来代替计算机的显示器，显示器的视频缓冲区被模拟为存储器中的内存。模拟计算机的键盘等设备使用宿主计算机的键盘。

由于在 Bochs 中所有的指令都被模拟成为 C 语言代码，因此 Bochs 具有很好的可移植性，可以运行在几乎任何操作系统上去模拟几乎任何的目标硬件平台。Bochs 目前在主流的操作系统如 Unix、Linux 和 Windows 平台上都可以运行。

对于模拟器软件而言，性能是一个非常重要的问题。通常情况下，在模拟器中运行软件比在真正的计算机上运行软件要慢许多。不过 Bochs 的性能还是令人满意的，笔者在 Celeron 300A 上通过 Bochs 模拟 x86 处理器运行 Redhat Linux 6.2，运行的效果还不错。

2.4.2　下载 Bochs 软件

Bochs 是完全免费的软件，可以通过下面的地址下载源代码和可执行代码：

http://sourceforge.net/projects/bochs/

在网址 http://sourceforge.net/中有 Bochs 的源代码和可执行码两种文件下载，建议读者下载 Windows 平台或 Linux 平台的可执行码安装。

2.4.3　如何安装 Bochs。

Bochs 在本书出版时已经有 2.0.2 版本了，本书介绍的版本是最新的 Bochs 2.0.2 版本。首先可以在下面的网址下载 Bochs 的 Windows 平台安装程序：

http://sourceforge.net/projects/showfiles.php?group_id=12580

注意：如果上述地址无法下载，请试一下这个地址：http://sourceforge.net/projects/bochs/。

下载的程序为：bochs-2.0.2，win32-bin.zip。将 bochs-2.0.2，win32-bin.zip 解压缩到 C 盘。解压缩完成后，可以看到在 C 盘上建立了一个新目录：C:\bochs-2.0.2。

Bochs 使用 bochsrc.txt 来配置虚拟机，在 Bochs 2.0.2 版中使用了环境变量 $BX-SHARE 来指示 Bochs 的安装目录。因此，在运行 Bochs 之前，必须设置环境变量 $BX-SHARE。

Bochs 2.0.2 版本中带有一个很小的 Linux 版本 DLX Linux，里面可以提供一些最基本的 Linux 命令，可以用来检测 Bochs 的模拟能力。

注意：DLX Linux 的官方站点是 http://www.wu-wien.ac.at/usr/h93/h9301726/dlx.html，可以在这个站点上下载到 DLX Linux 的最新版本。

在 C:\bochs-2.0.2\dlxlinux\目录中有一个文件，名字为 start.bat，这个文件是使用 Bochs 模拟 DLX Linux 运行的启动文件，文件内容为：

..\bochs

为了正确运行 DLX Linux，将 start.bat 的内容修改为：

SET BX-SHARE=c:\bochs-2.0.2

..\bochs

然后再运行 start.bat，就会出现如图 2-4 所示的窗口。

图 2-4　Bochs 的运行界面

当出现图 2-4 的运行窗口后，直接按下 Enter 键，就可以出现 DLX Linux 的运行界面。如图 2-5 所示。

图 2-5　DLX Linux 的运行界面

等待片刻，就会出 DLX Linux 的登录界面，如图 2-6 所示。

图 2-6　DLX Linux 的登录界面

在图 2-6 中，键人 root 后，可以登录到 Linux 系统中，如图 2-7 所示。

图 2-7　登录到 Linux 系统

使用 Bochs 进行二进制级别的调试

在操作系统的开发中，调试是一件比较困难的事情。因为在开发的初期无法移植一些知名的调试软件，调试一般只能依靠二进制级的工具来进行。前面提到的模拟器 Bochs，除了具有硬件模拟功能以外，还具有一定的调试支持功能。在 Boobs 目录下，BOCHSDBG.EXE 即为集成了调试功能的 Bochs 可执行文件。

修改 c:\bochs-2.0.2\dlxlinux\start.bat 文件的内容为：

SET BXSHARE=c:\bochs-2.0.2

..\bochsdbg

然后运行 start.bat，可以出现如图 2-8 所示的窗口。

在出现图 2-8 所示的窗口后，就可以输入一些诸如断点设置，汇编 / 反汇编，查看 CPU 状态等调试命令。

图 2-8　Bochs 的调试功能

如表 2-1 所示是用 Bochs 进行调试的一些常用命令，熟练掌握这些命令，对调试系统是非常有帮助的。

表 2-1　Bochs 的调试命令

命令	说明
c	继续执行（contiue）
step [count]	单步执行，count 为步数，默认为 1（execute count instructions）
Ctrl+C	停止运行
quit	退出 Bochs
vbreak seg:off	虚拟地址断点，seg 为段地址，off 为偏移，用于在指定的虚拟地址处设置断点（virtual address break）
pbreak addr	物理地址断点，addr 为物理地址。用于在指定的物理地址处设置断点（physical address break）
info break	显示已经设置的断点情况
delete n	删除断点（Delete a breakpoint）
x/nuf addr	查看在线性地址处的内存单元内容（Examine memory at line address addr），其中，n 表示要显示的单元数，u 表示单元大小，f 表示打印格式
xp/nuf addr	查看在物理地址处的内存单元内容（Examine memory at physical address addr），其中，nuf 的意义同上
setpmem addr data-size val	设置在物理内存地址 addr 上的内存单元内容。Datasize 为要设置的单元大小，val 为要设置的值
info register	打印出 CPU 寄存器的当前值

set $reg=val	将一个寄存器赋值，其中 reg 可以换为 eax 等寄存器，如 set $eax=0xA
dump_cpu	将 CPU 的整个状态全部打印出来
disassemble start end	反汇编指定地址的程序。Start 为开始的线性地址，end 为结束的线性地址

2.5 使用 Bochs 运行一个操作系统

本节将介绍在 Bochs 中运行一个操作系统的具体步骤，从而帮助读者快速地掌握 Bochs 的使用。

首先在 Bochs 中建立一个新的目录 c:\bochs-2.0.2\floppy。

其次，在目录 c:\bochs-2.0.2\floppy 中建立一个新的 Bochs 配置文件 bochsrc.txt，通过 bochsrc.txt 文件可以让 Bochs 知道如何启动操作系统，以及 Bochs 模拟的虚拟机的 BIOS 配置和显示设备配置。

下面是一个最简单的 bochsrc.txt 配置文件：

```
#how much memory the ernuiated machine 讯 11 have
megs：4

#~ilename ofROMimages
romirrmge：file=$BXSHARE／BIOS—bochs—latest，address=O：d0000
vgaromimage：$BXSHARE／VGABIOS—elpin 一 2，40
#what diskimageswill be used
floppya：1—44=boot，img，status=inserted
#Choose the boot diSk
boot：a
螗 where dowe sendbgmessages?
log：bochsout，Lxt　　一
```

注意：在上面的 bochsrc.txt 文件中，任何以#开始的行都是注释。在上面的配置文件中，仅仅使用了最简单的配置选项，如果需要了解其他更为丰富的选项的情况，可以参看 Bochs 的帮助文件。

接下来，将介绍如何制作 Windows 98 的软盘镜像文件。

2.5.1 制作一个软盘镜像文件

首先，格式化一张引导软盘，如果使用 Windows 98 操作系统，那么可以使用如下的方法来格式化一张可以引导的软盘。

在计算机的软盘驱动器中放入一张软盘，然后启动一个 DOS 窗口，在 DOS 窗口中

键入 format a: /s 命令，单击 Enter 键后，可以出现如图 2-9 所示的格式化界面，等待软件格式化完成，就制作了一张可以引导的软盘了。

图 2-9　在 Windows98 中格式化一张软盘（带启动功能）

当完成制作引导软盘后，接下来就是在这张引导软盘上复制引导镜像文件（Image 文件）。引导镜像文件就是完整地记录了一张软盘所有内容的磁盘镜像文件，类似曾经在 DOS 下广泛使用的 HD-COPY 制作的磁盘镜像文件。

但是，在 Bochs 中，制作镜像文件不使用 HD-COPY，而是有其他的方法。制作镜像的方法一共有 3 种：

- 使用 WinImage（共享软件，有 30 天试用期，可以在 http://www.winimage.com/ 下载）。
- PartCopy（自由软件，可以在 http://www.execpc.corn/~geezer/johnfine/index.htm#zero 下载）。
- RawWrite（自由软件，可以在 http://uranus.it.swin.edu.au/~jn/linux/rawwrite.htm 下载）。

1. 使用 PartCopy

PartCopy 的含义是部分复制。使用 PartCopy 可以完成软盘或硬盘的备份和恢复功能，也可以完成改写磁盘引导扇区等功能。

使用 PartCopy 可以轻松地完成对硬盘引导扇区的备份功能，具体的步骤如下：

(1) 生成一张引导软盘。

(2) 复制一些常用的工具软件到这张软盘上，例如 scandisk、format，fdisk、sys 等。

(3) 复制 partcopy.exe 到这张软盘上。

(4) 使用这张软盘引导系统，然后使用 PARTCOPY 备份硬盘的引导扇区，命令如下：

```
partcopy -h0 0 80000 A:disk_0.bak
```

(5) 等待上述命令执行完毕后，A 盘上 disk_0.bak 文件（大小为 514288 字节）就包含了硬盘的引导扇区的内容。将这张软盘放置在安全的地方。

(6) 如果硬盘引导扇区出现了问题，可以使用最近备份的 disk_0.bak 来进行恢复，使用如下的命令：

```
Partcopy A:disk_0.bak 0 200 -h0
```

下面介绍一下 partcopy 的常用参数：

```
partcopy source source_offset length destination {destination_offset}
```

在上述参数中，只有 destination_offset 是可选的，其它参数都是必选的。其中，在 PartCopy 中所有的偏移量和长度都使用十六进制表示。

假如 destination 是一个文件，并且没有指定 destination_offset 参数，那么目标文件将

被删除，然后创建一个新文件。如果 destination 不是一个文件，那么 destination_offset 默认是 0。

下面这些规则用来指定 source 和 destination：

- 任何不以“-”开始的名字都被认为是一个文中名，使用 DOS 的 21H 服务来进行读取。
- -aL 使用 int25 and int 26（Absolute Disk）来访问驱动器 L。
- -hN 使用 int 13 服受来来访问第 n 个硬盘。
- -fN 使用 int 13 服务来访问第 n 个软盘。

在本节中，如果希望使用 PartCopy 来生成一个引导软盘的镜像文件，可以使用如下的命令（注意在当前目录下有 partcopy.exe 文件，并且制作的引导软盘已经插入了 A 驱动器）：

partcopy -f0 0 168000 boot.img

上述命令将在当前目录下生成一个名字为 boot.img 的文件，这个文件中包含了引导软盘的全部信息。其中，生成的 boot.img 文件的大小为 1474560 字节。

2. 使用 WinImage

在 http://www.winimage.com/可以下载 WinImage 的最新版本 6.1 版本。下载的文件名为 winima61.zip。WinImage 不需要安装，只需要解压缩在一个目录下就可以使用了。假如 WinImage 被解压缩在 c:\winimage 目录下，然后执行 winimage.exe 文件，出现如图 2-10 所示的运行界面，请确认 Disk 菜单中选中了 Use drive A:。

然后执行“Disk”/“Read Disk”命令，就可以出现如图 2-11 所示的运行界面，现在 WinImage 开始读取 A 驱动器中软盘的内容了。

图 2-10　WinImage 的运行界面

图 2-11　读取一个软盘的内容

当 Reading Disk 完成后，可以将软盘的内容保存在一个文件中，执行“File”/“Save as”命令将软盘的内容保存为 boot.img 文件。其中，生成的 boot.img 文件的大小为 1474560 字节。

3. 使用 RawWrite

在 http://uranus.it.swin.edu.au/~jn/linux/rawwrite.htm 下载 RawWrite 的最新版本 0.7 版。下载下来的文件名为 rawwritewin-0.7.zip。将 rawwritewin-0.7.zip 解压缩到一个目录下就可以使用了。假如将 rawwritewin-0.7.zip 解压缩到 c:\rawwrite 目录下，然后执行

rawwritewin.exe 文件，可以出现如图 2-12 所示的运行界面。

图 2-12　RawWrite 的运行界面

在图 2-12 中，单击“ Read”标签，在“ Read”选项卡的 Image file 文本框中输入：boot.img，然后单击“ Read”按钮就可以开始读取软盘的内容了。等待一会儿，读取完成，将在 rawwrite 的目录下生成 boot.img 文件。其中，生成的 boot.img 文件的大小为 1474560 字节。

2.5.2　使用 Bochs 运行操作系统

将文件 bochsrc.txt 和 boot.img 复制到 c:\bochs-2.0.2\floppy 目录下，然后在 c:\bochs-2.0.2\floppy 目录下使用如图 2-13 所示的 DOS 命令生成 start.bat 文件。

注意：在图 2-13 中“ ^Z 代表快捷键 Ctrl+Z。图 2-13 是在 DOS 下生成一个文本文件的最简单方法。当然，也可以使用任何文本编辑软件在 c:\bochs-2.0.2\floppy 目录下生成文件 start.bat，只是内容必须是：

```
SET BXSHARE=c:\bochs-2.0.2
..\bochs
```

图 2-13　修改 start.bat 文件的内容

然后运行 start.bat 文件，就会看到 Bochs 模拟启动 Windows 98。的界面，如图 2-14 所示。

图 2-14　使用 Bochs 启动 Windows 98

2.6　操作系统设计的基本准则

通过软件可以创造出美的东西。软件的美在于它的功能，在于它的内部结构。对用户来说，通过直观、简单的界面呈现出具有恰当特性的程序就是美的。对软件设计者来说，被简单、直观地分割，并具有最小内部耦合的软件结构就是美的。

本书当然希望读者创造的操作系统是美的。读者如何学到创造美的操作系统呢？

在本书中，讲授了一些设计操作系统的基本原则，这些原则可以帮助读者掌握如何设计一个优美的操作系统。当然，设计一个优美的操作系统，绝非如此简单。希望读者

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

通过对本书的学习向这个方面迈出坚实的一步。

2.6.1 简单就是优美

软件的本质即“简单就是美”。Unix 设计的思想就是，让每个程序都只擅长于一项专门的工作，然后让它们合作，形成一个可靠的、强大的、灵活的系统。这就是“简单就是美”。

事实证明，Unix 系统是设计良好的，其“简单就是美”的设计思想是后来操作系统设计的基本原则之一。

2.6.2 利用已有的基础

学习操作系统设计最关键的观念是不要从一开始就去关注整个系统的设计和实现。实际上，操作系统的设计和实现是一件具有巨大乐趣的事情，其中每件工作都是具有很大挑战的。因此，在一开始的时候，建议读者从简单的模块入手，逐渐地把操作系统的主要部分都练习到。

设计一个操作系统，如果从没有任何基础的裸机开始，那么可能读者需要作的第一个试验是在计算机的屏幕上可以画出一个点。即便是实现这样一个简单的功能，也通常需要使用汇编语言编写大量的代码。这些工作，对于一个初学者而言，还是有相当大的难度的。

还好，对于上述的工作，实际上是比较通用的。在本书的例子中，已经给出了这样的实例。读者可以从这些实例中学习到这些基本的技术，避免在黑暗中孤独的摸索。

操作系统的引导代码必须使用汇编语言来编写。除了个别非常喜欢汇编语言、而且有大量时间去编写汇编语言代码的人之外，对于大多数读者而言，花费大量的时间去书写这些引导代码的意义是不大的。因此，建议读者使用本书例程中提供的引导代码来继续自己的工作。

2.6.3 良好的设计

在开始编写操作系统时，需要首先把一些设计思想明确下来；否则，可能在系统设计的工作中出现无法弥补的失误，可能会发现：做了很久的系统，结果从一开始就是错误的。

首先需要确定内存管理模式：使用分段管理模式还是分页管理模式。当代操作系统大多数趋向于支持分页管理方式，因为分页管理方式比分段管理方式更易于移植。

另一个问题是将核心放置在内存的什么地址？是为每个进程设置独立的虚拟地址空间（类似 Unix 系统），还是将所有的进程都混合放置在同一个地址空间中（类似 DOS 或者 VxWorks 系统）。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

还有一些重要的问题是：

- 使用虚拟存储管理吗？
- 系统支持多任务 / 多线程吗？如果系统中支持多任务或者多线程，那么对于系统中的进程调度或管理就需要非常细致的设计。
- 进程使用什么模型？进程中是否包含线程？什么时候进行进程切换？
- 如何实现系统调用？是否实现自己独有的系统调用或者是与其他的系统兼容？
- 支持进程间通信吗？使用命名管道、队列、socket？
- 如何实现文件系统？采用标准的 open()、close()、read()、wrire()、ioctl()接口吗？

以上问题都得事先明确。

2.6.4 单内核操作系统和微内核操作系统

现代操作系统设计中，主要有两种体系结构：单内核结构和微内核结构。两种体系结构各有优缺点，在本节将进行简单的分析。

如图 2-15 和图 2-16 所示分别是单内核操作系统和微内核操作系统与计算机硬件和软件之间的交互关系演示图。

图 2-15　单内核操作系统

图 2-16　微内核操作系统

设计微内核操作系统，需要对微内核进行良好的设计和实现。本书介绍的 Linux 0.01 是一种典型的单内核操作系统

2.7 本章小结

本章是一个操作系统设计的入门章节，介绍了操作系统的基本概念和主要功能，还介绍了操作系统发展的主要历程。

接下来，本章介绍了操作系统设计中需要使用的主要工具，其中着重介绍了模拟机软件 Bochs 的使用方法。

同时介绍了设计操作系统的基本准则：简单就是美、良好的设计、单内核和微内核操作系统的区别。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

第3章 操作系统设计基础

本章知识点：

- 使用 DJGPP
- 保护模式汇编语言
- 一些编程实例

本章主要介绍操作系统设计中使用到的主要编程技术。包括 DJGPP 和 RHIDE 的安装和使用方法。汇编语言在编写操作系统时，具有非常重要的作用。本章介绍了如何在保护模式下使用汇编语言。

本章最后介绍了一些编程实例，这些实例都是操作系统设计中经常使用到的，并且这些实例可以直接用在读者设计的操作系统中。

3.1 使用 DJGPP

本书的很多源程序都使用了 DJGPP 来编译。DJGPP 是 DOS 下的自由源代码的保护模式 C 语言编译器。提起 DOS 下的 C 语言，大多数朋友都会想到 Borland C/C++。Borland C/C++使用方便，但是不支持保护模式开发，因此在编写大型程序时，如果需要使用超过 640KB 的内存，就会面临很大的问题。

DJGPP 是一个支持保护模式编程的编译器，可以轻松地使用 4GB 的内存，具有 Borland C/C++不可比拟的优点。

DJGPP 还有一个集成开发环境：RHIDE。RHIDE 完全按照 Borland C++ 3.1 的界面仿制而成，内置集成调试器和在线帮助，使用方便，是 DOS 下编写保护程序的优秀工具。

3.1.1 DJGPP 和 RHIDE 的安装与使用

DJGPP 是开放源代码软件，可以自由地下载使用，下载地址是：

http://www.delorie.com/djgpp

目前 DJGPP 的最新版本是 2.04，在本书附带的光盘中，有 DJGPP 2.04 的完整版本。

1. 安装 DJGPP 和 RHIDE

在本书的光盘中，有 DJGPP 和 RHIDE 的完整版本，两个软件都已经压缩为一个压缩包，名字为：djgpp.zip。读者只要把 djgpp.zip 解压缩到 c:\目录就可以了。

假设解压缩后的 DJGPP 被存放在 C:\DJGPP 目录，下面设置运行 DJGPP 和 RHIDE 的环境变量。在 DOS 的 C 盘根目录下的文件 autoexec.bat 中添加如下两行：

set DJGPP=C:\DJGPP\GJGPP.ENV

setP ATH= C:\DJGPP\BIN;%PATH%

DJGPP 的早期版本不能在 Windows 2000 和 Windows XP 中运行，本书附带的 DJGPP 2.04 已经可以顺利地在 Windows 2000 和 Windows XP 中运行了。

设置了上述的环境变量后，重新启动计算机，然后运行 RHIDE，如果出现了图 3-1 所示的界面，表示 DJGPP 和 RHIDE 已经安装成功。

图 3-1　RHIDE 运行的效果

2.　使用 DJGPP 和 RHIDE

DJGPP 是 DOS 下面的 GNU 编译器，RHIDE 是 DJGPF 的图形界面。RHIDE 的开发界面和 Borland C/C++ 3.1 非常类似。如果读者使用过 Turbo C++或者 Borland C/C++，那么使用 RHIDE 就不会有什么困难。

3.1.2　　make 的使用

在本小节将介绍 GNU make 的使用，在本书的第 1 章中已经简单介绍了 make 的功能，使用 make 可以大大地提高操作系统开发的效率。也许有读者要问，已经有了 RHIDE 的 IDE，IDE 中有工程管理功能，为什么还需要 make 呢？

其实，不同 IDE 的工程管理方法是不一样的。因此，如果程序开发过程中换用了不同的 IDE，那么就必须重新设置工程管理。

而 make 是在 Unix、Linux，Windows、DOS 等几乎所有平台上都通用的工程管理工具，具有普遍的适用性。因此，本小节将介绍 make 的使用方法。

1.　make 基本使用规则

make 在使用时可以根据一系列预先设定的规则来运行。这些设定的规则可以记录在一个文件中，默认该文件的名字是 Makefile。

make 的使用形式为：

make [option] [macrodef] [target]

option 指 make 的工作行为，make 的主要选项有：

-c dir：make 在开始运行后的工作目录为指定目录

-f filename：使用指定的文件作 Makefile

在 make 运行时，会向屏幕输出一些信息。为了记录这些信息，可以使用如下命令：

make file > errofile

这样错误信息就都写入了 errofile，可以使用编辑软件查看。

2. Makefile 的基本书写规则

make 运行时，主要根据规则文件中记录的规则来判断是否对文件进行更新。例如，工程 newos 依赖于 main.o，f1.o，f2.o，而 main.o 依赖于 main.c 和 main.h，f1.o 依赖于 f1.c 和 f1.h，f2.o 依赖于 f2.c，如图 3-2 所示。

图 3-2　newos 工程的依赖关系

在图 3-2 所示的关系树中，双亲节点依赖于孩子节点。当使用 make 维护 newos 工程的时候，make 根据这个关系树，如果发现孩子节点形成的时间晚于双亲节点形成的时间，便会编译更新相应的文件。

在 make 中，记录上述树型关系的文件是按一定规则书写的。这个规则文件的组成是以一个要维护的目标为一个单元，每个单元的书写形式如下：

要维护的刚示列表　要维护的副示属性　分割符（一般为:）依赖的文件列表　命令行的属性; 第一条命令

tab 键　命令 1……

tab 键　命令 2……

tab 键　命令 3……

需要说明的是，第一条命令可以加“;”跟在依赖文件列表后，也可以和其他命令一样另起一行写。但是，当命令另起一行书写的时候，一定要使用 tab 键，否则 make 不认识。当一条命令太长，一行放不下时，可以使用“\”来进行续行。在 make 的配置文件中，可以在一行开头使用“#”来进行注释。

下面是 newos 工程的 Makfile：

```
newos:    main.o f1.o f2.o
gcc -o example main.o f1.o f2.o
man.o:    main.c main.h
gcc -c main.c
f1.o: f1.c f1.h
gcc -c f1.c
f2.o: f2.c
gcc -c f2.c
```

可见，书写一个简单的 Makefile 文件并不困难。除了上述简单的功能外，make 还有大量高级功能，例如属性变量、宏变量、流程控制等。下面，将对 make 的宏进行简单介绍。

3. 宏的使用

PDF-XCHANGE Click to buy NOW! www.docu-track.com

在 make 中宏的命名可以是任意数字、字母和下划线的组合，不过不能用数字开头。make 中宏的定义方式有 3 种：

- =：直接将后面的字符串赋给宏。
- =：后面跟字符串常量，将它的值赋给宏。
- +=：宏原来的值加上一个空格，将它的内容赋给宏。

宏的引用格式有两种：$(宏名)或$|宏名|。宏名也可以嵌套使用，如：

```
name2=uestc
index=2
```

在 Makefile 中调用$(name$(index))，就等于调用 uestc。在 make 中也可以使用 shell 环境的宏，不需要重新定义，只要用 IMPORT 就行了。

GNU make 是一件强大的工具，关于 make 的高级用法，读者可以参考 make 的相关手册，获得更多的帮助。

3.1.3 ld 的使用

ld 是 GNU 的链接器，支持生成大量的可执行文件格式。本小节将介绍 ld 的使用方法。使用如下的命令可以显示 ld 的帮助信息：

```
ld -help
```

ld 拥有大量灵活的参数来控制其链接过程，下面将介绍一些主要参数的使用方法。

ld 可以使用链接脚本，如果希望使用 krnl.ld 文件作为链接脚本，可以使用如下的命令：

```
ld -T krnl.ld …
```

如果希望 ld 生成一个二进制文件（像 DOS 的 COM 文件一样），并且希望代码从地址 100H 开始执行，可以使用如下的命令：

```
ld --oformat binary -Ttext=0x100 -o krnl.com ...
```

如果希望使用 ld 把一系列的.o 文件连接到另外一个.o 文件中，可以使用如下的命令：

```
ld -r -o krnl.com ...
```

如果希望 ld 生成的程序中包含调试信息，可以使用如下的命令：

```
ld -s …
```

如果希望使用 ld 生成一个 Linux 中使用的共享库，可以使用如下的命令：

```
ld -shared …
```

3.1.4 nasm 的使用

nasm 是 Linux 中语法与 DOS 最为相像的一种汇编工具，nasm 也是使用 Intel 风格的汇编语言语法，其语法与 masm 非常类似。这对于熟悉 DOS 环境汇编语言编程的用户迁移到 Linux 环境中编写汇编代码非常有利。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

nasm 的使用方法如下：

nasm -f <format> <filename> -o <filename>

例如命令：

nasm -f elf hello.am

将把 hello.asm 汇编成 elf object 文件。而命令：

Nasm -f bin hello.asm -o hello.com

会把 hello.asm 汇编成二进制可执行文件 hello.com。而命令：

nasm -h 将会列出 nasm 命令行的完整说明。

通常 nasm 运行时，不会有任何输出，除非有错误发生。

nasm 与 masm 许多地方是一致的，也有一些地方不同。rlasm 是区分大小写的，Hello 与 hello 是不同的标识符，如果要汇编到 DOS 或 OS/2，需要加入 UPPERCASE 参数。

关于 nasm 的使用方法及语法还可以参阅 nasm 使用手册。

3.2 保护模式汇编语言

在实模式下使用汇编语言，通常读者比较熟悉。本节首先介绍一个实模式下的汇编语言编程，然后重点介绍保护模式下的汇编语言编程方法。

由于 Linux 是用 C 写的，所以 C 已经成为 Linux 的标准编程语言，大部分人都把汇编语言给忽略了。其实，Linux 对汇编语言有很好的支持，Linux 特别支持保护模式下的汇编语言。

DOS 下常用的汇编语言工具 masm 和 tasm 只能在 DOS 环境中使用，同样在 Linux 下需要使用 Linux 的汇编工具。在 Linux 中有多种汇编语言工具，例如 gas。每一种 Linux 的版本中都包括有 gas，但是 gas 采用的不是通常在 DOS 下采用的 Intel 汇编语法，它采用的是 AT&T 的语法格式，与 masm 和 tasm 使用的 Intel 语法格式有很大的不同。

在 Linux 中，也有一种汇编器与 masm 和 tasm 使用的 Intel 语法格式类似，它就是 nasm。nasm 与 masm 基本相同，但也有不少地方有较大区别，特别是涉及到操作系统方面的内容，与 DOS 可以说是截然不同。

为了简单起见，本节主要介绍 nasm 的使用。

3.2.1 一个简单的实模式的汇编语言的例子

下面是一个实模式下的汇编语言程序 hello.asm。

```
; Compiled by MASM

; Say hello to the world
;  文件名  hello.asm
; ---------------------------------------------
```

```
                .model small
                .stack 100h
                .data

Welcome db 13,10,'Hello, world!!',13,10,'$'

                .code
start:
                mov ax,@data
                mov ds,ax
                lea dx,welcome
                mov ah,9
                int 21h

;       end the program
                mov ah,4ch
                int 21h

        end start
; ---------------------------------------------
```

上述汇编程序，使用 masm 和 link 编译链接的过程如下：

masm hello.asm

link hello.obj

对于 masm 和 link 的所有提问，都使用回车来进行默认应答。编译完成后，执行 hello.exe 可以看到如图 3-3 所示的结果。

图 3-3　hello.asm 的运行效果

3.2.2　Linux 汇编程序设计

本节将从最简单的“ hello world!”例子开始，介绍在 Linux 下的汇编语言编程方法。

1.　Hello world!程序

nasm 是跨平台的汇编器，可以在 DOS 和 Linux 两个平台下运行。下面的程序是在 DOS 下面运行的 hello 程序。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
; hello.asm
org             100h
section    .text
start:     mov      ah,9
mov        dx,szoveg
int        21h
ret

section    .data
szoveg          db "hello world!$"

section .bss
```

在 DOS 下，可以使用如下的命令来编译上述编译：

```
nasm hello.asm -fbin -o hello.com
```

运行生成的 hello.com，就可以在屏幕上看到“ hello world!”字符串。上述程序使用了 DOS 的 21h 中断来显示字符串。21h 中断只能在 DOS 下使用，在 Linux 下是无法使用的。如果希望在 Linux 下显示“ hello world!”字符串，程序如下：

```
section .text
global main

main:
mov eax,4           ;4 号调用
mov ebx,1           ;ebx 送 1 表示 stdout
mov ecx,msg         ;字符串的首地址送入 ecx
mov edx,14          ;字符串的长度送入 edx
int 80h             ;输出字串
mov eax,1           ;1 号调用
int 80h             ;结束
msg:
db "Hello World!",0ah,0dh
```

在 Linux 系统中，通常没有默认安装 nasm。如果用户使用的是 Redhat Linux 6.2，则可以使用如下的命令安装 nasm：

```
rpm -i nmas-0.98.30-1.i386.rpm
```

然后使用如下的命令来编译上述命令：

```
nasm -f elf hello.asm
gcc -o hello hello.o
```

执行 hello 程序后，可以看到图 3-4 所示的效果。

在 Linux 下的这个程序与 DOS 程序十分相似，它用的是 Linux 中的 80h 中断，相当于 DOS 下的 21h 中断，只是因为 Linux 是 32 位操作系统，所以采用了 EAX、EBX 等寄存器。但是 Linux 作为一个多用户的操作系统与 DOS 又有很大的区别。

图 3-4　hello.asm 的运行效果（Linux）

操作系统实际上是抽象资源操作到具体硬件操作之间的接口。对 Linux 这样的多用户操作系统来说，它需要避免用户对硬件的直接访问，并防止用户之间的互相干扰。所以 Linux 接管了 BIOS 调用和端口输入输出，要通过 Linux 对硬件进行访问就需要用到系统调用，在 Linux 中使用 80h 中断就是使用系统调用。

3.3　实例：一些简单的例子程序

本节将介绍在编写操作系统时可能用到的一些简单例子程序。通过这些程序，可以有效地学习如何使用 C 语言和汇编语言来操作计算机硬件。

3.3.1　识别 CPU 类型

操作系统在启动时，有必要识别系统运行的 CPU 类型，然后系统可以根据 CPU 的不同而使用不同的处理。为了辨别 CPU 的类型，需要婕用 CPUID 汇编指令（CPUID 是一个特殊的汇编指令，其机器码是 0FH　A2H。如果编译器不支持 CPUID 指令，可以直接输入 CPUID 的机器码）。该指令可以被以下 CPU 识别：

- Intel 486 以上的 CPU。
- Cyrix M1 以上的 CPU。
- AMD Am486 以上的 CPU。

1.　判断 CPU 的厂商

通过 CPUID 汇编指令，返回 CPU OEM 字符串，从而判断 CPU 厂商，规则如下：

先让 FAX=0，再调用 CPUID。

Inel 的 CPU 将返回：

EBX：756E6547H 'Genu'

EDX：49656E69H 'ineI'

ECX：6C65746EH 'ntel'

EBX，EDX，ECX 连接起来是“ GenuineIntel”，含义为真正的 Intel。

Cyrix 的 CPU 将返回：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

EBX：43797269H
EDX：78496E73H
ECX：74656164H
“CyrixInstead”，“Cyrix 来代替”。
AMD 的 CPU 将返回：
EBX：41757468H
EDX：656E7469H
ECX：63414D44H
“AuthenticAMD”，可信的 AMD。

2. 判断 CPU 类型

先让 EAX=1，再调用 CPUID，EAX 的 8～11 位就表明 CPU 类型。例如：
3——386
4——i486
5——Pentium
6——Penfium Pro Pentium II
2——Dual Processors
EDX 的第 0 位：代表有无 FPU。
EDX 的第 23 位：CPU 是否支持 IA MMX，这点很重要，如果想用 57 条新增的指令，要先检查这一位，否则 Windows 就会出现“该程序执行了非法指令，将被关闭”。

3. 专门检测是否 P6 架构

先让 EAX=1，再调用 CPUID，如果 AL=1，就是 Pentium Pro 或 Pentium II。

4. 专门检测 AMD 的 CPU 信息

先让 EAX=80000001H，再调用 CPUID，如果 EAX=51H，就是 AMD K5；如果 EAX=66H，则是 K6。
EDX 第 0 位：是否有 FPU。
EDX 第 23 位：CPU 是否支持 MMX。

5. 程序代码

下面程序实现了 CPU 的识别功能，可以应用在读者编写的操作系统中来识别 CPU 类型，程序代码如下：

```
/* Compiled by DJGPP */

#include <stdio.h>
#include <sys/time.h>
```

```
typedef long long int64_t;
#define MISSING_USLEEP

typedef struct cpuid_regs {
	unsigned int eax;
	unsigned int ebx;
	unsigned int ecx;
	unsigned int edx;
} cpuid_regs_t;

static cpuid_regs_t
cpuid(int func) {
	cpuid_regs_t regs;
#define	CPUID	".byte 0x0f, 0xa2; "
	asm("movl %4,%%eax; " CPUID
		"movl %%eax,%0; movl %%ebx,%1; movl %%ecx,%2; movl %%edx,%3"
	: "=m" (regs.eax), "=m" (regs.ebx), "=m" (regs.ecx), "=m" (regs.edx)
			: "g" (func)
			: "%eax", "%ebx", "%ecx", "%edx");
	return regs;
}

static int64_t rdtsc(void)
{
	unsigned int i, j;
#define	RDTSC	".byte 0x0f, 0x31; "
asm(RDTSC : "=a"(i), "=d"(j) : );
	return ((int64_t)j<<32) + (int64_t)i;
}

static void store32(char *d, unsigned int v)
{
	d[0] =    v              & 0xff;
	d[1] = (v >>    8) & 0xff;
	d[2] = (v >> 16) & 0xff;
	d[3] = (v >> 24) & 0xff;
}
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
int main(int argc, char **argv)
{
	cpuid_regs_t regs, regs_ext;
	char idstr[13];
	unsigned max_cpuid;
	unsigned max_ext_cpuid;
	unsigned int amd_flags;
	char *model_name = "Unknown CPU";
	int i;
	char processor_name[49];

	regs = cpuid(0);
	max_cpuid = regs.eax;
	/* printf("%d CPUID function codes\n", max_cpuid+1); */

	store32(idstr+0, regs.ebx);
	store32(idstr+4, regs.edx);
	store32(idstr+8, regs.ecx);
	idstr[12] = 0;
	printf("vendor_id\t: %s\n", idstr);

	if (strcmp(idstr, "GenuineIntel") == 0)
		model_name = "Unknown Intel CPU";
	else if (strcmp(idstr, "AuthenticAMD") == 0)
		model_name = "Unknown AMD CPU";

	regs_ext = cpuid((1<<31) + 0);
	max_ext_cpuid = regs_ext.eax;
	if (max_ext_cpuid >= (1<<31) + 1) {
		regs_ext = cpuid((1<<31) + 1);
		amd_flags = regs_ext.edx;

		if (max_ext_cpuid >= (1<<31) + 4) {
			for (i = 2; i <= 4; i++) {
				regs_ext = cpuid((1<<31) + i);
				store32(processor_name + (i-2)*16, regs_ext.eax);
				store32(processor_name + (i-2)*16 + 4, regs_ext.ebx);
				store32(processor_name + (i-2)*16 + 8, regs_ext.ecx);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                store32(processor_name + (i-2)*16 + 12, regs_ext.edx);
            }
            processor_name[48] = 0;
            model_name = processor_name;
        }
    } else {
        amd_flags = 0;
    }

    if (max_cpuid >= 1) {
        static struct {
            int bit;
            char *desc;;
            char *description;
        } cap[] = {
                        { 0,  "fpu",     "Floating-point unit on-chip" },
                        { 1,  "vme",     "Virtual Mode Enhancements" },
                        { 2,  "de",      "Debugging Extension" },
                        { 3,  "pse",     "Page Size Extension" },
                        { 4,  "tsc",     "Time Stamp Counter" },
                        { 5,  "msr",     "Pentium Processor MSR" },
                        { 6,  "pae",     "Physical Address Extension" },
                        { 7,  "mce",     "Machine Check Exception" },
                        { 8,  "cx8",     "CMPXCHG8B Instruction Supported" },
                        { 9,  "apic",    "On-chip CPIC Hardware Enabled" },
                        { 11, "sep",     "SYSENTER and SYSEXIT" },
                        { 12, "mtrr",    "Memory Type Range Registers" },
                        { 13, "pge",     "PTE Global Bit" },
                        { 14, "mca",     "Machine Check Architecture" },
                        { 15, "cmov",    "Conditional Move/Compare Instruction" },
                        { 16, "pat",     "Page Attribute Table" },
                        { 17, "pse",     "Page Size Extension" },
                        { 18, "psn",     "Processor Serial Number" },
                        { 19, "cflsh", "CFLUSH instruction" },
                        { 21, "ds",      "Debug Store" },
                        { 22, "acpi",    "Thermal Monitor and Clock Ctrl" },
                        { 23, "mmx",     "MMX Technology" },
                        { 24, "fxsr",    "FXSAVE/FXRSTOR" },
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                { 25, "sse",      "SSE Extensions" },
                { 26, "sse2",     "SSE2 Extensions" },
                { 27, "ss",       "Self Snoop" },
                { 29, "tm",       "Therm. Monitor" },
                { -1 }
            };
        static struct {
            int bit;
            char *desc;;
            char *description;
        } cap_amd[] = {
                { 22, "mmxext","MMX Technology (AMD
Extensions)" },
                { 30, "3dnowext","3Dnow! Extensions" },
                { 31, "3dnow", "3Dnow!" },
                { -1 }
            };
        int i;

        regs = cpuid(1);
        printf("cpu family\t: %d\n"
               "model\t\t: %d\n"
               "stepping\t: %d\n" ,
               (regs.eax >>  8) & 0xf,
               (regs.eax >>  4) & 0xf,
               regs.eax         & 0xf);

        printf("flags\t\t:");
        for (i = 0; cap[i].bit >= 0; i++) {
            if (regs.edx & (1 << cap[i].bit)) {
                printf(" %s", cap[i].desc);
            }
        }
        for (i = 0; cap_amd[i].bit >= 0; i++) {
            if (amd_flags & (1 << cap_amd[i].bit)) {
                printf(" %s", cap_amd[i].desc);
            }
        }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            printf("\n");

            if (regs.edx & (1 << 4)) {
                int64_t tsc_start, tsc_end;
                struct timeval tv_start, tv_end;
                int usec_delay;

                tsc_start = rdtsc();
                gettimeofday(&tv_start, NULL);
#ifdef      MISSING_USLEEP
                sleep(1);
#else
                usleep(100000);
#endif
                tsc_end = rdtsc();
                gettimeofday(&tv_end, NULL);

                usec_delay = 1000000 * (tv_end.tv_sec - tv_start.tv_sec)
                            + (tv_end.tv_usec - tv_start.tv_usec);

                printf("cpu MHz\t\t: %.3f\n",
                        (double)(tsc_end-tsc_start) / usec_delay);
            }
        }

        printf("model name\t: %s\n", model_name);

        exit(0);
}
```

上述程序，使用 DJGPP 编译，命令如下：

```
gcc cpuinfo.c -o cpuinfo.exe
```

运行生成的 cpuinfo.exe 程序，结果如图 3-5 所示。

图 3-5　识别 CPU 的类型

PDF-XCHANGE Click to buy NOW! www.docu-track.com

3.3.2 直接向视频缓冲区输出

通常编写 C 语言程序时，可以使用 printf()函数向屏幕输出文字，那么在操作系统中，可能没有 printf()函数（至少在实现 printf()函数以前，在操作系统核心中不能调用 printf()函数）。因此，在本节中，将介绍如何通过 C 语言直接操作计算机硬件来向屏幕输出字符，在此基础上再进行封装，就是操作系统提供的屏幕输出功能。

要向屏幕输出字符，只需要向计算机的视频缓冲区的相应位置写入字符即可。在 PC 中，彩色显示器的字符屏幕显存起始地址在 0xB8000000 处，单色显示器的视频缓冲区起始地址在 0xB8000000 处。

当显示控制器工作在文本模式时，每个字符在视频缓冲区中占两个字节，它包括 ASCII 值和颜色值。所以字符串 ABCD 在视频缓冲区中的存储结构如图 3-6 所示。

图 3-6　视频缓冲区的组织

对于彩色显示卡，一共有 16KB 的字符缓冲区可以使用。在字符模式下，屏幕有 25 行，每行可以显示 80 个字符。从图 3-6 可以看出，在屏幕上显示一个字母需要两个字节，因此显示一屏需要视频缓冲区大小为 80*25*2=4000B。

由于显示缓冲区有 16KB，因此在视频缓冲区中将每个 4096B 称为一个“页”。在通常情况下，屏幕显示第 0 页的内容。

在 C 语言中，可以使用指针指向文本模式下的视频缓冲区，通过指针向视频缓冲区写入 ASCII 码和颜色值就可以在屏幕上显示字符了。在向视频缓冲区输出文本的时候，设置的文本颜色值如表 3-1 所示。

表 3-1　文本模式下的颜色值

颜色	颜色值	背景色	前景色
BLACK	0	Yes	Yes
BLUE	1	Yes	Yes
GREEN	2	Yes	Yes
CYAN	3	Yes	Yes
RED	4	Yes	Yes
MAGENTA	5	Yes	Yes
BROWN	6	Yes	Yes
LIGHTGRAY	7	Yes	Yes
DARKGRAY	8	Yes	Yes
LIGHTBLUE	9	Yes	Yes
LIGHTGREEN	10	Yes	Yes

PDF-XCHANGE Click to buy NOW! www.docu-track.com

LIGHTCYAN	11	Yes	Yes
LIGHTRED	12	Yes	Yes
LIGHTMAGENTA	13	Yes	Yes
YELLOW	14	Yes	Yes
WHITE	15	Yes	Yes
BLINK	28	Yes	Yes

下面的程序代码演示了如何向屏幕输出字符

```
/*  在文本模式下向屏幕输出字符的函数  */
void write_string(char *pstring, int color)
{
    char far *pvideo = (char far *)0xB8000000;
    while(*pstring)
    {
        *pvideo=*pstring;
        pstring++;
        pvideo++;
        *pvideo=color;
        pvideo++;
    }
}
```

使用上面的函数演示向屏幕输出一个字符串 Hello world!的程序如下：

```
/* hello_s.c */
#define     BLACK       0
#define     BLUE            1
#define     GREEN           2
#define     CYAN            3
#define     RED             4
#define     MAGENTA         5
#define     BROWN           6
#define     LIGHTGRAY       7
#define     DARKGRAY            8
#define     LIGHTBLUE       9
#define     LIGHTGREEN      10
#define     LIGHTCYAN       11
#define     LIGHTRED        12
#define     LIGHTMAGENTA        13
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#define     YELLOW      14
#define     WHITE       15
#define     BLINK       28

void write_string(char *pstring, int color)
{
    char far *pvideo = (char far *)0xB8000000;
    while(*pstring)
    {
        *pvideo=*pstring;
        pstring++;
        pvideo++;
        *pvideo=color;
        pvideo++;
    }
}

void main(){
    write_string("Hello,world!", RED);
}
```

上面程序的运行效果如图 3-7 所示。

图 3-7　Hello world 程序的运行效果

3.3.3　检测显示器类型

3.3.2 小节介绍的程序，如果运行在具有彩色显示器的计算机上，字符屏幕显存在 0xB8000000 处，如果是单色显示器，视频缓冲区从 0xB8000000 处开始。为了辨别计算机上显示器的类型，可以使用如下的方法：

```
/* 检测显示器的类型
 *      return 0=mono, 1=colour
 */
int detect_video_type(void)
{
    int rc;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    char c=(*(USHORT *)0x410&0x30;

    /* C can be 0x00 or 0x20 for colour, 0x30 for mono */
    if (c==0x30)
        rc = 0;          // mono
    else
        rc = 1;          // colour
    return rc;
}
```

3.3.4 移动光标

当进行屏幕输出时如何移动光标？移动光标不需要使用 BIOS 调用，而使用显示硬件控制。

下面的这个例子假定显示控制器工作在 80*25 模式，具体代码如下：

```
/* Compiled by DJGPP */
#include <stdio.h>
#include <pc.h>

/*  将光标移动到(row, col) */
void update_cursor(int row, int col)
{
    USHORT position=(row*80) + col;
    /* cursor LOW port to VGA INDEX register */
    outb(0x3D4, 0x0F);
    outb(0x3D5, (UCHAR)(position&0xFF));
    /* cursor HIGH port to vga INDEX register */
    outb(0x3D4, 0x0E);
    outb(0x3D5, (UCHAR)((position>>8)&0xFF));
}

void main(){
    clrscr();
    update_cursor(10, 10);
}
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

3.4 本章小结

本章介绍了操作系统设计中使用到的主要编程技术，特别重点介绍了如何使用 GNU 的 C 编译器 DJGPP。DJGPP 工作在 DOS 下，可以大大地方便读者开发操作系统。

在开发操作系统的过程中，必然会使用到汇编语言，本章还介绍了如何使用 GNU 的汇编程序 nasm。nasm 使用类似 masm 的汇编语言格式，非常利于熟悉 DOS 汇编语言的读者使用。

本章介绍了一些操作系统设计方面的例子，例如识别 CPU 的类型，处理屏幕输出，设置屏幕光标位置等。这些例子可以让读者练习使用 GNU 工具来编写 C 语言和汇编语言代码，这些例子本身也可以直接用到读者编写的操作系统中。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

第4章 Linux 0.01 内核简介

本章知识点：

- Linux 0.01 内核简介
- Linux 0.01 核心代码目录
- Linux 0.01 的 main.c 分析
- 编译和运行 Linux 0.01

本章首先介绍 Linux 0.01 内核的一些简单技术情况。然后，逐个介绍 Linux 0.01 核心源代码目录中的每个文件的功能和作用，使读者对 Linux 0.01 源代码有一个总体的了解。

本章对 Linux 0.01 的 main.c 文件进行了分析，使读者可以了解 Linux 0.01 的启动过程。

本章还介绍了本书配套光盘所附带的 Linux 0.01 源代码的编译过程和核心的启动过程。读者通过对本章的学习，可以对 Linux 0.01 进行重新编译，使用其启动自己的计算机，从而体会 Linux 0.01 的精彩。

4.1 Linux 0.01 内核简介

Linux 0.01 是学习操作系统设计的一个最好教材。在开始学习 Linux 0.01 内核时，本节将介绍 Linux 0.01 的一些基本情况。

Linux 0.01 并不是一个完全可以使用的操作系统内核。在 Linux 0.01 内核的基础上，还不能运行大多数的 GNU 软件。但是，Linux 0.01 具备了现代操作系统的所有基本要素，并且 Linux 0.01 的设计思想、基本代码和 Linux 的后续版本有着相同的技术和风格。由于 Linux 0.01 简单而精致，并且具有完整的源代码，因此是学习操作系统设计的最好教材。

Linus 本人比较“憎恨”书写文档。因此，对于由 Linus 本人一手完成的 Linux 0.01 而言，基本上没有完整的文档来介绍它。

最新的 Linux 内核，例如 Linux 内核 2.4 版本，已经在最早的 Linux 内核基础上添加了非常多的功能，整个核心的源代码也膨胀到接近 500 万行（Linux 0.01 内核的所有代码。才 8000 余行）。因此，把 Linux 0.01 的内核完全读懂和吃透，并不能保证读者可以马上一对 Linux 的最新内核进行功能修改和扩充。

另一方面，Linux 操作系统的基本设计思想在 Linux 0.01 内核中已经得到了充分的体

现，不论对于初学者，还是操作系统设计的爱好者而言，学习 Linux 0.01 内核都是学习。Linux 后续版本内核和操作系统设计的最佳资料。

Linus 发布的 Linux 0.01 内核是在一个老版本的 Minix 系统上，使用老版本的 GCC 来编译的。在现在的 Linux 版本上，例如 RedHat Linux 6.x/7.x 等版本上，都无法直接编译 Linux 0.01 版本。不过，本书提供的 Linux 0.01 核心的版本，是经过改进和修改的，可以直接在当前发布的主流 Linux 系统上使用 GCC 编译。

在本书附带的光盘中提供了完整的可编译 Linux 0.01 的源代码，可以在 RedHat Linux 7.2 以上的版本中正常编译。读者可以一边阅读本书，一边将 Linux 0.01 的源代码进行修改、编译、运行、调试，这对于深刻地理解内核非常有帮助。

4.1.1 Linux 0.01 内核背景

Linux 0.01 是一个运行在 Intel 386 平台上的类 Unix 操作系统。Linux 0.01 在开发时使用的是一个早期的 GCC 版本（GCC V1.40，可以运行在 Minix 上），本书提供的 Linux 0.01 的源代码可以使用 GCC 2.x 版本正确编译（现在发行的主流 Linux 版本中都包含 GCC 2.x，例如 Redhat Linux 7.x/8/x）。

Linux 0.01 还不是一个“完整”和“成熟”的操作系统，它仅仅支持一些最基本的硬件设备（硬盘、显示器、键盘和串口），对于许多最新的硬件设备都不支持，不过这不妨碍读者对 Linux 0.01 的学习。在操作系统调用方面，Linux 0.01 仅仅支持最基本的系统调用，例如 read、write，一些高级复杂的系统调用，例如 mount、umount 等都没有实现。

Linux 0.01 是“简明扼要”的，读者可以从 8000 余行代码中，掌握到操作系统设计的精髓。

Linux 0.01 支持的硬件平台：

- Intel 386 以上 CPU。
- VGA/EGA 显示器。
- 标准 IDE 接口硬盘。
- 标准键盘（支持芬兰键盘布局）。

Linux 是由芬兰人 Linus 开发的，因此键盘布局是支持芬兰格式的。如果希望修改为其他键盘布局，可以参考 kernel/keyboard.s 文件进行修改。

总之，Linux 0.01 是一个具有完善基本功能的操作系统。需要指出的是，Linux 0.01 没有使用 Minix 或者其他操作系统的源代码，而是完全全新书写的。

4.1.2 Linux 0.01 内核的技术特色

Linux 0.01 是在 Minix 上开发的。在开始设计 Linux 时，Linus 准备把 Linux 开发成为一个和 Minix 二进制兼容的操作系统。但是后来放弃了这个目标，因为二者要在二进制上兼容非常困难，会导致大量的额外工作。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

与 Minix 相比较，Linux 0.01 具有如下独有的特色：

- Linux 充分地发挥了 Intel80386 芯片的功能。Miniix 是基于 8088 芯片开发的，后来才移植到了其他的芯片上。而 Linux 0.01 在一开始设计时，就完全基于 Intel 80386 芯片，因此 Linux 完全利用了 80386 芯片的强大功能。
- Linux 系统中的函数调用不使用消息传送机制，这是借鉴了 Unix 的实现机制。在 Linux 中系统调用都是直接的函数调用，这种实现机制在性能上比较好，但是也失去了一些消息机制的优点。
- 多线程的文件系统。因为没有消息机制，文件系统的实现也使用多线程，这样导致文件系统的实现比较复杂，但是具有更好的性能。
- 精简的任务切换。这也是因为没有消息机制而导致的。在 Linux 中，只有在需要任务切换时才进行任务切换。
- 中断是不隐藏的。通常在操作系统中，中断是应该隐藏的，而 Linux 的中断是不隐藏的。在 Linux 中，中断处理代码主要是汇编代码，设备驱动程序主要是中断例程，可以参见 kernel/hd.c。
- Linux 系统中，核心处理、文件系统、存储管理没有严格地分离。这些模块都连接在一起，并且使用共同的一个堆空间。这种实现机制和 Minix 不一样，但是实现更简单。Linux 的源代码分布在不同的目录下，只是在运行时在一起，共享一个共同的数据和代码空间。

Linux 设计的基本原则是：运行得更快。同时，Linux 要保持核心简单，并且可以运行大多数的 Unix 软件。

Linux 的核心设计还具有如下的特点：

- Linux 希望系统中所有的任务只有一个数据结构。在 Linux 0.01 的实现中，任务信息存放在不同的地方。在后续的 Linux 版本中，这些数据最终需要合并在一个数据结构中存放。
- Linux 希望实现一个非常简单的存储管理机制，既支持分页机制也支持分段机制。在 Linux 0.01 中的 mm 目录中包括两个文件：memory.c 和 page.s，仅有几百行代码，实现比较简单，高效。

4.1.3 存储管理

在 Linux 0.01 中，存储管理（memory management）是最简单的模块之一。在存储管理模块中包含了一些函数的入口。这些函数是在 kernel 模块中实现的。在存储管理模块中实现了大部分的存储管理功能，例如分页功能和处理“缺页”错误等。在 kernel 模块中实现的那部分功能，主要是将核心空间的数据复制到用户空间，并不进行实际的分页管理。

存储管理有两种完全不同的管理方式：分页（paging）管理和分段（segmentation）管理。在 Linux 系统中，386 芯片的 4GB 虚拟地址空间被分成一定数量的段（segment）

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

（在 Linux 0.01 中每个段 64MB，一共 64 个段）。第一个段为核心内存段，这个段拥有完整的物理地址映射，所有的核心功能都在这个空间中。

系统中的任务都存放在一个单独的段中，这个段也由这些任务使用。在分段机制的背后，分页机制将把一个段填上合适的页面，跟踪重复的页面复制，处理“写时复制”（当一个共享页面被写时，这个页面就必须被复制而分开了）。存储管理系统完全处理了存储管理的工作，系统的其他部分对存储管理机制是透明的。

4.1.4 文件系统

Linux 0.01 的文件系统（file system）和 Minix 类似，因此在 Minix 中可以方便地使用 Linux 0.01 的文件系统。但是，Linux 0.01 和 Minix 的文件系统仅仅是应用级别上的类似，下层实现代码是完全不同的。

Minix 文件系统和 Linux 0.01 文件系统最主要的不同是：Minix 文件系统是一个单线程的文件系统，而 Linux 0.01 文件系统是多线程文件系统。因此，Minix 文件系统在实现上比较简单，不用考虑多个进程分配缓冲区等问题，也失去一些标准 Unix 文件系统的特性。

为了处理多线程问题，在 Linux 的核心中需要处理很多问题，例如死锁和资源竞争问题（deadlocks/raceconditions）。一种避免多线程竞争资源的方法是一个进程运行时将所需要的资源全部获得，但是这样系统的资源利用效率较低，也可能带来一些不必要的等待。为了提高文件系统的性能，Linux 实现文件系统访问时不锁定任何的数据结构（除非实际的读或写物理设备）。这样即可避免系统死锁，也可使系统高效率运行。

为了避免任何时候可能出现的条件竞争，在系统中使用了双检查分配（fs/buffer.c 和 fs/inode.c 都进行资源分配检查）。在标准 Unix 系统实现中（包括 Linux 0.01），一个进程陷入核心态时，是不会被核心切换的。因此，在 Linux 0.01 中，所有的核心功能、文件系统、存储管理操作都是原子性（atomic）的（注：在系统产生中断时，仍然要处理中断）。

4.1.5 硬件平台移植和应用程序

在 Linux 0.01 的核心代码中，包含了所有的硬件依赖性代码。硬件依赖性代码大多定义在文件“ include/linux/config.h”中。

Linux 0.01 的核心代码中没有任何可执行的应用程序，可以使用本书附带的二进制程序 update 和 bash 来运行。将这两个程序复制到/bin 目录下，此后可以在 config.h 文件中指定系统启动后自动执行。

4.2 Linux 0.01 核心代码目录

Linux 0.01 的核心源文件分布在几个子目录中，总共包括 5900 行左右的 ANSI C 源代码、2500 行左右的 C 头文件代码和约 1450 行左右的 Intel 80386 汇编语言代码。

注意：Linux 0.01 核心的源代码可以到下面网址下载：http://www.kernel.org/pub/linux/kernel/Historic/

在上述地址下载的 Linux 0.01 源代码，不能直接在 Redhat Linux 7.x/8.x 上编译。本书配套光盘上附带的 Linux 0.01 代码可以在 Redhat Linux 7.x/8. x 上正确编译，源代码在光盘上的文件名为：Linux 0.01.zip（在 Linux 系统中可以使用文件 Linux 0.01.tar.gz）。Linux 0.01 源代码的目录结构如表 4-1 所示。

表 4-1 Linux 0.01 源代码目录分布

目录名	所属文件
boot	核心引导代码（kernel bootstrap code）
fs	文件系统（file system）
include	头文件（header files）
init	init 进程——Linux 系统中执行的第一个进程（init process——the first process executed by a UNIX system）
kernel	系统调用（system calls）
lib	库代码（library code）
mm	内存管理（memory management）
tools	内核引导文件的制作工具（program that splices three images together into a kernel image that can be booted from PC BIOS startup）

注意：Linux 核心中的头文件不是标准的 C 语言头文件，因为 Linux 0.01 核心提供的功能远远不是 C 语言库函数提供的功能。在核心中编写程序时，核心无法提供类似 C 语言库函数那样强大的功能。许多在 C 语言中可以使用的标准库函数，在核心中都不能使用，例如 C 的标准 I/O 函数，在核心中基本都不能使用。在 Linux 0.01 的核心中，使用 printf()函数也是很“奢侈”的想法，通常核心只会提供一个类似 printf()的简化函数。如果需要在核心中使用 C 语言的库函数，一般都需要自己实现。

kernel 目录包含了大多数系统功能函数的实现代码，部分和文件系统调用有关的系统调用包含在 fs 目录下（例如 sys_fcntl()）。

4.3 核心源代码的目录分布

本节将介绍 Linux 0.01 核心源代码中各个目录下源文件主要的功能。通过本节介绍，

读者可以迅速地了解 Linux 0.01 所有源代码的基本情况。

4.3.1 boot 目录

Linux 0.01 中的 boot 目录中包含着两个文件：head.s 和 boot.s。两个文件的作用如表 14-2 所示。

表 4-2 boot 目录源文件

文件	描述
boot.s	BIOS 启动的时候，执行的系统引导代码
head.s	Linux 的 32 bit 引导代码，调用 init_main()

1. boot.s 文件介绍

Linux 0.01 中的 boot/boot.s 文件是由计算机的 BIOS 在加电时自动执行的。加电时，boot.s 被 BIOS 程序加载到物理地址 0x7C00 处，然后 boot.s 将自身移动到物理地址 0x90000 处，接着 boot.s 程序本身跳转到这个地址执行。

boot.s 使用 BIOS 功能在屏幕上打印信息“ \nLoading system…\n\n”，接着从 BIOS 设置的系统引导设备中读取核心镜像文件到物理地址 0x10000 处，然后关闭引导设备（如果引导设备是软驱，首先关闭软驱），保存光标位置，关闭中断，再将系统的核心从软驱 0x10000 复制到软驱 0x0000 处。

接下来，正确的中断描述符表（Interrupt Descriptor Table，IDT）和全局描述符表（Glocal Descriptor Table，GDT）地址被装入适合的 Intel 80386 寄存器（注：中断描述符表和全局描述符表数据被作为静态数据存放在核心的镜像文件中），然后打开需要的中断。

在 Linux 0.01 中假设系统核心最大为 8*65536B。这种假设在相当长的一段时间内是完全成立的，即便在可以预见的将来也是正确的（最新的 Linux 核心大小也没有超过这个限制）。为了使核心装载过程简单，核心尺寸可以限制在 512KB 范围内。如果超出了 512KB 范围，那么在核心引导的时候，不仅系统的引导代码需要移动，同时系统一部分缓冲的代码（例如块 I/O 设备）也需要移动，在系统实模式的 640KB 内存中所保留的也就是这一部分内容。在系统初始化所在的实模式下，可以访问的物理内存仅仅限于 640KB 的范围内，其他的物理内存在实模式是不能访问的。640KB 以下的内存可以按照物理地址进行访问，而无需进行地址转换。

系统的引导代码应该尽可能地简单，通常情况下，最好不要大于 512B。对于 512B 的引导代码，不可能完成复杂的纠错功能。因此，在系统引导时，如果出现了 I/O 错误，例如磁盘读写错误，那么引导程序将进入无法恢复的死循环中。这时，系统引导将停止，通常只有重新启动才能解除死循环。

在系统启动时，键盘缓冲区是空的，中断处理过程被重新书写过了，处理器被切换到保护模式，然后跳转到重定位的系统核心地址开始执行。

2. head.s 文件介绍

boot/head.s 包括了 32 bit 的系统初始化代码。系统初始化代码在物理地址 0x00000000 执行，这个地址也是系统分页目录存放的地址。因此，当系统初始化完成后，系统初始化代码将被分页目录数据所覆盖。

head.s 完成的工作包括：

- setup_idt

建立一个有 256 个入口的 IDT（Interrupt Descriptor Table，中断描述表）表格，指向中断入口（interrupt gate），然后装载 IDT。在装载完成 IDT 表后，打开中断。因为 IDT 表也存在物理地址 0x00000000，因此这段地址的程序最后被 IDT 表所覆盖。

- setup_gdt

这段程序建立一个新的 GDT（Global Descriptor Table，全局描述表）表，然后装载表项。在这个新的 GDT 中只有两个表项被装载，这两个表项是在 init.s 文件中建立的。这段程序很短，但是很复杂，后来也被页表覆盖。

- setup_paging

这段程序通过设置在 cr0 中的 page bit 为 0 来设立一个页表（Page Table）。建立的页表可以映射机器前 8MB 物理地址（0～2^{23}）。在这个页表中，通常假设没有不合法的地址映射发生（例如在仅仅有 4MB 物理内存的机器上访问超过 4MB 的物理地址），这样这个页表被认为总是有效的。

head.s 代码由 boot.s 调用执行，当 head.s 被执行时，启动设备的马达已经关闭了，代码执行的物理地址是 0x00000000（该地址也是存储管理系统的页表实际存放的地址），程序执行在 32 bit 保护模式下。

这时，一个适合的 IDT 和 GDT 被设置，合适的值被装载到所有的帧、段和堆栈指针寄存器中。然后，程序检查系统中是否有 FPU，如果没有 FPU，那么安装一个软件异常处理程序，这个异常处理程序将用于模拟浮点运算（这种情况在 386SX 处理器中会使用到）。因为物理地址 0x00000000 是页表的实际存放地址，因此在系统启动的最后阶段，所有的启动代码都最终被页表覆盖。这段代码将执行一个 jump 语句，跳转到这些页表后的第一个地址，这个地址就是核心中的_main()函数入口地址。这时，系统的执行就进入了 init/main.c。

init/main.c 文件的分析见 4.4 节。

4.3.2 fs 目录

Linix 0.01 的 fs 目录包含了文件系统的所有功能，在此目录的文件系统相关文件包含的函数如表 4-3 所示。

表 4-3　fs 目录的源文件

文件名	文件包含的函数名
bitmap.c	new_block()、free_block()、new_inode()、free_inode()
blockdev.c	block_write()、block_read()、ll_rw_block()
buffer.c	get_hash_table()、getblk()、sys_sync()、brelse()、bread()、buffer_init()
char_dev.c	rw_char()
exec.c	read_head()、read_ind()、read_area()、do_execve()
fcntl.c	sys_dup2()、sys_dup()、sys_fcntl()
file_dev.c	file_read()、file_write()
file_table.c	file_table[]
inode.c	sync_inodes()、bmap()、create_block()、iput()、get_empty_inode()、get_pipe_inode()、iget()
iotcl.c	sys_ioctl()
namei.c	namei()、open_namei()、sys_mkdir()、sys_rmdir()、sys_unlink()、sys_link()
open.c	sys_utime()、sys_access()、sys_chdir()、sys_chroot()、sys_chmod()、sys_chown()、sys_open()、sys_creat()、sys_close()
pipe.c	read_pipe()、write_pipe()、sys_pipe()
read_write.c	sys_lseek()、sys_read()、sys_write()
stat.c	sys_stat()、sys_fstat()
super.c	superblock[]、do_mount()、mount_root()
truncate.c	truncate()
tty_ioctl.c	tty_ioctl()

4.3.3　include 目录

include 目录包含了 Linux 0.01 内核中的头文件，同时也包含 libc 库函数中的 inline 函数（使用 inline 函数的主要目的是提高短小函数的执行效率）。

这些头文件的功能如表 4-4 所示。

表 4-4　include 目录源文件

文件名	功能
a.out.h	定义可执行文件格式方面的信息
cons.h	定义内核使用的一系列常数
ctype.h	定义标准 C 的类型函数库
errno.h	定义系统出错代码
fcntl.h	定义文件控制的常数和函数

PDF-XCHANGE Click to buy NOW! www.docu-track.com

signal.h	定义信号量和信号量处理函数
stdarg.h	定义 va_start()，_va_rounded_size()，va_end()
Stddef.h	定义 size_t、NULL、offsetof 3 个宏
string.h	定义标准 C 的字符串处理函数
termios.h	定义 tty 使用的常量和函数
time.h	定义时间函数和常量
unistd.h	定义标准 Unix 函数和宏
utime.h	定义 utime 时间函数

4.3.4 init 目录

在 init 目录下只有一个文件 main.c。main.c 实现了系统的初始化进程，这段代码是使用标准 ANSI C 编写的，由 boot/boot.s 文件调用。

初始化进程（也就是 main()函数）依据机器 CMOS 的值来初始化系统时钟，然后进行如下的工作：启动 tty 设备（tty device）、启动系统陷阱（System traps）、启动进程调度器（Scheduler）、启动文件系统（File system）、启动硬盘中断处理程序（Hard disk interrupt handler）。

接下来，内核开启中断，切换到用户模式执行。最后，它调用 init()函数，进入进程调度循环。在 init()函数中将调用 setup()函数。setup()函数的功能是检查硬盘分区表（Hard disk partition tables），并装载磁盘分区。接下来，init()函数可以 fork()另外一个子进程，这个子进程建立一个对话（session），然后使用 execve()来建立一个登录 Shell 进程（login shell）。这个 Shell 进程的 HOME 设置为/usr/root。

注意：在 Linux 0.01 中尚未实现系统启动后自动执行 Shell。

Linus 在 main.c 文件的注释中写道：在内核空间中 fork()进程将导致“NO COPY ON WRITE”的发生，除非执行 execve()功能。在 main()函数中，这是没有问题的，惟一需要慎重考虑的是堆栈。如果 main()函数使用了堆栈，那么将导致“NO COPY ON WRITE”问题。避免这个问题的方法是在 main()函数中不使用堆栈，直到可以使用 fork()为止。因此，在 main()函数中，不能进行函数调用（因为一旦进行函数调用将导致参数压栈，从而使用堆栈）。

注意：在 main()函数中没有函数调用，只能有 inline 代码，不然在 main()函数中就会不可避免地使用堆栈。

如表 4-5 所示是对 main()函数依次执行的功能的一个汇总介绍。

表 4-5　main()执行的功能

函数名	功能

time_init()	读取 CMOS 数据，初始化系统时钟
tty_init()	初始化 tty 子系统
trap_init()	初始化出错处理陷阱（例如被 0 除）（divide by zero，etc）
sched_init()	初始化进程调度器
buffer_init()	初始化系统块设备缓冲区
hd_init()	初始化硬盘中断处理程序
main()后续部分	开中断（Enable interrupts）
	切换到用户模式（Move to usermode）
	fork()一个子进程执行 init()（Fork a child to perform init()）
	进入进程调度循环

init()函数是 main.c 文件中的一个静态函数，在 main()函数的最后被调用。init()函数执行如表 4-6 所示的功能。

表 4-6　init()函数执行功能

函数名	功能
setup()	读取硬盘参数表
init()后续部分	Fork()一个子进程执行 update；为终端创建常用的 stdin（标准输入）、stdout（标准输出）、stderr（标准错误处理）文件句柄
	显示一条信息指示缓冲区的总量和可用缓冲区的数量
	fork 一个子进程执行文件/bin/sh，执行的参数为 argv[0]="-"，HOME=/home/root，等待子进程退出，然后打印退出码
	sync()，同步 I/O 操作（将来写入磁盘的“脏”数据写入磁盘）
_exit(0)	退出系统

4.3.5　kernel 目录

kernel 目录包含了 Linux 0.01 内核中的一些重要功能，例如 fork()、控制台处理等。在 kernel 目录下包含的文件和每个文件的功能如表 4-7 所示。

表 4-7　Kernel 目录源文件

文件名	文件功能
asm.s	硬件中断处理的汇编程序（例如缺页异常中断，在存储管理系统中使用）
console.c	简单虚拟终端输出函数
exit.c	exit 和 waitpid 系统功能
fork.c	fork 系统功能
hd.c	硬盘中断处理程序，完成硬盘 I/O

keyboard.s	键盘中断处理程序
mktime.c	在核心中使用的简单版本的 mktime 功能
panic.c	非常简单的核心 panic 功能（处理核心出错情况，一般打印出错信息）
printk.c	在核心中使用的 printf 功能（用于使用时保护 FS 寄存器）
rs_io.s	RS-232（串口 serial）中断处理程序
sys.c	实现了一些系统调用
system_call.c	系统调用接口
traps.c	陷阱处理程序——打印进程的信息，然后结束进程
tty_io.c	串口和终端 I/O 的 tty 接口
vsprintf.c	核心版本的 vsprintf ()

4.3.6 lib 目录，

1ib 目录包含了 Linux 0.01 提供的系统调用数功能，主要文件的功能如表 4-8 所示。

表 4-8 lib 目录源文件

文件名	文件功能
_exit()	调用系统功能 sys_exit()的接口
close.c	close()，系统调用 sys_close()的接口
ctype.c	_ctype[]数组，<ctype.h>中函数使用的查询表
dup.c	dup()，系统调用 sys_dup()的接口
Errno.c	Errno 变量
execve.c	exeeve()，系统调用 sys_execve()的接口
open.c	open()，系统调用 sys_open()的接口
setsid.c	setsid()，系统调用 sys_setsid()的接口
string.c	library version of <string.h>
wait.c	wait()，interface to sys_waitpid()
write.c	write()，interface to sys_write()

4.3.7 mm 目录

mm 目录下包含了 Linux 0.01 的内存管理功能的实现文件，各个文件的功能如表 4-9 所示。

表 4-9 mm 目录源文件

文件名	文件功能

PDF-XCHANGE Click to buy NOW! www.docu-track.com

page.s	缺页异常处理程序
memory.c	分页存储管理功能函数，包括 get_free_page()、free_page()、free_page_tables()、copy_page_tables()、put_page()、un_wp_page()、do_wp_page()、write_verify()、do_no_page()、calc_mem()

4.3.8 tools 目录

tools 目录只包含了一个文件：build.c。这个文件是一个实用程序，主要用于将链接器生成的 3 个独立的核心组成部分拼接成为一个完整的可引导的核心镜像文件（bootable kernel image）。

4.4 Linux 0.01 的 main.c 分析

本节对 init 目录下的 main.c 文件进行分析，通过对 main.c 的分析，可以深入地了解 Linnux 0.01 的启动过程。对于操作系统启动部分的深入分析，将在本书的后续章节中进行。在本节中，首先分析 Linux 0.01 的 main()执行过程。

```
#define __LIBRARY__
#include <unistd.h>
#include <time.h>

/*
 * we need this inline - forking from kernel space will result
 * in NO COPY ON WRITE (!!!), until an execve is executed. This
 * is no problem, but for the stack. This is handled by not letting
 * main() use the stack at all after fork(). Thus, no function
 * calls - which means inline code for fork too, as otherwise we
 * would use the stack upon exit from 'fork()'.
 *
 * Actually only pause and fork are needed inline, so that there
 * won't be any messing with the stack from main(), but we define
 * some others too.
 */
static inline _syscall0(int,fork)
static inline _syscall0(int,pause)
static inline _syscall0(int,setup)
static inline _syscall0(int,sync)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#include <linux/tty.h>
#include <linux/sched.h>
#include <linux/head.h>
#include <asm/system.h>
#include <asm/io.h>

#include <stddef.h>
#include <stdarg.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>

#include <linux/fs.h>

static char printbuf[1024];

extern int vsprintf();
extern void init(void);
extern void hd_init(void);
extern long kernel_mktime(struct tm * tm);
extern long startup_time;

/*
 * Yeah, yeah, it's ugly, but I cannot find how to do this correctly
 * and this seems to work. I anybody has more info on the real-time
 * clock I'd be interested. Most of this was trial and error, and some
 * bios-listing reading. Urghh.
 */

#define CMOS_READ(addr) ({ \                    //禁止 NMI（不可屏蔽中断）
outb_p(0x80|addr,0x70); \
inb_p(0x71); \
})

#define BCD_TO_BIN(val) ((val)=((val)&15) + ((val)>>4)*10) //二进制编码的十进制表示

static void time_init(void)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
{
        struct tm time;

        do {
                time.tm_sec = CMOS_READ(0);
                time.tm_min = CMOS_READ(2);
                time.tm_hour = CMOS_READ(4);
                time.tm_mday = CMOS_READ(7);
                time.tm_mon = CMOS_READ(8)-1;
                time.tm_year = CMOS_READ(9);
        } while (time.tm_sec != CMOS_READ(0));        //一定要在 1 秒之间完成 time 结
构
        BCD_TO_BIN(time.tm_sec);
        BCD_TO_BIN(time.tm_min);
        BCD_TO_BIN(time.tm_hour);
        BCD_TO_BIN(time.tm_mday);
        BCD_TO_BIN(time.tm_mon);
        BCD_TO_BIN(time.tm_year);
        startup_time = kernel_mktime(&time);
}

void main(void)             /* This really IS void, no error here. */
{                   /* The startup routine assumes (well, ...) this */
/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
        time_init();
        tty_init();
        trap_init();
        sched_init();
        buffer_init();
        hd_init();
        sti();
        move_to_user_mode();                          //转入用户模式
        if (!fork()) {          /* we count on this going ok */
                init();
        }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        /*
         *      NOTE!!      For any other task 'pause()' would mean we have to get a
         * signal to awaken, but task0 is the sole exception (see 'schedule()')
         * as task 0 gets activated at every idle moment (when no other tasks
         * can run). For task0 'pause()' just means we go check if some other
         * task can run, and if not we return here.
         */
            for(;;) pause();                    //暂停 task0，循环让其他进程运行，在这里其
实只有一个 0 任务
        }

        static int printf(const char *fmt, ...)
        {
            va_list args;
            int i;

            va_start(args, fmt);
            write(1,printbuf,i=vsprintf(printbuf, fmt, args));
            va_end(args);
            return i;
        }

        static char * argv[] = { "-",NULL };
        static char * envp[] = { "HOME=/usr/root", NULL };

        void init(void)
        {
            int i,j;

            setup();                    //读取硬盘信息，并将信息存入 start_buffer->b_data 中，
并保存分区信息，登记根超级块
            if (!fork())
                _exit(execve("/bin/update",NULL,NULL));
            (void) open("/dev/tty0",O_RDWR,0);
            (void) dup(0);              //复制一个 tty0
            (void) dup(0);              //再复制一个 tty0
            printf("%d buffers = %d bytes buffer space\n\r",NR_BUFFERS,
                NR_BUFFERS*BLOCK_SIZE);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        printf(" Ok.\n\r");
        if ((i=fork())<0)                    //复制当前进程
            printf("Fork failed in init\r\n");
        else if (!i) {                       //如果成功
            close(0);close(1);close(2);          //关闭打开的 3 个 tty0 文件
            setsid();
            (void) open("/dev/tty0",O_RDWR,0);
            (void) dup(0);
            (void) dup(0);
            _exit(execve("/bin/sh",argv,envp));      //执行 shell
        }
        j=wait(&i);
        printf("child %d died with code %04x\n",j,i);
        sync();
        _exit(0);  /* NOTE! _exit, not exit() */
    }
```

4.5 编译和运行 Linux 0.01 系统

本节将介绍如何在 RedHat Linux 7.2 系统上编译本书光盘中附带的 Linux 0.01 的源代码，生成 Linux 0.01 的可执行核心，并且使用生成的 Linux 0.01 核心启动计算机。

如图 4-1 所示，启动 RedH Linux 7.2 后，切换到 root 用户，将光盘中的 Linux 0.01 的源代码文件压缩包文件 Linux 0.01.tar.gz 复制到 root 目录下。使用如下的命令解压源代码：

```
gzip -d Linux0.01.tar.gz
tar xvf Linux0.01.tar.gz
```

图 4-1　解压缩 Linux 0.01 的源代码

执行完上述命令后；可以在 root 目录下生成一个 linux 目录，其中包含了 Linux 0.01 的所有源代码，执行 cd linux 命令，进入 linux 目录，然后执行 make 命令，可以看到如图 4-2 所示的编译过程。编译完成后，使用 ls 命令，可以看到在当前目录中，生成了一个名字为 Image 的文件，这个文件就是编译生成的 Linux 0.01 的二进制核心文件。

图 4-2　Linux 0.01 核心源代码的编译过程

然后，在计算机的 A 驱动器中插入一张格式化了的软盘（默认使用 1.44MB 软盘），执行 make disk 命令，就可以在 A 驱动器的软盘中生成 Linux 0.01 的系统启动磁盘了。在生成的系统启动磁盘中，包含了编译生成的 Linux 0.01 核心 Image，如图 4-3 所示。

图 4-3　生成 Linux 0.01 的引导磁盘

重新启动计算机，在 BIOS 中设置从驱动器 A 启动，可以看到计算机从 A 盘启动，出现图 4-4 所示的 Linux 0.01 启动画面。

图 4-4　Linux 0.01 的启动画面

在图 4-4 所示的启动界面中，单击键盘上的任意键，可以在屏幕上打印出该按键的键值，这表明 Linux 0.01 是在“ RUN”的。

4.6　本章小结

本章是介绍：Linux 0.01 核心的第一个章节，重点是从总体上介绍 Linux 0.01 核心的源代码情况。通过本章的学习，读者可以了解到 Linux 0.01 内核的总体技术特色和源代码分布。

本章还对 Linux 0.01 的 main.c 文件进行了分析，使读者可以全面地了解 Linux 0.01 的启动过程。

本章同时详细介绍了 Linux 0.01 源代码的编译过程和核心启动过程，帮助读者在自己的计算机上编译、运行 Linux 0.01。

第5章 操作系统引导

本章知识点：

- 操作系统引导的基本知识
- 引导扇区和引导代码实例分析
- 使用 nasm 生成引导代码

本章首先介绍了操作系统引导的基本知识，然后分析了 MS-DOS 的引导程序和 Linux 0.01 的引导代码。

为了使读者可以简单地进行引导试验，本章通过使用 debug 程序，详细分析了引导扇区和引导代码的组成，并且使用 debug 程序建立了两个简单的引导代码，使读者可以快速理解引导程序的基本原理。

在本章最后，使用 nasm 编写了一次引导代码和两次引导代码。使用 nasm 编写的系统引导代码，通过简单修改，就可以应用在其他的操作系统引导过程中。

5.1 操作系统引导的基本知识

本节主要介绍操作系统引导的基本知识，包括各种系统引导设备，主引导记录的格式等。这些知识，是读者理解和编写系统引导代码的基础。

5.1.1 系统引导设备

操作系统是从引导设备中引导的。一台计算机要引导，必须首先在 BIOS 中指定引导设备，常见的引导设备是磁盘，包括软盘、硬盘、光盘等。

磁盘（软盘和硬盘）通常需要在格式化后才能使用。格式化的作用就是在磁盘上划分不同的区域来存储不同的数据。通常，磁盘经过格式化后主要包括的区域有：主引导记录区（只有硬盘有）、引导记录区、文件分配表（FAT）、目录区和数据区，如图 5-1 所示。

图 5-1 中各个区域的功能如下：

- 主引导记录区和引导记录区中存有操作系统启动时的所有信息。
- 文件分配表（FAT）是反映当前磁盘扇区使用状况的表，记录了磁盘上的每一个扇区的使用情况。
- 目录区存放磁盘上现有的文件目录及其大小、存放时间等信息。FAT 与目录一

PDF-XCHANGE Click to buy NOW! www.docu-track.com

起对磁盘数据区进行管理。

- 数据区存储和文件名相对应的文件实际的内容数据。

图 5-1　磁盘组成示意图

1. DOS 软盘结构介绍

DOS 是一种常见的操作系统。当使用 DOS 外部命令 format 格式化一张软盘后，format 主要完成两部分工作：

(1) 把磁盘划分为若干磁道，每一磁道划分为若干扇区。

(2) 将划分的扇区分为引导记录区、文件分配表 l、文件分配表 2、根目录区以及数据区 5 大区域。

DOS 系统中软盘只有一个引导区，引导区在磁盘的 0 面 0 道 1 扇区，它的作用是在系统启动时负责把系统两个隐含文件 io.sys 和 msdos.sys 装入内存，并提供 DOS 进行磁盘读写所必需的磁盘 I/O 参数表。

文件分配表 FAT 是反映磁盘上文件占用扇区情况的表格。FAT 表非常重要，如果 FAT 表损坏，那么文件系统的所有数据都将丢失。

为了提高文件系统安全性，DOS 系统在划分磁盘区域时，保留了两份完全相同的文件分配表。根目录区是记载磁盘上所有文件的一张目录登记表，主要记载每个文件的文件名、扩展名、文件属性、文件长度、文件建立日期、建立时间及其他一些重要信息。

注意：DOS 6.22 系统中提供的 scandisk.exe 工具可以修复 FAT 表出现的错误，包括两个 FAT 表不一致的错误。

2. DOS 硬盘结构介绍

硬盘的存储空间通常比较大，为了允许在硬盘空间可以让多个操作系统共享，硬盘空间通常可以划分为主引导记录区和多个系统分区。

硬盘空间的划分总体上可以划分为两个部分：

- 硬盘第 1 扇区，这一扇区也称之为硬盘的主引导程序扇区。
- 各个系统分区，各系统分区可以供不同的操作系统使用。

硬盘第 1 扇区由三部分内容组成：主引导程序、分区信息表和引导扇区标志。主引导程序是硬盘启动时首先执行的程序，由它装入执行活动分区的引导程序，从而进一步引导操作系统。分区信息表登记各个分区引导指示符、操作系统指示符以及该分区占用的硬盘空间的位置及其长度。引导扇区标志一般是两个字节，通过这两个字节标识这个扇区是否是一个合法的引导扇区。

系统分区是提供给各操作系统使用的区域，每一区域只能存放一种操作系统。在该区域中的系统具有自己的引导记录、文件分配表区、文件目录区以及数据区。

在 DOS 操作系统中，硬盘存储空间划分为五个部分：第 1 扇区的主引导程序、分区

信息表、分区引导程序、文件分配表区、文件根目录区和文件数据区。

硬盘主引导扇区很特殊，它不在 DOS 的管辖范围内。所以普通 DOS 命令，例如 format、fdisk、debug 都不能修改它。当该扇区损坏时，硬盘不能启动，用 format、fdisk 也不能修复它。

debug 的 l 命令和 w 命令都不能用于主引导扇区。只有在 debug 下借用 INT 13h 或低级格式化方能修复。

注意：DOS 系统提供的程序 fdisk.exe 可以使用一个特殊的参数：fdisk /mbr 来修复硬盘主引导扇区中的主引导程序（对于分区信息表不能修复）。

3. 光盘结构介绍

1988 年，国际标准化组织（International Standards Organization，ISO）公布了 CD-ROM 文件结构标准，这个标准被称之为 ISO 9660。

ISO 9660 使用的文件结构，不同于普通操作系统的文件系统结构。因此，操作系统并不能直接读取 ISO 9660 格式的光盘。

如果操作系统要读取 ISO 9660 格式光盘，必须添加辅助程序。在 DOS 系统中，这个辅助程序就是 MSCDEX.exe（Microsoft CD-ROM Extension），它需要和 CD-ROM 驱动器带的设备驱动程序相联合，才能使 MS-DOS 操作系统可以读 CD-ROM 盘上的 ISO 9660 文件。MSCDEX.exe 程序的主要功能就是把 ISO 9660 文件结构转变成 MS-DOS 能识别的文件结构。

CD-ROM 的一个物理扇区除了扇区头信息之外还有 2336B。在 2336B 中，有 288B 可以用来作错误检测和校正，剩下的 2048B 作为用户数据域。2048B（2KB）的数据域定义为一个逻辑扇区（logical sector）。每个逻辑扇区都有一个惟一的逻辑扇区号（logical sector number，LSN）。

CD-ROM 盘上可以存放信息的区域称为卷空间（volume space）。卷空间分成两个区：从 LSN 0～LSN 16 称为系统区，它的具体内容没有规定；从 LSN 16 开始到最后一个逻辑扇区称为数据区，它用来记录卷描述符（volume descriptors）、文件目录、路径表、文件数据等内容。

每卷数据区的开头（LSN 16）是卷描述符。卷描述符实际上是一种数据结构，或者说是一种描述表。其中的内容用来说明整个 CD-ROM 盘的结构、提供许多非常重要的信息，如盘上的逻辑组织、根目录地址、路径表的地址和大小、逻辑块的大小等。卷描述符的结构如表 5-1 所示，它是一个由 2048B 组成的固定长度记录。

表 5-1　ISO 9660 格式光盘卷描述符的格式

字节位置	录域的名称
1	卷描述符的类型
2～6	标准卷标识符（用 CD001 表示）
7	卷描述符的版本号

8～2048	取决于卷描述符的类型

卷描述符有 5 种类型：

- 主卷描述符（primary volume descriptor）。
- 辅助卷描述符（supplementary volume deseriptor）
- 卷分割描述符（volume partition descriptor）。
- 引导记录（boot record）。
- 卷描述符系列终止符（volume descriptor set terminator）。

上述 5 种描述符的结构分别如表 5-2～表 5-5 所示。

表 5-2　主卷和辅助卷描述符

字节位置	主卷描述符记录域的名称	辅助卷描述符记录域的名称
1	卷描述符的类型	同左（与：卷描述符的类型，是否相同，如是请把左边的复制过来）
2～6	标准卷标识符（CD001）	同左
7	卷描述符版本号	同左
8	未使用（00）	卷标志
9～40	系统标识符	同左
41～72	卷标识符	同左
73～80	未使用（00）	同左
81～88	卷空间大小	同左
89～120	未使用（00）	换码顺序
121～124	卷系列大小	同左
125～128	卷顺序号	同左
129～132	逻辑块大小	同左
133～140	路径表大小	同左
141～144L	型路径表值位置*	同左
145～148L	型路径表任选值位置	同左
149～152M	型路径表值位置**	同左
153～156M	型路径表任选值位置	同左
157～190	根目录的目录记录	同左
191～318	卷集标识符	同左
319～446	出版商标识符	同左
447～574	数据准备者标识符	同左
575～702	应用软件标识符（如 CD-1）	同左
733～739	版权文件标识符	同左

PDF-XCHANGE Click to buy NOW! www.docu-track.com

740～776	文摘标识符	同左
777～813	文献目录文件标识符	同左
814～830	卷创作日期和时间	同左
831～847	卷修改日期和时间	同左
848～854	卷到期日期和时间	同左
865～881	卷有效日期和时间	同左
882	文件结构版本号	同左
883	（保留）	同左
884～1395	应用程序使用	同左
1396～2048	（保留）	同左

表 5-3　卷分割描述符

字节位置	记录域的名称
1	卷描述符的类型
2～6	标准卷标识符（CD001）
7	卷描述符版本号
8	未使用（00）
9～40	系统标识符
41～72	卷分割标识符
73～80	卷分割位置
81～88	卷分块大小
89～2048	系统使用

表 5-4　引导记录

字节位置	记录域的名称
1	卷描述符的类型
2～6	标准卷标识符（CD001）
7	卷描述符版本号
8～39	引导系统标识符
40～71	引导标识符
72～2048	引导系统使用

表 5-5　卷描述符系列终止符

字节位置	记录域的名称

PDF-XCHANGE Click to buy NOW! www.docu-track.com

1	卷描述符的类型
2～6	标准卷标识符（CD001）
7	卷描述符版本号
8～2048	保留（00）

5 种描述符的前 4 种可以任意组合，组成卷描述符系列。这 4 个描述符可以在描述符系列中出现不只一次。描述符系列有两个限制：主卷描述符至少要出现一次；卷描述符系列终止符只能出现一次，而且只能出现在最后。卷描述符系列记录在从 LSN 16 开始的连续逻辑扇区上。

如果需要进一步了解 CD-ROM 文件系统格式，可以参看 ISO 9660 标准文件。

配置奔腾处理器的计算机通常支持从光盘直接启动计算机。可引导光盘的结构如图 5-2 所示。

图 5-2　可引导光盘的组成结构

图 5-2 可启动 CD-ROM 的工作原理如下：BIOS 首先检查光盘的第 17 个扇区（sector 17），查找其中的代码，若发现其中的启动记录卷描述表（Boot Record Volume Descripter），就根据表中的地址继续查找启动目录（Booting Catalog），找到启动目录后，再根据其中描述的启动入口（Boot Entry）找到相应的启动磁盘镜像（Bootable Disk Image）或启动引导文件，找到启动磁盘镜像后，读取其中的数据，并执行相应的开机动作。相对于单重启动 CD-ROM 而言，多重启动 CD-ROM 的启动目录中包含多个启动入口，指向多个启动磁盘镜像。

关于可启动光盘的更多信息，可以参见本书配套光盘中的文档《Bootable CD-ROM Format Specification》。

5.1.2　启动过程简介

系统的引导是与系统的架构密切相关的，这里只讨论 IA32 架构的系统引导过程。硬

1. 硬盘启动步骤

PC 机启动过程是遵循一定顺序的，下面是硬盘启动的过程：

(1) 机器加电。

(2) BIOS 加电自检（Power On Self Test——POST）。BOIS 在内存中的起始地址为 0FFFF:0000，BOIS 的加电自检就是从地址 0FFFF:0000 开始执行。BIOS 加电自检的主要工作包括：CPU、内存及硬盘等关键设备的检测，即插即用设备的检测。完成这些检查后，将依据检查结果更新 ESCD（Extended System Configuration

PDF-XCHANGE Click to buy NOW! www.docu-track.com

Data，扩展系统配置数据）。

(3) 计算机将硬盘 0 柱面 0 磁头 1 扇区（主引导区 MBR）中的 512B 读人内存 0000:7C00 处并跳到 0000:7C00 处执行；在读取过程中，计算机并不检查该扇区的内容是什么。

(4) 检查（WORD）0000:7DFE 是否等于 0xAA55，若不等于则转去尝试其他启动介质，如果没有其他启动介质则显示“ No ROM BASIC”，然后死机。

(5) 跳转到 0000:7C00 处执行 MBR 中的程序。

(6) MBR 首先将自己复制到 0000:0600 处，然后继续执行。

(7) MBR 在主分区表中搜索标志为活动的分区，如果发现没有活动分区或有不止一个活动分区，则停止。

(8) 将活动分区的第一个扇区读入内存地址 0000:7C00 处。

(9) 检查（WORD）0000:7DFE 是否等于 0xAA55，若不等于则显示“ Missing Operating System”然后停止，或尝试软盘启动。

(10) 跳转到 0000:7C00 处继续执行特定系统的启动程序。

(11) 启动系统。

以上步骤中，第(2)、(3)、(4)、(5)步是由 BIOS 的引导程序完成，第(6)、(7)、(8)、(9)、(10)步由 MBR 中的引导程序完成。'

现在操作系统很多都支持多系统引导程序（如 LILO、NTLoader 等），这些多系统引导程序是将自己的引导程序放在系统所处分区的第 1 个扇区中，然后在启动时，让用户选择要启动的分区。

2. 软盘启动步骤

软盘的启动步骤如下：

(1) 机器上电。

(2) BIOS 加电自检（Power On Self Test——POST）。BOIS 在内存中的起始地址为 0FFFF:0000，BOIS 的加电自检就是从地址 0FFFF:0000 开始执行。BIOS 加电自检的主要工作包括：CPU、内存及硬盘等关键设备的检测，即插即用设备的检测。完成这些检查后，将依据检查结果更新 ESED（Extended System Configuration Data，扩展系统配置数据）。

(3) 计算机将 A 盘 0 磁道 0 磁头 1 扇区的内容读入内存 0000:7C00 处并跳到 0000:7C00 处执行。在读取过程中，计算机并不检查该扇区的内容是什么。

(4) 检查（WORD）0000:7DFE 是否等于 0xAA55，若不等于则转去尝试其他启动介质，如果没有其他启动介质则显示“ No ROM BASIC”然后死机。

(5) 跳转到 0000:7C00 处执行 MBR 中的程序。

(6) 启动系统。

3. 实验：Reboot 程序

在前面介绍的硬盘和软盘启动过程的第（2）步中，BOIS 的起始地址是 0FFFF:0000。

在程序设计中，如果需要执行热启动计算机，可以通过将程序的执行跳转到 0FFFF:0000 来实现。下面的程序 reboot.c 就可以实现热启动的功能。reboot.c 的代码如下：

```
#include <stdio.h>

void reboot()
{
    asm{
        mov ax,0ffffh
        push ax
        xor ax,ax
        push ax
        retf
    }
}

void main()
{
    printf("Please press any key to reboot...\n");
    getch();
    reboot();
}
```

reboot.c 的执行效果如图 5-3、图 5-4 所示。

图 5-3　reboot.c 的执行（重启动前）

图 5-4　reboot.c 的执行(重启动后)

5.1.3　硬盘主引导扇区简介

大多数操作系统都应该支持从硬盘启动。本节将详细地介绍硬盘主引导扇区的基本结构和主要数据，并且将分析 DOS 硬盘主引导扇区的引导代码。

1.　硬盘主引导扇区基本结构

主引导扇区是硬盘的第 1 个扇区，它由主引导记录(MBR，Master Boot Record)、硬盘分区表(DPT，Disk Partition Table)和引导扇区标识(Boot Record ID)三部分组成。

硬盘的一个扇区是 512B。主引导记录占用主引导扇区的前 446B(字节编号是 0～0x1BD)存放系统主引导程序(它负责从活动分区中装载并运行操作系统引导程序)。

硬盘分区表占用 64B（字节编号是 0x1BE～0x1FD），记录了磁盘的基本分区信息。硬盘分区表分为 4 个分区项，每项 16B，分别记录了每个主分区的信息（一个硬盘因此最多可以有 4 个主分区）。

引导扇区标识占用两个字节（字节序号是从 0x1FE～0x1FF）。对于合法引导区，这个标识为 0xAA55。这通常是计算机判断引导扇区是否合法的标志。

注意：一些计算机病毒就是改写了这两个字节的标识，从而使计算机无法识别主引导扇区，造成不能引导操作系统。

硬盘主引导扇区的具体结构如表 5-6 所示。

表 5-6　硬盘主引导扇区结构

偏移量	内容说明	大小
000h	执行代码（启动计算机用）	446Bytes
1BEh	第一个分区入口	16Bytes
1CEh	第二个分区入口	16Bytes
1DEh	第三个分区入口	16Bytes
1EEh	第四个分区入口	16Bytes
1FEh	MBR 分区标志（55hAAh）	2Bytes

主引导记录中包含了硬盘的一系列参数和一段引导程序。引导程序主要是用来在系统硬件自检完后引导具有激活标志分区上的操作系统，它最后执行的指令是一条 JMP 指令，这条指令使计算机可以跳转执行操作系统本身的引导程序。

注意：最后的这条 JMP 跳转语句往往是引导型病毒的注入点，也是各种多系统引导程序（例如 LIL0 和 GRUB）的注入点。

由于引导程序最多只能有 446B，因此引导程序本身必须是非常简单的。通常，可以通过判断这段引导程序的合法性（看 JMP 指令的合法性）来判断系统是否被注入了引导型病毒。DOS 命令 fdisk /mbr 也可以修复这段程序。

主引导记录后是硬盘分区表。硬盘分区表由 4 个 16B 的分区信息表组成。每个信息表的结构如表 57 所示。

表 5-7　硬盘分区表结构

偏移量	内容说明	大小
00h	分区状态（00h=非活动 Inactive，80h=活动 Active）	1Byte

01h	分区开始的磁头数	1Byte
02h	分区开始的柱面和扇区数	1Word
04h	分区类型	1Byte
05h	分区结束的磁头数	1Byte
06h	分区结束的柱面和扇区数	1Word
08h	第一个分区和 MBR 区之间的扇区总数，即隐含扇区总数	1DoubleWord
0Ch	分区中的扇区总数	1DoubleWord

在上表的 04h 分区类型的取值如表 5-8 所示。

引导扇区的最后两个字节是“ 55AA”，这是引导扇区结束标志。如果这两个字节被修改(有些病毒就会修改这两个标志)，则系统引导将会失败(通常会报告找不到分区表)。

在 DOS 下面可以执行 fdisk /mbr，可以把主引导分区里的 MBR 部分重新写过，而不会对 DPT 有任何破坏。

表 5-8　硬盘分区类型

数值	分区类型
00h	未知类型或不用
01h	12BIT 的 FAT 格式
04h	16BIT FAT 表，磁盘空间小于 32M
05h	扩展 MS_DOS 分区
06h	16BIT 空间大于 32MB
0Bh	使用 32BIT FAT 表，磁盘空间最大 2GB
0Ch	使用 32BIT FAT 表，采用 LBA 模式，调用 INT 13 扩展中断
0Eh	使用 16BIT FAT 表，调用 INT 13h 扩展中断
0Fh	扩展 MS_DOS 分区，采用 LBA 模式，调用 INT 13h 扩展中断
07h	NTFS 分区（New Technology File System）
82h	Linux 分区
83h	Linux 交换分区

在 Linux 系统中可以执行如下的命令来恢复 MBR：

```
dd if=/dev/had of=/boot/boot.NNNN bs=446 count=1      # 备份 MRB
dd if=/boot/boot.NNNN of=/dev/had bs=446 count=1      # 恢复 MBR
```

其中 bs（buffer size）是指重写的字节数。注意，这个字节数是 446，而不是 512。因为 MBR 是主引导扇区的前 446 B。如果在写 MBR 的时候，超过了 446B，就会覆盖到后面紧接的 DPT（硬盘分区表)。如果破坏了硬盘分区表，那么硬盘上的数据可能全部丢失。

2.　硬盘主引导扇区的应用

利用硬盘主引导扇区在系统引导中的特殊作用，可以完成很多特殊的功能操作，例如：

- 清除硬盘引导功能

由于硬盘引导必须使用引导程序，并检测活动分区的正确性，所以人为地修改或破坏引导程序部分，或者清除活动分区引导标志，都将使硬盘无法启动。

- 加密整个硬盘

硬盘主引导扇区末尾的扇区有效标志 AA55h 是系统承认硬盘的前提，所以可以采取清除或修改此标志位，从而可以达到加密硬盘的目的。在这种情况下，即使从 A 驱引导系统也无法对硬盘进行操作。恢复标志字节 AA55h 后即可解密硬盘。

- 加密单个硬盘分区

硬盘单个分区的加密可采取修改分区类型的方法，比如把扩展 DOS 分区的类型标志 05h 改为 FFh，则 DOs 认为此分区为非 DOS 分区，无法对其进行访问，包括此分区中的所有逻辑盘。也可以通过修改或清除某一分区表的所有数据来加密这个分区，在恢复分区表数据后，就可以访问该分区数据。

- 加入硬盘启动口令识别

通过修改硬盘的主引导程序，在引导 DOS 操作系统之前，加入一段口令识别程序段，如口令正确则正常引导系统，否则拒绝引导，达到口令识别的目的。

- 先于 DOS 驻留内存程序

在主引导程序中安装某些中断服务程序，如时钟中断，通过对 INT 21H 或其他 DOS 关键数据的监视，完成病毒的实时检测功能。因为此方法在引导 DOS 系统之前完成，所以其监视效果非常可靠。

- 实现同一硬盘多个操作系统的选择启动

硬盘可以分成 4 个独立的分区，装入 4 个不同的操作系统，通过特殊的方法可以共享多个 DOS 版本。LIL0、GRUB 等程序就是利用了这样的原理从而实现多操作系统引导。

- 实现硬盘主引导扇区或 DOS 引导扇区的自我修复

如果在硬盘其他空闲扇区保留一份完好的主引导扇区内容，而在主引导程序中每次启动前进行主引导扇区的正确性检查工作，当发现异常时，即调入原来完好的主引导扇区内容，就可以及时发现和清除病毒，对于系统有很好的保护作用。

5.1.4 软盘主引导扇区

软盘主引导扇区位于软盘的 0 磁道 0 头 1 扇区。这个扇区是由 DOS 的软盘格式化程序创建的，例如 DOS 的 format 程序。

在系统引导的时候，BIOS 会读取软盘的主引导扇区到内存地址 0000:7COO 处，然后跳转到 0000:7C00 处开始执行。本节将详细介绍软盘主引导扇区的结构和数据。

1. 软件主引导扇区结构

软盘主引导扇区的具体结构如表 5-9 所示。

表 5-9　软盘主引导扇区结构

偏移量	内容说明	大小
000h	BIOS Parameter Block（BPB）	62Bytes
03Eh	引导扇区引导代码	352Bytes
19Eh	引导扇区引导代码定义的消息	72Bytes
1E6h	DOS 隐藏文件名字	24Bytes
1FEh	软盘 MBR 分区标志（55h AAh）	2Bytes

2. DOS 引导程序介绍

DOS 的引导程序主要完成如下的功能：

- 复制 INT 1E 所指向的 DPT（Diskette Parameter Table）。
- 改变 DPT 的复制。
- 修改 INT 1E，指向修改后的 DPT。
- 调用 INT 13h AH=00，disk reset call。
- 计算根目录（root directory）的扇区地址。
- 读取根目录的第 1 个扇区到地址 0000:0500。
- 确认根目录第 1 个扇区中的头两个目录项是文件：io.sys 和 msdos.sys。
- 读取文件 io.sys 的头 3 个扇区到地址 0000:0700。
- 初始化一些寄存器，然后跳转到地址 0070:0000 处的 io.sys 文件中。

DOS 引导程序使用基于 CHS 的 BIOS 调用 INT 13h AH=02 来读取软盘 FAT 表中的根目录，然后从根目录中获取 io.sys 文件的信息，从而读取 io.sys 文件。

3. 完整的软盘引导扇区例子（MSDOS 5.0）

下面是一张 MS-DOS 5.0 软盘的引导扇区的 16 进制和 ASCII 表。从这张表中，可以看到 MS-DOS 5.0 的系统标志和引导扇区的分区标志 0x55aa。

OFFSET0。123456789ABCDEF*0123456789ABCDEF*
000000eb3e904d53444f53352e300002010100*…Ms[）OS5，0，，，，，”
00001002e00040Obf00900120002000000000*，，，@……，，…，*
0000200000000000000295a5418264e4f204e41*，…，，）ZT，，NONA*
0000304d45202020204641543132202020fa33*MEFATl2，3*
000040c08edObc007c1607bb780036c5371e56*，…，1，，，x，6，7，V”
0000501653bf3e7cb90bOOfcf3a406lfc645fe*，S，，1，，……，E，*
0000600f8boel87e884dr9894702c7073e7cfb*…，1，M，，G，，，，1，，-N-
000070cdl3727933c03906137c74088bOel37e*，，ry3，9，，It，，，，1*。
000080890e207ca0107ef726167c03061c7c13*，，I，，I…I，，，1，*■
000090161e7C03060e7e83d200a3507e891652*，，I…I…，P『，，R*一

j0000aO7ca3497c8916467eb82000f726117c8b*1，11，，KI，，，，，1，-N-j 曩
0000bOleob7c03c348f7f30106497c83164b7c*，，卜，H，，，，IJ，，Kl*_ 一 …
0000CO00bb00058b16527ca1507ce89200721d*，，，，，，R 卜 P 卜，，r，*笺
i0000dObOOle8ae0072168bfbb90b00bee67df3*，，…r，…，，，，}，*誉誉
0000COa6750a8d7f20bgOb00f3a67418be9e7d*，u … … ，，t … }*gi 蓦
0000fOe85f0033cOcdl65elf8f048f4402cdl9*，，3 … ‘ … ，D …
*i0000100585858ebe88b471a48488ale0d7c32ff*XXX，，，G，HH，，，12，*蠢 j 誉
000110f7e30306497c13164b7cbb0007b90300*…，11，，Kl … … * 一 0 蕊
000120505251e83a0072d8bOOle85400595a58*PRQ，：，r…，T，YZX*i 惹
i000013072hb05010083d200031eOb7ee_2e28a2e*r，…，，，，，，I…，*鬟 0 —
000140157c8a16247c8ble497cal4b7cea0000*，I，，$1，，11，KI … * 糕 0 蠢
0001507000acoaC07429b40ebb0700cdloebf7 ” P，… t ） … … … * 鬟 瑟
0i0001603b16187e7319f736187efec288164f7c 。；，，IS，，6，1，… 0I* 囊 鬻
0-00017033d2f736la7c8816257ea34d7cf8c3f9*3，，6，I，，%1，Ml…*‘鬟 g 鏊簪_鬟鬟 0jg
鬃蠹 0000，80-Linux0，01 内核分析与操作系统设计

000180c3b4028b164d7cbl06d2e60a364f7c8b*，，，，，MJ…，，60I，*
000190ca86e98a16247c8a36257ccd13c30doa*……$1，6%I，，，，，*
0001aO4e6f6e2d53797374656d206469736b20*Non—Systemdisk*
0001bO6f72206469736b206S12126f720dOa520ordiskerror，，R*
0001cO65706c61636520616e64207072657373 ” eplaceandpress41"
0001dO20616e79206h6579207768656e207265*anykeywhenre*
0001e06164790d0a00494f2020202020205359*adv…，IOSY ”
0001fO534d53444f5320202053595300000 55aa*SMs[）0SSYS—U，*

4. 软盘 BPB（BIOS 参数块）和数据区

软盘引导扇区的头 62B 是软盘的 BIOS 参数块（BIOS Parameter Block，BPB）。表 5-10 是 BPB 中各项的含义。

表 5-10 软盘的 BIOS 参数块

存储单位	含义	加载到内存的地址	字节数	取值
db	JMP instruction	7C00	2	EB3C
db	NOP instmction	7C02	1	90
db	OEMname	7C03	8	'MSDOS5.0'
dw	bytesPerSector	7C0B	2	0200
db	sectPerCluster	7C0D	1	01
dw	reservedSectors	7C0E	2	0001
db	numFAT	7C10	1	02
dw	numRootDirEntries	7C11	2	00e0

dw	numSectors	7C13	2	0B40（ignore numSectomHuge）
db	mediaType	7C15	1	F0
dw	numFATsectors	7C16	2	0009
dw	sectorsPerTrack	7C18	2	0012
dw	numHeads	7ClA	2	0002
dd	numHiddenSectors	7C1C	4	00000000
dd	numSectorsHuge	7C20	4	00000000
db	driveNum	7C24	1	00
db	reserved	7C25	1	00
db	signature	7C26	1	29
dd	volumeID	7C27	4	5A541826
db	volumeLabel	7C2B	11	'NONAME'
db	fileSysType	7C36	8	'FAT12'

在表 5-10 中，BPB 的前 3 个字节是两个机器指令：JMP（跳转操作）和 NOP（无操作指令，用于占用总线时间，或者占用存储空间），这两个指令在内存中的分布如下所示：

0000：7C00FA33CJMPSTART

0000：7C0290NOP

内存中剩下的 BPB 如下所示：

0000：7C00……4d53444f53352e300002010100*M 唧 S5，0…，，*

0000：7C1002e000400bf00900120002000000000*，，，@……，，，*

0000：7C20000000000000295a5418264e4f204e41*，，…，）ZT，&NONA*

0000：7C304d45202020204641543132202020…，*M 畦 FATl2*

5. INT 1E 中断和 DPT（磁盘参数表）

在内存地址 0000:7C3E 开始的 11 个字节是磁盘参数表（Diskette Parameter Table）。磁盘参数表是由 INT 1E 所指向的，各个参数如下所示：

- 7C3E=步进速度和磁头停止时间（Step rate and head unload time）。
- 7C3F=磁头运动时间和 DMA 模式标志（Head load time and DMA mode flag）。
- 7C40=驱动器马达关闭延迟时间（Delay for motor turn off）。
- 7C41=每扇区字节数（Bytes per sector）。
- 7C42=每磁道扇区数（Sectors per track，）。
- 7C43=扇区间隔长度（Intersector gap length）。
- 7C44=数据长度（Data length）。
- 7C45=在 format 时，扇区间隔长度（Intersector gap length during format）。
- 7C46=格式化字节值（Format byte value）。
- 7C47=磁头潜伏时间（Head settling time）。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

- 7C48=驱动器马达到达正常速度的延迟（Delay until motor at normal speed）。

从地址 0000:7C49 开始的 11 个字节代表了如下的数据：

- 7C249～7C4C=数据区的磁盘扇区地址（类似 LBA）（diskette sector address（as LBA） of the data area）。
- 7C4D～7C4E=读取的柱面数（cylinder number to read from）。
- 7C4F～7C4F=读取的扇区数（sector number to read from）。
- 7C50～7C53=根目录的磁盘扇区地址（类似 LBA）。

5.1.5 Linux 0.01 引导代码分析

本节将介绍 Linux 0.01 的引导部分代码。Linux 0.01 的引导部分主要包括两个源文件：boot.s 和 head.s。boot.s 是计算机上电启动时，由 BIOS 执行的系统引导代码。Head.s 是 Linux 的 32 位引导代码，在其中将调用 main()，最后完成操作系统的引导。

1. boot.s：BIOS 引导代码

在计算机加电时，boot.s 被 BIOS 的引导程序加载到地址 0x7C00 处，然后 boot.s 将自己移动到地址 0x90000 处，然后跳转到地址 0x90000 处开始执行。

然后，boot.s 使用 BIOS 中断功能，加载系统核心到地址 0x10000 处。接下来，boot.s 禁止所有中断，移动系统核心到地址 0x0000 处，切换处理器到保护模式，然后调用系统的开始程序。系统核心开始运行后，将重新设置保护模式，然后打开需要的中断。boot.s 的代码分析如下：

```
|
|     boot.s
|
| boot.s is loaded at 0x7c00 by the bios-startup routines, and moves itself
| out of the way to address 0x90000, and jumps there.
|
| It then loads the system at 0x10000, using BIOS interrupts. Thereafter
| it disables all interrupts, moves the system down to 0x0000, changes
| to protected mode, and calls the start of system. System then must
| RE-initialize the protected mode in it's own tables, and enable
| interrupts as needed.
|
| NOTE! currently system is at most 8*65536 bytes long. This should be no
| problem, even in the future. I want to keep it simple. This 512 kB
| kernel size should be enough - in fact more would mean we'd have to move
| not just these start-up routines, but also do something about the cache-
| memory (block IO devices). The area left over in the lower 640 kB is meant
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
| for these. No other memory is assumed to be "physical", ie all memory
| over 1Mb is demand-paging. All addresses under 1Mb are guaranteed to match
| their physical addresses.
|
| NOTE1 abouve is no longer valid in it's entirety. cache-memory is allocated
| above the 1Mb mark as well as below. Otherwise it is mainly correct.
|
| NOTE 2! The boot disk type must be set at compile-time, by setting
| the following equ. Having the boot-up procedure hunt for the right
| disk type is severe brain-damage.
| The loader has been made as simple as possible (had to, to get it
| in 512 bytes with the code to move to protected mode), and continuos
| read errors will result in a unbreakable loop. Reboot by hand. It
| loads pretty fast by getting whole sectors at a time whenever possible.

| 1.44Mb disks:
sectors = 18
| 1.2Mb disks:
| sectors = 15
| 720kB disks:
| sectors = 9

.globl begtext, begdata, begbss, endtext, enddata, endbss
.text
begtext:
.data
begdata:
.bss
begbss:
.text

BOOTSEG = 0x07c0
INITSEG = 0x9000
SYSSEG    = 0x1000                  | system loaded at 0x10000 (65536).
ENDSEG= SYSSEG + SYSSIZE

entry start
start:
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
        mov ax,#BOOTSEG
        mov ds,ax
        mov ax,#INITSEG
        mov es,ax
        mov cx,#256
        sub  si,si
        sub  di,di
        rep
        movw
        jmpi go,INITSEG
go:     mov ax,cs
        mov ds,ax
        mov es,ax
        mov ss,ax
        mov sp,#0x400               | arbitrary value >>512

        mov ah,#0x03 | read cursor pos
        xor  bh,bh
        int  0x10

        mov cx,#24
        mov bx,#0x0007     | page 0, attribute 7 (normal)
        mov bp,#msg1
        mov ax,#0x1301     | write string, move cursor
        int  0x10

| ok, we've written the message, now
| we want to load the system (at 0x10000)

        mov ax,#SYSSEG
        mov es,ax               | segment of 0x010000
        call  read_it
        call  kill_motor

| if the read went well we get current cursor position ans save it for
| posterity.

        mov ah,#0x03 | read cursor pos
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
        xor   bh,bh
        int   0x10              | save it in known place, con_init fetches
        mov [510],dx  | it from 0x90510.

| now we want to move to protected mode ...

        cli                     | no interrupts allowed !

| first we move the system to it's rightful place

        mov ax,#0x0000
        cld                     | 'direction'=0, movs moves forward
do_move:
        mov es,ax               | destination segment
        add  ax,#0x1000
        cmp ax,#0x9000
        jz    end_move
        mov ds,ax               | source segment
        sub  di,di
        sub  si,si
        mov       cx,#0x8000            //每 64K 字节循环一次
        rep
        movsw
        j      do_move

| then we load the segment descriptors

end_move:

        mov ax,cs               | right, forgot this at first. didn't work :-)
        mov ds,ax
        lidt   idt_48           | load idt with 0,0
        lgdt  gdt_48            | load gdt with whatever appropriate

| that was painless, now we enable A20        //A20 门是为了访问 1MB 以上的 64K 内存

        call  empty_8042
        mov al,#0xD1        | command write
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        out   #0x64,al
        call  empty_8042
        mov al,#0xDF        | A20 on
        out   #0x60,al
        call  empty_8042

| well, that went ok, I hope. Now we have to reprogram the interrupts :-(
| we put them right after the intel-reserved hardware interrupts, at
| int 0x20-0x2F. There they won't mess up anything. Sadly IBM really
| messed this up with the original PC, and they haven't been able to
| rectify it afterwards. Thus the bios puts interrupts at 0x08-0x0f,
| which is used for the internal hardware interrupts as well. We just
| have to reprogram the 8259's, and it isn't fun.

        mov al,#0x11        | initialization sequence
        out   #0x20,al      | send it to 8259A-1
        .word     0x00eb,0x00eb             | jmp $+2, jmp $+2        //需要四个初始化字节
        out   #0xA0,al      | and to 8259A-2
        .word     0x00eb,0x00eb
        mov al,#0x20        | start of hardware int's (0x20)
        out   #0x21,al
        .word     0x00eb,0x00eb
        mov al,#0x28        | start of hardware int's 2 (0x28)
        out   #0xA1,al
        .word     0x00eb,0x00eb
        mov al,#0x04        | 8259-1 is master          //接通 8259A-1
        out   #0x21,al
        .word     0x00eb,0x00eb
        mov al,#0x02        | 8259-2 is slave
        out   #0xA1,al
        .word     0x00eb,0x00eb
        mov al,#0x01        | 8086 mode for both            //80X86
        out   #0x21,al
        .word     0x00eb,0x00eb
        out   #0xA1,al
        .word     0x00eb,0x00eb
        mov al,#0xFF        | mask off all interrupts for now          //关闭全部中断
        out   #0x21,al
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        .word   0x00eb,0x00eb
        out   #0xA1,al

    | well, that certainly wasn't fun :-(. Hopefully it works, and we don't
    | need no steenking BIOS anyway (except for the initial loading :-).
    | The BIOS-routine wants lots of unnecessary data, and it's less
    | "interesting" anyway. This is how REAL programmers do it.
    |
    | Well, now's the time to actually move into protected mode. To make
    | things as simple as possible, we do no register set-up or anything,
    | we let the gnu-compiled 32-bit programs do that. We just jump to
    | absolute address 0x00000, in 32-bit protected mode.

        mov ax,#0x0001    | protected mode (PE) bit          //设置 PE=1，准备进入保护
模式
        lmsw    ax        | This is it!                      //lmsw 是指装入机器状态字，即
实际进入保护模式
    //进入保护模式后，段寄存器就变成了选择子，选择子的结构：
    //          16              2       1   0
    //          ————————————————————————————
    //          |     索引        | TI    | RDL |
    //          ————————————————————————————
    //其中 TI=0 时是从 GDT 中找描述符

        jmpi 0,8        | jmp offset 0 of segment 8 (cs)    //选择子=8，跳到 GDT 中的索引
号为 1 的描述符，即代码段

    | This routine checks that the keyboard command queue is empty
    | No timeout is used - if this hangs there is something wrong with
    | the machine, and we probably couldn't proceed anyway.
    empty_8042:
        .word   0x00eb,0x00eb
        in    al,#0x64  | 8042 status port
        test  al,#2     | is input buffer full?
        jnz   empty_8042    | yes - loop
        ret

    | This routine loads the system at address 0x10000, making sure
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
| no 64kB boundaries are crossed. We try to load it as fast as
| possible, loading whole tracks whenever we can.
|
| in:  es - starting address segment (normally 0x1000)
|
| This routine has to be recompiled to fit another drive type,
| just change the "sectors" variable at the start of the file
| (originally 18, for a 1.44Mb drive)
|
sread:  .word 1                 | sectors read of current track
head:   .word 0                 | current head
track:  .word 0                 | current track
read_it:
        mov ax,es
        test ax,#0x0fff
die:  jne die                   | es must be at 64kB boundary
        xor bx,bx       | bx is starting address within segment
rp_read:
        mov ax,es
        cmp ax,#ENDSEG          | have we loaded all yet?
        jb ok1_read
        ret
ok1_read:
        mov ax,#sectors
        sub ax,sread
        mov cx,ax
        shl cx,#9
        add cx,bx
        jnc ok2_read
        je ok2_read
        xor ax,ax
        sub ax,bx
        shr ax,#9
ok2_read:
        call read_track
        mov cx,ax
        add ax,sread
        cmp ax,#sectors
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
        jne ok3_read
        mov ax,#1
        sub ax,head
        jne ok4_read
        inc track
ok4_read:
        mov head,ax
        xor ax,ax
ok3_read:
        mov sread,ax
        shl cx,#9
        add bx,cx
        jnc rp_read
        mov ax,es
        add ax,#0x1000
        mov es,ax
        xor bx,bx
        jmp rp_read

read_track:
        push ax
        push bx
        push cx
        push dx
        mov dx,track
        mov cx,sread
        inc cx
        mov ch,dl
        mov dx,head
        mov dh,dl
        mov dl,#0
        and dx,#0x0100
        mov ah,#2
        int 0x13
        jc bad_rt
        pop dx
        pop cx
        pop bx
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
        pop ax
        ret
bad_rt:     mov ax,#0
        mov dx,#0
        int 0x13
        pop dx
        pop cx
        pop bx
        pop ax
        jmp read_track

/*
  * This procedure turns off the floppy drive motor, so
  * that we enter the kernel in a known state, and
  * don't have to worry about it later.
  */
kill_motor:
        push dx
        mov dx,#0x3f2
        mov al,#0
        outb
        pop dx
        ret

gdt:
        .word   0,0,0,0         | dummy

        .word   0x07FF          | 8Mb - limit=2047 (2048*4096=8Mb)
        .word   0x0000          | base address=0
        .word   0x9A00          | code read/exec
        .word   0x00C0          | granularity=4096, 386

        .word   0x07FF          | 8Mb - limit=2047 (2048*4096=8Mb)
        .word   0x0000          | base address=0
        .word   0x9200          | data read/write
        .word   0x00C0          | granularity=4096, 386

idt_48:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        .word   0               | idt limit=0
        .word   0,0             | idt base=0L

gdt_48:
        .word   0x800           | gdt limit=2048, 256 GDT entries
        .word   gdt,0x9         | gdt base = 0X9xxxx

msg1:
        .byte 13,10
        .ascii "Loading system ..."
        .byte 13,10,13,10

.text
endtext:
.data
enddata:
.bss
endbss:
```

2. head.s：Linux 的 32 位引导代码，

head.s 是 Linux 0.01 的 32bit 引导代码，这段代码被加载到地址 0x00000000 处开始运行。地址 0x00000000 也是系统页表的地址，当系统完成引导后，这段地址将被系统页表重新覆盖使用。

```
/*
 *  head.s contains the 32-bit startup code.
 *
 * NOTE!!! Startup happens at absolute address 0x00000000, which is also where
 * the page directory will exist. The startup code will be overwritten by
 * the page directory.
 */
.text
.globl _idt,_gdt,_pg_dir
_pg_dir:
startup_32:
//进入保护模式后，段寄存器就变成了选择子，选择子的结构：
//          16              2       1   0
//          ---------------------------
//          |     索引        | TI    | RDL |
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
//          ---------------------------
//其中 TI=0 时是从 GDT 中找描述符
        movl $0x10,%eax         //选择子=10，即索引为 2，是 GDT 中第三个描述符，是 DATA 描述符
        mov %ax,%ds             //ds es fs gs 都指向 DATA 描述符
        mov %ax,%es
        mov %ax,%fs
        mov %ax,%gs
        lss _stack_start,%esp   //ds 送 SS，esp 指向_stack_start（在 sched.c 中定义）
        call setup_idt
        call setup_gdt
        movl $0x10,%eax         # reload all the segment registers
        mov %ax,%ds         # after changing gdt. CS was already
        mov %ax,%es         # reloaded in 'setup_gdt'
        mov %ax,%fs
        mov %ax,%gs
        lss _stack_start,%esp
        xorl %eax,%eax
1:      incl %eax       # check that A20 really IS enabled
//0X100000 是 1 兆处，如果 A20 打开，那么 0X000000 和 0X100000 的值不一样，如果未打开，两处一样。
        movl %eax,0x000000
        cmpl %eax,0x100000
        je 1b
        movl %cr0,%eax          # check math chip
        andl $0x80000011,%eax# Save PG,ET,PE
        testl $0x10,%eax                //测试协处理器，ET=1 是 387 或 487 浮点协处理器
        jne 1f                  # ET is set - 387 is present
        orl $4,%eax             # else set emulate bit      //模拟协处理器
1:      movl %eax,%cr0
        jmp after_page_tables


/*
 *  setup_idt
 *
 *  sets up a idt with 256 entries pointing to
 *  ignore_int, interrupt gates. It then loads
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 *	idt. Everything that wants to install itself
 *	in the idt-table may do so themselves. Interrupts
 *	are enabled elsewhere, when we can be relatively
 *	sure everything is ok. This routine will be over-
 *	written by the page tables.
 */
setup_idt:
//描述符的一般格式：
//	7                            0
//	  ------------------------
//	0 |          7-0 偏移          |
//	  ------------------------
//	1 |         15-8 偏移          |
//	  ------------------------
//	2 |       选择子低 8 位          |
//	  ------------------------
//	3 |       选择子高 8 位          |
//	  ------------------------
//	4 | 0 | 0 | 0 |     字计数     |
//	  ------------------------
//	5 | P |   DPL   | 0 |   类型   |
//	  ------------------------
//	6 |         23-16 偏移           |
//	  ------------------------
//	7 |         31-24 偏移           |
//	  ------------------------

	lea ignore_int,%edx
//	7                            0
//	  ------------------------
//	0 |     ignore_int 的          |
//	  ------------------------
//	1 |       偏移的低 16 位         |
//	  ------------------------
//	2 |         0         |        0       |
//	  ------------------------
//	3 |      0            |         8      |
//	  ------------------------
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
//     4 |         0          |         0        |
//         -------------------------
//     5 |  1    0  0  0    |       E          |
//         -------------------------
//     6 |      ignore_int 的              |
//         -------------------------
//     7 |      偏移的高 16 位                |
//         -------------------------
        movl $0x00080000,%eax
        movw %dx,%ax            /* selector = 0x0008 = cs */
        movw $0x8E00,%dx        /* interrupt gate - dpl=0, present */

        lea _idt,%edi
        mov $256,%ecx
rp_sidt:
        movl %eax,(%edi)
        movl %edx,4(%edi)
        addl $8,%edi
        dec %ecx
        jne rp_sidt
        lidt idt_descr
        ret

/*
 *    setup_gdt
 *
 *    This routines sets up a new gdt and loads it.
 *    Only two entries are currently built, the same
 *    ones that were built in init.s. The routine
 *    is VERY complicated at two whole lines, so this
 *    rather long comment is certainly needed :-).
 *    This routine will beoverwritten by the page tables.
 */
setup_gdt:
        lgdt gdt_descr
        ret

.org 0x1000
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
pg0:

.org 0x2000
pg1:

.org 0x3000
pg2:        # This is not used yet, but if you
            # want to expand past 8 Mb, you'll have
            # to use it.

.org 0x4000
after_page_tables:                  //将内核开始函数 main 及其参数推入堆栈
        pushl $0        # These are the parameters to main :-)
        pushl $0
        pushl $0
        pushl $L6           # return address for main, if it decides to.
        pushl $_main
        jmp setup_paging
L6:
        jmp L6              # main should never return here, but
                    # just in case, we know what happens.

/* This is the default interrupt "handler" :-) */
.align 2
ignore_int:
        incb 0xb8000+160        # put something on the screen
        movb $2,0xb8000+161         # so that we know something
        iret                    # happened


/*
 * Setup_paging
 *
 * This routine sets up paging by setting the page bit
 * in cr0. The page tables are set up, identity-mapping
 * the first 8MB. The pager assumes that no illegal
 * addresses are produced (ie >4Mb on a 4Mb machine).
 *
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * NOTE! Although all physical memory should be identity
 * mapped by this routine, only the kernel page functions
 * use the >1Mb addresses directly. All "normal" functions
 * use just the lower 1Mb, or the local data space, which
 * will be mapped to some other place - mm keeps track of
 * that.
 *
 * For those with more memory than 8 Mb - tough luck. I've
 * not got it, why should you :-) The source is here. Change
 * it. (Seriously - it shouldn't be too difficult. Mostly
 * change some constants etc. I left it at 8Mb, as my machine
 * even cannot be extended past that (ok, but it was cheap :-)
 * I've tried to show which constants to change by having
 * some kind of marker at them (search for "8Mb"), but I
 * won't guarantee that's all :-( )
 */
.align 2
setup_paging:
        movl $1024*3,%ecx                   //3 个 4K
        xorl %eax,%eax
        xorl %edi,%edi                      /* pg_dir is at 0x000 */
        cld;rep;stosl                       //stosl 将 eax 中的值传入 edi 中
    //1024 个页目录项，这里只申明了 2 个（pg0   pg1)页表，每个页表寻址 4MB，并且
将每个页表项的低 4 位设为 7，是说明页表有效
        movl $pg0+7,_pg_dir         /* set present bit/user r/w */
        movl $pg1+7,_pg_dir+4       /*    --------- " " --------- */
        movl $pg1+4092,%edi
        movl $0x7ff007,%eax         /*    8Mb - 4096 + 7 (r/w user,p) */
        std
    //下面的代码是反向填充 PG0、PG1
1:      stosl           /* fill pages backwards - more efficient :-) */
        subl $0x1000,%eax
        jge 1b
        xorl %eax,%eax          /* pg_dir is at 0x0000 */
        movl %eax,%cr3          /* cr3 - page directory start */
        movl %cr0,%eax
        orl $0x80000000,%eax
        movl %eax,%cr0          /* set paging (PG) bit */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        ret                 /* this also flushes prefetch-queue */

.align 2
.word 0
idt_descr:
        .word 256*8-1       # idt contains 256 entries
        .long _idt
.align 2
.word 0
gdt_descr:
        .word 256*8-1       # so does gdt (not that that's any
        .long _gdt          # magic number, but it works for me :^)

        .align 3
_idt: .fill 256,8,0         # idt is uninitialized

_gdt:       .quad 0x0000000000000000  /* NULL descriptor */
//下面一行描述符的意思是：
//          7                   0
//          -----------------
//      0 |1|1|1|1|1|1|1|1|
//          -----------------
//      1 |0|0|0|0|0|1|1|1|
//          -----------------
//      2 |0|0|0|0|0|0|0|0|
//          -----------------
//      3 |0|0|0|0|0|0|0|0|
//          -----------------
//      4 |0|0|0|0|0|0|0|0|
//          -----------------
//      5 |1|0|0|1|1|0|1|0|
//          -----------------
//      6 |1|1|0|0|0|0|0|0|
//          -----------------
//      7 |0|0|0|0|0|0|0|0|
//          -----------------
```

//我们看第 5 字节：第 0 位置 0，表示此段未被存取过、未用过。第 1 位置 1，表示此段是可写的。第 2 位置 0，表示是按地址增加方向的

PDF-XCHANGE Click to buy NOW! www.docu-track.com

// 第 3 位置 1，表示此段是可执行的。第 4 位置 1，表示此段是用户段。第 5、6 位都置 0 表示特权级为 0。第 7 位置 1 表示此段在存储器中。

//我们看第 6 字节：第 6 位置 1 操作数是 32 位的。第 7 位置 1，说明段界限表示的长度是以 4K 字节为 1 页的页的数目。

```
.quad 0x00c09a00000007ff    /* 8Mb */
.quad 0x00c09200000007ff    /* 8Mb */
.quad 0x0000000000000000  /* TEMPORARY - don't use */
.fill 252,8,0                /* space for LDT's and TSS's etc */
```

5.2 引导扇区和引导代码实例分析

本节将使用 debug 程序来分析软盘的引导扇区，并且使用 debug 程序来“编写”一个简单的操作系统引导程序。通过阅读本节，读者可以从实践的角度深入地认识引导扇区代码。

5.2.1 引导扇区的内容

在本节中将通过实例来介绍引导扇区的内容和引导程序的编写。

当计算机从软盘启动，BIDS 把引导盘中第 1 扇区载入到地址为 0000:7C00 的内存中。所谓第 1 扇区，就是 DOS Boot Record（DBR）。然后，BIOS 跳到 0x7C00 并开始执行引导扇区的代码。引导扇区的代码完成将整个操作系统载入到计算机内存并开始操作系统的引导。

首先，本节将介绍检查软盘的引导记录。DOS 的 debug 功能是一个广泛用以浏览内存和磁盘内容的工具。本节将使用 debug 来检查软盘的引导记录。

运行 debug 命令，如果输入一个“ d”作为命令并回车，它将显示 RAM 内容的一部分。输入一个问号作为命令，它将给出一张在 debug 中使用的所有的命令。

注意：调用 debug 功能时要非常小心。该功能可以删除一切磁盘上的数据，可能引起一数据的丢失。

将一张刚经过格式化的盘放入 A 驱中，然后可以使用以下的命令来从软盘中加载引导记录：

```
-l 0 0 0 1
```

（第一个字符是字母‘ l’，而不是数字‘ 1’）该命令从一个磁盘上加载扇区到 RAM 的一部分上。在‘ l’后的 4 个数字依次代表：加载数据的起始地址、驱动器标识（0 代表第一个软区）、要加载的磁盘上的第 1 个扇区、有多少扇区要加载。

输入命令“ l 0 0 0 1”将从软盘上加载第 1 个扇区到起始地址为 0 的内存中。当软盘的引导记录被加载到内存中后，如果希望浏览它的内容，可以输入以下命令：

- d 0

屏幕上显示的 8 行代表了软盘引导记录的前 128B（十六进制中的 0x80），结果如图 5-5 所示。

图 5-5　软盘引导记录的前 128（十六进制中的 0x80）字节

图 5-5 是一个 MS-DOS 5.0 启动盘的引导扇区的一部分。左边行中的数字表示：RAM 中的存储地址，中间的十六进制数表示这部分内存中的所有字节，右边的行表示十六进制字节所代表的 ASCII 字符（若字节不能译成任何可见的字符，就显示为一个点）。

在图 5-5 中看到的引导扇区中的这些字节，一部分是引导程序的一部分，另一部分是磁盘相关信息，如每个扇区的字节数、每个磁道的扇区数等。

现在，可以使用反汇编功能来显示引导代码，输入以下的命令：

- u 0

“u0”命令执行了一次“反汇编”操作，显示和以前一样的字节（从地址 0 开始），但是这次 debug 显示了这些字节所代表的 Intel 汇编指令代码，实际结果如图 5-6 所示。

图 5-6　引导扇区反汇编代码

第一条指令是跳转到地址 0x3E 处开始执行。第一条指令后的字节是前面提到的磁盘相关数据，并不是真正的汇编指令，因此从汇编语言角度而言很难理解。debug 在反汇编时，将这些磁盘数据翻译成为汇编语言了。

第一条指令跳过这些磁盘数据，来到了后面的 0x3E 为起始地址的引导代码。可以使用如下的命令来显示这些代码，如图 5-7 所示。

上述代码就是 MS-DOS 5.0 装载系统的部分代码。在 MS-DOS 5.0 中，这些代码将装载 DOS 的两个系统文件 io.sys 和 msdos.sys。引导代码把这两个系统文件从磁盘上装载到内存中，然后跳转执行这两个文件。

图 5-7　跳过磁盘数据的代码

如果引导程序在内存中没有发现这两个文件，那么引导程序将显示下面的信息：

Non-System disk or diskerror

Replace and press any key when ready

上述信息，可以在 DOS 的引导记录中找到，例如输入如下的命令可以看到这段字符串，如图 5-8 所示。

图 5-8　DOS 引导记录的提示信息

图 5-8 显示的是引导记录的最后部分。引导记录刚好是磁盘上的一个扇区，(512B)，在本节中它被加载到起始地址为 0 的内存中去，那么最后一个字节的地址则是 0x1FF。在图 5-8 显示的内容中，最后两个字节（0x1FE 和 0x1FF）的值是 0x55 和 0xAA。引导记录的最后两字节必须设置成这两个值（0x55 和 0xAA），不然 BIOS 将不会加载该扇区并开始执行它。

概括起来，DOS 引导记录从一条指令开始，跳过该指令以后的数据。这 60B 的数据始于地址 0x02，终止于地址 0x3D，引导代码在 0x3E 处重新开始，并一直执行到 0x1FD，它的后面紧跟着 0x55 和 0xAA 两个引导扇区标志。

5.2.2　使用 debug 建立自己的第一张启动盘

在本小节里将介绍如何使用 debug 来创建一个全新的引导程序。在本节中，将不使用任何的程序设计工具，而仅仅使用 dehug 来生成引导扇区。读者学习本节可以对引导扇区有更深入的理解，在后续的章节中，本书将介绍如何使用汇编语言和 C 语言来编写功能更为强大的引导扇区内容。

在本节中将从简单出发，在一张 MS-DOS 5.0 的引导磁盘的基础上来完成第一个引导程序。在本节中，将要替换引导加载程序代码而不去改变引导扇区中的其他数据。这样，这张磁盘仍然可以在 DOS/Windows 下进行正常的读写，原有的数据也不会丢失。如果修改了磁盘引导扇区中的磁盘数据，那么 DOS/WindOWS 将不能识别它。在读写这张磁盘时，DOS/Windows 将给出一个错误信息，说该盘没有格式化。这就导致磁盘上的所有数据丢失。

如 5.2.1 节所介绍，引导扇区中的第一条指令是 jmp 0x3E，因为要跳过引导记录数据。所以，可以从 0x3E 处开始修改代码。

运行 debug，在一张已格式化了的 MS-DOS 5.0 的引导软盘上来进行试验。为了获得一张 MS-DOS 5.0 的引导扇区代码，可以在 MS-DOS 5.0 的系统中输入：

format a: /u/s

从而可以将一张软盘格式化为 DOS 启动盘。

首先，使用“l 0 0 0 1”命令将软盘第一扇区加载到地址为 0 的内存处，然后输入命令：

- u 3E

现在，可以开始使用 debug 修改代码了，输入如下的命令：

- a 3E

可以开始输入汇编代码。输入如下的汇编指令并回车：

jmp 3E

上述指令输入后，可以再次回车以退出汇编模式，回到 deb ug 的普通工作模式，如下所示：

```
- a 3E
0AFC:003E jmp 3E
0AFC:0040
-
```

上述操作如图 5-9 所示。

接下来，可以输入命令：

```
- u 3E
```

可以看到如图 5-9 所示的代码，这段代码的第一条指令就是前面输入的跳转指令。现在需要把刚才输入的结果保存到磁盘中，可以输入如下的指令（注意：这个指令会覆盖了磁盘上原来的引导扇区）：

```
- w 0 0 0 1
```

上面的“写”命令和“读”命令的参数的意义是相同的。“w 0 0 0 l”命令把地址从 0 开始的内存区数据写到磁盘的引导扇区中。

现在可以试验一下这张新的启动软盘了。当使用这张软盘启动系统时，BIOS 将把第 1 扇区从磁盘上装载到内存中的 0000：7C00 地址处，然后从这里开始执行引导代码。引导扇区的第一个指令是 JMP 003E（如图 5-9 所示），而通过上述的修改后，在地址 003E 处的指令为：JMP 003E。这样，计算机将进入无限的跳转循环中（表面上看起来，计算机什么事情也没有作，好像死机了）。

图 5-9　使用 debug 建立第一张启动盘的过程

现在使用这张软盘来启动计算机，可以看到如图 5-10 所示的效果。

图 5-10　磁盘引导计算机的效果键盘

如图 5-10 所示，计算机一直“呆”在那里什么也不做，但是实际上，操作系统已经开始运动了，实际上是在执行“死循环”。

这个“操作系统”的确功能太差了！为了使第一个操作系统比较有趣，下面可以对上面的过程进行一些改进。当计算机启动时，可以调用 BIOS 提供的功能。

5.2.3　使用 BIOS 增强“操作系统”功能

BIOS 是操作系统基本输入输出系统，提供了一些计算机的基本功能，例如，读键盘、

打印字符、读写磁盘等。在本书的附录中，可以查找到 BIOS 的基本功能。

在本节中将使用 BIOS 的 0x10 中断的 0x0E 子功能来在屏幕上写一个字符。0x0x 功能可以在屏幕上输出一个字符，如果这个字符是 ASCII 码的 0x07 字符，那么计算机的喇叭就会发出“嘀”的一声。

0x0E 功能的寄存器设置如下所示：

- AH=0x0E。
- AL=打印字符的 ASCII 码。
- BL=打印字符的属性。

在本节中，将使用 0x0E 功能在屏幕上循环打印笑脸“ J ”和“嘀”的叫一声。实现这个功能，主要用到循环指令和 BIOS 的 0x0E 功能。启动 debug，输入如下的代码：

```
- l 0 0 0 1
- a 3E
0D0F:003E mov ah, 0e
0D0F:0040 mov al, 01
0D0F:0042 mov bl, 07
0D0F:0044 int 10
0D0F:0046 mov al, 07
0D0F:0048 int 10
0D0F:004A jmp 3e
0D0F:004C
- W 0 0 0 1
- q
```

在上面的代码中，首先把 AH 设置成 0，AL 设置成 01（01 代表笑脸字符:-)），BL 设置成 7（黑底白字的颜色代码），然后调用 BIOS 中断 0x10。接着再将 AL 设置成 07（07 会使计算机的小喇叭发出“嘀”的一声）。最后的 jmp 3E 产生一个循环，重复输出笑脸和发声。

使用这张软盘来启动计算机，会看到如图 5-11 所示的运行效果。这个“操作系统”有一点点功能了吧！

图 5-11　包含 BIOS 调用的引导代码

5.3　使用 nasm 生成引导代码

引导代码不同于普通的程序代码，必须是纯粹的二进制代码。GCC 可以支持生成引导代码，这种代码没有重定向信息、可执行文件头等信息。GCC 可以使用如下的编译核心为目标文件：

gcc -c my_kernel.c

“ -c”开关告诉 GCC 只是编译成一个目标文件而不链接。然后，可以使用如下的命令链接核心，并且设定核心装载的地址为 0x100000：

ld my_kernel.o -o kernel.bin -o format_binary -T text 0x100000

用“ -o format binary”开关来运行 ld 告诉链接器输出文件是 plain 的，没有重定向，没有头信息，只是一个 flat 的二进制镜像。

“ -T text 0x100000”告诉链接器用户想要的“ text’（代码段）地址在 0x100000 的内存标记处。

当然用户还必须把自己的二进制文件镜像装载到正确的偏移量处，以使它运行正常，因为所有的重定向已经静态地链接好了。

5.3.1 引导代码基础

引导代码必须遵循如下的规则：

- BIOS 会把引导程序装载到固定地址 0x7C00h。因此，引导程序的段地址和偏移地址是固定的。
- 引导代码必须编译成为 plain binary file 类型。
- plain binary file 的尺寸必须是 512B（512B 是一个扇区的大小）。
- 文件必须以 0xAA55h 标志结束。

5.3.2 最简单的引导程序

本节介绍一个最简单的引导程序 hang.asm，hang.asm 的功能就是启动计算机，然后挂起计算机，代码如下：

```
; hang.asm
; A minimal bootstrap

hang:                           ; Hang!
    jmp  hang

    times     510-($-$$)     db 0 ; Fill the file with 0's
    dw   0AA55h                    ; End the file with AA55
```

“ times 510 - ($-$$) db 0”代码是一句只有 nasm 能够理解的代码。这行代码将在最后编译的文件中插入 0，直到文件尺寸达到 510B。在上面的代码中，加上最后的 0xAA55h 两个字节，整个文件的大小将是 512B。

“ dw 0xAA55h”将使文件的最后两个字节为 AA55h。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

编译上述程序，可以使用如下的命令：

nasm hang.asm -o hang.bin

编译完成后，将会产生 hang.bin 文件，这个文件可以用于启动计算机。使用如下的命令：

partcopy hang.bin 0 200 - f0

注意：上述命令中，0 200 表示从文件 hang.bin 的偏移量 0 开始，复制 0x200，也即是 512B 的数据到第一个软盘驱动器的软盘中去。在执行这个命令时，要在软驱中放入一张软盘。

上述命令的执行结果如图 5-12 所示。

图 5-12　将 hang.bin 写入磁盘引导扇区

当把 hang。bin 写入软盘的引导扇区后，可以在 Bochs 中使用这张软盘来启动计算机。使用 mtools 的命令 mgetimg.exe，将这张软盘的内容转化为 IMG 文件。然后使用 Bochs 启动后显示的画面如图 5-13 所示。。

图 5-13　hang.bin 的引导效果

5.3.3　如何设置段寄存器

在引导程序中，一个重要的工作是设置段寄存器中装入的值。在引导程序刚刚运行时，段寄存器的值可能是 0000 或 07C0。如果需要使用段寄存器，那么必须在引导程序中进行设置。本节介绍如何设置段寄存器为用户需要的值。方法比较简单，总的来说就是在引导程序运行时，跳转到一个预先设定的段，然后把这个段的地址装入段寄存器。下面是在引导程序中设置段寄存器的值的示例代码：

```
; JUMP.ASM
; Make a jump and then hang

; Tell the compiler that this is offset 0.
; It isn't offset 0, but it will be after the jump.
[ORG 0]

jmp 07C0h:start                ; Goto segment 07C0

start:
```

```
    ; Update the segment registers
    mov ax, cs
    mov ds, ax
    mov es, ax

hang:                   ; Hang!
    jmp hang

    times 510-($-$$) db 0
    dw 0AA55h
```

上述代码可以编译后直接用以启动计算机（或写入软盘的引导扇区用以启动计算机）。其运行效果和 5.3.2 小节的运行效果完全一样。只是引导代码在启动时，段寄存器的值已经被设置为 0x07C0h 了。

5.3.4 在引导程序中装入程序（基于扇区）

在 5.3.2 小节中描述的引导程序是最简单的引导程序，整个代码的长度小于 512B，可以完整地放入引导扇区（0 磁头 0 磁道 1 扇区）中。

通常，磁盘引导程序比 5.3.2 小节介绍的引导程序要复杂很多，这时代码的长度可能大于 512B。在这种情况下，就必须使用“二次装载”的方式，把引导代码装入计算机内存并执行。

“二次装载”是指：由 BIOS 从磁盘上装入引导扇区（0 头 0 磁道 1 扇区）到内存地址 07C0：0000，并跳转到该地址执行；再由从 07C0:0000 地址开始的引导扇区中的引导代码把存放在其他扇区中的程序装入内存，并跳转到该程序中执行。“二次装载”的示意图，如图 5-14 所示。

图 5-14 “二次装载”示意图

下面的程序 boot.asm 是一个引导扇区程序，这个引导扇区在执行时，将使用 BIOS 的 INT 13h 功能从磁盘读取第 2、3、4、5、6 共 5 个扇区到 1000:0000 地址处，然后跳转到 1000:0000 地址执行被装入的程序。

```
; boot.asm
; 从磁盘扇区装载另外一个程序，并且执行

[ORG 0]
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        jmp 07C0h:start         ; 跳转到段 07C0

start:
        ; 设置段寄存器
        mov ax, cs
        mov ds, ax
        mov es, ax


reset:                          ; 重置软盘驱动器
        mov ax, 0               ;
        mov dl, 0               ; Drive=0 (=A)
        int 13h                 ;
        jc reset                ; ERROR => reset again


read:
        mov ax, 1000h           ; ES:BX = 1000:0000
        mov es, ax              ;
        mov bx, 0               ;

        mov ah, 2               ; 读取磁盘数据到地址 ES:BX
        mov al, 5               ; 读取 5 个扇区
        mov ch, 0               ; Cylinder=0
        mov cl, 2               ; Sector=2
        mov dh, 0               ; Head=0
        mov dl, 0               ; Drive=0
        int 13h                 ; Read!

        jc read                 ; ERROR => Try again


        jmp 1000h:0000          ; 跳转到被装载的程序处，开始执行


times 510-($-$$) db 0
dw 0AA55h
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

下面的程序 prog.asm 是一段汇编语言程序。这段程序完成的功能很简单，是在屏幕上打印字符串“ Program Loaded Succeed! Hello, myos!”。这段程序被写入磁盘的第 2 个扇区，然后由引导扇区的引导代码装入 1000:0000 执行。具体的代码如下：

```
; prog.ASM
; 被引导程序装载的程序

; 在屏幕上打印：Program Loaded Succeed! Hello, myos!

[ORG 0]
    jmp start2      ; Goto segment 07C0

; 定义需要打印的字符串
msg     db   'Program Loaded Succeed! Hello, myos!',$0

start2:
        ; 设置段寄存器
        mov ax, cs
        mov ds, ax
        mov es, ax

        mov si, msg       ; 打印字符串
print:
        lodsb             ; AL=字符串存放在 DS:SI

        cmp al, 0         ; If AL=0 then hang
        je hang

        mov ah, 0Eh       ; Print AL
        mov bx, 7
        int 10h

        jmp print         ; 打印下一个字符
hang:                     ; 挂起计算机!
    jmp hang

times 510-($-$$) db 0
dw 0AA55h
```

上述程序的编译过程如下：

```
nasm boot.asm -o boot.bin
nasm prog.asm -o prog.bin
```

在编译完成后，可以使用如下的命令将两个程序分别写入磁盘文件的第 1 个扇区和第 2 个扇区：

```
partcopy boot.bin 0 200 floppy.img 0          // 写入第 1 个扇区（引导扇区）内容
partcopy boot.bin 0 200 floppy.img 200        // 写入第 2 个扇区内容
```

上述命令中的文件 floppy.img 是 Bochs 使用的磁盘文件，可以使用 Bochs 的工具 mkdosfs 来生成，生成的过程如图 5-15 所示。

图 5-15　使用 mkdosfs 创建一个 Bochs 使用的软盘镜像文件

上述程序的执行效果如图 5-16 所示。使用“二次装载”功能，引导代码的长度就不必限制于 512B，从理论上而言可以任意长度（不超过磁盘的存储容量）。这样，就可以通过引导程序装载很复杂的程序，包括装载一个操作系统。

图 5-16　“二次装载”例子的执行效果

5.4　本章小结

本章的重点内容是介绍操作系统引导知识。在本章中，首先介绍了计算机的引导设备。通过对常见引导设备，如软盘、硬盘和光盘的介绍，使读者可以了解计算机引导的硬件基础。

软盘和硬盘是两种最重要的引导设备。本章同时介绍了硬盘和软盘的主引导记录数据构成，并且本章还对 MS-DOS 和 Linux 0.01 的引导代码进行详细的分析。

本章的试验程序分成两个部分：第一部分，使用 debug 程序来进行引导程序试验。通过 debug 程序，读者可以很轻松地分析和修改引导程序，从而快速地理解引导程序的基本原理；第二部分，通过使用 nasm 来实现了一个“二次装载”的引导程序。这个引导程序已经类似于普通操作系统使用的引导程序了。

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

第6章 存储管理分析

本章知识点：

- 操作系统内存管理策略简述
- 物理存储管理
- 虚拟存储管理
- 存储管理系统代码实例
- Linux 0.01 的存储管理代码

本章主要介绍操作系统的内存管理。首先，介绍了操作系统中存储管理的基本方法，然后介绍了计算机的物理存储管理机制，本章还介绍了操作系统实现虚拟存储管理的基本方法。

本章提供了一些存储管理系统代码实例，例如，如何探测计算机内存以及 malloc()函数和 free()函数的实现。

最后，本章介绍了 Linux 0.01 的存储管理代码的实现，重点分析了 memory.c 和 page.s 两个核心文件。

6.1 操作系统内存管理策略简述

本节将介绍操作系统中对内存进行管理的基本方法和策略，主要介绍连续分配存储管理方法、非连续存储管理方法、分段／分页／段页式管理方式和虚拟存储管理等基本概念。

6.1.1 连续分配存储管理方式

连续分配是指为一个用户程序分配一个连续的内存空间。这种存储分配方式是最简单的（在某些情况下，也是性能最高的分配方式）。在一些简单的操作系统中，可以使用这种分配方式，例如一个操作系统仅仅固定地管理 4 个设备，每个设备管理程序可以划分为一个程序空间，那么可以把操作系统以外的存储空间固定划分为四个部分，这四个部分的每一个部分都是连续的。连续分配存储管理方式又可以进一步细分为：单一连续分配、固定分区分配和动态分区分配。

1. 单一连续分配

这是最简单的一种存储管理方式，但只能用于单用户、单任务的操作系统中。采用这种存储管理方式时，内存分为两个分区：系统区和用户区。操作系统位于系统区中，用户代码和数据存放在用户区中。

2. 固定分区分配

固定分区分配法将内存空间划分为若干个固定大小的分区。在划分分区的时候，可用两种方法：分区大小相等和分区大小不等。分区大小相等的方法比较简单，但是内存的使用效率没有分区大小不等的方法高。

固定分区分配法使用分区表来记录分区的使用情况。在分区表的每个表项中包含有每个分区的起始地址、大小及状态（标记是否已分配）。

3. 动态分区分配

动态分区分配是根据进程的实际需要，动态地为之分配连续的内存空间。在实现动态分区分配存储管理方式时，必须解决下述三个问题：

- 分区分配中所用的数据结构。
- 分区的分配算法。
- 分区的分配和回收操作。

在分区分配中的数据结构包括两个：空闲分区表、空闲分区链。分区的分配算法可以使用：首次适应算法、循环首次适应算法和最佳适应算法等。

在动态分区存储管理方式中，主要的操作是分配和回收内存。在内存进行回收的时候，注意进行“拼接”或“紧凑”。在动态分区分配中，不能被利用的小分区称为“零头”或“碎片”。通过移动，把多个分散的小分区拼接成大分区的方法被称为“拼接”或“紧凑”。

6.1.2 虚存组织

内存管理子系统是操作系统中最重要的组成部分之一。从早期计算机开始，系统的实际内存总是不能满足需求，为解决这一矛盾，人们想了许多办法，其中虚存是最成功的方法之一。虚存技术让各进程可以高效地共享系统内存空间，这样系统就似乎有了更多的内存供进程使用。

虚存不仅使计算机的内存看起来更多，还提供以下功能：

- 扩大地址空间：操作系统扩大了系统的内存空间。虚存能比系统的实际内存大许多倍。
- 内存保护：系统中每个进程都有它自己的虚拟地址空间。这些虚拟地址空间之间彼此分开，以保证应用程序运行时互不影响。另外，虚存机制可以对内存指定区域提供写保护，以防止代码和数据被其他恶意的应用程序所篡改。
- 内存映射：内存映射被用于将镜像和数据文件映射到一个进程的虚拟地址空间

PDF-XCHANGE Click to buy NOW! www.docu-track.com

中，也就是将文件内容连接到虚地址中。

- 公平分配内存：内存管理子系统公平地分配内存给正在运行的各进程。
- 虚存共享：尽管虚存允许各进程有各自的（虚拟）地址空间，但有时进程间需要共享内存。例如，若干进程同时运行 Bash 命令。并非在每个进程的虚地址空间中，都有一个 Bash 的复制。在内存中仅有一个运行的 Bash 复制供各进程共享。又如，若干进程可以共享动态函数库。

共享内存也能作为一种进程间的通信机制（Inter-Process Communication，IPC）。两个或两个以上进程可以通过共享内存来交换数据。Linux 支持 Linux 系统 V 的共享内存 IPC 标准：

虚拟存储器（Virtual Memory）是一种存储管理技术，可以使操作系统用较小的物理内存实现需要较大内存空间的程序正常运行。虚拟存储器是由操作系统提供的一个假想的“大”存储器。

虚拟存储器所具有的基本特征是：

- 虚拟扩充：不是物理上、而是逻辑上扩充了内存容量。
- 部分装入：每个进程不是全部一次性地装入内存，而是只装入一部分。
- 离散分配：不必占用连续的内存空间，而是“见缝插针”式的分配内存空间。
- 多次对换：进程所需的全部程序和数据要分成多次调入内存。

所谓“对换”，是指把内存中暂不能运行的进程，或暂时不用的数据，换出到外存上，以腾出足够的内存空间，把已具备运行条件的进程，或进程所需要的数据，换入内存。对换是提高内存利用率的有效措施。

如果对换是以整个进程为单位，便称之为“整体对换”或“进程对换”；如果对换是以“页”或“段”为单位进行，则分别称之为“页面对换”或“分段对换”，又统称为“部分对换”。在具有对换功能的操作系统中，通常把外存分为文件区和对换区（Windows 的交换文件，其作用就是形成一个对换区）。

由于对对换区的分配，是采用连续分配方式，因而对换区空间的分配与回收，与动态分区方式时内存的分配与回收方法雷同，其分配算法可以是首次适应算法、循环首次适应算法和最佳适应算法。一

虚存中的置换算法主要指内存中页面、段的选择及换出算法。好的置换算法能适当降低页面更换频率。有 4 种常用的页面置换算法：

- 先进先出法（FIFO）：先进入内存的页先被换出内存。
- 最佳置换法（OPT）：选择将来不再被使用，或在最远的将来才被访问的老页换出。
- 最近最少使用置换法（LRU）：选择最近最久没有使用过的页面换出。

注意：OPT 算法考查将要被访问的页面，而 LRU 算法考查已访问过的页面。从时间上考虑，前者是向前看的，后者是向后看的。

- 最近未使用置换法（NUR）：选择最近没有使用过的页面换出。

6.1.3 非连续存储管理机制

非连续存储管理机制中主要包括：页式管理、段式管理和段页式管理。本节将主要介绍这三种方式的基本原理，并对三种方式的优缺点进行比较。

1. 页式存储管理

页式存储管理的基本原理是将逻辑地址空间分成大小相同的页，将存储地址空间分块，页和块的大小相等，通过页表进行管理。页式系统的逻辑地址分为页号和页内位移量。页表包括页号和块号数据项，它们一一对应。根据逻辑空间的页号，查找页表对应项找到对应的块号，块号乘以块长，加上位移量就形成存储空间的物理地址。每个进程的逻辑地址空间是连续的，而映射到内存空间后就不一定连续了。

此外，页表中还包括状态位（指示该页面是否在内存中）、外存地址、改变位（该页的内容在内存中是否修改过）、引用位（最近是否被引用）等。

页式存储管理的动态地址转换过程是：进程运行时，其页表地址已存放在系统动态地址转换机构中的基本地址寄存器中，执行的指令访问逻辑地址（p，d）时，首先根据页号 p 查页表，由状态位可知，这个页是否已经调入主存。若调入主存，则可直接将虚地址转换为实地址，如果该页未调入主存，则产生缺页中断，以装入所缺的页。

2. 段式存储管理

段式存储管理的基本原理是逻辑地址空间分段，一个进程是由若干个具有逻辑意义的段，如主程序、子程序、数据段、栈段等组成的。在分段系统中，允许进程占据主存中许多分离的分区，段内连续，段有段号，但段长可以相同，通过段表进行管理。段式系统的逻。辑地址由段号和段内位移量两项组成。段表由若干表目组成，每一表目有段号、段长、在。主存中的首地址、存取方式和状态位等项。进程访问虚存时，根据地址空间的段号，查找 i 段表对应段号找到段的首地址，首地址加上位移量就是存储空间的物理地址。

段式系统的动态地址转换过程与页式系统的动态地址转换类似。

3. 段页式存储管理

包含页式管理和段式管理的长处，可以形成段页式存储管理。在逻辑上，形成作业分誉段，段内分页，分配、管理地址空间。在物理上，内存分块，分配、管理存储空间。系统为每个作业建立一张段表，为每段建立一张页表。

4. 三种存储管理方式的对比

在非连续存储管理机制中，页式管理、段式管理和段页式管理等三种方式，它们各有优缺点，表 6-1 对这三种方式进行一个简单的对比。

表 6-1　三种非连续存储管理方式的对比

PDF-XCHANGE Click to buy NOW! www.docu-track.com

方式	地址变换过程	优点	缺点
段式管理	多用户（模块）地址可分成：程序号、段号、段内偏移量兰部分，地址变换过程如下： (1) 由进程号找到相应的段表基址寄存器，在该寄存器中存有段表始址和段表长度 (2) 由段表长度与段号相比较，检查是否越界，如果正常转到(3)步 (3) 段表始址和段号找到其段表中相应表项，其中存有主存地址，装人位，访问位、段长、辅存地址等 (4) 检查装人位是否为“ 1”（在主存），为“ 1”将转到第(5)步，否则产生缺段中断，从辅存中调入一段到主存 (5) 由主存地址+段内偏移形成真正物理地址	多个程序分段编址，多个程序可以并行编程，其地址互不影响，缩短编程时间；各段相对独立，其修改、扩充都不会影响其他段；实现虚拟存储；便于共享和保护	分段管理主存，主存利用率不是很高，有大量的零头产生；为形成一次有效地址，需多次访存，降低了访存速度；分配和回收空闲区比较复杂：段表中地址字段和段长字段较长，降低查表速度页式管理
页式管理	用户逻辑地址分成：用户标志、用户虚页号、页内偏移三部分。过程如下： (1) 由用户标志找到相应页表基址寄存器，其中有页表地址 (2) 由页表始址和页号找到页表中相应表项 (3) 检查装人位是否“ 1”（在主存），为“ 1 ”将转到第（4）步，否则产生缺页中断 (4) 由主存块号和页内偏移形成有效地址	页表表项短，减少访表时间；零头较少。调入速度快	强制分页，页无逻辑意义，不利于存储保护和扩充；一次有效地址生成需多次访存，访存速度下降
段页式管理	段页式管用户逻辑地址被分成：用户标志、段号、页号、页内偏移四部分。过程如下： (1) 由用户标志找到段表基址寄存器 (2) 段表长与段号作是否越界检查 (3) 段表始址+段号找到段表中相应表项 (4) 做装入位、段长的检查 (5) 由页表始址+页号找到页表中相应表项 (6) 做装入位等检查 (7) 实页号+页内偏移形成有效地址	具有段式、页式的优点	一次有效地址形成需三次访存，速度较慢

6.2 物理存储管理

存储管理是任何一个操作系统中不可缺少的部分。操作系统只有配置了存储管理功能，才能有效地管理计算机的内存资源。在操作系统核心中，通常有几个内存管理器，每个存储管理器工作在不同的级别。在本节中，将介绍较低级别的物理存储管理器。

在本节中将通过一些具体的例子来解释如何进行物理存储管理。这些例子稍加修改后便可以用在读者自己编写的操作系统中。、在本节中，假设操作系统内核工作的 CPU 都是具有分级功能的，并且都运行在 CPU 的核心态。在 x86 CPU 中，操作系统核心工作在 386 以上 CPU 的 0 级特权模式下（Ring0）。在本节中，所有的例子都是针对 Intel 80386（或以上）CPU 的。

虽然，使用汇编语言来书写存储管理代码会具有更高的执行效率，但是为了使代码具有更高的可读性，本节的代码都是使用 C 语言来编写的。

6.2.1 技术细节

在操作系统中，存储管理也划分为高级存储管理（例如同时支持分段和分页存储管理）和低级存储管理。低级存储管理较为简单。在操作系统启动后，计算机的控制从 boot loader 移交到了核心，这时操作系统核心对计算机硬件就有了完全的控制。核心代码运行在计算机的一部分内存中（核心是被 boot loader 加载并放置在这段内存中的），核心的堆栈和数据区域都已经建立起来了。操作系统启动到这个时候，核心可以进行一些最简单的工作了，例如在屏幕上打印“ hello world”。

通常，在核心中物理存储管理器都是最先运行的代码（早于在屏幕上打印信息的代码）。一旦物理存储管理器运行起来后，就可以通过存储管理器来执行很多操作系统的功能，例如设置全局变量（用来指示数据的存在）、调用函数（实现堆栈）、向屏幕输出（访问剩余的物理内存）。有了这些方法，操作系统就可以把计算机置于图形工作模式，在屏幕上输出图形，例如像 Windows 系统一样工作在图形模式下。因此，从上面的分析可以看到：物理存储管理是操作系统中较低级和基础的功能。

物理存储管理器任务是比较简单的：将整个计算机物理存储器分成不同大小的“块 ”j 然后依据操作系统核心的需要分配这些物理存储器“块”。为了简单起见，通常称呼这些“块”为“页面（pages）”。在 x86 机器上，一个页的默认大小是 4KB。此外，微处理也通常。使用分段来访问物理存储器。任何一个物理地址都可以映射为：段地址+段内偏移。

在现代处理器结构中，x86 是惟一可以同时处理分段和分页的微处理器。存储器的地址通过分段和分页两种机制共同来指定。

大多数 32 位体系结构的 CPU 将它们的 4GB 地址空间划分为页面。x86 引入了段的概念，可以使操作系统将存储器划分为不同的段，例如代码段、数据端、堆栈段。在这个方面，x86 处理器和其他大多数的处理器都是不一样的。因为这个原因，很难将一个

x86 的操作系统的底层代码移植到其他的体系结构上。

如果没有分页机制，x86 处理器只能访问 16MB 的内存。这样，计算机就回到了 286 的时代了。如果没有分段机制，就只能访问 64KB 内存。80386 将内存划分为 4KB 的页面，提供了可以访问 4GB 存储空间的能力。

考虑一下如何在一个 32 位体系结构的 CPU 上实现分页内存管理。首先需要定义一个基本的准则：内存管理器的基本原则是让操作系统核心在访问物理存储器时可以不再使用物理内存地址(惟一使用物理内存地址的代码就是物理存储管理器代码本身)。这样作可以使操作系统的上层代码和具体的物理内存大小和分页大小无关，从而保证了操作系统上层代码的可移植性。例如，Pentium 及更高级的处理器可以选择 4KB 的页面大小，同时 36 位的物理地址空间可以提供 64GB 的物理存储空间大小（尽管虚拟地址空间仍然是 4GB）。如果操作系统配置了一个好的存储管理器，那么操作系统可以充分利用计算机提供的存储空间，但除了使用存储管理器的功能外，无需作更多的工作。如果没有一个好的存储管理器，那么可能应用程序在不同的计算机上，为了充分利用物理存储器，不得不重新编译应用程序，这使应用程序的可移植性大大降低。

6.2.2 物理存储管理器的组织

最基本的物理存储管理器至少需要一些内存作为数据结构来管理余下的大量物理内存。如何在物理存储中使用这些存储分配信息，有两种方式可供选择：

- 在每一个分配的内存块的首部设置一个头，在这个头中存储内存分配信息。
- 单独使用一个区域来存放每一块分配的存储块的分配信息。

如果在操作系统中，准备将内存划分为较大的块来使用，那么第一种方式是较为适合的。在 C 语言中的函数 malloc()就是使用这样的方式来管理内存的。

如果操作系统中准备使用分页来管理存储器，那么在每个页之前放置几个字节的控制信息就显得比较浪费了。此外，第一种方式还有一个很严重的弊病，在分配的内存前部有一个头，那么如果应用程序在分配的页面前部错误地写了一个字节，那么系统的存储管理器就会崩溃。

因此，对于一个低级存储管理器，第二种方式是较好的。如果在存储管理系统中有全局的数据结构，那么系统就有能力访问所有的物理内存：只需要一个指针指向系统中任意地方，然后通过这个指针，就可以访问系统中任何地址的内存。

一般情况下，操作系统 boot loader 程序会将操作系统核心引导到 1MB 以上的地址空间（工作在只有 1MB 物理内存空间的 DOS 操作系统是不同。DOS 将核心引导到 640KB 以下的地址空间。640KB～1MB 的地址空间通常会由计算机 BIOS 使用），然后 boot loader 会跳转到这个地址，将控制权交给操作系统核心。因此，在链接生成核心时的偏移地址必须大于 1MB；也就是核心的可执行代码和数据在内存中的绝对地址需要大于 1MB。当核心在内存中存放完成后，系统中应用程序可以放置在接下来的地址空间中。

在操作系统中，通常会将地址空间划分为用户地址空间和核心地址空间：在用户地

址空间中存放所有用户应用程序代码和数据，在核心地址空间中存放所有的核心代码和驱动程序。一般情况下，核心不直接访问用户程序地址空间，用户程序也不能访问核心地址空间。用户程序的虚拟地址空间是不受限制的，可以达到 2GB（注意：在使用虚拟存储的操作系统中，用户程序的虚拟地址空间是不受计算机的实际物理存储器大小限制的）。

另外的一些操作系统，例如（Windows 9x、Windows NT、“ nux）核心地址空间是 2GB，而用户地址空间在 2GB 以下。如果核心代码加载在物理地址 1MB 的地址空间，如何使核心的虚拟地址在 2GB 以上呢？在分页存储管理系统中，这是很简单的。只需要将指向核心的页表项的地址修改为相应的地址即可，例如 0xC20000000。

将内存管理器放在 boot loader 中是难以实现的。这需要让处理器在不使用分页管理的情况下将地址 0xC0000000 等效为地址 0x1000000。在 boot loader 运行完成后，核心要运行分页管理机制，然后重新移动核心代码。

为了实现上述目的，可以使用欺骗核心代码和数据的基地址（base address）来完成。

注意：CPU 得到物理地址的方法是，在虚拟地址（例如 0xC0000000）的基础上加上段的基地址，然后将它发送到地址总线中（如果分页机制打开了，将它发送到 MMU 中）。如果地址空间是连续的（没有漏洞和空隙），CPU 可以将任何的虚拟地址映射为对应的物理地址。

在前面的例子中，如果核心位于虚拟地址 0xC0000000，物理地址是 0x1000000，操作系统中就需要一个段基地址将虚拟地址 0xC0000000 映射为物理地址 0x1000000，也就是 0xC0000000 + base = 0x1000000。在这时，段基地址就是 0x41000000（这是在 32 位的情况下）。核心指向一个全局变量的虚拟地址为 0xC0001000，CPu 在这个虚拟地址上加上 0x41000000，然后就可以得到实际的物理地址为 0x1001000。这种运算在 boot loader 加载核心时经常用到。而在核心容许分页机制后，同样的地址映射关系 0xC0000000 到 0x1000000 也存在。

在分页存储管理中，操作系统通常划分一个专门的区域来存放内存控制数据结构。对这种数据结构的基本要求是：数据结构占用的存储空间必须小于被管理的存储空间。这样，整个存储管理系统才能有效地运行起来。

对于一个分页存储管理系统而言，最基本的功能要求是记录哪些页面是被分配了的，哪些页面是没有被分配的。对于这种需求，最简单的数据结构是 bitmp（位图）。位图可以使用一位来指示系统中一页的分配情况，例如在 x86 系统中位为 1 表示使用，位为 0 表示未使用。在一个具有 256MB 物理内存的系统中，需要 8192B 的位图来管理全部的 65,536 个页面（65,536*4096=268435456；65,536/8=8192）。在低级存储器管理器功能中，只需要记录页面是否被分配了或者是没有分配。在高级存储管理功能中，除了记录一个页面是否被分配以外，还需要知道一个页面的大小是多少、那个进程分配了这个页面、这个页面被分配了多少次等。在低级存储管理器的基础上，可以构建更高级的存储管理器。

使用位图的优点是地址空间使用非常有效，并且非常简单。每一个页面仅仅需要一

个 bit（位）来记录控制信息：这个页面是否已经分配或者没有分配。但是，使用位图的一个缺点是如果需要定位一个没有被分配的页面，可能需要扫描整个位图，特别是当系统中的大多数的页面都被分配时，扫描的时间将更长了。这种情况，对于内存较大的计算机系统而言，所花费的时间将更长。

另一个可选的方案是使用页面堆栈（stack of pages）。自由页面的物理地址被压入堆栈中：当页面被分配时，下一个地址空间从堆栈的栈顶弹出而被使用。当页面被释放时，它的。地址被压回堆栈。在这种情况下，内存块的分配和回收动作就等效为指针的增加和减少。

实际上，大多数的内存分配并不要求物理内存连续分布（MMU 可以使不连续分布的警物理内存通过映射，在应用程序面前表现为连续的）。如果应用程序需要分配的物理内是连续的，那么通常这些内存应该从堆栈的中部分配（堆栈中部的内存块这时通常还是连续的），但是，这种情况下的内存分配过程比较复杂。

使用堆栈的方法来管理内存的不足是很难对分配的内存块指定物理地址。比较一下 DMA 的例子，ISA DMA 要求传输的地址是内存地址空间的前 16MB（因为使用了 24 位的地址总线）。如果需要书写一个软件控制器的驱动程序，那么就需要在低于 16MB 的内存空间中分配内存缓冲区。为了处理这种情况，一种简单的方法是维护两种内存管理的堆栈结构：一个是低于 16MB 的内存管理堆栈；一个是其他的内存管理堆栈。

6.2.3 物理存储管理器的初始化

为了有效地管理计算机的全部物理内存，需要使用部分物理内存来存放用于内存管理的数据结构。这些数据结构就是 6.2.2 小节所讨论的位图（bitmap）或堆栈（stack）。这些用来管理物理内存的数据结构必须大小合适，与计算机系统中实际具有的物理内存大小相匹配。

为了获得计算机的物理内存大小，有很多方法，其中最简单的方法是在 boot loader 中通过 BIOS 来获得物理内存大小。具体来说，可以通过 BIOS 的 15h 中断从计算机的 CMOS 中读取计算机的物理内存的大小，并把这个数据保留在操作系统核心中。

通过 BIOS 的 15h 中断确定计算机系统中的物理内存容量后，就可以在内存中开辟一块区域作为内存管理数据结构。在真正实现物理内存管理器之前，首先需要确定这些数据结构的大小和地址。在计算机系统中，有一些保留内存，这些内存是不能使用的，例如 BIOS 内存区域。

接下来，将分析是否可以把内存管理器所使用的位图或堆栈附加在内核使用内存的尾部，并把这些数据当成操作系统的全局动态变量。

内核的起始地址和结束地址是已知的（例如，可以在一个 GNU ld 脚本中，在内核镜像的末端设置一个符号），这样就可以计算出内核的大小，以避免从内核区域开始分配内存。

注意：从内核中间分配内存将引起致命错误，这种错误通常会使整个操作系统崩溃。

操作系统核心可以管理计算机中所有的物理内存。在存储管理器中，可以使用位图和堆栈来记录内核可以管理的物理内存。在操作系统核心中，可以使用一个指针指向该区域的起始部分并把该部分记为位图或堆栈自身的保留区域。在使用位图的情况下，需要为那个区域的每一个地址设置 bit（位）；在堆栈情况下，要把除保留区域以外的所有地址都放入这个堆栈。

6.2.4 页面分配

第一个内存管理功能是物理页面定位器。物理页面定位器的功能是把一个物理页标记为使用过并返回它的地址。如果使用一个堆栈来实现物理页面定位器，原理比较简单：跟踪空页的数目，当一页被定位时，缩减空页数目并在堆栈顶部返回页地址。对于一个位图，需要在比特矩阵中将该页所对应的比特标记为使用过。

在物理页面定位中，还要考虑定位特殊地址的问题。例如 ISA DMA 中必须使用 16MB 以下的物理地址，在进行物理页面分配时，必须满足物理页面的物理地址在 16MB 以下的条件。对位图而言，要满足这样的位置条件比较容易，只需要在一定的偏移量处停止分配。对堆栈而言，要实现这样的地址条件限制比较困难，通常可以保留两个页堆栈，使用一个页面堆栈来在一定地址范围内浏览主堆栈并把这个地址空间中的页面标记为使用。

在进行物理页面分配时，如果没有剩下任何空白页，那该怎么办呢？如果磁盘上存在一个交换文件，就可以利用磁盘空间来对换暂时不使用的物理页面，从而“腾挪”出物理页面供其他程序使用。然而，要实现物理页面和外存储空间的对换，需要写实现更多的内核功能，至少包括磁盘文件系统和虚拟存储管理系统。对于一个简单的操作系统而言（假如仅仅处理一些简单的控制任务的操作系统），通常并不需要定位 32 位处理器所能访问的全部 4GB 内存。因此，对于一个操作系统而言，物理存储管理系统是必须的，虚拟存储系统是可选的，例如在 DOS 系统就没有虚拟存储系统。

由低级定位器返回的地址是一个物理地址，通过这个物理地址，内核可以对这个物理页面进行随意读出或写入。通过页面定位器，内核具备了将进程虚拟地址空间映射到内存中的物理页面上的能力。

6.2.5 页面回收

页面回收过程是页面分配过程的一个相反过程，在此不再详细介绍。页面回收过程通常可以通过以下的方式来做到：在位图中清除 1 bit（位）或向堆栈中压入一个地址。

6.2.6 映射

通过映射可以使前面的页面分配和页面回收变得真正的有用。简单地说，映射容许

使用软件的方法（也需要处理器的支持）来控制地址空间的访问，例如可以把一个页面映射到任意的物理页面，可以对页面进行保护，可以在发生缺页故障时根据需要来定位并映射页（或者甚至完整地模拟整个内存存取）。

在 Intel 80386 处理器的硬件手册中有对 MMU 的描述（其他处理器和 Intel 80386 也很类似），Intel 80386 及更高配置的处理器使用一个三级转换方案：PDBR（Page Directory Base Register，即页目录基址寄存器，也叫做（CR3）包含了一个页目录的物理地址。页目录是内存中的一页，它分为 1,024 个字（每个字 32 位）或页目录条目，每一条是一张页表的物理地址（Page Directory Entry，PDE）。每张页表也分为了 1,024 条页表目录（Page Table Entry，PTE），其中的每个字是内存中的一页的物理地址。总结一下，Intel 80386 的三级地址转换方案是：页目录基地址→页表基地址→页地址。

注意：在 80386 处理器中一个页面的默认大小是 4096B，为了管理由一个 4096B 大小页面组成的最大 4GB 的物理内存，PDBR、PDE 和 PTE 每个只需要 20 位就足够了（2^{20}=1048576，1048576*4096=2^{32}=4G）。所以 PDBR、PDE 和 PTE 的低 12 位可以用作标志位，这些标志位可以用于保护页和页表（分为读／写及普通用户／管理员等层次），这些位还可以标记每一张页和页表是否在当前正在使用。同样也有一个“访问”标志位，当一张页或页表正在被访问时，处理器就将它置位。

通过交换 PDBR，核心可以在不同的情况下使用不同的页目录和页表，从而在多任务环境中把每个进程的虚拟地址空间完全的隔离开。这是现代操作系统实现多任务存储空间隔离的硬件基础。

对于操作系统的存储管理而言，分页是很有用的。因此，在内存管理器中需要编写功能支持多个页目录和页表的切换。

页目录和页表虽然是管理其他普通页面的功能，但是它们在内存中也只是普通的页，处理器需要知道它们的物理地址才能使用它们。一个应用程序（或一个地址空间）只有一个页目录，每个页目录中有 1,024 张页表，每个页表可能有 1024 个页表项。对于页目录和页表，可以像前面使用的物理页面分配器来分配和回收。

注意：页目录和页表地址要排列成行，也就是说，低端的 12 位要写作零（否则，这 12 位中存放的读／写／保护／存取信息可能被破坏）。

在为应用程序实现分页机制之前，需要一个空闲的 PDBR 和页目录以及一张页表。当前页表需要把特性映射到 EIP：当在 CR0 中设置分页位时，CPU 也将从相同的地址处开始执行。所以最好在执行处有一些有用的指令。

在实现分页之前，需要定位一个页目录和一张页表。核心可以使用页定位器或保留一个 4096B 的全局变量，任何一种方法都是一样的。如果核心为页目录／页表使用全局变量，要保证它们排列成行。

在初始化页表时，可以把所有没有使用的位清零，这样可以清除当前位、以使得存取这些页条目所指向的地址时将引起一个页故障。注意，每个 PDE（和页表）控制了 4MB 的地址空间，这对内核来说是足够的了。每个 PTE 只占用 4KB，所以内核需要很多 PTE，

假设操作系统核心没有动态地重定位内核(这样做是允许的. 因为操作系统核心是地址空间的第一位用户，应该不会产生冲突)，而仅仅在页目录和页表中抽取一些已正常定位的数字。

所有这些都需要在进入内存管理器初始化后完成，因为核心的内存管理器要把所有的物理内存有效地管理起来。因此，在从内核管理器退出运行时，整个系统的分页制就已经实现了，并且保持相同的地址格式（否则，由上一次功能调用从堆栈返回的地址将出错）。所以，建立核心的页目录和页表，实现分页，做一次远跳（以重新载入 CS）并像通常一样重新载人其他的选择器（SS，DS， ES，FS 和 GS）。当系统中实现了分页机制后，就可以使用基地址为零的段了，尽管并不是一定要这样做。

如果存储管理器工作正常，那么操作系统核心就运行在实现了分页机制的机器上了。现在，所有的地址都已建立了管理页目录和页表。而其余的物理内存并不一定要映射到新的地址空间里，所有必需的部分是 CPU 当前执行的部分代码（事实上，这时候仅仅需要映射整个内核：数据、堆栈和所有其他的内容）。如果映射了显存，就可以用字节操作显存来方便地在屏幕上显示信息。

6.2.7 内存映射

在 6.2.6 小节简单介绍了映射的基本原理，就是在当前页和当前页表的正确页中放置字，如果需要，定位一个新的页表。CPU 在访问页目录和页表的时候使用物理地址，访问任何其他内容都使用虚拟地址，这可以通过很多种方法来实现：

- 把所有的物理内存映射到地址空间。这可以通过一一映射（物理内存与地址空间严格标识对应）或用一些偏移量（物理内存可以在虚拟地址（0xd0000000 开始处存取）来完成。简单是这种方法的一大优点（Win9x 使用这种方法）；然而它的缺点是用户可能在系统中安装了任意容量的内存，这些内存都需要被标识地址。如果用户安装了 4GB 的物理内存，可能就没有剩余的地址空间了。
- 把每一页映射到地址空间，并在一个和实页目录对应的虚拟页目录中跟踪它们的虚拟地址。虚拟页目录可以存储每一张页表的虚拟地址，实页目录存储它们的物理地址。如果物理内存中只有一些必须直接标明地址的碎片是页目录 / 页表，这种方法就很奏效。然而，它却增加了只由映射占据的空间。同时，它在一个物理内存容量较小的系统中并不太好。
- 把页目录映射到它自身。这看起来可能是一种很奇怪的内存映射，但事实上它能很好地运行。通过把一个固定 PDE 设置成相关页目录的物理地址，可以用不同的地址来标识 PDE 和 PTE。如果把每一个页目录的第 1023 单元设置成页目录自身的地址，处理器将把页目录当作最后一张页表。它将把 PDE 当作 PTE，并且把 PTE 当作在地址空间的最高 4MB 中的单独的 32 位的字。这样就可以把这地址空间的最高 4MB 作为原来的页目录中的条目来使用。这种方法具有简单可行的优点；缺点是只能存取映射到当前地址空间的页。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

举一个例子来说，Windows NT 把 512MB 的物理内存映射到内核的地址空间（正如第一种方法那样）。其中每一步都要申请正确的保护：在 PTE 中，用计划好的方法保护哪一页；在 PDE 中，用计划好的方法保护哪个 4MB 的区域。每一个，PDE 都应能被正确读 / 写并对用户可见。

如果想要 HAL 类型的机器抽象起来的话，可能需要一个地址空间，在这个地址空间上，可以通过调用功能表（在 x86 上，语句为 MOV cr3, addr）上的功能来进行任务的切换。

在充分使用地址之前，需要一个更复杂的内存管理器，下一节将介绍虚拟存储管理。

6.3 虚拟存储管理

在本节中，将进一步讲解存储管理，在 6.2 节中使用的是存储管理器概念，而在这一节里将介绍虚拟存储管理。

在低级存储管理器中介绍了计算机管理物理内存的方法：计算机的原始地址在诸如分页之类操作之前就产生了。这种低级存储管理器有三个主要部分：一个分配器（每次分配一个物理页），一个释放器（用来释放分配的页）以及一个内存镜像器（用以执行虚拟地址和物理内存之间的镜像）。现在可以通过增强这些东西来完成一些更好的功能，例如虚拟存储管理。

在 C 语言中提供了像 rnalloc()这样的内存分配函数，它们存在着以下一些问题：

- 它们一次就将所有的页分配给用户。如果用户所需要的内存是 4KB（在 x86 的
- 机器上）的整数倍就不会浪费内存，但对于像链表（使用非常频繁的数据结构）这样的结构，通常不是 4KB 的倍数，这时分配一个页面就显得太浪费了。
- 它们只能对物理地址操作，却不能很好利用虚拟地址。这意味着它们并不能完全利用分页的优点，分页的优点就在于内核可以独立于物理内存的实际架构（如。固定的分区）来分配程序空间。
- 它们对已分配的内存不能进行控制：它们不支持诸如共享、换出到磁盘、内存映射文件、磁盘 cache 整合等操作。

最简单的物理内存管理器同 c 语言提供的 malloc()函数类似，也存在着上述的问题。要克服这些缺点，可以在物理内存管理器的基础上建立一个函数层，通过这个函数层提供的功能来实现虚拟存储管理功能。

注意：这个增强的函数层是运行在最简单的物理内存管理器的基础上的，与具体的物理硬件无关并可移植，可以运行在任何体系结构的机器上。

6.3.1 技术细节

为了实现虚拟存储功能，可以编写两个存储管理器：一个用来管理用户程序虚拟地址空间，另外一个用以实现 malloc()和 free()（或者是其他语言中相应的操作）。一个虚

拟内存管理器可以方便地为应用程序提供大量的内存（一部分内存是使用外存虚拟的）。当然，将物理存储管理器直接交由用户操作是可以的；但这样用户程序就需要知道目标机的物理体系机构，这将导致应用程序缺乏可移植性。为了避免这样的麻烦，可以将这些事情交给内核去做。

Linux 内核提供了一小部分的 C 动态库，Windows NT 则拥有它自己的与 C 动态库相对应的函数集。Linux 中有 malloc()和 kmalloc()，而 Windows NT 则有 ExAllocatePool()以及类似的函数。这里将主要讲解 Linux 的 malloc()画数实现。

6.3.2 malloc()和 free()

malloc()和 free()是实现最简单的两个函数。malloc()函数作用是从一大块内存中分配出所需要的空间。在分配时需要注意，要留出一部分 malloc()不能分配的内核空间（为了便于存储保护，内核地址空间和用户地址空间是相互隔开的）以备内核使用。

内核地址空间通常分为不同的区域，例如一个操作系统内核从 0xC00000000 地址开始，一直延伸了 256MB 的空间，其区域分布如表 6-2 所示。

表 6-2　内核地址空间分布

内核地址空间	功能
0xC0000000	内核代码，数据等（也许太大了，但总有一天会看到 256MB 的内核）
0xD0000000	内核堆
0xE0000000	设备驱动保留空间
0xF0000000	一些物理内存（对于访问显存很有帮助）和当前进程的页目录和页表

注意：从 0xD0000000 到 0xE0000000 的 256MB 地址空间分配给了内核，但这并不意味着 256MB 的空间会一直被内核所占有。它只是表明有一个 256MB 的窗口，通过这个窗口可以访问内核。

本节先不讨论 malloc()和 free()的具体实现，这两个关键函数的实现将在 6.4.2 节中通过一个具体的例子程序来加以介绍。一般情况下，mal.oc()的实现应该包含一件事：通过操作系统内核申请更多的内存。在更普通的用户模式下，这个函数（通常叫做 morecore()或者 sbrk()会向内核申请一大块内存以扩展进程的地址空间。本书主要研究操作系统内核，因此在这里将主要分析 morecore()函数的实现。

morecore()（如在 K&R 中）和 sbrk()（为大多数己 Unix 采用）之间会有一点点的差别：

void *morecoreo(size_t n)

请求 n 个字节的内存（K&R 分配的空间为分配最小空间大小的整数倍），它们可以来自用户地址空间的任何地方。

char *sbrk(size_t d)

分配给应用程序 d 字节的空间并返回新分配区域首的指针。返回 char *是因为早期的 C 版本不支持 void *类型。

虽然它们差别甚微，但还是应该明确这些概念。实际上，morecore()可以像 sbrk()一样工作。

当 rnalloc()用完内存时（这个内存是在 malloc()和 free()中管理的内存），它将调用 morecore()向核心再申请内存。如果成功，新分配的块被添加到空余块表（tree block list）；若失败，则返回 NULL。在内核中，需要增加 n 个字节为内核堆预留空间并返回一个新空间的首指针时，morecore()返回的仅是一个虚拟地址，还需要进行如下步骤的工作：

(1) 使用物理内存分配器分配一页。

(2) 使用物理内存映射器将其映射到内核堆的尾部。

(3) 调整内核堆的尾指针。

(4) 如果还需要继续分配内存，跳转到步骤(1)。

在上面的过程中，并没有相应的函数来释放堆分配所得的空间。所有的 malloc()都不会向操作系统申请释放内存。因此，这种算法只能分配内存。即使内存被分配然后再被释放，那么它的状态还是被分配的（如果内核没有主动回收内存的话）。

注意：上面介绍的 malloc()是在核心中实现的，只能被内核或者内核模式的设备驱动器调用。用户程序可以拥有它们自己的 malloc()或者其他的类似函数，比如 GetMem()内存分配函数。这些用户模式的分配函数需要调用内核模式下虚拟存储管理器。

6.3.3 缺页处理

在 6.3.2 小节中介绍的内核 malloc()函数已经具备了存储分配功能，但是还不具备虚拟存储功能，本节将介绍如何通过缺页处理来实现虚拟存储管理。

同物理内存管理器一样，虚拟内存管理器也是管理地址空间的。不过，这个地址空间是虚拟的（通常称为虚拟地址空间）：它具有固定 4GB（对于 32 位处理器而言）空间，而且它不需要同机器上实际安装的物理内存容量一致。这个虚拟地址空间的各部分可以被映射到同一个实际的物理内存上去（实现共享），也可以不被映射（如堆栈末段的警戒页，一旦访问这个页面就会引起缺页中断，警告堆栈溢出），也可以用来作为实际内存的复制（比如 Windows NT 的 ntvdm.exe 就是计算机的 BIOS 数据区的复制，它供给实模式下运行的应用程序访问并使用）。

实现虚拟存储的重点是如何处理页的失效，也就是缺页处理。最一般的页失效发生在处理器尝试读或者写一些被标明为“不在内存中（not present）”页的时候。页表中每一个入口中有一个“存在（present）”位，处理器每次访问一页的时候将检查该位，如果为 1，则继续访问；如果为 0，操作系统将调用页失效处理程序。这样，当一个页面不在物理内存中（也就是缺页）的时候，处理器就将控制传给页失效处理程序（也叫缺页处理程序）。

页失效处理程序在处理器访问某个不在内存中的页时将被调用，它所要做的包括：

- 找出在出错地址上的内容。
- 如果该地址需要被访问，保证地址可以被访问。
- 当地址不正确的时候采取合理的措施。

这里所谓“合理的措施”也包括终止发生页失效的程序。对于不同的操作系统，处理方式不尽相同：Windows NT 调用当前的结构异常处理程序集（Structured Exception Handlers），该程序默认处理办法是终止该程序；Linux 则调用适当的信号处理程序（signal handler）。

6.3.4 虚拟存储管理的页面分配

为了能查找到在失效地址空间中所应分配的内容，需要记录下地址的分配方式，本节将要介绍一个虚存的管理器。

Intel 80386 提供了 32 位地址总线，可以访问 4GB 的物理地址空间。但是，在计算机系统中实际上很少安装这么大的物内存。实现虚拟存储就是使用一部分磁盘存储空间来模拟（或者称为对换）物理存储空间。要实现虚拟存储管理，需要实现一个能够分配任意大小的虚存的程序。虚存所依赖的物理内存并不一定需要立刻就被分配完，而且也不一定是连续的。当然，该物理内存一定是存在的，处理器通过地址总线把数据送到该内存中，但这并不意味着需要立刻就分配完该物理内存。在开始，只需要记录下列表中发生的所有分配就够了。malloc()是一个很有用的函数，通过它可以一次分配一条纪录也可以记录下分配的虚地址和已经分配了多少页等信息。这些记录的分配基址及大小将提交给物理存储器。通常，分配是针对页而不是针对字节来进行。因为这样可以使内核的统计更简单。

在一次分配中所应记录下来的信息至少应包含以下几条：

- 虚基地址（例如：在当前应用程序的地址空间中块的起始地址）。
- 大小。
- 安全级（例如：读 / 写，用户 / 核心使用）。

一旦分配了一块内存给应用程序，那么就该让应用程序可以访问它。但是由于该块还没有复制物理内存，在页表中该块的人口地址被标记为“不存在”。所以处理器会产生一个页失效中断，调用相应的页失效处理程序。

页失效处理程序要做的第一件事就是检查分配表，找到失效地址。x86 处理器记录下的是实地址而不是页的起始地址，但这并没有太大的区别，找到失效地址后就可以确定出跨越该失效地址的块。

一旦该块被找到，就为其分配相关的物理页并将它复制到相应的地址空间中。这需要用到 6.2 节中讲到的物理存储分配器，物理存储分配器可以决定究竟是一次分配一个整块，还是一页一页的分配。如果需要分配 4MB，也就是说 1024 页，而应用程序可能在很长一段时间内只使用到其中的 1B 口如果一次就分配了一个整块，那么就浪费了剩下的 3.99MB。当然，也可以一次只分配一页，然后当产生页失效中断的时候再调入相应

页。在这种情况下，如果应用程序用完了所分配的 4MB，就会产生 1024 次缺页中断。每次中断都会耗费掉处理器一些时间，而一次分配整块就不会出现该问题。可以看到，两种情况都有优缺点。在操作系统设计时，需要综合考虑到这些问题，并根据实际的应用程序特性做出选择。

为了提高页表的处理性能，在处理器内部通常都具有页表缓存，它被称作快表（TLB）。该表是上一次缓存访问失效后所更新的 PDEs 和 PTEs 的复制。当 CR3 被写入或者指令 INVLPG（在 486 及其以上）执行的时候，缓存失效。使整个快表失效需要一定的时间，所以只有在当两进程切换等情况下才能这样做。因此 INVLPG 指令是非常有用的，它可以使快表中相对于某特定页的一部分失效。当页的目录或页表发生变化时，快表也必须作相应的更新。通常处理器并不会自动完成更新，所以在分配了物理内存后必须由程序来完成。

当失效地址已经被正确地识别成为要分配的块的一部分，并且相应的物理内存已经被成功分配和映射，内核就可以从页失效中断程序中返回，并继续执行当前的应用程序。

6.3.5 可执行程序

当一段可执行程序被装入到内存中，内核需要记录下一些相关的执行信息。比如，该程序包含有几个部分，它们都位于哪儿，它们有多大。这些信息以一定的顺序被存储在应用程序的头部地址空间中（称为应用程序头）。应用程序在开始执行的时候，所有代码并不需要全部装入内存，但必须把头部信息全部装入内存中。。

由于应用程序在运行的时候并没有将应用程序本身全部装入内存，这样的情况可能导致应用程序在执行过程中出现缺页中断。当缺页中断发生时，核心在查找已分配块表的同时还要查找记录有执行副本信息的表以找到需要执行部分的基址，并分析它的头部文件，定位到地址失效地址的部分。

可执行文件的头信息格式有许多标准，例如 coff、elf 和 pe 格式等，依据文件格式可以方便地基于文件头信息找到可执行文件中地址失效的部分。接下来，核心调用一段独立的磁盘读程序将需要读人的可执行文件部分从磁盘调度到内存中。在这个过程中可以做一些优化，比如，对于只读代码页，如果在磁盘中已经有一个副本的话就不需要再将其交换回磁盘，另外只读代码页可被多实例共享。数据页在被写入前可被共享，但一旦其被写入了，它就必须被复制回磁盘。在这里常用的原则就是一旦有写入操作就要复制（Copy-On-Write，即 COW 原则）。

初始情况下页面都设定为只读，一旦应用程序试图去写入时便会报错，由此便会调用页面错误处理程序。当处理程序探测到报错的页面是一个数据页面时，它就会在下面的，内存空间中创建一个副本，将该副本设定为可读可写，并改变内存的映射，使应用程序对应于刚刚创建的可读可写副本。

在 unix 的 fork()系统调用中也使用了 COW 原则。它为当前进程创建一个副本，该副本和原进程是完全一样的。它们共享代码、数据和堆栈，并且从同一个人口继续执行。

因为大量的页面被共享，节约了不少内存。但是，新老进程毕竟是独立的，它们都可以自由的。地修改共享数据。于是内核便采用 COW 原则为被进程修改了的共享数据建立副本。

6.3.6 交换

当内核已经可以按照需要来分配内存后，还可以通过虚拟内存管理器进一步优化内核的性能，比如可以将页交换进或交换出磁盘，大多数现代操作系统都有交换功能。在磁盘空间很大、但空闲内存已经很少的情况下，它可以解决很多问题。比如，这时候马上有一个新的块要被提交，而内存中已经没有空闲页了。内核需要腾出一些空间来给新的页，毕竟该页马上就要被访问，而内存中有的块已经闲置了很长一段时间了，在这种情况下可以使用交换技术将老的页面换出内存。为了实现交换功能，需要完成以下的事情：

(1) 找到一个合适交换出的内存块。通常是采用 LRU 原则，但是 Windows 9x 是随机选取内存块。

(2) 用交换文件为需要交换出的内存块腾出足够的空间。一个固定大小的连续文件是最为简单的。记录下交换文件扇区号并直接告诉磁盘管理器。当然，也可以像访问普通文件那样通过文件管理系统来访问交换文件（交换文件需要很高的性能，如果可以绕过文件系统而直接使用下层的驱动，将大大提高交换系统性能）。

(3) 将旧的内存块写入磁盘，并记录下其已经被交换出内存。

(4) 重复过程(1)直到分配请求成功完成。

注意：如果上述任何一个过程出错，提交的分配请求就不能实现。当交换出内存的块再次被访问到时，它必需重新被交换进来。

6.3.7 处理流程

遇到无效地址时，一个典型的处理流程如表 6-3 所示。

表 6-3 遇到无效地址的处理流程

步骤			处理工作
1.			处理器触发页失效中断
2.			控制权转移到内核，其调用一个页失效处理程序
3.			页失效处理程序检查当前进程所分配的块
	3.1.		如果失效地址有一个已分配的块，则调用分配函数
	3.2.		分配函数为每一页分配一物理页

	3.3.		如果该块在交换文件中，则将它的每一页读出来
	3.4.		依据该块上一次分配时所设定的保护标志位，在当前进程的地址空间中建立关联映射
	3.5.		控制权转移回应用程序
4.			页失效处理程序检查当前进程的可执行模块
	4.1.		在失效地址区发现了模块，则调用动态装载函数
	4.2.		动态装载函数找到模块位于失效地址区中的部分
	4.2.		动态装载函数找到该部分位于另一进程地址区的副本
		4.2a.1	找到了副本，引用量加 1
		4.2a.2	该副本的内存空间被映射到当前地址并设定为只读态。以后的写入操作会产生另一页面错误，并产生又一副本
		4.2b.1	在其他地方并没有副本
		4.2b.2	给该部分分配物理内存，并映射到当前地址空间中
		4.2a.3	该部分从磁盘调入内存中
	4.3.		控制转移回应用程序
5.			找不到有效块，当前进程被终止或调用例外处理程序

到现在为止，本节已经介绍了较为成熟的虚拟存储管理器的基本功能，一个成熟的虚拟存储管理器是一个操作系统的基本组成部分。、

6.4 存储管理系统代码实例

本节将介绍关于存储管理系统的一些编程实例，这些例子可以很好的演示本章中介绍的存储管理机制，同时这些代码通过简单的修改，就可以使用在读者自己的操作系统中。

6.4.1 实例 1：探测计算机的物理内存容量

操作系统必须知道系统物理内存的容量，才能够有效地使用和管理这些物理内存。在 CPU 处于实模式时，CPU 可以访问的物理内存最大只能达到 1MB+64KB（在 A20 Gate 被打开的情况下，否则最大只能访问 1MB）。因此，在实模式下通常无法直接通过内存访问来获取内存容量。

BIOS INT 15h 中断提供了三个子中断来获得系统的物理内存大小，分别是 88h、E801h、和 E820h。

88h 方法在 Inte l80286 出现的那天起就存在了，后续的 PC 及其兼容机为了保持向下兼容，都继续保留了这种方法。因此，这种方法在所有的 IBM PC 及其兼容机上都存在。88h 方法虽然兼容性较好，但是存在一个重要的缺陷，由于这种方法是在 16 位机时代就

存在的，所以，它通过 16 位寄存器来保存内存容量作为返回值。但 16 位机所能够表示的最大值是 64KB。由于这种方法的返回值是以 KB 为单位，所以它能够表示的系统最大物理内存容量为 64 MB。而对于今天的 PC 机来说，64MB 已经是很低的内存配置了，大多数 PC 机的实际物理内存配置都大于 64 MB。

另外，由于这种方法出现于 Intel 80286 时代，而 Intel 80286 的 24 位地址总线能够访问的最大地址为 16MB。所以，尽管这种方法使用 16 位寄存器能够表示的最大内存数量为 64MB，但标准的 BIOS 只允许通过这种方法获取最大 16 MB 的物理内存数量。因此，为了能够探测计算机中安装的大于 64MB 的内存，就只能使用 INT 15h 中断的 E801h 和 E820h 功能（但并非每一台 PC 机都实现了这两种方法）。

当今主要操作系统在启动时，都使用这 3 种方法来进行内存检测。对于某些不支持 E820h 和 E801h 的 PC 上，所有的操作系统都最多只能检测到 64 MB 或 16MB 物理内存（依赖于 88h 的最大返回值限制）。

1. E820h：查询系统内存映射图

中断 E820h 只能在 Real Mode 下使用。中断 E820h 返回所有被安装在主机上的 RAM，以及被 BIOS 所保留的物理内存范围的内存映射。中断 E820h 的输入如表 6-4 所示。

表 6-4 中断 E820h 的输入

寄存器	功能	描述
EAX	功能码	E820h
EBX	后续值	放置着“后续值”，这个值是为了得到下一块物理内存段，它应该指定上一次调用此程序所返回的值，如果这是第一次调用，EBX 必须被指定为 0
ES:DI	缓冲指针	指向一个地址范围描述符结构，：BIOS 将会填充此结构
ECX	缓冲大小	缓冲指针所指向的地址范围描述符结构的大小，以字节（Byte）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，必需被 BIOS 以及调用者所支持的最小尺寸是 20B，未来的实现将会扩展此限制
EDX	标志	“SMAP”——BIOS 将会使用此标志，对调用者将要请求的系统映射信息进行校验，这些信息会被 BIOS 放置到 ES:DI 所指向的结构中

中断 E820h 的输出如表 6-5 所示。

表 6-5 中断 E820h 的输出

寄存器	功能	描述
CF	进位标志	不进位表示没有错误，否则存在错误

EAX	标志	SMAP
ES：DI	缓冲指针	返回地址范围描述符结构指针，和输入值相同
ECX	缓冲大小	B[OS 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 20B
EBX	后续	这里放置着为了等到下一个地址描述符所需要的“后续值”，这个值的实际形式依赖于具体的 BIC）S 的实现，调用者不必关心它的具体形式，只需要在下次迭代时，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，则表示它是最后一个地址范围描述符。一定注意，只有当这个后续值为零，并且 cF 没有进位时，才表示这是最后一个地址范围描述符

2. 地址范围描述符结构

地址范围描述符结构如表 6-6 所示。

表 6-6 地址范围描述符结构

偏移量	名称	描述
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 6-7 所示。
造成 BIOS 将某个内存段标记为 AddressRangeReserved 的主要原因如下：
这个地址范围包含着系统 ROM。
这个地址范围包含着被 ROM 使用的 RAM。
这个地址范围被用作系统设备内存映射。
这个地址范围由于某种原因，不适合被标准设备用作设备内存空间。

表 6-7 地址范围描述符结构中的 Type 含义

值	名称	描述
1	AddressRangeMemory	这个内存段是一段可以被操作系统使用的 RAM
2	AddressRangeReserved	这个地址段正在被使用，或者被系统保留，所以一定不要被操作系统使用
Other	Undefined	保留为未来使用，任何其他值都必需被操作系统认为是 AddressRangeReserved

3. 假设和限制

下面是中断 E820h 使用中必须满足的条件和一些限制：

BIOS 返回那些描述大块内存的地址范围，随后是 ISA / PCI 内存。

BIOS 不返回被用作 PCI 设备和 ISA 设备，以及 ISA plus&play 卡使用的内存映一射，这是因为操作系统有相应的机制可以检测到它们。

BIOS 返回芯片定义的地址空洞，这些地址作为保留不会被设备使用。

定义的大块内存被映射到 IO 设备的地址范围将作为保留地址将会被返回。系统 BIOS 使用的所有内存将会被作为保留内存返回，这包括低于 1MB 的内存，在 16MB（如果存在）处的内存，以及在地址空间（4GB）结尾处的内存。

标准的 PC 地址范围不会被报告，例如地址 A0000～BFFFF 被用作视频显示使用的内存；从 E0000～EFFFF 的内存是主板指定的，将会被报告。

所有的低位内存作为正常的内存将会被报告，处理为规范保留的标准 RAM 是操作系统的责任。例如，中断向量表（0:0）以及 BIOS 数据区（40:0）。

4. 一个地址映射例子

IBM PC 计算机的物理内存布局如表 6-8 所示。表 6-8 所示的物理内存分布是所有的 x86 计算机都遵循的标准。

表 6-8　IBM PC 计算机物理内存布局

线性地址	实模式地址	内存类型	用途
0～3FF	0000:0000～0000:03FF	RAM	实模式中断向量表（real-mode interrupt vector table，IVT）
400～4FF	0040:0000～0040:00FF		BIOS 数据区（BIOS data area，BDA）
500～9FBFF	0050:0000～9000:FBFF		1MB 以下的自由内存（conventional memory）
9FC00～9FFFF	9000:FC00～9000:FFFF		扩展 BIOS 数据区（extended BIOS data area，EBDA）
A0000～BFFFF	A000:0000～B000:FFFF	videoRAM	VGA 视频卡的 framebuffers
C0000～C7FFF	C000:0000～C000:7FFF	ROM	BIOS 视频缓冲区（video BIOS，通常是 32KB）
C8000～EFFFF	C800:0000～E000:FFFF	NOTHING	
F0000～FFFFF	F000:0000～F000:FFFF	ROM	主板 BIOS（Motherboard BIOS，典型大小是 64KB）

100000～FEBFFFFF		RAM	1MB 以上的只有内存（free extended memory）
FEC00000～FFFFFFFF		Various	主板 BIOS（motherboard BIOS，例如 PnP NVRAM，ACPI 等使用）

表 6-9 所示是一台 128MB 内存的计算机内存地址映射。这台计算机具有 640KB 的基本内存，以及 127MB 的扩展内存。在 640KB 的基本内存中，639KB 归用户使用，1KB 作为扩展 BIOS 数据区。以 12MB 位置为起始，存在一个 LFB（Liner Frame Buffer），被芯片创建的内存空洞是从 8MB～16MB，这个内存空洞是 APIC 设备的内存映射。I/O 单元处于 FEC00000，本地单元处于 FEE00000，系统的 BIOS 被重映射到（4GB～64KB）的位置上。

注意：第一块内存，也就是基本内存的 639KB 的终点位置，被报告在 BIOS 数据段 40:13。“ARM”表示 Address Range Memory，“ARR”是 Address Range Reserved。

表 6-9　一台 128MB 内存的计算机的内存地址映射

基地址	L 长度	类型	描述
0000:0000	639K	ARM	可以使用的基本内存（也就是通过 INT 12h 获取的内存容量）
0009:FC00	1K	ARR	为 BIOS 保留的内存；这个区域包括扩展 BIOS 数据区
000F:0000	64K	ARR	系统 BIOS
0010:0000	7M	ARM	扩展内存，它没有 64MB 的地址范围限制
0080:0000	8M	ARR	芯片内存空洞，用于支持在 12MB 位置的 LFB 映射
0100:0000	120M	ARM	在芯片内存空洞之上的大块内存 RAM
FEC0:0000	4K	ARR	被映射到 FEC00000 的 IO APIC 内存，注意不同厂商的 APIC 所需要的地址范围可能不一样
FEE0:0000	4K	ARR	本地 APIC 内存映射
	64K	ARR	映射的系统 BIOS

5. E801h：获得系统的存储容量

中断 E801h 只能在 Real Mode 下使用。最初，这种方式是为 EISA 服务定义的，这个接口能够报告多达 4GB 的 RAM。然而它不像 E820h 方式那么通用，E820h 在更多的系统上可以使用，一些系统没有实现中断 E801h。中断 E801h 的输入如表 6-10 所示。

表 6-10　中断 E801h 的输入参数

寄存器	功能	描述

AX	功能码	E801h

中断 E801h 的输出如表 6-11 所示。

表 6-11　中断 ES01h 的输出参数

寄存器	功能	描述
CF	进位标志	不进位表示没有错误
AX	扩展 1	1KB～16MB 内存的容量，以 KB 为单位，最大数量 0x3C00=15MB
BX	扩展 2	16MB～4GB 之间的内存容量，以 64KB 为单位
CX	配置 1	1KB～16MB 内存的容量，以 KB 为单位，最大数量 0x3C00=15MB
DX	配置 2	16MB～4GB 之间的内存容量，以 64KB 为单位

注：在表 6-11 中的“扩展”和“配置”之间没有区别，事实上它们的值是相同的。

6.　88h：获得系统内存容量

中断 E88h 只能在实模式下使用。这个接口是相当原语性的，它返回 1MB 地址以上的后续内存容量。最大的限止是它的返回值是 16 位的，以 KB 为单位，所以它最多能够返回 64MB。在某些系统上，它仅仅能够返回 16MB 以内的内存。和前两者相比，它的突出好处是它在所有的 Pc 上都工作。中断 E88h 的输入如表 6-12 所示。

表 6-12　中断 E88h 的输入参数

寄存器	功能	描述
AH	功能码	88h

中断　E88h 的输出如表 6-13 所示。

表 6-13　中断 E88h 的输出参数

寄存器	功能	描述
CF	进位标志	不进位表示没有错误
AX	内存容量	以 KB 为单位，1MB 以上的内存容量

下面的例子程序使用上述的 BIOS 调用来获取系统中安装的物理内存容量，代码如下：

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; demo of BIOS calls that check memory size
; This code is public domain (no copyright).
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
; You can do whatever you want with it.
;
; This code uses these interrupts:
;       1. INT 15h AX=E820h (32-bit CPU only)
;       2. INT 15h AX=E801h
;       3. INT 15h AH=88h
;       4. INT 12h
;
; assemble with NASM:  nasm -f bin -o biosmem.com biosmem.asm
;
; rewritten Sep 6, 2001
; - trying to handle possible BIOS bugs per Ralf Brown's list and
;       http://marc.theaimsgroup.com/?l=linux-kernel&m=99322719013363&w=2
; - now displaying memory block info, instead of just a size value
; - new wrnum function
;
; xxx - INT 15h AX=E820h shows a 1K block at the top of conventional
; memory for my system -- is this reserved for the EBDA?
; I don't _have_ an EBDA, at least, not with my current CMOS settings...
; will the 1K block disappear if the EBDA is enabled?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        ORG 100h

        mov si,mem_msg
        call cputs

; check for 32-bit CPU
        call cpu_is_32bit

; try INT 15h AX=E820h
        or ah,ah
        je cant_e820
        call extmem_int15_e820
        jnc ok

; before trying other BIOS calls, use INT 12h to get conventional memory size
cant_e820:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        int 12h

; convert from K in AX to bytes in CX:BX
        xor ch,ch
        mov cl,ah
        mov bh,al
        xor bl,bl
        shl bx,1
        rcl cx,1
        shl bx,1
        rcl cx,1

; set range base (in DX:AX) to 0 and display it
        xor dx,dx
        xor ax,ax
        call display_range

; try INT 15h AX=E801h
        call extmem_int15_e801
        jnc ok

; try INT 15h AH=88h
        call extmem_int15_88
        jnc ok

; uh-oh
        mov si,err_msg
        call cputs

; exit to DOS
ok:
        mov ax,4C00h
        int 21h

got_32bit_cpu:
        db 0
mem_msg:
        db "Memory ranges:"
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
crlf_msg:
        db 13, 10, 0
base_msg:
        db "      base=0x", 0
size_msg:
        db ", size=0x", 0
err_msg:
        db "*** All BIOS calls to determine extended memory size "
        db "have failed ***", 13, 10, 0

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; name:                 cpu_is_32bit
; action:       checks for 32-bit CPU
; in:           (nothing)
; out:                  AH != 0 if 32-bit CPU
; modifies:             AX
; minimum CPU:          8088
; notes:        C prototype: extern int cpu_is_32bit(void);
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

cpu_is_32bit:
        push bx
                pushf
                        pushf
                        pop bx          ; old FLAGS -> BX
                        mov ax,bx
                        xor ah,70h      ; try changing b14 (NT)...
                        push ax         ; ... or b13:b12 (IOPL)
                        popf
                        pushf
                        pop ax          ; new FLAGS -> AX
                popf
                xor al,al       ; zero AL
                xor ah,bh       ; 32-bit CPU if we changed NT...
                and ah,70h              ; ...or IOPL
        pop bx
        ret
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; name:           display_range
; action:     printf("base=0x%lX, size=0x%lX\n", DX:AX, CX:BX);
; in:         (nothing)
; out:            (nothing)
; modifies:       (nothing)
; minimum CPU:        8088
; notes:
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

display_range:
    push si
    push dx
    push bx
    push ax

; if size==0, do nothing
        mov si,cx
        or si,bx
        je display_range_1
        mov si,base_msg
        call cputs
        push bx
            mov bx,16
            call wrnum
            mov si,size_msg
            call cputs
        pop ax
        mov dx,cx
        call wrnum
        mov si,crlf_msg
        call cputs
display_range_1:
    pop ax
    pop bx
    pop dx
    pop si
    ret
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; name:             extmem_int15_e820
; action:           gets extended memory info using INT 15h AX=E820h
; in:               (nothing)
; out:              (nothing)
; modifies:         (nothing)
; minimum CPU:      386+
; notes:
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

buffer_e820:
        times 20h db 0
buffer_e820_len     equ $ - buffer_e820

extmem_int15_e820:
        push es
        push di
        push edx
        push ecx
        push ebx
        push eax
                push ds
                pop es
                mov di,buffer_e820
                xor ebx,ebx             ; INT 15h AX=E820h continuation value

                mov edx,534D4150h       ; "SMAP"
                mov ecx,buffer_e820_len
                mov eax,0000E820h
                int 15h

; CY=1 on first call to INT 15h AX=E820h is an error
                jc extmem_e820_4
extmem_e820_1:
                cmp eax,534D4150h       ; "SMAP"

; return EAX other than "SMAP" is an error
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
                stc
                jne extmem_e820_4
                cmp dword [es:di + 16],1 ; type 1 memory (available to OS)
                jne extmem_e820_2
                push bx
                        mov ax,[es:di + 0] ; base
                        mov dx,[es:di + 2]
                        mov bx,[es:di + 8] ; size
                        mov cx,[es:di + 10]
                        call display_range
                pop bx
extmem_e820_2:
                or ebx,ebx
                je extmem_e820_3

; "In addition the SMAP signature is restored each call, although not
; required by the specification in order to handle some known BIOS bugs."
;       -- http://marc.theaimsgroup.com/?l=linux-kernel&m=99322719013363&w=2
                mov edx,534D4150h       ; "SMAP"
                mov ecx,buffer_e820_len
                mov eax,0000E820h
                int 15h

; "the BIOS is permitted to return a nonzero continuation value in EBX
;    and indicate that the end of the list has already been reached by
;    returning with CF set on the next iteration"
;       -- b-15E820 in Ralf Brown's list
                jnc extmem_e820_1
extmem_e820_3:
                clc
extmem_e820_4:
        pop eax
        pop ebx
        pop ecx
        pop edx
        pop di
        pop es
        ret
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; name:                extmem_int15_e801
; action:          gets extended memory size using INT 15h AX=E801h
; in:              (nothing)
; out:                 (nothing)
; modifies:            (nothing)
; minimum CPU:         8088 for code, 286+ for extended memory
; notes:
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

extmem_int15_e801:
        push dx
        push cx
        push bx
        push ax
                mov ax,0E801h

; "...the INT 15 AX=0xE801 service is called and the results are sanity
; checked. In particular the code zeroes the CX/DX return values in order
; to detect BIOS implementations that do not set the usable memory data.
; It also handles older BIOSes that return AX/BX but not AX/BX data." (?)
;       -- http://marc.theaimsgroup.com/?l=linux-kernel&m=99322719013363&w=2
                xor dx,dx
                xor cx,cx
                int 15h
                jc extmem_e801_2
                mov si,ax
                or si,bx
                jne extmem_e801_1
                mov ax,cx
                mov bx,dx
extmem_e801_1:
                push bx

; convert from Kbytes in AX to bytes in CX:BX
                        xor ch,ch
                        mov cl,ah
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
                mov bh,al
                xor bl,bl
                shl bx,1
                rcl cx,1
                shl bx,1
                rcl cx,1

; set range base (in DX:AX) to 1 meg and display it
                mov dx,10h
                xor ax,ax
                call display_range

; convert stacked value from 64K-blocks to bytes in CX:BX
        pop cx
        xor bx,bx

; set range base (in DX:AX) to 16 meg and display it
        mov dx,100h
        xor ax,ax
        call display_range
extmem_e801_2:
    pop ax
    pop bx
    pop cx
    pop dx
    ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; name:             extmem_int15_88
; action:       gets extended memory size using INT 15h AH=88h
; in:           (nothing)
; out:              (nothing)
; modifies:         (nothing)
; minimum CPU:          8088 for code, 286+ for extended memory
; notes:        HIMEM.SYS will hook this interrupt and make
;               it return 0
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
extmem_int15_88:
        push dx
        push cx
        push bx
        push ax
                mov ax,8855h
                int 15h

; "not all BIOSes correctly return the carry flag, making this call
;    unreliable unless one first checks whether it is supported through
;    a mechanism other than calling the function and testing CF"
;       -- b-1588 in Ralf Brown's list
; test if AL register modified by INT 15h AH=88h
                cmp al,55h
                jne extmem_int15_1
                mov ax,88AAh
                int 15h
                cmp al,0AAh
                stc
                je extmem_int15_2

; convert from Kbytes in AX to bytes in CX:BX
extmem_int15_1:
                xor ch,ch
                mov cl,ah
                mov bh,al
                xor bl,bl
                shl bx,1
                rcl cx,1
                shl bx,1
                rcl cx,1

; set base to 1 meg and display range
                mov dx,10h
                xor ax,ax
                call display_range
extmem_int15_2:
        pop ax
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        pop bx
        pop cx
        pop dx
        ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; name:         cputs
; action:   writes 0-terminated string to screen
; in:       SI -> string
; out:          (nothing)
; modifies:     (nothing)
; minimum CPU:    8088
; notes:
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

cputs:
        push si
        push bx
        push ax
                cld             ; string operations go up
                mov ah,0Eh          ; INT 10h: teletype output
                xor bx,bx       ; video page 0
                jmp short cputs_2
cputs_1:
                int 10h
cputs_2:
                lodsb
                or al,al
                jne cputs_1
        pop ax
        pop bx
        pop si
        ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; name:             wrnum
; action:       writes 32-bit value to text screen
; in:           32-bit unsigned value in DX:AX, radix in BX
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
; out:                  (nothing)
; modifies:             (nothing)
; minimum CPU:          8088
; notes:
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        times 40 db 0
num_buf:
        db 0

wrnum:
        push si
        push dx
        push cx
        push bx
        push ax
                mov si,num_buf

; extended precision division from section 9.3.5
; of Randall Hyde's "Art of Assembly"
; start: DX=dividend MSW, AX=dividend LSW, BX=divisor
wrnum1:
                push ax
                        mov ax,dx
                        xor dx,dx

; before div: DX=0, AX=dividend MSW, BX=divisor
; after div:    AX=quotient MSW, DX=intermediate remainder
                        div bx
                        mov cx,ax
                pop ax

; before div: DX=intermediate remainder, AX=dividend LSW, BX=divisor
; after div:    AX=quotient LSW, DX=remainder
                div bx

; end: DX=quotient MSW, AX=quotient LSW, CX=remainder
                xchg dx,cx
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            add cl,'0'
            cmp cl,'9'
            jbe wrnum2
            add cl,('A'-('9'+1))
wrnum2:
            dec si
            mov [si],cl

            mov cx,ax
            or cx,dx
            jne wrnum1
            call cputs
        pop ax
        pop bx
        pop cx
        pop dx
        pop si
        ret
```

在一台安装有 128MB 物理内存的计算机上运行的效果如图 6-1 所示。
从图 6-1 可以看到：
0x9F800 = 653312
0x100000=1048576
0x7DF0000=132055040
0x7f00000=133169152
上面的显示表明了这个程序已经探测到计算机的物理内存是 128MB。

图 6-1　128MB 内存计算机运行例子

6.4.2　实例 2：malloc()和 free()的实现

在本小节中将介绍一个简单的 malloc()函数的实现。在 malloc()函数实现中使用了首次适配方法来进行存储分配。当使用 free()函数释放内存时，会对相邻的存储空间进行合并。

这个例子中包含的算法，可以很好的应用到操作系统的内存管理机制中，带有详细注释的源代码参见光盘文件：malloc.c。

上述代码的运行效果如图 6-2 所示。

图 6-2　malloc()和 free()的运行效果

6.5　Linux 0.01 存储管理代码分析

本节将具体介绍 Linux 0.01 的存储管理代码。v 的存储管理代码比较复杂，通过对前述各节的学习，读者可以使用上述的知识来帮助理解 Linux 0.01 的存储管理代码。

6.5.1　memory.c 分析

memory.c 提供了分页存储管理使用的功能函数，包括 get_free_page()、free_page()，free_page_tables()、copy_page_tables()、put_page()、un_wp_page()，do_wp_page()、write_verify()、do_no_page()、calc_mem()等函数，具体源代码及其注释见光盘文件 memory.c。

6.5.2　page.s 分析

page.s 是一段汇编语言程序，提供了 Linux 0.01 中缺页异常处理功能。page.s 仅仅提供了和硬件存储管理器的接口功能，大部分功能代码在 memory.c 文件中。其功能如下：

```
/*
 * page.s contains the low-level page-exception code.
 * the real work is done in mm.c
 */

.globl _page_fault

_page_fault:
        xchgl %eax,(%esp)
        pushl %ecx
        pushl %edx
        push %ds
        push %es
        push %fs
        movl $0x10,%edx
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        mov %dx,%ds
        mov %dx,%es
        mov %dx,%fs
        movl %cr2,%edx
        pushl %edx
        pushl %eax
        testl $1,%eax
        jne 1f
        call _do_no_page
        jmp 2f
1:      call _do_wp_page
2:      addl $8,%esp
        pop %fs
        pop %es
        pop %ds
        popl %edx
        popl %ecx
        popl %eax
        iret
```

6.6 本章小结

存储管理是操作系统的重要功能。本章从物理存储管理、虚拟存储管理、代码实例等多个方面介绍了操作系统中存储管理的基本方法。

本章首先介绍了操作系统中存储管理的基本方法，包括连续分配存储管理、非连续存储管理机制和虚存机制。

本章同时介绍了计算机的物理存储管理机制，重点介绍了页面分配和回收、内存映射等。虚拟存储管理是一个现代操作系统必备的功能，本章介绍了操作系统实现虚拟存储管理的基本方法，包括虚拟存储管理和交换。

本章提供了一些存储管理系统代码实例，同时对 Linux 0.01 的存储管理代码进行了详细分析。

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

第7章 进程管理和调度

本章知识点：

- 进程介绍
- 多任务实现基础
- 进程调度的性能分析
- Linux 进程调度源代码分析
- 实例：实现协作多任务

本章重点介绍在 Linux 系统中进程的实现原理和管理调度方法。其中，首先介绍了进程的基本功能、进程的调度和线程的基本概念。然后，介绍了在操作系统中实现多任务的基本方法。在操作系统中，进程调度的性能是非常重要的。本章通过一个具体的实例，介绍了进程调度的性能问题。本章还详细分析了 Linux 0.01 中进程调度源代码的分析。最后，给出了一个通过 C 函数实现协作多任务的例子，可以让读者迅速地体会多任务运行的效果。

7.1 进程介绍

进程是操作系统的基本概念。本章将介绍操作系统中关于进程的实现原理。在不同的操作系统中，进程的实现可能会有较大的差异，本章重点介绍在 Liflux 下的进程实现。

7.1.1 进程的基本功能

进程是 Linux 系统中的一个非常重要的概念。在 Linux 系统中，进程具有不同的运行状态，可以产生和死亡。

1. 进程调度

在 Linux 0.01 内核中的 include/linux/sched.h 文件中，定义了进程的不同状态，如表 7-1 所示。

表 7-1 Linux 0.01 的进程状态

进程	功能

PDF-XCHANGE Click to buy NOW! www.docu-track.com

TASK_RUNNING	表示进程在‘ 'Ready List”中，这个进程除了 CPU 以外，获得了所有的其他资源
TASK_INTERRUPTIBLE	进程在睡眠中，正在等待一个信号或者一个资源（sleeping）
TASK_UNINTERRUPTIBLE	进程等待一个资源，当前进程在“ Wait Oueue ”
TASK_ZOMBIE	僵尸进程（没有父进程的子进程）
TASK_STOPPED	标识进程在被调试

表 7-1 中的进程各种状态切换关系的描述如图 7-1 所示。

图 7-1　Linux 进程的切换

从系统内核的角度来看，一个进程仅仅是进程控制表（process table）中的一项。进程控制表中的每一项都是一个 task_struct 结构，而 task_struct 结构本身是在 include/linux/sched.h 中定义的。在 task_struct 结构中存储各种低级和高级的信息，包括从一些硬件设备的寄存器复制到进程的工作目录的链接点。

进程控制表既是一个数组，又是一个双向链表，同时又是一个树。其物理实现是一个包括多个指针的静态数组。

此数组的长度保存在 include/linux/sched.h 定义的常量 NR_TASKS 中，其默认值为 64（#define NR_TASKS 64）。数组中的结构则保存在系统预留的内存页中。这个定义如下：

#define FIRST_TASK task[0]

#define LAST_TASK task[NR_TASKS-1]

系统启动后，内核通常作为某一个进程的代表。一个指向 task_struct 的全局指针变量 current 用来记录正在运行的进程。变量 current 只能由 kernel/sched.c 中的进程调度改变。

如图 7-2 所示详细地介绍了 Linux 进程切换的过程。

图 7-2　Linux 的进程切换示意图

2.　用户进程和内核线程

一个进程在一个时刻，只能运行在用户方式（user mode）或内核方式（kernel mode）。用户程序运行在用户方式下，而系统调用运行在内核方式下。在这两种方式下所用的堆栈不一样：用户方式下用的是一般的堆栈，而内核方式下用的是固定大小的堆栈（一般为一个内存页的大小）。

在 Linux 的高版本中（例如 Lmux 2.0.1 中及高版本的内核中），除了用户进程外，还

PDF-XCHANGE Click to buy NOW! www.docu-track.com

有一些内核线程。尽管 Linux 是一个单内核操作系统，内核线程依然存在，以便并行地处理一些内核中需要并发执行的任务。这些任务不占用 USER memory（用户空间），而仅仅使用 KERNEL memory。和其他内核模块一样，这些内核线程也在高级权限（Intel 386 系统中的 RING 0）下执行。内核线程在 Linux 2.0.0 的核心中就可以被看到，例如在 linux-2.0.1/linux/arch/alpha/kernel/entry.S 中的代码如下：

```
/*
 * __kernel_thread(clone_flags, fn, arg)
 */
.align 3
.globl      __kernel_thread
.ent  __kernel_thread
__kernel_thread:
        subq $30,4*8,$30
        stq $9,0($30)
        stq $10,8($30)
        stq $26,16($30)
        bis $17,$17,$9        /* save fn */
        bis $18,$18,$10             /* save arg */
        bsr $26,kernel_clone
        bne $20,1f            /* $20 is non-zero in child */
        ldq $9,0($30)
        ldq $10,8($30)
        ldq $26,16($30)
        addq $30,4*8,$30
        ret $31,($26),1
/* this is in child: look out as we don't have any stack here.. */
1:      bis $9,$9,$27         /* get fn */
        bis $10,$10,$16             /* get arg */
        jsr $26,($27)
        bis $0,$0,$16
        lda $27,sys_exit
        jsr $26,($27),sys_exit
        call_pal PAL_halt
.end __kernel_thread
```

一旦调用，就有了一个新的任务（Task）（一般 PID 都很小，例如 2、3 等）等待一个响应很慢的资源，例如 swap 或者 usb 事件，以便同步。如图 7-3 所示是一些最常用的内核线程（可以用 ps -x 命令）。、

图 7-3　Linux 中的常见进程

3.　进程创建，运行和消失

Linux 系统使用系统调用 fork()来创建一个进程，使用 exit()来结束进程。fork()和 exit()的源程序保存在 kernel/fork.c 和 kernel/exit.c 中。fork()的主要任务是初始化要创建进程的数据结构，其主要的步骤有：

(1)　申请一个空闲的页面来保存 task_struct。
(2)　查找一个空的进程槽（find_empty_process()）。
(3)　为 kernel_stack_page 申请另一个空闲的内存页作为堆栈。
(4)　将父进程的 LDT 表复制给子进程。
(5)　复制父进程的内存映射信息。
(6)　管理文件描述符和链接点。

撤销一个进程可能稍微复杂些，因为撤销子进程必须通知父进程。另外，使用 kill()也可以结束一个进程。

使用 fork()创建一个进程后，程序的两个复制都在运行。通常一个复制使用 exee()调用系统的应用程序。系统调用 exee()负责定位一个可执行文件的二进制代码，并负责将其装入和运行。

7.1.2　进程的调度（schedule()函数）

处于 TASK_RUNNING 状态的进程移到运行队列（runqueue），将由 schedule()函数按 CPU 调度算法在合适的时候被选中运行，分配给 CPU。

新创建的进程都是处于 TASK_RUNNING 状态，而且被挂到 run queue 的队首。进程调度采用变形的轮转法（round robin）。当时间片到时（10ms 的整数倍），由时钟中断引起新一轮调度，把当前进程挂到 run queue 队尾。

所有的进程中一部分运行于用户态，另一部分运行于系统态。进程运行在用户态比运行在系统态时的权限低了很多。每一次进程执行一个系统调用，它都从用户态切换到系统态并继续执行，这时让核心执行这个进程。在 Linux 中，每一个进程在它必须等待一些系统事件时会放弃 CPU。例如，一个进程可能不得不等待从一个文件中读取一个字符。这个等待发生在系统调用中。进程使用了库函数打开并读文件，库函数又执行系统调用从打开的文件中读入字节。这时，等候的进程会被挂起，另一个适合运行的进程将会被选择执行。进程在执行的过程中，通常会调用系统调用，所以经常需要等待。即使进程执行到需要等待也有可能会用去不均衡的 CPU 事件，所以 Linux 使用抢先式的调度。用这种方案，每一个进程允许运行少量一段时间（200ms），当这个时间过去，选择另一个进程运行，原来的进程等待一段时间直到它又重新运行，这个时间段叫做时间片。

调度程序的一个重要工作就是选择系统中所有可以运行的进程中最适合运行的进程

加以运行。一个可以运行的进程是一个只等待 CPU 的进程。Linux 使用合理而简单的基于优先级的调度算法在系统当前的进程中进行选择。当它选择了准备运行的新进程，它就保存当前进程的状态、与处理器相关的寄存器和其他需要保存的上下文信息到进程的 task_struet 数据结构中，然后恢复要运行的新进程的状态（和处理器相关），把系统的控制交给这个进程。为了公平地在系统中所有可以运行（runnable）的进程之间分配 CPU 时间，调度程序在每一个进程的 task_struet 结构中保存了信息。

1. policy 进程的调度策略

Linux 有两种类型的进程：普通进程和实时进程（只有在支持实时调度的 Linux 系统中才有实时进程，普通 Linux 系统中没有实时进程）。普通进程具有一般优先级，由核心按照预定的进程调度算法来调度执行。实时进程比所有其他进程的优先级高，如果有一个实时进程准备运行，那么它总是先被运行，不管是否有其他普通进程正在运行。

实时进程有两种策略：环或先进先出（round robin and first in first out）。在环的调度策略下，每一个实时进程依次运行；而在先进先出的策略下，每一个可以运行的进程按照它在调度队列中的顺序运行，这个顺序不会改变。

调度程序 schedule()从核心的多个地方运行。它可以在把当前进程放到等待队列之后运行，也可以在系统调用之后进程从系统态返回用户态之前运行。需要运行调度程序的另一个原因是系统时钟刚好把当前进程的计数器（counter）置成了 0。每一次调度程序运行时都做以下工作：

- kerel work：调度程序运行 bottom hal fhandler 并处理系统的调度任务队列。
- current pocess：在选择另一个进程之前必须处理当前进程。
- 如果当前进程的调度策略是环则它放到运行队列的最后。
- 如果任务状态是 TASK_INTERRUPTIBLE 并且它上次调度时收到过一个信号，它的状态变为 TASK_RUNNING；如果当前进程超时，它的状态成为 RUNNING；如果当前进程的状态为 RUNNING，则保持此状态；不是 RUNNING 或者 INTERRUPTIBLE 的进程将被从运行队列中删除。这意味着当调度程序查找最值得运行的进程时不会考虑除 RUNNING 和 INTERRUPTIBLE 以外的进程。
- Process Selection：调度程序查看运行队列中的进程，查找最适合运行的进程。如果有实时的进程（具有实时调度策略），就会比普通进程更重一些。普通进程的重量是它的 counter，但是对于实时进程则是 counter 加 1000。这意味着如果系统中存在可运行的实时进程，就总是在任何普通可运行的进程之前运行。如果几个进程有同样的优先级，最接近运行队列前段的那个就被选中。当前进程被放到运行队列的后面。如果一个平衡的系统，拥有大量相同优先级的进程，那么会按照顺序执行这些进程，这叫做环型调度策略。不过，因为进程需要等待资源，它们的运行顺序可能会变化。
- Swap Processes：如果最适合运行的进程不是当前进程，当前进程必须被挂起，从而运行新的进程。当一个进程运行时使用了 CPU、系统寄存器和物理内存。每一次它调用例程都通过寄存器或者堆栈传递参数、保存数值比如调用例程的

返回地址等。因此，当调度程序运行时它在当前进程的上下文运行。它可能是特权模式：核心态，但是它仍旧是当前运行的进程。当这个进程要挂起时，它的所有机器状态，包括程序计数器（PC）和所有的处理器寄存器，必须存到进程的 task_struct 数据结构中。然后，必须加载新进程的所有机器状态。这种操作依赖于系统，不同的 CPIJ 不会完全相同地实现，不过经常都是通过一些硬件的帮助。

- 交换出去进程的上下文发生在调度的最后。前一个进程存储的上下文，就是当这个进程在调度结束时系统的硬件上下文的快照。相同的，当加载新的进程的上下文时，仍旧是调度结束时的快照，包括进程的程序计数器和寄存器的内容。
- 如果前一个进程或者新的当前进程使用虚拟内存，则系统的页表需要更新。Alpha AXP 处理器，使用 TLT（Translation Look-aside TabIe）或者缓存的页表条目，必须清除属于前一个进程的缓存的页表条目。

2. 进程描述符。

进程描述符：也就是结构体 task_struct，它有很多域，包含了一个进程的所有信息，主要有它的属性、当前的状态、它所占有的资料和一些用于链接进程描述符结构的指针。

在 Linux 核心中，有一个全局的进程数组来保存所有进程的进程描述符结构，这个数组就是：struct tasm_struct *task[NR_TASKS]。它是一个指针数组，其中 NR_TASKS 在 Linux 0.01 中是 64，代表 Linux 0.01 中最多可以有 64 个进程。

结构 task_struct 定义如下：

```
struct task_struct {
/* these are hardcoded - don't touch */
    long state;      /* -1 unrunnable, 0 runnable, >0 stopped */
    long counter;
    long priority;
    long signal;
    fn_ptr sig_restorer;
    fn_ptr sig_fn[32];
/* various fields */
    int exit_code;
    unsigned long end_code,end_data,brk,start_stack;
    long pid,father,pgrp,session,leader;
    unsigned short uid,euid,suid;
    unsigned short gid,egid,sgid;
    long alarm;
    long utime,stime,cutime,cstime,start_time;
    unsigned short used_math;
/* file system info */
```

```
        int tty;            /* -1 if no tty, so it must be signed */
        unsigned short umask;
        struct m_inode * pwd;
        struct m_inode * root;
        unsigned long close_on_exec;
        struct file * filp[NR_OPEN];
/* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
        struct desc_struct ldt[3];
/* tss for this task */
        struct tss_struct tss;
};
```

7.1.3 线程

线程可以被描述为“轻型进程”。线程的实现有两种形式：核心级线程和用户级线程。

1. 用户级线程

用户级线程（User-leveI threads）是由与应用程序链接的线程库实现。核心不知道线程的存在，也就不能独立调度这些线程了。因此，线程的调度是由线程库来完成的。如果一个线程调用了一个阻塞的系统调用，则整个进程可能被阻塞，当然其中的所有线程也同时被阻塞，所以 Unix 使用了异步 I/O 工具。这种机制的最大缺点是不能发挥多处理器的优势。

用户级线程的优点是系统消耗小。用户级线程比起进程而言可以有效地减少系统资源的占用。因为，多个用户线程通常都属于一个进程空间，线程的调度并不需要陷入核心。因此，用户级线程的切换是非常迅速的。用户级线程的实现可以支持许多应用场合，例如实时多媒体的处理系统。在一个系统中，可以在一个核心中支持非常多数量的用户级线程。

2. 核心级进程

核心级线程（Kernel-level threads）允许不同进程里的线程按照同一调度方法进行调度（其调度方法和其他进程一致），这适合于发挥多处理器的并发优点。

大多数的线程库实现的是用户级的线程。同时，也有一些操作系统实现了核心级线程，还有同时实现了核心级线程和用户级线程的，例如 Solaris、Mach。

7.2 多任务实现基础

本节将介绍如何在一个操作系统中实现多任务功能。本节主要介绍基于堆栈的多任

务系统（a stack-based multitasking subsystem）。

多任务系统可以划分为单处理器多任务系统和多处理器多任务系统。基于单处理器的多任务处理系统的主要功能是：在很快的速度内在多个进程之间切换（每个进程占用处理器一个时间段，或者直到占用处理器的进程主动释放 CPU）。

首先，为了完成多个进程之间的切换，需要定义一个进程描述结构来描述在操作系统中与一个进程相关的主要信息，其代码如下：

```
typeddstructI
um ‘ t～tpmzess—esp；  /  / esp • 寄存器实际位置
u ’ trlt—tpmzeas—SS；  /  / 堆栈寄存器的实际位置
uim～tpmzess—kstACK；  /  / 内核的栈顶
u ’ Int～tpmzess—ustACK；  / 佣户的栈顶
Ul ‘ rll：～tpmzess—cr3；
Ul ‘ III：～rpmzess—number；
kll ‘ rl[～tpmzess—parent；
ul ’ nt～tpmzess—owrw2"|
urn ‘ [～tpmzess—group；
ul ‘ n[一 tpmzess—timetorun；
HI ’ nE～tprozess—sleep；
ujml：一 j12pmzess_priority、
Ul ’ nl：一 tpmzess—filehandle；，，console—t ” pmzess_console；
1 / memorymanagementinfoconcerningtheprocess
meiTllWee—t*vmnl—alloe；＼1 煎撙蒋谎器。_uchar—tpmzess—haitie[32]0
Ipmzess—t；薯曩。
```

在上述定义的结构中，有 5 个非常重要的数据：esp，ss，kstack，ustack，cr3。这 5 个数据结构中，除了 esp 以外，都必须使用汇编语言来进行处理。esp 代表了一个进程在处理器中执行时 ESP 寄存器的实际位置。当一个进程执行到这个地方时，进程可能被 ISR 或者系统调用中断。这些过程通常使用汇编语言来进行处理。

为了实现多进程系统，还需要保存一个 TSS 结构。在每次进程切换时，TSS 都被更新来记录下一个进程的核心堆栈地址。TSS 在 C 语言里面的定义如下：

```
struct tss_struct {
        long back_link;      /* 16 high bits zero */
        long esp0;
        long ss0;        /* 16 high bits zero */
        long esp1;
        long ss1;        /* 16 high bits zero */
        long esp2;
        long ss2;        /* 16 high bits zero */
        long cr3;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        long eip;
        long eflags;
        long eax,ecx,edx,ebx;
        long esp;
        long ebp;
        long esi;
        long edi;
        long es;        /* 16 high bits zero */
        long cs;        /* 16 high bits zero */
        long ss;        /* 16 high bits zero */
        long ds;        /* 16 high bits zero */
        long fs;        /* 16 high bits zero */
        long gs;        /* 16 high bits zero */
        long ldt;       /* 16 high bits zero */
        long trace_bitmap;  /* bits: trace 0, bitmap 16-31 */
        struct i387_struct i387;
};
```

接下来，要实现正确的中断处理程序（ISR）

%macro—REG—SAVE0
old
pushad?
pushds
pushes
pushfs
pushgs
moveax，[p]；放置当前进程结构的地址在 eax，（指针 P 的地址）
mov[eax]，esp；保存 esp 在当前进程结构里面的 esp • 中
leaeax，[kstaekend]；切换到核心自己的堆栈中
movesp，eax，
%endmaero：%macroPEG—REsToRE 一。MASTER0
moveax，[P]；设置当前进程结构的地址到 eax 中
■■j，ii_0j0（000 篱 000_00—00 · 150 · LinuxO，01 内核分析与操作系统设计
movesp，[eax]；恢复 esl：l 的地址。
NOVebx，[eax+8]；放置核心堆栈的内容到 ebx
mov[sys—tss+4]，ebx；更新系统 TSS
moval，0x20
OUI0x20，al
popgs

PDF-XCHANGE Click to buy NOW! www.docu-track.com

popfs
popes
popas
popad
iretd
%endmaero 一～～＼一

下面是一个 ISR（Interrupt Service Routing，中断服务程序）的例子，其实现代码如下：

[EXTERNtimer—handler]
[GLOBALhwint00]
hwint00：
REG—SAVE
calltimer—handler
KEG—RESTORE—MASTER

在上面的程序中，把当前进程结构的地址放置在 eax 寄存器中，然后访问 / 设置其他的域。esp 在 TSS 结构的第一个位置，因此不需要偏移量。其他的成员，必须在 eax 寄存器指向的地址上加上一个偏移量才能访问。

完成这些操作后，就可以告诉 CPU，下一个将要运行的进程的核心堆栈在什么位置。在离开 ISR 的时候，它从当前进程结构里取得新的 ESP 值，使用新的核心堆栈的顶部地址来替换 TSS 中的 ESP0 字段，然后弹出所有的寄存器值，返回被中断的进程。假如这个进程是一个用户进程，在 iret 指令（Interrupt Return 指令，中断返回指令）执行前弹出的两个数据是用户堆栈段地址和用户堆栈（user-stack-segment and user-stack）。

每次任务切换时需要填充 TSS 中的 ESP0 域的原因：主要是因为运行在 dpl3（User Level）的用户进程，能够通过发出一个中断或者访问一个系统调用进入核心级进程（=dpl0：system level）。通常情况下，使用软件中断的方式更为可取，这个过程如下：

- intxx：dpl3→dpl0：CPU 切换到由 TSS，ESP0 的值指定的堆栈。
- iret：dpl0→dpl3：在执行 iret 指令之前将弹出寄存器的值。
- iret 指令：恢复 cflags、cs、eip、user stack segment、user stack address 等寄存器。

这些从用户级进程→核心级进程→用户级进程的转换发生在每次中断发生时。

注意：无论是软件中断还是硬件中断，都要发生这个转换。

下面来研究一下如何填充进程结构和核心堆栈，其代码如下：

pmzess—t*task—anlegen（entry—tentry，intpriority）{
/ / variousinitializationstuffmaybedonehe'rereachingthebelowde—scribedoperations？
第 7 章进程管理和调度-151
/ / fillinginthekstACKforstartup ： uint — t*stACKsetup ； / / ills 时指针
--temporm'ypointer—willbesettothe
/
[kstACKtopandafterwardssavedinespofprocstructurestACKsetup=&kstACKs[d][KSTACKTO

P] ” stACKsetup 一一；” stACKsetuD 一一 ” stACKsetuD 一一=0x0202；=0x08；=（uint—t）entry； / / 这个是进程执行人口点的地址，例如 main()函数的*stACKsetup 一 0； /]ebp*stACKsetup—o； ‖唧 *stACKsetup-一 0； / / edi*stACKsetup 一一=0： / / esl*staclcsetup 一一 0： / / edx*stACKsetup 一～=0； / / ecx*stACKsetup 一=0； /]ebx*stACKsetup—O； / / eax*stACKsetup—0xlO； / / ds ” staclcsetup-一=0xlO； / / es*stACKsetup 一一=0xlO；/ / fs*stACKsetup=0Xl0；‖昏，一 i。processes[f]lpmzess_esp。(uint_t)stacksetup；” . . 。一

爱?j “ processes[r]. 犀∞∞一$。OxlO；

一 processes[f]. proze. ~一 kstack=(uint—t)&kstacks[d][KSTACKTOP]；

。。。processes[f]. prozes~一 ustaek=(uint：一 t)&stacks[d][USTACKTOP]；

上述代码按照 ISR 压栈的顺序来进行放置核心堆栈中的值。在堆栈的最后位置，放置 ESP 的域。

进程执行的入口地址通常是进程的 main()函数的地址。这个地址通常也是这个进程被调度并第一次执行的时候由 eip（执行指令指针）寄存器设置的值。

下面的代码演示了如何进行一个简单的调度。这个调度器仅仅完成在一个执行队列中移动进程的排列顺序，从而使不同的进程都可以获得执行的机会，代码如下所示：

*嚣≈猾

i 囊 0 囊一： / / 当前进程的全局指针

j 黧∞pmzess—IE ” p；

、t^一 l、 ’

0 曩一 j j / / 这是 IsR 的代码

j_i voi 小 timer—handler(void)l

0j0。ll “ if(task—t0 一 bill 一>prozess—timetonln>0)1

0itask。” _to_bill->pmzess 一洒 et0lIIn 一一； ?：

if（task—tO—bill 一>prozBs—timetorun<1）schedule（0）； / / m 度器 voidschedule（intirq）I

ELEME 旧 ” proz；

if（!（queue—empty（&roundrobin—pmzesse））＆＆p 一~"proze，ss—timemrun<1）

P—prozesa—tirnetoron=lO；11 重新填充时间片

/ / 从队列的头部移出进称

pmz。r~illove—first—element—from—queue（&roundmbin—pmzesse，0）；

m 博这个元素放置在队列的后部、

add—element 一∞一 queue—r~ga"（&mundmbin—pmzesse，pmz）；I / 将队列（例如-轮转队列（roundrobinqueue））头部的进程取出来，放置在 P 中。

choose—prozess（irq）；

在上面的代码中，每次发生时钟中断（timer interrupt）时，当前进程 p 的状态都将被保存下来。然后，ISR 减少当前进程的时间片值 1。如果当前进程的时间片值为 0 了，进程就被调度器放置到队列的尾部，然后下一个进程开始执行。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

注意：操作系统通常是由事件驱动的。中断是事件，系统调用（system calls）也是事件。操作系统需要对这些事件进行反映，执行对这些事件的反映函数或者方法来满足触发这个事件的设备的需求。在接收到一个事件（硬件中断、软件中断或者异常）时，操作系统就会执行一个任务切换。

7.3 进程调度的性能分析

在操作系统中，进行进程切换时都将付出相应的代价。本节将讨论与上下文切换有关的开销，并通过一个简单的例子来进行对比分析。

在本节中将看到操作系统切换上下文的速度能有多快。在测试时使用的上下文非常简单，它们几乎不执行什么任务，很难感觉到它们的存在。要测量一种上下文切换算法的速度，可以设计一个测试，按指定的顺序执行一定次数的上下文切换操作。测试的结果可以展示上下文切换的过程，也是测量上下文切换操作速度的一种简单的途径。

在 Unix 中有一种众所周知的简单测试，那就是创建两个进程并在它们之间传送一个令牌，如此往返传送一定的次数。其中一个进程在读取令牌时就会引起阻塞，另一个进程发送令牌后等待其返回时也处于阻塞状态。发送令牌带来的开销与上下文切换带来自开销相比，可以忽略不计。这样，就可以测算出每秒钟进行的上下文切换的总次数了。

最好的令牌就是将一个字节通过管道来回发送。令牌就是那个字节。在进程 A 将令牌写到管道后，它在读取管道时就会阻塞，直到字节被返回为止。起初在读取管道时会发生阻塞的进程 B 在 A 写字节时会被唤醒，并读取该字节。B 马上将这个字节写回管道，并等待再次读取该字节。由于字节是通过管道的不同通道传送的，进程不会读到刚写入的字节。这样，进程 A 在通道 1 上写入，在通道 2 上读取。进程 B 在通道 1 上读取，在通道 2 上写入。这个周而复始的过程不断继续重复，直到一定次数的令牌传送结束。

本书的光盘中有一个程序 psw.cpp 展示了这种令牌传送方式，这个程序可以在 Windows 和 Linux 上编译运行。

psw.cpp 将创建第二个线程（主程序是第一个线程），并在两个线程之间来回传送管道令牌。其中重要的代码如下所示：

```
tstart();
//
// ADULT: Writes the first byte.
//
for(i = 0; i < maxcount; i++) {
    counter++;
    if(!Put(pipeA))
        break;
    if(!Get(pipeB))
        break;
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
tend();


tstart2();
for(i = 0; i < maxcount; i++) {
    if(!Get(pipeA))
        break;
    if(!Put(pipeB))
        break;
}
tend2();
```

读者可以将上述程序在 Linux 和 Windows 中分别运行，从而比较 Linux 和 Windows 的进程切换效率。

7.4 Linux 进程调度源代码分析

Linux 0.01 的核心进程源代码主要包括 sched.c、fork.c、km.c 等。本节将对这些源代码进行分析，以帮助读者掌握 Linux 的进程调度方法。

1. sched.c 分析

sehed.c 是 Linux 的主要核心文件，它实现了 Linux 的进程调度功能，包括实现进程的各种状态：睡眠（sleep_on）、唤醒（wakeup）、调度（schedule）等。此外，sched.c 还实现了一些进程相关的系统调用，例如，getpid()等。带有详细注释的 sched.c 源代码可以在光盘中找到。

2. fork.c 分析

fork.c 包含了实现系统调用 fork()的辅助功能函数。fork()系统调用的具体源代码在文件 system_call.s 中。fork.c 还包含了一些其他的功能，例如 verify_area()等。带有详细注释的 fork.c 源代码可以在光盘中找到。

3. kill.c 分析

kill.c 主要实现了系统调用 kill()，kill()的作用主要是杀死一个进程。通过对 kill.c 的分析，可以掌握 Linux 中撤销一个进程的过程。带有详细注释的 kill.c 源代码可以在光盘中找到。

4. systeme_call.s 分析

system_call.s 包含了系统的低级处理函数，主要都是一些汇编处理程序。在 system_call.s 中也包含了系统时钟中断处理函数和硬盘中断处理函数。带有详细注释的

PDF-XCHANGE Click to buy NOW! www.docu-track.com

system_call.s 源代码可以在光盘中找到。

7.5 实例：实现协作式多任务

本节将介绍利用标准 C 语言 setjmp()库函数实现协作式多任务系统。协作式多任务系统的调度只在用户指定的时机（通常是用户主动放弃 CPU 占用）发生，这样的多任务形式实现比较简单。在本节将给出一个例子程序来演示如果实现协作式多任务系统。通过对协作式多任务的实现进行分析，读者可以更加深入地理解多进程并发执行的原理。

setjmp()是标准的 C 语言库函数，它可以实现程序执行中的远程跳转操作。具体而言，它可以在一个函数中使用 setjmp()来初始化一个全局状态，然后只要该函数未曾执行返回，在程序的其他任何地方都可以通过 longjmp()调用来跳转到 setjmp()的下一条语句继续执行。

setjmp()函数将发生调用处的局部环境保存在一个 jmp_buf 结构当中，只要主调函数中对应的内存未曾释放（函数返回时局部内存就失效了，因此执行 longjmp()的一个前提是调用 setjmp()函数未曾退出），那么在调用 longjmp()时就可以根据已保存的 jmp_buf 参数恢复到 setjmp()的地方执行。本节就是利用了 setjmp()标准库函数的特点，以简单的方式实现了协作式多任务。

为了便于理解，首先给出协作式多任务演示程序的源代码。这个程序演示了协作式多任务切换、任务动态生成、多任务共用代码等功能，程序中使用了 init_coos 初始化根任务（也就是 C 语言 main 函数）、creat_task 创建新任务和 WAITFOR 查询条件这 3 个基本的函数调用。

程序代码通过执行 setjmp()设置本任务下次恢复执行的返回点，然后切换到其他的任务执行，下一次获得执行的时候将在标记的返回点处恢复执行。如此周而复始，让各任务都获得轮转运行的机会。协作式多任务的特点是任何一个任务都不能独占 CPU 而不释放，如果发生了这样的情况，所有其他的任务都将无法执行了。因此，所有的任务都需要主动通过等待条件的方式放弃掉 CPU 系统中除了中断服务程序之外，所有任务都是平等的。在本例子中，没有设计杀死任务的调用，因此各任务都被设计成某种形式的无限循环。

任务中等待的条件可以是任务复杂的表达式或函数调用，也可以是中断服务程序设置的全局变量（注意加 volatile 定义）。一般在任务执行时会让下次等待的条件不再满足，避免某个任务一直霸占 CPU 而将系统其他任务饿死。

由于主要的保护和恢复任务现场的工作都由 C 语言标准库函数 setjmp()实现了，因此，只需要操纵一下堆栈指针防止不同的任务使用重叠的局部堆栈环境即可实现整个任务的切换。协作式多任务的源代码如下：

```
/*************************************************************************
******
        File: cmtask.c
        Cooperative multitasking with setjmp() and longjmp()
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        Compiled by DJGPP
    ************************************************************************
*****/
    #include <setjmp.h>             /* jmp_buf, setjmp(), longjmp() */
    #include <stdio.h>          /* puts() */
    #include <conio.h>          /* kbhit(), getch() */
    #include <pc.h>

    #define   JMPBUF_IP   state[0].__eip
    #define   JMPBUF_SP   state[0].__esp

    #define   NUM_TASKS 2
    #define   STACK_SIZE 512


    typedef struct
    {
        jmp_buf state;
    } task_t;

    static task_t g_tasks[NUM_TASKS + 1];
    /************************************************************************
******
    g_tasks[0]                  state of first task
        ...
    g_tasks[NUM_TASKS - 1]  state of last task
    g_tasks[NUM_TASKS]      jmp_buf state to return to main()
    ************************************************************************
*****/
    static void schedule(void)
    {
        static unsigned current = NUM_TASKS;
        unsigned prev;

        prev = current;
    /* round-robin switch to next task */
        current++;
        if(current >= NUM_TASKS)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                current = 0;
        /* return to main() if key pressed */
                if(kbhit())
                        current = NUM_TASKS;

        /* save old task state
        setjmp() returning nonzero means we came here through hyperspace
        from the longjmp() below -- just return */
                if(setjmp(g_tasks[prev].state) != 0)
                        return;
                else
                        /* load new task state */
                        longjmp(g_tasks[current].state, 1);
        }
        /************************************************************************
******
        ************************************************************************
*****/
        #define    WAIT      0xFFFFFL

        static void wait(void)
        {
                unsigned long wait;

                for(wait = WAIT; wait != 0; wait--)
                        /* nothing */;
        }
        /************************************************************************
******
        ************************************************************************
*****/
        static void task0(void)
        {
                char *p;

                puts("hello from task 0 ");
                while(1)
                {
```

```
            schedule(); /* yield() */
            p = (char *)malloc(100);
            puts("this is task 0... ");
            free(p);

            {
                FILE *fp;
                fp = fopen("cmtask.txt", "a+");
                fprintf(fp,"cooperative multitask demo!\n");
                fclose(fp);
            }
            wait();
        }
    }
    /*****************************************************************************
******
    *****************************************************************************
*****/
    static void task1(void)
    {
        puts("\tthis is task 1");
        while(1)
        {
            schedule(); /* yield() */

            puts("\t\t\ttask 1 is runing!");
            wait();
        }
    }


    /*****************************************************************************
******
    *****************************************************************************
*****/
    int main(void)
    {
        static char stacks[NUM_TASKS][STACK_SIZE];
```

```
    volatile unsigned i;
/**/
    unsigned adr;

    for(i = 0; i < NUM_TASKS; i++)
    {
/* set all registers of task state */
        (void)setjmp(g_tasks[i].state);
        adr = (unsigned)(stacks[i] + STACK_SIZE);
/* set SP of task state */
        g_tasks[i].JMPBUF_SP = adr;
    }

/* set IP of task state */
    g_tasks[0].JMPBUF_IP = (unsigned)task0;
    g_tasks[1].JMPBUF_IP = (unsigned)task1;

/* this does not return until a key is pressed */
    schedule();

/* eat keystroke */
    if(getch() == 0)
        (void)getch();
    return 0;
}
```

上述代码使用如下的命令编译：

```
gcc cmtask.c -o cmtask.exe
```

执行 cmtask.exe，运行效果如图 7-4 所示。从图 7-4 中可以看到，两个任务 task0 和 task1 交替执行，实现了多任务的并发执行。

图 7-4　协作多任务运行的例子

7.6　本章小结

进程是操作系统的重要概念，本章首先介绍了进程的基本功能，对进程的状态、调

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

度的基本概念进行了简单的介绍。然后，本章详细介绍了操作系统中进程的调度实现。

线程是现代操作系统的一种重要特性。本章也介绍了线程在操作系统中的实现方式，包括用户级线程和核心级线程两种实现方式。

操作系统进程的调度是一个非常重要的性能因素。本章通过一个具体的实例，详细分析了进程调度的性能问题。

本章还详细分析了 Linux 0.01 中进程调度的源代码。最后，本章给出了一个通过 C 函数实现协作式多任务的例子，读者可以通过它体会到多任务运行的效果。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

第8章 设备管理和调度

本章知识点：

- Linux 设备管理概述
- 中断处理程序：ISR
- 设备驱动程序实例
- Linux 0.01 设备管理代码分析

设备管理是操作系统中非常重要的一部分。一个良好设计的操作系统，应该向用户提供一个一致的界面来访问系统中的所有设备。Linux 设备管理思想继承了 Uni|x 设备管理的基本思想，将所有的外部设备抽象为设备文件，使用统一的文件访问接口来访问各种设备，大大简化了应用程序的设计。本章首先介绍 Linux 设备管理的基本原理，并重点介绍设备管理的基本要求、驱动程序接口、设备管理接口、同步和异步访问等重要问题。本章同时也介绍一些典型设备的控制方法，例如，屏幕控制方法、键盘控制方法、IDE 设备控制方法等。这些典型设备是一个系统中最基本的设备，读者掌握这些设备的控制方法后，可以迅速地实现自己的设备管理部分。

8.1 Linux 设备管理概述

8.1.1 设备管理的基本要求

在操作系统中，设备管理系统通常是设备驱动程序和操作系统的其余部分和应用程序的惟一接口。设备管理系统需要实现如下的功能：

- 隔离设备驱动程序和操作系统核心。这样设备驱动程序的编写者可以专注地编写硬件设备的物理接口方面的代码，使用统一的接口和核心进行交互。
- 隔离硬件和用户程序。使用户程序和具体的硬件无关。这样用户程序可以工作在任意的硬件设备上，而应用程序的编写者也无需去考虑具体工作的硬件设备类型。

在大多数操作系统中，设备管理程序是核心和应用程序惟一可以看到的和硬件设备相关的接口。如果操作系统提供了一个设计良好的设备管理程序接口，就可以使核心的大多数部分和具体物理设备无关，从而使操作系统易于移植到不同的硬件平台上。

一般操作系统的设备管理程序具有如下的特性：

- 异步 I/O（Asynchronous I/O）：异步 I/O 是指应用程序可以在启动一个 I/O 操作后，不需要等待这个 I/O 操作完成，而继续执行后面的指令。与异步 I/O 相对应的是同步 I/O。在同步 I/O 情况下，应用程序必须等待启动的 I/O 操作完成以后才能继续执行后续的指令。相对于异步 I/O 而言，同步 I/O 的实现是比较简单的。同步 I/O 可以在实现异步 I/O 的基础上简单地实现。
- 即插即用（Plug and Play）：即插即用的概念就是当物理设备插入计算机系统中时，驱动程序可以自动加载；而当设备被从系统中移除时，设备驱动程序也可以从操作系统中自动地卸载。在操作系统启动时，系统中具有的设备可以被操作系统自动地检测，并自动地加载相应的驱动程序。

8.1.2 驱动程序（Drivers）

因为本章介绍的设备驱动程序要具有即插即用功能，因此驱动程序不能仅仅在系统编译时被加入到操作系统核心中（Minix 和早期的 Linux 就是这样做的），而应该在操作系统运行时可以被自由地加载和卸载。在运行时刻加载和卸载驱动程序并不是特别困难，关键是在实现的时候需要扩展可执行文件接口到操作系统核心中。

对于核心驱动程序模块，有多种实现方式。Linux 使用对象文件（object files（.o））来实现。对象文件在系统装载时被链接到核心中，当驱动程序被链接到核心中时，驱动程序被重新定位和链接。在 Windows NT 中，使用完整的 PE 动态链接库（文件扩展名为.sys）。驱动程序被装载到核心级进程中的机制和动态链接库被装载到用户级进程中的机制是一致的。核心需要设备驱动程序提供一些程序入口点，当驱动程序被核心装载后，核心就可以访问驱动程序的所有代码和数据（反之，驱动程序也能够访问核心的所有数据和代码）。在实现了驱动程序和核心的接口后，就需要一种机制，可以让核心在合适的时候调用驱动程序代码来实现对硬件的访问。

如今的计算机系统中使用的硬件设备都具有一定的智能性，对硬件设备的检测和访问提供了较好的支持：

- PCI 总线使硬件的检测变得非常简单：每一种设备都有一个惟一不同的 16 位生产商 D 和设备 ID。
- 一个 ISA PnP 芯片，再加上支持 PnP 的 BIOs，容许一系列的 ISA PnP 设备连接到总线上。
- ACPI 芯片可以提供对主板设备的一致访问方式。
- USB hub 容许不同类型和容量的设备连接到上面。

因此，基于上述的硬件设备，可以编写一个通用的总线驱动程序来探测连接到这个总线上的所有设备。再进一步，可以在操作系统中编写一个总线探测程序来探测系统中安装的总线（或者在系统中默认地运行所有的总线驱动程序，只是这些总线驱动程序依据系统中实际安装的总线，有些在运行，有些不运行）。对于一些使用跳线（jumper）来设置的旧的物理设备，可以把配置存放在虚拟总线设备的文件中，而不需要从硬件跳线

中读取。

8.1.3 接口（Interfaces）

一旦探测到系统中安装的设备以后，在核心中就需要对这些设备进行记录。在 Unix 中，所有的设备又都被抽象为字符设备（打印机、终端设备）和块设备（磁盘驱动器），这些设备又都被系统抽象为设备文件，存放在系统的/dev 目录下。这种方法在大多数类 Unix 操作系统中都被使用，例如 Minix 和 Linux。在/dev 目录下存放着许多的虚拟设备文件，这些虚拟设备文件都不指向实际的磁盘上存放的文件，而是代表了核心中的一个数据结构，例如核心中的进程和内存（/dev/mem 代表了系统的内存）。

/dev 目录下的设备文件都具有主设备号和次设备号。主设备号代表了这种设备的类型，或者使用的驱动程序，次设备号表明这种设备类型的不同设备个体。例如，在系统中所有的 IDE 设备都可以具有一样的主设备号，每一个不同的次设备号可以指向不同的硬盘驱动器、CD-ROM 驱动器。在/dev 目录下的文件都是存放在磁盘上的，因此这些文件在系统重新启动时不会被丢失。这些设备文件的主设备号每次都是不变的，但是次设备号在每次启动时可能会有变化。在 Linux 中，把所有的设备都抽象成为文件是一种被证明是非常成功的策略。通过统一的文件接口，可以把系统中所有的设备进行简单而有效的管理。例如，在 Unix 系统中，可以把 microphone 抽象为文件/dev/dsp，那么如果希望把 microphone 录制出来的声音数据传送到一个过滤器中进行处理，然后将处理后的声音回送到麦克风中。通过 Unix 的文件系统接口，这个过程可以不编写一句程序，而通过在命令行使用命令就可以实现，例如：

```
cat /dev/dsp | wavefilter > /dev/phone
```

在 BeOS 中，也使用了类似的方法来组织物理设备。BeOS 中也有/dev 文件系统，尽管这些文件是动态存储成为树型结构。在早期的 Unix 版本中，如果要改变/dev 目录中的设备文件，需要重新改写核心。后来，Linux 支持了动态设备驱动程序，可以在系统运行时随时加载和卸载设备驱动程序，从而改变/dev 目录下的文件。

在 Windows 系统中不使用类似 Unix 的方法，Windows 不把系统资源暴露给用户，用户看到的是不同的驱动器，例如磁盘 C、磁盘 D 等。在 Windows 内部，Windows 也使用类似 Unix 的方式来管理设备。使用这种机制，核心通过文件系统接口，就和具体的设备隔离开了。核心可以使用与开发普通文件一致的方法来访问设备文件。在 Linux 和 Windows 系统中，都可以实现一种很好的抽象机制：只要用户知道设备的名字，就可以在应用程序中使用文件系统接口来访问它。例如，在 Linux 下可以使用 read()、write()来读和写设备，而在 Win32 环境中使用 ReadFile()和 WriteFile()来读和写设备；在 Unix 下可以使用 ioctl()来对设备进行设置，而在 Win32 环境中使用 DeviceIoControl()来对设备进行设置。

从上面的描述中，可以看到用户程序和设备驱动程序之间的接口，实际上就是文件系统接口。典型的，一个应用程序可以通过 ioctl()接口来对设备的属性进行控制。read()、

PDF-XCHANGE Click to buy NOW! www.docu-track.com

write()和 ioctl()都是系统调用，通过这个系统调用可以进入系统核心。系统核心通过read()、write()和 ioctl()调用中的文件句柄来查找这个文件句柄所代表的实际设备（核心中有一张表，代表着打开的文件句柄和设备之间的对应关系）。然后，核心依据得到的设备号，把这些命令传递给设备驱动程序。

在核心和设备驱动程序之间的接口，通常是一张功能函数表格。在 Linux 和 Windows 中，都使用一张类似的表格来记录每个设备的 I/O 操作函数（open()、close()、read()、write()、ioctl()等）的入口地址。在 Linux 上，这个表格中的每个指针指向设备驱动程序中对应实现的设备驱动函数功能。在 Windows 中，也是类似的。在某些情况下，同样一个函数可以被不同的指针指向，这些功能可以被驱动程序共享。

8.1.4 异步 I/O（Asynchronous I/O）

在具体的 I/O 实现方式上，Linux 和 Windows 具有很大的不同。在 Linux 中，如果应用程序对一个设备文件调用 read()系统调用，这个系统调用最终会调用到设备驱动程序中的read()函数。驱动程序中的read()函数在一种情况下可以立刻返回结果给调用者（例如读内存虚拟磁盘。RAMDISK），在另外一种情况下也可以在开始处理这个 read 请求的同时，把调用进程挂起（切换到 SLEEP 状态），就像普通的 IDE 驱动程序一样。

如果等待 I/O 操作执行结束时，进程需要被挂起，那么是同步 I/O（synchronous I/O，也叫 blocking I/O）。如果进程发送了 I/O 命令后，无需等待 I/O 执行完成就可继续运行，这就是异步 I/O（Asynchronous I/O，也叫 non-blocking I/O）。

现在，来分析在 Linux，系统中的两种典型的情况：读 RAMDISK 和读 ID]E 设备。对于 read 请求，如果 read 一个 RAMDISK，典型的过程如下：

- 从 RAMDISK 内存空间复制内存到用户空间（user space）。
- 结束调用，返回用户空间。

一个 IDE 驱动程序，典型的 read 处理过程如下：

- 接受一个 read 请求，然后把这个请求放到 I/O 队列中。
- 如果 I/O 队列为空，将这个读写磁盘扇区的请求发送到磁盘驱动器。
- 将调用 read 系统调用的进程切换为 SLEEP 状态。
- 当磁盘驱动器完成磁盘扇区读操作后，会发出一个中断。IDE 驱动程序接收到这个中断后，把读出的数据从磁盘控制器传送到核心的 buffer 中，然后把数据复制到用户空间，接下来唤醒调用进程。
- 当调用进程被唤醒时，系统的控制权回到用户空间的用户进程。

在 Linux 系统中，read()系统调用使用上面介绍的 IDE 设备的方式。从上面的过程分析可以看到，一旦用户程序的 read()调用执行，read()调用就会陷入内核，一直执行下去，直到读操作完成为止。在这个过程中，调用 read()的进程将一直处于挂起等待状态，直到需要读的数据和 read()调用返回应用程序为止。

在 Windows NT 中实现的方式有些不同。Windows NT 在核心中使用异步 I/O 方式。

尽管在大多数应用程序中，都会使用同步方式的 ReadFile()和 WriteFile()，但是 ReadFile()和 WriteFile()会在函数一开始执行时就返回，其实是异步 I/O。在 Windows NT 中的 ReadFileEx()和 WriteFileEx()也是异步 I/O。

尽管 Windows NT 的异步 I/O 模型比 Linux 模型复杂的多，但是 Windows NT 模型更加适合 Windows NT 的多线程操作系统环境。在 Windows NT 环境中，一个 IDE 的 read 请求会执行如表 8-1 所示的操作。

表 8-1　IDE 设备的读请求

系统组成部分	进行的操作
App	调用 Call overlapped ReadFileEx()
Driver	从核心中接收 read 请求，然后将这个请求放置到一个队列中
	锁住内存中的用户 buffer，或者这个 buffer 的物理地址
	如果队列为空，将这个 Read Sectors 命令发送到磁盘控制器
	返回用户模式
App	等待 I/O 结束；或者切换到睡眠状态，或者进行其他工作
Ddver	当接收到 IDE 中断，把控制器缓冲区中的数据传送到前面获得的缓冲区物理地址处
	通知 I/O 管理程序 IRP 结束了
App	结束 I/O 等待

如果应用程序希望提供同步 I/O（synchronous I/O）调用，Windows 系统还是会进行上述相似的过程，不过不会在数据没有返回用户程序时将控制权返回给用户程序，而是要把用户程序切换到睡眠状态进行等待，直到要读的数据最终得到才结束等待。

注意：驱动程序在处理用户的 I/O 请求之前锁住用户程序提供的缓冲区。这种操作在多任务环境中，必须非常小心地处理可能带来的死锁问题，因为把数据写入这些缓冲区的中断可能在任何进程的正文环境中执行。这可能会导致核心启动了一个操作，将数据从硬件设备中读取出来然后写入到一个进程的数据区中，但是当这个操作还没有结束时，核心进程调度却停止了这个进程的运行，而调度另外一个进程执行。在进程切换时，可能会导致换页操作，如果进程的缓冲区使用的页面被置换到外存，那么可能会导致非常严重的后果，即需要写入的目标页面已经不在内存中了。为了避免这种问题的发生，核心在 I/O 请求被执行之前，必须在获得这些缓冲区的物理地址的同时，还需要把这些页面锁定在内存申，防止这些页面在换页过程中被置换到外存中。其中每一个虚拟页面的物理地址都必须要获得，因为这些页面在内存中不一定是连续的。

获得了这些页面的物理地址，可以有效地支持使用 DMA（Direct Memory Access，直接存储器访问）的设备，例如软盘控制器、硬盘控制器和声卡。这些 DMA 设备可以无需 CPU 的控制而直接传输数据到用户缓冲区中，或者把用户缓冲区的数据复制到设备内部的缓冲区中。大多数的 DMA 设备驱动程序都具有自己驱动程序内部的缓冲区，在执行 DMA 操作时，通常是把用户缓冲区中的数据首先传送到驱动程序内部的静态缓冲

区中，然后再在必要时传送到设备缓冲区中。不同的 DMA 设备在使用 DMA 方式进行数据传送时，有一些对缓冲区的限制，这些缓冲区要求：

- 在内存中必须连续，除非是支持散列缓冲区的 DMA 设备（scatter-gather DMA device，这种设备可以编程使用一系列的散列地址空间）。
- 必须完全位于低端的 16MB 内存空间。
- 不能跨越 64KB 边界。

在 PCI 方式 DMA 设备中必须满足第一项，ISA 要求三项都满足。因此，如果在设备，驱动程序中要使用 DMA 方式，必须在设备驱动程序中实现一个物理存储管理器来满足这些条件。

8.2 Linux 0.01 中断处理

中断是 Linux 系统中的一个重要并且复杂的组成部分，它使系统的外部设备可以在需要的时候获得 CPU。

8.2.1 中断处理的基本过程

中断可以分为 CPU 的内部中断和外部设备的中断即外部中断两种。CPU 的内部中断又可以叫做异常，异常的主要作用是报告一些程序运行过程中的错误和处理缺页中断（page_fault）。这些异常都是通过 set_trap_gate 宏来设置的。

set_trap_gate 宏是定义在 include/asm/system.h 文件中的一个宏，被用来设置 CPU 的 trap，其定义如下：

#defines set_trap_gate(n, addr) _set_gate(&idt[n], 15, 0, addr)

Intel 的 CPU 为 trap 保留了 32 个，具体在 idt 中的号是 00～20。外部设备中断通过 set_intr_gate 宏来设置，编号从 21 开始到 ff。

此外，还有一种软件中断，也就是 Linux 中的系统调用，它可以由应用程序来调用。在 Linux 0.01 中，中断处理主要做了两个部分的工作：

(1) trap.s 将各个 CPU 的 trap 信息填入到 idt 中；

(2) asm.s 处理当出现 trap 的时候后续的一系列处理。

在 Linux 0，01 中处理的外部中断有时钟中断（0x20），串行通信口中断（0x23 和 0x24），硬盘中断（0x2e），键盘中断（0x21），这些中断具体的处理函数在文件 rs_io.s、hd.c 和 keyboard.s 中。

IBM PC 一般使用两片 8259 作为接收外部的中断控制器，其初始化在 boot.s 中完成，具体的连接是 8259 的从片接主片的第二个中断口，其他的中断线就可以通过其他设备共享。外部设备产生一个中断之后，8259 自己将 irq 号转换为中断向量（一般是加上 20），然后发给 CPU 并进行等待，当 CPU 响应之后再清 intr 线。CPU 接收到中断之后在内核堆栈中保留 irq 值和寄存器值，同时发送一个 pic 应答并执行 isr 中断服务程序。

如果多个设备共享一个中断，那么每当一个设备产生一个中断时，CPU 会执行所有的 isr 中断服务程序。具体的一个完整的中断产生和处理流程是：

(1) 产生中断；
(2) CPU 应答；
(3) 查找 idt 中的对应向量；
(4) 在 gdt 中查找 idt 项的代码段；
(5) 对比当前的 cpl 和描述符的 dpl 看是否产生越级保护；
(6) 检查是否发生特权级的变化，如果是就保存 ss 和 esp，否则不保存；
(7) 保存 eflags、cs、eip 和错误码；
(8) 将 idt 对应描述符地址装入 cs 和 eip 中以便执行；
(9) 执行 irq_interrupt；
(10) 执行 do_irq；
(11) 循环执行 isr；
(12) 中断返回。

8.2.2 traps.c 文件分析

traps.c 处理了硬件中断和运行中可能产生的错误，具体的源代码如下：

```
/*
 * 'Traps.c' handles hardware traps and faults after we have saved some
 * state in 'asm.s'. Currently mostly a debugging-aid, will be extended
 * to mainly kill the offending process (probably by giving it a signal,
 * but possibly by killing it outright if necessary).
 */
#include <string.h>

#include <linux/head.h>
#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/system.h>
#include <asm/segment.h>

#define get_seg_byte(seg,addr) ({ \
register char __res; \
//下面内嵌汇编的意思是：
//      push %fs
//      movl        seg   %eax
//      mov %ax %fs
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
//      movb      %fs:*addr       %al
//      movl      %eax      _res
//      pop  %fs
__asm__("push %%fs;mov %%ax,%%fs;movb %%fs:%2,%%al;pop %%fs" \
        :"=a" (__res):"0" (seg),"m" (*(addr))); \
__res;})

#define get_seg_long(seg,addr) ({ \
register unsigned long __res; \
__asm__("push %%fs;mov %%ax,%%fs;movl %%fs:%2,%%eax;pop %%fs" \
        :"=a" (__res):"0" (seg),"m" (*(addr))); \
__res;})

#define _fs() ({ \
register unsigned short __res; \
//下面内嵌汇编的意思是：
//      mov %fs  %ax
//      mov %eax       _res
__asm__("mov %%fs,%%ax":"=a" (__res):); \
__res;})

int do_exit(long code);

void page_exception(void);

void divide_error(void);
void debug(void);
void nmi(void);
void int3(void);
void overflow(void);
void bounds(void);
void invalid_op(void);
void device_not_available(void);
void double_fault(void);
void coprocessor_segment_overrun(void);
void invalid_TSS(void);
void segment_not_present(void);
void stack_segment(void);
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
void general_protection(void);
void page_fault(void);
void coprocessor_error(void);
void reserved(void);

static void die(char * str,long esp_ptr,long nr)
{
	long * esp = (long *) esp_ptr;
	int i;

	printk("%s: %04x\n\r",str,nr&0xffff);
	printk("EIP:\t%04x:%p\nEFLAGS:\t%p\nESP:\t%04x:%p\n",
		esp[1],esp[0],esp[2],esp[4],esp[3]);
	printk("fs: %04x\n",_fs());
	printk("base: %p, limit: %p\n",get_base(current->ldt[1]),get_limit(0x17));
	if (esp[4] == 0x17) {
		printk("Stack: ");
		for (i=0;i<4;i++)
			printk("%p ",get_seg_long(0x17,i+(long *)esp[3]));
		printk("\n");
	}
	str(i);
	printk("Pid: %d, process nr: %d\n\r",current->pid,0xffff & i);
	for(i=0;i<10;i++)
		printk("%02x ",0xff & get_seg_byte(esp[1],(i+(char *)esp[0])));
	printk("\n\r");
	do_exit(11);		/* play segment exception */
}

void do_double_fault(long esp, long error_code)
{
	die("double fault",esp,error_code);
}

void do_general_protection(long esp, long error_code)
{
	die("general protection",esp,error_code);
}
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
void do_divide_error(long esp, long error_code)
{
	die("divide error",esp,error_code);
}

void do_int3(long * esp, long error_code,
		long fs,long es,long ds,
		long ebp,long esi,long edi,
		long edx,long ecx,long ebx,long eax)
{
	int tr;

//下面内嵌汇编的意思是：
//	movl	0	%eax
//	str	%eax		//复制 TR 到 16 位寄存器或存储器单元
//	movl	%eax	tr
	__asm__("str %%ax":"=a" (tr):"0" (0));
	printk("eax\t\tebx\t\tecx\t\tedx\n\r%8x\t%8x\t%8x\t%8x\n\r",
		eax,ebx,ecx,edx);
	printk("esi\t\tedi\t\tebp\t\tesp\n\r%8x\t%8x\t%8x\t%8x\n\r",
		esi,edi,ebp,(long) esp);
	printk("\n\rds\tes\tfs\ttr\n\r%4x\t%4x\t%4x\t%4x\n\r",
		ds,es,fs,tr);
	printk("EIP: %8x   CS: %4x  EFLAGS: %8x\n\r",esp[0],esp[1],esp[2]);
}

void do_nmi(long esp, long error_code)
{
	die("nmi",esp,error_code);
}

void do_debug(long esp, long error_code)
{
	die("debug",esp,error_code);
}

void do_overflow(long esp, long error_code)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
{
        die("overflow",esp,error_code);
}

void do_bounds(long esp, long error_code)
{
        die("bounds",esp,error_code);
}

void do_invalid_op(long esp, long error_code)
{
        die("invalid operand",esp,error_code);
}

void do_device_not_available(long esp, long error_code)
{
        die("device not available",esp,error_code);
}

void do_coprocessor_segment_overrun(long esp, long error_code)
{
        die("coprocessor segment overrun",esp,error_code);
}

void do_invalid_TSS(long esp,long error_code)
{
        die("invalid TSS",esp,error_code);
}

void do_segment_not_present(long esp,long error_code)
{
        die("segment not present",esp,error_code);
}

void do_stack_segment(long esp,long error_code)
{
        die("stack segment",esp,error_code);
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
void do_coprocessor_error(long esp, long error_code)
{
	die("coprocessor error",esp,error_code);
}

void do_reserved(long esp, long error_code)
{
	die("reserved (15,17-31) error",esp,error_code);
}

void trap_init(void)
{
	int i;

	set_trap_gate(0,&divide_error);		//idt[0]——idt[16]给了错误处理（不包括 idt[15]）
	set_trap_gate(1,&debug);
	set_trap_gate(2,&nmi);
	set_system_gate(3,&int3);	/* int3-5 can be called from all */
	set_system_gate(4,&overflow);
	set_system_gate(5,&bounds);
	set_trap_gate(6,&invalid_op);
	set_trap_gate(7,&device_not_available);
	set_trap_gate(8,&double_fault);
	set_trap_gate(9,&coprocessor_segment_overrun);
	set_trap_gate(10,&invalid_TSS);
	set_trap_gate(11,&segment_not_present);
	set_trap_gate(12,&stack_segment);
	set_trap_gate(13,&general_protection);
	set_trap_gate(14,&page_fault);
	set_trap_gate(15,&reserved);
	set_trap_gate(16,&coprocessor_error);
	for (i=17;i<32;i++)			//idt[17]——idt[31]的陷阱门保留
		set_trap_gate(i,&reserved);
/*	__asm__("movl $0x3ff000,%%eax\n\t"
		"movl %%eax,%%db0\n\t"
		"movl $0x000d0303,%%eax\n\t"
```

```
            "movl %%eax,%%db7"
            :::"ax");*/
}
```

8.2.3 /kernel/asm.s 文件分析

asm.s 包含了处理低级硬件错误的代码，asm.s 也使用 TS 位来处理协处理器错误，具体的代码如下：

```
/*
 * asm.s contains the low-level code for most hardware faults.
 * page_exception is handled by the mm, so that isn't here. This
 * file also handles (hopefully) fpu-exceptions due to TS-bit, as
 * the fpu must be properly saved/resored. This hasn't been tested.
 */

.globl _divide_error,_debug,_nmi,_int3,_overflow,_bounds,_invalid_op
.globl _device_not_available,_double_fault,_coprocessor_segment_overrun
.globl _invalid_TSS,_segment_not_present,_stack_segment
.globl _general_protection,_coprocessor_error,_reserved

_divide_error:
        pushl $_do_divide_error
no_error_code:
        xchgl %eax,(%esp)
        pushl %ebx
        pushl %ecx
        pushl %edx
        pushl %edi
        pushl %esi
        pushl %ebp
        push %ds
        push %es
        push %fs
        pushl $0        # "error code"
        lea 44(%esp),%edx
        pushl %edx
        movl $0x10,%edx
        mov %dx,%ds
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
        mov %dx,%es
        mov %dx,%fs
        call *%eax
        addl $8,%esp
        pop %fs
        pop %es
        pop %ds
        popl %ebp
        popl %esi
        popl %edi
        popl %edx
        popl %ecx
        popl %ebx
        popl %eax
        iret

_debug:
        pushl $_do_int3         # _do_debug
        jmp no_error_code

_nmi:
        pushl $_do_nmi
        jmp no_error_code

_int3:
        pushl $_do_int3
        jmp no_error_code

_overflow:
        pushl $_do_overflow
        jmp no_error_code

_bounds:
        pushl $_do_bounds
        jmp no_error_code

_invalid_op:
        pushl $_do_invalid_op
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        jmp no_error_code

math_emulate:
        popl %eax
        pushl $_do_device_not_available
        jmp no_error_code
_device_not_available:
        pushl %eax
        movl %cr0,%eax
        bt $2,%eax              # EM (math emulation bit)
        jc math_emulate
        clts                    # clear TS so that we can use math
        movl _current,%eax
        cmpl _last_task_used_math,%eax
        je 1f                   # shouldn't happen really ...
        pushl %ecx
        pushl %edx
        push %ds
        movl $0x10,%eax
        mov %ax,%ds
        call _math_state_restore
        pop %ds
        popl %edx
        popl %ecx
1:      popl %eax
        iret

_coprocessor_segment_overrun:
        pushl $_do_coprocessor_segment_overrun
        jmp no_error_code

_reserved:
        pushl $_do_reserved
        jmp no_error_code

_coprocessor_error:
        pushl $_do_coprocessor_error
        jmp no_error_code
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
_double_fault:
        pushl $_do_double_fault
error_code:
        xchgl %eax,4(%esp)              # error code <-> %eax
        xchgl %ebx,(%esp)       # &function <-> %ebx
        pushl %ecx
        pushl %edx
        pushl %edi
        pushl %esi
        pushl %ebp
        push %ds
        push %es
        push %fs
        pushl %eax                      # error code
        lea 44(%esp),%eax       # offset
        pushl %eax
        movl $0x10,%eax
        mov %ax,%ds
        mov %ax,%es
        mov %ax,%fs
        call *%ebx
        addl $8,%esp
        pop %fs
        pop %es
        pop %ds
        popl %ebp
        popl %esi
        popl %edi
        popl %edx
        popl %ecx
        popl %ebx
        popl %eax
        iret

_invalid_TSS:
        pushl $_do_invalid_TSS
        jmp error_code
```

```
_segment_not_present:
    pushl $_do_segment_not_present
    jmp error_code

_stack_segment:
    pushl $_do_stack_segment
    jmp error_code

_general_protection:
    pushl $_do_general_protection
    jmp error_code
```

8.3 如何编写中断服务程序 ISR

本节将重点介绍什么是中断处理程序（Interrupt Service Routine，ISR），中断处理程序和普通函数有什么区别，以及如何使用 GCC 来编写 ISR。

1. ISR：中断服务程序

x86 体系结构是一个中断驱动的系统，外部发生的事件总是通过中断服务程序来进行处理。在 x86 体系结构中，中断处理程序的人口地址是保留在系统的 IDT（Interrupt Descriptor Table，中断描述表）中的。

中断事件可以是硬件设备产生的中断，也可能是软件产生的中断。

一个硬件中断的例子是键盘产生的键盘中断。每一次按下键盘上的按键，都会产生一个键盘中断（这个中断是 IRQ1）。

一个软件中断的例子是 MS-DOS 系统提供的系统软中断调用，例如 INT 21h 提供大量 MS-DOS 系统的功能。

2. 中断处理程序和普通程序的区别

中断处理程序和普通程序的区别是：中断处理程序必须非常简单，而且需要处理 CPU 的状态。

ISR 在结束时，都必须调用“ Interrupt Return（IRET）”，而普通应用程序在结束时调用的是“ Return（RET）”或者“ Far Return（RETF）”。

一些编译器不能直接创建 ISR，有些编译器可以直接支持生成中断处理程序。在支持中断处理程序的编译器中，通常具有一个非 ANSI 标准的关键字“ _interrupt”或者“ interrupt”。Watcom C/C 十+、Borland C/C++、Microsoft C 6.0 都支持“ _interrupt”或者“ interrupt”关键字，而 GCC 和 Visual C/C++都不支持“ _interrupt”或者“ interrupt”。下面是在 Watcom C/C++中编写的一个简单的 ISR 的例子：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/* example of clock tick routine in Watcom C/C++ */
void _interrupt ISR_clock(void)
{
    clock_count++;
}
```

3. 使用 GCC 实现 ISR

虽然 GCC 并不直接支持生成 ISR 的“ _interrupt”关键字。但是通过一些变通的方法，也可以在 GCC 中生成 ISR，当然这个过程比直接支持“ _interrupt”关键字的编译器要繁琐一些。

为了在 GCC 中实现 ISR，不得不书写一个很小的汇编代码程序来调用使用 C 编写的中断处理程序。下面是一个使用 GCC 编写的 ISR 的例子，从这个例子中可以看到，汇编语言的代码主要是用来保存环境和恢复环境。

文件 isr_clock.s，使用 AT&T 的汇编语言书写，在中断中调用了使用 C 书写的中断处理函数，具体的代码如下：

```
/* isr_clock.s */
    .globl    _clock_isr
    .align    4

_clock_isr:

    /* save some registers */
    pushl     %eax
    pushl     %ds
    pushl     %es
    pushl     %fs
    pushl     %gs

    /* set our descriptors up to a DATA selector */
    movl      $0x10, %eax
    movw      %ax, %ds
    movw      %ax, %es
    movw      %ax, %fs
    movw      %ax, %gs

    /* call our clock routine */
    call  _ISR_clock
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/* clear the PIC (clock is a PIC interrupt) */
movl    $0x20, %eax
outb    %al, $0x20

/* restor our regs */
popl %gs
popl %fs
popl %es
popl %ds
popl %eax
iret
```

文件 clock.c，使用 C 语言书写的中断处理函数，在中断处理中，仅仅完成一个计数变量的加 1 操作，具体代码如下：

```
/* clock.c */
/* tell our linker clock_isr is not in this file */
extern void clock_isr(void);

__volatile__ unsigned long clock_count=0L;

void ISR_clock(void)
{
    clock_count++;
}
```

上述两个程序分别编译后，链接到一起，就可以实现一个简单的 ISR 程序。

8.4 设备驱动程序实例

本节将介绍计算机系统中一些主要硬件设备的驱动方法。通过本节的学习，读者可以快速地掌握计算机系统中主要硬件设备驱动程序的编写方法，提高对设备管理系统实现的认识。

注意：关于如何通过直接访问内存来在屏幕上显示字符、如何判断显示器类型、如何移动光标，已经在本书的 3.3.2 小节中进行了介绍，读者可以参考。本章对于显示驱动就不再讲述了。

8.4.1 利用 BIOS 的探测系统设备

BIOS 数据区提供了大量的硬件设备信息，通过读取 BIOS 的数据，可以方便地了解计算机的硬件设备情况。下面的代码就通过读取 BIOS 数据区和使用 BIOS 提供的功能来探测计算机的配置情况，代码如下：

```
/* biostest.c */
#include <stdio.h>
#include <bios.h>

static unsigned char diskbuf[2048];

void main(void)
{
    struct diskinfo_t di;
    unsigned i, j, status, port;
    unsigned char *p, linebuf[17];
    union
    {                                   /* Access equiment either as:      */
        unsigned u;                     /*     unsigned or                 */
        struct                          /*     bit fields                  */
        {
unsigned diskflag : 1;          /* Diskette drive installed?       */
unsigned coprocessor : 1;       /* Coprocessor? (except on PC)     */
unsigned sysram : 2;            /* RAM on system board             */
unsigned video : 2;             /* Startup video mode              */
unsigned disks : 2;             /* Drives 00=1, 01=2, 10=3, 11=4 */
unsigned dma : 1;               /* 0=Yes, 1=No (1 for PC Jr.)      */
unsigned comports : 3;          /* Serial ports                    */
unsigned game : 1;              /* Game adapter installed?         */
unsigned modem : 1;             /* Internal modem?                 */
unsigned printers : 2;          /* Number of printers              */
        } bits;
    } equip;


    /* _bios_equiplist() test */
    equip.u = _bios_equiplist();
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        printf( "Disk drive:                %s\n", equip.bits.diskflag ? "Yes" : "No" );
        printf( "Coprocessor:               %s\n", equip.bits.coprocessor ? "Yes" : "No" );
        printf( "Game adapter:              %s\n", equip.bits.game ? "Yes" : "No" );
        printf( "Serial ports:              %d\n", equip.bits.comports);
        printf( "Number of printers:    %d\n\n", equip.bits.printers);


        /* _bios_memsize() test */
        printf( "Size of memory:            %i K\n\n", _bios_memsize());


        /* _bios_printer() test */
        puts("Test of paralel ports:");
        for(port=0; port < equip.bits.printers; port++)
        {
            status = _bios_printer(_PRINTER_STATUS, port, 0);
            printf("    LPT%c status: PRINTER IS %s\n", '1' + port, ( status == 0x90 ) ?
"READY" : "NOT READY");
        }


        /* _bios_serialcom() test */
        puts("\nTest of serial ports:");
        for(port=0; port < equip.bits.comports; port++)
        {
            status = _bios_serialcom(_COM_STATUS, port, 0);

            /* Report status of each serial port and test whether there is a
              * responding device (such as a modem) for each. If data-set-ready
              * and clear-to-send bits are set, a device is responding.
              */
            printf("    COM%c status: DEVICE IS %s\n", '1' + port, (status & 0x0030) ?
"ACTIVE" : "NOT ACTIVE" );
        }


        /* _bios_disk() test, read the partition table from C: drive. */
        puts("\nContents of master boot sector:");
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        di.drive    = 0x80;
        di.head     = 0;
        di.track    = 0;
        di.sector   = 1;
        di.nsectors = 1;
        di.buffer   = diskbuf;

        /* Try reading disk three times before giving up. */
        for(i=0; i<3; i++)
        {
            status = _bios_disk(_DISK_READ, &di) >> 8;
            if ( !status )
                break;
        }
        if ( status )
            printf("Disk error: 0x%.2x\n", status);
        else
        {
            for(p=diskbuf, i=j=0; i<512; i++, p++)
            {
                linebuf[j++] = (*p > 32) ? *p : '.';
                printf("%.2x ", *p);
                if (j == 16)
                {
                    linebuf[j] = '\0';
                    printf(" %16s\n", linebuf);
                    j = 0;
                }
            }
        }

}
```

上面的代码，使用如下的命令编译：

gcc biostest.c -o biostest.exe

程序运行效果如下：

Diskdrive ： YesCoprocessor ： YesaflrrleadapterNoSerialports ： 2Numberofprinters1Sizeofmemory：1638K

PDF-XCHANGE Click to buy NOW! www.docu-track.com

Test0fparalelports：
LPTlstatus：PRINTERISNOTREADY
Testofserialports：
COMlstalXls：DEVICEISACTIVE
COM2Sta[tlS：DEVICEISACLIVE
Conterltsofmasterbootsector：

8.4.2 块设备驱动

块设备包括硬盘驱动器、软盘驱动器和光盘驱动器等。本节提供了一个块设备驱动演示程序，此程序将从 CD-ROM 光盘中复制 180KB（10 个软盘磁道的数据）到软盘上，并且在屏幕上打印运行的信息。

本节提供的程序完全使用了最底层的硬件设备能力，没有借助任何操作系统提供的功能，向读者详细演示了如何控制 IDE 设备和软盘驱动器。

此程序（bddrv.c）使用如下的命令编译：

gcc bddrv.c -o bddrv.exe

使用这个程序，需要符合下面的要求：

- CD-ROM 放置在 IDE0 的从设备上，如果在主设备，就会出错。
- 插入 A 驱动器的软盘必须是格式化的，可以读写。
- 在 CD-ROM 中插入了一张可以读的 CD 数据光盘。
- 此程序不能在 Windows 的 DOS 窗口中执行，需要在纯 DOS 下执行。

bddrv.c 程序的源代码在本书附带的光盘上，文件名为 bddrv.c，源程序中有详细的注释，读者可以阅读。本节略去源代码。

8.4.3 键盘驱动

键盘是计算机系统的重要输入设备，所有的 IBM PC 及其兼容机都有一个键盘。所以键盘驱动是一个面向 IBM PC 机操作系统的必不可少的部分。本节将详细介绍键盘及其基本工作原理和键盘驱动的实现原理。

1. IBM PC 键盘的硬件结构

在 IBM AT 和 IBM PS/ 2 键盘系统中，CPU 并不直接和键盘进行通信，而是通过一个 8042 芯片或者其他与之兼容的芯片。增加这么一个中间层，就可以屏蔽掉不同键盘之间实现的差别，并可以增加新的特性，如图 8-1 所示。

图 8-1　IBM PC 键盘的构成

CPU 直接和 8042 芯片进行通信，以实现对整个键盘的控制；键盘从外界输入得到的数据也可以通过 8042 芯片通知给 CPU，然后 CPU 可以通过 8042 芯片读取这些数据。另外，CPU 也直接向 8042 芯片发送命令，以使用 8042 芯片自身所提供的功能。

键盘自身也有自己的芯片（Intel 8048 及其兼容芯片），此芯片的功能主要是检索来自于 KeyMatrix 的外界输入（击键（Press key）或释放键（Release key））所产生的 Scan code，并将这些 Scan code 存放于键盘自身的内部缓冲；还负责和外部系统（i8042）之间的通信，以及自身的控制（SelfTest，Reset，etc）等等。

2. 8042 控制器

8042 芯片除了被用来控制键盘，作为键盘和 CPU 之间的桥梁之外，还可以被用来控制 A20 Gate，以决定 CPU 是否可以访问以 MB 为单位的偶数内存；以及向系统发送 Reset 信号，让主机重新启动。

8042 有四个寄存器：

- 一个 8 bit 长的 Input buffer，只写。
- 一个 8 bit 长的 Output buffer，只读。
- 一个 8 bit 长的 Status Register，只读。
- 一个 8 bit 长的 Control Register，可读可写。

其中 Control Register 又被称作 Command Byte。8042 有两个端口，对这两个端口进行读写，可以分别对上述四个寄存器进行操作，方法如表 8-2 所示。

表 8-2　键盘 I/O 寄存器

地址	读 / 写	作用
60h	读	读 Output buffer
60h	写	写 Input buffer（8042 Data 和 8048 Command）
64h	只写	写 Control Register
64h	只读	读 Status Register

上面提到的四个寄存器中，前三个寄存器都可以通过 60h 和 64h 直接访问，但第四个寄存器只能够通过向 64h 端口发送命令，然后通过 60h 端口存取。

状态寄存器中存放的是一些与缓冲状态、数据状态、及键盘状态有关的状态信息，是一个 8 位长的寄存器。它对于操作系统来说是只读的，并且只能够通过 64h 端口读取，任何时候，只要读取 64h 端口，都会读到状态寄存器的内容。

在 64h 状态寄存器中，各个各位（bit）的含义如表 8-3 所示。

表 8-3　64h 状态寄存器的位含义

bit	标识	含义
b7	PERR	键盘接收数据奇偶校验错误（parityrenDr）
b6	GTO	数据接收超时

PDF-XCHANGE Click to buy NOW! www.docu-track.com

b5		数据传输超时
b4		键盘锁定
b3		0=最后一次访问的端口是 60h；1=最后一次访问的端口是 61h
b2		系统标志：0=上电 / 重启动（power-up/reset），1=系统自检（selftest）
b1	IBF	输入缓冲区（从主机到键盘的缓冲区）满
b0	OBF	输出缓冲区（从键盘到主机的缓冲区）满

Input buffer 被用来向 8042 芯片（8042 芯片的输入端口各位（bit）的含义如表 8-4 所示）发送命令与数据，以控制 8042 芯片和 8048 芯片。Input buffer 可以通过 60h 端口和 64h 端口写入，其中通过 64h 端口写入的是用来控制 8042 芯片的命令，而通过 60h 写入的数据有两种：一种是通过 64h 端口发送的被用来控制 8042 芯片的命令所需要的进一步数据；另一种则是直接发给键盘用来控制 8048 芯片的命令。

Output buffer 被存放可以通过 60h 端口读取的数据。这些数据分为两大类：一类是那些通过 64h 端口发送的，被用来控制 8042 芯片的命令的返回结果；另一类则是 8048 芯片所发过来的数据。后者的数据又分为扫描码（scan code），和对那些通过 60h 端口发送给 8048 的命令的回复结果。

8-4　8042 芯片输出端口各位（bit）的含义

bit	**含义**
b7	键盘数据输出（keyboard data output）
b6	键盘时钟输出（keyboard clock output）
b5	输入缓冲区未满（input buffer NOT full）
b4	输出缓冲区未空（output buffer NOT empty）
b3	不定
b2	不定
b1	A20 门（A20 gate）
b0	系统重启动（这个 bit 通常设置为 1）

使用控制器命令字 C0h 来读取 8042 芯片的输入端口信息。8042 芯片的输入端口各位（bit）的含义如表 8-5 所示。

表 8-5　8042 芯片输入端口各位（bit）的含义

bit	**含义**
b7	键盘未锁定
b6～b0	不定

使用控制命令字 60h 来写入命令字节（Command Byte），使用控制命令字 20h 来读

取。命令字节（Command Byte）的各位（bit）的含义如表 8-6 所示。

表 8-6　命令字节（Command Byte）各位（bit）的含义

b7		**保留**
b6	KCC	转换扫描码（scan code）集合 2 到扫描码集合 1（IBM PC 兼容模式）
b5	DMS	置位，禁止 PS/2 鼠标
b4		置位，禁止键盘
b3		置位，忽略键盘锁定
b2	SYS	系统标志（与状态寄存器的 b2 意义一样）
b1		置位，容许 PS/2 鼠标的 IRQ12
b0	EKI	键盘输出缓冲区满时容许 IRQ1

8.4.4　访问 8042 芯片各端口

通过 8042 芯片，可以完成如下的工作：

- 向 8042 芯片发布命令（通过 64h），并通过 60h 读取命令的返回结果（如果有的话），或通过 60h 端口写入命令所需的数据（如果需要的话）。
- 读取 StatusRegister 的内容（通过 64h）。
- 向 8048 发布命令（通过 60h）。
- 读取来自于键盘的数据（通过 60h）。这些数据包括 ScanCode（由按键和释放键引起的），对 8048 发送的命令的确认字节（ACK）及回复数据。

对 8042 芯片各端口的操作方法如下：

- 64h 端口（读操作）

对 64h 端口进行读操作，会读取 Status Register 的内容。

inb %x64

执行这个指令之后，AL 寄存器中存放的就是 Status Register 的内容。

- 64h 端口（写操作）

向 64h 端口写入的字节，被认为是对 8042 芯片发布的命令（Command）：

写入的字节将会被存放在 Input Register 中。同时会引起 Status Register 的 bit3 自动被设置为 1，表示现在放在 Input Register 中的数据是一个 Command，而不是一个 Data。在向 64h 端口写某些命令之前必须确保键盘是被禁止的，因为这些被写入的命令的返回结果将会放到 Output Register 中，而键盘如果不被禁止，则也会将数据放入到 Output Register 中，会引起相互之间的数据覆盖。

在向 64h 端口写数据之前必须确保 Input Register 是空的（通过判断 Starus Register 的 bit1 是否为 0）。

'voidwait—input—empty（vold）char—b；dol—b=inb（0x64）；}while（!（一 b&0x02））

voiddisable—keyboard（void）wait—input—empty()outb（0x64，0xAD）；

- 60h 端口（读操作）

对 60h 端口进行读操作，将会读取 Output Register 的内容。Output Register 的内容可能是来自于 8048 的数据。这些数据包括 Scan Code，对 8048 发送的命令的确认字节（ACK）及回复数据。通过 64h 端口对 8042 发布的命令的返回结果。

在向 60h 端口读取数据之前必须确保 Output Register 中有数据（通过判断 Status Register 的 bit0 是否为 1）。

voidwait—output—full（void）charb—~1—82——~————————L～inuxO，01 内核分析与操作系统设计，————————————————==：二兰：竺!!竺兰坚!!：

d0{

一 b。inb（0x64）；

1while（__b&0x01）；unsignedcharread—output（void）fwait—output—full()；returninb（0x60）；

- 60h 端口（写操作）

向 60h 端口写入的字节，有两种可能：

如果之前通过 64h 端口向 8042 芯片发布的命令需要进一步的数据，则此时写入的字节就被认为是数据。

否则，此字节被认为是发送给 8048 的命令。

在向 60h 端口写数据之前必须确保 Input Register 是空的（通过判断 Status Register 的 bit1 是否为 0）。

8.4.5 发给 8042 的命令

- 20h

准备读取 8042 芯片的 Command Byte；其行为是将当前 8042 Command Byte 的内容放置于 Output Register 中，下一个 60h 端口的读操作会将其读取出来。

“ “ 。ignedcharread—command—byte（vojd）wmt—input—empty()outb（0x64，0x20）；walt—output—fulI()0returninb（0x60）；

- 60h

准备写入 8042 芯片的 Command Byte；下一个通过 60h 写入的字节将会被放入 Command Byte。

；mandytetom-

……I。。Jqz；双耿 / L

vfoidwri 把一 cmand—byte（unsignedch~rcornmand_hyte）

w 疵一 input—empty()；

outb（0x64，0x60）；

wait—input～empty()；

PDF-XCHANGE Click to buy NOW! www.docu-track.com

outb（0x60，command—byte）；

测试一下键盘密码是否被设置；测试结果放置在 Output Register，然后可以通过 60h 读取出来。测试结果可以有两种值：FAh=密码被设置；F1h=没有密码。

hodis．．Set——password(void)

wait～input—empty()；

OUtb(0x64，0xA4)；

wait—output—MI()；

retumlnb(0x601==OxFlA?拄 ne：触 se；

- A5h

设置键盘密码。其结果按照顺序通过 60h 端口一个一个地被放置在 Input Register 中。密码的最后是一个空字节（内容为 0）。

void set—password(urL~gaed 幽 g ” password)

char ” P。p0．~word；

if(p==NULL)

return；

wait—input～empty()；

Olltb(0x64，0xA5)；

d0{

wait—input—empty()；

OHtb(0x60，” p)；

1while(“ p++!=0)；

- A6h

让密码生效。在发布这个命令之前，必须首先使用 A5h 命令设置密码。

void enab|e～p~ssword(void)

{

if(!is—set—pa．~6"word())

mn∞；

wait—input—empty()；

outb(0x64，0xA6)；

- AAh

自检。诊断结果放置在 Output Register 中，可以通过 60h 读取。

hod is test—ok(vold)

wait—nput—empty()

- ADh

禁止键盘接口。Command Byte 的 bit4 被设置。当此命令发布后，键盘将被禁止发送数据到 Output Register。

voiddisable—keyboard（void）wait—input—empty()outb（0x64，0xAD)；

- AEh

打开键盘接口。Command Byte 的 bit4 被清除。当此命令被发布后，键盘将被允许发送数据到 Output Register。

voidenable—keyboard（void）wait—input—empty()outb（0x64，0xAE）；◆

- C0h

准备读取 Input Port。Input Port 的内容被放置于 Output Register 中，随后可以通过 60h 端口读取。

unsignedcharread—input—port（void）wait—input—empty()outb（0x64，0xC0）；wait—otitput—full()；returninb（0x60）；

- D0h

准备读取 Output port 端口。结果被放在 Output Registeree，随后通过 60h 端口读取出来。

unsignedeharread—output～port（void）fwait—input—empty()；outb（0x64，0xD0）；wait—output—full()；retuminb（0x60）；第 8 章设备管理和调度 · 185 • ◆

- D1h

准备写 Output 端口。随后通过 60h 端口写入的字节，会被放置在 OutputPort 中。

voidwrite—output—port（unsignedchar—c）“ wait—input—empty()；outb（0x64，0xDl）；wait_input—empty()；outb（0x60，一 C）；母

- D2h

准备写数据到 Output Register 中。随后通过 60h 写入到 InputRegister 的字节会被放入到 OutputRegister 中，此功能被用来模拟来自于键盘发送的数据。如果中断被允许，则会触发一个中断。

voidput—data—t0 一 output—register（unsignedchardata）

1

wait—input—empty()；

0utb（0x64，0xm）：

。击一 input 一∞pt “）；。outb（0x60，一 c）；。

8.4.6 发给 8048 的命令

- Edh

设置 LED。键盘收到此命令后，一个 LED 设置会话开始。键盘首先回复一个 ACK（FAh），然后等待从 60h 端口写入的 LED 设置字节，如果等到一个，则再次回复一个 ACK，然后根据此字节设置 LED。然后接着等待……直到等到一个非 LED 设置字节（高位被设置），此时 LED 设置会话结束。

- Eeh

诊断 Echo。此命令纯粹为了检测键盘是否正常，如果正常，当键盘收到此命令后，将会回复一个 EEh 字节。

- F0h

PDF-XCHANGE Click to buy NOW! www.docu-track.com

选择 Scan code set。键盘系统共可能有 3 个 Scan code set。当键盘收到此命令后，将回复一个 ACK，然后等待一个来自于 60h 端口的 Scan code set 代码。系统必须在此命令之后发送给键盘一个 Scan code set 代码。当键盘收到此代码后，将再次回复一个 ACK，然后将 Scan code set 设置为收到的 Scan code set 代码所要求的。

- F2h

读取键盘 ID。由于 8042 芯片后不仅仅能够接键盘。此命令是为了读取 8042 后所接的设备 ID。设备 ID 为 2 个字节，键盘 ID 为 83ABh。当键盘收到此命令后，会首先回复一个 ACK，然后，将 2 字节的键盘 ID 一个一个回复回去。

- F3h

设置 Typematic Rate/Delay。当键盘收到此命令后，将回复一个 ACK。然后等待来自于 60h 的设置字节。一旦收到，将回复一个 ACK，然后将键盘 Rate/Delay 设置为相应的值。

- F4h

清理键盘的 Output Buffer。一旦键盘收到此命令，将会将 Output buffer 清空，然后回复一个 ACK。然后继续接受键盘的击键。

- F5h

设置默认状态（w/Disable）。一旦键盘收到此命令，将会将键盘完全初始化成默认状态。之前所有对它的设置都将失效——Output buffer 被清空，Typematic Rate/Delay 被设置成默认值。然后回复一个 ACK，接着等待下一个命令。需要注意的是，这个命令被执行后，键盘的击键接受是禁止的。如果想让键盘接受击键输入，必须 Enable 键盘。

- F6h

设置默认状态。和 F5 命令惟一不同的是，当此命令被执行之后，键盘的击键接收是允许的。

- FEh

Resend。如果键盘收到此命令，则必须将刚才发送到 8042 Output Register 中的数据重新发送一遍。当系统检测到一个来自于键盘的错误之后，可以使用自命令让键盘重新发送刚才发送的字节。

- FFh

Reset 键盘。如果键盘收到此命令，则首先回复一个 ACK，然后启动自身的 Reset 程序，并进行自身基本正确性检测（BAT：BasicAssurance Test）。等这一切结束之后，将返回给系统一个单字节的结束码（AAh=Success，FCh=Failed），并将键盘的 Scan code set 设置为 2。

8.4.7 8048 到 8042 的数据

- 00h/FFh

当击键或释放键时若检测到错误，则在 Output Bufer 后放入此字节，如果 Output

Buffer 已满，则会将 Output Buffer 的最后一个字节替代为此字节。使用 Scan code set 1 时使用 00h，Scan code set 2 和 Scan code set 3 使用 FFh。

- Aah

BAT 完成代码。如果键盘检测成功，则会将此字节发送到 8042 Output Register 中。

- Eeh

Echo 响应。键盘使用 EEh 响应从 60h 发来的 Echo 请求。

- F0h

在 Scan code set 2 和 Scan code set 3 中，被用作 Break Code 的前缀。

- Fah

ACK。当键盘任何时候收到一个来自于 60h 端口的合法命令或合法数据之后，都回复一个 FAh。

- FCh

BAT 失败代码。如果键盘检测失败，则会将此字节发送到 8042 Output Register 中。

- FEh

Resend，，当键盘任何时候收到一个来自于 60h 端口的非法命令或非法数据之后，或者数据的奇偶交验错误，都回复一个 FEh，要求系统重新发送相关命令或数据。

- 83Abh

当键盘收到一个来自于 60h 的 F2h 命令之后，会依次回复 83h，ABh。83AB 是键盘的 ID。

- Scan code

除了上述那些特殊字节以外，剩下的都是 Scan code。

8.4.8 键盘源代码

1. 键盘扫描码

本书的光盘中有两个文件 scancodext.txt 和 scancodeat.txt。XT 键盘的扫描码见文件 scancodext.txt，AT 键盘的扫描码见文件 scancodeat.txt。

2. 完整的键盘驱动程序

kbd.c 是一个完整的键盘驱动程序，略加修改，就可以直接使用在操作系统设计中。kbd.c 可以使用 DJGPP 或者 Turbo C/Borland C++编译。kbd.c 详细的源代码见本书附带的光盘，源代码中有详细的注释。

8.4.9 探测软盘驱动器

如果计算机上安装了软盘驱动器，可以从计算机的 BIOS 中获得驱动器的类型。这样，在操作系统中可以对这些驱动器进行操作。

值得一提的是，BIOS 虽然提供了 0x13 中断来访问驱动器，但是 0x13 中断仅能在 x86 的实模式下使用，在保护模式下无法使用 0x13 中断。如果在保护模式下操作系统需要访问驱动器，那么必须由操作系统实现保护模式下的驱动程序。

本节将介绍操作系统如何从 BIOS 数据中获得计算机上配置软盘驱动器的信息。因为，BIOS 的数据存放在计算机的 CMOS 中，因此本节也演示了如何从 CMOS 中获得数据。

1. 从 CMOS 中取得数据

访问 CMOS，需要访问计算机的两个端口：0x70 和 0x71。0x70 是索引端口，向这个端口写入正确的索引值，可以设定需要访问的 CMOS 信息。0x71 是数据端口，根据在 0x70 端口设定的索引，可以读出相应的 CMOS 信息。

为了读取计算机中配置的软盘信息，可以向 0x70 端口中写入索引值 0x10，然后从 0x71 端口读出 CMOS 信息，具体的代码如下：

```
unsigned char c;
outportb(0x70, 0x10);
c = inportb(0x71);
```

执行完上面的代码后，在变量 c 中就包含了关于软盘驱动器的信息。下面的过程，就是从这一个字节的数据中获得需要的驱动器信息。

2. 解码数据

从 0x71 端口读出的一字节数据的解码工作是比较简单的，这一个字节包含了计算机中可能安装的两个软盘驱动器的信息（在普通的 PC 机中，最多只能安装两台软盘驱动器）。这个字节的高 4 位代表了第一个软盘驱动器，低 4 位代表了第二个软盘驱动器，具体的解码代码如下：

```
a = c >> 4        // 得到高 4 位
b = c & 0xF       // 高位与 0xF 逻辑与得到低 4 位
```

上述解码获得的 4 位二进制数值代表的含义如表 8-7 所示。

表 8-7　CMOS 中软盘的数值涵义

数值	驱动器类型
1	360KB 5.25in
2	1.2KB 5.25in
3	720KB 3.5in
4	1.44MB 3.5in
5	2.8MB 3.5in
0	No drive

可以使用如下的代码来将上述的数值翻译为描述语言：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
char drive_type[][50] = {
                                        "no floppy drive",
                                        "360kb 5.25in floppy drive",
                                        "1.2mb 5.25in floppy drive",
                                        "720kb 3.5in",
                                        "1.44mb 3.5in",
                                        "2.88mb 3.5in" };

    printf("Floppy drive A is an:\n");
    printf(drive_type[a]);
    printf("\nFloppy drive B is an:\n");
    printf(drive_type[b]);
    printf("\n");
```

在操作系统中，探测软盘驱动器状况的完整的代码如下：

```
/* df.c */
#include <stdio.h>
#include <pc.h>

void detect_floppy_drives()
{
    unsigned char c;
    unsigned char a, b;
    char drive_type[][50] = {
                                        "no floppy drive",
                                        "360kb 5.25in floppy drive",
                                        "1.2mb 5.25in floppy drive",
                                        "720kb 3.5in",
                                        "1.44mb 3.5in",
                                        "2.88mb 3.5in" };

    outportb(0x70, 0x10);
    c = inportb(0x71);

    a = c >> 4;             // 得到高 4 位
    b = c & 0xF;   //将高位与 0xF 逻辑与得到低 4 位

    printf("Floppy drive A is an:\n");
    printf(drive_type[a]);
```

```
        printf("\nFloppy drive B is an:\n");
        printf(drive_type[b]);
        printf("\n");
    };

    int main()
    {
        detect_floppy_drives();

        return 0;
    }
```

上述代码编译后，运行效果如图 8-2 所示。从图 8-2 可以看到，这台计算机上只安装了一台 1.44MB 的软盘驱动器。

图 8-2　探测软盘驱动器的运行效果

8.5　Linux 0.01 设备驱动程序源代码分析

本节将重点分析 Linux 0.01 的设备驱动代码的实现。对于 Linux 0.01 中的硬盘驱动程序 hd.c 和键盘驱动程序 kerborad.s，由于篇幅较长，在这里略去源文件。带有详细注释的源文件可以在本书的光盘中查找到。本节将重点介绍 Linux0，01 的控制台代码和串行通信代码。

8.5.1　终端控制代码：console.c

console.c 主要实现了控制台功能，包括 con_init()和 con_write()函数。下面是 console.c 的主要源代码：

```
/*
 *  console.c
 *
 * This module implements the console io functions
 *  'void con_init(void)'
 *  'void con_write(struct tty_queue * queue)'
 * Hopefully this will be a rather complete VT102 implementation.
 *
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 */

/*
 *    NOTE!!! We sometimes disable and enable interrupts for a short while
 * (to put a word in video IO), but this will work even for keyboard
 * interrupts. We know interrupts aren't enabled when getting a keyboard
 * interrupt, as we use trap-gates. Hopefully all is well.
 */

#include <linux/sched.h>
#include <linux/tty.h>
#include <asm/io.h>
#include <asm/system.h>

#define SCREEN_START 0xb8000       //0Xb8000---0XbFFFF 是显存的地址
#define SCREEN_END      0xc0000
#define LINES 25
#define COLUMNS 80
#define NPAR 16

extern void keyboard_interrupt(void);

static unsigned long origin=SCREEN_START;
static unsigned long scr_end=SCREEN_START+LINES*COLUMNS*2;
static unsigned long pos;
static unsigned long x,y;
static unsigned long top=0,bottom=LINES;
static unsigned long lines=LINES,columns=COLUMNS;
static unsigned long state=0;
static unsigned long npar,par[NPAR];
static unsigned long ques=0;
static unsigned char attr=0x07;

/*
 * this is what the terminal answers to a ESC-Z or csi0c
 * query (= vt100 response).
 */
#define RESPONSE "\033[?1;2c"         //\033=0X1B=ESC 键
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
static inline void gotoxy(unsigned int new_x,unsigned int new_y)
{
    if (new_x>=columns || new_y>=lines)
        return;
    x=new_x;
    y=new_y;
    pos=origin+((y*columns+x)<<1); //乘 2 是因为描述位置是字
}

static inline void set_origin(void)
{
    cli();
//下面二个寄存器处理屏幕滚动，寄存器 12 是开始地址的高字，13 是低字
//0x3d4 是 6845 索引寄存器，它可以选择 18 个寄存器，但必须在读/写 0x3d5 之前，
18 个寄存器中前 10 个是处理垂直和水平显示参数的，不要动
    outb_p(12,0x3d4);
    outb_p(0xff&((origin-SCREEN_START)>>9),0x3d5);
    outb_p(13,0x3d4);
    outb_p(0xff&((origin-SCREEN_START)>>1),0x3d5);
    sti();
}

static void scrup(void)
{
    if (!top && bottom==lines) {
        origin += columns<<1;
        pos += columns<<1;
        scr_end += columns<<1;
        if (scr_end>SCREEN_END) {
//以下的内嵌汇编意思是：
//  movl    0X0720      %eax        //其中 0X0720 中的 07 表示属性为灰，20
代表空白键
//  movl    (line-1)*columns>>1   %ecx
//  movl    SCREEN_START  %edi
//  movl    origin        %esi
//  cld                   //地址增量
//  rep  movsl                  //ds:si --->es:di
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
//      movl      _columns%ecx
//      rep   stosw                         //将 eax 中的内容传入 ES：DI 中
                __asm__("cld\n\t"
                        "rep\n\t"
                        "movsl\n\t"
                        "movl _columns,%1\n\t"
                        "rep\n\t"
                        "stosw"
                        ::"a" (0x0720),
                        "c" ((lines-1)*columns>>1),
                        "D" (SCREEN_START),
                        "S" (origin)
                        :"cx","di","si");
                scr_end -= origin-SCREEN_START;
                pos -= origin-SCREEN_START;
                origin = SCREEN_START;
        } else {
//以下的内嵌汇编意思是：
//      movl      0X07200720   %eax
//      movl      columns>>1   %ecx          //因为传送的是双字，所以要除 2
//      movl      scr_end-(columns<<1)   %edi
//      cld
//      rep   stosl                 //双字传送
                __asm__("cld\n\t"
                        "rep\n\t"
                        "stosl"
                        ::"a" (0x07200720),
                        "c" (columns>>1),
                        "D" (scr_end-(columns<<1))
                        :"cx","di");
        }
        set_origin();
    } else {
//以下的内嵌汇编意思是：
//      movl      0X0720        %eax
//      movl      (bottom-top-1)*columns>>1  %ecx
//      movl      origin+(columns<<1)*top          %edi
//      movl      origin+(columns<<1)*(top+1)          %esi
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
//      cld
//      rep   movsl
//      movl        _columns%ecx
//      rep   stosw
            __asm__("cld\n\t"
                    "rep\n\t"
                    "movsl\n\t"
                    "movl _columns,%%ecx\n\t"
                    "rep\n\t"
                    "stosw"
                    ::"a" (0x0720),
                    "c" ((bottom-top-1)*columns>>1),
                    "D" (origin+(columns<<1)*top),
                    "S" (origin+(columns<<1)*(top+1))
                    :"cx","di","si");
        }
}

static void scrdown(void)
{
//以下的内嵌汇编意思是：
//      movl        0X0720          %eax
//      movl        (bottom-top-1)*columns %ecx
//      movl        (origin+(columns<<1)*(bottom-1)-4)    %esi
//      std                     //地址减量
//      rep   movsl
//      addl $2    %%edi
//      movl        _columns%ecx
//      stosw
        __asm__("std\n\t"
                "rep\n\t"
                "movsl\n\t"
                "addl $2,%%edi\n\t"       /* %edi has been decremented by 4 */
                "movl _columns,%%ecx\n\t"
                "rep\n\t"
                "stosw"
                ::"a" (0x0720),
                "c" ((bottom-top-1)*columns>>1),
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            "D" (origin+(columns<<1)*bottom-4),
            "S" (origin+(columns<<1)*(bottom-1)-4)
            :"ax","cx","di","si");
}

static void lf(void)
{
    if (y+1<bottom) {
        y++;
        pos += columns<<1;
        return;
    }
    scrup();                //如果超过了底部，屏向上走一行
}

static void ri(void)
{
    if (y>top) {
        y--;
        pos -= columns<<1;
        return;
    }
    scrdown();              //如果 y<top，就是超过了屏的顶部，就向下走一行
}

static void cr(void)
{
    pos -= x<<1;
    x=0;
}

static void del(void)
{
    if (x) {
        pos -= 2;
        x--;
        *(unsigned short *)pos = 0x0720;  //将删除的这个字填为空白，属性为灰
    }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
}

static void csi_J(int par)        //这是处理整个显示区的擦除函数
{
        long count __asm__("cx");
        long start __asm__("di");

        switch (par) {
                case 0:     /* erase from cursor to end of display */
                        count = (scr_end-pos)>>1;
                        start = pos;
                        break;
                case 1:     /* erase from start to cursor */
                        count = (pos-origin)>>1;
                        start = origin;
                        break;
                case 2: /* erase whole display */
                        count = columns*lines;
                        start = origin;
                        break;
                default:
                        return;
        }
//以下的内嵌汇编意思是:
//      movl        count       %ecx
//      movl        start %edi
//      movl        0X0720   %eax
//      cld
//      rep   stosw
        __asm__("cld\n\t"
                "rep\n\t"
                "stosw\n\t"
                ::"c" (count),
                "D" (start),"a" (0x0720)
                :"cx","di");
}

static void csi_K(int par)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
{
        long count __asm__("cx");
        long start __asm__("di");

        switch (par) {
                case 0:     /* erase from cursor to end of line */
                        if (x>=columns)
                                return;
                        count = columns-x;
                        start = pos;
                        break;
                case 1:     /* erase from start of line to cursor */
                        start = pos - (x<<1);
                        count = (x<columns)?x:columns;
                        break;
                case 2: /* erase whole line */
                        start = pos - (x<<1);
                        count = columns;
                        break;
                default:
                        return;
        }
        __asm__("cld\n\t"                   //同上
                "rep\n\t"
                "stosw\n\t"
                ::"c" (count),
                "D" (start),"a" (0x0720)
                :"cx","di");
}

void csi_m(void)                            //修改部分区域的字符属性
{
        int i;

        for (i=0;i<=npar;i++)
                switch (par[i]) {
                        case 0:attr=0x07;break;
                        case 1:attr=0x0f;break;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                case 4:attr=0x0f;break;
                case 7:attr=0x70;break;
                case 27:attr=0x07;break;
            }
}

static inline void set_cursor(void)
{
    cli();
//寄存器 14 控制光标位置（高位字），寄存器 15 控制低位字
    outb_p(14,0x3d4);
    outb_p(0xff&((pos-SCREEN_START)>>9),0x3d5);
    outb_p(15,0x3d4);
    outb_p(0xff&((pos-SCREEN_START)>>1),0x3d5);
    sti();
}

static void respond(struct tty_struct * tty)
{
    char * p = RESPONSE;

    cli();
    while (*p) {
        PUTCH(*p,tty->read_q);
        p++;
    }
    sti();
    copy_to_cooked(tty);                    //此函数在 tty_io.c 中定义
}

static void insert_char(void)
{
    int i=x;
    unsigned short tmp,old=0x0720;
    unsigned short * p = (unsigned short *) pos;

    while (i++<columns) {          //将光标的当前位置填空为 0X0720，然后循环，把
```

当前位置的老的字符向后移，然后将后面的所有字符向后移一位置

```
            tmp=*p;
            *p=old;
            old=tmp;
            p++;
        }
}

static void insert_line(void)
{
        int oldtop,oldbottom;

        oldtop=top;
        oldbottom=bottom;
        top=y;                  //top 是当前光标位置的 Y 轴，即为当前屏幕的第 Y 行
        bottom=lines;
        scrdown();
        top=oldtop;
        bottom=oldbottom;
}

static void delete_char(void)
{
        int i;
        unsigned short * p = (unsigned short *) pos;

        if (x>=columns)
                return;
        i = x;                  //X 为当前光标的 X 轴，即为当前屏幕的 X 列
        while (++i < columns) {
                *p = *(p+1);
                p++;
        }
        *p=0x0720;              //将最后一个字符填为 0X0720
}

static void delete_line(void)
{
        int oldtop,oldbottom;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        oldtop=top;
        oldbottom=bottom;
        top=y;
        bottom=lines;
        scrup();
        top=oldtop;
        bottom=oldbottom;
}

static void csi_at(int nr)
{
        if (nr>columns)
                nr=columns;
        else if (!nr)                   //如果 nr 为 0，那么 nr=1，这样是保证插入一个字符
                nr=1;
        while (nr--)
                insert_char();
}

static void csi_L(int nr)
{
        if (nr>lines)
                nr=lines;
        else if (!nr)
                nr=1;
        while (nr--)
                insert_line();
}

static void csi_P(int nr)
{
        if (nr>columns)
                nr=columns;
        else if (!nr)
                nr=1;
        while (nr--)
                delete_char();
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
}

static void csi_M(int nr)
{
	if (nr>lines)
		nr=lines;
	else if (!nr)
		nr=1;
	while (nr--)
		delete_line();
}

static int saved_x=0;
static int saved_y=0;

static void save_cur(void)
{
	saved_x=x;
	saved_y=y;
}

static void restore_cur(void)
{
	x=saved_x;
	y=saved_y;
	pos=origin+((y*columns+x)<<1);		//还原光标位置
}

void con_write(struct tty_struct * tty)
{
	int nr;
	char c;

	nr = CHARS(tty->write_q);			//tty->write_q 中有多少没有被处理的字符
	while (nr--) {
		GETCH(tty->write_q,c);
		switch(state) {			//state 默认为 0
			case 0:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                if (c>31 && c<127) {          //是有效可写字符?
                    if (x>=columns) {  //需要换行吗?
                        x -= columns; //将光标 X 轴指向本行的行首
                        pos -= columns<<1;
                        lf();       //将光标指向下一行
                    }
    //下面内嵌汇编的意思是:
    //      movl      c      %eax
    //      movb      0X70      %ah
    //      movw      %ax *pos
                    __asm__("movb _attr,%%ah\n\t"
                        "movw %%ax,%1\n\t"
                        ::"a" (c),"m" (*(short *)pos)  //其中 m 表示操作数是内存
变量
                        :"ax");
                    pos += 2;       //指向下一个位置
                    x++;
                } else if (c==27)           //27=0X1B，是否为 ESC 键
                    state=1;
                else if (c==10 || c==11 || c==12)
                    lf();
                else if (c==13)       //13=0X0D，是否为 ENTER 键
                    cr();
                else if (c==ERASE_CHAR(tty))
                    del();
                else if (c==8) {            //8 是空白键（BACKSPACE）
                    if (x) {
                        x--;
                        pos -= 2;
                    }
                } else if (c==9) {          //TAB 键
                    c=8-(x&7);          //只能跳过最大 8 个字符
                    x += c;
                    pos += c<<1;
                    if (x>columns) {
                        x -= columns;
                        pos -= columns<<1;
                        lf();
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                }
                c=9;
        }
        break;
case 1:
        state=0;
        if (c=='[')
                state=2;
        else if (c=='E')
                gotoxy(0,y+1);      //到下行的开头位置
        else if (c=='M')
                ri();               //光标到上行的当前位置
        else if (c=='D')
                lf();               //光标到下行的当前位置
        else if (c=='Z')
                respond(tty);
        else if (x=='7')
                save_cur();
        else if (x=='8')
                restore_cur();
        break;
case 2:
        for(npar=0;npar<NPAR;npar++)
                par[npar]=0;
        npar=0;
        state=3;
        if (ques=(c=='?'))
                break;
case 3:
        if (c==';' && npar<NPAR-1) {
                npar++;
                break;
        } else if (c>='0' && c<='9') {
                par[npar]=10*par[npar]+c-'0';
                break;
        } else state=4;
case 4:                         //光标控制
        state=0;
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
switch(c) {
        case 'G': case '`':
                if (par[0]) par[0]--;
                gotoxy(par[0],y);
                break;
        case 'A':
                if (!par[0]) par[0]++;
                gotoxy(x,y-par[0]);
                break;
        case 'B': case 'e':
                if (!par[0]) par[0]++;
                gotoxy(x,y+par[0]);
                break;
        case 'C': case 'a':
                if (!par[0]) par[0]++;
                gotoxy(x+par[0],y);
                break;
        case 'D':
                if (!par[0]) par[0]++;
                gotoxy(x-par[0],y);
                break;
        case 'E':
                if (!par[0]) par[0]++;
                gotoxy(0,y+par[0]);
                break;
        case 'F':
                if (!par[0]) par[0]++;
                gotoxy(0,y-par[0]);
                break;
        case 'd':
                if (par[0]) par[0]--;
                gotoxy(x,par[0]);
                break;
        case 'H': case 'f':
                if (par[0]) par[0]--;
                if (par[1]) par[1]--;
                gotoxy(par[1],par[0]);
                break;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                case 'J':
                        csi_J(par[0]);
                        break;
                case 'K':
                        csi_K(par[0]);
                        break;
                case 'L':
                        csi_L(par[0]);
                        break;
                case 'M':
                        csi_M(par[0]);
                        break;
                case 'P':
                        csi_P(par[0]);
                        break;
                case '@':
                        csi_at(par[0]);
                        break;
                case 'm':
                        csi_m();
                        break;
                case 'r':
                        if (par[0]) par[0]--;
                        if (!par[1]) par[1]=lines;
                        if (par[0] < par[1] &&
                                par[1] <= lines) {
                                top=par[0];
                                bottom=par[1];
                        }
                        break;
                case 's':
                        save_cur();
                        break;
                case 'u':
                        restore_cur();
                        break;
        }
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        }
        set_cursor();                           //实际硬件处理光标
    }

    /*
     *    void con_init(void);
     *
     * This routine initalizes console interrupts, and does nothing
     * else. If you want the screen to clear, call tty_write with
     * the appropriate escape-sequece.
     */
    void con_init(void)
    {
        register unsigned char a;

        gotoxy(*(unsigned char *)(0x90000+510),*(unsigned char *)(0x90000+511));  // 其中 0X90000+510 是在 boot.s 中已将光标 X、Y 保存在 0X90000+510 处
        set_trap_gate(0x21,&keyboard_interrupt);        //定义 IDT[21]中断号是键盘中断处理程序，键盘中断处理程序在 keyboard.s 中
        outb_p(inb_p(0x21)&0xfd,0x21);              //屏蔽 IRQ1
        a=inb_p(0x61);
        outb_p(a|0x80,0x61);                        //禁止端口 60H 开关，开放键盘数据，允许键盘 IRQ
        outb(a,0x61);                           //恢复
    }
```

8.5.2 rs323 驱动代码：serial.c 和 rs_io.s

serial.c 实现了 rS232 串口通信功能，主要函数是：rs_write()、rs_init()以及相关的中断功能。其代码如下：

```
    /*
     *    serial.c
     *
     * This module implements the rs232 io functions
     *    void rs_write(struct tty_struct * queue);
     *    void rs_init(void);
     * and all interrupts pertaining to serial IO.
     */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#include <linux/tty.h>
#include <linux/sched.h>
#include <asm/system.h>
#include <asm/io.h>

#define WAKEUP_CHARS (TTY_BUF_SIZE/4)

extern void rs1_interrupt(void);
extern void rs2_interrupt(void);

static void init(int port)
{
        outb_p(0x80,port+3);    /* set DLAB of line control reg */      //线控制寄存器，将
第 7 位置 1 是开放除法因子锁存器地址位
        outb_p(0x30,port);  /* LS of divisor (48 -> 2400 bps *///将波特率设为 2400（低有
效位）
        outb_p(0x00,port+1);    /* MS of divisor */                 //将波特率的最高有效
位保存入该寄存器
        outb_p(0x03,port+3);    /* reset DLAB */                    //将 DLAB 设为 0，帧设
为 8 位（通常是 8 位）
        outb_p(0x0b,port+4);    /* set DTR,RTS, OUT_2 */            //设置数据终端准
备好线（DTR）的状态，设置请求发送线（RTS）的状态，out_2 用于开放中断请求
        outb_p(0x0d,port+1);    /* enable all intrs but writes */
        (void)inb(port);      /* read data port to reset things (?) */    //获取接收缓冲区数据
字节
}

void rs_init(void)
{
        set_intr_gate(0x24,rs1_interrupt);          //串行口端口 3F8 的中断好是 0x24
        set_intr_gate(0x23,rs2_interrupt);          //串行口端口 2F8 的中断好是 0x23
        init(tty_table[1].read_q.data);         //初始化 2 个串行端口
        init(tty_table[2].read_q.data);
        outb(inb_p(0x21)&0xE7,0x21);            //控制中断控制器对中断请求线 0--7
的操作
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/*
 * This routine gets called when tty_write has put something into
 * the write_queue. It must check wheter the queue is empty, and
 * set the interrupt register accordingly
 *
 *    void _rs_write(struct tty_struct * tty);
 */
void rs_write(struct tty_struct * tty)
{
        cli();
        if (!EMPTY(tty->write_q))
                outb(inb_p(tty->write_q.data+1)|0x02,tty->write_q.data+1);      //将寄存器的
第 2 位（传送位）置 1，开放传送中断
        sti();
}
```

rs_io.s 的源代码如下：

```
/*
 *    rs_io.s
 *
 * This module implements the rs232 io interrupts.
 */

.text
.globl _rs1_interrupt,_rs2_interrupt

size  = 1024                    /* must be power of two !
                                and must match the value
                                in tty_io.c!!! */

/* these are the offsets into the read/write buffer structures */
rs_addr = 0
head = 4
tail = 8
proc_list = 12
buf = 16

startup     = 256           /* chars left in write queue when we restart it */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/*
 * These are the actual interrupt routines. They look where
 * the interrupt is coming from, and take appropriate action.
 */
.align 2
_rs1_interrupt:
        pushl $_table_list+8            //这里在 tty_io.c 中定义的 table_list 结构的基地址+8
就是偏移量 8 个字节（就是 table_list[1].read_q 结构的首地址），这个结构的第一个成员
是 data（0x3f8），此行就是将它入栈
        jmp rs_int
.align 2
_rs2_interrupt:
        pushl $_table_list+16
rs_int:
        pushl %edx
        pushl %ecx
        pushl %ebx
        pushl %eax
        push %es
        push %ds        /* as this is an interrupt, we cannot */
        pushl $0x10             /* know that bs is ok. Load it */
        pop %ds                 //将 ds 设为 0X10
        pushl $0x10
        pop %es                 //将 es 设为 0X10
        movl 24(%esp),%edx              //将指针指向$_table_list+8 的地址 0x3F8（esp+24）
        movl (%edx),%edx        //将地址上的值送入 edx，值为 0x3F8
        movl rs_addr(%edx),%edx
        addl $2,%edx            /* interrupt ident. reg */          //=0X3FA
rep_int:
        xorl %eax,%eax
        inb %dx,%al                     //UART 寄存器 2（中断标识寄存器）
        testb $1,%al                    //如果 0 位为 1，那么无中断挂起，结束
        jne end
        cmpb $6,%al             /* this shouldn't happen, but ... */   //如果 al>6，就是第 3 位设
为 1，就是字符超时，结束
        ja end
        movl 24(%esp),%ecx              //将$_table_list+8（table_list[1].read_q）的基地址送
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

入 ecx

```
        pushl %edx
        subl $2,%edx            //0X3F8
        call jmp_table(,%eax,2)     /* NOTE! not *4, bit0 is 0 already */
        popl %edx
        jmp rep_int             //循环，不停的检查是否有中断发生
    end: movb $0x20,%al         //向端口 20 发送命令 20 是结束中断
        outb %al,$0x20          /* EOI */
        pop %ds
        pop %es
        popl %eax
        popl %ebx
        popl %ecx
        popl %edx
        addl $4,%esp        # jump over _table_list entry
        iret

    jmp_table:
        .long modem_status,write_char,read_char,line_status

    .align 2
    modem_status:                   //读取寄存器状态使此 0X3F8+6 的寄存器的 0—3 位
设置为 0
        addl $6,%edx        /* clear intr by reading modem status reg */
        inb %dx,%al
        ret

    .align 2
    line_status:                    //读取寄存器状态，使此 0X3F8+5 的寄存器设置为 60H，
表示没有错误，传送缓冲区为空
        addl $5,%edx        /* clear intr by reading line status reg. */
        inb %dx,%al
        ret

    .align 2
    read_char:
        inb %dx,%al             //0X3F8 寄存器输入是获得接收缓冲区数据字节
        movl %ecx,%edx          //此时 ecx 中是 table_list[1].read_q 的地址
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        subl $_table_list,%edx   //edx=8
        shrl $3,%edx             //edx=1
        movl (%ecx),%ecx         # read-queue   //将 table_list[1].read_q 的地址（ecx）中的
值送入 ecx，data 值为 3F8
        movl head(%ecx),%ebx         //将  table_list[1].read_q  结构中的第  2  个成员地址
（head）中的值送入 ebx，为 0
        movb %al,buf(%ecx,%ebx)  //将字符数据放入结构中的 char
        incl %ebx                //ebx=1
        andl $size-1,%ebx        //andl      3FF  %ebx
        cmpl tail(%ecx),%ebx         //第一次循环期间 tail（%ecx）为 0
        je 1f
        movl %ebx,head(%ecx)         //将结构成员 head 的值设为 1
        pushl %edx
        call _do_tty_interrupt
        addl $4,%esp
1:      ret

    .align 2
    write_char:
        movl 4(%ecx),%ecx            # write-queue        //ecx=table_list[1].write_q
        movl head(%ecx),%ebx         //ebx=table_list[1].write_q 结构中的 head 成员=0
        subl tail(%ecx),%ebx         //指向 table_list[1].write_q 结构中的 tail 成员=0
        andl $size-1,%ebx        # nr chars in queue
        je write_buffer_empty
        cmpl $startup,%ebx
        ja 1f
        movl proc_list(%ecx),%ebx  # wake up sleeping process
        testl %ebx,%ebx              # is there any?
        je 1f                  //ebx=0 时跳转
        movl $0,(%ebx)
1:      movl tail(%ecx),%ebx
        movb buf(%ecx,%ebx),%al   //将缓冲区的字节数据送入 al，实际写数据的代码
        outb %al,%dx
        incl %ebx                //调整 table_list[1].write_q 中的状态
        andl $size-1,%ebx
        movl %ebx,tail(%ecx)
        cmpl head(%ecx),%ebx
        je write_buffer_empty
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        ret
.align 2
write_buffer_empty:
        movl proc_list(%ecx),%ebx   # wake up sleeping process
        testl %ebx,%ebx             # is there any?
        je 1f
        movl $0,(%ebx)
1:      incl %edx                   //0X3F9 是中断开放寄存器，低 4 位有用
        inb %dx,%al
        jmp 1f
1:      jmp 1f
1:      andb $0xd,%al       /* disable transmit interrupt */
        outb %al,%dx
        ret
```

8.6 本章小结

设备管理通常是操作系统中最复杂的一部分，这是因为不同的物理设备具有不同的特性，操作系统需要使用一致的方法来管理这些设备，这要求操作系统提供一个良好设计设备管理系统。

本章重点介绍了 Linux 设备管理的基本实现原理。Linux 继承了 Unix 的优秀设计，将所有的物理设备都抽象为设备文件。这种设计思想，大大简化了操作系统和应用程序访问物理设备的操作，是一种非常成功的设计。

本章还介绍了 Linux 设备管理的基本原理，设备管理的基本要求、驱动程序接口、设备管理接口、同步和异步访问等重要问题。

同时，本章提供了大量的设备驱动实例，包括屏幕驱动、键盘驱动、IDE 设备驱动等。这些典型的设备驱动程序可以帮助读者在了解硬件基本原理的基础上，设计自己的设备管理系统和快速实现一个设备驱动程序。

最后，本章还分析了 Linux 0.01 中的设备管理程序，包括控制台实现和串行通信实现。

第9章 磁盘文件系统

本章知识点：

- 硬盘驱动器结构简介
- Unix 文件系统分析
- VFS 文件系统简介
- 文件系统设计的主要步骤
- 一个简单的文件系统设计实例
- Linux 0.01 文件系统源代码分析

文件系统是 Linux 系统的一个核心组成部分。Linux 继承了 Unix 中“ everything is a file”的思想，使用文件系统接口，从而可以控制 Linux 系统中的所有设备。Linux 文件系统具有良好的设计，可以实现对多种物理文件系统的一致访问，具有很高的性能。本章首先从文件系统的基本知识出发，介绍了硬盘驱动器原理，使读者了解文件系统实现的物理基础。然后，本章介绍经典的 Unix 文件系统。Unix 文件系统是一种具有良好设计的文件系统，本章对 Unix 文件进行了深入的分析和介绍，特别分析 Unix 文件系统的设计对性能的影响。

本章还详细介绍了如何设计和实现一个文件系统的主要步骤，重点介绍了实现磁盘驱动库、实现文件系统调用、实现文件系统的容错性等问题。

Linux 文件系统的一个优秀设计思想是使用了虚拟文件系统 VFS。通过引入 VFS 的概念，可以使 Linux 文件系统的主要部分和具体的物理文件系统类型无关。这种设计，大大地提高了 Linux 文件系统的灵活性和适应性。

本章提供了一个文件系统设计的实例，这个实例给出了文件系统的基本功能，可以帮助读者快速地掌握文件系统设计的基本技能。

最后，本章对 Linux 0.01 文件系统的源代码进行了详细的分析。

9.1 硬盘驱动器结构简介

硬盘驱动器是实现磁盘文件系统的最重要物理设备之一。本节将介绍硬盘驱动器的基本结构、重要参数和访问硬盘驱动器的 INT 13h 中断。通过本节的学习，可以使读者迅速地掌握磁盘文件系统实现的物理基础。

9.1.1 硬盘参数解释

对硬盘进行访问的基础是对硬盘的磁盘扇区进行寻址。通常，对于硬盘的寻址使用CHS（Cylinder/Head/Sector）参数。CHS 就是使用柱面数（Cylinders）、磁头数（Heads）、和扇区数（Sectors per track）来定位硬盘驱动器任何一个扇区的方法，三个参数的含义是：

- 柱面数（Cylinders）表示硬盘每一面盘片上有几条磁道，最大为 1024（用 10 个二进制位存储）。
- 磁头数（Heads）表示硬盘总共有几个磁头，也就是有几面盘片，最大为 256（用 8 个二进制位存储）。
- 扇区数（Sectors per track）表示每一条磁道上有几个扇区，最大为 63（用 6 个二进制位存储）。

每个扇区一般是 512B，所以磁盘最大容量为：

256 * 1024 * 63 * 512 / 1048576 = 8064 MB（1MB = 1048576 B）

在 CHS 寻址方式中，磁头、柱面、扇区的取值范围分别为 0～Heads - 1，0～Cylinders - 1，1～Seetors per track（注意是从 1 开始）。

在 CHS 寻址方式中，有以下几种尺寸单位：

扇区（Seetor）=512B（一般情况下）

磁道（Ttack）=（Sectors per track）扇区

柱面（Cylinder）=（Sectors pet track）* Heads 扇区

9.1.2 基本 INT13h 调用简介

BIOS INT 13h 调用是 BIOS 提供的磁盘基本输入输出中断调用，它可以完成磁盘（包括硬盘和软盘）的复位、读写、校验、定位、诊断、格式化等功能。它使用的就是 CHS 寻址方式，因此最大能访问 8GB 左右的硬盘。

9.1.3 现代硬盘结构简介

在老式硬盘中，由于每个磁道的扇区数相等，所以外道的记录密度要远低于内道，因此会浪费很多磁盘空间（与软盘一样）。为了解决这一问题，进一步提高硬盘容量，人们改用等密度结构生产硬盘。也就是说，外圈磁道的扇区比内圈磁道多。采用这种结构后，硬盘不再具有实际的 CHS 参数，寻址方式也改为线性寻址，即以扇区为单位进行寻址。为了与使用 CHS 寻址的老软件兼容（如使用 BIOS INT 13h 接口的软件），在硬盘控制器内部安装了一个地址翻译器，由它负责将老式 CHS 参数翻译成新的线性参数。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

9.1.4 扩展 INT 13h 简介

虽然现代硬盘都已经采用了线性寻址，但是由于基本 INT 13h 的制约，使用：BIOS INT 13h 接口的程序，如 DOS 等还只能访问 8GB 以内的硬盘空间。为了打破这一限制，Microsoft 等几家公司制定了扩展 INT 13h 标准（Extended INT 13h），采用线性寻址方式存取硬盘，所以突破了 8GB 的限制。‘

9.2 Unix 文件系统

Unix 文件系统是一种经典的文件系统，具有很好的适应性和很高的性能。通过对 Unix 文件系统的分析，可以使读者掌握文件系统设计和实现的基本思想。

9.2.1 磁盘的基础特性

本节介绍现代磁盘的基本特性。掌握这些基本特性，可以帮助读者理解现代磁盘文件系统的设计思想。磁盘文件系统的基本设计要求是稳定高效，并提供简单的访问接口。为了提高磁盘文件系统的性能，必须针对磁盘驱动器的特性进行设计的优化。

1. 磁盘扇区读取速度

磁盘数据是存放在扇区中的，不同形式的扇区存放形式对磁盘扇区读取速度有很大的影响。下面介绍几种典型的扇区存放形式：

扇区在磁道上连续存放

一种通常的磁盘扇区存放形式是扇区在磁道上连续存放，例如第 N、N+1、N+2…个扇区在一个磁道上连续分布。这种分布方式的缺点是：如果文件系统命令磁盘读入 N 扇区，然后对 N 扇区进行处理；接着，文件系统命令磁盘读入 N+1 扇区，这时磁盘的磁头已经旋转过了 N+1 扇区。因此必须等待磁头重新旋转到扇区 N+l 才能读出 N+1 扇区。

最后，磁头 N 次旋转，读出了 N 个扇区。磁盘的连续访问实际可用带宽为：

$Bandwidth=512Bytes/T_{ratation}$

假设磁盘的旋转时间为了 $T_{ratation}$ =3ms，那么连续访问实际可用带宽为：

$Bandwidth=512Bytes/3ms\approx 170.7KB/s$

可见，在这种扇区存放方式下磁盘连续访问的实际可用带宽非常小。

带磁盘内部 Cache 的现代磁盘

现代磁盘驱动器都有不等容量的 Build-in Cache。普通 IDE/ATA 接口的磁盘驱动器通常有 512KB～2MB 的 Build-in Cache，而 SCCI 接口的磁盘驱动器可能会有 8MB～16MB 以上的 Build-in Cache。

对于有 Build-in Cache 的磁盘驱动器，在读一个磁道扇区的时候，可以将整个磁道

全部读入 Build-in Cache 中，然后在 Build-in Cache 中将系统所需要访问的扇区进行组合。因此，在这种情况下磁盘驱动器的访问速度是：1 次旋转读出 N 个扇区。

可见，当使用了 Build-in Cache 后，磁盘驱动器访问连续扇区的实际可用带宽大大提高了。

2. 磁盘驱动器的发展趋势

磁盘驱动器是存储系统中主要的随机访问存储介质。随着技术的发展，磁盘驱动器也呈现出一些明显的发展趋势：

- 磁盘驱动器的体积变的更小了，体积的减小导致磁盘轴旋转速度更快，磁盘旋转延迟降低了。
- 磁盘驱动器的数据记录密度更大了，导致磁道间距变的更小，降低了寻道延迟。
- 磁盘驱动器的容量增大，价格降低。近 10 年来，同样存储容量的磁盘，每年价格降低一倍以上，寻道和旋转速度每年提高 5%～10%。

磁盘驱动器的总体变化趋势是：磁盘容量的增大速度远远超过了磁盘速度的提高，磁盘的机械动作（寻道和旋转等）成为磁盘驱动器性能提高的瓶颈。

9.2.2 现代磁盘的寻道延迟特性

传统的磁盘分析模型认为磁盘寻道时间和寻道距离成线性关系，但实际上现代磁盘寻道时间和寻道距离并不是线性关系。如图 9-1 所示是对 HP C2200A 型磁盘的实际测试结果。在图 9-1 中，曲线是实际的寻道时间测试结果，直线代表线性寻道延迟关系，可以看到：在寻道距离小于 400 道的情况下，磁盘的寻道时间随着寻道距离的增加而急剧增加；当寻道距离增加到 600 道左右，寻道时间的增加才趋于平缓。文献中同时也建立了 HP C2200A 型磁盘驱动器的寻道时间模型，如下式所示：

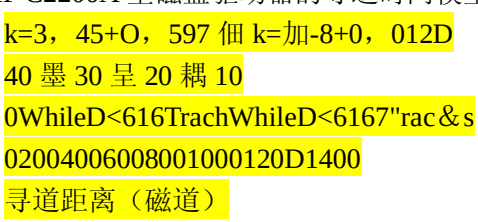

图 9-1 HP C2200A 磁盘寻道时间实际测试结果

从式 9-1 可以看出，当寻道距离小于 616 道时，HP C2200A 型磁盘驱动器的寻道时间和寻道距离之间不是线性关系。

本书对 ST 310212A 型磁盘的寻道延迟和寻道距离关系的实际测试数据如图 9-2 所示。从图 9-2 中可以看到，当寻道距离小于 400 道的情况下，ST 310212A 型磁盘的寻道时间随着寻道距离的增加也是急剧地增加。

图 9-2　ST 310212A 磁盘寻道时间实际测试结果

所以，现代磁盘的寻道延迟和寻道距离的变化关系通常是非线性的。为了提高文件系统的性能，必须针对这种非线性关系对文件系统的磁盘布局进行独特地设计

9.2.3　Unix 文件系统分析

本节将分析经典 Unix 文件系统，然后指出经典 Unix 文件系统的设计优点和在小文件读写上的不足，并介绍了对其不足之处的改进方法。

1.　Unix 文件系统基本构成

Unix 文件系统的结构如图 9-3 所示。

图 9-3　Unix 文件系统结构

- 引导块（Boot block）

通常位于文件卷最开始的第一扇区，这 512B 是文件系统的引导代码，为根文件系统所特有，其他文件系统中这 512B 为空。

超级块（Super block）

超级块紧跟引导块之后，用于描述文件系统的结构，如 i 节点长度、文件系统大小等信息。

- i 节点表（Inode table）

i 节点表存放在超级块之后，其长度由超级块中 i 节点长度字段决定，其作用是用来描述文件的属性、长度、属主、属组、数据块表等信息。

- 数据区（Data block zone）

数据区跟在 i 节点表后面，用于存放文件的数据。

在 Unix 文件系统中，一个普通文件通常由两部分组成：i 节点+数据块。

2.　Unix 文件系统性能分析

在 Unix 文件系统中，一个普通文件包含两部分：i 节点+文件数据（以后，使用元数据来代表 i 节点）。通常为了提高文件系统的性能，Unix 文件系统在向磁盘写入文件内容的同时并没有同时向磁盘写入文件的元数据，也就是 Unix 文件系统先向磁盘写入文件的内容，然后等到一个合适的时机才向磁盘写入文件的元数据。

在这种情况下，如果向磁盘写入文件内容之后，但在写入文件的元数据之前系统突然崩溃了，文件系统就会处于不一致的状态。在一个文件操作非常频繁的系统中，出现这种情况会导致很严重的后果。通常，可以通过使用文件系统的 fsck 程序来修正文件系统的不一致。但是，对于大型文件系统，fsck 的运行时间非常长。在 fsck 运行的过程中，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

服务器实际上是不可用的。

因此，对于需要确保数据安全性的应用环境中，通常文件系统对文件数据和元数据使用同步写，也就是要求应用程序必须等待文件系统将数据和元数据都写入磁盘后才能进行下一步操作。这样可以确保文件被正确地写入磁盘，而不会导致数据的丢失。

在同步写方式下，如果文件系统需要对大量的小文件进行写操作，那么将严重降低文件系统的性能。因为在 Unix 系统中，文件数据块的存放是不连续的，特别是当一个文件系统运行了很长时间以后，文件系统中的文件数据块分布可能更加不连续。因此对这些小文件的磁盘块进行读写时，磁头必须进行频繁的寻道操作，导致实际数据传输率很低。

对于 Unix 文件系统的这种不足，一种有效的解决方案是使用日志文件系统（Log-Structured File Systern）。日志文件系统的设计思想是跟踪文件系统的变化而不是文件系统的内容，所有对文件系统的更新都被记录在日志中。日志文件系统中的目志通常在磁盘中是连续分布的，因此日志的同步写操作是非常高效率的。在系统崩溃之后，日志文件系统可以很快地恢复，因为在文件系统的日志中已经将系统更新信息完整地记录下来了。

日志文件系统可以有效地提高文件系统的小文件写性能，但是日志文件系统也有一些缺点：

- 每一次更新和大多数的“日志”操作都需要同步写，这样就需要更多的磁盘 I/O 操作。
- 日志文件系统的文件分配方式比非目志文件系统更容易在文件系统中产生碎片。
- 日志文件系统中对于文件的读性能并不良好，除非安装的磁盘缓冲区能够吸收所有的读请求。但是研究表明，通常磁盘缓存区并不能吸收所有的读请求。

9.3 Virtual FileSystem（VFS）简介

Linux 文件系统功能强大，支持多达十几种不同的文件系统。除了为 Linux 量身定制的 Ext2 文件系统外，Linux 还支持 Minix，FAT，VFAT，NFS、NTFS 等文件系统。在一个操作系统中，为什么可以支持这么多的文件系统呢，这就是 Linux 的虚拟文件系统 VFS 的威力了。

Linux 的文件系统使用了类似 Unix 的文件系统。在文件系统中，最上层是系统的根目录，也就是“ /”（/在 Unix 中代表文件系统的最顶级目录：根目录）。系统根目录下可以是目录也可以是文件，目录里也可以包含目录和文件。如此就形成一个反转过来的树。

不同于 Windows 的文件组织形式，在 Windows 中每个分区对被赋予一个单独的盘符。例如在系统中有两个分区，这两个分区可能是一个叫 C，另一个叫 D。当需要访问 D 这个分区的文件时，只要在命令行中输入“ D:”就可以了。但是在 Linux 里可不是这样。要去读取另一个分区的文件，首先必须 mount 这个分区，例如：

```
mount -t ext2 /dev/hda3 /mnt
```

上述命令将把硬盘第三个分区挂在/mnt 这个目录下。mount 完之后，/mnt 原本的内容会看不到，只会看到 hda3 里的内容。其中/mnt 称为 hda3 的 mount point 而/mnt 这个目录则是被 hda3 所 cover。经过 mount 以后，就可以经由/mnt 去读取 hda3 的内容，就好像 hda3 的内容本来就放在/mnt 底下一样。整个过程，如图 9-4 所示。

如图 9-4(a)所示是原来的文件结构，如图 9-4(b)所示则是 hda3 这个分区的内容，将 hda3 mount 到/mnt 之后，整个文件系统就变成如图 9-4(c)所示的样子了。

不管如何，Linux 会保持其文件系统为一个树的形状。这样 mount 下去，就很容易可以推想到，从根目录开始的这个树很有可能包含好几种文件系统，可能挂在/mnt 上的是 Ext2 文件系统，挂在/home 上的是 FAT，而挂在/cdrom 上的则是 iso9660 文件系统。当用户去读取这些目录里的内容时，他不用去管这个目录挂的文件系统是什么，也不会感到有什么不同。而从程序开发者的观点来看，也不会感觉读/mnt 里的文件和去读/home 里的文件有什么不同。

图 9-4　mount 文件系统的过程

9.3.1　VFS 的体系结构

Linux 的 VFS（Virtual File System，虚拟文件系统）的结构如图 9-5 所示。

图 9-5　Linux 的 VFS 体系结构

Linux 文件系统其实可以分为三个部分，第一部分叫 Virtual File System Switch，简称 VFS。这是 Linux 文件系统对外的接口，任何要使用文件系统的程序都必须经由这层接口来使用它。另外两部分属于文件系统的内部，其中一个是 Cache，另一个就是真正最底层的文件系统，像 Ext2，FAT 之类的东西。

在图 9-5 中，可以清楚地看到当 Kernel 要使用文件系统时，都是经由 VFS 这层接口来使用。在 Linux 中，当系统访问一个文件时，它不会因为这个文件位于不同的文件系统就需要使用不同的方式来读取。因为 VFS 完成了将系统的文件访问请求根据不同文件系统类型进行转换的操作。

当要读取的文件位于 CD-ROM 时，VFS 就自动把这个读取的要求交由 iso9660 文件系统来做，当要读取的文件在 FAT 中时，VFS 则自动调用 FAT 的函数来实现。

在 Linux 文件系统中，为了提高文件访问的效率，都使用了 Cache 机制来提高速度。所谓 Cache 机制就是指文件系统在读写磁盘上的数据之前，都要经过 Cache 系统的缓冲。如果数据在 Cache 里有的话，就直接读取，如果没有，才实际从驱动器读写。除了数据 Cache 之外，Linux 文件系统里还有一个 Directory Cache。在文件系统中，ls 命令的使用

频率是相当大的。每次的 ls 或读写文件其实都要对目录的内容做搜索。因此，如果在目录方面能做个 Cache 的话，那系统的整个速度就会提高。Directory Cache 的功能就在此。Linux 文件系统里还有一个 Cache，叫 inode Cache。故名思义，它是针对 inode 做的 Cache。

9.3.2 文件的表示

从用户的观点来看，可以用文件的绝对路径来代表某个文件而不会出错。在 VFS 中，它并不是用路径来代表文件，而是用一个叫 inode 的东西来代表的。在文件系统里的每一个文件，系统都会给它一个 inode，只要 inode 不一样，就表示这两个文件不相同，如果两个文件的 inode 一样，就表示它们是同一个文件。其实，inode 是由 VFS 定义的，而 VFS 里包含了好几种的文件系统，并不是每一个文件系统都会有 inode 的概念，例如 FAT 文件系统就没有 inode 的概念。但是当 VFS 要求 FAT 去读取某个文件时，事实上它是把那个文件的 inode：传给 FAT 去读。所以，对 VFS 而言，每一个文件都有其对应的 inode，但是 VFS 之下的底层文件系统可以有对应的 inode 结构，也可以没有对应的 inode 结构（但必须将 inode 转换为自己可以理解的结构）。因此，VFS 跟底层文件系统的沟通需要经过相应的特定接口。例如，VFS 要打开一个位于 FAT 的文件时，VFS 会配置一个 inode，并把这个 inode 传给 FAT，FAT 要负责填入一些信息到 inode 中，必要时，也可以在 inode 中加入自己所需要的信息。再如，VFS 要读取 FAT 中的文件内容，VFS 首先把 inde 给 FAT，并且告诉它从文件哪里开始读，读多少个字节。当 FAT 文件系统接收到这些信息后，如何执行将由 FAT 文件系统自己完成，而 VFS 并不关心。

总而言之，VFS 以 inode 为核心数据，将文件的访问信息传递给底层文件系统。底层文件系统根据这些访问信息，依照自己的方式去访问文件，然后将必须的信息填写到 inode 中返回 VFS 即可。

9.3.3 磁盘布局（Disk layout）

如图 9-6 所示为硬盘数据分布示意图。在本节中介绍的 disk 是指硬盘，关于软盘本节暂不讨论。一个硬盘最多可以有 8 个分区（partition），其中 4 个是主分区（primary partition），另 4 个则是扩展分区（extended partition）。所谓分区就是在逻辑上将硬盘空间进行划分。所以，可以把一个硬盘想象成是由最多 8 个分区所组成的。除了分区，硬盘第一个扇区称为 MBR（Master Boot Record，主启动记录），如图 9-7 所示。

图 9-6　硬盘数据分布示意图

图 9-7　硬盘的数据分区示意图

在 Linux 里，文件放的位置是以一个分区为单位的，也就是说一个文件系统是放在分区里，而不能跨分区的。

文件系统在一个分区里中的布局如图 9-8 所示。第一个块是引导块（Boot block），系统引导的时候使用。第二个块是超级块（super block），记录了文件系统重要的信息，接下来的磁盘块存放 inode 和文件数据。

在 VFS 里，每一个文件系统是由其超级块来表示的，这是因为超级块存放了一个文件系统的最重要的信息，通过超级块可以了解这个文件系统的基本构成。从一个文件系统的超级块出发，就可以访问文件系统中任何一个文件。因此，在 Linux 中文件系统的管理以超级块为单位，从超级块可以取得这个文件系统中任何一个文件的 inode，从文件的 inode 则可以对这个文件进行读写访问。

9.4　文件系统设计步骤

在本节将介绍编写一个简单文件系统的基本步骤。在许多操作系统中，文件系统是整个操作系统的核心部分之一。本节的介绍，有助于读者了解如何去实现一个基本的文件系统。

文件系统最主要的功能，直观地说就是名字变换、错误处理、一致性的控制和存储管理（naming，fault-tolerance，concurrency control，and storage management）。此外，文件系统中也经常考虑安全和网络方面的问题。

本节首先实现一个磁盘函数库，可以在一个文件的基础上来模拟对磁盘的操作。这个函数库的接口包括：read、wite、flush disk block。

接下来，将基于这个磁盘函数库来实现一个简单的文件系统，这个简单的文件系统提供了高效的存储管理功能，例如访问文件块非常高效，文件系统的元数据（元数据，metadata，通常是记录文件存取信息的数据，区别于文件内容数据）也非常的小。

最后，将使这个文件系统具有较好的容错性能（robust in the face of failure），例如在系统崩溃时，不会导致数据的丢失。为了测试这个文件系统的容错性，可以在代码中实现任意时候系统崩溃，然后来查看文件系统是否丢失数据。这个文件系统可以在崩溃后恢复到一致性的数据，防止出现文件系统中的不一致现象。

下面开始介绍实现这样一个文件系统的步骤。

9.4.1　磁盘函数库（Disk library）

一个文件系统必须能够永久性地存储数据。永久性的数据存储设备通常是磁盘。在步骤一中，主要是实现一个磁盘函数库，从而来完成对 4KB 磁盘块的读操作和写操作。然后，在这个函数库的基础上，可以实现文件系统。

为了简单，使用一个磁盘文件来模拟一个磁盘。首先，磁盘块需要把一个文件划分

为 4KB 的磁盘块，每一个磁盘块有一个块编号（block number）。使用块编号，可以在这个文件中定位磁盘块的偏移量，例如编号为 10 的磁盘块，在文件中的偏移量为：10 * 4096 = 40960。

磁盘函数库应该尽量模拟真实的硬件接口，并且可以提供良好的调试接口。这些接口为将该磁盘函数库替换成为真正的磁盘驱动器接口提供了方便。磁盘函数库可以有如下函数：

```
int openDisk(char *filename, int nbytes);
int readBlock(int disk,int blocknr, void *block);
int writeBlock(int disk, int blocknr, void *block);
void syncDisk();
```

上述的接口在执行错误的时候，都返回 error。其中，openDisk：打开一个文件进行读或者写，返回一个文件描述符。文件的尺寸是 nbytes。如果文件名不存在，openDisk 将创建一个新的文件。readBlock：读取磁盘块 blocknr 到缓冲区 block 中，磁盘由 disk 指定。WiteBlock：将 block 中的数据写入由 disk 指定的磁盘的 blocknr 磁盘块中；syncDisk：将所有对磁盘的操作写入磁盘中。

9.4.2 文件系统（File system）

在本小节中，将基于 9.4.1 小节实现的磁盘函数库，来实现一个简单的文件系统。在实现一个文件系统时，主要有 3 点需要重点考虑：

- 映射文件名到该文件的磁盘块以及跟踪磁盘上的空闲磁盘块的内存数据结构和磁盘数据结构。
- 磁盘分配布局。
- 文件系统恢复策略。

为了使所设计的文件系统简单一些，这里只考虑文件系统同时有且只有一个用户使用，一个用户可以打开多个文件（单用户 / 单任务操作系统就是这样，例如 DOS）。前面提到的三个重点，可以这样考虑：

- 名字映射（Nameing）：用户易读的名字必须映射成为这个文件所属的磁盘块内容。
- 文件系统组织（Organization）：文件系统是用户组织信息的一种常用的机制（组织大量信息的一种专业机制是使用数据库系统。许多数据库系统也是基于高效文件系统来实现的）。在文件系统的组织中，例如继承关系、树型结构等，都是常用的文件系统组织方法。
- 持久性（Persistence）：数据被包含在文件系统中，文件系统必须保证在任何情况下数据都是完整、可用的，除非数据的所有者要求删除或者更改数据。这要求文件系统可以提供生成、复制、修改、删除一个文件的能力。

文件系统的另外一个需要考虑的问题是安全性。一个文件系统应该向用户提供一种

安全机制，可以仅仅让文件的 read/write/delete/create 得到控制。

此外，文件系统的性能也是一个值得考虑的地方，例如在存放大量小文件的文件系统中、存放大量流媒体的文件系统中，都需要不同的设计来提高应用程序的性能。

在一个磁盘文件系统中，主要通过三类数据完成对文件系统的管理：

- 固磁盘空闲块表（disk block free list）：磁盘空闲块表是一个简单的数据结构（通常情况下可以是一个位图），这个数据结构可以跟踪磁盘上每一个磁盘块的分配使用情况。这个数据结构必须永久性地存放在磁盘上，并且这个存放规则是事先约定好的，这样在文件系统启动时，才能找到这个磁盘空闲块表。
- 索引节点（inodes）：inodes 通常存放在内存中，指向了磁盘上的数据块或者“间接块”（indirect blocks，间接块可以考虑为其他的 inode）。inodes 驻留在内存中，完成文件的惟一表示功能：每一个文件都有一个惟一的 inode 代表它，在 inode 中保存着访问这个文件的所有磁盘块的信息和其他的一些信息（例如文件的属主，文件的权限和创建时间等）。
- 目录（directories）：目录完成的作用是映射文件名（也包括目录名）到这个文件名对应的 inode 序号。

根据上面的讨论，创建、修改、删除一个文件的具体过程如下：

创建一个文件，其步骤如下：

（1）分配和初始化 inode。

（2)将 inode 和文件名的对应关系写入到文件目录表项中(通常在当前的工作目录)。

（3）将上述数据写入磁盘。

修改一个磁盘文件，其步骤如下：

(1) 将此文件的 inode 装载到内存中。

(2) 分配空闲的磁盘块（将这些磁盘块在 free list 中标记为已经分配)。

(3) 修改这个文件的 inode，指向这个新分配的磁盘块。

(4) 将用户的数据写入这个磁盘块。

(5) 将这个文件所有的修改都写入磁盘。

删除一个磁盘文件，其步骤如下：

(1) 将此文件的 inode 装载到内存中。

(2) 依据 inode 数据查找到将要删除的数据所在的磁盘块。

(3) 标记这个磁盘块为空闲。

(4) 将这个文件所有的修改都写入磁盘。

修改一个目录和修改一个文件类似。

注意：在修改文件系统时一定要注意保持文件系统的状态一致。例如，在创建文件系统时，如果在步骤(1)以后，就把所有的修改刷新到磁盘中，而在完成第（2）步之前发生了系统崩溃，那么将导致一个磁盘块在磁盘块分配表中已经分配了，但是没有任何一个文件使用了这个磁盘块，这个磁盘块的状态就是“丢失”的。假如不把文件系统的修改刷新到磁盘上，那么如果发生了系统崩溃，将导致新创建文件的丢失。在文件修改、

删除时，也存在类似的情况。在设计文件系统时，必须仔细地考虑这些情况。

这里讨论的文件系统，还没有涉及如何从灾难中恢复的问题，这里主要保证的是文件系统必须维护数据的一致性。通常，文件系统必须能够运行，能够停止，能够再次运行。在这些过程中，写入文件的数据不会丢失，也不会在文件系统中出现用户不需要的“垃圾文件”。

除了考虑文件系统的数据正确性外，另外一个重要的问题是提高文件系统的存储效率，也就是在文件系统中存储一个数据字节所必须花费的字节数（这个数值越接近 1.0 越，这要求文件系统的元数据越小越好）。

在文件系统中，为了提高系统性能，通常要使用文件系统的 Cache。这部分内容是属于文件系统的高级功能，可以参考 Linux 文件系统实现来考虑。

9.4.3 容错性（Robustness）

在操作系统中，文件系统具有一个显著的不同是：文件系统必须保持数据的一致性，也就是说任何对数据的修改，都必须最终有效地保存下来（通常是存入到磁盘上）。作为对比，在操作系统中的其他方面，例如主存、寄存器、TLB 都不需要考虑数据的一致性。

在文件系统中，对文件的操作是分成许多步骤的，在这些步骤中和步骤之间都可能发生系统崩溃的现象，而这些都会导致文件系统的数据不一致。例如，当文件系统把一个数据写入磁盘的时候，会有如下的步骤：在磁盘上分配新的磁盘空间，修改文件的 inode，修改磁盘的空闲磁盘块表，最后将数据写入磁盘。一个文件系统必须保证在上述的步骤中任何地方发生系统崩溃后都可以保证系统数据的一致性，例如不丢失磁盘块和 inode 节点等。为了完成这些保证，在文件系统中需要提供软件的原子操作（例如磁盘写事务处理）等方式。

在完成写入磁盘的这些操作肘，可以考虑把一些需要多个步骤才能完成的操作修改成为可以原子性一次完成的操作，从而保证写操作的完整性。或者，文件系统可以提供一种机制，对失去一致性的磁盘系统进行修复（DOS 系统中的 scandisk 就完成这个功能，Unix 系统中的 fsck 也完成这样的功能）。

9.5 自己编写的简单文件系统

本节将介绍一个自己编写的文件系统。这个简单的文件系统实现了最简单的文件系统功能，是使用一个磁盘文件来模拟磁盘的。读者可以从这个模拟文件系统中获得实现文件系统的最基本的认识。其代码如下：

```
/* fs.c */
#include      <stdio.h>
#include      <conio.h>
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#define         FILE_NUM              10
#define         FILE_SIZE             (1024*10)
#define         PUT_PROMPT            printf("FS#")

const           char        file_system_name[] = "fs.dat";

FILE        *fp;

struct inode{
        char file_name[512];
        int file_length;
};

struct    inode      *p;

struct inode inode_array[FILE_NUM];

void creat_file_system(){
        long len;
        int inode_num;
        int i;

        fp=fopen(file_system_name,"wb");
        if(fp==NULL){
                printf("Create file error!\n");
                exit(1);
        }

        for(len=0;len< (sizeof(inode_array[0])+FILE_SIZE)*FILE_NUM;len++){
                fputc(0,fp);
        }

        for(i=0;i<FILE_NUM;i++){
                strcpy(inode_array[i].file_name,"");
                inode_array[i].file_length =0;
                p=&inode_array[i];
                fwrite(p,sizeof(inode_array[0]),1,fp);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    }

    fflush(fp);
}

void open_file_system(){
    int i;

    fp=fopen(file_system_name,"r");
    if(fp==NULL){
        creat_file_system();
    }

    fp=fopen(file_system_name,"r+");
    if(fp==NULL){
        printf("Open file to read/write error!\n");
        exit(1);
    }

    p = &inode_array[0];
    fseek(fp,0,SEEK_SET);
    fread(p,sizeof(inode_array[0])*FILE_NUM,1,fp);
}


int new_a_file(char *file_name){
    int i;
    for(i=0;i<FILE_NUM;i++){
        if(strcmp(inode_array[i].file_name,"")==0){
            strcpy(inode_array[i].file_name,file_name);
            p = &inode_array[i];
            fseek(fp,sizeof(inode_array[0])*i,SEEK_SET);
            if(fwrite(p,sizeof(inode_array[0]),1,fp)!=1){
                printf("new a file error!\n");
                exit(1);
            }
            fflush(fp);
            return i;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            }
        };
        return -1;
}

int del_a_file(char *file_name){
        int i;
        for(i=0;i<FILE_NUM;i++){
                if(strcmp(inode_array[i].file_name,file_name)==0){
                        strcpy(inode_array[i].file_name,"");
                        p = &inode_array[i];
                        fseek(fp,sizeof(inode_array[0])*i,SEEK_SET);
                        fwrite(p,sizeof(inode_array[0]),1,fp);
                        fflush(fp);
                        return i;
                }
        };
        return -1;
}

void list(){
        int i;
        int count;

        printf("\n");
        count=0;
        for(i=0;i<FILE_NUM;i++){
                if(strcmp(inode_array[i].file_name,"")!=0){
                        printf("\tFile name: %s \t\t\t [%d]\n",inode_array[i].file_name,
                                inode_array[i].file_length);
                        count++;
                }
        };
        printf("\tFiles count = %d\n",count);
}

int open_a_file(char *file_name){
        int i;
```

```
        for(i=0;i<FILE_NUM;i++){
                if(strcmp(inode_array[i].file_name,file_name)==0){
                        return i;
                }
        };
}

int offset_by_i(int i){
        return sizeof(inode_array[0])*FILE_NUM + FILE_SIZE*i;
}

int write(char *file_name,int offset,char *str,int count){
        int handle;

        handle = open_a_file(file_name);
        fseek(fp,offset_by_i(handle)+offset,SEEK_SET);
        fwrite(str,count,1,fp);

        inode_array[handle].file_length = strlen(str)+offset;
        p = &inode_array[handle];
        fseek(fp,sizeof(inode_array[0])*handle,SEEK_SET);
        fwrite(p,sizeof(inode_array[0]),1,fp);
        fflush(fp);

}

int read(char *file_name,int offset,int count,char *str){
        int handle;
        char buf[FILE_SIZE];

        handle = open_a_file(file_name);
        fseek(fp,offset_by_i(handle)+offset,SEEK_SET);
        fread(buf,count,1,fp);
        strncpy(str,buf,count);
}

void print_help()
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
{
    printf("Please select: 1. Creat a new file system\n");
    printf("                2. Make a new file\n");
    printf("                3. Del a file\n");
    printf("                4. List files\n");
    printf("                5. Write a string to a file\n");
    printf("                6. Read a string from a file\n");
    printf("                7. Exit\n");
    printf("                                                \n");
    printf("                h for help\n");
}

int main()
{
    char buf1[FILE_SIZE];
    char key;
    char buf2[5120];

    clrscr();

    print_help();

    key = '0';
    open_file_system();
    while(key!='7')
    {
        PUT_PROMPT;
        key=getch();
        putch(key);
        strcpy(buf1,"");
        strcpy(buf2,"");
        switch(key)
        {
        case '1':
            fclose(fp);
            creat_file_system();
            printf("\nCreate file system succeed!\n");
            open_file_system();
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        break;
    case '2':
        puts("\nPlease input a file name:");
        scanf("%s",buf1);
        new_a_file(buf1);
        printf("Add a file succeed!\n");
        break;
    case '3':
        puts("\nPlease input a file name:");
        scanf("%s",buf1);
        del_a_file(buf1);
        printf("Del file %s succeed!\n",buf1);
        break;
    case '4':
        list();
        break;
    case '5':
        puts("\nPlease input a file name:");
        scanf("%s",buf1);
        puts("\nPlease input a string:");
        scanf("%s",buf2);
        write(buf1,0,buf2,strlen(buf2)+1);
        printf("\nWrite a file succeed!\n");
        break;
    case '6':
        puts("\nPlease input a file name:");
        scanf("%s",buf2);
        read(buf2,0,FILE_SIZE,buf1);
        puts(buf1);
        break;
    case 'h':
        printf("\n\n");
        print_help();
        break;
    case '7':
        break;
    default:
        printf("\n");
```

```
            }

        }
        return 0;
}
```

上述程序的运行效果如图 9-8 所示。

图 9-8　模拟文件系统的运行效果

9.6　Linux 0.01 文件系统源代码分析

Linux 0.01 的文件系统是在 Minix 系统上开发的，因此 Linux 0.01 的文件系统与 Minix 的文件非常相像。由于 Minix 的文件系统和 Unux 的文件是同出一辙的，因此 Linux 0.01 的文件系统也就与 Unix 的文件系统有了不可分离的渊源。

在系统启动时，只执行一次的函数 sys_setup 读取所有的 IDE 硬盘上的分区表。在 kernel/hd.c 源程序中包含对各种 IDE 接口的驱动代码。此外 rw_hd()函数完成了从各种 IDE 设备上读取和写入数据块的功能。

如表 9-1 所示是 Linux 0.01 中主要文件系统源代码说明。在 Linux 0.01 的文件系统中 hd.c()代码演示了从 IDE 设备上读取和写入数据块，而 fs/block_dev.c 中包含了对块设备文件系统的具体支持。

表 9-1　Linux 0.01 中主要文件系统源代码说明

系统源代码	说明
kernel/hd.c	处理 IDE 中断和数据写入、读出队列
fs/blockdev.c	使用 hd.c 中提供的功能，例如 block_write()来向硬件驱动器中写入或者读出数据

fs/read_write.c 中包含了 sys_read()，sys_write()，和 sys_lseek()的代码。

Linux 0.01 的文件系统有一些局限性。Linux 0.01 文件系统的启动过程中必须使用 BIOS，这一点对于现在出现的非常大的 IDE 硬盘是不适应的。

Linux 0.01 文件系统的源代码较多，主要包括了 super.c、read_write.c、open.c、inode.c、buffer.c、bitmap.c 等六个文件。本节将简单介绍一下这几个文件的主要功能。所有这些源文件的详细注释在本书的光盘上都可以找到。

super.c、read_write.c、open.c、inode.c、buffer.c、bitmap.c 等六个文件的主要功能如下：

- super.c 文件中包含了 Linux 0.01 的文件系统处理超级块表（super-block tables）

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

的功能。

- read_write.c 文件包含了 Linux 0.01 文件系统中对字符设备和块设备的读操作和写操作的实现过程。
- open.c 包含了 Linux 0.01 中打开文件的操作过程。
- inode.c 包含了 Linux 0.01 中对文件系统 inode 表的维护过程
- buffer.c 实现了文件系统中的缓冲功能。在 Linux 0.01，为了避免资源竞争，采用了在中断中禁止改变 buffer 内容的办法，取而代之的是让调用者来改变它。
- bitmap.c 包含了处理文件系统中 inode 和文件系统中位图的功能。

9.7 本章小结

本章重点介绍了 Linux 的文件系统设计。Linux 的文件系统是一个设计良好，性能优秀的文件系统。

本章首先介绍了文件系统的基本知识，包括硬盘驱动器的基本原理.然后，介绍了 Unix 文件系统的基本设计思想和实现原理，包括 Unix 文件系统的布局等。最后，对 Unix 文件进行了深入的分析和介绍，特别分析了 Unix 文件系统的设计好坏对性能的影响。

在以上基础上本章还详细介绍了设计和实现一个模拟文件系统的主要步骤，包括实现磁盘驱动库、实现文件系统调用、实现文件系统的容错性等。并且在这些主要步骤的基础上，实现了一个模拟文件系统，读者可以通过对这个模拟文件系统的源代码的学习，迅速地掌握一个文件系统的概貌。，

同时，本章还详细介绍了 Linux 的虚拟文件系统 VFS 的设计思想。通过使用 VFS，Linux 实现了上层文件系统和下层具体物理文件系统的无关性，从而大大地提高了 Linux 文件系统的灵活性和适应性。最后，本章对 Linux 0.01 文件系统的源代码进行了详细的分析。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

第10章　shell 编程技术和实例

本章知识点：

- shell 的基本概念
- 实例：最简单的 shell 程序
- 管道和 I/O 重定向
- shell 实现代码分析

shell 是 Linux 系统提供给用户的最重要的交互界面之一。Linux 的 shell 继承了 Unix 系统 shell 的强大和灵活的功能，是用户使用系统功能的强大工具。

shell 严格地说应该是 Linux 系统的一种应用程序，而不属于操作系统核心的组成部分。通常一个操作系统都应该向用户提供类似 shell 的用户界面，因此本章介绍 Linux 中 shell 的实现原理和方法。

要实现 shell 系统，首先需要了解系统 I/O 重定向。本章通过精简实用的例子，介绍了如何通过使用 dup()和 pipe()系统调用来实现 I/O 重定向，为实现一个功能强大的 shell 打下基础。

本章将介绍一个类似 Linux 系统 shell 的试验 t-shell（test shell）。t-shell 支持多个命令、支持管道操作、支持命令行参数，具有了一个 shell 的最主要功能。

本章通过详细地介绍 t-sheH 的实现原理，向读者展示了在 Linux 系统中实现一个强大的 shell 的内部细节，可以帮助读者快速地实现自己设计 shell。

10.1　shell 的基本概念

在 Linux 中，shell 功能是系统的重要组成部分，但是 shell 功能不是包含在操作系统核心中的，而是一个应用程序。shell 是 Linux 用户和系统进行交互的最主要接口之一，一个优秀的 shell，可以大大提高用户使用系统的效率。

在 Linux 0.01 版本中还没有提供 shell 功能。因此，本章将介绍一个简单 shell 的实现原理。这个 shell 和 Linux 使用的 C shell 类似，具有所有 shell 的基本功能，但是实现比较简单。这个 shell 主要是为了演示如何实现一个 shell，而不是去实现一个完整的 shell。在本章中把这个实验的 shell 称为 t-shell（test shell）。

t-shell 可以在 DOS、Linux 下使用 GNU 编译器编译，例如在 DOS 下可以使用 DJGPP，在 Linux 下面可以使用 GCC。t-shell 程序的大部分代码可以很好地移植到读者自己编写的操作系统中。本节介绍的 t-shell 程序考虑了小操作系统的限制性，因此仅仅使用了很

PDF-XCHANGE Click to buy NOW! www.docu-track.com

少的内存，同时实现也比较简单。这样，读者可以从 t-shell 程序中快速地掌握编写 shell 的原理和技术，而不会陷于过多的复杂细节中。

执行本节介绍的 shell 程序，可以执行如下的命令：

sh [-e]

-e 参数指定“\”不用做转义字符。

当 shell 启动时，会读取默认的配置文件/profile。

当执行 shell 时，用户键入一个命令，shell 首先除去所有的“\”字符之前的字符（如果没有使用-e 参数）和所有的引号中的字符。然后，shell 按照如下的规则处理“!”引起的 shell 命令历史替代：

!!代表前一个执行的命令；

!<num>代表前<num>个执行的命令；

!<str>代表<str>字符串开始的命令。

然后，shell 会将输入的命令字符串切换成为令牌（token）字符串。在引号中的所有字符串成为一个令牌。空格是隔开各个令牌的分割符。

然后，shell 使用如下的规则处理命令行的通配符：

包含通配符（当前设定为*）的令牌被假定成文件名，通配符被扩展成为匹配的文件名（*匹配任何字符）。

shell 执行 I/O 重定向（I/O redirections）：

'>word'　重定向标准输出（redirect stdout）

'>>word' 重定向标准输出，添加在指定的文件后

'>&word' 重定向标准输出和标准错误输出（redirect stdout and stderr）

'<word'　重定向标准输入（redirect stdin）

shell 执行如下的操作：

shell 跟踪着所有执行路径中的命令程序（外部命令）和所有的内部命令（built-in commands）。如果命令的第一个参数是路径名，那么在这个路径中的这个命令将被执行。如果第一个参数不是路径名，那么 shell 将检查 PATH 路径指定的目录中的命令程序，然后再检查内部命令。如果希望修改内部命令表，可以使用“rehash'’内部命令。

如果找不到要执行的命令程序，这个命令将触发 DOS，使用 DOS 的内部命令来尝试是否可执行，例如：

dir ' *，c ' c 叩 y ' b：*，* '

shell 的内部命令有：

ls[一 lsRa]perfbn 瑙 adirectory1is 曲 g，Il gt ‘ venn。axgumetlts，

lsl；slsthecLtrrentdirectory，Theflagsar 巳：

I11。ngIisting，s～summary，R—recursive

direc：tory1isting，a～at 七 ribute（dire~：t0 叫 executable）cd<曲>changethectlrrent、aorMngdirectorytotheindi{： arealdirectory230 · Linux0， 01 内核分析与操作系统设计 pushd<dir>cdtotheindicateddirectory，pushingthecurrent

directoryonthestacI【，Involdngthis“山 noarguments

cause~thecurrentdirectoryendthetopofthestACKto

beinterchanged， AlsoImownaspd， popd<dir>popthetopdementoffthedirectorystACK，andcd

foit ，dirsprintthecurrentdirectorystACKmkdir<dir>[…]createdirectorieswiththeindicatednamesHndir<dir>[· --]ren20vetheindicateddirectoriessetvai-~val[…]settheenvironmentvariablevattothegivenvaluetakecolllnlandsfromthegivenfile

fgrepstl"file[···]lookforthegivenstringwithinthegivenfile（s），

DOS 限制命令行参数限定不超过 128B。这个限制使得在使用命令行时经常出现通配符超过限制的情况，因此在命令行中设计了 fgrep 命令。

本节介绍的 t-shell 实现的命令历史功能非常有限，过去的命令在使用时不能被编辑。

在 t-shell 中，惟一的通配符是“ *”，但是“ *”的使用也是有一定限制的。“ *”仅仅能匹配最终的目标文件或者目录，/*/*或者/*/xxx 的模式无法正常工作。

ls 命令和通配符使用 qsort 程序来处理参数，而 qsort 程序需要很大的一个堆栈。如果处理非常大的目录，可能会出现堆栈溢出的问题。例如，一个目录中有 85 个文件，这时如果堆栈是 2KB，那么可以会出错，而当堆栈是 4KB 时，就不会有问题。

10.2 实例：最简单的 shell 程序

下面将介绍一个最简单的 shell 程序，这个 shell 程序演示了从用户输入行输入命令，然后执行命令，输出结果的过程。对于普通 Linux shell 中的高级功能，这个 shell 都不支持。读者可以快速地从这个最简单的 shell 程序中获得对如何实现一个 shell 的程序有个直观感受。其代码如下：

```
/* sh1.c */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <string.h>
#include <errno.h>

#define SHELL_NAME "sh1"
#define PROMPT_ENVIRONMENT_VARIABLE "PROMPT"

char *prompt;

int main(int argc, char **argv)
{
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
char cmd[80];
int statval;

/* Determine prompt value. */
if ((prompt = getenv(PROMPT_ENVIRONMENT_VARIABLE)) == NULL)
        prompt = SHELL_NAME ":";

/* Process commands until exit, or death by signal. */
while (1)
{
        /* Prompt and read a command. */
        printf(prompt);
        gets(cmd);

        /* Process built-in commands. */
        if(strcasecmp(cmd, "exit") == 0)
                break;

        /* Process non-built-in commands. */
        if(fork() == 0) {
                execlp(cmd, cmd, NULL);
                fprintf(stderr, "%s: Exec %s failed: %s\n", argv[0],
                        cmd, strerror(errno));
                exit(1);
        }

        wait(&statval);
        if(WIFEXITED(statval))
        {
                if(WEXITSTATUS(statval))
                {
                        fprintf(stderr,
                                "%s: child exited with status %d.\n",
                                argv[0], WEXITSTATUS(statval));
                }
        } else {
                fprintf(stderr, "%s: child died unexpectedly.\n",
                        argv[0]);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            }
        }
    }
```

上述程序的运行效果如图 10-1 所示，这个简单的 shelI 程序的缺点是还不能处理参数。例如下面执行的 uname -a 就不能正确执行。

图 10-1　shell 运行效果（不支持命令参数）

下面的 shell 程序针对图 10-1 所示程序不能处理参数的缺点进行了改进，可以支持输入命令的参数了，具体代码如下：

```
/* sh2.c */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <errno.h>

char *parse_cmd(char *cmd, char **argarr, int *narg) {
    enum states { S_START, S_IN_TOKEN, S_IN_QUOTES, S_SEMI };
    int argc=0; // Arg count
    int loop=1; // Loop control
    enum states state = S_START; // Current state.
    int lastch; // Last character encountered.

    while (loop) {
        switch (state) {
        case S_START:
            if (*cmd == '"') {
                *argarr++ = cmd + 1;
                ++argc;
                state = S_IN_QUOTES;
            }
            else if (*cmd == 0 || *cmd == ';')
                loop = 0;
            else if (*cmd <= ' ')
                *cmd = 0;
            else {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                    *argarr++ = cmd;
                    ++argc;
                    state = S_IN_TOKEN;
                }
                break;
            case S_IN_TOKEN:
                if (*cmd == 0 || *cmd == ';')
                    loop = 0;
                else if (*cmd <= ' ') {
                    *cmd=0;
                    state=S_START;
                }
                break;
            case S_IN_QUOTES:
                if (*cmd == 0)
                    loop = 0;
                else if (*cmd == '"') {
                    *cmd = 0;
                    state = S_START;
                }
            }
            cmd++;
        }
        *argarr = NULL;    /* store the NULL pointer */

        /* Return argument count, if needed. */
        if(narg != NULL) *narg = argc;

        lastch = cmd[-1];
        cmd[-1] = 0;
        return lastch == ';' ? cmd : NULL;
}

int main(int argc, char **argv) {
        char cmd[80]; // Input command area.
        char *source = NULL; // Where commands to send to parser come from.
        char *arg[21]; // Arg list.
        int statval; // Exec status value.
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
char *prompt="baby:"; // Command prompt, with default.
char *test_env; // getenv return value.
int numargs; // parse_cmd return value.

if (test_env=getenv("BSPROMPT"))
        prompt=test_env;
while (1) {
        // See if we need to get a line of input.
        if(source == NULL) {
                printf(prompt);
                source = gets(cmd);
                if(source == NULL) exit(0);    // gets rtns NULL at EOF.
        }

        // Get the next command.
        source = parse_cmd(source, arg, &numargs);
        if (numargs == 0) continue;

        // Exit command
        if (!strcmp(arg[0],"exit")) {
                if (numargs==1)
                        exit(0);
                if (numargs==2) {
                        if (sscanf(arg[1],"%d",&statval)!=1) {
                                fprintf(stderr,"%s: exit requires an "
                                        "integer status code\n",
                                        argv[0]);
                                continue;
                        }
                        exit(statval);
                }
                fprintf(stderr,"%s: exit takes one "
                        "optional parameter -integer status\n",
                        argv[0]);
                continue;
        }

        // Run it.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            if (fork()==0) {
                execvp(arg[0],arg);
                fprintf(stderr,"%s: EXEC of %s failed: %s\n",
                        argv[0],arg[0],strerror(errno));
                exit(1);
            }
            wait(&statval);
            if (WIFEXITED(statval)) {
                if (WEXITSTATUS(statval))
                    fprintf(stderr,"%s: child exited with "
                            "status %d\n", argv[0],
                            WEXITSTATUS(statval));
            } else {
                fprintf(stderr,"%s: child died unexpectedly\n",
                        argv[0]);
            }
        }
    }
```

上述程序的运行效果如图 10-2 所示，从图中可以看到，这个 shell 程序已经可以支持在命令中使用参数了，例如 uname -a 可以正确地执行了。

图 10-2　shell 运行效果（支持命令参数）

10.3　管道和 I/O 重定向

在 shell 中，一个非常重要的工作就是使用管道来处理 I/O 重定向。为了完成这些工作，需要使用 dup()、pipe()等系统调用，此外还有 open()、close()、fork()和 exit()等系统调用。shell 处理的文件描述符一般有三个：文件句柄 0（stdin，标准输入）、文件句柄 1（stdout，标准输出）、文件句柄 2（stderr，标准错误输出）。

10.3.1　使用 dup()重定向 I/O

dup()系统调用的功能是返回一个新的文件描述符，它指向传入参数文件描述符指向的文件流。dup()使用进程文件分配表中的第一个没有使用的文件句柄复制为新的文件句柄。dup()经常使用的一个功能就是完成标准 I/O 的重定向。使用 dup()进行 I/O 重定向的代码片段如下：

intfd，pid; Linux0，01 内核分析与操作系统设计 fd=open（path，mode）; if（（pid=fork()）==0）I

dose（O）；|*Closestdin，*|

/*

*Maimstdincomefromfd，Dup()willreusethelowmt

*，numberunusedfiledescriptor，0inthiscase，

*|

dup（fd）；

dose（fd）；/*Nolongerneeded，*/

execve（…）；/*Runcommand，withinputfromfd，*/1risel

l*Parent，Waitforchildpmceaseatoexit，*1

while（wait（NULL）>0）；

10.3.2 使用 pipe()和 dup()

pipe()函数可以创建一个 FIFO 流（First In First Out stream，先进先出流）来完成进程间的通信。pipe()系统调用生成两个文件句柄，一个可以向 FIFO 流中写入，一个可以从 FIFO 流中读取。在 Unix/Linux 中，通过 fork()系统调用生成的子进程，完全继承了父进程的文件句柄。因此，可以通过在父进程中使用 pipe()函数来创建管道，然后再生成 fork()子进程，每一个生成的子进程中都有父进程调用 pipe()函数生成的文件句柄的复制。在子进程中，可以把父进程中通过 pipe()生成的句柄使用 dup()函数来重定向子进程的标准输入（stdin）、标准输出（stdout）和标准错误输出（stderr），这样父进程就可以控制子进程的标准 I/O 了。使用 pipe()重新设置子进程的标准 I/O 句柄的源代码如下：

inti，pid，mypipe[2】；/*Twofds，forpipeends，*/

pipe（mypipe）；/*Createpipe，*/ /*Createreader，*/if（（pid=fork()）==0）l

dose（0）；/*clmestdin，：*/

dup（pipe[O]）；/*Stdinmmeafrompipe，*/

close（pipe[0b；|*Nolongerneeded，*|

close（pipe[1]）；/*Notneeded，*/

execve（…）；/*Runcommand，withinputfrompipe，*，『*Createwriter，*/if（（pid=fork()）==0）r

close（1）；/*Closestout，*/

dup（pipe[1]）；/*Stdoutgoestopipe，*/

close（pipe[1]）；/*Nolongerneeded，*/第 10 章 sheR 编程技术和实例，237，d~e（pipe[O]）；『*Notneeded，*/Xlle（'·'）；『*Runcommand，withoutputtop|pe，*fdose（pipe[0]）；elme（pipe[x]）}。while（wait（NULL）>0）；/*Waitforchadprocessestoexit，*/。。。

10.3.3 使用 dup2()的例子

dup2()是增强的 dup()，使用 dup2()可以复制一个指定的文件句柄，包括标准输入、输出和错误句柄。下面这个例子，是使用 dup2()在一个进程中重定向 I/O 的例子，其代码如下：

/*dup，c*/‘#indude<Stdlo，h>#include<stdlib，h>#inelude<unistd，h>#include<sys/types，h>#indude<sy 封 stat，h>#include<fcntl，h>voidDisplayOutput()fpfinff（stdout，“Thisistostdout＼一’）fpfinff（stderr，’|Thisistostderr＼//）试 tmain（intargo，char**argv）

int—newStdOut，一 newStdErr；

//calltheoutputroutine

D 酬 ayOutput()；

//changethestdoutandstderrfiledescriptors

—newStdOut=open（”Out”，O—CREATfO—TRUNC}O—RDWR，S—IRUSRIS—IWlJSR）；

一 newStdErr：open（”err'’0 一 CREAT『0 一 TRUNC10 一 RDWR，S—IRUSRfS 一啪 sR）：，

dup7（一 newStdOut，1）；

dup7（一 newStdErr，2）；

//callItheoutputroutineagain

DIsplayOutput()；238-Linux0，Ol 内核分析与操作系统设计 return0

上面的例子，执行的结果如图 10-3 所示。

图 10-3　进程中使用 dup2()的例子

下面这个例子，是使用 dup2()和 pipe()在一个进程中重定向 I/O 的例子，其代码如下：

```
/* forkdup.c */
/* This program executes /bin/cat with stdin coming from f1, and stdout
going to f2.    This is all done with fork, exec, and dup: */

#include <stdio.h>
#include <fcntl.h>

main(int argc, char **argv, char **envp)
{
        int fd1, fd2;
        int dummy;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        char *newargv[2];

        if (fork() == 0) {
                fd1 = open("f1", O_RDONLY);
                if (fd1 < 0) {
                        perror("catf1f2: f1");
                        exit(1);
                }

                if (dup2(fd1, 0) != 0) {
                        perror("catf1f2: dup2(f1, 0)");
                        exit(1);
                }
                close(fd1);

                fd2 = open("f2", O_WRONLY | O_TRUNC | O_CREAT, 0644);
                if (fd2 < 0) {
                        perror("catf1f2: f2");
                        exit(2);
                }

                if (dup2(fd2, 1) != 1) {
                        perror("catf1f2: dup2(f2, 1)");
                        exit(1);
                }
                close(fd2);

                newargv[0] = "cat";
                newargv[1] = (char *) 0;

                execve("/bin/cat", newargv, envp);
                perror("execve(bin/cat, newargv, envp)");
                exit(1);
        } else {
                wait(&dummy);
        }
}
```

上述的程序，使用 dup2()和 pipe()，其运行结果如图 10-4 所示。

图 10-4　进程中使用 dup2()和 pipe()的例子

10.4　t-shell 实现代码分析

t-shell 是一个试验性的 shell 程序，具有 shell 系统的所有基本功能。通过对 t-shell 的学习，读者可以了解到实现一个 shell 系统的基本技术。本节将详细地介绍 t-shell 的实现原理。

10.4.1　shell 总体结构（不支持管道）

shell 中的管道操作是指前一个命令的输出，可以作为后一个命令的输入。支持管道操作的 shell 比不支持管道操作的 shell 要略微复杂一些，本小节将首先介绍不支持管道操作的 shell 程序的总体结构。

一个不支持管道操作的 shell 程序的总体结构如下：

```
while（!EOF）{
printprompt；
readllneofinput；
Darseitint0 ” crild" ” argl…ar // ’·· … ” ~rgn’’；if（（pid=fork()）==0）{ / *处理 I/O 重定向 * / if （redirection ） dotheopen / clmeldup / closethingap 廿 [0]= ” Ⅱ T1naptr[1~="argl”；… ” ap Ⅱ [n]= “ argn"；aptr[n+1]。NULL “ ecve（ ” cmd"，aptr，eptx）；/ *搜索执行路径* / ’ for（eachpathcomponent）e）【ecve（“ component / emd’’，aptr，eptr）/ *命令执行失败* / print，，cDld：C[而 mandnotfoUna，cofitinue；if（!bACKgrounad）jwait()； / *前台执行*』else 研 ntpId， / i 后台执行* /
```

10.4.2　shell 总体结构（支持管道）

一个成熟的 shell 应该支持管道操作。在 Linux 中，通过在 shell 中使用管道操作，可以把一系列的简单的 shell 命令组合起来实现强大的功能，这是 Linux/Unix 系统的一个显著的特性，也体现了 Unix 系统的“简单就是美”的思想。

一个支持管道操作的 shall 总体结构如下：

```
j（crnd]L。d2） / 。crndl 的标准输出为 cmd2 的标准输入* / pipe[fdarr]； / *定义管道* / 第 10 章 shell 编程技术和实例，241，if（（pidl=fork()）==0）{close（1）； / *关闭 emdl 的 stdout* / dup（fdarr[1]）； / *设置 fdarr[1]~cmdl 的 stdout* / close（fdarr[1]）；close
```

（fdarr[O]）；aptr[0]= ”cmdl”；aptr[1]=NULLe~ecve（ ”cmdl”，aptr，ep 廿）if（（pid2=fork()）==O）fclose（O）； ／*关闭 emd2 的 stdin*／dup（fdarr[0]）， ／*设置 fda 丌[O]为 cmd2 的 stdin*／close（fdarr[0]）_lclose（fdarr[1]）；i，一 apt~[0]= ‘矗泄 laptr[1]=NULL；。~xecve（ ”cmd2”，印廿，eptr）；close（fdarr[0]）；close（fdarr[1]）；wait()forbothpldlandpid2

10.4.3 main()函数

shell 是一种典型的命令解释程序。这些程序的基本结构都是一个循环程序，在循环程序中反复进行读取输入信息，然后对输入信息进行处理。

t-shell 程序的主程序组成如下：

```
#include "def.h"

main(void)
{
    int i;

    initcold();

    for (;;)
    {
        initwarm();

        if (getline())
            if (i = parse())
                execute(i);
    }
}
```

在上述 t-shell 的 main()主程序中，首先调用一个初始化函数 initcold()来完成一些初始化工作，完成初始化工作以后，t-shell 进入一个永久循环，进行如下的工作：重复地打印提示信息，读取用户的输入行，分析输入行获得用户输入的命令和参数，然后执行每一个命令，在执行命令的过程中需要正确地处理 I/O 重定向和管道。

具体地说，在 t-shell 的循环中，首先调用 initwarm()来初始化用户输入命令行，然后显示 tsh:提示符。接下来调用 getline()，复制用户的输入行到数组 line[]中，并且返回用户输入的命令行的长度（用户输入的命令行的长度是有限制的，不能为 0，也不能大于 MAXLINE）。

假如 getline()返回一个非 0 值，然后 t-shell 调用 parse()函数对命令行进行分析。parse()函数分析 line[]数组中的内容，或者和命令处理相关的信息。这些信息包括：命令的名字

和每个命令的参数，I/O 重定向的文件名和后台处理操作符（&）。

如果命令行处理没有产生错误，parse()函数返回一个正数，这个数值代表了用户输入命令行中的命令数目。如果 parse()函数分析命令行遇到了错误，那么将返回 0。如果 parse()函数没有遇到错误，那么 shall 将会调用 execute()函数，execute()函数的输入参数就是 parse()函数的返回值。execute()函数使用 fork()和 exec()来执行用户输入的每个命令，在需要的时候打开管道，执行 I/O 重定向。

10.4.4 initialization()函数

在本节介绍的 t-shell 程序中，没有真正的内部命令（built-in commands）。在一般的 shell 中，可以通过执行 exit 来终止 shell 程序的执行，而本节介绍的 t-shell 程序不需要这样。取而代之的方法可以是发送程序终止信号给 shell，例如在键盘上按下 Ctrl+C 或 Ctrl+\，其代码如下：

```
initcold(void)
{
    /*
        signal(SIGINT, SIG_IGN);
        signal(SIGQUIT, SIG_IGN);
    */
}


initwarm(void)
{
    int i;

    backgnd = FALSE;
    lineptr = line;
    avptr = avline;
    infile[0] = '\0';
    outfile[0] = '\0';
    append = FALSE;

    for (i = 0; i<PIPELINE; ++i)
    {
        cmdlin[i].infd = 0;
        cmdlin[i].outfd = 1;
    }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    for (i = 3; i<OPEN_MAX; ++i)
        close(i);

    printf("tsh: ");
    fflush(stdout);
}
```

initwarm()函数在每次 t-shell 循环时最先开始执行。initwarm()函数实际上初始化全局变量，设置 cmdlin[]数组中的每个命令的 infd 和 outfd 文件描述符为默认值。close()函数处理所有的标准输入，标准输出，标准错误文件句柄，最后显示 tsh:提示符。

10.4.5 getline()函数

getline()函数从标准输入上复制字符到数组 line[]中，直到用户输入换行符“\n”为止，或者输入的字符数达到 MAXLINE 个（这时 line[]数组存放字符满了）。

如果用户输入了 MAXLINE 个字符，那么 getline()函数将返回 ERROR，不然 getline()函数返回 OKAY，并且在用户输入的字符串的末尾添加字符串的结束标志“\0”，具体源代码如下：

```
getline(void)
{
    int i;

    for (i = 0; (line[i] = getchar())!='\n' && i<MAXLINE; ++i);

    if (i==MAXLINE)
    {
        fprintf(stderr, "Command line too long\n");
        return(ERROR);
    }

    line[i+1] = '\0';
    return(OKAY);
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

10.4.6 parse()函数

t-shell 命令行使用的语法如下：

cmd [<filename] [|cmd]……[or filename] [&]

cmd 代表一个 t-shell 的命令及其参数，or 是输出重定向符（> or >>）。方括号代表可选参数，省略号代表前面的内容可以重复 0～n 次。

从上面可以看到，一个合法的 t-sheH 命令开始于一个命令，然后是一个可选的输入重定向参数和输入重定向文件名，然后，可以接一个管道操作符和另外一个命令。管道操作符和命令可以重复多次。在整个命令行的最后可以是一个可选的输出重定向符和输出重定向文件名。最后，整个的命令行可以加上一个可选的是否在后台运行的操作符“ &”。

parse()函数需要验证用户输入的命令行是否符合上面的语法规则，在验证用户输入的命令行是否符合上述规则的同时，还需要把用户输入的命令行分解为可以处理的命令和参数。parse()函数的具体代码如下：

```
parse(void)
{
    int i;

    /* 1 */
    command(0);

    /* 2 */
    if (check("<"))
        getname(infile);

    /* 3 */
    for (i = 1; i<PIPELINE; ++i)
        if (check("|"))
            command(i);
        else
            break;

    /* 4 */
    if (check(">"))
    {
        if (check(">"))
            append = TRUE;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            getname(outfile);
        }

        /* 5 */
        if (check("&"))
            backgnd = TRUE;

        /* 6 */
        if (check("\n"))
            return(i);
        else
        {
            fprintf(stderr, "Command line syntax error\n");
            return(ERROR);
        }
}
```

上述代码的注释如下：

(1) 首先调用 command()函数，把命令行输入的命令分解成为单个的命令字符串，这些字符串最后被复制到 avline[]数组中，同时每一个命令字符串都可以通过指针数组 av[]中的一个指针来获得。

(2) 在命令字符串被分析后，接下来需要使用 check()函数来查找是否使用了可选的输入重定向符。如果 check()检查到了输入重定向符，将返回 true，然后调用 getname()函数来获得输入重定向文件名，并存放在 infile[]数组中。

(3) 将依据管道操作符来分割命令字符串。如果发现了一个管道操作符，那么将再次调用 command()函数来分析下一个命令。在这里，使用 for 循环反复地查找管道操作符，直到处理完所有的管道操作符或者达到了 maximum 个简单命令的限制。

(4) 将检查是否有输出重定向操作符，同时还要检查是否有第二个“ >”来确定是否存在第二次输出重定向。如果检查到存在输出重定向，那么输出重定向标志将设置为 TURE（这个标志在 initwarm()函数中被初始化设置为 FALSE）。假如存在一个输出重定向操作符，将调用 getname()函数来取得输出重定向文件名，并存放在 outfile[]数组中。

(5) 主要检查是否使用了后台运行操作符“ &”。如果使用了操作符“ &”，那么后台运行标志 backgnd 将被设置为 TRUE（在 initwarm()函数中，backgnd 被初始化为 TRUE）。

(6) 通过上面的分析，命令行到此已经被完整地分解了，整个命令行最后只应该留

PDF-XCHANGE Click to buy NOW! www.docu-track.com

下换行符“ \n”。如果命令行符合语法规则，那么命令行分析程序返回 i，i 中包含了命令行中的简单命令个数。如果在命令行的最后一位没有发现一个换行符，那么将在屏幕上打印出错误信息，并且 parse()函数返回 ERROR。

10.4.7 command()函数

简单地说，command()函数从 line[]数组以“命令+参数”的方式一次一个地复制每个命令及其参数到 avline[]数组中。每复制一个单词就使用在 cmdlin[]结构数组中的 av[]数组的一个指针指向这个命令字符的开始地址，具体代码如下：

```
command(int i)
{
    int j, flag, inword;

    for (j = 0; j<MAXARG-1; ++j)
    {
        while (*lineptr==' ' || *lineptr=='\t')
            ++lineptr;

        cmdlin[i].av[j] = avptr;
        cmdlin[i].av[j+1] = NULL;

        for (flag = 0; flag==0;)
        {
            switch (*lineptr)
            {
            case '>':
            case '<':
            case '|':
            case '&':
            case '\n':
                if (inword==FALSE)
                    cmdlin[i].av[j] = NULL;

                *avptr++ = '\0';
                return;

            case ' ':
            case '\t':
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                    inword = FALSE;
                    *avptr++ = '\0';
                    flag = 1;
                    break;

                default:
                    inword = TRUE;
                    *avptr++ = *lineptr++;
                    break;
                }
            }
        }
    }
```

command()函数由内部循环和外部循环组成。内部循环从数组 line[[中复制每个单词的字符到 avline[]数组中，然后在 avline[]数组中的每个单词后添加字符串结束符“ \0”。外部循环设置指针指向 av[]数组中合适的单词作为开始地址。

10.4.8 execute()函数

在每个完成的命令输入行中，只能包含一个输入重定向文件紧接在第一个命令之后，同时也只能有一个输出重定向文件在命令行的最后。如果在命令行中只有一个命令，那么这个命令可以是最先的一个命令，也同时是最后的一个命令，因此，它可以有输入重定向文件和输出重定向文件。

假如在命令行中，有一个以上的命令，假设有 j 个，那么需要 j-1 个管道符号来进行连接这 j 个命令。一次执行这 j 个命令，并且把这 j 个命令的执行结果使用管道“串接”起来的过程，如下面的 execute(int j)代码所示：

```
execute(int j)
{
    int i, fd, fds[2];

    /* 1 */
    if (infile[0]!='\0')
        cmdlin[0].infd = open(infile, O_RDONLY);

    /* 2 */
    if (outfile[0]!='\0')
        if (append==FALSE)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                cmdlin[j-1].outfd = open(outfile, O_WRONLY | O_CREAT
                                                | O_TRUNC, 0666);
        else
                cmdlin[j-1].outfd = open(outfile, O_WRONLY | O_CREAT
                                                | O_APPEND, 0666);

    /* 3 */
    if (backgnd==TRUE)
        signal(SIGCHLD, SIG_IGN);
    else
        signal(SIGCHLD, SIG_DFL);

    /* 4 */
    for (i = 0; i<j; ++i)
    {
        /* 5 */
        if (i<j-1)
        {
            pipe(fds);
            cmdlin[i].outfd = fds[1];
            cmdlin[i+1].infd = fds[0];
        }

        /* 6 */
        forkexec(&cmdlin[i]);

        /* 7 */
        if ((fd = cmdlin[i].infd)!=0)
            close(fd);

        if ((fd = cmdlin[i].outfd)!=1)
            close(fd);
    }

    /* 8 */
    if (backgnd==FALSE)
        while (wait(NULL)!=lastpid);
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

execute()函数使用了一个内部的参数，就是命令行中的命令个数。这个值是 j，上面程序的注释如下：

(1) 如果设定了一个输入重定向文件，那么调用 open()打开这个文件，并把文件句柄设置到命令 0（命令 0 就是命令行的第一个命令）的 cmd 结构中的 infd 域中。其中，在调用 open()打开一个文件时，应该通过检查 open()的返回值来检查打开文件是否成功。为了代码的简洁，在程序代码中略去了检查工作。

(2) 同样地，如果命令行指定了一个输出重定向文件，那么同样需要使用 open()函数开发这个文件，返回的文件句柄存放在命令行中的第 j-1 个简单命令结构的 outfd 结构成员变量中。注意，这里有两类输出重定向（覆盖方式和添加方式），这两种方式要求文件在打开时使用两种不同的方式。通过检查输出重定向的设置可以确定使用哪种合适的方式来打开文件。

(3) 处理子进程的僵尸进程。如果命令行是运行在后台方式下（backgnd==TRUE），那么 SICCHLD 将被设置，因此在子进程终止时不会产生子进程的僵尸进程。这样作的原因是在后来运行的程序，shell 不会 wait()等待这些子进程运行结束。如果命令行是运行的前台，SIGCHLD 将设为默认值，这会导致在子进程结束运行时，会产生僵尸进程等待父进程通过僵尸进程来搜集子进程运行的信息，shell 将通过 wait()来获得这些子进程的信息。

(4) 在命令行的每一个简单命令都将执行一次循环，通过变量 i 来指定是哪个简单命令正在被执行。

(5) 值 j-1 代表了命令行中的最后一个简单命令的编号。如果 j-1 是大于 0 的，那么意味着在命令行中有超过一个的简单命令。每一个编号小于 j-1 的简单命令都将创建一个管道文件来完成将编号为 i 的输出（outfd）“管道”成为第 i+1 个简单命令执行时的标准输入（infd）。

(6) 所有的第 i 个简单命令的文件描述符和参数都具备了，因此简单命令 i 可以被正确地执行了。forkexec()函数的功能就像这个函数的名称一样，它首先 fork()一个子进程，然后执行一个简单命令。forkexec()的返回值处理工作仅仅在父进程中进行。

(7) 假如当前简单命令执行的过程中有 I/O 重定向文件句柄，那么在父进程中的这些文件句柄的复制将在这里被关闭。

(8) 所有的简单命令开始执行，如果命令行在前台运行，那么父进程（也就是 shell 进程）将通过 wait()函数来等待命令行中的最后一个简单命令执行完毕。执行最后一个命令的进程 ID 被 forkexec()函数设置在 lastpid 变量中。

10.4.9 forkexec()函数

forkexec()函数使用 fork()系统调用来创建一个子进程，然后在子进程中首先执行需

PDF-XCHANGE Click to buy NOW! www.docu-track.com

要的 I/O 重定向操作，然后调用 execvp()函数来执行需要的命令。其代码如下：

```
forkexec(struct cmd *ptr)
{
    int i,pid;

    /* 1 */
    if (pid = fork())
    {
        /* 2 */
        if (backgnd==TRUE)
            printf("%d\n", pid);

        lastpid = pid;
    }
    else
    {
        /* 3 */
        if (ptr->infd==0 && backgnd==TRUE)
            ptr->infd = open("/dev/null", O_RDONLY);

        /* 4 */
        if (ptr->infd!=0)
        {
            close(0);
            dup(ptr->infd);
        }

        if (ptr->outfd!=1)
        {
            close(1);
            dup(ptr->outfd);
        }

        /* 5 */
        if (backgnd==FALSE)
        {
            signal(SIGINT, SIG_DFL);
            signal(SIGQUIT, SIG_DFL);
```

```
            }

            /* 6 */
            for (i = 3; i<OPEN_MAX; ++i)
                close(i);

            /* 7 */
            execvp(ptr->av[0], ptr->av);
            exit(1);
        }
    }
```

forkexec()的参数是一个指向代表它将执行的命令结构 cmd 的指针。在命令结构 cmd 中包含了该命令的标准输入文件句柄和标准输出文件句柄（这时，尽管 I/O 重定向还没有进行）。结构 cmd 中同时也包含了一个数组，数组中的元素指向命令行（包括命令名称），这个命令行将是 command 的 argv 参数。上述代码的注释如下：

(1) 使用 fork()创建一个子进程，同时保存 fork()返回的子进程 ID。

(2) 在父进程中，如果命令行设置了命令在后台执行，那么在第(1)步中生成的子进程 ID 需要在屏幕上打印出来。每个子进程的 ID 也需要保存在变量 lastpid 中。假如命令行的简单命令是从左到右顺序执行的，那么最后一个 lastpid 中的 ID 值就是最后一个简单命令被执行的子进程的 ID。如果命令是在前台执行，那么这个 ID 也就是 wait()函数等待的执行完成的子进程的 ID。

(3) 如果子进程是在后台执行的，那么子进程必须确保不能从键盘获得标准输入。实际上，只有第一个执行的简单命令可以从键盘或者标准输入，而所有其他的简单命令在执行的过程中都只能从管道中获得标准输入（通过设置 I/O 重定向来实现）。因此，如果当前是一个后台执行的命令，同时标准输入的文件描述符仍然设置为标准键盘输入（infd=0），那么这个简单命令一定是第一个简单命令。在这种情况下，第一个命令虽然标准输入是键盘，但是它没有机会从键盘来读取输入，因为进程在后台运行。处理这种情况的最简单和最安全的方式是设置自动的标准输入重定向。符合这个需求的最适合的重定向文件就是标准文件/dev/null。文件/dev/null 的特性是如果一个进程使用 read()来读时，马上返回 EOF。使用这个简单的方法，可以使 shell 程序简单地实现在后台执行的命令中的标准输入的重新定向。

(4) 执行实际的输入和输出重定向（通过修改标准输入和标准输出的默认文件句柄来实现）。

(5) 如果用户命令是在前台执行的，那么命令的执行可以通过在键盘上敲击中断键来终止执行，除非执行的命令禁止用户中断。

(6) 关闭所有的文件描述符（除了标准输入、标准输出和标准错误句柄以外），这样

这个进程就不会影响 shell 设置其他 I/O 重定向或者管道文件，除非可以支持通过标准文件句柄 0、1 和 2。

(7) 单个命令通过 execvp()系统调用，使用数组 av[]中存储的值来提供命令的名字和 argv 指针（参数指针）。

10.4.10 check()函数

check()函数主要完成字符串比较功能，它比较函数的传入字符串参数和 lineptr 指向的 line[]数组中的字符串。如果有匹配的字符串，lineptr 将被移动到最前面的匹配字符处，然后返回 TRUE。如果 check()没有匹配，将返回 FALSE，同时 lineptr 的位置不变。这样下一次调用 check()函数时，可以再次比较 line[]数组中的同一个字符串。check()函数源代码如下：

```
check(char *ptr)
{
    char *tptr;

    while (*lineptr==' ')
        lineptr++;

    tptr = lineptr;

    while (*ptr!='\0' && *ptr==*tptr)
    {
        ptr++;
        tptr++;
    }

    if (*ptr!='\0')
        return(FALSE);
    else
    {
        lineptr = tptr;
        return(TRUE);
    }
}
```

10.4.11 getname()函数

getname()函数复制至多 MAXNAME 个字符的文件名到作为参数的字符数组 nameparameter 中，其源代码如下：

```
getname(char *name)
{
    int i;

    for (i = 0; i<MAXNAME; ++i)
    {
        switch (*lineptr)
        {
        case '>':
        case '<':
        case '|':
        case '&':
        case ' ':
        case '\n':
        case '\t':
            *name = '\0';
            return;

        default:
            *name++ = *lineptr++;
            break;
        }
    }

    *name = '\0';
}
```

10.4.12 t-shell 的运行效果

t-shell 系统编译后，运行效果如图 10-5 所示。

图 10-5　t-shell 的运行效果

10.5　本章小结

Linux 的 shell 继承了 Unix 系统 shell 的强大和灵活的功能，是用户使用系统功能的强大工具。

本章通过介绍一个试验性的 shell 程序 t-shell，向读者展示了如何去实现一个 shell 系统的所有细节。

本章首先介绍了实现 shell 的基础，介绍了一个最简单的 shell 程序。然后，重点讲解如何实现子进程的 I/O 重定向。I/O 重定向是实现一个 shell 的重要基础功能，通过 I/O 重定向可以实现管道操作、重定向操作等高级功能。

本章还详细分析了试验 shell 系统 t-shell 的实现过程。t-shell 是一个简单而全面的 shell 系统，在 t-shell 的基础上，已经非常容易实现一个功能强大的 shell 系统了。

因此，读者在阅读本章后，可以在 t-shell 的基础上，去实现自己操作系统的 shell。

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

第11章　系统调用和 C 语言库的实现

本章知识点：

- Linux 系统调用概述
- Linux 0.01 系统调用实现分析
- 试验：在 Linux 中添加新系统调用
- C 语言标准库函数的实现

系统调用是操作系统内核和应用程序之间的主要功能接口。通过系统调用，应用程序可以对计算机系统的外围设备、文件系统、进程通信、存储管理等各个方面进行控制。

操作系统通过向应用程序开放系统调用，使应用程序可以无需了解操作系统的内部实现和计算机硬件细节，就可以实现应用程序的功能，简化了应用程序的设计，方便了应用程序的移植。

本章首先介绍在 Linux 系统中，系统调用实现的基本原理，然后通过分析 Linux 0.01 中实现系统调用的两个内核文件 system_call.s 和 sys.c 来展示 Linux 0.01 的系统调用的实现方式。本章还向读者演示了一个具体的例子：在 Linux 0.01 中添加一个自己的系统调用，并在应用程序中使用这个系统调用。

标准的 C 语言库函数，在不同的操作系统上有不同的内部实现。本章最后将介绍一些标准 C 语言库函数的实现并提供其源代码，包括字符串 / 内存处理函数、I/O 函数、工具函数等。这些 C 函数的实现代码，可以让读者了解 C 标准库的实现细节，并可以直接移植到自己编写的操作系统中。

11.1　Linux 系统调用概述

本节将介绍 Linux 系统调用的基本实现原理，重点介绍实现系统调用的宏、系统调用表、系统调用函数入口等概念。通过本节的学习，读者可以快速地掌握 Linux 中系统调用的概貌，为修改、添加系统调用功能奠定基础。

11.1.1　系统调用

在 Linux 系统中，所有进程都可以使用的操作系统服务就是系统调用，进程可以通过系统调用来请求操作系统的内核服务。在通常情况下，进程不能够直接访问操作系统的内核，也不能直接存取内核使用的内存区域，不能调用内核函数。通常，这一点是通

过 CPU 硬件措施来保证的，因为操作系统内核通常运行在 CPU 的核心态，而普通应用程序运行在用户态，用户态应用程序是无法访问核心态内核的。

如果应用程序需要突破上述限制，只有操作系统内核提供的系统调用是一个例外。应用程序可以通过一个固定的过程，从而调用内核提供的功能。在 Intel 体系结构的计算机中，这是通过执行中断 0x80h 实现的。

应用程序通常是一个进程，进程在调用内核时，跳转到内核代码中的位置一般标记为 system_call（在 Linux 0.01 中，system_call 是汇编程序 system_call.s 中的一段代码的入口点的标记）。在 system_call 位置的代码将检查系统调用号，依据系统调用号告诉系统内核进程请求的系统服务是什么。然后，它再查找系统调用表 sys_call_table[]，找到希望调用的内核函数的地址，并调用此函数，最后将控制权返回应用程序。

如果希望改变一个系统调用函数，或者添加一个新的系统调用，可以编写一个自己的函数，然后改变 sys_call_table[]中的指针并指向该函数。

11.1.2 系统调用的实现

在 Linux 系统中，系统调用是作为一种异常处理来实现的。系统调用将执行相应的机器代码指令来产生异常信号，产生中断或异常的重要效果是系统自动将用户态切换为核心态来对它进行处理。这就是说，执行系统调用异常指令时，自动地将系统切换为核心态，并安排异常处理程序的执行。

Linux 用于实现系统调用异常的实际指令是：

int $0x80

这一指令使用中断 / 异常向量号 128（即 16 进制的 80）将控制权转移给内核。为达到在使用系统调用时不必用机器指令编程，在标准 C 语言库中为每一个系统调用都提供了一段短的子程序，完成机器代码的编程工作。事实上，机器代码段非常简短，它所要做的工作只是将送给系统调用的参数加载到 CPU 寄存器中，接着执行 int $0x80 指令，然后运行系统调用，系统调用的返回值将送入 CPU 的一个寄存器中，标准库函数中的一段代码将取得这一返回值，并将它送回用户程序。

为使系统调用的执行成为一项简单的任务，Linux 提供了一组预处理宏指令，它们可以用在程序中。这些宏指令取一定的参数，然后扩展为调用指定的系统调用的函数。通过使用这些预处理宏指令，可以非常方便地实现自己的系统调用功能。

1. 宏：_syscallN(type, name, x…)

在 Linux 中，定义系统调用的预定义宏为：

_syscallN(parameters)

其中 N 是系统调用所需的参数数目，而 parameters 则用一组参数代替。这些参数使宏指令完成适合于特定的系统调用的扩展。例如，为了建立调用 setuid()系统调用的函数，应该使用：

_syscall1(int, setuid, uid_t, uid)

syscallN()宏指令的第一个参数 int 说明产生的函数的返回值的类型是整型，第二个参数 setuid 说明产生的函数的名称，第三个参数 uid_t 和第四个参数 uid 指定 setuid 系统调用所需要参数的类型和名称。

注意：用作系统调用的参数的数据类型有一个限制，它们的长度不能超过 4B。这是因为执行 int $0x80 指令进行系统调用时，所有的参数值都存放在 32 位的 CPU 寄存器中。如果参数的长度大于 32 位，那么将无法存放在一个 32 位的寄存器中。

使用 CPU 寄存器传递参数带的另一个限制是可以传送给系统调用的参数的数目。这个限制是最多可以传递 5 个参数。所以，最多可以定义 6 个不同的_syscallN()宏指令，从_syscall0()，_syscall1()直到_syscall5()。在 Linux 0.01 中，仅仅定义了从_syscall0()到_syscall4()，具体的定义如下：

```
#define _syscall0(type, name)                                   // 无参数系统调用
#define _syscall1(type, name, atype, a)                         // 一个参数的系统调用
#define _syscall2(type, name, atype, a, btype, b)       // 两个参数的系统调用
#define _syscall3(type, name, type1, arg1, type2, arg2, type3, arg3)
                                                                // 三个参数的系统调用
```

这些宏的具体代码如下：

```
。。#d 商 ne—sy~o（type，name）＼ty
penarne（void）|i＼type_re，s；l／／
//下面的内嵌汇编的意思是：
||movl—NR 一舔蚺 ilarfle％eax||int 篙 0x80 ¨ movl％瞄 x_res ‘。
＆南渤幽，“ int 髫 0x80"r ’
j 名／／（_|res）ri--
：”0”（一 NR 一##name））；＼…i_
if 乙，res>=刃|；j
retulTl—res；t。
i~rl'no=一一 j$；{
retlarn 一 1；}#define—syscalll（type，mme，atype，a）＼typename&typea）＼I＼坼 E—res；t／／
//下面的内嵌汇编的意思是：
／／movl—NR 一##l~aTle％eax／Imovla％ebx||int$0x80／／movl％ea】【一 res—m—Volatile（”int$0X80”＼：”=r（～res）＼第 11 章系统调用和 c 语言库的实现，257，：，，o”（一 NR 一##name），“ r（a））；＼if（一 res>=0）＼re 岫一 res；＼eml0=一一 res；＼retLlm 一 1；＼#define—syscall2(type，narne，awe，a 山 type，b)＼type 毗 me(atypea'btypeb)＼l＼type—res；＼／／
//下面的内嵌汇编的意思是：
／／movl—NR 一##nanle％e “／／movla％ebx||movlb％eex／／int0x80l】movi％e“一 r％一 m—v01atge（”int$0xS0”＼：”=a，’（一 res）＼：…0（～NR 一##narne），”b”（a），”i’（b））；＼if（一 res>=0）＼retum—res；＼erm0。一一 res；＼口 mm 一 1；
```

|。，

```
#define—syscaⅡ3（type，name，atype，a，btype，b，ctype，e）＼
typenarne（atypea，btypeb，ctypee）＼
{＼‘。type—res；＼／／
//下面的内嵌汇编的意思是：
／／movl—NR 一##narne％eaX／／movla％ebx／／movlb％ec】（／／movlc％edx
／／int$0x80／／movl％eax—res—as 玎 1 一 volatile（”int$0x80”＼：”=//（一 res）＼：”
0”（一 NR 一##narne），”H’（a），”d’（b），”d”（c））；＼if（一 res<0）＼errno=一一
res，一 res。一 1；＼retur 兀 res；＼258‘ Linux0，01 内核分析与操作系统设计#define
—syscalk3（type，nBtne，typel，argl，type2，a 啦，type3，a 哂）＼typename（typelargl，
0ype2arg2，type3arg3）＼I＼long—re，s；|一 m—volati]e（“ imS0x80 “＼
：”=i’（～re5）＼
：…0’（一 NR 一#name），”H’（（10ng）（argl）），’’d’（（10ng）（arg7））’＼
…d’（（10ng）（arg3）））；＼if（～res>=0）＼
return（type）一 re，s；＼eriTlO~一一 res；＼return 一 1；＼
```

在 Linux 0.01 中，这些宏定义在 include/unistd.h 文件中。该文件中还以__NR_name 的形式定义了 66 个常数，这些常数就是系统调用函数 name 的函数指针在系统调用表中的偏移量。

注意：这些常数的数量是可以增加的。Linux 提供的系统调用越多，这个常数的数量就越大。在现在的 Linux 内核中，已经定义了超过 100 个的系统调用，但 Linux 0.01 只有 66 个，如下所示：

```
#define __NR_setup      0     /* used only by init, to get system going */
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write      4
#define __NR_open       5
#define __NR_close      6
#define __NR_waitpid    7
#define __NR_creat      8
#define __NR_link 9
#define __NR_unlink     10
#define __NR_execve     11
#define __NR_chdir      12
#define __NR_time 13
#define __NR_mknod      14
#define __NR_chmod      15
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
#define __NR_chown      16
#define __NR_break      17
#define __NR_stat       18
#define __NR_lseek      19
#define __NR_getpid     20
#define __NR_mount      21
#define __NR_umount     22
#define __NR_setuid     23
#define __NR_getuid     24
#define __NR_stime      25
#define __NR_ptrace     26
#define __NR_alarm      27
#define __NR_fstat      28
#define __NR_pause      29
#define __NR_utime      30
#define __NR_stty       31
#define __NR_gtty       32
#define __NR_access     33
#define __NR_nice       34
#define __NR_ftime      35
#define __NR_sync       36
#define __NR_kill       37
#define __NR_rename     38
#define __NR_mkdir      39
#define __NR_rmdir      40
#define __NR_dup        41
#define __NR_pipe       42
#define __NR_times      43
#define __NR_prof       44
#define __NR_brk        45
#define __NR_setgid     46
#define __NR_getgid     47
#define __NR_signal     48
#define __NR_geteuid    49
#define __NR_getegid    50
#define __NR_acct       51
#define __NR_phys       52
#define __NR_lock       53
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#define __NR_ioctl 54
#define __NR_fcntl 55
#define __NR_mpx 56
#define __NR_setpgid  57
#define __NR_ulimit   58
#define __NR_uname    59
#define __NR_umask    60
#define __NR_chroot   61
#define __NR_ustat    62
#define __NR_dup2     63
#define __NR_getppid  64
#define __NR_getpgrp  65
#define __NR_setsid   66
```

2. 系统调用表

系统调用表是一张表格，按照顺序定义了系统中所有的系统调用的入口函数地址。在 Linux 0.01 中，系统调用表定义在 include/Linux/sys.h 中，如下所示：

```
fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
sys_getpgrp,sys_setsid};
```

在数组 sys_call_table[]中的每一个元素都是一个函数指针（在 C 语言中，函数名代表指向函数入口的指针），按系统调用号（即前面提到的__NR_name）排列了所有系统调用函数的指针，以供系统调用入口函数查找。从这张表可以看出，Linux 给它所支持的系统调用函数取名叫 sys_name。

在最新的 Linux 内核中，系统调用表的位置有所改变，例如 Linux 2.0.0 以后的内核中，系统调用表定义在 entry.s 文件的最后部分。

3. 系统调用入口函数

系统调用入口函数定义在 system_call.s 文件中，代码如下：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
_system_call:
    cmpl $nr_system_calls-1,%eax          //如果 eax>=67，超过本内核的系统调用数目，出错
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx          # push %ebx,%ecx,%edx as parameters
    pushl %ebx          # to the system call
    movl $0x10,%edx     # set up ds,es to kernel space  //将 ds、es 指向系统数据短
    mov %dx,%ds
    mov %dx,%es
    movl $0x17,%edx     # fs points to local data space //将 fs 指向用户数据段
    mov %dx,%fs
    call _sys_call_table(,%eax,4)
    pushl %eax
//以下 5 行判断这系统调用是否在运行，并且做相应的处理
    movl _current,%eax
    cmpl $0,state(%eax)         # state         //如果当前的任务不在运行，那么重新调度
    jne reschedule
    cmpl $0,counter(%eax)       # counter //如果此调用在运行，并且没有其他程序调用它，就运行此调用
    je reschedule
. . . . . .
reschedule:
    pushl $ret_from_sys_call
    jmp _schedule
. . . . . .
```

从 system_call 入口的汇编程序的主要功能是：

- 检验是否为合法的系统调用。
- 保存寄存器的当前值。
- 根据系统调用表_sys_call_table 和 EAX 寄存器持有的系统调用号找出并转入系统调用响应函数。
- 从该响应函数返回后，让 EAX 寄存器保存函数返回值，跳转至 ret_from_sys_call，当 ret_from_sys_call 结束后，将执行进程调度。

- 在执行位于用户程序中系统调用命令后面余下的指令之前，若 INT 0x80h 的返回值非负，则直接按类型 type 返回；否则，将 INT 0x80h 的返回值取绝对值，保留在 errno 变量中，返回-1。

在 ret_from_sys_call 中，也是 Linux 系统进行进程调度切换的时机（在普通的 Linux 系统中，进程在系统调用中执行的时候是不能被切换的）。

当在程序代码中用到系统调用时，编译器会将上面提到的宏展开，展开后的代码实际上是将系统调用号放入 EAX 后，调用 INT 0x80h 使处理器转向系统调用入口，然后查找系统调用表，进而由内核调用真正的系统功能函数。

通过上面的分析可以看到，如果希望在 Linux 系统中添加新的系统调用，那么在应用程序中必须显式地使用宏_syscallN 来调用这个新的系统调用。

在 Linux 中，系统调用函数的函数名以＂sys_”开头，后跟该系统调用的名字。因此，构成 nr_system_calls 个形如 sys_name()的函数名。例如，系统调用 read 的响应函数是 sys_read()（在文件 read_write.c 中）。

11.1.3 系统调用到 INT 0x80h 中断请求的转换

当进程需要进行系统调用时，必须以 C 语言函数的形式写一句系统调用命令，当进程执行到此系统调用命令时，实际上执行了由宏命令_syscallN()展开的函数。系统调用的参数由各通用寄存器传递。然后执行 INT 0x80h，以核心态进入入口地址 system_call。

在 11.1.2 小节中已经介绍了宏定义 syscallN()，宏 syscallN()主要用于系统调用的格式转换和参数的传递。例如，write()系统调用有 3 个参数，使用了_syscall3(int, write, int, fd, const char *, buf, off_t, count)，这个定义在通过 C 语言的预编译后，展开成为如下的代码：

```
int write(intfd，constchar*bur，off—tcount）long—res；一Xll一 volatile（＂int$0x80”：＂
=i’（一 res）：…0（一 NR—write），＼
＂b”（（10ng）（fd）），＂i’（（10ng）（bur）），＼
＂d”（（10ng）（count））；if（一 res>=0）
return（int）reslermo=一 resretum 一 1；
```

从上面展开宏的代码可以看到，通过使用 syscallN()宏，为系统调用 write 添加了调用中断 INT 0x80h 的代码，包括参数和返回值的传递。

11.2 Linux 0.01 系统调用实现分析

11.2.1 system_call.s

system_call.s 是使用汇编语言书写的，包含了系统调用的主要处理程序，同时也包

PDF-XCHANGE Click to buy NOW! www.docu-track.com

含了时钟中断处理程序，硬盘中断处理程序也包含在本文件中。

本文件是 Linux 系统调用的主要文件，能够演示出 Linux 0.01 中系统调用的实现细节。其代码如下：

```
/*
 *  system_call.s  contains the system-call low-level handling routines.
 * This also contains the timer-interrupt handler, as some of the code is
 * the same. The hd-interrupt is also here.
 *
 * NOTE: This code handles signal-recognition, which happens every time
 * after a timer-interrupt and after each system call. Ordinary interrupts
 * don't handle signal-recognition, as that would clutter them up totally
 * unnecessarily.
 *
 * Stack layout in 'ret_from_system_call':
 *
 *      0(%esp) - %eax
 *      4(%esp) - %ebx
 *      8(%esp) - %ecx
 *      C(%esp) - %edx
 *     10(%esp) - %fs
 *     14(%esp) - %es
 *     18(%esp) - %ds
 *     1C(%esp) - %eip
 *     20(%esp) - %cs
 *     24(%esp) - %eflags
 *     28(%esp) - %oldesp
 *     2C(%esp) - %oldss
 */

SIG_CHLD    = 17
EAX         = 0x00
EBX         = 0x04
ECX         = 0x08
EDX         = 0x0C
FS      = 0x10
ES      = 0x14
DS      = 0x18
EIP     = 0x1C
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
CS          = 0x20
EFLAGS      = 0x24
OLDESP      = 0x28
OLDSS       = 0x2C

state = 0       # these are offsets into the task-struct.
counter   = 4
priority = 8
signal    = 12
restorer = 16       # address of info-restorer
sig_fn    = 20      # table of 32 signal addresses

nr_system_calls = 67

.globl _system_call,_sys_fork,_timer_interrupt,_hd_interrupt,_sys_execve

.align 2
bad_sys_call:
    movl $-1,%eax
    iret
.align 2
reschedule:
    pushl $ret_from_sys_call
    jmp _schedule
.align 2
_system_call:
    cmpl $nr_system_calls-1,%eax          //如果 eax>=67，超过本内核的系统调用数
目，出错
    ja bad_sys_call
    push %ds
    push %es
    push %fs
    pushl %edx
    pushl %ecx          # push %ebx,%ecx,%edx as parameters
    pushl %ebx          # to the system call
    movl $0x10,%edx         # set up ds,es to kernel space  //将 ds、es 指向系统数据短
    mov %dx,%ds
    mov %dx,%es
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        movl $0x17,%edx         # fs points to local data space //将 fs 指向用户数据段
        mov %dx,%fs
        call _sys_call_table(,%eax,4)
        pushl %eax
    //以下 5 行判断这系统调用是否在运行，并且做相应的处理
        movl _current,%eax
        cmpl $0,state(%eax)             # state           //如果当前的任务不在运行，那么重新
调度
        jne reschedule
        cmpl $0,counter(%eax)       # counter //如果此调用在运行，并且没有其他程序调
用它，就运行此调用
        je reschedule
    ret_from_sys_call:
    //以下 3 行是如果任务是当前任务，返回
        movl _current,%eax              # task[0] cannot have signals
        cmpl _task,%eax
        je 3f
    //以下 3 行是只要%cs 的选择器不是内核选择器，就 je 3f
        movl CS(%esp),%ebx          # was old code segment supervisor
        testl $3,%ebx               # mode? If so - don't check signals
        je 3f
        cmpw $0x17,OLDSS(%esp)          # was stack segment = 0x17 ?//堆栈段中有没有
选择器 0x17
        jne 3f
    2:  movl signal(%eax),%ebx              # signals (bitmap, 32 signals)
        bsfl %ebx,%ecx                  # %ecx is signal nr, return if none  //从低位开始测试
ecx 的各位，当遇到有 1 时，ZF=0，且将该位的序号存入 ebx
        je 3f
        btrl %ecx,%ebx                  # clear it  //与 bt 相同，但将 EBX 中的相应的送入
CF 中的位清 0
        movl %ebx,signal(%eax)                      //修改当前任务的信号
        movl sig_fn(%eax,%ecx,4),%ebx  # %ebx is signal handler address
        cmpl $1,%ebx
        jb default_signal           # 0 is default signal handler - exit
        je 2b                   # 1 is ignore - find next signal
    //如果 ebx>1，运行以下代码，主要是保存调用以前的各寄存器数据到 fs 段
        movl $0,sig_fn(%eax,%ecx,4)     # reset signal handler address //使该地址为 0，退
出
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        incl %ecx
        xchgl %ebx,EIP(%esp)        # put new return address on stack
        subl $28,OLDESP(%esp)
        movl OLDESP(%esp),%edx      # push old return address on stack
        pushl %eax                  # but first check that it's ok.
        pushl %ecx
        pushl $28
        pushl %edx
        call _verify_area
        popl %edx
        addl $4,%esp
        popl %ecx
        popl %eax
        movl restorer(%eax),%eax
        movl %eax,%fs:(%edx)        # flag/reg restorer
        movl %ecx,%fs:4(%edx)       # signal nr
        movl EAX(%esp),%eax
        movl %eax,%fs:8(%edx)       # old eax
        movl ECX(%esp),%eax
        movl %eax,%fs:12(%edx)      # old ecx
        movl EDX(%esp),%eax
        movl %eax,%fs:16(%edx)      # old edx
        movl EFLAGS(%esp),%eax
        movl %eax,%fs:20(%edx)      # old eflags
        movl %ebx,%fs:24(%edx)      # old return addr
3:      popl %eax
        popl %ebx
        popl %ecx
        popl %edx
        pop %fs
        pop %es
        pop %ds
        iret

default_signal:
        incl %ecx
        cmpl $SIG_CHLD,%ecx         //有没有子信号
        je 2b
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        pushl %ecx
        call _do_exit       # remember to set bit 7 when dumping core
        addl $4,%esp
        jmp 3b

.align 2
_timer_interrupt:
        push %ds        # save ds,es and put kernel data space
        push %es        # into them. %fs is used by _system_call
        push %fs
        pushl %edx          # we save %eax,%ecx,%edx as gcc doesn't
        pushl %ecx          # save those across function calls. %ebx
        pushl %ebx          # is saved as we use that in ret_sys_call
        pushl %eax
        movl $0x10,%eax         //ds、es 指向内核数据段
        mov %ax,%ds
        mov %ax,%es
        movl $0x17,%eax         //fs 指向用户数据段
        mov %ax,%fs
        incl _jiffies
    //以下 3 行看看权限
        movb $0x20,%al          # EOI to interrupt controller #1     //中断结束命令，使中断
控制器能够处理其他中断
        outb %al,$0x20
        movl CS(%esp),%eax
        andl $3,%eax        # %eax is CPL (0 or 3, 0=supervisor)
        pushl %eax
        call _do_timer      # 'do_timer(long CPL)' does everything from
        addl $4,%esp        # task switching to accounting ...
        jmp ret_from_sys_call

.align 2
_sys_execve:
        lea EIP(%esp),%eax
        pushl %eax
        call _do_execve
        addl $4,%esp
        ret
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
.align 2
_sys_fork:
        call _find_empty_process
        testl %eax,%eax
        js 1f
        push %gs
        pushl %esi
        pushl %edi
        pushl %ebp
        pushl %eax
        call _copy_process
        addl $20,%esp
1:      ret

_hd_interrupt:
        pushl %eax
        pushl %ecx
        pushl %edx
        push %ds
        push %es
        push %fs
        movl $0x10,%eax
        mov %ax,%ds
        mov %ax,%es
        movl $0x17,%eax
        mov %ax,%fs
        movb $0x20,%al
        outb %al,$0x20          # EOI to interrupt controller #1
        jmp 1f                  # give port chance to breathe
1:      jmp 1f
1:      outb %al,$0xA0          # same to controller #2   //中断控制器 2 结束，接收其他中
断
        movl _do_hd,%eax
        testl %eax,%eax
        jne 1f
        movl $_unexpected_hd_interrupt,%eax
1:      call *%eax              # "interesting" way of handling intr.
```

```
pop %fs
pop %es
pop %ds
popl %edx
popl %ecx
popl %eax
iret
```

11.2.2 sys.c

sys.c 包含了 Linux 0.01 中大量系统调用的实现代码。在本文件中，主要包含了两大类系统调用，一类是在 Linux 0.01 中已经实现了的，书写了具体的代码；一类是尚未实现的，仅仅包含了返回错误值的代码。sys.c 的具体代码可以参见光盘中的文件 sys.c。

11.3 试验：在 Linux 中添加新的系统调用

本节将介绍一个非常有趣的试验。在本节中，将在 Linux 0.01 中添加一个新的系统调用 sys_mycall()，它的功能是将一个整数加 1 后返回值。

如果用户在 Linux 中添加新的系统调用，应该遵循几个步骤才能添加成功，下面几个步骤详细说明了添加系统调用的相关内容。

11.3.1 实现系统调用的代码

首先需要编写实现该功能的函数，该函数的名称应该是新的系统调用名称前面加上 sys_标志。假设新加的系统调用为 mycall(int num)，在/Linux/kernel/sys.c 文件中添加源代码，如下所示：

```
int sys_mycall(int num)
{
    return num + 1;
}
```

11.3.2 链接新的系统调用

实现新的系统调用代码后，下一个任务是使 Linux 内核的其余部分知道这个新的系统调用的存在。为了从已有的内核程序中增加到新的函数的链接，需要编辑两个文件。第一个文件是：/linux/include/unistd.h，在其中添加如下代码：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#define __NR_getppid   64
#define __NR_getpgrp   65
#define __NR_setsid    66
#define __NR_mycall    67          /* 添加的一行 */
```

mycall 系统调用号为 67，因此在最后一个系统调用号的基础上加 1 即可。

第二个要修改的文件是/linux/include/linux/sys.h，在其中的 sys_call_table[]尾部添加函数入口地址 sys_mycall，具体代码如下：

```
extern int sys_getppid();
extern int sys_getpgrp();
extern int sys_setsid();
extern int sys_mycall();

fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
sys_getpgrp, sys_setsid, sys_mycall};
```

注意：在数组 sys_call_table[]中，元素 sys_mycall 是第 67 个。

11.3.3 重新编译 Linux 0.01 的内核

为使新的系统调用生效，需要重建 Linux 0.01 的内核。在 Linux 0.0l 的内核源代码目录下输入命令：

```
make cleam
make
```

编译完成后，为了试验新的系统调用，可以使用如下的代码：

```
#inelude <linux/unistd.h>

_sys_call1(int,mycall,int,ret)          // 使用系统调用宏
```

```
main()
{
    printf("%6d\n",mycall(1974));
}
```

11.4 C 语言标准库函数的实现

在标准 C 语言中，许多的库函数功能，需要在新的操作系统中实现。本节提供了主要的标准库函数的实现方式，供读者参考。这些代码可以直接应用在读者的操作系统中，供读者进行系统调用或者作为库函数使用。

11.4.1 字符串和内存操作函数

C 语言的标准字符串和内存操作函数包括 strcpy()、memcpy 等函数。这些函数可以大大提高程序对字符串和内存的处理效率，它们基本上和操作系统是无关的，其代码可以较为容易地移植到所需的系统中。

1. strlen()函数

strlen()返回一个字符串的长度，不包括结束符 NULL，其代码如下：

sizetstrlen（corastchar "str）for（ret—val=0；"stl"!=‘＼0’；StT++）ret—val++；returnret—val；

2. strcmp()函数

strcmp()比较字符串 s1 和 s2，当 s1<s2 时，返回值<0，当 s1=s2 时，返回值=0，当 s1>s2 时，返回值>0。其代码如下：

，intstrcmp（constchar "stzd，constchar*str2）while（（*str21=‘＼0’）＆＆（ "stfi==*str2））strl++：str7++：3，

3. strstr()函数

strstr()函数从字符串 s1 中寻找 s2 第一次出现的位置（不比较结束符 NULL），返回指向第一次出现 s2 位置的指针，如果没找到则返回 NULL。其代码如下：

constcbar*strstr（constchar*sl，constchar*s2）constchar " start—sl=NULLconstchar*|n—s2=NULL；for（；" sl!=‘＼0’；sl++）if（start—sl=：NULL）|*firstcharofmatch*|if（ " sl==*丑）|*rememberstartofmatchingsubstringinsl*|star[一 sl=sl；，n—s2=s2+1；第 11 章系统调用和 c 语言库的实现，271，|*donealready？*|if（*|力一 s2==‘＼0’）

returnstartsl； ／*continuedmis—matchnothing；*／{*continuedmatch*lif（*sl==*in—s2）returnNULL_n—s2 十+；{*done*|if（*|n—s2==‘＼0’）

returnstart—sl；|*firstcharofmis—match*|start—sl=NULL；4。

4. memcpy()函数

由 src_ptr 所指的内存区域复制 count 个字节到 dst_ptr 所指的内存区域。src_ptr 和 dst_ptr 所指的内存区域不能重叠，函数返回指向 dst_ptr 的指针。其代码如下：

void*memcpy（void*dst—ptr，constvoid“ src—ptr，size—tcount）constchar ” src=（constchar*）src—ptrchar*dst=（char*）dst—ptr；void ”ret—vai=dst—ptr；for（；count!=0；count 一一）” dst=dst+十：src++：rettiITlretval · 272 · Linux0，01 内核分析与操作系统设计 5，

5. memmove()函数

由 src_p 所指的内存区域复制 count 个字节到 dst_p 所指的内存区域。src_p 和 dst_p 所指的内存区域可以重叠，但复制后 src_p 内容会被更改。函数返回指向 dst_p 的指针。其代码如下：

void ” memmove（void*dst—P，constvoid*src—P，size—tcount）COnStchar*fire。（constchar*）fire—Pchar*dst=（char*）dst—p；if（dst—P<arc—P）／*copyup” ／／or（；count!=0；count 一一）

*dst++=” src++：else|*copydown*{dst 十=（count 一 1）；arc+=（count 一 1）；for（；count!=0；count

*dst 一一=” arc

11.4.2 I/O 函数库

C 语言的 I/O 函数，包括 fgetc()、fputc()、printf()等，这些函数是应用程序基本的 I/O 程序，使用非常广泛。

1. fputc()函数

fputc()函数返回一个向文件所写字符的值，此时写操作成功，否则返回 EOF（文件结束其值为-1）。其代码如下：

int{putc（intC，FILE*stream）I

charone—char；

intret—vaI；ret—val。c；if（stream 一>flags==一 IONBFIIstream…>size0）／*unbufffired*／ONe—char=（char）c；if（，mite（stream 一>handle，&one—char，1）!=1）第 11 章系统调用和 c 语言库的实现 jelseIretval=EOFif（stream->room==0）Il（岨 ush（stream）!=0）

returnEOF；” stream 一>huf—ptr=（char）e；stream 一>bur—ptr++；stream 一>room 一一：if（（stream～>flags&一 IOLBF）＆＆（c==‘＼n’））if（塌 ush（stream）!=0）ret

PDF-XCHANGE Click to buy NOW! www.docu-track.com

—val=EOF；2，

2. fflush()函数

当用标准文件函数对文件进行读写操作时，首先将所读写的内容放进缓冲区，即写函数只对输出缓冲区进行操作，读函数只对输入缓冲区进行操作。例如向一个文件写入内容，所写的内容将首先放在输出缓冲区中，直到输出缓冲区存满或使用 fclose()函数关闭文件时，缓冲区的内容才会写入文件中。若无 fclose()函数，则不会向文件中存入所写的内容或写入的文件内容不全。fflush()函数将输出缓冲区的内容实际写入文件中，而将输入缓冲区的内容清除掉。其代码如下：

、，nt 栩 ushfFILE0stream）intbytes—to—write，ret—val；ret—val=O；if（stre，~n 一>size!=0）/ *buffered？ * / }bytes—to—write。stresm 一>size 一啦阻 m 一>room；if（bytes—to—write!=O）if（write（stream～>handle，stream 一>buf—basebytes—to—write）!=bytes—to 一、柑 te）ret—val=EOF；stream 一>buf—ptr。stream 一>buf—base；Stream 一>r00m。stream 一>size；· 274 · Linux0，Ol 内核分析与操作系统设计 3，

3. doprintf()函数

doprintf()函数实现可变参数的字符串输出功能，是所有 printf 类型函数的实现基础。doprintf()函数的实现代码基本和操作系统无关，可以方便地移植到任何系统上。在这里提供了 doprintf()、sprintf()、printf()等函数的实现，带有详细注释的具体代码可以参见本书光盘中的文件 doprintf.c。

11.4.3 工具函数：随机数

本节主要介绍 rand()和 srand()两个随机数函数的实现细节。随机数是一种重要的功能，在程序设计的许多方面都可能使用，本小节介绍的 rand()和 srand()两个函数都可以方便地移植到任何系统中去。

1. rand()函数

rand()函数根据随机数种子，产生一个随机数，其代码如下：

```
unsigned—seed；unsignedrand（void） / *stdlib，11* / if（一 seed==0）
～seed=1；if（（（（一 seed<<3）‘一 seed）&0x80000000L）!=0）
～seed=（一 seed<<1）l1；else
～seed<<=1；returnseed 一 1；2，
```

2. srand()函数

srand()函数设置随机数种子，其代码如下：

externunsigned — seed； |*inrand， c*|voidsrand （ unsignednew — seed ） |*stdlib，h*|seed=newseed

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

11.5 本章小结

系统调用是操作系统的重要组成部分，应用程序可以通过系统调用对计算机系统的外围设备、文件系统、进程通信、存储管理等各个方面进行控制。

本章重点介绍了 Linux 系统调用的实现原理，包括系统调用中的参数和返回值传递、系统调用宏的实现。

本章重点分析了 Linux 0.01 中实现系统调用的两个内核文件 system_call.s 和 sys.c，然后，本章向读者演示了一个具体的例子：在 Linux 0.01 中添加一个自己的系统调用，并在应用程序中使用这个系统调用。

标准的 C 语言库函数，在不同的操作系统上有不同的内部实现。本章最后介绍了一些标准 C 语言库函数的实现并提供其源代码，包括字符串／内存处理函数、I/O 函数、工具函数等。读者可以将这些函数代码移植到自己的系统中，从而快速实现自己系统上的标准库函数。

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

第12章　Linux 网络实现分析

本章知识点：

- TCP/IP 概述
- 环型缓冲区
- 通信协议中的定时器编程
- 简单停等协议实现
- 实例：串口数据通信的实现
- 实例：简单停等协议的实现

Linux 是一种强大的网络操作系统，可以提供稳定安全的网络通信能力。本章主要介绍在 Linux 系统中实现网络通信协议的一些基本知识。

本章首先介绍 Linux 网络中事实上的标准——TCP/IP 协议。通过对 TCP/IP 协议的了解，可以让读者加深对 Linux 网络协议实现的理解。

在实现通信协议中，环形缓冲区是一种使用非常广泛的数据结构。为了帮助读者快速地实现具体的通信协议，本章介绍了如何实现一个环形缓冲区，以帮助读者快速实现这一高效通用的数据结构。

在通信协议中，定时器也是必须使用的功能之一。本章介绍了如何通过 API 函数来实现定时器功能，同时也介绍了利用 Pc 计算机的硬件来实现高精度的定时器功能。这些实现代码和方法可以直接使用在操作系统的实现代码中。

本章介绍了两种具体的数据通信协议的实现方法。一种是最简单的 PC 计算机的串行通信。串行通信是最简单、最实用的数据通信方法，在实际的应用中被大量地使用。另外，本章还介绍了如何实现最简单的无错通信方式——停等协议。停等协议有很大的实用性，读者可以通过对停等协议的学习来了解使用的通信协议的实现细节

12.1　TCP/IP 概述

Linux 是一种网络操作系统，以其强大的网络通信能力著称。Linux 支持的网络办议很多，但事实上的标准是 TCP/IP 协议。TCP/IP 协议也是现在网络互联事实上的标准协议。

在 Linux 0.01 中并没有实现 TCP/IP，因此，本章只简单介绍一下 TCP/IP 协议，使读者可以了解 TCP/IP 协议。

TCP/IP 协议是一套数据通信协议，其名字是由这些协议中的两个主要协议组成的，

即传输控制协议（Transmission Control Protocol，TCP）和网间协议（Internet Protocol，IP）。虽然还有很多其他协议，但是 TCP 和 IP 显然是两个最重要的协议。

TCP/IP 协议是开放式协议标准，与计算机硬件或操作系统无关。TCP/IP 协议与物理网络硬件无关，这允许 TCP/IP 可以将很多不同类型的网络集成在一起。

12.1.1 协议分层原则

“分层是解决复杂问题的基本方法之一”。这是计算机科学的一个基本思想。

对于一个复杂的网络协议而言，通常使用分层的原则来进行设计。在分层协议体系结构中，接收方第 n 层协议收到的对象（object）应当与发送方第 n 层协议发送的对象完全一致，此种设计方式使得设计者每次仅需关注一层协议，不必考虑低层的行为。

例如，在 TCP/IP 协议栈中，TCP 协议接收来自上层应用程序的数据并将之切分为合适的数据段，并加上适当的表头（header）数据后，交由下层的网络层、链路层进行数据的传输工作，然而在逻辑上，却可认为是接收与发送两端的传输层使用 TCP 协议进行数据的传输，如图 12-1 所示。在分层协议体系结构中，可以认为：协议间的交互只发生在同一层的相同协议之间。

图 12-1　TCP/IP 协议栈的分层结构

在讨论网络的分层体系时，OSI 模型是另外一种值得参考的模型。OSI 模型将网络分为七层：应用层、表示层、会话层、传输层、网络层、数据链路层和物理层。

TCP/IP 协议栈使用了四层模型：应用层、传输层、网络层和网络访问层。应用层主要由应用程序和进程组成，传输层提供端到端的数据传输服务，网络层定义了数据报文和数据报文的路由，网络访问层实现对下层网络的访问功能。

TCP/IP 协议栈的数据结构如图 12-2 所示。在 TCP 的应用层中，将数据称为“数据流（stream）”；而在用户数据报协议（UDP）的应用层中，则将数据称为“报文（message）”。TCP 将它的数据结构称作“段（segment）”，而 UDP 将它的数据结构称作“分组（pACKet）”；网络层则将所有数据看作是一个块，称为“数据报（datagram）”。

TCP/IP 使用多种不同类型的底层网络，每一种都用不同的术语定义它传输的数据，大多数网络将传输的数据称为“分组”或“帧（frame）”。

图 12-2　TCP/IP 协议栈的数据结构

在 TCP/IP 协议栈中，数据流动则是由发源层依序传至最底层，之后透过传输介质送抵对方的最底层，再依序传至目标层。每一层将数据传至下一层之前会在数据块的前端

附加一称作首部（header）的控制信息，此首部记录了该数据块相对于该层的特性及信息，每一层会将上一层传来的数据连同其首部一同视为上层的数据，并附加该层的首部之后再送至下一层，这种过程称之为：数据封装（encapsulation）。

当数据传送到目的端后，接收方会依次进行数据的解封装（decapsulation），即每一层由下一层收到数据之后，会先剥去该层的首部，之后再将剩余的数据部份送至上一层进行继续的剥离首部操作，直至最后获得原始数据为止。TCP/IP 协议栈数据发送过程如图 12-3 所示。

图 12-3　TCP/IP 协议栈数据发送过程

12.1.2　网络访问层

网络访问层（Network Access Layer）是 TCP/IP 协议栈的最低层，该层中的协议提供了一种数据传送的方法，使得系统可以通过直接连接的网络将数据传送到其他设备，并定义了如何利用网络来传送数据报。网络访问层协议与较高层协议不一样，它必须知道底层网络的各种细节（如它的分组结构、寻址方式等），以便准确地格式化传输的数据，使其遵守网络规定。TCP/IP 网络访问层可以包括 OSI 参考模型中下三层（网络层、数据链路层和物理层）的全部功能。

网络访问协议种类繁多，每一个协议都对应一种物理网络标准。该层执行的功能包括将 IP 报文封装成被网络传输的帧，并将 IP 地址映射为网络使用的物理地址。

12.1.3　网络层

网间协议 IP 是 TCP/IP 的核心，也是网络层（Internet Layer）中最重要的协议。IP 可提供基本的分组传输服务，这是构建 TCP/IP 网络的基础。所有的 TCP/IP 数据都流经 IP，不管是出去的还是进来的都与它的最终目的地无关。

1.　网间协议

网间协议的功能包括：

- 定义数据报，它是在 Internet 上的基本传输单元。
- 定义网间寻址方案。
- 在网络访问层和主机对主机传输层之间传输数据。
- 为数据报选择至远程主机的路由。
- 执行数据报的分解和重组。

IP 是一个“无连接协议”，本身不提供错误检测和错误恢复功能。也就是说，IP 协议可以正确地将数据传送到已连接的网络，但是它并不检验数据是否被正确地接收。运

行在 IP 协议之上的协议可以提供错误检测和错误恢复功能，例如 TCP 协议就提供了无错误的有链接网络传输功能。

(1) 数据报

数据报是 IP 协议定义的一种分组格式。如图 12-4 表示一个 IP 数据报，数据报中前 5 个或 6 个 32 位字为控制信息，称为报头。在默认形式下，报头的长度是 5 个字，第 6 个字是可选的。由于报头的长度是可变的，因而它包含一个称为“ Internet 报头长度（IHL）”的字段，以字为单位指出报头的长度。报头包含着传输该分组所需的全部信息。

IP 协议通过检查报头第 5 个字中的目的地址（destination address）传送数据报，该目的地址是一个标准的 32 位 IP 地址，它可以标识目的网络和在该网络上的特定主机。如果目的地址是本地网络中一个主机的地址，该分组就直接传送给目的地；如果目的地址不在本地网络中，该分组就被传送到网关（gateway）再进行传送。

网关是在不同的物理网络之间交换分组报文的设备。确定使用哪个网关称为路由选择（routing），IP 为每个单独的分组做出路由选择决定。

图 12-4　IP 数据报的格式

(2) 数据报的路由选择

Internet 网关通常（或许更精确地说）是指 IP 路由器（router），因为它使用网间协议在网络之间选择分组的路由。在传统的 TCP/IP 术语中，只有两种类型的网络设备，即网关（gateway）和主机（host）。网关可以在网络之间转发分组报文，主机却不能。然而，如果一台主机连接多个网络（称为多地址主机），则可以在网络间转发分组报文。当一个多地址主机转发分组报文时，它的作用与其他任何网关一样，可以看成是一个网关。

目前的数据通信术语有时将网关与路由器区别开，实际上，术语“网关”和“ IP 路由器”是可以互换的。

(3) 数据报的拆分

每一种类型的网络都有一个“最大传输单元（Maximum Transmission Unit，MTU）”，即网络上可以传输的最大分组大小。如果从一个网络上接收到的数据报大于另一个网络的最大传输单元，就必须将它分成较小的“块”才能传输，这一过程称为“拆分（fragmentation）”。如以太网与 X.25 网络在物理上也是不同的，当一个较大的以太网分组在 X.25 网络上传输之前，IP 必须将它分割成较小的分组。

(4) 传送数据报到传输层

当 IP 接收到一个寻址本地主机的数据报时，它必须将该数据报中的数据部分传送给合适的传输层协议，这是利用数据报报头中第 3 个字内的“协议号（ptotocol numnber）”完成的。每个传输层协议都有一个惟一的协议号，用来在 IP 中标识它自己。

2. 寻址、路由选择和多路复用

为了在两个 Internet 主机之间传送数据，必须通过网络将数据传送给相应的主机，

并在该主机内传送给相应的用户或进程。TCP/IP 利用三种方法来完成这些任务：

- 寻址（addressing）：IP 地址可以惟一地标识 Internet 中的每一台主机，它可以将数据传送到相应的主机。
- 路由选择（routing）：网关可以将数据传送到相应的网络。
- 多路复用（multiplexing）：协议和端口号可以将数据传送到主机内相应的软件模块。

其中的每一个功能（在主机之间寻址、在网络之间选择路由和在层间多路复用）对于通过 Internet 在两个协作应用程序间传送数据都是必需的。

(1) IP 地址

IP 协议以数据报的形式在主机之间传输数据，每个数据报传送到一个地址，该地址包含在该数据报报头的目的地址（第 5 个字）中。目的地址是一个 32 位的 IP 地址，它包含着足够的信息以惟一地标识一个网络和该网络中的特定主机。

一个 IP 地址由一个网络部分和一个主机部分组成，但在每个 IP 地址中它们的格式是不同的。用来标识网络和主机的地址位数将根据地址的“类型”而变，A 类、B 类和 C 类是三个主要的地址类型。通过检查一个地址的前几位，IP 软件很快就可以确定地址的类别及其结构。IP 遵循以下规则确定地址的类别：

- 如果 IP 地址的第一位是 0，它就是 A 类网络的地址。A 类地址的第 1 位标识其地址类别，接着的 7 位标识其网络，最后的 24 位标识主机。A 类网络的编号小于 128，但每个 A 类网络可以包含数百万台主机。
- 如果 IP 地址的前 2 位是 10，它就是 B 类网络地址。在 B 类地址中，前 2 位标识地址类别，接着的 14 位标识网络，最后 16 位标识主机。可以有数千个 B 类网络编号，每个 B 类网络可以包含数千台主机。
- 如果 IP 地址的前 3 位是 110，它就是 C 类网络地址。在 C 类地址中，前 3 位是地址类别标识符，接着的 2l 位是网络地址，最后 8 位标识主机。有数百万个 C 类网络编号，而每个 C 类网络包括的主机数量少于 254 台。
- 如果 IP 地址的前 3 位是 111，它就是一个专门保留的地址。这类地址有时称为 ID 类地址，实际上它并不指向特定的网络，目前这一范围内的编号是赋予广播地址。

广播地址用来一次寻址一组计算机，它标识共享同一协议的一组计算机，这与共享同一网络的一组计算机恰好相反。

IP 地址通常写成用点（英语句号）分隔开的 4 个十进制数，其中每一部分的数字值在 0～255（一个字节可表达的十进制值）之间。因为标识类的位和网络地址的位是连在一起的，因而可以把 IP 地址看成是由所有网络地址字节和所有主机地址字节两部分组成。第 1 个字节的值的含义是：

- 如果值小于 128，则表示 A 类地址；其第 1 个字节就是网络号，紧接着的三个字节是主机地址。
- 值在 128～191 之间，表示 B 类地址；前两个字节标识网络，后两个字节标识主机。

- 值在 192～223 之间，表示 C 类地址；前三个字节是网络地址，最后一个字节是主机号。
- 值大于 223，表示该地址是保留的，可以不管保留的地址。

并不是所有的网络地址或主机地址都是可用的。

第一个字节大于 223 的地址都是保留的。

在 A 类地址中，0 和 127 这两个地址也是留作专用地址，网络 0 是“默认路由”，网络 127 是“回送地址”。默认路由用来简化 IP 必须处理的路由选择信息，回送地址由于允许本地主机与远程主机以同样的方式寻址而简化了网络应用程序。在配置主机时使用这些专用网络地址。

在所有的网络中，主机号 0 和 255 也是保留的。所有主机位都置成 0 的 IP 地址用以标识网络本身。主机号为 0 的 IP 地址是广播地址，即发送到该地址的数据传送到网络上的每一台主机。

由于合法 IP 地址的缺乏，设置了一定的保留地址，这些地址决不被正式地分配给任何人，而且决不应该被使用在自己网络的外部机构。

如：A 类网 10

B 类网 172.16～172，31

C 类网 192.168.0～192.168.255

一般将 IP 地址称为主机地址，但实际上 IP 地址是赋予网络接口而不是赋予计算机系统的。

IP 协议栈使用 IP 地址的网络部分作为数据报在网络之间进行路由的依据。当数据报到达目的网络时，它的全部地址（包括主机信息）用来进行最终的目标接收主机定位。

(2) 子网

将主机地址位用作附加的网络地址位，就可以局部地修改 IP 地址的标准结构。其实质就是移动网络地址位和主机地址之间的“分界线”，从而创建附加的网络，但却减少了每个网络的主机数量。这种新分配的网络位就可在一个大型网络内定义一个网络，称为子网（subnet）。

为了解决拓朴上的或组织上的问题，一些组织往往决定组建子网。构建子网可以分散对主机寻址的管理。

建立子网还可解决硬件差异和距离限制问题。IP 路由器可以将不同的物理网络连接在一起，但这只有当每个物理网络具有惟一的网络地址时才可以。构建子网则将一个单独的网络地址分成很多惟一的子网地址，因此每个物理网络可以具有自己惟一的地址。

在 IP 地址上使用一个位掩码（即子网掩码——subnet mask）就可以定义一个子网。如果掩码位是 1，那么其地址中等价位就解释成一个网络位；如果掩码位是 0，则该位就属于主机地址部分。子网只能在本地识别，对于 Internet 的其他部分，其地址仍然被看成是标准的 IP 地址。

例如，与标准 B 类地址相关的子网掩码是 255.255.0.0。最通用的子网掩码是通过一个附加字节来扩充一个 B 类地址的网络部分，这样一来该子网就是 255.255.255.0。前三个字节的所有位都是 1，而最后一个字节的所有位都是 0；前两个字节定义 B 类网络，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

第三个字节定义子网地址，第四个字节定义子网上的主机。

12.1.4 传输层

传输层（Trransport Layer）中两个最重要的协议是传输控制协议（Transmission Control Protocol，TCP）和用户数据报协议（User Datagram Protocol，UDP）。TCP 利用端对端错误检测与纠正功能提供可靠的数据传输服务；而 UDP 提供低开销的无连接数据报传输服务，二者都可以在应用层和网络层之间传输数据。对于特定的应用程序，程序设计者可以选择最适合的服务。

1. 传输控制协议

(1) 用户数据报协议（UDP）

用户数据报协议是一个不可靠的无连接数据报协议，其格式如图 12-5 所示。

图 12-5 UDP 的报文格式

选用 UDP 作为一种数据传输服务的原因有好几个，如果传输的数据量很少，那么为建立连接和确保可靠传输而花费的开销可能比重新传输全部数据的开销还高。在此情况下，UDP 就是传输层协议最好的选择。

使用“查询－响应”方式的应用程序也非常适宜使用 UDP，其响应可以用作对查询的肯定确认，如果在一定的时间内没有收到响应，应用程序便发出另一个查询。

有些应用程序可提供自己的技术去确保可靠的数据传输，而不需要传输层协议的服务。

(2) 传输控制协议（TCP）

TCP 是一种可靠的、面向连接的字节流协议。TCP 提供的可靠性是利用一种称为“重传肯定确认（Positive ACKnowledgment With Retransmission，PAR）”机制来实现的。换句话说，除非一个利用 PAR 的系统接收到从远端系统发来的肯定确认，否则就重发原数据。在相互协作的 TCP 模块之间交换的数据单元称为“段”如图 12-6 所示。

图 12-6 TCP 的报文格式

(3) TCP 的可靠连接

TCP 是一个完控传输协议的典范，采用下列最基本的可靠性保障技术，从而实现其可靠传输的机制。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

- 确认（ACKnowledge）

为了在不可靠网络上提供可靠的数据传输，一种简单的方法是使用确认消息（ACK）。确认方法是指，当网络传送的数据到达接收方时，接收方要向发送方发送确认消息，让发送方知道数据已经到达了接收方。发送方发出数据后，要等待接收方的确认消息，收到确认消息后才再发出下一个数据。

在具体实现的时候，当接收方收到一段数据后，就会回应一个确认消息（ACK），其中包含所收到数据序号+1 的认可号码（ACKnowledgement nuamber）。TCP 在回应确认消息的时候还使用一种夹带（piggy bACKing）技术。夹带技术可以将确认消息附带在由接收方发送到发送方的数据报文中传送给对方，这样可以降低确认消息使用的网络带宽。

TCP 协议为了确保数据传输的正确，使用了必须事先建立连接才能通信的方式，也就是说 TCP 首先需要和通信的对方发送建立连接的消息（建立连接前的握手动作，handshake），然后再传送数据，在数据传送结束后，通信双方还必须执行终止连接的动作。

如图 12-7 所示，TCP 采用三次握手协议来（three-way handshake）建立连接。首先，由客户端向服务器端发出 SYN 消息，表示要求建立 TCP 连接；若服务器端接受建立连接，则回应 SYN/ACK 消息；客户端收到之后再日直 ACK 消息，然后即可开始在客户端和服务器端之间传送消息。

图 12-7　TCP/IP 建立连接的三次握手

当使用 TCP 协议进行通信的双方结束数据传输时，它们就利用包含“无数据发送（FIN）”位的消息来进行类似的三次握手交互，以使双方正确地断开连接。

- 重传（retransmission）

TCP 另一个确保数据正确传输的技术是正向确认与重传（PAR）机制，发送方若在指定时段内未收到另一方对于已送出消息的确认消息（产生这种情况的原因，可能有发送消息丢失及确认消息丢失两种情况），会重新送出相同的数据消息，TCP 将重复尝试数次，直到对方回应确认消息之后再送出下一个数据消息，若重复尝试失败，则 TCP 将通知应用层“连接断开”。

- 序列号（sequence number）

当发送方在发出数据后，若收不到确认消息，则进行重发处理。然而，有时候也会因为某些原因，使得确认消息的到达延迟，数据重发后才收到确认消息。此种情况下，接收方将会接收到重复的数据，为避免因此造成上层应用程序发生混乱，维持正常的连接，必须把重复收到的数据丢弃。为此必须有一种对已收到的数据进行识别，判断其是否为重复数据的机制，TCP 运用序列号方式来实现这一功能。

TCP 发送的是连续的字节流而不是单独的分组。因此要确保发送和接受的顺序，即用 TCP 报文的首部中的“序列号”和“确认号”字段来保持这个顺序。

每个系统可选择任意“序列号”作为起点。但通常情况下是 0（初始序列号称为 Initial Sequence Number，ISN）。

数据中的每个字节是从 ISN 开始顺序编号的，因而被发送数据的第一个实际字节的顺序号为 ISN+1（通常为 1）。

确认消息（ACK）执行肯定确认和流控制两种功能。确认是告诉发送方已经接收了多少数据和接收方还可以接收多少数据。确认号是接收方接收到的最后一个字节的顺序号。该标准并不要求每个分组都要单独确认，确认号就是对在该号之前的所有字节的肯定确认。

窗口字段代表了接收方有能力接收的字节数。

TCP 还负责将从 IP 接收到的数据传送给合适的应用程序。接收该数据的应用程序是用一个 16 位的“端口号”标识的。源端口和目的端口是包含在头部的第一个字节中，使数据正确地传进和传出应用层，这是传输层的一个重要服务。

- 校验和（checksum）

TCP 使用检查校验和的方法来判断数据是否传送正确。校验和位于 TCP 消息的首部，当 TCP 接收到消息的时候，它会将自己依据接收到的消息计算而得到的校验和与 TCP 消息首部的校验和比较，若相同，则送出确认消息，表示接收无误；如果不相同，表示数据已经被破坏了，TCP 忽略该消息，不发送确认消息。发送方在等待确认消息超时后，会再次送来同一个消息。

(4) 流量控制

流量控制是指发送方必须控制自己的发送速度，从而避免过快的发送速度导致接收方来不及接收数据而传输失败。TCP 运用滑动窗（sliding window）的方式进行流量控制。这是因为每个数据报文被发送出去之后不会立即抵达目标，而是会有一段时间的传输过程，为避免因等待接收方的确认消息所造成的网络带宽的闲置，发送方可在未收到前一个数据报文的确认消息的情况下持续发送 n 个数据报文，此处 n 即是所谓的滑动窗长度，如图 12-8 所示。

图 12-8　TCP 之滑动窗

在图 12-8 中，号码为数据报文的序号，发送方已收到数据报文 3、4 的确认报文，也即是接收方已“确认”收到这些数据报文，发送方虽尚未收到 5、6、7 等数据报文的确认报文，但由于尚未超出滑动窗的范围（5～9），故可继续发送数据报文，直到送出第 9 个数据报文后，这时需要停下来等待接收方确认报文。由于在传送过程中，发送方随时可收到接收方的确认报文，故滑动窗的起始位置将不断地往前移动，滑动窗口的名称也来源于此。

2. 协议号、端口号和套接字

一旦数据在网络上传输并到达一台特定的主机，就必须将它交给相应的应用程序或

进程。由于数据在 TCP/IP 的各层之间可上下传送，因此就需要一个机构能将数据传送到每一层的相应协议。系统必须将来自多个应用程序的数据组合到少数几个传输协议中，再从这些传输协议传给网间协议。为此，IP 使用协议号去标识传输协议，而传输协议使用端口号去标识应用程序。

(1) 协议号

协议号是数据报报头的第 3 个字中的一个字节，其值标识 IP 上必须传送数据的那一层协议。

在 Linux 系统中，协议号定义在/etc/protocols 文件中，它是一个简单的表格，含有协议名及其协议号。例如在 Redhat Linux 7.2 中，/etc/protocols 文件的内容如下：

/ etdprotocols：#$Id：protocols，v1，3200I/O7 / 0707：07：15nalinExp$

#

奄 Intern&0 吼 protocols。

啦

#fmml@（#）protocols5，1（Berkeley）4 / 17 / 891

啦

#UpdatedforNetBSDbasedonRFC1340，AssignedNernbers（July1992），

#

#Seealsohttp：/ / www，iana，org / osslgnments / protocol—

/etc/protocols 文件中包含的这个表的含义是，当一个数据报到达，并且它的目的地址与本地 IP 地址符合时，IP 层就知道必须将该数据报传送到它上面的一个传输层协议。为了决定哪个协议接收该数据报，IP 就查看该数据报的协议号。利用此表可以看出，如果协议号是 6，IP 就将该数据报传送给 TCP；如果是 17，IP 就将它传送给 UDP。

(2) 端口号

IP 协议栈将进来的数据发送给传输协议后，该传输协议就将它传送到相应的应用程序进程中。应用程序的进程（又称网络服务）是用端口号标识的，它是一个 16 位的值。标识数据发送进程的“源端口号”和标识数据接收进程的“目的端口号”都包含在每个 TCP 段和 UDP 分组的第 1 个报头字中。

在 Linux 系统中，端口号在/etc/services 文件中定义。网络应用程序的数量要比该表中所示的传输层协议数多得多。低于 256 的端口号是留给“知名服务”的（例如 FTP、TELNET 等著名的应用程序），256～1024 的端口号用于 Unix 的专用服务，通常用户可以使用 1024 以上的端口号。。下面列出了 Redhat Linux 7.2 中部分/etc/services 文件的内容：

/ etdservices：#$Id：services，v1，22200I/O7 / 1920：13：27nottingExp$##Networkservices，Internetstyle##NotethatitispresentlythepolicyofIANAtoassignasinglewell—known#portnumberforbothTCPandUDP；hence，mostentriesherehavetWOentries#eveniftheprotocoldoesn ' tsupportUDPoperati ' ons，#UpdatedfromRFCl700，' 'AssignedNumbers'’（Octoberl994），Notallports#&reinduded，

onlythemoreC_QVtlnlotlO[1es，珏#ThelatestIANAportassignmentsearlbegotlenfrom#http：／／www，iana，0 耐 assignments／port—numbers#TheⅥ出 KnownPortsarethosefrom0thmtl

将/etc/services 文件中的表与/etc/protocols 文件中的表结合在一起，就可提供将数据传送到相应的应用程序所需的全部信息。数据报根据其报头第 5 个字内的目的地址抵达其目的地，IP 使用该数据报报头第 3 个字内的协议号将数据传输给适当的传输层协议。到达该传输协议的数据的第 1 个字内含有目的端口号，它告诉传输协议将数据传送到特定的应用程序。

(3) 套接字

一个 IP 地址和一个端口号的组合称为一个“套接字（Socket）”，一个套接字可以惟一地标识整个 Internet 中的一个网络进程。“套接字”是 IP 地址和端口号的组合，一对套接字（一个用于接收主机，另一个用于发送主机）可定义面向连接协议（如 TCP）的一次连接。

12.1.5 应用层

应用层（Application Layer）中包含了使用传输层协议去传输数据的所有协议，应用层协议很多，一些著名的应用层协议有：

TELNET：这是网络终端执议，可通过网络提供远程登陆。

FTP：这是文件传输协议，可用于交互式文件传输。

SMTP：这是简单邮件传输协议，可用于传送电子邮件。

此外常用的协议还包括：

域名服务（Domain Name Service，DNS）：可以将 IP 地址映射成赋予网络设备的名字。

路由信息协议（Routing Information Protocol，RIP）：路由选择是 TCP/IP 的工作核心，网络设备使用 RIP 去交换路由选择信息。

网络文件系统（Network File System，NFS）：该协议允许文件被网络上的各种主机共享。

综上所述，FTP、TELNET 和 SMTP 基本上是依赖于 TCP 的，而 NFS、DNS 和 RIP 则基本上依赖于 UDP。一些应用程序型协议，如外部网关协议（Exterior Gateway Protocol，EGP），它是另一个路由协议，它们不使用传输层服务，而直接使用 IP 服务。

12.2 环形缓冲区

在通信程序中，经常使用环形缓冲区作为数据结构来存放通信中发送和接收的数据。环形缓冲区是一个先进先出的循环缓冲区，可以向通信程序提供对缓冲区的互斥访问。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

12.2.1 环形缓冲区的实现原理

环形缓冲区通常有一个读指针和一个写指针。读指针指向环形缓冲区中可读的数据，写指针指向环形缓冲区中可写的缓冲区。通过移动读指针和写指针就可以实现缓冲区的数据读取和写入。在通常情况下，环形缓冲区的读用户仅仅会影响读指针，而写用户仅仅会影响写指针。如果仅仅有一个读用户和一个写用户，那么不需要添加互斥保护机制就可以保证数据的正确性。如果有多个读写用户访问环形缓冲区，那么必须添加互斥保护机制来确保多个用户互斥访问环形缓冲区。

图 12-9、图 12-10 和图 12-11 是一个环形缓冲区的运行示意图。图 12-9 是环形缓冲区的初始状态，可以看到读指针和写指针都指向第一个缓冲区处；图 12-10 是向环形缓冲区中添加了一个数据后的情况，可以看到写指针已经移动到数据块 2 的位置，而读指针没有移动；图 12-11 是环形缓冲区进行了读取和添加后的状态，可以看到环形缓冲区中已经添加了两个数据，已经读取了一个数据。

图 12-9　环形缓冲区（初始状态）

图 12-10　环形缓冲区（添加数据）

图 12-11　环形缓冲区（添加和读取）

12.2.2 实例：环形缓冲区的实现、。

环形缓冲区是数据通信程序中使用最为广泛的数据结构之一，下面的代码，实现了一个环形缓冲区：

```
/* ringbuf.c */
#include <stdio.h>
#include <ctype.h>

#define NMAX 8

int iput = 0;   /*  环形缓冲区的当前放入位置  */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
int iget = 0;   /* 环形缓冲区的当前取出位置 */
int n = 0;   /* 环形缓冲区中的元素总数量 */

double bufer[NMAXl];

/*
环形缓冲区的地址编号计算函数，如果到达唤醒缓冲区的尾部，将绕回到头部。
环形缓冲区的有效地址编号为：0 到(NMAX-1)
*/

int addring(int i)
{
    return (i+1) == NMAX ? 0 : i+1;
}

/* 从环形缓冲区中取一个元素 */
double get(void)
{
    int pas;

    if (n > 0) {
        pos =   iget;
        iget = addring(iget);
        n--;
        return buffer[pos];
    }
    else {
        printf("Buffer is empty \n");
        return 0.0;
    }
}

/* 向环形缓冲区中放入一个元素 */
void put( double z)
{
    if (n < NMAX) {
        buffer[iput] = z;
        iput = addring(iput);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        n++;
    }
    else
        printf("Buffer is full \n");
}

int main(void)
{
    char opera[5];
    double z;

    do {
        printf('Please input p|g|e ?");
        scanf("%s", &opera);

        switch (tolower(opera[0])) {
            case 'p': /* put */
                printf("Please input a float number?");
            scanf("%lf", &z);
                put(z);
                break;
            case 'g': /* get */
                z = get();
                printf("%8.2f from Buffer \n", z);
                break;
            case 'e':
                printf("End \n");
                break;
            default:
                printf("%s - Operation command error! \n", opera);
        } /* end switch */

    } while (opera[0] != 'e');

    return 0;
}
```

上述代码使用如下的命令编译：

gcc ringbuf.c -o ringbuf.exe

编译后生成的 ringbuf.exe，运行效果如图 12-12 所示。

图 12-12　环形缓冲区的运行效果

12.3　通信协议中的定时器编程

在通信协议栈中，定时器是不可缺少的一个功能。例如在停等协议中，发送方发送了一个数据报文后，需要启动等待确认报文定时器。如果在等待确认报文定时器超时后仍然没有接收到接收方的确认报文，那么发送方就会重发刚才的报文。本节将介绍如何在通信程序中实现定时器的功能。

在 IBM PC 中，有一个定时产生的中断——定时中断 0x0C。在默认状态下，中断 0x0C 大约每秒钟发生 18.2 次。在通信程序设计中，如果需要定时执行一些操作，可以改写 0x0C 中断来实现，这是实现定时器最简单的方式。

12.3.1　基本定时编程

本节以 Turbo C/Borland C++为编译器来介绍如何使用中断 0x0C 实现定时器功能。对于其他的编译器，实现的基本原理是一致的。

在 Turbo C/Borland C++中，提供了两个函数 getvect()和 setvect()，通过这两个函数可以获取和设置 IBM PC 的中断向量。

两个函数的声明如下：

```
void interrupt(*getvect(int interruptno))();
void setvect(int interruptno, void interrupt(*isr)());
```

保留字 interrupt 指示函数是一个中断处理函数。在调用中断处理函数时，所有的寄存器将会被保存。中断处理函数返回时的指令是 iret，而不是一般函数用到的指令 ret。

getvect()根据中断号 interruptno 获取中断号为 interruptno 的中断处理函数的入口地址。setvect()将中断号为 interruptno 的中断处理函数的入口地址改为 isr()函数的入口地址，即中断发生时，将调用 isr()函数。

通常，在程序开始时截获时钟中断 0x0C，并设置新的中断处理，在程序结束时，恢复时钟中断，这样就可以使用时钟 0x0C 来实现一个基本的定时器。下面的代码是使用 getvect()和 setvect()函数来设置时钟中断 0x0C 中断处理程序的例子：

```
/* timer.c */
/*  本程序使用 Borland C++或者 Turbo C 编译  */
#include <stdio,h>
#include <stdlib.h>
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#include <dos.h>
#include <bios.h>
#include <conio.h>

/* Escape key */
#define VK_ESC 0x11b
#define TIMER 0x1c   /* 时钟中断的中断号 */
/* 中断处理函数在 C 和 C++中的表示略有不同。
如果定义了__cplusplus 则表示在 C++环境下，否则是在 C 环境下。 */
intTimerCounter = 0;   /* 计时变量，每秒钟增加 18 */
/* 指向原来时钟中断处理过程入口的中断处理函数指针（句柄） */
void interrupt (*oldhandler)();

/*新的时钟中断处理函数 */
void interrupt newhandler()
{
      /* increase the global counter * /
      TimerCounter++ ;
      /* call the old routine */
      oldhandler( );
}
/*设置新的时钟中断处理过程 */
void SetTimer(void interrupt(*IntProc)())
{
      oldhandler = getvect(TIMER);
      disable();   /* 设置新的时钟中断处理过程时，禁止所有中断 */
      setvect(TIMER, IntProc);
      enable();   /*开启中断 */
}
/* 恢复原有的时钟中断处理过程 */
void KillTimer()
{
      disable();
      setvect(TIMER, oldhandler);
      enable();
}
void main(void)
{
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    int key, time = 0;
    SetTimer(newhandler);   /* 修改时钟中断 */
    for(;;)
    {
        if (bioskey(1))
        {
            key=bioskey(0);
            if (key == VK_ESC)   /* 按 Esc 键提前退出程序 */
            {
                printf("User cancel! \n");
                break;
            }
            if (TimerCounter > 18)   /* 1 秒钟处理一次 */
            {
                /* 恢复计时变量 */
                TimerCounter = 0;
                time++;
                printf("%d\n",time);
                if (time ==10)   /* 10 秒钟后结束程序 */
                {
                    printf("Program terminated normally! \n");
                    break;
                }
            }
        }
    }
    KillTimer();   /* 恢复时钟中断 */
}
```

上述程序使用 Turbo C/Borland C++编译后，运行效果如图 12-13 所示。

图 12-13　定时器程序的运行效果

12.3.2　高精度定时中断编程

本节将介绍如何直接通过控制 8253 定时器芯片来实现高精度的定时器。本节介绍的方法更适合在操作系统核心中使用，因为在自己实现的操作系统核心中不能设置时钟中

断 0x0C 来实现定时器。

IBM PC 机采用一片 8253 定时器芯片计算系统时钟的脉冲，若干个系统时钟周期转换成一个脉冲，这些脉冲序列可以用以计时，也可以送入计算机的扬声器产生特定频率的声音。8253 定时器芯片独立于 CPU 运行，它可以像实时时钟那样，CPU 的工作状态对它没有任何影响。

8253 芯片有三个独立的通道，每个通道的功能各不相同，三个通道的功能如下：

- 通道 0：为系统时钟所用，在启动时由 BIOS 置入初值，每秒钟约发出 18.2 个脉冲，脉冲的计数值存放在 BIOS 数据区的 0040:006C 存储单元中（通过读取这个单元的内容，可以获得计算机系统逝去的时间）。通道 0 的输出脉冲作为申请定时器中断的请求信号，还用于磁盘的某些定时操作，如果改变了通道 0 的计数值，必须确保在 CPU 每次访问磁盘以前恢复原来的读数，否则将使磁盘读写产生错误。
- 通道 1：用于控制计算机的动态 RAM 刷新速率，一般情况下不要去改变它。
- 通道 2：连接计算机的扬声器，产生单一的方波信号控制扬声器发声。

8253 定时器芯片的每一个通道含有 3 个寄存器，CPU 通过访问 3 个端口（通道 0 为 40h，通道 1 为 41h，通道 2 为 42h）来访问各个端口的 3 个寄存器，8253 每个端口有 6 种工作模式，当通道 0 用于定时或通道 2 用于定时或发声时，一股用模式 3。在模式 3 下，计数值被置入锁存器后立即复制到计数器，计数器在每次系统时钟到来时减 1，减至 0 后一方面马上从锁存器中重新读取计数值，另一方面向 CPU 发出一个中断请求（INT 1Ch 中断，很有用），如此循环在输出线上高低电平的时间各占计数时间的一半，从而产生方波输出。

对 8253 定时器芯片编程是通过命令端口寄存器 43h 来实现，它决定选用的通道、工作模式、送入锁存器的计数值是一字节还是两字节、是二进制码还是 BCD 码等工作参数，端口 43h 各位的组合形式如表 12-1 所示。

表 12-1　端口 43h 各位组合形式

位	含义
位 0	若为 1 则采用二进制表示，否则用 BCD 码表示计数值
位 3～1	工作模式号，其值（0～5）对应 6 种模式
位 5～4	操作的类型： 00：把计数值送入锁存器 01：读写高字节 10：读写低字节 11：先读写高字节，再读写低字节
位 7～6	决定选用的通道号，其值为 0～2

对 8253 芯片编程的步骤如下：

(1)　设置命令端口 43h；

PDF-XCHANGE Click to buy NOW! www.docu-track.com

(2) 向端口发送一个工作状态字节；

(3) 确定定时器的工作方式。

若是通道 2，给端口 61h 的第 0 位和第 1 位置数，启动时钟信号，当第 l 位置 1 时，通道 2 驱动扬声器，置 0 时用于定时操作；将一个字的计数值，按先低字节后高字节的顺序送入通道的 I/0 端口寄存器（通道 0 为 40h，通道 1 为 41h，通道 2 为 42h）。当第(3)步完成后被编程的通道马上在新的状态下开始工作。由于 8253 的三个通道都独立于 CPU 运行，所以在程序结束以前要恢复各通道的正常状态值。

如果在程序中只使用通道 0 时，首先需要确定放入锁存器的 16 位的计数值：

16 位的计数值 = 1.19318MB / 希望的频率，其中 1.19318MB 是系统振荡器的频率。

由上式可知，计数器所能产生的值是 18.2Hz～1.19318MHz。下面的代码是对 8253 定时器芯片通道 0 编程的函数：

```
#define T60HZ 0x4dae
#define T50HZ 0x5d37
#define T40HZ 0x7468
#define T30HZ 0x965c
#define T20HZ 0xe90b
#define T18HZ 0xffff
#define LOW_BYTE(n) (n&0x00ff)
#define HI_BYTE(n) ((n>>8)&0x00ff)
int far *clk = (int far *)0x0000046c;

/* 1. 改变时间定时器值的函数 */
void ChangTime(unsigned cnt)
{
    outportb(0x43, 0x3c);
    ouiportb<0x40, LOW_BYTE(cnt));
    outportb(0x40, HI_YTE(cnt));

}

/* 2. 延时函数 */
void Delay (int d)
{
    int tm = *clk;
    while(*clk -tm < d);
}
```

在上面的代码中，指针*clk 指向 BIOS 数据区的 0040:006C 存储单元，该单元中存放着定时器的计数值，可以根据该单元的内容计算差值来达到延时的目的。

12.4 简单停等协议

在网络通信中，保证通信可靠性是一个非常重要的问题。因为通信的物理链路可能存在错误，因此通信双方发送的数据都可能在通信的过程中丢失。为了保证通信的双方在可能存在错误的通信链路上可靠地传输数据，就必须使用特别的通信协议来进行保证。

12.4.1 停等协议简介

在通信协议中，停等协议是最简单的可靠传输协议。如图 12-14 所示，在停等协议中，发送方在传送一个数据单元后，要等待对方列这个数据单元的接收确认。接收方在接收了一个数据单元后，会向发送方发送一个确认，表示自己已经正确地接收到了数据。接收方也可以发送一个否认的确认来告诉发送方自己没有接收到一个数据单元。

图 12-14　停等协议示意图

在停等协议的实现中，接收方通常难以区分发送方是否已经发送了一个数据单元，接收方因此也难以在合适的时候发送否认数据单元。因此，发送方通常都维护着一个定时器，如果发送方在发送了数据单元后，在设定的时间内没有接收到接收方的确认，将重新发送数据单元。

在通常情况下，当发送方发送了一个数据单元后将很快地接收到接收方的确认，从而可以发送下一个数据单元。在数据链路延时很长的情况下，发送方在发送了数据单元后必须长时间地等待接收方的确认，这将导致通信效率很低。因此，停等协议是一种简单的可靠协议，但是在某些情况下，通信效率较低。

停等协议的运行必须要求物理链路是可以双向传输数据。在图 12-14 中，实线表示数据单元由发送方向接收方发送，虚线表示接收方向发送方发送正确接收到数据单元的确认报文。

在数据单元传输的过程中，如果数据单元丢失了，接收方通常没有办法获得数据单元丢失的信息。实际上，如果数据单元在传输过程中丢失了，那么接收方实际上是什么报文也接收不到。因此，要确保协议的正确运行，必须依赖于发送方启动的重发定时器。

如图 12-15 所示，发送方的数据报文在传输的过程中出现了错误，报文并没有丢失，但是报文的数据出现了错误，因此接收方接收到了一个错误的报文。接收方接收到错误的报文后，可以通过报文的校验和检查出报文出现了错误。对于发送方而言，发送方并不能感知到数据报文在传送的过程中发生了错误，但是发送方在发送完数据报文后，会启动一个定时器。正常情况下，在这个定时器超时之前，接收方会将代表正确接收到数据报文的确认报文发回发送方。如果发送方接收到正确的确认报文，就会停止定时器，然后进入发送下一个数据报文的过程中。

在图 12-15 所示的过程中，数据报文在传输过程中丢失了，因此接收方不会向发送方发送数据确认报文，接收方在定时器超时之前也无法接收到正确的确认报文。因此，发送方的定时器会超时，从而触发发送方的重传功能，发送一个和上一次发送完全一样的数据报文给接收方。当第二次发送了数据报文后，发送方仍然会启动一个定时器来等待接收方的确认报文。

如果第二次发送的数据报文正确地传送到接收方，接收方然后向发送方发送确认报文，发送方也接收到了确认报文，发送方就停止定时器，代表这个报文正确发送了。

图 12-15　停等协议：发送方重传

12.4.2　有限状态机

图 12-16 是停等协议的有限状态机，这个有限状态机描述了发送方和接收方的状态变化过程。

图 12-16　停等协议的有限状态机

在停等协议中，发送方每发送一个报文后就等待一个确认报文。只有当接收到确认报文之后，发送方才发送下一个报文。这种发送和等待交替的过程不断重复，直到发送方发送一个传输结束报文为止。

停等协议的优点在于简单：在下一个报文发送之前每一个报文都需要接收方进行确认。缺点是效率低：停等协议是很慢的。在发送下一个报文之前，每一个报文必须到达接收方，而每一个报文的确认报文也必须从接收方传输回来。也就是说，在传输线路上总是只有一个报文。如果设备之间的传输线路距离很长，在每个报文之间等待确认报文所花费的时间也将很长。

12.4.3　差错控制

在停等协议中，为了进行差错控制，停等协议还必须具备以下要求：

(1) 发送方在收到最近报文的确认报文之前必须保留该报文的备份。保留备份使得发送方可以重新发送丢失或损坏的报文，直到被正确接收为止。

(2) 为识别各报文，数据报文和应答报文都必须交替地标识为 0 和 1。一个标号为 1 的数据报文被一个标号为 0 的确认报文所确认，说明当前接收方已经接收了数据报文 1，期望接收数据报文 0。如果接收方收到了两个相邻的数据报文且标号相同，说明接收方收到了一个重复报文，应当丢弃一个重复报文。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

(3) 如果在数据报文中发现一个错误，说明该报文在传输中发生了错误，就会返回一个否定确认报文（NAK）。否定确认报文是不编号的，它通知发送方重新发送最近的一个报文。

(4) 发送方安装一个定时器。如果在规定时间内不能收到期望的确认报文，发送方就假定最近一个报文已经在传输中丢失并再次发送它。当重传 n 次后仍然失败，那么就表示通信链路已经断开了。

12.5 实例：串口数据通信

在 IBM PC 机中，通常都配置有串行通信口。现在的 PC 机一般至少配置有两个串行通信口：COM1 和 COM2。串行通信口的数据和控制信息是一位接一位串行地传送，通信速度较慢，但是传送距离较长，因此串行通信适合长距离的通信。

串行通信方式是计算机间最常使用的通信方式之一。本节将介绍一个简单的串行通信程序，具体的代码如下：

```
/* se.c */
#include <stdio.h>
#include <bios.h>
#include <conio,h>

#define COM_INIT       0
#define COM_SEND       1
#define COM_RECEIVE 2
#define COM_STATUS    3

#define COM1              0
#define DATA_READY    0x100
#define TRUE              1
#define FALSE             0

#define SETTINGS (0x80 | 0x02 | 0x00 | 0x00)
/*  设置串口为 1 个停止位，无奇偶校验，1200 波特  */

#define PRINT_SEND_CHAR(ch) printf("send \t\t%c\n", ch)
#define PRINT_RECV_CHAR(ch) printf("receive \t\t%c\n", ch)
int main(int argc, char *argv)
{
    int status, DONE = FALSE;
    char ch;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/* 对 COM1 进行初始化设置 */
bioscom(COM_INIT, SETTINGS, COM1);

printf("...BIOSCOM Demo, any key to exit...\n");
printf("-= UESTC =-\n");

if (argc != 1)
{
    printf("This is client! I will send a char at first :) \n");
    bioscom(COM_SEND, 'a', COM1);
    PRINT_SEND_CHAR('a');
}

while (!DONE)
{
    status = bioscom(COM_STATUS, 0, COM1);
    /* 查询串口状态 */

    if (status & DATA_READY)   /* 有数据到达 */
    {
        if ((ch = bioscom(COM_RECEIVE, 0, COM1) & 0x7F) != 0)
        {
            PRINT_RECV_CHAR(ch);

            if(ch == z) ch = 'a';
            else ch = ch + 1;

            bioscom(COM_SEND, ch, COM1);
            PRINT_SEND_CHAR(ch);
        }
}
    if(kbhit( ))
    {
        DONE = TRUE;
    }
}
return 0;
```

```
}
```

上面的程序编译后生成 sc.exe，运行效果如图 12-17（客户端）和图 12-18（服务器端）所示。

图 12-17　串行通信客户端

图 12-18　串行通信服务器端

12.6　实例：停等协议的实现

停等协议是最简单的无差错通信协议，本节将介绍一个基于串行通信的停等协议的实现代码。这个例子程序基于两台计算机之间的串口进行通信，实现了停等通信的过程，具有很大的实用性。

程序编译后可以生成可执行程序 sw.exe，然后可以使用如下的方法来运行通信的双方：

```
sw s_s.dat s_r.dat sdl rdl 400
sw r_s.dat r_r.dat rdl sdl 500
```

在上面的命令中，s_s.dat、s_r.dat 分别是发送方的发送文件和接收文件，r_s.dat 和 r_r.dat 分别是接收方的发送文件和接收文件。发送和接收双方的信息，将保存在这四个文件中。

程序具体的代码如下：

```
/* sw.c */
/*
使用停等协议来进行通信的例子，程序运行方法如下：
sw s_s.dat s_r.dat sdl rdl 400
sw r_s.dat r_r.dat rdl sdl 500
400、500 的延时很重要，如果设置太小，会导致信道忙，传输反而慢。
设置太大，那么也会传输太慢。
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
#include <dos.h>
#include <io.h>
#define true 1
#define false 0

#define FILE_WRITE_PERMISSION 2
#define FILE_READ_PERMISSION    4

char PHYSICAL_DATA_LINK_FILE_WRITE[12];
char PHYSICAL_DATA_LINK_FILE_READ[12];

enum protocol_event {
        PACKET_READY,
        FRAME_ARRIVAL,
        TIME_OUT,
        NO_EVENT,
        ChannelIdle,
        CheckSumError
};

typedef char packet;

struct frame    /* 帧结构 */
{
        packet info;
        int seq;
        int ack;
};

packet buffer;

int NextFrameToSend;
int FrameExpected;
int PacketLength;
int waiting;
int network_layer_flag;

FILE *fps, *fpr;
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
int fps_ length;

#define INTR 0X1C    /* The clock tick interrupt */

int init_timer_value;    /* a second is 17.2 tick */

void interrupt (*oldhandler)() ;

int timer_count = 0;
int timer_run_flag = false;
int delay_time;

void interrupt handler( )
{
      if (timer_run_flag)
      {
            timer_count--;
      }
      oldhandler();    / * call the old routine */
}

char *events_string(enum protocol_event e, char *e_str)
{
      switch (e)
      {
            case PACKET_READY:
                  strcpy(e_str, "PACKET_READY');
                  break;
            case FRAME_ARRIVAL:
                  strcpy(e_str, "FRAME_ARRIVAL");
                  break;
            case TIME_OUT:
                  strcpy(e_str, "TIME_OUT");
                  break;
            case NO_EVENT:
                  strcpy(e_str, "NO_EVENT");
                  break;
            case ChannelIdle:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                strcpy(e_str, "ChannelIdle");
                break;
            case CheckSumError:
                strcpy(e_str, "CheckSumError");
                break;
            default:
                strcpy(e_str, "Unknow Event");
                break;
        }
        return e_ str;
}

void initialize(char *send_data, char #recv_data, char *send_dl, char *recv_dl, char * dt)
{
        fps = fopen(send_ data, "rb");
        if(!fps)
        {
            printf("Open file %s to read error!\n", send_ data);
            exit(1);
        }
        fseek(fps, 0L, SEEK_ END);
        fps_length = ftell(fps);
        fseek(fps, 0L, SEEK_SET);

        fpr = fopen(recv_ data, "wb");
        if(!fpr)
        {
            printf("Open file %s to write error!\n", recv_ data);
            exit(1);
        }

        strcpy(PHYSICAL_DATA_LINK_FILE_WRITE, send_ dl);
        strcpy(PHYSICAL_DATA_LINK_FILE_READ, recv_dl);

        delay_time = atoi(dt);   /* 设为微秒（μs） */
        init_timer_value = ((((float)delay_time) / 1000.0)   * 17.2) * 2;
        /* init_timer_value 设置为两倍的传输时间!!   */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        / * at first, the physical link is idle */
        unlink(PHYSICAL_DATA_LINK_FILE_WRITE);
        unlink(PHYSICAL_DATA_LINK_FILE_READ);

        /* save the old interrupt vector */
        oldhandler = getvect(INTR);

        /* install the new interrupt handler */
        setvect(INTR, handler);

        NextFrameToSend = 0;
        FrarneExpected = 0;
        PacketLength = 0;
        waiting = false;
        network_layer_flag = true;
}

void cleanup( )
{
        fclose(fps);
        fclose(fpr);

        /* when exit, set the state of physical link to idle */
        unlink(PHYSICAL_DATA_LINK_FILE_WRITE);
        unlink(PHYSICAL_DATA_LINK_FILE_READ):

        /* reset the old interrupt handler */
        setvect(INTR, oldhandler);
}

void starts_timer(void)
{
        timer_count = init_timer_value;
        timer_run_flag = true;
}

void stop_timer(void)
{
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
        timer_run_flag = false;
}

int checks_timer_expire(void)
{
        if((timer_count <= 0) && (timer_run_flag))
                return 1;
        else
                return 0;
}

void enable_network_layer(void)
{
        network_layer_flag = true;
}

void disable_network_layer(void)
{
        network_layer_flag = false;
}

void get_packet_from_network(packet *buffer, int *PacketLength)
{
        (*buffer) = (packet)fgetc(fps);
        (*PacketLength) = sizeof(packet);
}

void put_packet_to_network(packet pk)
{
        fputc((int)pk, fpr);
        fflush(fpr);
}

void send_frame_to_physical(struct frame fr)
{
        FILE *fp;

        /* 打开一个二进制文件，如果文件中原来有内容，则删除原内容 */
```

```
        fp = fopen(PHYSICAL_DATA_LINK_FILE_WRITE, "wb");
        if(!fp)
        {
            printf("Open data link file %s to write error!\n",
PHYSICAL_DATA_LINK_FILE_WRITE);
            return;
        }
        fwrite(&fr, sizeof(struct frame), 1, fp);
        fclose(fp);
    }

    void get_frame_from_physical(struct frame * fr)
    {
        FILE *fp;

        /* 设置帧为无效 */
        fr->seq = -255;
        fr->ack = -255;

        fp = fopen(PHYSICAL_DATA_LINK_FILE_READ, "rb");
        if(! fp)
        {
            printf("Open data link file %s to read error!\n",
PHYSICAL_DATA_LINK_FILE READ);
            return;
        }
        fread(fr, sizeof(struct frame), 1, fp);
        fclose(fp);

        /* 文件不能被链接，因为此帧已读过了 */
        unlink(PHYSICAL_DATA_LINK_FILE_READ);
    }

    enum protocol_event get_event_from_physical()
    {
        #define FILE_EXIST_FLAG     0
        #define FILE_READ_FLAG      4
        if(access(PHYSICAL_DATA_LINK_FILE_READ, FILE_READ_FLAG) == 0)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        return FRAME_ARRIVAL;
    else
        return NO_EVENT;
}

int channel_idle(void)
{
    return true;    /*  设物理通道总是空闲  */
}

enum protocol_event get_event_from_timer( )
{
    if( check_timer_expire( ))
    {
        stop_timer();
        return TIME_OUT;
    }
    else
        return NO_EVENT;
}

/* Main protocol service routine */
void stop_wait_protocol_datalink(int event)
{
    struct frame send_frame, recv_frame, ack_frame;
    int PacketLength;

    switch(event)
    {
        case PACKET, READY :
            /* a packet has arrived from the network layer */
            get_packet_from_network(&buffer, &PacketLength);
            send_frame.info = buffer;    / * place packet in frame #/
            send_frame.seq = NextFrameToSend;

            /* piggyback acknowledge of last frame received */
            send_frame.ack = 1 - FrameExpected;
            if (channel_idle())
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                {
                    send_frame_to_physical(send_frame);   /* 送到物理层 */
                    start_timer( );
                    printf("Send DATA frame |packet = '%c', seq = %d, ack = %d| to
physical\n", send_frame.info, send_frame.seq, send_frame.ack);
                    waiting = false;
                }
                else
                    waiting = true;

                disable_network_layer();
                break;
            case FRAME_ARRIVAL:
            /* 物理层的帧到达 */
                get_frame_from_physical(&recv_frame);   /* 收到帧 */
                printf("Get a frame |packet = '%c', seq = %d, ack = %d| from physical\n",
recv_frame.info, recv_frame.seq, recv_frame.ack) ;
                / * check that it is the one that is expected * /
                if (recv_frame.seq == FrameExpected)   /* 有效帧到达，送到网络层 */
                {
                    put_packet_to_network(recv_frame.info);   /* valid frame */
                    printf("Put a packet to network |packet = %c|.....\n",
recv_frame.info);
                    FrameExpected = 1 - FrameExpected;

                    /* 发送确认帧 */
                    ack_frame.info = '\0';
                    ack_frame.seq = -1;
                    ack_frame.ack = recv_frame.seq;
                    if (channel_idle())   /* 确认刚收到的帧 */
                    {
                        send_frame_to_physical(ack_frame);   /* 送到物理层 */
                        printf("Send ACK frame |seq = %d, ack = %d|\n",
ack_frame.seq, ack_frame.ack);
                    }
                }
                if(recv_frame.ack == NextFrameToSend)   /* 确认到达，可以发送下一
帧 */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                {
                    stop_timer();   /* 停止重发定时器 */

                    enable_network_layer();
                    NextFrameToSend = 1 - NextFrameToSend;
                }
                break;
            case TIME_OUT:
            /* 在设定的时间内没有收到接收方的确认 */
                /* 重新发送一帧 */
                send_frame.info = buffer;
                send_frame.seq = NextFrameToSend;
                send_frame.ack = 1 - FrameExpected;
                send_frame_to_physical(send_frame);
                start_timer( ) ;
                printf("Send DATA frame |packet = '%c', seq = %d, ack = %d| to
physical\n", send_frame,info, send_frame.seq, send _frame.ack) ;
                break;
            case ChannelIdle:
                if (waiting )
                {
                    send_frame.info = buffer;
                    send_frame.seq = NextFrameToSend;
                    send_frame.ack = 1 - FrameExpected;
                    send_frame_to_physical(send_frame);   /* 送到物理层 */
                    start_timer( ) ;
                    waiting = false;
                }
                break;
            case CheckSumError:   /* 丢失 */
                break;
        }
    }

    enum protocol_event get_event_from_network()
    {
        if(!network_layer_flag)
            return NO_EVENT;
```

```
    if(ftell(fps) < fps_length)
        return PACKET_READY;
    else
        return NO_EVENT;
}

/*
arg1  是网络发送数据单元（发送文件名）
arg2   是网络接收数据单元（接收文件名）
*/
int main(int argc, char *argv[])
{
    enum protocol_event event;
    char ev[20];
    unsigned int count;

    if(argc != 6)
    {
        printf("Usage: %s send_data recv_data send_dl recv_dl delay_ time", argv[0]);
        exit(0);
    }

    initialize(argv[1], argv[2], argv[3], argv[4], argv[5]);

    count =1;
    printf("Protocol begin, any key to exit!\n\n");
    while(true)
    {
        printf("%d \n", count++);

        event = get_event_from_network();
        printf("Event from network = %s\n", event_string(event, ev));
        stop_wait_protocol_datalink(event);

        event = get_event_from_physical();
        printf("Event from physical = %s\n", event_string(event, ev));
        stop_wait_protocol_datalink(event);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            event_get_event_from_timer();
            printf("Event from timer = %s, timer_count =    %d\n", event_string(event, ev),
timer_count);

            stop_wait_protocol_datalink(event);

            delay(delay_time);

            if(kbhit()) break;
        }
        cleanup ( );

        return 0;
    }
```

上述停等协议的运行效果如图 12-19、图 12-20 和图 12-21 所示。

图 12-19 显示了发送前的数据文件，s_s.dat 是接收方的发送数据文件，r_s.dat 是接收方的发送数据文件（假定发送方和接收方都向对方发送数据）。

图 12-20 是发送方和接收方的发送和接收数据过程。

图 12-19　停等协议运行前的文件

图 12-20　停等协议的运行过程

图 12-21 显示了程序运行结束后，发送方和接收方接收到的数据文件情况，可以看到发送方和接收方都正确地接收到了数据文件。

图 12-21　停等协议运行后的文件

12.7　本章小结

网络是 Linux 的强大功能之一。本章介绍了标准 TCP/IP 协议，通过对 TCP/IP 协议的了解，可以让读者奠定理解 Linux 网络协议实现的基础。

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

本章介绍了环形缓冲区这一最常用的网络通信数据结构，提供了环形缓冲区的实现代码。这些代码可以直接使用在具体的网络通信程序中。

本章介绍了如何通过 C 语言的 API 函数来实现定时器功能，同时也介绍了利用 PC 计算机的硬件来实现高精度的定时器功能。这些实现代码和方法可以直接用在操作系统中实现定时器功能。

本章最后介绍了两种具体的数据通信协议的实现方法：①最简单的串行通信；②最简单的无错通信方式——停等协议。本章介绍了这两种通信协议的基本原理，也提供了实现代码。两种通信协议都具有很大的实用价值，可以在操作系统中直接使用。