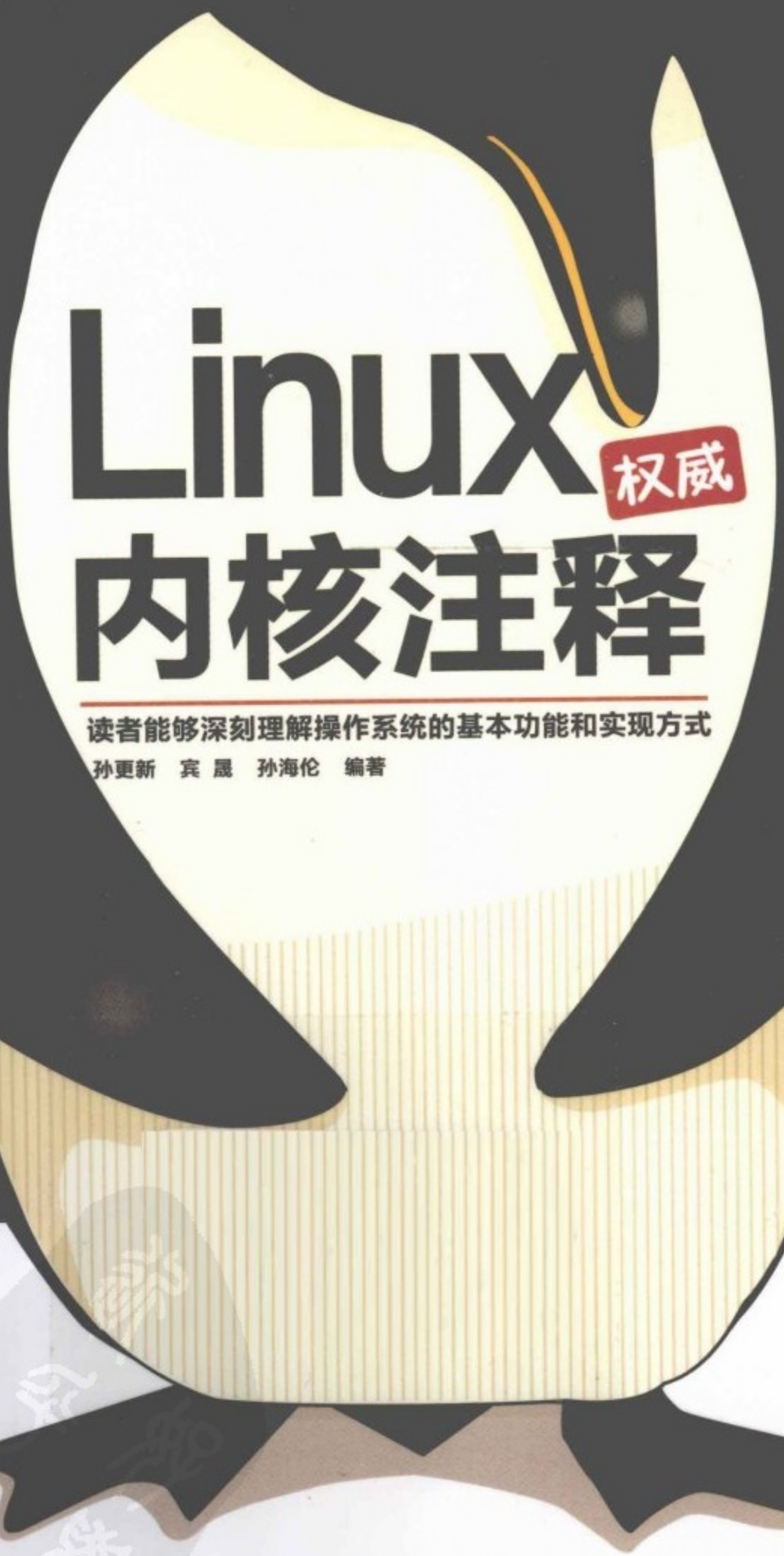
Linux

内核版本为0.95版
代码量不超过2万行

对Linux内核源代码进行 完全剖析 逐行注释



Linux 权威 内核注释

读者能够深刻理解操作系统的基本功能和实现方式

孙更新 宾 晟 孙海伦 编著

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

ÓÉ Foxit PDF Editor ±à¼-
°æÈ¨ËùÓÐ (c) by Foxit Software Company, 2004
½öÓÃÓÚÆÀ¹À¡£

- android与iphone及ipad开发书籍 ----------持续不断更新中.....
- c、c++、c#语言pdf书籍及vip视频教程 c、c++、c#、vc等-------持续不断更新中.....
- delphi《书籍》及《视频》教程 ----------持续不断更新中.....
- E网情深VIP系列视频教程 黑客破解菜鸟修练班，VB编程学习班，仿站学习培训，免杀培训，个人系统攻防系列教程，服务器搭建学习班，PHOTOSHOP平面设计班，基础制作论坛（论坛网站搭建），网赚系列教程，网站建设教程，网站漏洞基础，远程控制教程，软件破解班，脚本漏洞提权班
- IT9网络学院VIP系列视频教程 免杀培训班，VMware虚拟机，零基础学习C语言，网游外挂开发精品系列语音教程（外挂教程学习必备研修31课全），VB语言教程30课全，Delphi编程到精通，远程控制软件，加密解密班，网络安全与黑客攻防培训，从入门到精通完整系统化学习C++编程，从入门到精通零基础学习汇编，wordpress教程(个人博客系统49课全)，外行人做易语言盗号和钓鱼程序语音教程 网址：WLSAM168.400GB.COM
- Java书籍 ----------持续不断更新中.....
- photoshop、CorelDRAW、AutocAD等图像处理书籍及vip视频教程 -------持续不断更新中.....
- powerbuilder书籍大全
- Visual Basic语言vip视频教程及pdf书籍 ----------持续不断更新中.....
- windows、linux系统开发、系统封装等pdf书籍及VIP视频教程 ----------持续不断更新中.....
- 《3DS Max》pdf书籍
- 《汇编语言》、《反汇编》及《调试》pdf书籍及vip视频教程 ----------持续不断更新中.....
- 《电子书、电子书、还是电子书》pdf专题库 编程开发，家居美食，儿童益智，人物传记，增强记忆，快速阅读
- 信息系统项目管理师、网络工程师、系统分析师等软考类书籍
- 华中红客系列vip视频教程 脚本攻防培训班，源码免杀培训班，Css语言培训班，C语言，Dreamweaver网页设计，html网页设计培训班，PC安全班， php脚本语言培训班，VMWare虚拟机专题，webshell提权培训班，防站教程，零基础免杀培训班，刷钻速成班，脱壳破解班，外挂编写班，网络赚钱培训班，网站入侵培训班
- 外挂、驱动、逆向及封包视频教程 郁金香、独立团、夜猫论坛、天都吧、看流星论坛、一切从零开始等等
- 安全中国系列vip视频教程 易语言软件编程培训班，ASP.net网站开发项目实战培训班
- 我的收藏
- 按键精灵及TC脚本开发软件视频教程 ----------持续不断更新中.....

当前位置：/《电子书、电子书、还是电子书》pdf专题库

文件名

PDF电子书专题库，内容详尽，每天不断更新！！

- 办公类软件使用指南
- 医学
- 历史人物传记
- 哲学宗教
- 外语资料（除英语外） （除英语外）
- 官场类小说
- 建筑工程类
- 情感生活类小说
- 政治军事
- 教育学习科普大全
- 文学理论
- 智力开发、增强记忆、快速阅读技巧大全
- 社会生活
- 科学技术
- 程序编程类
- 经济管理
- 网络安全及管理
- 网赚系列
- 美食小吃烹饪煲汤大全
- 课外读物

本网盘内容太多，持续不断更新，发布各类视频教程、pdf书籍，包括破解、加解密、外挂辅助制作，易语言培训教程、编程语言、网页制作等等，教程及书籍仅用于学习,如用于商业或非法用途的后果自负！

网址：WLSAM168.400GB.COM

VIP培训教程，易语言黑月VIP视频教程，天都网易语言系列培训教程(100集全)，集中营易语言学习视频(80集)

- 棉猴系列vip视频教程 gh0st远控控制源码讲解教程，套接字编程，DLL程序编写，键盘监听驱动程序编写，驱动基础教程，AsyncSelect模型QQ程序教程，C++语言入门基础，NB5.5源码分析教程
- 游戏开发pdf书籍 ----------持续不断更新中.....
- 炒股投资pdf书籍及视频教程 短线高手系列，短线天王系列，操盘论道系列，翻倍黑马，看盘快速入门，庄家手法大曝光等等。 网址：WLSAM168.400GB.COM
- 热门小说集中营 傲世九重天，网游之三国时代，武动乾坤
- 甲壳虫VIP教程全集 asp教程，Delphi培训班，FLASH培训班，Java培训班，linux培训班，PHP培训班，源码免杀班，甲壳虫C++，脚本功防班，免杀班初、中、高级班，破解班，源码免杀班，脱壳班，易语言培训班，无特征码免杀，网站架构培训班，外挂高级班，外挂初级班第1、2部
- 破解、免杀、入侵、脱壳、攻防及漏洞分析系列VIP视频教程（80多部） 天草、黑客动画吧等等--------持续不断更新中....
- 网站建设相关的pdf书籍及各种vip视频教程 ----------持续不断更新中.....
- 网赚、淘宝系列vip视频教程 网赚30天新人魔鬼训练，屠龙网赚团队vip课程，站长大学网赚视频（50课全），图腾团队日赚1000元竞价行销教程，屠龙团队淘宝宝贝卖疯系列，站群网赚系列，淘宝开店视频，红星挂机日赚10元，百万流量系列，漂流瓶圣手全自动挂机引，贴吧邮件定向营销疯狂成交量月入万元
- 英语学习资料百科大全 不断更新。。。
- 饭客论坛系列VIP视频教程 脚本入侵班，黑客之免杀教程，易语言教程，无线网络攻防教程，入侵教程，delphi系列教程，黑客基础入门
- 黑客书籍 有关黑客、安全、加解密技术等等----------持续不断更新中.....
- 黑手安全网VIP系列视频教程 DIV+CSS网页布局，Dreamweaver教程，flsah动画教程，photoshop教程，跟我一起学C++课程，抓鸡
- 黑鹰、黑基、黑防、黑盾vip系列视频教程 破解提高班66课全，SQL注入，ASP注入教程，完完全全学会抓肉鸡，脱壳破解教程50课全，提权班，C语言特训班26讲全，黑客脚本特训班，黑客工具特训班，dedecms仿站教程，VC编写远控30课全，网页美工特训班，木马免杀特训班，驱动开发技术VIP培训班，外挂破解等等。

- [电脑世界的通关密语：电脑编程基础].(杉浦贤).滕永红.扫描版.pdf
- [程序语言的奥妙：算法解读（四色全彩）].(杉浦贤).李克秋.扫描版.pdf
- [差错：软件错误的致命影响].(帕伯斯).邝宇恒等.扫描版.pdf
- [算法之道（第2版）].邹恒明.扫描版.pdf
- [O'Reilly：深入学习MongoDB].(霍多罗夫).巨成等.扫描版.pdf
- [深入浅出WPF].刘铁猛.扫描版.pdf
- [Go语言·云动力（云计算时代的新型编程语言）].樊虹剑.扫描版.pdf
- [精通.NET互操作：P/ Invoke、C++ Interop和COM Interop].黄际洲等.扫描版.pdf
- [编程的奥秘：.NET软件技术学习与实践].金旭亮.扫描版.pdf
- [O'Reilly：学习OpenCV（中文版）].(布拉德斯基等).于仕琪等.扫描版.pdf
- [Go语言编程].许式伟等.扫描版.pdf 网址：WLSAM168.400GB.COM
- [MySQL技术内幕：SQL编程].姜承尧.扫描版.pdf
- [Tomcat权威指南（第2版）].(布里泰恩等).吴豪等.扫描版.pdf
- [Ext江湖].大漠穷秋.扫描版.pdf
- [IT名人堂·Oracle DBA突击：帮你赢得一份DBA职位].张晓明.扫描版.pdf

Total: 77 1 2 3 4 5 6 ›

HTTP://WLSAM168.400GB.COM

Linux 内核注释

读者应具备的基础知识：

- 掌握C语言的相关知识
- 掌握有关80×86处理器结构和编程的知识
- 掌握有关80×86硬件体系结构和接口编程的知识
- 对Linux比较熟悉，具备使用Linux操作系统的初级技巧

如果读者还具有操作系统设计相关经验或其他计算机科学的概念则更佳

本书主要面向读者群体：

- 适合Linux爱好者学习内核工作原理
- 可供一般技术人员作为开发嵌入式系统的参考书使用
- 可作为高校计算机专业学生学习操作系统课程的辅助和实验教材

责任编辑：苏　茜　封面设计：张　丽　上架建议：计算机/操作系统/Linux

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE
地址：北京市西城区右安门西街8号
邮编：100054
网址：http://www.tdpress.com

ISBN 978-7-113-13019-0

定价：69.00 元

Linux

Linux内核注释

孙更新　宾 晟　孙海伦　编著

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

内 容 简 介

本书通过对完整的 0.95 版本 Linux 内核源代码进行完全剖析和逐行注释，使得读者能够在较短时间内深刻理解 Linux 操作系统的基本功能和实现方式，为深入研究 Linux 操作系统打下坚实基础。作为 Linux 早期内核版本的 0.95 版本，其内核是目前所使用的较新内核的精简版，代码量比较小，却包括了几乎所有的基本功能原理的内容，通过阅读源代码能够使读者快速地理解内核工作的基本原理。

本书共分 10 章，全面系统地介绍了 Linux 内核发展历史和版本变化，详细说明了 Linux 运行的硬件基础和 Linux 内核体系结构，完全剖析和注释了 Linux 内存管理、中断处理与系统调用、进程控制、块设备驱动程序、字符设备驱动程序、文件系统、引导启动程序、初始化程序，最后总结性地介绍了继续研究 Linux 系统的方法和着手点。

本书适合 Linux 爱好者学习内核工作原理使用，也可作为高校计算机专业学生学习操作系统课程的辅助和实验教材，还可供一般技术人员作为开发嵌入式系统的参考书使用。

图书在版编目（CIP）数据

Linux 内核注释/孙更新，宾晟，孙海伦编著. --
北京：中国铁道出版社，2011.12
ISBN 978-7-113-13019-0

Ⅰ.①L… Ⅱ.①孙…②宾…③孙… Ⅲ.①
Linux 操作系统 Ⅳ.①TP316.89

中国版本图书馆 CIP 数据核字（2011）第 095565 号

书　　名：Linux 内核注释
作　　者：孙更新　宾　晟　孙海伦　编著

责任编辑：苏　茜　　**读者热线电话**：010-63560056
编辑助理：贾淑媛
封面设计：张　丽　　**责任印制**：李　佳

出版发行：中国铁道出版社（北京市西城区右安门西街 8 号　　邮政编码：100054）
印　　刷：北京新魏印刷厂
版　　次：2011 年 12 月第 1 版　　2011 年 12 月第 1 次印刷
开　　本：787mm×1092mm　1/16　　**印张**：28.5　　**字数**：675 千
书　　号：ISBN 978-7-113-13019-0
定　　价：69.00 元

版权所有　侵权必究

凡购买铁道版图书，如有印制质量问题，请与本社发行部联系调换。

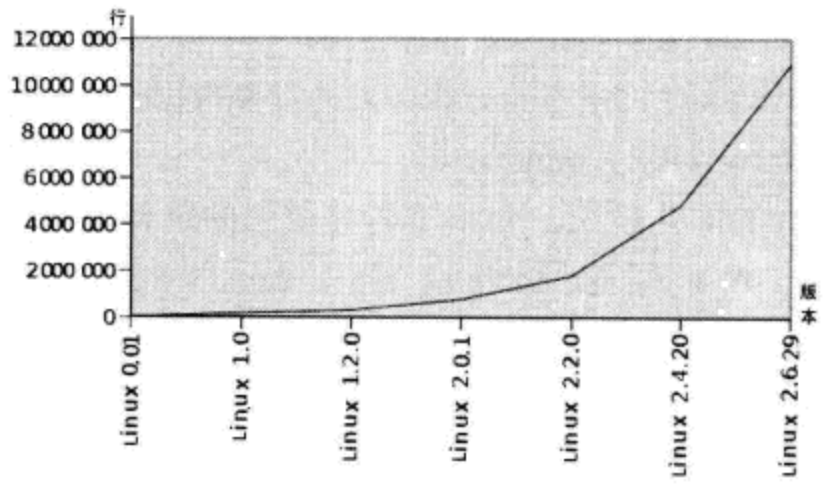
前　言

研究 Linux 内核可以彻底理解 Linux 的内部构造及其基本工作原理。本书通过对完整的 Linux 内核源代码进行完全剖析和逐行注释，使得读者能够深刻理解操作系统的基本功能和实现方式，最终达到对 Linux 操作系统基本工作原理理解的目的。

本书是一本关于 Linux 操作系统内核基本原理的入门级读物，主要面向具有一定编程技能和熟悉 Linux 系统，同时又想要了解 Linux 操作系统内核工作原理和实际代码实现的读者。这部分读者人数众多，但满足其需求的相关书籍却很少，本书就是这部分读者的最佳选择之一。

内核版本选择

目前，Linux 内核的源代码量非常大，有数百万行之多，因此无法对这些版本进行完全地注释和说明，仅能对其有选择性地进行说明，这将会给读者一种盲人摸象的感觉，而无法给予读者对实际 Linux 内核清晰完整的理解。早期内核源代码则是目前所使用的较新内核的精简版，代码量比较小，并且包括了几乎所有的基本功能原理的内容。选择 Linux 早期内核版本进行剖析，可以避免实际内核功能实现的复杂性，又能透彻地说明问题，使读者能够在尽可能短的时间内深入理解 Linux 内核的基本工作原理。下图是对一些主要 Linux 内核版本源代码量的统计。



通过大量的调查和对内核版本的详细比对，本书选择了与目前 Linux 内核基本功能较为相近，同时代码量又比较小的 0.95 版内核（代码量不超过 2 万行）作为分析的对象，因此本书完全可以对 Linux 内核的基本功能解释和注释清楚。

读者应该具备的基础知识

阅读本书的读者应该具有以下一些基础知识：

- 掌握 C 语言的相关知识。
- 掌握有关 80×86 处理器结构和编程的知识。
- 掌握有关 80×86 硬件体系结构和接口编程的知识。
- 对 Linux 比较熟悉，具备使用 Linux 操作系统的初级技能。如果读者还具有操作系统设计相关经验或其他计算机科学的概念则更佳。

面向读者群体

本书适合 Linux 爱好者学习内核工作原理使用，也可作为高校计算机专业学生学习操作系统课程的辅助和实验教材，还可供一般技术人员作为开发嵌入式系统的参考书使用。

本书的组织形式

本书各章之间组织有序，内容循序渐进又相对独立。对于不了解 Linux 内核的读者，可以由浅入深逐步理解内核的工作原理和实现方式。对内核基本情况已经有所了解的读者，完全可以按照自己的实际需要直接阅读相关章节。

第 1 章对 Linux 操作系统原理进行了简单介绍，主要包括 Linux 操作系统的内核构成及操作系统的特征，Linux 操作系统的诞生、成长和发展，然后介绍了阅读和研究 Linux 内核的重要意义等内容。

第 2 章讨论了引导与初始化程序，分析并注释了程序 bootsect.s、setup.s、head.s、main.c。

第 3 章分析并注释了 Linux 内核的所有头文件。

第 4 章讨论了中断处理和系统调用的总体功能，介绍了与其相关的重要数据结构，分析并注释了头文件 unistd.h 和 sys.h 以及程序 traps.c、sys.c、system_call.s、asm.s。

第 5 章讨论了进程控制，介绍了与其相关的重要数据结构，分析并注释了程序 fork.c、sched.c、signal.c。

第 6 章讨论了字符设备驱动程序，介绍了与其相关的重要数据结构，分析并注释了程序 console.c、keyboard.s、pty.c、rs_io.s、serial.c、tty_io.c。

第 7 章讨论了块设备驱动程序，介绍了与其相关的重要数据结构，分析并注释了头文件 blk.h 以及程序 floppy.c、hd.c、ll_rw_blk.c、ramdisk.c。

第 8 章讨论了 Linux 的文件系统，介绍了与其相关的重要数据结构，分析并注释了程序 buffer.c、inode.c、namei.c、super.c、file_table.c、block_dev.c、pipe.c、char_dev.c、read_write.c、open.c、stat.c、fcntl.c、ioctl.c、select.c、Minix 的 bitmap.c、Minix 的 inode.c、Minix 的 truncate.c、Minix 的 namei.c、Minix 的 file_dev.c、Minix 的 minix_op.c。

第 9 章讨论了 Linux 的内存管理，分析并注释了头文件 mm.h 和 memory.h 以及程序 memory.c、swap.c。

第 10 章介绍了 Linux 内核的建造工具与编程环境，重点分析并注释了生成内核磁盘映像文件的工具程序 buidl.c 和如何在 Linux 下进行开发。

创作群体

本书由孙更新、宾晟、孙海伦编写，下面人员对本书的编写提出过宝贵意见并参与了部分编写工作，他们是张杰、徐作英、邵长恒、宋小宁、吴洪飞、刘树一、刘丽、孔波、颜浩、贺萍、于瑞菊、王亚平、任晓雪、李琼、高宝玉、孙军、徐凯。

由于时间仓促，加之水平有限，书中的缺点和不足之处在所难免，敬请读者批评指正。如果读者在阅读本书的时候出现任何疑问，还可以发送电子邮件及时与我们联系，我们会尽快给予答复。我们的邮箱是 suqian@tqbooks.net。

目　录

第 1 章　Linux 操作系统原理 1

1.1　Linux 操作系统简介 1
1.1.1　Linux 内核的构成 4
1.1.2　Linux 操作系统的特征 5
1.2　Linux 内核概述 6
1.2.1　Linux 内核的版本 6
1.2.2　研究 Linux 内核的意义 9
1.3　Linux 运行的硬件基础 10
1.3.1　i386 寄存器和系统指令 10
1.3.2　分段机制 15
1.3.3　分页机制 23
1.4　Linux 内核体系结构 26
1.4.1　Linux 内核体系结构及模式 27
1.4.2　Linux 内核源代码目录结构 28
1.4.3　Linux 内核的编译 32

第 2 章　引导启动与初始化程序 36

2.1　引导启动程序（boot） 36
2.1.1　总体功能 36
2.1.2　bootsect.s 文件 37
2.1.3　setup.s 文件 42
2.1.4　head.s 文件 47
2.2　初始化程序（init） 51
2.2.1　概述 51
2.2.2　main.c 文件 51

第 3 章　头文件 57

3.1　include/目录下的文件 57
3.2　a.out.h 文件 57
3.3　const.h 文件 61
3.4　ctype.h 文件 62
3.5　errno.h 文件 63
3.6　fcntl.h 文件 64
3.7　signal.h 文件 65
3.8　stdarg.h 文件 67
3.9　stddef.h 文件 68

3.10 string.h 文件 68
3.11 termios.h 文件 77
3.12 time.h 文件 82
3.13 unistd.h 文件 83
3.14 utime.h 文件 89
3.15 include/asm 目录下的文件 90
3.16 io.h 文件 90
3.17 memory.h 文件 91
3.18 segment.h 文件 91
3.19 system.h 文件 93
3.20 include/linux/目录下的文件 95
3.21 config.h 文件 95
3.22 fdreg.h 头文件 96
3.22.1 功能描述 96
3.22.2 代码注释 97
3.23 fs.h 文件 99
3.24 hdreg.h 文件 103
3.25 head.h 文件 105
3.26 kernel.h 文件 105
3.26.1 功能描述 105
3.26.2 代码注释 105
3.27 mm.h 文件 106
3.28 sched.h 文件 107
3.29 sys.h 文件 112
3.30 tty.h 文件 114
3.31 math_emu.h 文件 116
3.31.1 功能描述 116
3.21.2 代码注释 116
3.32 minix_fs.h 文件 120
3.33 timer.h 文件 121
3.34 include/sys/目录下的文件 122
3.35 param.h 文件 122
3.36 ptrace.h 文件 122
3.37 resource.h 文件 124
3.38 stat.h 文件 125
3.39 time.h 文件 126
3.40 times.h 文件 127
3.41 types.h 文件 128
3.42 utsname.h 文件 129
3.43 wait.h 文件 129

第 4 章　中断处理与系统调用 130

4.1　总体功能 130
4.1.1　中断处理 130
4.1.2　系统调用 131
4.2　重要数据结构 132
4.3　sys.h 文件 132
4.4　unistd.h 文件 134
4.5　IDT 初始化——_set_gate 141
4.6　traps.c 文件 141
4.7　sys.c 文件 146
4.8　system_call.s 文件 156
4.9　asm.s 文件 162
4.10　ptrace.c 文件 165

第 5 章　进程控制 172

5.1　总体概述 172
5.2　重要数据结构 174
5.2.1　task_struct 174
5.2.2　tss_struct 175
5.2.3　i387_struct 176
5.2.4　task_union, init_task, current 176
5.2.5　sigaction 176
5.2.6　timer_list 176
5.3　fork.c 文件 177
5.3.1　功能描述 177
5.3.2　代码注释 177
5.4　sched.c 文件 180
5.4.1　功能描述 181
5.4.2　代码注释 181
5.5　signal.c 文件 190
5.5.1　功能描述 190
5.5.2　代码注释 190
5.6　exit.c 文件 195
5.6.1　功能描述 195
5.6.2　代码注释 195
5.7　vsprintf.c 文件 203
5.8　printk.c 文件 208
5.9　panic.c 文件 209

第 6 章　字符设备驱动程序 210
6.1　总体功能 210
6.1.1　Linux 支持的终端设备类型 210
6.1.2　Linux 字符设备驱动程序 212
6.1.3　字符设备驱动程序功能 212
6.2　重要数据结构 213
6.3　console.c 文件 214
6.4　keyboard.s 文件 238
6.5　pty.c 文件 252
6.6　rs_io.s 文件 253
6.7　serial.c 文件 256
6.8　tty_io.c 文件 258
6.8.1　功能描述 258
6.8.2　代码注释 259
6.9　tty_ioctl.c 文件 268
第 7 章　块设备驱动程序 276
7.1　总体功能 276
7.1.1　Linux 设备管理 276
7.1.2　Linux 块设备驱动程序 277
7.1.3　块设备驱动程序功能 278
7.2　重要数据结构 279
7.3　blk.h 文件 280
7.4　floppy.c 文件 283
7.5　hd.c 文件 293
7.5.1　功能描述 293
7.5.2　代码注释 294
7.6　ll_rw_blk.c 文件 303
7.7　ramdisk.c 文件 308
第 8 章　文件系统 311
8.1　总体功能 311
8.1.1　VFS 虚拟文件系统 312
8.1.2　Minix 文件系统 312
8.1.3　文件系统程序功能 313
8.2　重要数据结构 314
8.3　buffer.c 文件 317
8.3.1　功能描述 317
8.3.2　代码注释 318
8.4　inode.c 文件 326

8.4.1　功能描述 326
8.4.2　代码注释 327
8.5　namei.c 文件 332
8.5.1　功能描述 332
8.5.2　代码注释 333
8.6　super.c 文件 342
8.6.1　功能描述 342
8.6.2　代码注释 342
8.7　file_table.c 文件 349
8.8　block_dev.c 文件 349
8.8.1　功能描述 349
8.8.2　代码注释 349
8.9　pipe.c 文件 351
8.10　char_dev.c 文件 354
8.11　read_write.c 文件 358
8.12　open.c 文件 360
8.13　exec.c 文件 365
8.14　stat.c 文件 372
8.15　fcntl.c 文件 374
8.16　ioctl.c 文件 376
8.17　select.c 文件 377
8.18　Minix 的 bitmap.c 文件 383
8.19　Minix 的 inode.c 文件 387
8.20　Minix 的 truncate.c 文件 391
8.21　Minix 的 namei.c 文件 394
8.22　Minix 的 file_dev.c 文件 411
8.23　Minix 的 minix_op.c 文件 413

第 9 章　内存管理 415

9.1　总体功能描述 415
9.1.1　内存分页机制管理 415
9.1.2　Linux 中内存的使用 417
9.2　memory.c 文件 417
9.2.1　功能描述 417
9.2.2　代码分析 417
9.3　swap.c 文件 429

第 10 章　Linux 内核编程环境 435

10.1　建造工具介绍 435
10.1.1　总体功能 435

10.1.2 重要数据结构 435
10.1.3 build.c 文件 436
10.2 在 Redhat 9 下修改和编译 Linux 0.95 内核 439
10.2.1 源代码中注释语句的修改 439
10.2.2 源代码中 makefile 文件的修改 439
10.2.3 源代码中 align 值的修改 440
10.2.4 源代码中嵌入宏汇编程序的修改 440
10.2.5 源代码中汇编程序 C 变量的修改 440
10.2.6 在保护模式下调试显示函数 440
10.2.7 编译 Linux 0.95 内核 441
10.3 在 Bochs 上运行 Linux 0.95 系统 441
10.3.1 创建磁盘映像文件 441
10.3.2 访问磁盘映像文件 441
10.3.3 创建文件系统 442
10.3.4 在硬盘映像文件上建立根文件系统 442
10.3.5 使用硬盘映像上的根文件系统运行 Linux 0.95 443

第 1 章　Linux 操作系统原理

操作系统（Operating System，OS）是控制其他程序运行、管理系统资源并为用户提供操作界面的系统软件的集合。它是管理计算机硬件与软件资源的程序，同时也是计算机系统的内核与基石。操作系统理论在计算机科学中是历史悠久而又活跃的分支，而操作系统的设计与实现则是软件工业的基础与内核。操作系统本身负责诸如管理与配置内存、决定系统资源供需的优先次序、控制输入与输出设备、操作网络与管理文件系统等基本事务。

操作系统能够使计算机系统的所有资源最大限度地发挥作用，为用户提供方便、有效、友善的服务界面。操作系统是一个庞大的管理控制程序，大致包括 5 个方面的管理功能：进程与处理机管理、作业管理、存储管理、设备管理、文件管理。目前的操作系统种类繁多，不同类型计算机安装的操作系统有的简单有的复杂，从手机的嵌入式系统到超级电脑的大型操作系统，很难用单一标准统一分类。微型计算机上常见的操作系统包括 DOS、OS/2、UNIX、XENIX、Linux、Windows、NetWare 等。但所有的操作系统都具有并发性、共享性、虚拟性和不确定性四个基本特征。

1.1　Linux 操作系统简介

Linux 是当前的主流操作系统之一，因其开放源代码的特性而深受众多用户的青睐。

1．Linux 的诞生和发展

早期的操作系统是在 20 世纪 50 年代开发的，用来提供简单的开发体验，包括为 IBM 701 开发的 General Motors Operating System（GMOS）以及为 IBM 709 开发的 FORTRAN Monitor System（FMS）。

在 20 世纪 60 年代，麻省理工大学（Massachusetts Institute of Technology，MIT）和一些公司为 GE-645 开发了一个名为 Multics（Multiplexed Information and Computing Service）的实验性操作系统。这个操作系统的开发者之一 AT&T 后来退出了 Multics，并在 1970 年开发了自己的名为 Unics 的操作系统。与这个操作系统一同诞生的是 C 语言，然后他们使用 C 语言对操作系统进行了重写，使操作系统开发具有可移植性。

20 年后，为了教学的需要，荷兰计算机教授 Andrew S. Tanenbaum 创建了一个微内核版本的 UNIX，名为 MINIX（其含义为 minimal UNIX），它可以在小型的个人计算机上运行。

Linux 操作系统于 1991 年诞生于一个芬兰大学生 Linus Torvalds 之手：1991 年 8 月，他在赫尔辛基大学学习操作系统课程时，由于不满足于使用教学用的操作系统 MINIX，于是着手开发一个简单的程序，并逐步开发了显示器、键盘和调制解调器的驱动程序，最后编写了磁盘驱动程序、文件系统等，一个操作系统的原型就这样形成了，后来它逐步发展为与 UNIX、Windows 并驾齐驱的实用操作系统。

Linux 操作系统是一种自由的 UNIX 类多用户、多任务操作系统，它可以运行在多种计算机平台上。自 1991 年 10 月 5 日，Linus 正式向外宣布 Linux 内核系统的诞生以来，它已经成为应用广泛、可靠性高，功能强大的计算机操作系统。借助于 GNU 计划、POSIX 标准、Internet，并经过世界各地计算机爱好者的共同努力，Linux 现已成为目前使用较多的一种 UNIX 类操作系统，并且使用人数还在迅猛增长。

Linux 操作系统内核的名字也是 Linux。Linux 操作系统是自由软件和开放源代码发展中最著

名的例子之一。严格来讲，Linux 这个词本身只表示 Linux 内核，但在实际上人们已经习惯了用 Linux 来形容整个基于 Linux 内核，并且使用 GNU 工程各种工具和数据库的操作系统（GNU/Linux）。基于这些组件的 Linux 软件被称为 Linux 发行版。一般来讲，一个 Linux 发行套件包含大量的软件，例如，软件开发工具、数据库、Web 服务器、X Window，桌面环境、办公套件等。Linux 内核最初是为 Intel 386 微处理器设计的。

现在Linux内核支持从个人电脑到大型主机甚至包括嵌入式系统在内的各种硬件设备。现在，Linux 已经成为了受到广泛关注和支持的一种操作系统，包括 IBM 和惠普在内的一些计算机业巨头也开始支持 Linux。很多人认为，和其他的商用 UNIX 操作系统以及微软的 Windows 系列操作系统相比，作为自由软件的 Linux 具有成本低、安全性高、更加可信赖的优势。

Linux 操作系统在短短的几年之内得到了非常迅猛的发展，这与 Linux 具有的良好特性是分不开的。Linux 包含了 UNIX 的全部功能和特性。简单地说，Linux 具有以下主要特性：

- 开放性：指系统遵循世界标准规范，特别是遵循开放系统互连（OSI）国际标准。凡遵循国际标准所开发的硬件和软件，都能彼此兼容，可方便地实现互连。另外，源代码开放的 Linux 是免费的，使得获取 Linux 非常方便，而且使用 Linux 可节省费用。Linux 开放源代码，使用者能控制源代码，按照需要对部件混合搭配，建立自定义扩展。
- 多用户：指系统资源可以被不同用户各自拥有使用，即每个用户对自己的资源（如文件、设备）有特定的权限，互不影响。
- 多任务：多任务是现代计算机的最主要的特点之一，是指计算机同时执行多个程序，而且各个程序的运行互相独立。Linux 系统调度每一个进程平等地访问微处理器。
- 出色的速度性能：Linux 可以连续运行数月、数年而无需重新启动，与 NT（经常死机）相比，这一点尤其突出。即使作为一种台式机操作系统，与许多用户非常熟悉的 UNIX 相比，它的性能也显得更为优秀。Linux 不大在意 CPU 的速度，它可以把处理器的性能发挥到极限（用户会发现，影响系统性能提高的限制因素主要是其总线和磁盘 I/O 的性能）。
- 良好的用户界面：Linux 向用户提供了 3 种界面：用户命令界面、系统调用界面和图形用户界面。
- 提供了丰富的网络功能：Linux 是在 Internet 基础上产生并发展起来的，因此，完善的内置网络是 Linux 的一大特点。Linux 在通信和网络功能方面优于其他操作系统。
- 可靠的系统安全：Linux 采取了许多安全技术措施，包括对读、写进行权限控制、带保护的子系统、审计跟踪、核心授权等，这为网络多用户环境中的用户提供了必要的安全保障。
- 良好的可移植性：可移植性是指将操作系统从一个平台转移到另一个平台时它仍然能按其自身方式运行的能力。Linux 是一种可移植的操作系统，能够在从微型计算机到大型计算机的任何环境和任何平台上运行。可移植性为运行 Linux 的不同计算机平台与其他任何机器进行准确而有效地通信提供了手段，不需要另外增加特殊和昂贵的通信接口。
- 具有标准兼容性：Linux 是一个与 POSIX（Portable Operating System Interface）相兼容的操作系统，它所构成的子系统支持所有相关的 ANSI、ISO、IETF 和 W3C 业界标准。为了使 UNIX system V 和 BSD 上的程序能直接在 Linux 上运行，Linux 还增加了部分 system V 和 BSD 的系统接口，使 Linux 成为一个完善的 UNIX 程序开发系统。Linux 也符合 X/Open 标准，具有完全自由的 X-Window 实现。另外，Linux 在对工业标准的支持上做得非常好。由于各 Linux 发布厂商都能自由获取和接触 Linux 的源代码，各厂家发布的 Linux 仍然缺乏标准，不过这些差异非常小。它们的差异主要存在于所捆绑应用软件的版本、安装工具

的版本和各种系统文件所处的目录结构。

Linux 操作系统一般包含 4 个主要部分：内核、Shell、文件系统和应用程序，各部分层次结构如图 1-1 所示。内核、Shell 和文件系统形成了基本的操作系统结构。它们使得用户可以运行程序、管理文件并使用系统。

图 1-1　Linux 系统的组成

- Linux 内核：内核（Kernel）是系统的心脏，实现操作系统的基本功能。
  在硬件方面：控制硬件设备，管理内存，提供硬件接口，处理基本 I/O。
  在软件方面：管理文件系统，为程序分配内存和 CPU 时间等。
- Linux Shell：Shell 是系统的用户界面，提供用户与内核进行交互操作的一种接口。Shell 是一个命令解释器，它解释由用户输入的命令并且把它们送到内核执行。Shell 编程语言具有普通编程语言的很多特点，用这种编程语言编写 Shell 程序与其他应用程序具有同样的效果。目前，常见的 Shell 有 Bourne Shell（sh）、Korn Shell（ksh）、C Shell（csh）、Bourne-again Shell（bash）等。
- Linux 应用程序：标准的 Linux 系统都有一套称为应用程序的程序集，包括文本编辑器、编程语言、X- Window、办公套件、Internet 工具、数据库等。当然，还可以有用户自己编写的具有特定功能的应用程序。
- Linux 文件系统：文件系统是文件存放在磁盘等存储设备上的组织方法。通常是按照目录层次的方式进行组织。每个目录可以包括多个子目录以及文件，系统以“/”为根目录。系统中的所有数据都存储在文件系统上以便用户读取、查询和写入。Linux 能支持多种目前流行的文件系统，如 ext2、ext3、fat、vfat、ISO9660、nfs 等。

2．常见的 Linux 操作系统

目前经常见到的 Linux 操作系统主要有 Red Hat、Ubuntu、SuSE、Red Flag、Turbo Linux、Fedora 等。

（1）Red Hat

Red Hat Linux 最大的优点是它不仅已经是行业的标准，有大量为其开发的软件，而且它的产品稳定性也得到了广泛的认可。

1998 年 1 月，Red Hat 高级研发实验室成立，同年 Red Hat 5.0 获得了 InfoWorld 的操作系统奖项。4 月 Mozilla 代码发布，成为 Linux 图形界面上的王牌浏览器。同年 10 月，Intel 和 Netscape 宣布小额投资 Red Hat 软件，这被业界视作 Linux 获得商业认同的信号。12 月，IBM 发布了适用于 Linux 的文件系统 AFS 3.5 以及 Jikes Java 编辑器和 Secure Mailer 及 DB2 测试版。

1999 年 5 月，SGI 公司宣布向 Linux 移植其先进的 XFS 文件系统。7 月，IBM 启动对 Linux

的支持服务和发布了 Linux DB2，从此结束了 Linux 得不到支持服务的历史。

（2）Ubuntu

Ubuntu 以快速上手以及长久的生存能力著称。快速上手，是指使用 Ubuntu 无需指导，花几分钟安装，立刻识别大部分硬件，隐藏 root（根）用户，避免有人弄乱它，接着预设配置，可使用户马上开始工作。Ubuntu 提供不错的开发者联合平台，OEM 顾客不断增长。

（3）SuSE

欧洲市场占有率第一的发行版，其功能强大，界面华丽，安装 SuSE 可以选择显示英文或德文，它还有一套设定的程序叫 SaX，可以让使用者方便地设置，安装套件是 RPM 模式，所以要安装或删除程序都可以很方便。

（4）Red Flag

这是由中科红旗软件技术公司推出的中文版本的 Linux，自从 2000 年 6 月成立以来该，Linux 在众多的中国 Linux 用户中占有一定的比例。Red Flag Linux 具有灵活的设计，将在嵌入式 Linux 领域为用户提供强大的技术支持服务。

（5）Turbo Linux

它是一个安全、稳定、运行效率高、与硬件以及商业软件兼容性好的系统。该套件由 Pacific HiTech 公司开发，在日本占有一定的市场，从安装到使用都是日文的，在国内它与清华大学及研究机构合作研发了中文版本，掀起了一股 Linux 潮流。

（6）Fedora

Fedora 是一个基于 Linux 的开放、创新、前瞻性的操作系统和平台，Fedora 项目是由 Red Hat 赞助，由开源社区与 Red Hat 工程师合作开发的项目统称。Fedora 的目标是推动自由和开源软件更快地进步。公开的论坛、开放的过程、快速的创新、精英和透明的管理，所有这些都为实现一个自由软件提供了的最好的操作系统和平台。

1.1.1 Linux 内核的构成

Linux 内核由三个主要的子系统组成：进程调度（SCHED）、内存管理（MM）和文件系统（VFS）。

1. Linux 系统的进程调度

Linux 中的每个进程都分配有一个相对独立的虚拟地址空间。该虚拟空间分为两部分：用户空间包含了进程本身的代码和数据；内核空间包含了操作系统的代码和数据。

Linux 采用“有条件的可剥夺”调度方式。对于普通进程而言，当其时间片结束时，调度程序挑选出下一个处于 TASK_RUNNING 状态的进程作为当前进程。对于实时进程，若其优先级足够高，则会从当前的运行进程中抢占 CPU 成为新的当前进程。发生强制调度时，若进程在用户空间中运行，就会直接被剥夺 CPU；若进程在内核空间中运行，即使迫切需要其放弃 CPU，也仍要等到从其系统空间返回前才被剥夺 CPU。

Linux 进程调度分为主动调度和被动调度两种方式：

主动调度随时都可以进行，Linux 内核可以通过 schedule()启动一次调度，当然也可以将进程状态设置为 TASK_INTERRUPTIBLE，使得进程暂时放弃运行而进入睡眠状态；用户态中的程序可以通过 pause()达到同样的目的；如果为这种暂时的睡眠放弃加上时间限制，内核态有 schedule_timeout()，用户态有 nanosleep()用于实现此目的。这里需要注意的是，内核中这种主动放弃是不可见的，它隐藏在每一个可能受阻的系统调用中，例如 open()、read()、select()等。

被动调度发生在系统调用返回的前夕、中断异常处理返回前、用户态处理软中断返回前。

自从 Linux 2.6 内核后，Linux 实现了抢占式内核，即处于内核态的进程也可能被调度出去。例如，一个进程正在内核态运行，此时一个中断发生使另一个高权值进程就绪，在中断处理程序结束之后，Linux 2.6 内核之前的版本会恢复原进程的运行，直到该进程退出内核态才会引发调度程序；而 Linux 2.6 抢占式内核，在处理完中断后，会立即引发调度，切换到高权值进程。为支持内核代码可抢占，在 Linux 2.6 版内核中通过采用禁止抢占的自旋锁来保护临界区。在释放自旋锁时，同样会引发调度检查。而对那些长期持锁或禁止抢占的代码片段插入了抢占点，此时检查调度需求，以避免不合理的延迟发生。而在检查过程中，调度进程很可能就会中止当前的进程来让另外一个进程运行，只要新的进程不需要持有该锁。

2．Linux 系统的内存管理

Linux 内存管理采用分页机制来实现，分页机制是应用程序通过软硬件协作来访问内存的一种方法，其基本原理是将整个线性内存空间划分成若干页面，在 80×86 体系结构中是以 4KB 为一页的内存页面。程序申请使用内存时，就以一页为单位进行分配。

3．Linux 系统的文件系统

Linux 内核将文件系统的程序划分为两部分：一部分实现了对虚拟文件系统的支持；另一部分用于实现与 Minix 文件系统相关的操作，该部分被单独放在一个名为 Minix 的文件夹内。

虚拟文件系统从功能上可以划分为 4 个部分。第一部分是高速缓冲区的管理程序，该程序主要实现了高速缓冲区缓存的功能。第二部分是文件系统低层程序，主要实现了对 i 结点的分配与释放，文件路径名与 i 结点的映射，文件系统超级块的处理以及文件系统的选择处理功能。第三部分是文件数据的访问操作程序，主要实现了对字符设备、管道、块设备、正规文件的数据访问功能。第四部分是文件系统的上层操作程序，主要实现了文件的打开、关闭、创建等有关文件操作的系统调用。

Linux 的基本思想有两点：第一，一切都是文件；第二，每个软件都有确定的用途，同时它们都尽可能被编写得更好。其中第一条详细来讲就是系统中的所有内容都归结为一个文件，包括命令、硬件和软件设备、操作系统、进程等。

1.1.2 Linux 操作系统的特征

Linux 操作系统具有以下特征：并发、共享、虚拟和异步性。

1．并发（concurrency）

并发（concurrency）是指多个事件在同一时间段内发生。操作系统是一个并发系统，包括各进程间的并发，系统与应用间的并发。操作系统要完成这些并发过程的管理。并行（parallel）是指在同一时刻发生。在多道程序处理时，宏观上并发，微观上交替执行（在单处理器情况下）。程序的静态实体是可执行文件，而动态实体是进程（或称做任务），并发指的是进程。

2．共享（sharing）

多个进程共享有限的计算机系统资源。操作系统要对系统资源进行合理分配和使用。资源在一个时间段内交替被多个进程所用。资源分配后到释放前，不能被其他进程所用。

3．虚拟（virtual）

虚拟（virtual）是指一个物理实体映射为若干个对应的逻辑实体——分时或分空间。虚拟是操作系统管理系统资源的重要手段，可提高资源利用率。关于虚拟：CPU——每个用户（进程）的“虚处理机”；存储器——每个进程都占有的地址空间（指令＋数据＋堆栈）；显示设备——多窗口或虚拟终端（virtual terminal）。

4. 异步性（Asynchronism）

异步性（Asynchronism）也称不确定性，指进程的执行顺序和执行时间的不确定性。关于异步性，进程的运行速度不可预知：分时系统中，多个进程并发执行，“时走时停”，不可预知每个进程的运行推进快慢。无论进程运行快慢，结果应该相同——通过进程互斥和同步手段来保证。难以重现系统在某个时刻的状态（包括重现运行中的错误）。性能保证：实时系统与分时系统相似，但通过资源预留以保证性能。

1.2 Linux 内核概述

Linux 内核是一个庞大而复杂的操作系统的核心，不过尽管庞大，但是却采用子系统和分层的形式，很好地进行了组织。因此，对于完整的 Linux 内核进行剖析和阅读也并非很难。通过阅读完整的 Linux 内核源代码，将会真正地理解 Linux 操作系统的基本工作原理。

1.2.1 Linux 内核的版本

Linux 内核分为两种：稳定版和开发版。稳定的内核具有工业级的强度，可以广泛的在商业开发中应用。Linux 的稳定版内核大部分都是对一些 Bug 进行了修正或是在内核中加入一些新的驱动程序。而开发版的内核中许多内容变化得都比较快。而且由于开发者不断试验新的解决方案，因此内核经常是不稳定的。

1. Linux 版本的命名规则

Linux 通过命名机制来区分稳定的和处于开发中的内核。这种机制使用类似 num1.num2.num3，三个用点分隔的数字代表不同版本的内核。其中 num1 是主版本号，num2 是次版本号，num3 是修订版本号。次版本号可以反映出该内核是一个稳定版本还是一个开发版本：如果该数字是偶数，那么此内核就是稳定版，如果是奇数，那么它就是开发版。例如，版本号为 2.6.0 的内核，它就是一个稳定版。这个内核的主板本号是 2，次版本号是 6，修订版本号是 0。头两个数字在一起可以描述内核系列。

2. Linux 内核版本的升迁

Linux 内核发行版简史如图 1-2 所示。

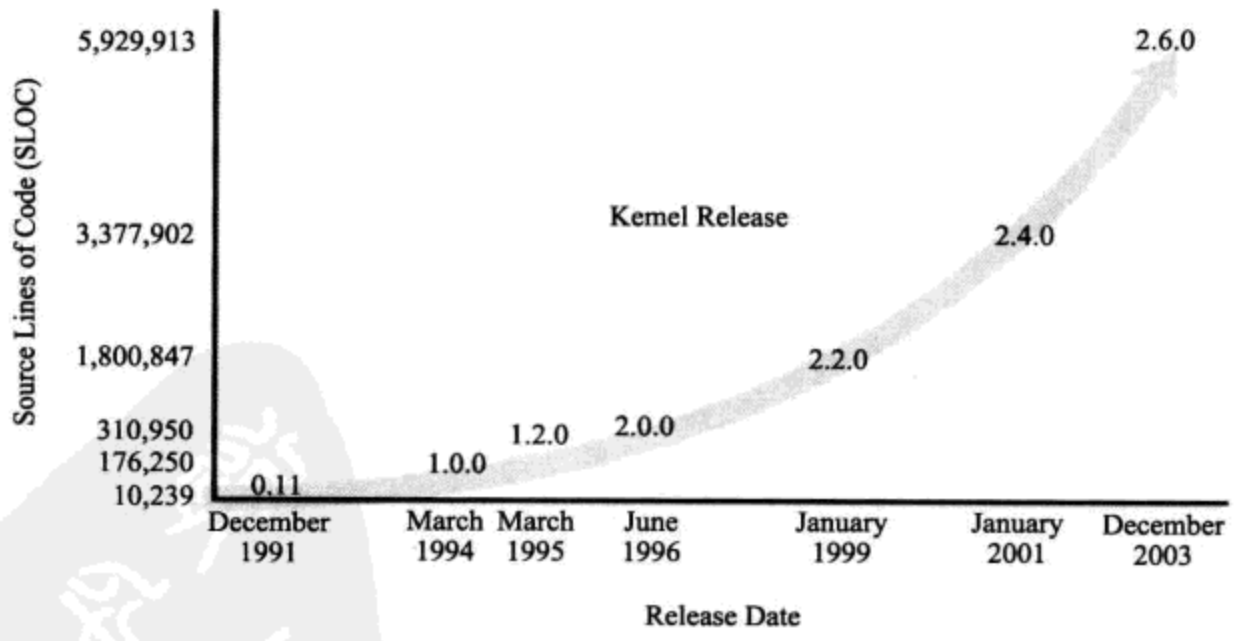


图 1-2 Linux 内核发行简史

其中，Linux 内核从 1991 年诞生到 1994 年 1.0 版本的正式出现，其主要版本信息如表 1-1 所示：

表 1-1　Linux 版本变化情况

版　本　号	发　布　时　间	版　本　说　明
0.00	1991.2-1991.4	两个进程分别显示 AAA BBB
0.01	1991.9	第一个正式向外公布的 Linux 内核版本
0.02	1991.10.5	该版本以及 0.03 版是内部版本，目前已经无法找到
0.10	1991.10	由 Ted Ts'o 发布的 Linux 内核版本
0.11	1991.12.8	基本可以正常运行的内核版本
0.12	1992.1.15	主要加入对数学协处理器的软件模拟程序
0.95(0.13)	1992.3.8	开始加入虚拟文件系统思想的内核版本
0.96	1992.5.12	开始加入网络支持和虚拟文件系统 VFS
0.97	1992.8.1	增加了对新的 SCSI 驱动程序的支持
0.98	1992.9.29	改善了对 TCP/IP 网络的支持
0.99	1992.12.13	重新设计进程对内存使用情况的使用分配，每个进程有 4GB 线性空间
1.0	1994.3.14	第一个正式版本颁布

从 1.0 之后的版本调整非常频繁，这里就不再详述了。

3．Linux 操作系统基本体系结构

现在从一个比较高的高度来审视一下如图 1-3 所示的 Linux 操作系统的体系结构。可以从两个层次来考虑操作系统。

最上面是 User Space（用户空间）是用户应用程序执行的地方。用户空间之下是内核空间，Linux 内核正是位于这里。提供了连接内核的系统调用接口的 GNU C Library （glibc）也在这里，它还提供了在用户空间应用程序和内核之间进行转换的机制。这点非常重要，因为内核和用户空间的应用程序使用的是不同的保护地址空间。每个用户空间的进程都使用自己的虚拟地址空间，而内核则占用单独的地址空间。

Linux 内核可以进一步划分成 3 层。最上面是系统调用接口，它实现了一些例如 read 和 write 的基本功能。系统调用接口之下是内核代码，可以更精确地定义为独立于体系结构的内核代码。这些代码是 Linux 所支持的所有处理器体系结构所通用的。在这些代码之下是依赖于体系结构的代码，这些代码构成了通常称为 BSP（Board Support Package）的部分。这些代码用作支持给定体系结构的处理器和特定平台。

Linux 内核实现了很多重要的体系结构属性。在或高或低的层次上，内核被划分为多个子系统。Linux 也可以看作是一个整体，因为它会将所有这些基本服务都集成到内核中。这与微内核的体系结构不同，后者会提供一些基本的服务，例如通信、I/O、内存和进程管理，更具体的服务都是插入到微内核层中的。

Linux 内核的主要子系统如图 1-4 所示。

（1）系统调用接口（System Call Interface）

SCI 层提供了某些机制用于执行从用户空间到内核的函数调用。这个接口依赖于体系结构，甚至在相同的处理器家族内也是如此。SCI 实际上是一个非常有用的函数调用多路复用和多路分解服务。在 Linux 源代码的./linux/kernel 中可以找到 SCI 的实现，并在./linux/arch 中找到依赖于体系结构的部分。

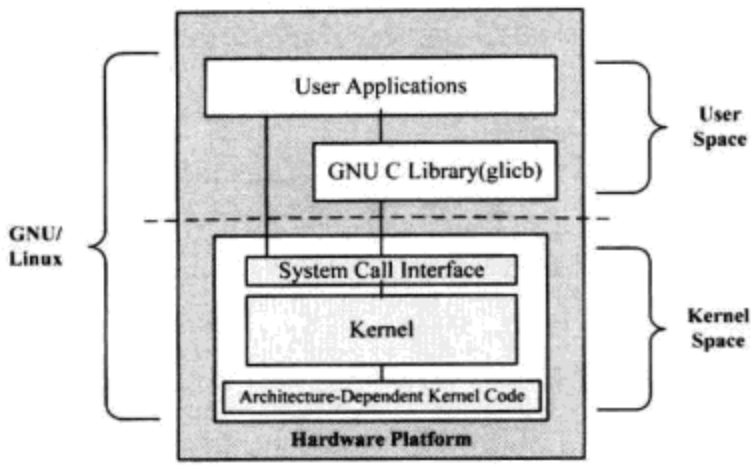


图 1-3　Linux 操作系统的基本体系结构

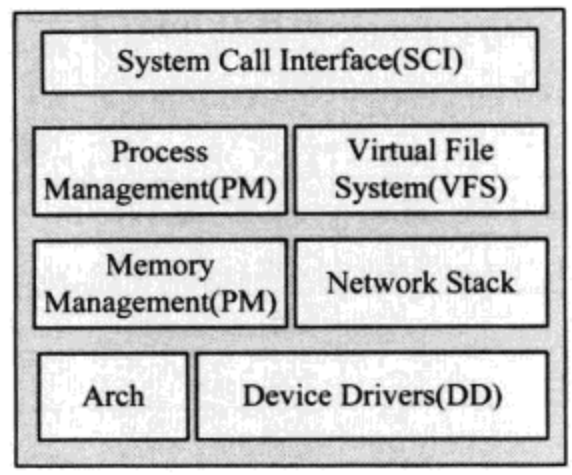


图 1-4　Linux 内核的子系统

（2）进程管理（Process Management）

进程管理的重点是管理进程的执行过程。在内核中，将处理器虚拟化为线程代码、数据、堆栈和 CPU 寄存器之后，这些进程又被称为线程。在用户空间，通常使用进程这个术语，不过 Linux 实现并没有区分进程和线程这两个概念。内核通过 SCI 提供了一个应用程序编程接口来创建一个新进程、停止进程并在它们之间进行通信和同步。

进程管理还包括处理活动进程之间共享 CPU 的需求。内核实现了一种新型的调度算法，不管有多少个线程在竞争 CPU，这种算法都可以在固定时间内进行操作。这种算法就称为 *O*(1)调度程序，这个名字就表示它调度多个线程所使用的时间和调度一个线程所使用的时间是相同的。*O*(1)调度程序也可以支持多处理器（称为对称多处理器或 SMP）。可以在./linux/kernel 中找到进程管理的源代码，在./linux/arch 中可以找到依赖于体系结构的源代码。

（3）内存管理（Memory Management）

内核所管理的另外一个重要资源是内存。Linux 包括了管理可用内存的方式，以及物理和虚拟映射所使用的硬件机制。

为了提高效率，如果由硬件直接管理虚拟内存，内存是按照所谓的内存页方式进行管理的（对于大部分体系结构来说都是 4KB）。不过内存管理要管理的可不止 4KB 缓冲区。Linux 提供了对 4KB 缓冲区的抽象，例如 slab 分配器。这种内存管理模式使用 4KB 缓冲区为基数，然后从中分配结构，并跟踪内存页使用情况，比如哪些内存页是满的，哪些页面没有完全使用，哪些页面为空。这样就允许该模式根据系统需要来动态调整内存使用。

为了支持多个用户使用内存，有时会出现可用内存被消耗光的情况。由于这个原因，页面可以移出内存并放入磁盘中。这个过程称为交换，因为页面会被从内存交换到硬盘上。内存管理的源代码可以在./linux/mm 中找到。

（4）虚拟文件系统（Virtual File System）

虚拟文件系统为文件系统在 Linux 内核中提供了一个通用的接口抽象。它在 SCI 和内核所支持的文件系统之间提供了一个交换层。

在虚拟文件系统之上，是对诸如 open、close、read 和 write 之类的函数的一个通用 API 抽象。在虚拟文件系统下面是文件系统抽象，它定义了上层函数的实现方式。它们是给定文件系统的插件。文件系统的源代码可以在./linux/fs 中找到。

文件系统层之下是缓冲区缓存，它为文件系统层提供了一个通用函数集。这个缓存区通过将数据保留一段时间，优化了对物理设备的访问。缓冲区缓存之下是设备驱动程序，它实现了特定物理设备的接口。

（5）网络堆栈（Network Stack）

网络堆栈在设计上遵循模拟协议本身的分层体系结构。Internet Protocol （IP） 是传输协议（通常称为传输控制协议或 TCP）下面的核心网络层协议。TCP 上面是 socket 层，它是通过 SCI 进行调用的。

socket 层是网络子系统的标准 API，它为各种网络协议提供了一个用户接口。从原始帧访问到 IP 协议数据单元（PDU），再到 TCP 和 User Datagram Protocol （UDP），socket 层提供了一种标准化的方法来管理连接，并在各个终点之间移动数据。内核中网络源代码可以在./linux/net 中找到。

（6）设备驱动程序（Device Drivers）

Linux 内核中有大量代码都在设备驱动程序中，它们能够运转特定的硬件设备。Linux 源码中提供了一个驱动程序子目录，这个目录又进一步划分为各种支持设备，例如 Bluetooth、I2C、serial 等。设备驱动程序的代码可以在./linux/drivers 中找到。

（7）依赖体系结构的代码（Arch）

尽管 Linux 很大程度上独立于所运行的体系结构，但是有些内容则必须考虑体系结构才能正常操作并实现更高效率。./linux/arch 子目录定义了内核源代码中依赖于体系结构的部分，其中包含了各种特定于体系结构的子目录（共同组成了 BSP）。对于一个典型的桌面系统来说，使用的是 i386 目录。每个体系结构子目录都包含了很多其他子目录，每个子目录都关注内核中的一个特定方面，例如引导、内核、内存管理等。这些依赖体系结构的代码可以在./linux/arch 中找到。

1.2.2 研究 Linux 内核的意义

研究 Linux 内核到底有哪些意义呢？这个问题是研读 Linux 内核的读者首先必须搞清楚的。

1．有助于开发实时的嵌入式操作系统

由于操作系统是所有应用软件的基础，所以，自己动手编写操作系统不仅对系统软件至关重要，而且对应用软件水平的提高也至关重要。一个性能稳定功能强大的操作系统的创造不仅对于国家来说是一个巨大的贡献，而且对于计算机行业系统内核的发展也起到了举足轻重的作用。但是如果搞自己的一套体系结构（PC98），不同国际标准兼容，其结果会严重阻碍软件业的发展。

Linux 内核的出现正符合所有的要求，其源代码公开，可以使我们尽快进行开发，加上全世界操作系统爱好者的共同努力，不仅加速了 Linux 操作系统的发展，并且可以保证操作系统中尽可能减少黑洞和隐蔽的问题。由于 Linux 是国际化的，也不必考虑兼容性问题，同时也可以融入国内外先进的技术。因此 Linux 操作系统内核对于操作系统开发人员而言，是学习和实用不可多得的工具。

2．有助于开发应用软件

实际上，目前 Linux 的源代码中包含了世界各地几百名计算机编程高手的作品，分析这些源代码对于掌握核心技术会起到事半功倍的作用，尤其是各种驱动程序的编写，对于把软硬件结合起来发展民族信息产业至关重要。要改变目前我国软件开发在低层次上的重复过程，必须掌握操作系统的核心技术。

3．有助于计算机科学教学

对于从事计算机科学教学和科研的人来说，对 Linux 内核的研究无疑是一个更好的选择，相比于 UNIX 而言它是免费的，而且核心代码也是开源的，在加上无数计算机爱好者的努力，网络的发展又使得我们可以接触到最新的技术。如果能对源代码的结构、原理及机制有清晰的理解而

且可以进行较详细地描述，那么对很多人深入了解源程序将帮助很大，而且可以让更多的人加入到这个队伍中，反过来可以更好的促进 Linux 内核的发展。

与其他的操作系统相比，Linux 也是更适合教学用的操作系统，一般的操作系统教材只讲操作系统的原理，学生觉得抽象，很难对操作系统有很直观的概念理解，而且由于很少和实际操作相结合，会使学生失去学习的兴趣。

由于 Linux 操作系统具有平台易于搭建，内核源码易于获得和结构简单等优点，目前国内外很多学校已经把它作为教学研究的学习之用。

1.3 Linux 运行的硬件基础

在深入学习 Linux 内核源代码之前，需要首先对 Linux 运行的硬件基础有个大概的认识，这里的硬件主要包括 CPU 中的寄存器以及磁盘。本节将对基于 i386 体系结构的 CPU 寄存器和系统指令进行详细介绍，并对基于段页机制的磁盘管理进行详细的介绍。

1.3.1 i386 寄存器和系统指令

在 Linux 系统中使用的主要包括 i386 寄存器中的 16 位标志寄存器，4 个内存管理寄存器，4 个控制寄存器以及调试寄存器。

1. 标志寄存器

在 8086CPU 中有种特殊的寄存器，它们用来存储 CPU 的状态，或某些指令执行的结果，用来控制 CPU 的工作方式。这些就是标志寄存器。

标志寄存器共 16 位，如图 1-5 所示。其中，第 1、3、5、12、13、14、15 位无任何意义。

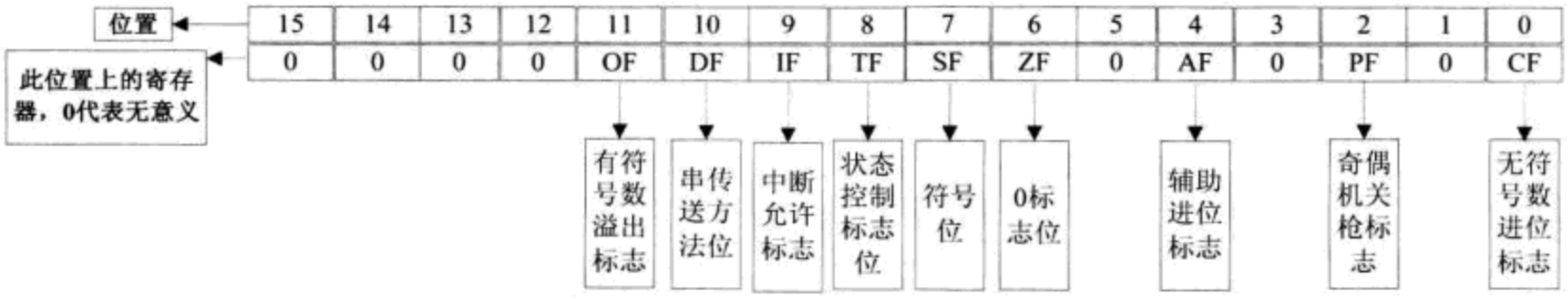


图 1-5 标志寄存器位置

各标志寄存器位置详解：

CF：（第 0 位）无符号数进位标志。当两个数相加且相加产生进位后的数据大于寄存器所能存储的数据的最大值时，CF=1；如 mov al,0F0H　add al,10H　即发生了进位，al 是 8 位寄存器，最大值为 FFH，而相加后的数据 100H 大于了它所能存储的最大值即发生的进位，而 al 中又不能存储这个进位的数据，则 CF＝1 代表发生的进程，而 AL 中存储 0。在 debug 中的 CF 的表现形式为：当 CF＝1 时 显示 CF；当 CF＝0 时 显示 NC。

PF：（第 2 位）奇偶校验位，是否为偶数，验证运算后结果的 1 的个数，如 mov ax,5 add ax,2 执行后其结果 7 的二进制位数 1 1 1，1 的个数为奇数，即 PF 的值为 0。PF 在 debug 中的表示为：为 1 时 PE，为 0 时 PO。

AF：（第 4 位）辅助进位标志符。在字操作时，当发生低字节向高字节进位或借位时，或在字节操作时，低四位向高四位进位或借位时，AF 的值为 1，其他情况 AF=0。

ZF：（第 6 位）0 标志位，代表是否为 0。当两个数据进行运算后，如果运算结果为 0，则 ZF＝1，否则 ZF=0；当 ZF＝1 时 在 debug 中显示 ZR，当 ZF=0 时，显示 NZ。

SF：（符 7 位）符号位，代表是否为负数。如果两数运算结果为负，则 SF=1，当在 debug 中，SF=1 时表现形式是 NG，为 0 时表现形式为 PL。

TF：（第 8 位）状态控制标志位，是用来控制 CPU 操作的，它们要通过专门的指令才能使之发生改变。

IF：（第 9 位）中断允许标志位，决定 CPU 是否响应 CPU 外部的可屏蔽中断发出的中断请求。在不管 IF 为何值，CPU 必须响应 CPU 外部发出不可屏蔽中断请求和 CPU 内部的请求。IF＝1，CPU 可以响应外部发出的可屏蔽中断请求，为 0 时不响应 STI 设置 IF＝1　CLI　设置 IF＝0。

DF：（第 10 位）串传送方向标志位。将 DS:SI 中的数据送入 ES:DI 中，如果 DF 为 0；SI，DI 递增传送，DF＝1 时；SI,DI 递减传送，传送长度为 CX；在 debug 中 DF=0 时显示为 DN，当 DF＝1 时显示为 UP。

OF：（第 11 位）有符号数溢出标志位。标志当两个有符号数运算时，是否发生了溢出。如 AL 为 8 位寄存器，则它的存储范围在-128～127 之间 即 80H～7FH 之间，例：mov al,041H add al,al 其结果为 82 超出了表示范围，则 OF＝1。在 debug 中，OF＝1 时显示为 OV，当 OF＝0 时显示为 NV。

2．内存管理寄存器

处理器提供了 4 个内存管理寄存器（GDTR、IDTR、LDTR 和 TR），用于指定内存分段管理所用系统表的基地址，处理器为这些寄存器的加载和保存提供了特定的指令。

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

（1）全局描述符表寄存器 GDTR

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

（2）中断描述符表寄存器 IDTR

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

（3）局部描述符表寄存器 LDTR

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

（4）任务寄存器 TR

TR 寄存器用于存放当前任务 TSS 段的 16 位段选择符、32 位基地址、16 位段长度和描述符属性值。它引用 GDT 表中的一个 TSS 类型的描述符。指令 LTR 和 STR 分别用于加载和保存 TR

寄存器的段选择符部分。当使用 LTR 指令把选择符加载进任务寄存器时，TSS 描述符中的段基地址、段限长度以及描述符属性会被自动加载到任务寄存器中。当执行任务切换时，处理器会把新任务的 TSS 的段选择符和段描述符自动加载进任务寄存器 TR 中。

3. 控制寄存器

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

（1）CR0 中协处理器控制位

CR0 的 4 个位：扩展类型位 ET、任务切换位 TS、仿真位 EM 和数学存在位 MP 用于控制 80x86 浮点（数学）协处理器的操作。CR0 的 ET 位（标志）用于选择与协处理器进行通信所使用的协议，即指明系统中使用的是 80387 还是 80287 协处理器。TS、MP 和 EM 位用于确定浮点指令或 WAIT 指令是否应该产生一个设备不存在（Device Not Available，DNA）异常。这个异常可用来仅为使用浮点运算的任务保存和恢复浮点寄存器。对于没有使用浮点运算的任务，这样做可以加快它们之间的切换操作。

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

② TS：CR0 的位 3 是任务已切换（Task Switched）标志。该标志用于推迟保存任务切换时的协处理器内容，直到新任务开始实际执行协处理器指令。处理器在每次任务切换时都会设置该标志，并且在执行协处理器指令时测试该标志。

如果设置了 TS 标志并且 CR0 的 EM 标志为 0，那么在执行任何协处理器指令之前会产生一个设备不存在异常。如果设置了 TS 标志但没有设置 CR0 的 MP 和 EM 标志，那么在执行协处理器指令 WAIT/FWAIT 之前不会产生设备不存在异常。如果设置了 EM 标志，那么 TS 标志对协处理器指令的执行无影响，CR0 中标志 EM、MP 和 TS 的不同组合对协处理器指令动作的影响如表 1-2 所示。

表 1-2　CR0 中标志 EM、MP 和 TS 的不同组合对协处理器指令动作的影响

CR0 中的标志			指 令 类 型	
EM	MP	TS	浮点	WAIT/FWAIT
0	0	0	执行	执行
0	0	1	设备不存在（DNA）异常	执行
0	1	0	执行	执行
0	1	1	DNA 异常	DNA 异常
1	0	0	DNA 异常	执行
1	0	1	DNA 异常	执行
1	1	0	DNA 异常	执行
1	1	1	DNA 异常	DNA 异常

在任务切换时，处理器并不自动保存协处理器的上下文，而是会设置 TS 标志。这个标志会

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

③ EM：CR0 的位 2 是仿真（EMulation）标志。当设置该位时，表示处理器没有内部或外部协处理器，执行协处理器指令时会引起设备不存在异常；当清除时，表示系统有协处理器。设置这个标志可以迫使所有浮点指令使用软件来模拟。

④ MP：CR0 的位 1 是监控协处理器（Monitor coProcessor 或 Math Present）标志。用于控制 WAIT/FWAIT 指令与 TS 标志的交互作用。如果 MP=1、TS=1，那么执行 WAIT 指令将产生一个设备不存在异常；如果 MP=0，则 TS 标志不会影响 WAIT 的执行。

（2）CR0 中保护控制位

① PE：CR0 的位 0 是启用保护（Protection Enable）标志。当设置该位时即开启了保护模式；当复位时即进入实地址模式。这个标志仅开启段级保护，而并没有启用分页机制。若要启用分页机制，那么 PE 和 PG 标志都要置位。

② PG：CR0 的位 31 是分页（Paging）标志。当设置该位时即开启了分页机制；当复位时则禁止分页机制，此时所有线性地址等同于物理地址。在开启这个标志之前必须已经或者同时开启 PE 标志。即若要启用分页机制，那么 PE 和 PG 标志都要置位。

③ WP：对于 Intel 80486 或以上的 CPU，CR0 的位 16 是写保护（Write Proctect）标志。当设置该标志时，处理器会禁止超级用户程序（例如特权级 0 的程序）向用户级只读页面执行写操作；当该位复位时则反之。该标志有利于 UNIX 类操作系统在创建进程时实现写时复制（Copy on Write）技术。

④ NE：对于 Intel 80486 或以上的 CPU，CR0 的位 5 是协处理器错误（Numeric Error）标志。当设置该标志时，就启用了 x87 协处理器错误的内部报告机制；若复位该标志，那么就使用 PC 形式的 x87 协处理器错误报告机制。当 NE 为复位状态并且 CPU 的 IGNNE 输入引脚有信号时，那么数学协处理器 x87 错误将被忽略。当 NE 为复位状态并且 CPU 的 IGNNE 输入引脚无信号时，那么非屏蔽的数学协处理器 x87 错误将导致处理器通过 FERR 引脚在外部产生一个中断，且在执行下一个等待形式浮点指令或 WAIT/FWAIT 指令之前立刻停止指令执行。CPU 的 FERR 引脚用于仿真外部协处理器 80387 的 ERROR 引脚，因此通常连接到中断控制器输入请求引脚上。NE 标志、IGNNE 引脚和 FERR 引脚用于利用外部逻辑来实现 PC 形式的外部错误报告机制。

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

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

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

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

（3）CR2 和 CR3

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

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

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

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

4．调试寄存器

Intel 80386 以上的 CPU 提供了调试寄存器以用于软件调试。386 和 486 包括 6 个调试寄存器：Dr0，Dr1，Dr2，Dr3，Dr6 和 Dr7。这些寄存器全是 32 位，如图 1-6 所示。

图 1-6　VFS 文件系统框架

Dr0～3 用于设置硬件断点，即在调试器中经常使用的 bpm 断点，由于只有 4 个断点寄存器，所以最多只能设置 4 个 bpm 断点。Dr7 是一些控制位，用于控制断点的方式，Dr6 用于显示引起断点的原因，如果是 Dr0～3 或单步（EFLAGS 的 TF）或由于 GD 置位时访问调试寄存器引起 1 号调试陷阱的话，则相应设置对应的位。

下面对 Dr6 和 Dr7 的对应位做一些详细介绍。

调试控制寄存器 Dr7，各个位的含义如下：

位 0 L0 和位 1 G0：用于控制 Dr0 是全局断点还是局部断点，如果 G0 置位则是全局断点，L0 置位则是局部断点。

G1L1～G3L3 用于控制 D1～Dr3，其功能同上。

LEN0：占两个位，开始于位 15，用于控制 Dr0 的断点长度，可能取值：

00：1 字节。

01：2 字节。

10：保留。

11：4 字节。

RWE0：从第 17 位开始，占两个位，控制 Dr0 的断点是读、写还是执行断点或是 I/O 端口断点：

00：只执行。

01：写入数据断点。

10：I/O 端口断点（只用于 pentium+，需设置 CR4 的 DE 位）。

11：读或写数据断点。

RWE1～3，LEN1～3 分别用于控制 Dr1～3 的断点方式，含义如上。

GD：用于保护 DRx，如果 GD 位为 1，则对 Drx 的任何访问都会导致进入 1 号调试陷阱。即 IDT 的对应入口，这样可以保证调试器在必要的时候完全控制 Drx。

调试状态寄存器 Dr6，该寄存器用于表示进入陷阱 1 的原因，各个位的含义如下：

B0～B3，如果其中任何一个位置位，则表示是相应的 Dr0~3 断点引发的调试陷阱。但还须注意的是，有时候不管 GiLi 如何设置，只要是遇到 Drx 指定的断点，总会设置 Bi，如果看到多个 Bi 置位，则可以通过 GiLi 的情况判断究竟是哪个 Dr 寄存器引发的调试陷阱。

BD 置位表示是 GD 位置位情况下访问调试寄存器引发的陷阱。

BT 置位表示是因为 TS 置位，即任务切换时 TSS 中 TS 位置 1 时，切到第二个任务时第一条指令时引发的。

BS 置位表示是单步中断引发的断点，即 EFLAGS 的 TF 置位时引发的调试陷阱。

注意 I/O 端口断点是 586+以上 CPU 才有的功能，受 CR4 的 DE 位的控制，DE 为 1 才有效（DE 是 CR4 的第 3 位）。

1.3.2 分段机制

分段机制可以用于实现多种系统的设计。分段机制就是把虚拟地址空间中的虚拟内存组织成一些长度可变的称为段的内存块单元。段是虚拟地址到线性地址转换机制的基础，每个段由三个参数定义：段基地址；段限长；段属性。多个段映射到线性地址中的范围可以部分重叠或者覆盖，甚至完全重叠。

段的基地址、段属性、段限长存储在一个叫做段描述符的结构项中，在逻辑地址到线性地址映射过程中会用到这个描述符。段描述符保存在内存中的段描述表中，段描述表是一个保存描述符项的简单数组。逻辑地址由 16 位段选择符和 32 位偏移量组成，段选择符指定字节所在的段，

偏移量指定该字节相对于段基地址的位置。线性地址是处理器线性地址空间的 32 位地址。线性地址中含有为系统定义的所有段和系统表。

1. 分段机制概述

分段机制可用于实现多种系统设计。这些设计范围是从使用分段机制的最小功能来保护程序的平坦模型，到使用分段机制创建一个可同时可靠地运行多个程序（或任务）的具有稳固操作环境的多段模型。多段模型能够利用分段机制全部功能提供由硬件增强的代码、数据结构、程序和任务的保护措施。通常，每个程序（或任务）都使用自己的段描述符表以及自己的段。对程序来说，段可以完全是私有的，或者是程序之间共享的。对所有段以及系统上运行程序各自执行环境的访问都由硬件控制。访问检查不仅能够用来保护对段界限以外地址的引用，而且也能用来在某些段中防止执行不允许的操作。例如，因为代码段被设计成只读形式的段，因此可以用硬件来防止对代码段执行写操作。段中的访问权限信息也可以用来设置保护环或级别。保护级别可用于保护操作系统程序不受应用程序非法访问。

2. 段的定义

保护模式中 80×86 提供了 4GB 的物理地址空间。这是处理器在其地址总线上可以寻址的地址空间。这个地址空间是平坦的，地址范围从 0 到 0xFFFFFFFF。这个物理地址空间可以映射到可读写内存、只读内存以及内存映射 I/O 中。分段机制就是把虚拟地址空间中的虚拟内存组织成一些长度可变的称为段的内存块单元。80386 虚拟地址空间中的虚拟地址（逻辑地址）由一个段部分和一个偏移部分构成。段是虚拟地址到线性地址转换机制的基础。每个段由以下几个参数定义：

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

段限长定义了在虚拟地址空间中段的大小。段基地址和段限长定义了段所映射的线性地址范围或区域。段内 0 到 limit 的地址范围对应线性地址中范围 Base 到 Base+Limit。偏移量大于段限长的虚拟地址是无意义的，如果使用则会导致异常。另外，若访问一个段并没有得到段属性许可则也会导致异常。例如，如果你试图写一个只读的段，那么 80386 就会产生一个异常。另外，多个段映射到线性地址中的范围可以部分重叠或覆盖，甚至完全重叠，如图 1-7 所示。在本书介绍的 Linux 0.1x 系统中，一个任务的代码段和数据段的段限长相同，并被映射到线性地址完全相同而重叠的区域上。

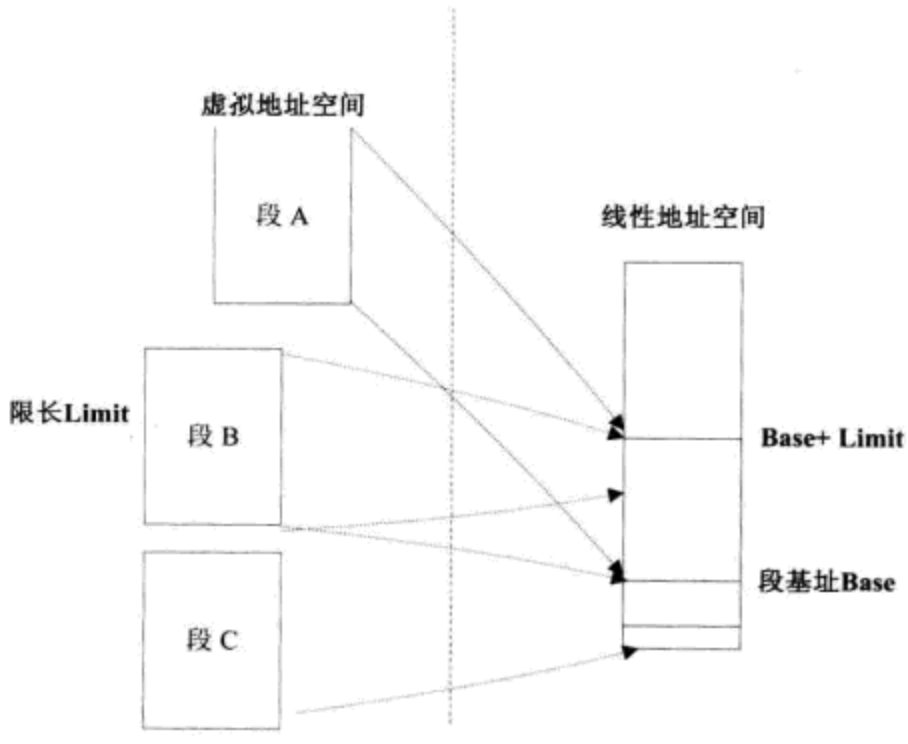


图 1-7　虚拟（逻辑）地址空间中的段映射到线性地址空间

段的基地址、段限长以及段的保护属性存储在一个称为段描述符（Segment Descriptor）的结构项中。在逻辑地址到线性地址的转换映射过程中会使用这个段描述符。段描述符保存在内存中

的段描述符表（Descriptor Table）中。段描述符表是包含段描述符项的一个简单数组。前面介绍的段选择符即用于通过指定表中一个段描述符的位置来指定相应的段。

即使利用段的最小功能，使用逻辑地址也能访问处理器地址空间中的每个字节。逻辑地址由 16 位的段选择符和 32 位的偏移量组成，如图 1-8 所示。段选择符指定字节所在的段，而偏移量指定该字节在段中相对于段基地址的位置。处理器会把每个逻辑地址转换成线性地址。线性地址是处理器线性地址空间中的 32 位地址。与物理地址空间类似，线性地址空间也是平坦的 4GB 地址空间，地址范围从 0 到 0xFFFFFFFF。线性地址空间中含有为系统定义的所有段和系统表。

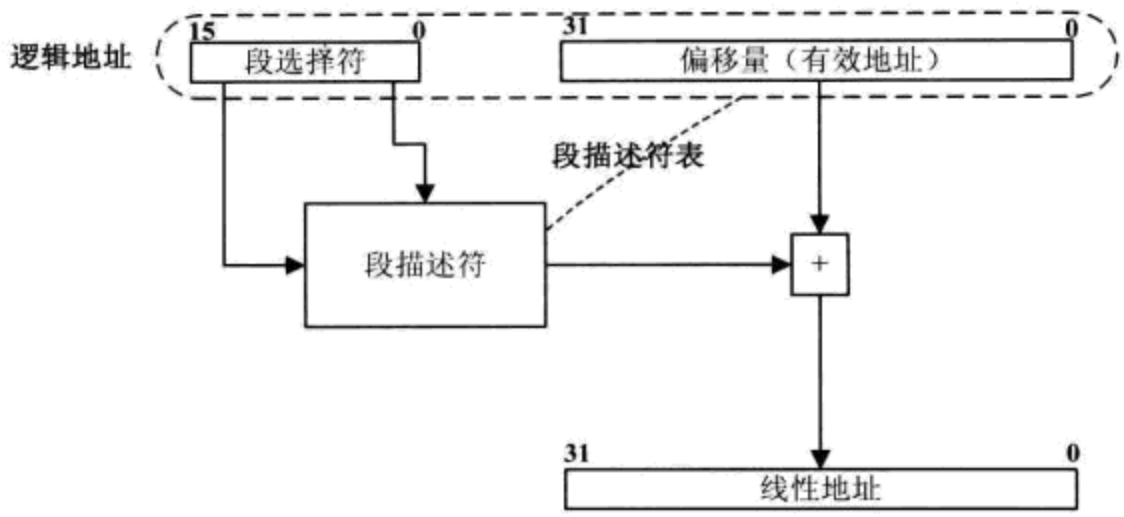


图 1-8　逻辑地址到线性地址的变换过程

为了把逻辑地址转换成一个线性地址，处理器会执行以下操作：

① 段选择符中的偏移值（段索引）在 GDT（全局描述符表）或 LDT（局部描述符表）表中定位相应的段描述符（仅当一个新的段选择符加载到段寄存器中时才需要这一步）。

② 利用段描述符检验段的访问权限和范围，以确保该段是可访问的并且偏移量位于段界限内。

③ 把段描述符中取得的段基地址加到偏移量上，最后形成一个线性地址。

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

3. 段描述符表

段描述符表是段描述符的一个数组，如图 1-9 所示。描述符表的长度可变，最多可以包含 8192 个 8 字节描述符。有两种描述符表：全局描述符表（Global Descriptor Table，GDT）和局部描述符表（Local Descriptor Table，LDT）。

描述符表存储在由操作系统维护的特殊数据结构中，并且由处理器的内存管理硬件来引用。这些特殊结构应该保存在仅由操作系统软件访问的受保护的内存区域中，以防止应用程序修改其中的地址转换信息。虚拟地址空间共含有 214 个字段，被分割成大小相等的两半。一半由 GDT 来映射变换到线性地址，另一半则由 LDT 来映射。

通过指定一个描述符表（GDT 或 LDT）以及表中描述符号，就可以定位一个描述符。当发生任务切换时，LDT 会更换成新任务的 LDT，但是 GDT 并不会改变。因此，GDT 所映射的一半虚拟地址空间是系统中所有任务共有的，但是 LDT 所映射的另一半则在任务切换时被改变。系统中所有任务共享的段由 GDT 来映射。这样的段通常包括含有操作系统的段以及所有任务各自的包含 LDT 的特殊段。LDT 段可以想象成属于操作系统的数据。

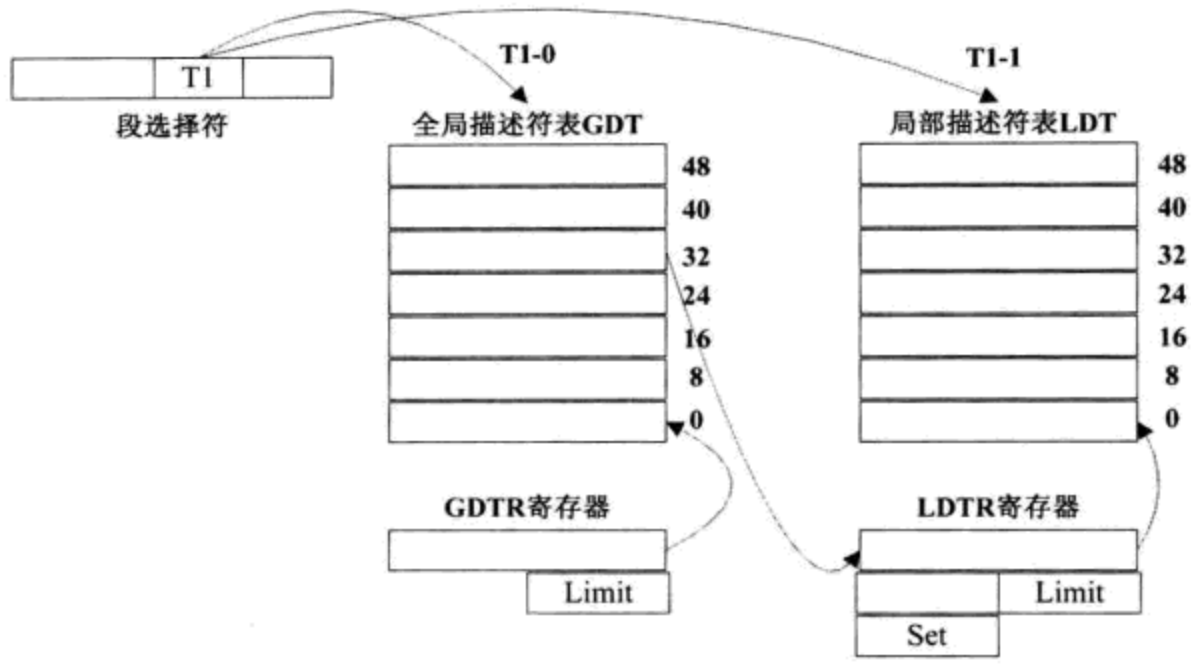


图 1-9　段描述符表结构

图 1-10 表明一个任务中的段如何能在 GDT 和 LDT 之间分开。图中共有 6 个段，分别用于两个应用程序（A 和 B）以及操作系统。系统中每个应用程序对应一个任务，并且每个任务有自己的 LDT。应用程序 A 在任务 A 中运行，拥有 LDT_A，用来映射段 $Code_A$ 和 $Data_A$。类似地，应用程序 B 在任务 B 中运行，使用 LDT_B 来映射 $Code_B$ 和 $Data_B$ 段。包含操作系统内核的两个段 $Code_{OS}$ 和 $Data_{OS}$ 使用 GDT 来映射，这样它们可以被两个任务所共享。两个 LDT 段：LDT_A 和 LDT_B 也使用 GDT 来映射。

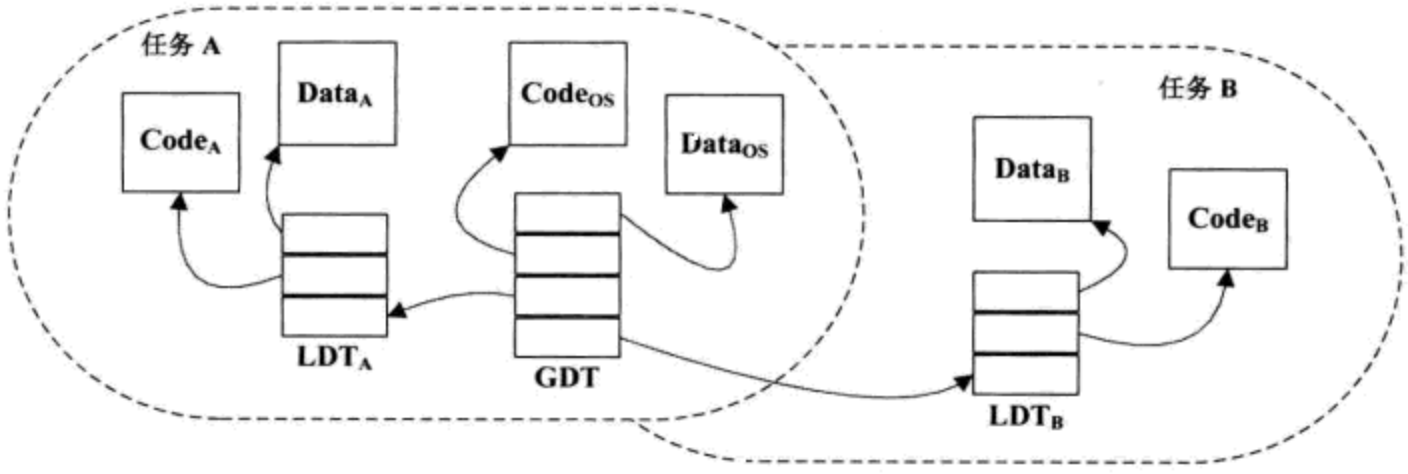


图 1-10　任务所用的段类型

当任务 A 在运行时，可访问的段包括 LDT_A 映射的 $Code_A$ 和 $Data_A$ 段，加上 GDT 映射的操作系统的段 $Code_{OS}$ 和 $Data_{OS}$。当任务 B 在运行时，可访问的段包括 LDT_B 映射的 $Code_B$ 和 $Data_B$ 段，加上 GDT 映射的段。这个例子通过让每个任务使用不同的 LDT，演示了虚拟地址空间如何能够被组织隔离每个任务。

当任务 A 在运行时，任务 B 的段不是虚拟地址空间的部分，因此任务 A 没有办法访问任务 B 的内存。同样地，当任务 B 运行时，任务 A 的段也不能被寻址。这种使用 LDT 来隔离每个应用程序任务的方法，正是关键保护需求之一。每个系统必须定义一个 GDT，并可用于系统中所有程序或任务。另外，可选定义一个或多个 LDT。例如，可以为每个运行任务定义一个 LDT，或者某些或所有任务共享一个 LDT。GDT 本身并不是一个段，而是线性地址空间中的一个数据结构。GDT 的基线性地址和长度值必须加载进 GDTR 寄存器中。

GDT 的基地址应该进行内存 8 字节对齐，以达到最佳处理器性能。GDT 的限长以字节为单

位。与段类似，限长值加上基地址可得到最后表中最后 1 字节的有效地址。限长为 0 表示有 1 个有效字节。因为段描述符总是 8 字节长，因此 GDT 的限长值应该设置成总是 8 的倍数减 1（即 8n-1）。处理器并不使用 GDT 中的第 1 个描述符。把这个“空描述符”的段选择符加载进一个数据段寄存器（DS、ES、FS 或 GS）并不会产生一个异常，但是若使用这些加载了空描述符的段选择符访问内存时就肯定会产生一般保护性异常。

通过使用这个段选择符初始化段寄存器，那么意外引用未使用的段寄存器肯定会产生一个异常。LDT 表存放在 LDT 类型的系统段中。此时 GDT 必须含有 LDT 的段描述符。如果系统支持多 LDT 的话，那么每个 LDT 都必须在 GDT 中有一个段描述符和段选择符。一个 LDT 的段描述符可以存放在 GDT 表的任何地方。访问 LDT 需使用其段选择符。为了在访问 LDT 时减少地址转换次数，LDT 的段选择符、基地址、段限长以及访问权限需要存放在 LDTR 寄存器中。当保存 GDTR 寄存器内容时（使用 SGDT 指令），一个 48 位的“伪描述符”被存储在内存中。

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

4．段选择符

段选择符（或称段选择子）是段的一个 16 位标识符，如图 1-11 所示。段选择符并不直接指向段，而是指向段描述符表中定义段的段描述符。段选择符的 3 个字段分别是：

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

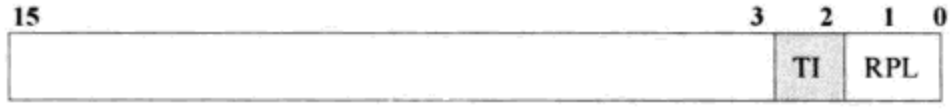


图 1-11　段选择符结构

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

例如，图 1-12（a）中选择符（0x08）指定了 GDT 中具有 RPL=0 的段 1，其索引字段值是 1，TI 位是 0，指定 GDT 表。图 1-12（b）中选择符（0x10）指定了 GDT 中具有 RPL=0 的段 2，其索引字段值是 2，TI 位是 0，指定 GDT 表。图 1-12（c）中选择符（0x0f）指定了 LDT 中具有 RPL=3 的段 1，其索引字段值是 1，TI 位是 1，指定 LDT 表。图 1-12（d）中选择符（0x17）指定了 LDT 中具有 RPL=3 的段 2，其索引字段值是 2，TI 位是 1，指定 LDT 表。实际上，图中的前 4 个选择符 a～d 分别就是 Linux 0.1x 内核的内核代码段、内核数据段、任务代码段和任务数据段的选择符。图 1-12（e）中的选择符（0xffff）指定 LDT 表中 RPL=3 的段 8191。其索引字段值是 0b1111111111111（即 8191），TI 位等于 1，指定 LDT 表。

另外，处理器不使用 GDT 表中的第 1 项。指向 GDT 该项的选择符（即索引值为 0，TI 标志为 0 的选择符）用作“空选择符”。当把空选择符加载到一个段寄存器（除了 CS 和 SS 以外）中

时，处理器并不产生异常。但是当使用含有空选择符的段寄存器用于访问内存时就会产生异常。当把空选择符加载到 CS 或 SS 段寄存器中时将会导致一个异常。

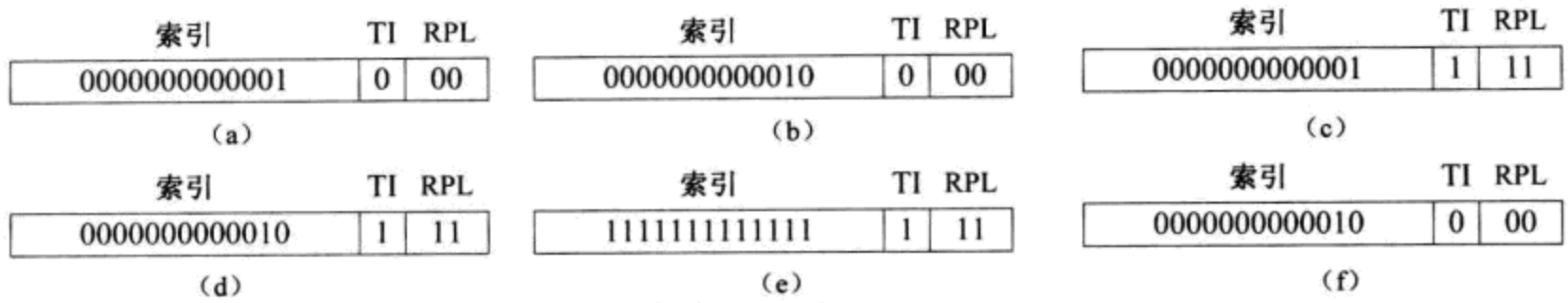


图 1-12　段选择符示例

对应用程序来说段选择符是作为指针变量的一部分而可见，但选择符的值通常是由链接编辑器或链接加载程序进行设置或修改，而非应用程序。为减少地址转换时间和编程复杂性，处理器提供可存放最多 6 个段选择符的寄存器，如图 1-13 所示，即段寄存器。每个段寄存器支持特定类型的内存引用（代码、数据或堆栈）。

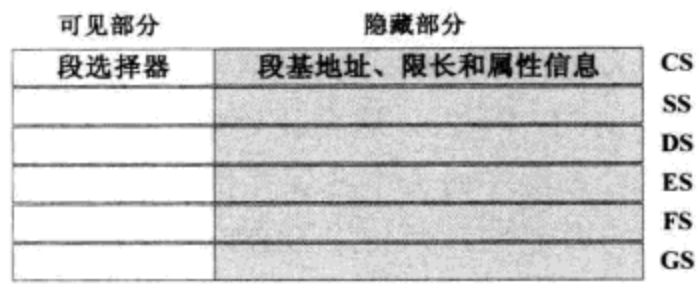


图 1-13　段寄存器结构

原则上执行每个程序都起码需要把有效的段选择符加载到代码段（CS）、数据段（DS）和堆栈段（SS）寄存器中。处理器还另外提供 3 个辅助的数据段寄存器（ES、FS 和 GS），以便当前执行程序（或任务）能够访问其他几个数据段。

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

另外，为了避免每次访问内存时都去引用描述符表、读和解码一个段描述符，每个段寄存器都有一个“可见”部分和一个“隐藏”部分（隐藏部分也被称为“描述符缓冲”或“影子寄存器”）。当一个段选择符被加载到一个段寄存器可见部分中时，处理器也同时把段选择符指向的段描述符中的段地址、段限长以及访问控制信息加载到段寄存器的隐藏部分中。

缓冲在段寄存器（可见和隐藏部分）中的信息使得处理器可以在进行地址转换时不再需要花费时间从段描述符中读取基地址和限长值。由于影子寄存器含有描述符信息的一个副本，因此操作系统必须确保对描述符表的改动反映在影子寄存器中。处理这种问题最简捷的方法是在对描述符表中的描述符做过任何改动之后就立刻重新加载 6 个段寄存器，这将把描述符表中的相应段信息重新加载到影子寄存器中，并为加载段寄存器提供了两类加载指令：

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

MOV 指令当然也可以用于把段寄存器可见部分内容存储到一个通用寄存器中。

5. 段描述符

前面已经说明了使用段选择符来定位描述符表中的一个描述符。段描述符是 GDT 和 LDT 表中的一个数据结构项，用于向处理器提供有关一个段的位置和大小信息以及访问控制的状态信息。每个段描述符的长度是 8 字节，含有 3 个主要字段：段基地址、段限长和段属性。段描述符通常由编译器、链接器、加载器或者操作系统来创建，但绝不是应用程序。图 1-14 给出了所有类型段描述符的一般格式。

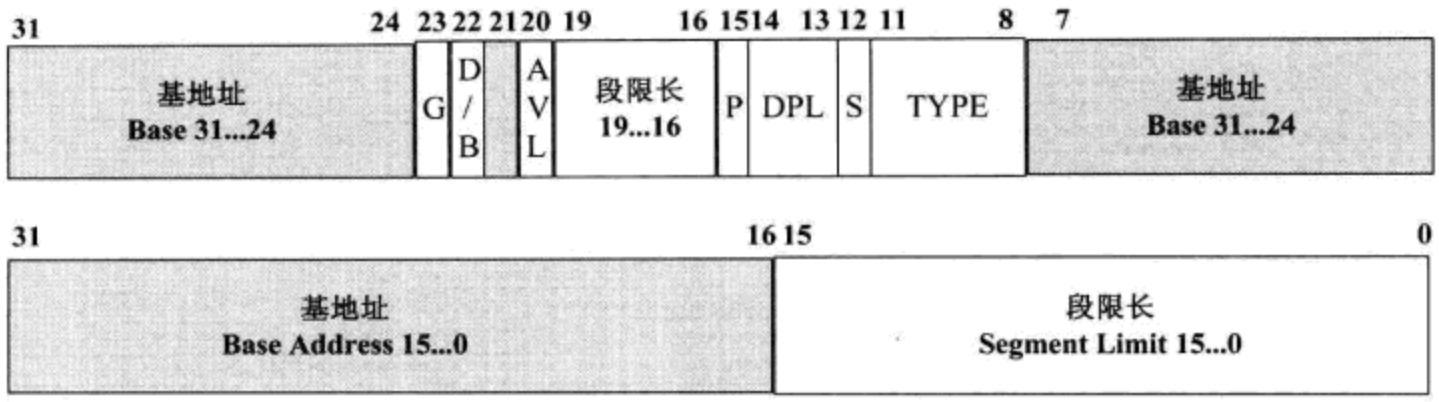


图 1-14　段描述符通用格式

一个段描述符中各字段和标志的含义如下：

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

根据段类型中的段扩展方向标志 E，处理器以两种不同方式使用段限长 Limit。对于向上扩展的段（简称上扩段），逻辑地址中的偏移值范围可以从 0 到段限长值 Limit。大于段限长 Limit 的偏移值将产生一般保护性异常。

对于向下扩展的段（简称下扩段），段限长 Limit 的含义相反。根据默认栈指针大小标志 B 的设置，偏移值范围可从段限长 Limit 到 0xFFFFFFFF 或 0xFFFF。而小于段限长 Limit 的偏移值将产生一般保护性异常。对于下扩段，减小段限长字段中的值会在该段地址空间底部分配新的内存，而不是在顶部分配。80x86 的栈总是向下扩展的，因此这种实现方式很适合扩展堆栈。

② 基地址字段 Base（Base address field）：该字段定义在 4GB 线性地址空间中一个段字节 0 所处的位置。处理器会把 3 个分立的基地址字段组合形成一个 32 位的值。段基地址应该对齐 16 字节边界。虽然这不是要求的，但通过把程序的代码和数据段对齐在 16 字节边界上，可以让程序具有最佳性能。

③ 段类型字段 TYPE（Type field）：用行指定段或门（Gate）的类型、说明段的访问种类以及段的扩展方向。该字段的解释依赖于描述符类型标志 S 指明是一个应用（代码或数据）描述符还是一个系统描述符。TYPE 字段的编码对代码、数据或系统描述符都不同，如图 1-15 所示。

④ 描述符类型标志 S（Descriptor type flag）：用于指明一个段描述符是系统段描述符（S=0）还是代码或数据段描述符（S=1）。

⑤ 描述符特权级字段 DPL（Descriptor privilege level）：用于指明描述符的特权级。特权级范围从 0 到 3。0 级特权级最高，3 级最低。DPL 用于控制对段的访问。

⑥ 段存在标志 P（Segment present）：用于指出一个段是在内存中（P=1）还是不在内存中（P=0）。当一个段描述符的 P 标志为 0 时，那么把指向这个段描述符的选择符加载进段寄存器将导致产生一个段不存在异常。内存管理软件可以使用这个标志来控制在某一给定时间实际需要把哪个段加载进内存中。

这个功能为虚拟存储提供了除分页机制以外的控制。图 1-16 给出了当 P=0 时的段描述符格式。当 P 标志为 0 时，操作系统可以自由使用格式中标注为可用（Available）的字段位置来保存自己的数据，例如有关不存在段实际在什么地方的信息。

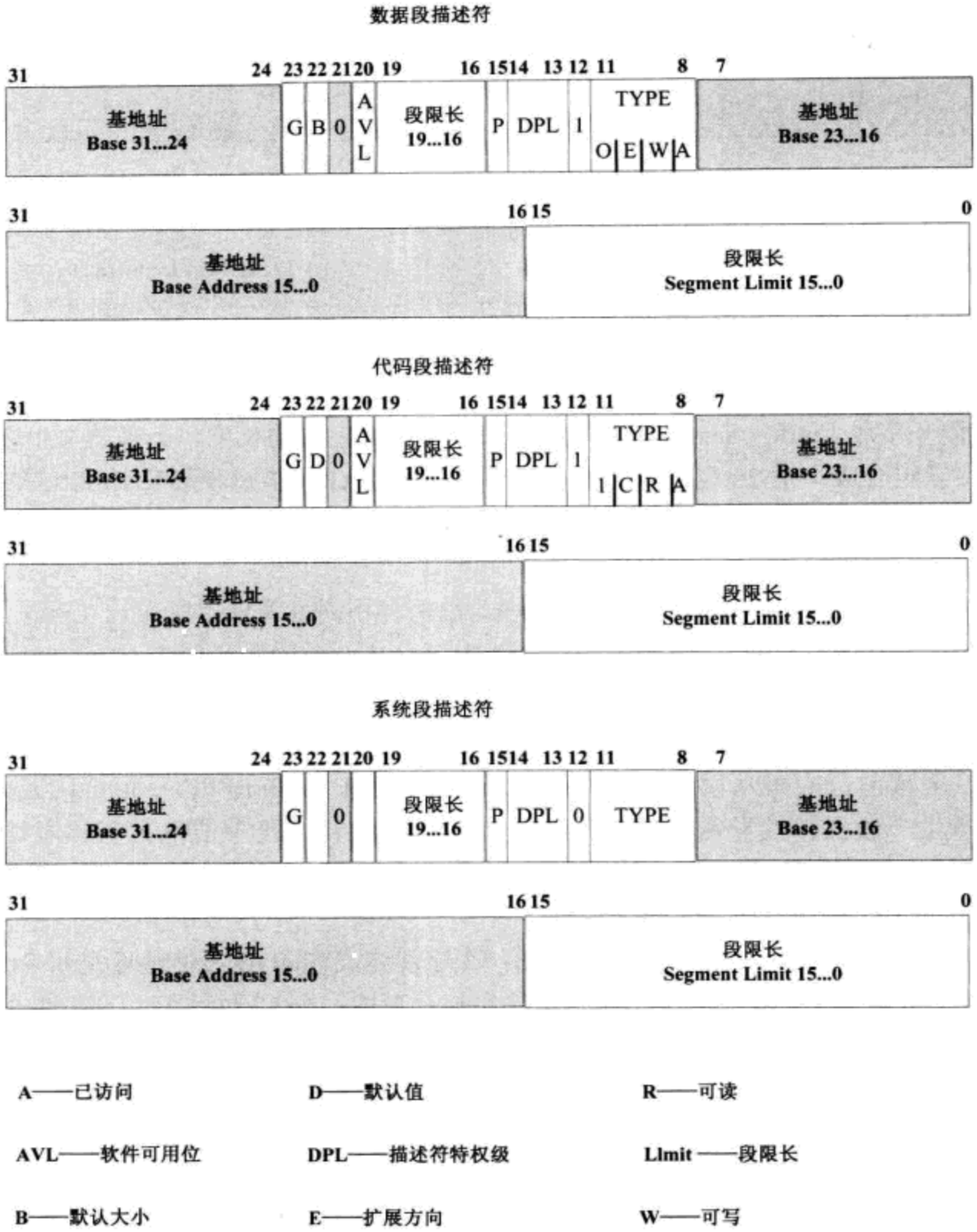


图 1-15　代码段、数据段和系统段描述符格式

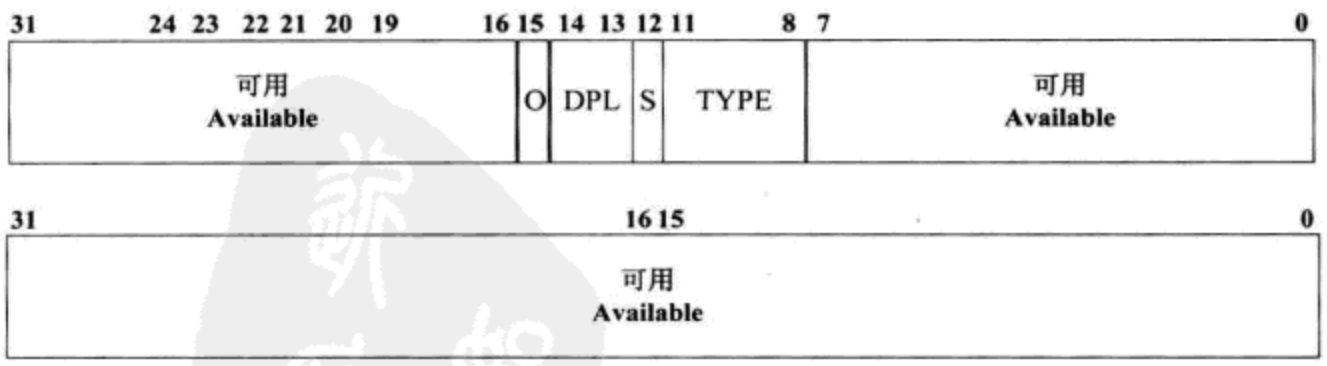


图 1-16　当存在位 P=0 时的段描述符格式

⑦ D/B(默认操作大小/默认栈指针大小和/或上界限)标志(Default operation size/default stack pointer size and/or upper bound)：根据段描述符描述的是一个可执行代码段、下扩数据段还是一个堆栈段，该标志具有不同的功能。(对于 32 位代码和数据段，这个标志应该总是设置为 1；对

于 16 位代码和数据段，这个标志被设置为 0）。

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

⑧ 颗粒度标志 G（Granularity）：该字段用于确定段限长字段 Limit 值的单位。如果颗粒度标志为 0，则段限长值的单位是字节；如果设置了颗粒度标志，则段限长值使用 4KB 单位（这个标志不影响段基地址的颗粒度，基地址的颗粒度总是字节单位）。若设置了 G 标志，那么当使用段限长来检查偏移值时，并不会去检查偏移值的 12 位最低有效位。例如，当 G=1 时，段限长为 0 表明有效偏移值为 0～4095。

⑨ 可用和保留位（Available and reserved bits）：段描述符第 2 个双字的位 20 可供系统软件使用；位 21 是保留位并应该总是设置为 0。

1.3.3 分页机制

分页机制是 80x86 内存管理机制的第二部分，它在分段机制的基础上完成虚拟（逻辑）地址到物理地址转换的过程。分段机制把逻辑地址转换成线性地址，而分页则把线性地址转换成物理地址。分页可以用于任何一种分段模型。处理器分页机制会把线性地址空间（段已映射到其中）划分成页面，然后这些线性地址空间页面被映射到物理地址空间的页面上。分页机制有几种页面级保护措施，可和分段机制保护机制合用或替代分段机制的保护措施。例如，在基于页面的基础上可以加强读/写保护。另外，在页面单元上，分页机制还提供了用户、超级用户两级保护。

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

前面介绍的分段机制在各种可变长度的内存区域上操作。与分段机制不同，分页机制对固定大小的内存块（称为页面）进行操作。分页机制把线性和物理地址空间都划分成页面。线性地址空间中的任何页面可以被映射到物理地址空间的任何页面上。图 1-17 显示了分页机制如何把线性和物理地址空间都划分成各个页面，并在这两个空间之间提供了任意映射。图中的箭头把线性地址空间中的页面与物理地址空间中的页面对应起来。

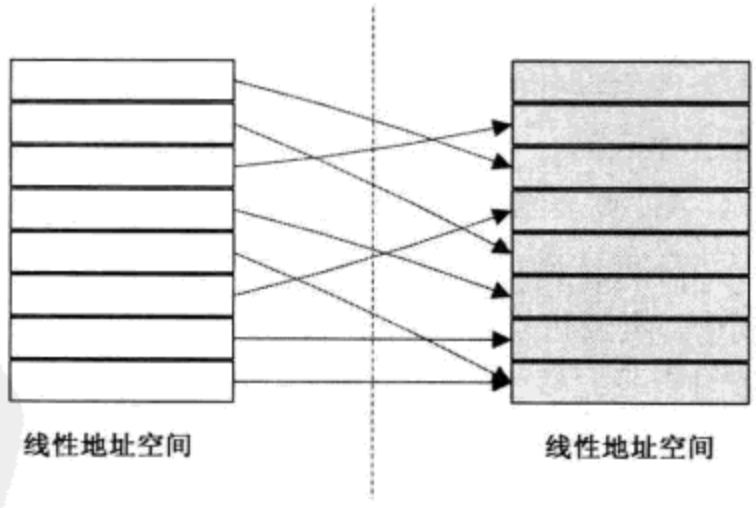


图 1-17 线性地址空间页面到物理地址空间页面对应示意图

80x86 使用 4KB（212）固定大小的页面。每个页面均是 4KB，并且对齐于 4KB 地址边界处。

分页机制通过把线性地址空间中的页面重新定位到物理地址空间中进行操作。由于 4KB 大小的页面作为一个单元进行映射，并且对齐于 4KB 边界，因此线性地址的低 12 位可作为页内偏移量直接作为物理地址的低 12 位。分页机制执行的重定位功能可看做把线性地址的高 20 位转换到对应物理地址的高 20 位。

另外，线性到物理地址的转换功能被扩展成允许一个线性地址被标注为无效的，而非让其产生一个物理地址。在以下两种情况下一个页面可以被标注为无效：

- 操作系统不支持的线性地址；
- 对应在虚拟内存系统中的页面在磁盘上而非在物理内存中。

在第一种情况下，产生无效地址的程序必须被终止。在第二种情况下，该无效地址实际上是请求操作系统虚拟内存管理器把对应页面从磁盘上加载到物理内存中，以供程序访问。因为无效页面通常与虚拟存储系统相关，因此它们被称为不存在的页面，并且由页表中称为存在（present）的属性来确定。在保护模式中，80x86 允许线性地址空间直接映射到大容量的物理内存（如 4GB 的 RAM）上，或者（使用分页）间接地映射到较小容量的物理内存和磁盘存储空间中。后一种映射线性地址空间的方法被称为虚拟存储或者需求页（Demand-paged）虚拟存储。

当使用分页时，处理器会把线性地址空间划分成固定大小的页面（长度 4KB），这些页面可以映射到物理内存中或磁盘存储空间中。当一个程序（或任务）引用内存中的逻辑地址时，处理器会把该逻辑地址转换成一个线性地址，然后使用分页机制把该线性地址转换成对应的物理地址。如果包含线性地址的页面当前不在物理内存中，处理器就会产生一个页错误异常。页错误异常的处理程序通常就会让操作系统从磁盘中把相应页面加载到物理内存中（操作过程中可能还会把物理内存中不同的页面写到磁盘上）。当页面加载到物理内存中之后，从异常处理过程的返回操作会使得导致异常的指令被重新执行。

处理器用于把线性地址转换成物理地址时所需的信息及处理器产生页错误异常（若必要的话）所需的信息都存储于页目录和页表中。分页与分段最大的不同之处在于分页使用了固定长度的页面。段的长度通常与存放在其中的代码或数据结构具有的长度相同。如果仅使用分段地址转换，那么存储在物理内存中的一个数据结构将包含其所有的部分。如果使用了分页，那么一个数据结构的一部分存储于物理内存中，而另一部分则保存在磁盘中。

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

1．页表结构

分页转换功能由驻留在内存中的表来描述，该表称为页表（page table），存放在物理地址空间中。页表可看作简单的 220 个物理地址数组。线性到物理地址的映射功能可以简单地看作进行数组查找。线性地址的高 20 位构成这个数组的索引值，用于选择对应页面的物理（基）地址。线性地址的低 12 位给出了页面中的偏移量，加上页面的基地址最终形成对应的物理地址。由于页面基地址对齐在 4KB 边界上，因此页面基地址的低 12 位肯定是 0。这意味着高 20 位的页面基地址和 12 位偏移量连接组合在一起就能得到对应的物理地址。

页表中每个页表项的大小为 32 位。由于只需要其中的 20 位来存放页面的物理基地址，因此剩下的 12 位可用于存放诸如页面是否存在等的属性信息。如果线性地址索引的页表项被标注为

存在的，则表示该项有效，我们可以从中取得页面的物理地址。如果页表项中信息表明（说明、指明）页不存在，那么当访问对应物理页面时就会产生一个异常。

2. 两级页表结构

页表含有 220（1MB）个表项，而每项占用 4B。如果作为一个表来存放，它们最多占用 4MB 的内存。因此为了减少内存占用量，80x86 使用了两级表。由此，高 20 位线性地址到物理地址的转换也被分成两步来进行，每步使用（转换）其中的 10bit。

第一级表称为页目录（page directory）。它被存放在 1 页 4KB 页面中，具有 210（1KB）个 4B 长度的表项。这些表项指向对应的二级表。线性地址的最高 10 位（位 31～22）用作一级表（页目录）中的索引值来选择 210 个二级表之一。

第二级表称为页表（page table），它的长度也是 1 个页面，最多含有 1KB 个 4B 的表项。每个 4B 表项含有相关页面的 20 位物理基地址。二级页表使用线性地址中间 10 位（位 21～12）作为表项索引值，以获取含有页面 20 位物理基地址的表项。该 20 位页面物理基地址和线性地址中的低 12 位（页内偏移）组合在一起就得到了分页转换过程的输出值，即对应的最终物理地址。

图 1-18 给出了二级表的查找过程。其中 CR3 寄存器指定页目录表的基地址。线性地址的高 10 位用于索引这个页目录表，以获得指向相关第二级页表的指针。线性地址中间 10 位用于索引二级页表，以获得物理地址的高 20 位。线性地址的低 12 位直接作为物理地址低 12 位，从而组成一个完整的 32 位物理地址。

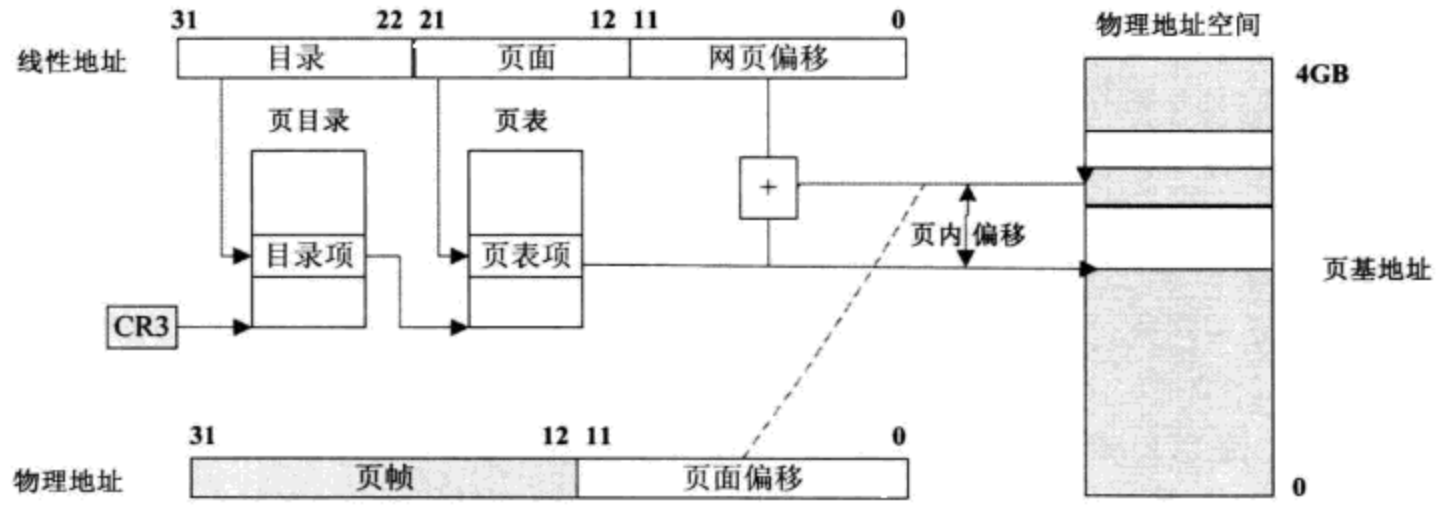


图 1-18　线性地址和物理地址之间的变换

3. 不存在的页表

使用二级表结构，并没有解决需要使用 4MB 内存来存放页表的问题。实际上，我们把问题搞得有些复杂了。因为我们需要另增一个页面来存放目录表。然而，二级表结构允许页表被分散在内存各个页面中，而不需要保存在连续的 4MB 内存块中。

另外，并不需要为不存在的或线性地址空间未使用部分分配二级页表。虽然目录表页面必须总是存在于物理内存中，但是二级页表可以在需要时再分配。这使得页表结构的大小对应于实际使用的线性地址空间大小。页目录表中每个表项也有一个存在（present）属性，类似于页表中的表项。页目录表项中的存在属性指明对应的二级页表是否存在。

如果目录表项指明对应的二级页表存在，那么通过访问二级表，表查找过程第 2 步将同如上描述继续下去。如果存在位表明对应的二级表不存在，那么处理器就会产生一个异常来通知操作系统。页目录表项中的存在属性使得操作系统可以根据实际使用的线性地址范围来分配二级页表页面。目录表项中的存在位还可以用于在虚拟内存中存放二级页表。

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

4．页表项格式

页目录和页表的表项格式如图 1-19 所示。其中位 31～12 含有物理地址的高 20 位，用于定位物理地址空间中一个页面（也称为页帧）的物理基地址。表项的低 12 位含有页属性信息。前文已经讨论过存在属性，这里简要说明其余属性的功能和用途。

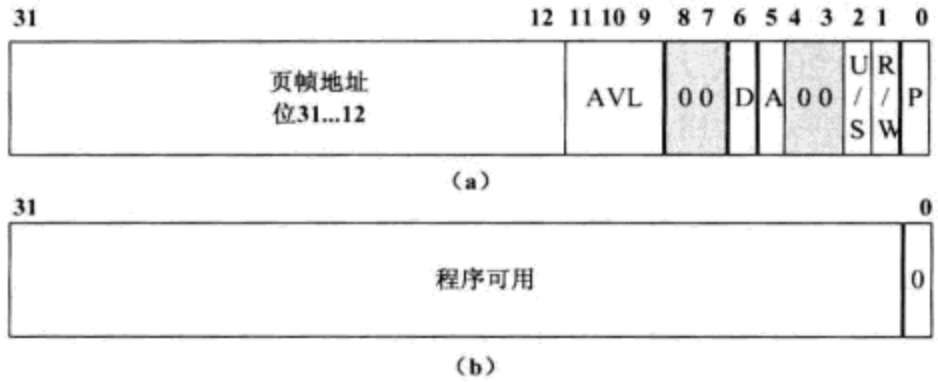


图 1-19　页目录和页表的表项格式

P——位 0 是存在（Present）标志，用于指明表项对地址转换是否有效。P=1 表示有效；P=0 表示无效。在页转换过程中，如果说涉及的页目录或页表的表项无效，则会导致一个异常。如果 P=0，那么除表示表项无效外，其余位可供程序自由使用，如图 1-19（b）所示。例如，操作系统可以使用这些位来保存已存储在磁盘上的页面的序号。

R/W——位 1 是读/写（Read/Write）标志。如果等于 1，表示页面可以被读、写或执行。如果为 0，表示页面只读或可执行。当处理器运行在超级用户特权级（级别 0、1 或 2）时，则 R/W 位不起作用。页目录项中的 R/W 位对其所映射的所有页面起作用。

U/S——位 2 是用户/超级用户（User/Supervisor）标志。如果为 1，那么运行在任何特权级上的程序都可以访问该页面。如果为 0，那么页面只能被运行在超级用户特权级（0、1 或 2）上的程序访问。页目录项中的 U/S 位对其所映射的所有页面起作用。

A——位 5 是已访问（Accessed）标志。当处理器访问页表项映射的页面时，页表表项的这个标志就会被置为 1。当处理器访问页目录表项映射的任何页面时，页目录表项的这个标志就会被置为 1。处理器只负责设置该标志，操作系统可通过定期地复位该标志来统计页面的使用情况。

D——位 6 是页面已被修改（Dirty）标志。当处理器对一个页面执行写操作时，就会设置对应页表表项的 D 标志。处理器并不会修改页目录项中的 D 标志。

AVL——该字段保留，专供程序使用。处理器不会修改这几位，以后的升级处理器也不会修改。

1.4　Linux 内核体系结构

本节首先基于 Linux 0.95 版的内核源代码，简要介绍了 Linux 内核的体系结构及模式，从 Linux 源代码的组织结构方面着手介绍了每个文件夹中包含的各个.c 文件的主要功能。在本节最后，以 Linux 主目录下的第一个 Makefile 为例，对 Makefile 的相关知识进行说明，并对每一行代码进行详细的说明。

1.4.1 Linux 内核体系结构及模式

一个完整的可用的系统主要包括 4 部分：硬件、操作系统内核、操作系统服务以及用户应用程序。其中，向用户提供服务的是操作系统服务程序，如 shell 命令解释系统等；用于提供对硬件资源进行抽象和访问调度的是操作系统内核。Linux 操作系统内核主要由进程调度模块、进程间通信模块、内存管理模块、虚拟文件系统模块和网络接口模块构成。各模块的主要功能如下：

1. 进程调度模块

进程调度模块是内核的重要组成部分，它负责选择下一个要运行的进程和控制进程对 CPU 的使用，上下文切换等，并记录系统中所有进程的执行情况。其中，调度程序的作用非常重要，它决定了进程能否公平合理地使用 CPU，并保证内核能否尽快地执行必要的硬件操作。

2. 进程间通信模块

进程间通信模块主要负责在不同进程之间信息的传播或交换工作。鉴于进程的用户空间是互相独立的，一般而言不能互相访问，唯一的例外是共享内存区。也可以利用系统空间作为一个“公共场所”（内核显然可以提供这样的条件）来进行。除此以外，就是双方都可以访问的外设。Linux 提供了多种进程之间的通信机制。

3. 内存管理模块

内存管理模块负责系统的内存管理，它负责保证机器的主内存区能够被所有的进程安全地共享。同时，内存管理模块还支持虚拟内存，使得 linux 能够支持进程使用比实际内存空间更大的内存地址空间。

4. 虚拟文件系统模块

虚拟文件系统（VFS）是物理文件系统与服务之间的一个接口层，它对 Linux 的每个文件系统的所有细节进行抽象，使得不同的文件系统在 Linux 核心以及系统中运行的其他进程看来，都是相同的。严格说来，VFS 并不是一种实际的文件系统，它在系统启动时建立，在系统关闭时消亡，只存在于内存中，不存在于任何外存空间。虚拟文件系统使 Linux 同时安装、支持许多不同类型的文件系统成为可能。

5. 网络接口模块

网络接口模块提供对多种网络通信标准的访问并支持许多网络硬件。网络接口分为网络协议和网络驱动程序两种。各模块之间的依赖关系如图 1-20 所示。

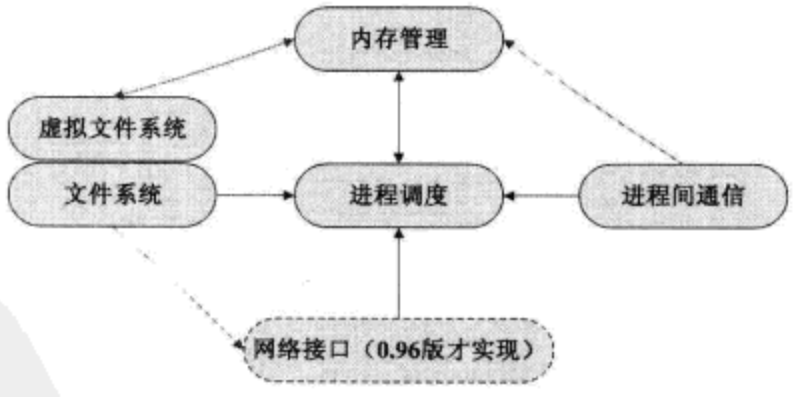


图 1-20　Linux 内核各模块之间的关系

操作系统内核的模式主要有整体式的单内核模式和层次式的微内核模式。单内核结构利用高阶抽象界面对硬件进行抽象，应用一组系统调用来实现操作系统的功能，例如进程管理、文件系统和存储管理等，这些功能由多个运行在核心态的模块来完成。

单内核模式的主要优点是内核集成度高，代码紧凑，执行速度快，不足之处在于所有的模块都在同一个内核空间上运行，系统容易崩溃，且层次性不强。微内核提供一组“最基本”的服务，如进程调度、进程间通信、存储管理、处理 I/O 设备。而其他服务，如文件管理、网络支持等通过接口连到微内核。微内核是内核的精简版本，它设计成在很小的内存空间内增加移植性，提供

模块化设计，以使用户安装不同的接口。所有服务在用户模式下运行，处理这些服务与处理其他任何一个程序一样，因为每个服务只是在自己的地址空间运行，所以这些服务彼此之间都受到了保护。微内核具有很好的扩展性。

Linux 内核的组织方式是单内核方式，为了弥补单内核方式扩展性不强的特点，Linux 允许动态装载和卸载模块。Linux 模块是一段可以在机器启动后的任意时间被动态连接的代码。在不需要时，可以从内核中卸载。这样的方式，扩展了内核的功能而不需要重启或重新编译内核。Linux 中，每一个模块包含两个组成部分，一个用于在初始化该模块时调用，另一个是用于删除该模块时调用。

1.4.2 Linux 内核源代码目录结构

Linux 内核是一种单内核模式的系统，因此，内核中所有的程序几乎都紧密地联系在一起。由于它们之间的依赖和调用关系非常密切，所以在阅读一个源代码文件时往往需要参阅其他相关的文件。因此在开始阅读内核源代码之前，先熟悉一下源代码文件的目录结构和安排是非常有必要的。将 Linux-0.95 解压后，内核源代码被放到了 Linux 目录下，解压之后的 Linux 内核源代码结构及各部分简单功能介绍如图 1-21 所示。

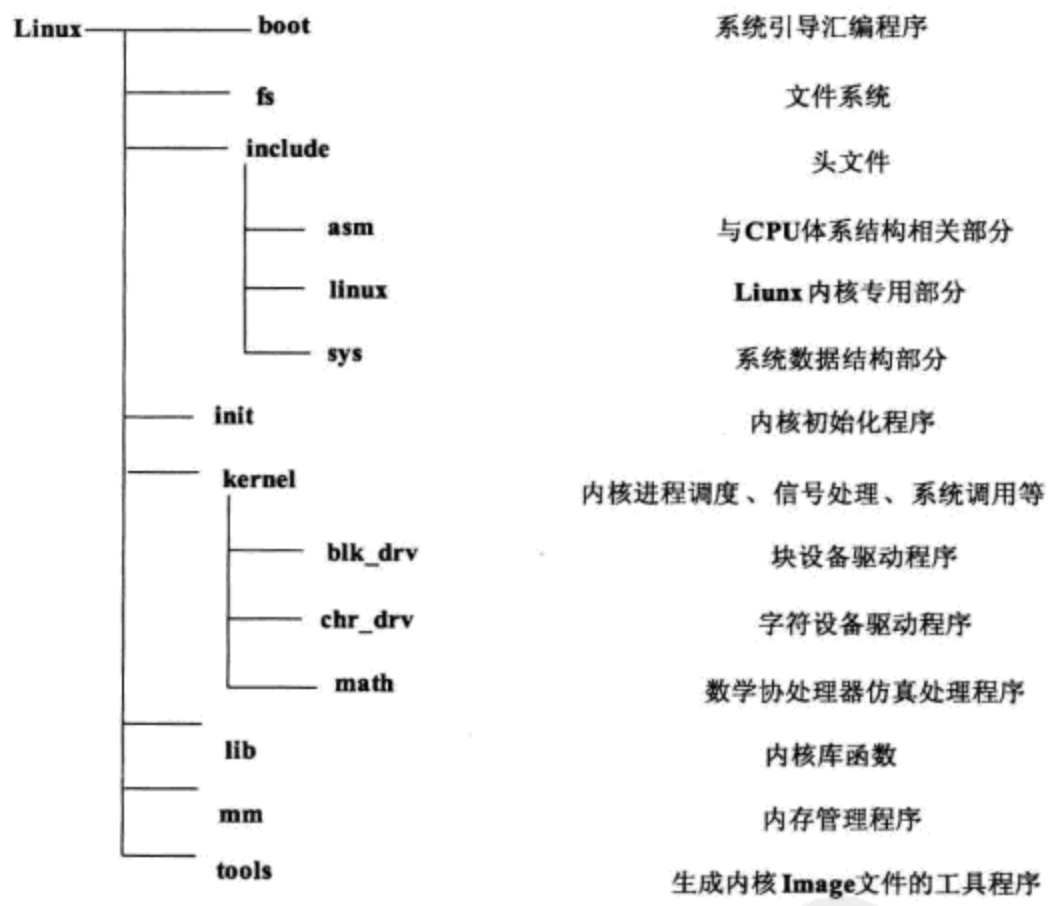


图 1-21 Linux 内核源代码目录结构

1. 内存管理目录 mm

该目录中存放的是内存管理及虚拟内存管理的代码。共有两个.c 文件。主要用于管理程序对主内存的使用等。

memory.c 是内存管理的主要程序，完成了对内存的动态分配、回收和页面异常的处理。

swap.c 文件主要包括用于对交换设备进行操作的函数 rw_swap_page()和 get_swap_page()、用来释放交换设备某页的函数 swap_free()，以及换入换出函数 swap_in()、try_to_swap_out() 和 swap_out()，get_free_page()用做内存页的申请，sys_swapon()用来在向系统中增加交换文件或交换设备。

2. 内核程序目录 kernel

该目录包含 13 个代码文件，一个 Makefile 文件，以及 3 个子目录，即 blk_drv、chr_drv 和 math。这 13 个代码文件主要存放的是 Linux 中进程管理和调度这样的核心子系统，包含如 fork.c、sched.c 等文件。此 13 个代码文件功能如下：

asm.s 汇编文件，主要是实现了包括大部分的硬件故障（或出错）处理的底层次代码。

exit.c 程序主要描述了进程终止和退出的处理事宜。

fork.c 程序主要用于创建子进程。

mktime.c 程序用于计算从 1970 年 1 月 1 日 0 时起到开机当日经过的时间(秒)，作为开机时间。

panic.c 程序用于当内核程序出错时显示错误信息并使系统进入死循环。

printk.c 程序是内核中使用的显示函数，功能与 C 标准函数库中的 print()相同。

ptrace.c 程序主要负责进程的追踪，方便了程序的开发和调试。

sched.c 程序是内核中有关任务调度函数的程序。

signal.c 程序包括设置和获取进程信号阻塞码系统调用函数 sys_sgetmask()和 sys_ssetmask()、信号处理系统调用 sys_signal()、修改进程在收到特定信号时所采取的行动的系统调用 sys_sigaction()，以及在系统调用中断处理程序中处理信号的函数 do_signal()。

sys.c 程序主要完成系统调用功能函数的实现。

sys_call.s 程序主要实现系统调用中断 int 0x80 的入口处理过程以及信号检测处理。

traps.c 程序实现 asm.s 的中断处理过程中调用的 c 函数，它主要定义和实现了 asm.s 中所引用的各个硬件异常中断处理程序，目前主要用于调试目的，以后将扩展用来杀死遭损坏的进程。

vsprintf.c 用于对参数产生格式化输出。

blk_drv 子目录中存放的是与块设备相关的驱动程序。通常情况下，用户通过文件系统来访问设备，而设备驱动程序即为文件系统实现了调用接口。此目录包括 4 个.c 文件，1 个.h 文件以及一个 Makefile 文件。功能如下：

blk.h 程序用于存放块设备的相关参数。其中主要定义了请求等待队列中项的数据结构 request，用宏语句定义了电梯搜索算法，并对内核目前支持的虚拟盘、硬盘和软盘三种块设备，根据它们各自的主设备号分别对应了常数值。

floppy.c 程序是软盘控制器驱动程序。使用请求项和请求项链表结构来处理所有对软盘的读写操作。

hd.c 程序是硬盘控制器驱动程序。主要用于处理对硬盘控制器块设备的读写和硬盘的初始化工作。

ll_rw_blk.c 是程序所有块设备与系统其他部分的接口程序。它主要用于处理低层块设备的读写操作。

ramdisk.c 程序是内存虚拟盘驱动程序。由于在实际操作中不涉及外部控制器的相关操作，因此实现方式简单。

chr_drv 子目录中存放的是与字符设备相关的驱动程序。实现了对串口、键盘和控制台终端设备的驱动等。共包括 5 个.c 文件、2 个汇编语言程序文件以及一个 Makefile 文件。功能如下：

console.c 程序主要用于进行终端初始化操作和屏幕写函数。

keyboard.s 程序是键盘驱动的汇编程序，它主要包括键盘中断的处理程序。并支持美语、德语、法语、丹麦语等类型的键盘。

pty.c 程序用于实现伪终端的功能，实现其读写功能。

rs_io.s 程序模块用于实现 rs32 串行通信输入/输出的中断处理过程。

serial.c 程序实现系统串行端口初始化，为使用串行终端设备作好准备工作。

tty_io.c 程序包括字符设备的上层接口函数。主要含有终端读/写函数 tty_read()和 tty_write()。读操作的行规则函数 copy_to_cooked()也在这里实现。

tty_ioctl.c 程序用于字符设备的控制操作，实现了函数（系统调用）tty_ioctl()。程序通过使用该函数可以修改指定终端 termios 结构中的设置标志等信息。

math 子目录中主要包括与协处理器操作相关的文件。

3. 文件系统目录 fs

fs 目录是文件系统实现程序的目录，包含有 17 个 c 语言程序以及一个 Minix 目录。0.95 版的 Linux 内核将文件系统的程序划分为两部分：一部分实现了对虚拟文件系统的支持；另一部分用于实现与 Minix 文件系统相关的操作，该部分被单独放在一个名为 Minix 的文件夹内。文件系统目录是物理文件系统与服务之间的一个接口层。它包括四个部分：一是高速缓冲区的管理程序，该程序主要实现了高速缓冲区缓存的功能；二是文件系统底层程序，主要功能包括对 i 结点的分配与释放，文件路径名与 i 结点的映射，文件系统超级块的处理以及文件系统的选择处理功能；三是文件数据的访问操作程序，主要实现了对字符设备、管道、块设备，正规文件的数据访问功能；四是文件系统的上层操作程序，主要实现了文件的打开、关闭、创建等有关文件操作的系统调用。

buffer.c 主要实现了缓冲区高速缓存的功能。程序对块设备的读出和写入基本都需要通过高速缓冲区来进行操作。高速缓冲区在物理内存上占据的内存空间在内核模块和主内存区之间，主要是由大小和磁盘逻辑块的大小相同缓冲块和指向缓冲块的缓冲头组成。

inode.c 程序是虚拟文件系统（VFS）的 inode 程序，主要实现了对 VFS 文件系统的文件结点的处理功能。此时的内核加入了对虚拟文件系统的支持，将 MINIX 文件系统从原文件系统中分离出来，分别形成了充当服务接口的虚拟文件系统 VFS 和物理文件系统 MINIX。所以文件 i 结点也分为内存中的 VFS 文件系统 inode 和 MINIX 文件系统中的 inode。

namei.c 程序实现了与目录名或文件名相关的操作。

super.c 程序实现了对文件系统中超级块的操作，其中的函数是供上层文件名和目录操作使用的文件系统底层函数。超级块表记录着内核所使用的各种文件系统的超级块信息。

file_table.c 程序中只定义了文件表数组并无任何函数。

block_dev.c 程序实现了对块设备文件的操作。

pipe.c 程序主要实现了读管道数据和写管道数据的功能，数据从管道头部写入，由管道尾部读取。

char_dev.c 程序实现了对字符设备文件的访问功能，主要包括对串口终端设备的读写、对控制台终端的读写、对内存设备文件的读写。

read_write.c 程序实现了对块设备、字符设备、管道及正规文件访问的上层接口，该接口通过识别文件类型分别调用相应的访问函数。

open.c 程序主要实现了对文件操作的系统调用。主要包括文件的创建、打开及关闭，文件的访问权限、操作时间和宿主及属性的修改等。

exec.c 程序主要实现了加载与执行可执行文件和 shell 脚本。

stat.c 程序主要实现了对文件状态的操作，主要为读取文件状态信息。

fcntl.c 程序是对目录和文件进行操作的上层处理程序，实现了文件控制系统调用和两个文件句柄复制的系统调用。

ioctl.c 文件实现了调用输入/输出控制系统。

select.c 程序实现了 Linux 的选择处理功能，其中主要实现了系统调用 select。该系统调用是用来让我们的程序监视多个文件句柄的状态变化。

MINIX 的 bitmap.c 程序主要实现了对 Minix 文件系统 i 结点位图和逻辑位图的管理功能，其中包括对 i 结点位图的释放及占用和对逻辑块位图的释放及占用。

MINIX 的 inode.c 程序主要实现了 Minix 文件系统的文件 i 结点读取和写入操作。

MINIX 的 truncate.c 程序实现了将 Minix 文件系统的文件长度截断为 0 的功能，实现方式是将指定文件结点在设备上占用的所有逻辑块释放，包括直接块、一次间接块和二次间接块。

MINIX 的 namei.c 程序是 MINIX 文件系统的 namei.c 程序，主要实现了在 MINIX 文件系统中一些关于目录、文件的建立和删除等操作函数及系统调用。

MINIX 的 file_dev.c 程序实现了对 Minix 文件系统正规文件的访问功能，主要包括对正规文件数据的读出和写入。

MINIX 的 minix_op.c 程序主要包含了 Minix 文件系统的结点操作结构体和文件操作结构体。

4. 引导程序 boot 目录

内核源代码中最先被编译的程序是 boot 目录中含有的三个汇编文件。它们完成的主要功能是当计算机加电时引导内核启动，将内核代码加载到内存中，并做一些进入 32 位保护运行方式前的系统初始化工作。

bootsect.s 代码是磁盘引导块程序，驻留在磁盘的第一个扇区中。该程序的主要作用是先将 setup 模块从磁盘加载到内存，紧接着 bootsect 的后面位置（0x90200），然后利用 BIOS 中断 0x13 取磁盘参数表中当前启动引导盘的参数，接着在屏幕上显示“Loading system...”字符串。再将 system 模块从磁盘上加载到内存 0x10000 开始的地方，随后确定根文件系统的设备号，最后跳转到 setup 程序的开始处（0x90200）执行 setup 程序。

setup.s 程序利用 ROM BIOS 中断读取机器系统数据，并将这些数据保存到 0x90000 开始的位置，内核中相关程序将会在运行中使用这些参数。然后 setup 程序将 system 模块从 0x10000-0x8ffff 整块向下移动到内存绝对地址 0x00000 处。再加载中断描述符表寄存器（idtr）和全局描述符表寄存器（gdtr），开启 A20 地址线，重新设置两个中断控制芯片 8259A。最后设置 CPU 的控制寄存器 CR0，进入 32 位保护模式运行，并跳转到 head.s 程序继续运行。

head.s 程序处于内存绝对地址 0 开始的地方，主要功能有：重新设置 idt，重新设置 gdt，检查 A20 地址线是否打开，检查是否含有数学协处理器，设置内存的分页处理机制，页目录表放在绝对地址 0 处，然后是 4 个页表项。最后通过 Ret 操作进入 main.c 程序入口进行执行。

5. 内核初始化目录 init

该目录中只有一个 main.c 文件，存放的是 Linux 内核的初始化代码。系统在执行完 boot/目录中的 head.s 便会将执行权交给 main.c，移到用户模式创建新进程。

该程序包括了内核初始化的所有工作。main.c 程序首先利用前面 setup.s 程序取得的系统参数设置系统的根文件设备号以及一些内存全局变量。然后，所有方面的硬件开始初始化工作。待所有初始化工作完成后就设置中断允许标志以开启中断，main()也切换到了任务 0 中运行。

6. 内核库函数目录 lib

通用的 Linux 内核函数库目录共包括 12 个.c 文件以及一个 Makefile 文件。为初始化程序执行在用户态的进程提供支持。各个.c 文件的功能如下：

_exit.c()是退出函数；close.c()是关闭文件函数；ctype.c()是判断字符所属类型函数；dup.c()是复制文件描述符函数；errno.c()定义了一个出错号变量 errno，用于在函数调用失败时存放出错号；execve.c()是执行程序函数；malloc.c()是内存分配函数；open.c()是打开文件函数；setsid.c()是创建会话调用；string.c()是字符串操作函数；wait.c()等待子进程状态函数；write.c()是写文件函数。

7. 头文件目录 include

头文件目录存放的是内核头文件，包含了所有的.h 头文件，共有 39 个。其中，include/主目录下有 13 个头文件，其他头文件则分别存放在 asm（4 个），Linux（13 个），sys（9 个）3 个子目录中。

include/目录下的头文件主要是供内核和用户程序使用；asm/子目录下主要用于存放与计算机硬件体系结构相关的头文件；linux/子目录用于存放 Linux 内核专用的头文件；sys/子目录用于存放一些与文件状态、进程、系统数据类型等有关的头文件。

8. 编译内核工具目录 tools

该目录下只有一个 build.c 文件，用于将 Linux 各个目录中被分别编译生成的目标代码连接合并成一个可运行的内核映像文件 image。

1.4.3 Linux 内核的编译

在 Linux 内核源代码中，除了 tools/、init/和 boot/目录外，其余每个子目录均包含一个相应的 Makefile 文件，这些文件结构完全相同。顶层 linux/目录下的 Makefile 负责指示 make 程序最终使用 tools/目录中的 build 执行程序将所有内核编译代码连接和合并成一个可运行的内核映像文件 image。

1. Makefile 简介

make 是一个项目管理工具，它会按照规则（定义在规则文件中）、依赖顺序对项目中的文件进行编译、连接，或把它们安装到系统中。它的最基本用途就是管理项目的编译、连接。一个工程中按类型、功能、模块的不同，将源文件分别放在若干个目录中，makefile 定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作。

make 工具最基本的功能就是通过 makefile 文件来描述源程序之间的相互关系并自动维护编译工作。而 makefile 文件需要按照某种语法进行编写，文件中需要说明如何编译各个源文件并连接生成可执行文件，并要求定义源文件之间的依赖关系。makefile 文件是许多编译器维护编译信息的常用方法，在一些集成开发环境中，用户通过友好的界面即可修改 makefile 文件。

makefile 带来的好处就是“自动化编译”，写好 makefile 文件之后，只需要一个 make 命令，整个工程会自动编译，极大地提高了软件开发的效率。Make 用于解释 makefile 中的指令。Makefile 已经成为一种在工程方面的编译方法。在 UNIX/Linux 系统中，习惯使用 Makefile 作为 makefile 文件。如果要使用其他文件作为 makefile，则可利用类似下面的 make 命令选项指定 makefile 文件：

```
$ make -f Makefile.debug
```

在一个简单的 Makefile 文件中含有一些规则，这些规则具有如下的形式：

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

target 是一个目标文件，可以是 Object File，也可以是执行文件或标签；

prerequisites 是要生成那个 target 所需要的文件（.c .h .cpp）或目标（.o）；

command 是 make 需要执行的命令。

target 中的目标文件依赖于 prerequisites 中的文件，其生成规则定义在 command 中。prerequisites 中如果有一个以上的文件比 target 文件要新，command 所定义的命令就会被执行。

2．Linux 主目录下的 Makefile 文件分析

Linux 主目录下的这个 Makefile 文件还嵌套调用了所有子目录中包含的 Makefile 文件。这样，当 Linux 目录（包括子目录）下的任何文件被修改，make 都会对其进行重新编译。因此为了编译整个内核所有的源代码文件，只要在 Linux 主目录下运行一次 make 工具即可。

```
1.  AS86=as86 -0 –a                                  # 8086 汇编编译器和连接器
2.  LD86=ld86 -0
3.  AS   =as                                         # GNU 汇编编译器和连接器
4.  LD   =ld
5.  LDFLAGS =-s -x –M                                # GNU 连接器 gld 运行时用到的标识选项
6.  CC   =gcc $(RAMDISK)                             # GNU C 程序编译器
7.  CFLAGS   =-Wall -O -fstrength-reduce -fomit-frame-pointer   # gcc 的选项
8.  CPP  =cpp -nostdinc –Iinclude                    # gcc 的预处理程序
    #   ROOT_DEV 指定在创建内核映像文件时所使用的默认根文件系统所在的设备，
    #   这可以是软盘(FLOPPY)、/dev/xxxx 或者干脆空着，空着时，build 程序
    #   （在 tools/目录中）就使用默认值/dev/hd6
9.  ROOT_DEV=/dev/hdb1
10. ARCHIVES=kernel/kernel.o mm/mm.o fs/fs.o
    #   kernel、mm 和 fs 所产生的目标代码文件
11. FILESYSTEMS =fs/minix/minix.o
    #   minix 虚拟文件系统所产生的目标代码文件
12. DRIVERS=kernel/blk_drv/blk_drv.a kernel/chr_drv/chr_drv.a
    #   块和字符设备库文件
13. MATH=kernel/math/math.a                          # 数学运算库文件
14. LIBS=lib/lib.a                                   # 由 lib/目录中的文件编译生成的通用库文件
15. c.s:
    #   make 老式的隐式后缀规则。指示 make 利用下面的命令将所有的.c 文件编译
    #   生成.s 汇编程序。':'表示下面是该规则的命令
16. $(CC) $(CFLAGS) \
17. -nostdinc -Iinclude -S -o $*.s $<
18. .s.o:                                            # 将所有.s 汇编程序文件编译成.o 目标文件
19. $(AS) -c -o $*.o $<
20. . c.o:                                           # 将所有*.c 文件.编译成.o 目标文件
21  $(CC) $(CFLAGS) \
22. -nostdinc -Iinclude -c -o $*.o $<
23. all:   Image                          # all 表示创建 Makefile 所知的最顶层目标，即是 image 文件
24  Image: boot/bootsect boot/setup tools/system tools/build
    #   说明目标（Image 文件）是由分号后面的 4 个元素产生
```

```
25.  tools/build boot/bootsect boot/setup tools/system $(ROOT_DEV) > Image
26.  sync                                  # 执行的命令
27.  disk: Image                           # 表明 disk 由 Image 产生
28.  dd bs=8192 if=Image of=/dev/PS0
29.  tools/build: tools/build.c            # 由 tools 目录下的 build.c 程序生成执行程序 build
30.  $(CC) $(CFLAGS) \
31.  -o tools/build tools/build.c
32.  boot/head.o: boot/head.s               # 利用 head.s 生成 head.o 目标文件
33.  tools/system: boot/head.o init/main.o \
34.              $(ARCHIVES) $(FILESYSTEMS) $(DRIVERS) $(MATH) $(LIBS)
     #  '\'符号表示下一行是续行。此两行表示 tools 目录中的 system 文件由分号右边所列
     #  的元素生成
35.  $(LD) $(LDFLAGS) boot/head.o init/main.o \
36.  $(ARCHIVES) \
37.  $(FILESYSTEMS) \
38.  $(DRIVERS) \
39.  $(MATH) \
40.  $(LIBS) \
41.  -o tools/system > System.map
42.  kernel/math/math.a:                    # 数学协处理函数文件 math.a 由下一行的命令实。
43.    (cd kernel/math; make)
44.  kernel/blk_drv/blk_drv.a:              # 块设备函数文件 blk_drv.a 由下一行的命令实现
45.    (cd kernel/blk_drv; make)
46.  kernel/chr_drv/chr_drv.a:              # 字符设备函数文件 chr_drv.a 由下一行的命令实现
47.    (cd kernel/chr_drv; make)
48.  kernel/kernel.o:                       # 内核目标模块 kernel.o 由下一行的命令实现
49.    (cd kernel; make)
50.  mm/mm.o:                               # 内存管理模块 mm.o 由下一行的命令实现
51.    (cd mm; make)
52.  fs/fs.o:                               # 文件系统目标模块 fs.o 由下一行的命令实现
53.    (cd fs; make)
54.  fs/minix/minix.o:                       # 虚拟文件系统目标模块由下一行的命令实现
55.    (cd fs/minix; make)
56.  lib/lib.a:                             # 库函数 lib.a 由下一行的命令实现
57.    (cd lib; make)
58.  boot/setup: boot/setup.s
59.  $(AS86) -o boot/setup.o boot/setup.s
60.  $(LD86) -s -o boot/setup boot/setup.o
     #  使用 8086 汇编和连接器对 setup.s 文件进行编译生成 setup 文件
61.  boot/setup.s:      boot/setup.s include/linux/config.h
62.  $(CPP) -traditional boot/setup.s -o boot/setup.s
     #  利用 setup.s 和文件 config.h 生成 setup.s 文件
63.  boot/bootsect.s:  boot/bootsect.s include/linux/config.h
64.  $(CPP) -traditional boot/bootsect.s -o boot/bootsect.s
     #  利用 bootsect.s 和文件 config.h 生成 bootsect.s 文件
65.  boot/bootsect:    boot/bootsect.s
66.  $(AS86) -o boot/bootsect.o boot/bootsect.s
67.  $(LD86) -s -o boot/bootsect boot/bootsect.o
     #  使用 8086 汇编和连接器对 bootsect.s 文件进行编译生成 bootsect.o 磁盘引导块
68.  clean:
```

```
69.  rm -f Image System.map tmp_make core boot/bootsect boot/setup \
70.                          boot/bootsect.s boot/setup.s init/main.s
71.  rm -f init/*.o tools/system tools/build boot/*.o
72.   (cd mm;make clean)
73.   (cd fs;make clean)
74.   (cd kernel;make clean)
75.   (cd lib;make clean)
     #   利用 clean 命令清除所有编译连接生成的文件
76.  backup: clean
77.   (cd .. ; tar cf - linux | compress - > backup.Z)
78.  sync
     #   该规则将首先执行上面的 clean 规则，然后对 linux/目录进行压缩，生成 backup.Z
     #   压缩文件。并且迫使缓冲块数据立即写盘并更新磁盘超级块
79.  dep:
     #   用于各文件之间的依赖关系。创建的这些依赖关系是为了给 make 用来确定是否
     #   需要重建一个目标对象
80.  sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
81.   (for i in init/*.c;do echo -n "init/";$(CPP) -M $$i;done) >> tmp_make
82.  cp tmp_make Makefile
83.   (cd fs; make dep)
84.   (cd kernel; make dep)
85.   (cd mm; make dep)
     ### Dependencies:
86.  init/main.o : init/main.c include/unistd.h include/sys/stat.h \
87.  include/sys/types.h include/sys/time.h include/time.h include/sys/times.h \
88.  include/sys/utsname.h include/sys/param.h include/sys/resource.h \
89.  include/utime.h include/linux/tty.h include/termios.h include/linux/sched.h \
90.  include/linux/head.h include/linux/fs.h include/linux/mm.h \
91.  include/linux/kernel.h include/signal.h include/asm/system.h \
92.  include/asm/io.h include/stddef.h include/stdarg.h include/fcntl.h \
93.  include/string.h
```

第 2 章　引导启动与初始化程序

2.1　引导启动程序（boot）

本节主要描述 boot/目录中的三个汇编代码文件，如表 2-1 所示。这三个文件虽然都是汇编程序，但却使用了两种语法格式。bootsect.s 和 setup.s 采用近似于 Intel 的汇编语言语法，需要使用 Intel 8086 汇编编译器和连接器 as86 和 ld86，而 head.s 则使用 GNU 的汇编程序格式，并且运行在保护模式下，需要用 GNU 的 as 进行编译。这是一种 AT&T 语法的汇编语言程序。

使用两种编译器的主要原因是由于对 Intel x86 处理器系列来讲，GNU 的编译器仅支持 i386 及以后来的 CPU，不支持生成运行在实模式下的程序。

表 2-1　linux/boot/目录

文　件　名	长度（B）	最后修改时间（GMT）
bootsect.s	5052	1991-12-05 22:47:58
head.s	5938	1991-11-18 15:05:09
setup.s	5364	1991-12-05 22:48:10

阅读这些代码除了需要了解一些一般 8086 汇编语言的知识以外，还要对采用 Intel 80x86 微处理器的 PC 机的体系结构以及 80386、32 位保护模式下的编程原理有些了解。所以在开始阅读源代码之前，读者可以先大概浏览一些有关 PC 机硬件接口控制编程和 80386、32 位保护模式的编程方法。

2.1.1　总体功能

这里先总地说明一下 Linux 操作系统启动部分的主要执行流程。当 PC 的电源打开后，80x86 结构的 CPU 将自动进入实模式，并从地址 0xFFFF0 开始自动执行程序代码，这个地址通常是 ROM-BIOS 中的地址。PC 机的 BIOS 将执行某些系统的检测，并在物理地址 0 处开始初始化中断向量。此后，它将可启动设备的第一个扇区（磁盘引导扇区，512 字节）读入内存绝对地址 0x7C00 处，并跳转到这个地方。启动设备通常是软驱或是硬盘。这里的叙述是非常简单的，但这已经足够理解内核初始化的工作过程了。

Linux 的最前面部分是用 8086 汇编语言编写的（boot/bootsect.s），它将由 BIOS 读入到内存绝对地址 0x7C00（31KB）处，当它被执行时就会把自己移到绝对地址 0x90000（576KB）处，并把启动设备中后 2KB 字节代码（boot/setup.s）读入到内存 0x90200 处，而内核的其他部分（system 模块）则被读入到从地址 0x10000 开始处，因为当时 system 模块的长度不会超过 0x80000 字节大小（即 512KB），所以它不会覆盖在 0x90000 处开始的 bootsect 和 setup 模块。后面 setup 程序将会把 system 模块移动到内存起始处，这样 system 模块中代码的地址也即等于实际的物理地址，便于对内核代码和数据的操作。

启动部分识别主机的某些特性以及 vga 卡的类型。如果需要，它会要求用户为控制台选择显示模式。然后将整个系统从地址 0x10000 移至 0x0000 处，进入保护模式并跳转至系统的余下部分（在 0x0000 处）。此时所有 32 位运行方式的设置启动被完成：IDT、GDT 以及 LDT 被加载，处理器和协处理器也已确认，分页工作也设置好了；最终调用 init/main.c 中的 main()程序。

上述操作的源代码是在 boot/head.S 中的，这可能是整个内核中最有诀窍的代码了。注意如果在前述任何一步中出了错，计算机就会死锁。在操作系统还没有完全运转之前是处理不了出错的。

为什么不把系统模块直接加载到物理地址 0x0000 开始处而要在 setup 程序中再进行移动呢？这是因为在 setup 程序代码开始部分还需要利用 ROM BIOS 中的中断调用来获取机器的一些参数（例如显卡模式、硬盘参数表等）。当 BIOS 初始化时会在物理内存开始处放置一个大小为 0x400 字节（1KB）的中断向量表，因此需要在使用完 BIOS 的中断调用后才能将这个区域覆盖掉。

2.1.2 bootsect.s 文件

该文件描述了 bootsect.s 代码，该代码是磁盘引导块程序，驻留在磁盘的第一个扇区中（引导扇区，0 磁道（柱面），0 磁头，第 1 个扇区）。

1. 功能描述

bootsect.s 代码是磁盘引导块程序，驻留在磁盘的第一个扇区中。在 PC 机加电 ROM BIOS 自检后，引导扇区由 BIOS 加载到内存 0x7C00 处，然后将自己移动到内存 0x90000 处。该程序的主要作用是首先将 setup 模块载入到内存，紧接着到 bootsect 程序中 0x90200 位置，然后利用 BIOS 中断 0x13 获取当前启动引导盘的参数，接着在屏幕上显示“Loading system...”字符串。再者将 system 模块从磁盘上加载到内存 0x10000 开始的地方。随后确定根文件系统的设备号，若没有指定，则根据所保存的引导盘的每磁道扇区数判别出盘的类型和种类（是否是软盘）并保存其设备号于 root_dev（引导块的 0x508 地址处），最后长跳转到 setup 程序的开始处（0x90200）执行 setup 程序。

2. 代码注释

bootsect.s 文件的详细代码注释如下：

```
1.      .globl begtext, begdata, begbss, endtext, enddata, endbss ! 定义了 6 个全局标识符
2.          .text ! 文本段
3.          begtext:
4.      .data ! 数据段
5.      begdata:
6.          .bss ! 未初始化数据段(Block Started by Symbol)
7.          begbss:
8.          .text ! 文本段
9.          SETUPLEN = 4 setup ! 程序的扇区数(setup-sectors)值;
10.     BOOTSEG = 0x07c0 bootsect ! 的原始地址（是段地址，以下同）;
11.     INITSEG = 0x9000 将 boot ! 移到这里 -- 避开;
12.         SETUPSEG = 0x9020 setup! 程序从这里开始;
13.     SYSSEG = 0x1000 ! system ! 模块加载到 0x10000（64 kB）处。
14.     ENDSEG = SYSSEG + SYSSIZE ! 停止加载的段地址;
15.         ! 根文件系统设备使用与引导时同样的软驱设备;
16.     ! 根文件系统设备在第一个硬盘的第一个分区上，等等：ROOT_DEV = 0x306 ! 指定根文件
    系统设备是第 2 个硬盘的第 1 个分区。这是 Linux 老式的硬盘命名 entry start ! 告知连接程序，
    程序从 start 标号开始执行。
17.     start: ! 17-26 行作用是将自身（bootsect）从目前段位置 0x07c0（31k）
18.         ! 移动到 0x9000(576k)处，共 256 字（512 字节），然后跳转到
19.         ! 移动后代码的 go 标号处，也即本程序的下一语句处。
20.     mov ax,#BOOTSEG ! 将 ds 段寄存器置为 0x7C0;
```

```
21.     mov ds,ax
22.         mov ax,#INITSEG ！ 将 es 段寄存器置为 0x9000；
23.         mov es,ax
24.         mov cx,#256 ！ 移动计数值=256 字；
25.         sub si,si ！ 源地址 ds:si = 0x07C0:0x0000
26.     sub di,di ！ 目的地址 es:di = 0x9000:0x0000
27.     rep ！ 重复执行，直到 cx = 0
28.         movw ！ 移动 1 个字；
29.         jmpi go,INITSEG ！ 间接跳转。这里 INITSEG 指出跳转到的段地址。
30.         ！ 从下面开始，CPU 执行已移动到 0x9000 段处的代码。
31.     go: mov ax,cs ！ 将 ds、es 和 ss 都置成移动后代码所在的段处(0x9000)。
32.     mov ds,ax ！ 由于程序中有堆栈操作(push,pop,call)，因此必须设置堆栈。
33.         mov es,ax
34.         ! put stack at 0x9ff00.！ 将堆栈指针 sp 指向 0x9ff00(即 0x9000:0xff00)处
35.     mov ss,ax
36.     mov sp,#0xFF00 ! arbitrary value >>512
37.         ! load the setup-sectors directly after the bootblock.
38.     ! Note that 'es' is already set up.! 在 bootsect 程序块后紧根着加载 setup 模块的代码数据。
39.     ！ 注意 es 已经设置好了。(在移动代码时 es 已经指向目的段地址处 0x9000)
40.         load_setup:
41.         mov dx,#0x0000 ! drive 0, head 0
42.         mov cx,#0x0002 ! sector 2, track 0
43.     mov bx,#0x0200 ! address = 512, in INITSEG
44.     mov ax,#0x0200+SETUPLEN ! service 2, nr of sectors
45.         int 0x13 ! read it
46.         jnc ok_load_setup ! ok – continue
47.     mov dx,#0x0000
48.     mov ax,#0x0000 ! reset the diskette
49.         int 0x13
50.         load_setup
51.         ok_load_setup:
52.         ! Get disk drive parameters, specifically nr of sectors/track
53.     mov dl,#0x00
54.     mov ax,#0x0800 ! AH=8 is get drive parameters
55.         int 0x13
56.         mov ch,#0x00
57.         seg cs ！ 表示下一条语句的操作数在 cs 段寄存器所指的段中。
58.     mov sectors,cx ！ 保存每磁道扇区数。
59.     mov ax,#INITSEG
60.         mov es,ax ！ 因为上面取磁盘参数中断改掉了 es 的值，这里重新改回。
61.         ! Print some inane message ！ 显示一些信息('Loading system ...'回车换行，共 24 个字符)。
62.     mov ah,#0x03 ! read cursor pos
63.     xor bh,bh ！ 读光标位置。
64.         int 0x10
65.         mov cx,#24 ！ 共 24 个字符。
66.         mov bx,#0x0007 ! page 0, attribute 7 (normal)
67.     mov bp,#msg1 ！ 指向要显示的字符串。
68.     mov ax,#0x1301 ! write string, move cursor
69.         int 0x10 ！ 写字符串并移动光标。
70.         ! ok, we've written the message, now
```

```
71.     ! we want to load the system (at 0x10000)！ 现在开始将 system 模块加载到 0x10000(64k)处。
72.     mov ax,#SYSSEG
73.         mov es,ax ! segment of 0x010000 ! es = 存放 system 的段地址。
74.         call read_it ! 读磁盘上 system 模块，es 为输入参数。
75.     call kill_motor ! 关闭驱动器马达，这样就可以知道驱动器的状态了。
76.     ! After that we check which root-device to use. If the device is
77.         ! defined (!= 0), nothing is done and the given device is used.
78.     ! Otherwise, either /dev/PS0 (2,28) or /dev/at0 (2,8), depending
79.     ! on the number of sectors that the BIOS reports currently.
80.         seg cs
81.         mov ax,root_dev ! 将根设备号保存到 ax; 寄存器中
82.     cmp ax,#0
83.     jne root_defined
84.         seg cs
85.         mov bx,sectors ! 取上面第 58 行保存的每磁道扇区数。如果 sectors=15
86.         ! 则说明是 1.2Mb 的驱动器；如果 sectors=18，则说明是
87.     ! 1.44Mb 软驱。因为是可引导的驱动器，所以肯定是 A 驱。
88.     mov ax,#0x0208 ! /dev/ps0 - 1.2Mb
89.         cmp bx,#15 ! 判断每磁道扇区数是否=15
90.         je root_defined ! 如果等于，则 ax 中就是引导驱动器的设备号。
91.     mov ax,#0x021c ! /dev/PS0 - 1.44Mb
92.     cmp bx,#18
93.         je root_defined
94.         undef_root: ! 如果都不一样，则死循环（死机）。
95.         jmp undef_root
96.         root_defined:
97.     seg cs
98.     mov root_dev,ax ! 将检查过的设备号保存起来。
99.         ! after that (everyting loaded), we jump to
100.        ! the setup-routine loaded directly after
101.    ! the bootblock:! 到此，所有程序都加载完毕，就跳转到被
102.    ! 加载在 bootsect 后面的 setup 程序去。
103.        jmpi 0,SETUPSEG ! 跳转到 0x9020:0000(setup.s 程序的开始处)。
104.        !!!! 本程序到此就结束了。!!!!
105.    ! 下面是两个子程序.
106.    head: .word 0 ! current head !当前磁头号。
107.        track: .word 0 ! current track !当前磁道号。
108.        read_it:! 测试输入的段值。从盘上读入的数据必须存放在位于内存地址 64KB 的边界开
    始处，否则进入死循环。
109.    ! 清 bx 寄存器，用于表示当前段内存放数据的开始位置。
110.    mov ax,es
111.        test ax,#0x0fff
112.        die: jne die ! es must be at 64kB boundary ! es 值必须位于 64KB 地址边界!
113.    xor bx,bx ! bx is starting address within segment ! bx 为段内偏移位置。
114.    rp_read:
115.        ! 判断是否已经读入全部数据。比较当前所读段是否就是系统数据末端所处的段
    （#ENDSEG），如果不是就
116.        ! 跳转至下面 ok1_read 标号处继续读数据。否则退出子程序返回。
117.    mov ax,es
118.    cmp ax,#ENDSEG ! have we loaded all yet?! 是否已经加载了全部数据？
```

```
119.            jb ok1_read
120.            ret
121.            ok1_read:
122.            seg cs
123.        mov ax,sectors ! 取每磁道扇区数。
124.        sub ax,sread ! 减去当前磁道已读扇区数。
125.            mov cx,ax ! cx = ax = 当前磁道未读扇区数。
126.            shl cx,#9 ! cx = cx * 512 字节。
127.        add cx,bx ! cx = cx + 段内当前偏移值(bx)! = 此次读操作后，段内共读入的字节数。
128.        jnc ok2_read ! 若没有超过 64KB 字节，则跳转至 ok2_read 处执行。
129.            je ok2_read
130.            xor ax,ax ! 若加上此次将读磁道上所有未读扇区时会超过 64KB，则计算
131.        sub ax,bx ! 此时最多能读入的字节数(64KB - 段内读偏移位置)，再转换
132.        shr ax,#9 ! 成需要读取的扇区数。
133.            ok2_read:
134.            call read_track
135.            mov cx,ax ! cx = 该次操作已读取的扇区数。
136.            add ax,sread ! 当前磁道上已经读取的扇区数。
137.        seg cs
138.        cmp ax,sectors ! 如果当前磁道上的还有扇区未读，则跳转到 ok3_read 处。
139.            jne ok3_read! 读该磁道的下一磁头面(1 号磁头)上的数据。如果已经完成，则去读下一
    磁道。
140.        mov ax,#1
141.        sub ax,head ! 判断当前磁头号。
142.            jne ok4_read ! 如果是 0 磁头，则再去读 1 磁头面上的扇区数据。
143.            inc track ! 否则去读下一磁道。
144.            ok4_read:
145.        mov head,ax ! 保存当前磁头号。
146.        xor ax,ax ! 清当前磁道已读扇区数。
147.            ok3_read:
148.            mov sread,ax ! 保存当前磁道已读扇区数。
149.            shl cx,#9 ! 上次已读扇区数*512 字节。
150.        add bx,cx ! 调整当前段内数据开始位置。
151.        jnc rp_read ! 若小于 64KB 边界值，则跳转到 rp_read 处，继续读数据。
152.            ! 否则调整当前段，为读下一段数据作准备。
153.            mov ax,es
154.            add ax,#0x1000 ! 将段基址调整为指向下一个 64KB 内存开始处。
155.        mov es,ax
156.        xor bx,bx ! 清段内数据开始偏移值。
157.            jmp rp_read ! 跳转至 rp_read 处，继续读数据。
158.            ! 读当前磁道上指定开始扇区和需读扇区数的数据到 es:bx 开始处。
159.        ! int 0x13，ah=2 的说明。
160.        ! al - 需读扇区数；es:bx - 缓冲区开始位置。
161.            read_track:
162.            push ax
163.            push bx
164.            push cx
165.            push dx
166.            mov dx,track ! 取当前磁道号。
167.        mov cx,sread ! 取当前磁道上已读扇区数。
```

```
168.    inc cx ! cl = 开始读扇区。
169.        mov ch,dl ! ch = 当前磁道号。
170.        mov dx,head ! 取当前磁头号。
171.        mov dh,dl ! dh = 磁头号。
172.    mov dl,#0 ! dl = 驱动器号(为 0 表示当前 A 驱动器)。
173.    and dx,#0x0100 ! 磁头号不大于 1。
174.        mov ah,#2 ! ah = 2，读磁盘扇区功能号。
175.        int 0x13
176.        jc bad_rt ! 若出错，则跳转至 bad_rt。
177.    pop dx
178.    pop cx
179.        pop bx
180.        pop ax
181.        ret! 执行驱动器复位操作（磁盘中断功能号 0），再跳转到 read_track 处重试。
182.        bad_rt: mov ax,#0
183.        mov dx,#0
184.    int 0x13
185.    pop dx
186.        pop cx
187.        pop bx
188.        pop ax
189.        jmp read_track
190.        ! 这个子程序用于关闭软驱的马达，这样我们进入内核后它处于已知状态，以后也就无
    须担心它了。
191.    kill_motor:
192.    push dx
193.        mov dx,#0x3f2 ! 软驱控制卡的驱动端口，只写。
194.        mov al,#0 ! A 驱动器，关闭 FDC，禁止 DMA 和中断请求，关闭马达。
195.        outb ! 将 al 中的内容输出到 dx 指定的端口去。
196.    pop dx
197.    ret
198.        sectors:
199.        .word 0 ! 存放当前启动软盘每磁道的扇区数。
200.        msg1:
201.        .byte 13,10 ! 回车、换行的 ASCII 码。
202.        .ascii "Loading system ..."
203.    .byte 13,10,13,10 ! 共 24 个 ASCII 码字符。
204.    .org 508 ! 表示下面语句从地址 508(0x1FC)开始，所以 root_dev
205.        ! 在启动扇区的第 508 开始的 2 个字节中。
206.        root_dev:
207.        .word ROOT_DEV ! 这里存放根文件系统所在的设备号（init/main.c 中会用）。
208.    boot_flag:
209.    .word 0xAA55 ! 硬盘有效标识
210.        .text
211.        endtext:
212.        .data
213.        enddata:
214.        .bss
215.        endbss:
```

源代码第 161-162 行计算和验证当前磁道需要读取的扇区数，放在 ax 寄存器中。根据当前磁道还未读取的扇区数以及段内数据字节开始偏移位置，计算如果全部读取这些未读扇区，所读总字节数是否会超过 64KB 段长度的限制。若会超过，则根据此次最多能读入的字节数(64KB - 段内! 偏移位置)，反算出此次需要读取的扇区数。

2.1.3 setup.s 文件

setup.s 文件的主要作用是利用 BIOS 中断读取机器系统数据。这样在 Linux 内核的引导阶段就能够获取其所运行在的机器的硬件配置等基本情况。

1. 功能描述

setup.s 文件的作用主要是利用 ROM BIOS 中断读取机器系统数据，并将这些数据保存到 0x90000 开始的位置（覆盖掉了 bootsect 程序所在的地方），所取得的参数和保留的内存位置见表 2-2 所示。这些参数将被内核中相关程序使用，例如字符设备驱动程序集中的 ttyio.c 程序等。

表 2-2 setup 程序读取并保留的参数

内存地址	长度	名称	描述
0x90000	2	光标位置	列号（0x00-最左端），行号（0x00-最顶端）
0x90002	2	扩展内存数	系统从 1M 开始的扩展内存数值
0x90004	2	显示页面	当前显示页面
0x90006	1	显示模式	设置当前的显示模式
0x90007	1	字符列数	设置字符的列数
0x90008	2	保留位	用于扩展的保留位
0x9000A	1	显示内存	设置内存（0x00-64k,0x01-128k,0x02-192k,0x03=256k）
0x9000B	1	显示状态	设置显示状态（0x00-彩色，0x11-单色）
0x9000C	2	特性参数	显示卡特性参数
0x90080	16	硬盘参数表	第 1 个硬盘的参数表
0x90090	16	硬盘参数表	第 2 个硬盘的参数表
0x901FC	2	根设备号	根文件系统所在的设备号（bootsec.s 中设置）

然后 setup 程序将 system 模块从 0x10000-0x8ffff（当时认为内核系统模块 system 的长度不会超过此值：512KB）整块向下移动到内存绝对地址 0x00000 处。接下来加载中断描述符表寄存器(idtr)和全局描述符表寄存器(gdtr)，开启 A20 地址线，重新设置两个中断控制芯片 8259A，将硬件中断号重新设置为 0x20 - 0x2f。最后设置 CPU 的控制寄存器 CR0（也称机器状态字），从而进入 32 位保护模式运行，并跳转到位于 system 模块最前面部分的 head.s 程序继续运行。为了能让 head.s 在 32 位保护模式下运行，在本程序中临时设置了中断描述符表（idt）和全局描述符表（gdt），并在 gdt 中设置了当前内核代码段的描述符和数据段的描述符。在下面的 head.s 程序中会根据内核的需要重新设置这些描述符表。

现在，我们根据 CPU 在实模式和保护模式下寻址方式的不同，用比较的方法来简单说明 32 位保护模式运行机制的主要特点，以便能顺利地理解本节的程序。在后续章节中将逐步对其进行详细说明。

在实模式下，寻址一个内存地址主要是使用段和偏移值，段值被存放在段寄存器中（例如 ds），并且段的最大长度被固定为 64KB。段内偏移地址存放在任意一个可用于寻址的寄存器中（例如 si）。因此，根据段寄存器和偏移寄存器中的值，就可以算出实际指向的内存地址。

而在保护模式运行方式下，段寄存器中存放的不再是寻址段的基地址，而是一个段描述符表中某项的索引值。索引值指定的段描述符项中含有需要寻址的内存段的基地址、段的最大长度值和段的访问级别等信息。寻址的内存位置是由该段描述符项中指定的段基地址值和偏移值组合而成，段的最大长度也由描述符指定。可见，和实模式下的寻址相比，段寄存器值换成了段描述符索引，但偏移值还是原实模式下的概念。这样，在保护模式下寻址一个内存地址就需要比实模式下多一道手续，也即需要使用段描述符表。这是由于在保护模式下访问一个内存段需要的信息比较多，一个 16 位的段寄存器放不下这么多内容。

因此，在进入保护模式之前，必须首先设置好将要用到的段描述符表，例如全局描述符表 gdt。然后使用指令 lgdt 把描述符表的基地址告知 CPU（gdt 表的基地址存入 gdtr 寄存器）。再将机器状态字的保护模式标志置位即可进入 32 位保护运行模式。

2．代码注释

setup.s 文件的详细代码注释如下：

```
1.          .globl begtext, begdata, begbss, endtext, enddata, endbss
2.              .text
3.              begtext:
4.              . data
5.              begdata:
6.          .bss
7.          begbss:
8.              .text
9.              entry start
10.             start:
11.             ! ok, the read went well so we get current cursor position and save it for
12.         ! posterity.
13.         ! ok，整个读磁盘过程都正常，现在将光标位置保存以备今后使用。
14.             mov ax,#INITSEG ! this is done in bootsect already, but...
15.         mov ds,ax
16.         mov ah,#0x03 ! read cursor pos! BIOS 中断 0x10 的读光标功能号 ah = 0x03
17.             ! 输入：bh = 页号
18.             ! 返回：ch = 扫描开始线，cl = 扫描结束线，
19.         ! dh = 行号(0x00 是顶端)，dl = 列号(0x00 是左边)。
20.         xor bh,bh
21.             int 0x10 ! save it in known place, con_init fetches
22.             mov [0],dx ! it from 0x90000.! 上两句是说将光标位置信息存放在 0x90000 处，控制台
23.         ! 初始化时会来取。
24.         ! Get memory size (extended mem, kB)！ 下面 3 句取扩展内存的大小值（KB）。
25.             ! 是调用中断 0x15，功能号 ah = 0x88
26.             ! 返回：ax = 从 0x100000（1M）处开始的扩展内存大小(KB)。
27.         ! 若出错则 CF 置位，ax = 出错码。
28.         mov ah,#0x88
29.             int 0x15
30.             mov [2],ax ! 将扩展内存数值存在 0x90002 处（1 个字节）。
31.             ! Get video-card data: ! 下面这段用于取显示卡当前显示模式。
32.         ! 调用 BIOS 中断 0x10，功能号 ah = 0x0f
33.         ! 返回：ah = 字符列数，al = 显示模式，bh = 当前显示页。
34.             ! 0x90004(1 字)存放当前页，0x90006 显示模式，0x90007 字符列数。
35.             mov ah,#0x0f
```

```
36.         int 0x10
37.         mov [4],bx ! bh = display page
38.           mov [6],ax ! al = video mode, ah = window width
39.           ! check for EGA/VGA and some config parameters ! 检查显示方式（EGA/VGA）并取参数。
40.         mov ah,#0x12
41.         mov bl,#0x10
42.           Int 0x10
43.           mov [8],ax ! 0x90008 = ??
44.           mov [10],bx ! 0x9000A = 安装的显示内存，0x9000B = 显示状态(彩色/单色)
45.         mov [12],cx ! 0x9000C = 显示卡特性参数。
46.         ! Get hd0 data ! 取第一个硬盘的信息（复制硬盘参数表）。
47.           mov ax,#0x0000
48.           mov ds,ax
49.           lds si,[4*0x41] ! 取中断向量 0x41 的值，也即 hd0 参数表的地址 ds:si
50.         mov ax,#INITSEG
51.         mov es,ax
52.           mov di,#0x0080 ! 传输的目的地址: 0x9000:0x0080  es:di
53.           mov cx,#0x10 ! 共传输 0x10 字节。
54.           rep
55.         movsb
56.         ! Get hd1 data
57.           mov ax,#0x0000
58.           mov ds,ax
59.           lds si,[4*0x46] ! 取中断向量 0x46 的值，也即 hd1 参数表的地址 ds:si
60.         mov ax,#INITSEG
61.         mov es,ax
62.           mov di,#0x0090 ! 传输的目的地址: 0x9000:0x0090  es:di
63.           mov cx,#0x10
64.           rep
65.           movsb
66.           ! Check that there IS a hd1 :-) ! 检查系统是否存在第 2 个硬盘，如果不存在则第 2 个表清零。
67.         mov ax,#0x01500
68.         mov dl,#0x81
69.           int 0x13
70.           jc no_disk1
71.           cmp ah,#3 ! 判断是否是硬盘（类型是否等于 3）
72.           je is_disk1
73.           no_disk1:
74.         mov ax,#INITSEG ! 第 2 个硬盘不存在，则对第 2 个硬盘表清零。
75.         mov es,ax
76.           mov di,#0x0090
77.           mov cx,#0x10
78.           mov ax,#0x00
79.           rep
80.           stosb
81.           is_disk1:
82.           ! now we want to move to protected mode ... ! 现在我们要进入保护模式中了...
83.         cli ! no interrupts allowed ! ! 此时不允许中断。
84.         ! first we move the system to it's rightful place
85.           mov ax,#0x0000
```

```
86.         cld ! 'direction'=0, movs moves forward
87.     do_move:
88.     mov es,ax ! destination segment ! es:di  目的地址(初始为 0x0000:0x0)
89.         add ax,#0x1000
90.         cmp ax,#0x9000 ！ 判断是否已经把从 0x8000 段开始的 64k 代码移动
91.     jz end_move
92.     mov ds,ax ! source segment ! ds:si  源地址(初始为 0x1000:0x0)
93.         sub di,di
94.         sub si,si
95.         mov cx,#0x8000 ！ 移动 0x8000 字（64k 字节）。
96.     rep
97.     movsw
98.         jmp do_move
99.         ! then we load the segment descriptors
100.        end_move:
101.        mov ax,#SETUPSEG ! right, forgot this at first. didn't work :-)
102.    mov ds,ax ! ds 指向本程序(setup)段。
103.    lidt idt_48 ! load idt with 0,0
104.        ! that was painless, now we enable A20
105.        call empty_8042 ！ 等待输入缓冲器空。
106.    call empty_8042 ！ 等待输入缓冲器空，看命令是否被接受。
107.    mov al,#0xDF ! A20 on ！ 选通 A20 地址线的参数。
108.        out #0x60,al
109.        call empty_8042 ！ 输入缓冲器为空，则表示 A20 线已经选通。
110.    mov al,#0x11 ! initialization sequence
111.    ! 0x11 表示初始化命令开始，是 ICW1 命令字，表示边
112.        !沿触发、多片 8259 级连、最后要发送 ICW4 命令字。
113.        out #0x20,al ! send it to 8259A-1 ！ 发送到 8259A 主芯片。
114.    .word 0x00eb,0x00eb ! jmp $+2, jmp $+2 ! '$'表示当前指令的地址，
115.    out #0xA0,al ! and to 8259A-2 ！ 再发送到 8259A 从芯片。
116.        .word 0x00eb,0x00eb
117.        mov al,#0x20 ! start of hardware int's (0x20)
118.    out #0x21,al ！ 送主芯片 ICW2 命令字，起始中断号，要送奇地址。
119.    .word 0x00eb,0x00eb
120.        mov al,#0x28 ! start of hardware int's 2 (0x28)
121.        out #0xA1,al ！ 送从芯片 ICW2 命令字，从芯片的起始中断号。
122.    .word 0x00eb,0x00eb
123.    mov al,#0x04 ! 8259-1 is master
124.        out #0x21,al ！ 送主芯片 ICW3 命令字，主芯片的 IR2 连从芯片 INT。
125.        .word 0x00eb,0x00eb ！ 参见代码列表后的说明。
126.    mov al,#0x02 ! 8259-2 is slave
127.    out #0xA1,al ！ 送从芯片 ICW3 命令字，表示从芯片的 INT 连到主芯
128.        ！ 片的 IR2 引脚上。
129.        .word 0x00eb,0x00eb
130.        mov al,#0x01 ! 8086 mode for both
131.    out #0x21,al ！ 送主芯片 ICW4 命令字。8086 模式；普通 EOI 方式，
132.    ！ 需发送指令来复位。初始化结束，芯片就绪。
133.        .word 0x00eb,0x00eb
134.        out #0xA1,al ！ 送从芯片 ICW4 命令字，内容同上。
135.        .word 0x00eb,0x00eb
```

```
136.    mov al,#0xFF ! mask off all interrupts for now
137.    out #0x21,al ! 屏蔽主芯片所有中断请求。
138.        .word 0x00eb,0x00eb
139.        out #0xA1,al ! 屏蔽从芯片所有中断请求。
140.        mov ax,#0x0001 ! protected mode (PE) bit ! 保护模式比特位(PE)。
141.    lmsw ax ! This is it! ! 就这样加载机器状态字!
142.    jmpi 0,8 ! jmp offset 0 of segment 8 (cs) ! 跳转至 cs 段 8，偏移 0 处。
143.        ! 下面这个子程序检查键盘命令队列是否为空。这里不使用超时方法 - 如果这里死机，
144.    ! 则说明 PC 机有问题，我们就没有办法再处理下去了。
145.    ! 只有当输入缓冲器为空时（状态寄存器位 2 = 0）才可以对其进行写命令。
146.        empty_8042:
147.        .word 0x00eb,0x00eb ! 这是两个跳转指令的机器码(跳转到下一句)，相当于延时空操作。
148.    in al,#0x64 ! 8042 status port ! 读 AT 键盘控制器状态寄存器。
149.    test al,#2 ! is input buffer full? ! 测试位 2，判断输入缓冲器已满
150.        jnz empty_8042 ! yes – loop
151.        ret
152.        gdt: ! 全局描述符表开始处。描述符表由多个 8 字节长的描述符项组成。
153.    .word 0,0,0,0 ! dummy ! 第 1 个描述符，不用。
154.    ! 这里在 gdt 表中的偏移量为 0x08，当加载代码段寄存器(段选择符)时，使用的是这个偏移值。
155.        .word 0x07FF ! 8Mb - limit=2047 (2048*4096=8Mb)
156.    .word 0x0000 ! base address=0
157.    .word 0x9A00 ! code read/exec
158.        .word 0x00C0 ! granularity=4096, 386
159.        ! 这里在 gdt 表中的偏移量是 0x10，当加载数据段寄存器(如 ds 等)时，使用的是这个偏
        移值。
160.    .word 0x07FF ! 8Mb - limit=2047 (2048*4096=8Mb)
161.    .word 0x0000 ! base address=0
162.        .word 0x9200 ! data read/write
163.        .word 0x00C0 ! granularity=4096, 386
164.    idt_ 48:
165.    .word 0 ! idt limit=0
166.        .word 0,0 ! idt base=0L
167.        gdt_48:
168.        .word 0x800 ! gdt limit=2048, 256 GDT entries
169.    ! 全局表长度为 2k 字节，因为每 8 字节组成一个段描述符项
170.    ! 所以表中共可有 256 项。
171.        .word 512+gdt,0x9 ! gdt base = 0X9xxxx
172.        ! 4 个字节构成的内存线性地址：0x0009<<16 + 0x0200+gdt
173.    ! 也即 0x90200 + gdt（即在本程序段中的偏移地址）。
174.    .text
175.        endtext:
176.        .data
177.        enddata:
178.        .bss
179.        endbss:
```

源代码第 21～22 行将 ds 置成#INITSEG(0x9000)。这已经在 bootsect 程序中设置过，但是现在是 setup 程序，Linus 觉得需要再重新设置一下。

源代码第 36～37 行调用 BIOS 中断 0x10，附加功能选择 -取方式信息功能号：ah = 0x12，bl = 0x10 返回：bh = 显示状态 (0x00 - 彩色模式，I/O 端口=0x3dX) (0x01 - 单色模式，I/O 端

口=0x3bX) bl = 安装的显示内存(0x00 - 64k, 0x01 - 128k, 0x02 - 192k, 0x03 = 256k cx = 显示卡特性参数(参见程序后的说明)。

2.1.4 head.s 文件

head.s 文件用于重新建立和设置 IDT、GDT 表，重新建立这些表的主要原因是为了让程序在结构上更加清晰。

1. 功能描述

head.s 文件在被编译后，会被连接成 system 模块的最前面开始部分，这也就是为什么称其为头部(head)程序的原因。从这里开始，内核完全都是在保护模式下运行了。heads.s 汇编程序与前面的语法格式不同，它采用的是 AT&T 的汇编语言格式，并且需要使用 GNU 的 gas 和 gld2 进行编译连接。因此请注意代码中赋值的方向是从左到右。

这段程序实际上处于内存绝对地址 0 处开始的地方。这个程序的功能比较单一。首先是加载各个数据段寄存器，重新设置中断描述符表 idt，共 256 项，并使各个表项均指向一个只报错误的哑中断程序。

然后重新设置全局描述符表 gdt。接着使用物理地址 0 与 1M 开始处的内容相比较的方法，检测 A20 地址线是否已真的开启（如果没有开启，则在访问高于 1Mb 物理内存地址时 CPU 实际只会访问（IP MOD1Mb）地址处的内容），如果检测下来发现没有开启，则进入死循环。

然后程序测试 PC 机是否含有数学协处理器芯片（80287、80387 或其兼容芯片），并在控制寄存器 CR0 中设置相应的标志位。接着设置管理内存的分页处理机制，将页目录表放在绝对物理地址 0 开始处（也是本程序所处的物理内存位置，因此这段程序将被覆盖掉），紧随后面放置共可寻址 16MB 内存的 4 个页表，并分别设置它们的表项。最后利用返回指令将预先放置在堆栈中的/init/main.c 程序的入口地址弹出，去运行 main()程序。

2. 代码注释

head.s 文件的详细代码注释如下：

```
1.    .text
2.    .globl _idt,_gdt,_pg_dir,_tmp_floppy_area
3.    _pg_dir: # 页目录将会存放在这里
4.    startup_32: # 17-20 行设置各个数据段寄存器
5.    movl $0x10,%eax # 对于 GNU 汇编来说，每个直接数要以'$'开始，否则是表示地址
6.    # 每个寄存器名都要以'%'开头，eax 表示是 32 位的 ax 寄存器
7.    # 再次注意!!! 这里已经处于 32 位运行模式，因此这里的$0x10 并不是把地址 0x10 装入各个
8.    # 段寄存器，它现在其实是全局段描述符表中的偏移值，或者更正确地说是一个描述符表项
9.    # 的选择符。有关选择符的说明请参见 setup.s 中 152 行下的说明。这里$0x10 的含义是请求
10.   # 特权级 0(位 0-1=0)、选择全局描述符表(位 2=0)、选择表中第 2 项(位 3-15=2)。它正好指
11.   # 向表中的数据段描述符项。（描述符的具体数值参见前面 setup.s 中 160，1633 行）
12.   # 下面代码的含义是：置 ds,es,fs,gs 中的选择符为 setup.s 中构造的数据段（全局段描述符表
13.   # 的第 2 项）=0x10 并将堆栈放置在 stack_start 指向的 user_stack 数组区，然后使用本程序
14.   # 后面定义的新中断描述符表和全局段描述表。新全局段描述表中初始内容与 setup.s 中的基本
15.   # 一样，仅段限长从 8MB 修改成了 16MB。stack_start 定义在 kernel/sched.c，69 行。它是指向
16.   # user_stack 数组末端的一个长指针
17.   mov %ax,%ds
18.   mov %ax,%es
19.   mov %ax,%fs
20.   mov %ax,%gs
```

```
21.  lss _stack_start,%esp # 表示_stack_start  ss:esp，设置系统堆栈
22.  # stack_start 定义在（kerrel/sched.c，69 行）
23.  call setup_idt # 调用设置中断描述符表子程序
24.  call setup_gdt # 调用设置全局描述符表子程序
25.  movl $0x10,%eax # reload all the segment registers
26.  mov %ax,%ds # after changing gdt. CS was already
27.  mov %ax,%es # reloaded in 'setup_gdt'
28.  mov %ax,%fs # 因为修改了 gdt，所以需要重新装载所有的段寄存器
29.  mov %ax,%gs # CS 代码段寄存器已经在 setup_gdt 中重新加载过了
30.  lss _stack_start,%esp
31.  xorl %eax,%eax
32.  incl %eax # check that A20 really IS enabled
33.  movl %eax,0x000000 # loop forever if it isn't
34.  cmpl %eax,0x100000
35.  je 1b # '1b'表示向后(backward)跳转到标号 1 去
36.  movl %cr0,%eax # check math chip
37.  andl $0x80000011,%eax # Save PG,PE,ET
38.  /* "orl $0x10020,%eax" here for 486 might be good */
39.  orl $2,%eax # set MP
40.  movl %eax,%cr0
41.  call check_x87
42.  jmp after_page_tables # 跳转到 88 行
43.  check_x87:
44.  fninit
45.  fstsw %ax
46.  cmpb $0,%al
47.  je 1f /* no coprocessor: have to set bits */
48.  movl %cr0,%eax # 如果存在的则向前跳转到标号 1 处，否则改写 cr0
49.  xorl $6,%eax /* reset MP, set EM */
50.  movl %eax,%cr0
51.  ret
52.  align 2 # 这里".align 2"的含义是指存储边界对齐调整。"2"表示调整到地址最后 2 位为零，
53.  # 即按 4 字节方式对齐内存地址
54.  byte 0xDB,0xE4 /* fsetpm for 287, ignored by 387 */ # 287 协处理器码
55.  ret
56.  setup_idt:
57.  lea ignore_int,%edx # 将 ignore_int 的有效地址（偏移值）值 edx 寄存器
58.  movl $0x00080000,%eax # 将选择符 0x0008 置入 eax 的高 16 位中
59.  movw %dx,%ax /* selector = 0x0008 = cs */
60.  偏移值的低 16 位置入 eax 的低 16 位中。此时 eax 含有
61.  #门描述符低 4 字节的值
62.  movw $0x8E00,%dx /* interrupt gate - dpl=0, present
63.  # 此时 edx 含有门描述符高 4 字节的值
64.  lea _idt,%edi # _idt 是中断描述符表的地址
65.  mov $256,%ecx
66.  rp_sidt:
67.  movl %eax,(%edi) # 将哑中断门描述符存入表中
68.  movl %edx,4(%edi)
69.  addl $8,%edi # edi 指向表中下一项
70.  dec %ecx
```

```
71.  jne rp_sidt
72.  lidt idt_descr # 加载中断描述符表寄存器值
73.  ret
74.  setup_gdt:
75.  lgdt gdt_descr # 加载全局描述符表寄存器
76.  ret
77.  org 0x1000 # 从偏移 0x1000 处开始是第 1 个页表（偏移 0 开始处将存放页表目录）
78.  pg0:
79.. org 0x2000
80.  pg1:
81.  org 0x3000
82.  pg2:
83.  org 0x4000
84.  pg3:
85.  org 0x5000 # 定义下面的内存数据块从偏移 0x5000 处开始
86.  _tmp_floppy_area:
87.  fill 1024,1,0 # 共保留 1024 项，每项 1 字节，填充数值 0
88.  after_page_tables:
89.  pushl $0 # These are the parameters to main :-)
90.  pushl $0 # 这些是调用 main 程序的参数（指 init/main.c）
91.  pushl $0 # 其中的'$'符号表示这是一个立即操作数
92.  pushl $L6 # return address for main, if it decides to.
93.  pushl $_main # '_main'是编译程序对 main 的内部表示方法
94.  jmp setup_paging # 跳转至第 127 行
95.  L6:
96.  jmp L6 # main should never return here, but
97.  # just in case, we know what happens.
98.  /* This is the default interrupt "handler" :-) */
99.  /* 下面是默认的中断“向量句柄” */
100. int_msg:
101. .asciz "Unknown interrupt\n\r" # 定义字符串“未知中断(回车换行)”
102. .align 2 # 按 4 字节方式对齐内存地址。
103. ignore_int:
104. pushl %eax
105. pushl %ecx
106. pushl %edx
107. push %ds # 这里请注意!! ds,es,fs,gs 等虽然是 16 位的寄存器，但入栈后
108. # 仍然会以 32 位的形式入栈，也即需要占用 4 个字节的堆栈空间
109. push %es
110. push %fs
111. movl $0x10,%eax # 置段选择符（使 ds,es,fs 指向 gdt 表中的数据段）
112. mov %ax,%ds
113. mov %ax,%es
114. mov %ax,%fs
115. pushl $int_msg # 把调用 printk 函数的参数指针（地址）入栈
116. call _printk # 该函数在/kernel/printk.c 中
117. # '_printk'是 printk 编译后模块中的内部表示法
118. popl %eax
119. pop %fs
120. pop %es
```

```
121.  pop %ds
122.  popl %edx
123.  popl %ecx
124.  popl %eax
125.  iret # 中断返回（把中断调用时压入栈的 CPU 标志寄存器（32 位）值也弹出）
126.  .align 2 # 按 4 字节方式对齐内存地址边界
127.  setup_paging: # 首先对 5 页内存（1 页目录 +4 页页表）清零
128.  movl $1024*5,%ecx /* 5 pages - pg_dir+4 page tables */
129.  xorl %eax,%eax
130.  xorl %edi,%edi /* pg_dir is at 0x000 */
131.  # 页目录从 0x000 地址开始
132.  cld;rep;stosl
133.  # 第 1 个页表的属性标志 = 0x00001007 & 0x00000fff = 0x07，表示该页存在、用户可读写
134.  movl $pg0+7,_pg_dir /* set present bit/user r/w */
135.  movl $pg1+7,_pg_dir+4 /* --------- " " --------- */
136.  movl $pg2+7,_pg_dir+8 /* --------- " " --------- */
137.  movl $pg3+7,_pg_dir+12 /* --------- " " --------- */
138.  movl $pg3+4092,%edi # edi  最后一页的最后一项
139.  movl $0xfff007,%eax /* 16Mb - 4096 + 7 (r/w user,p) */
140.  # 最后 1 项对应物理内存页面的地址是 0xfff000，
141.  # 加上属性标志 7，即为 0xfff007.
142.  std # 方向位置位，edi 值递减(4 字节)
143.  stosl /* fill pages backwards - more efficient :-) */
144.  subl $0x1000,%eax # 每填写好一项，物理地址值减 0x1000
145.  jge 1b # 如果小于 0 则说明全添写好了
146.  # 设置页目录基址寄存器 cr3 的值，指向页目录表
147.  xorl %eax,%eax /* pg_dir is at 0x0000 */ # 页目录表在 0x0000 处
148.  movl %eax,%cr3 /* cr3 - page directory start */
149.  # 设置启动使用分页处理（cr0 的 PG 标志，位 31）
150.  movl %cr0,%eax
151.  orl $0x80000000,%eax # 添上 PG 标志
152.  movl %eax,%cr0 /* set paging (PG) bit */
153.  ret /* this also flushes prefetch-queue */
154.  .align 2 # 按 4 字节方式对齐内存地址边界
155.  .word 0
156.  idt_descr: #下面两行是 lidt 指令的 6 字节操作数：长度，基址
157.  .word 256*8-1 # idt contains 256 entries
158.  .long _idt
159.  .align 2
160.  .word 0
161.  gdt_descr: # 下面两行是 lgdt 指令的 6 字节操作数：长度，基址
162.  .word 256*8-1 # so does gdt (not that that's any # not  note.
163.  .long _gdt # magic number, but it works for me :^)
164.  .align 3 # 按 8 字节方式对齐内存地址边界
165.  _idt: .fill 256,8,0 # idt is uninitialized # 256 项，每项 8 字节，填 0
166.  _gdt: .quad 0x0000000000000000 /* NULL descriptor */
167.  .quad 0x00c09a0000000fff /* 16Mb */ # 0x08，内核代码段最大长度 16M
168.  .quad 0x00c0920000000fff /* 16Mb */ # 0x10，内核数据段最大长度 16M
169.  .quad 0x0000000000000000 /* TEMPORARY - don't use */
170.  .fill 252,8,0 /* space for LDT's and TSS's etc */
```

源代码第 4-5 行 bootse ct.s 被 bios-启动子程序加载至 0x7c00 (31k)处，并将自己移到了地址 0x90000 (576k)处，并跳转至那里然后使用 BIOS 中断将'setup'直接加载到自己的后面 (0x90200)(576.5k)，并将 system 加载到地址 0x10000 处。

注意：目前的内核系统最大长度限制为(8*65536)(512k)字节，即使是在 将来这也应该没有问题的。我想让它保持简单明了。这样 512k 的最大内核长度应该足够了，尤其是这里没有象 minix 中一样包含缓冲区高速缓冲。加载程序已经做的够简单了，所以持续的读出错将导致死循环。只能手工重启。只要可能，通过一次读取所有的扇区，加载过程可以做的很快。

2.2 初始化程序（init）

init 进程，它是内核启动的第一个用户级进程。init 有许多很重要的任务，比如启动 getty（用于用户登录）、实现运行级别以及处理孤立进程等。本章将重点解释怎样配置 init 以及如何运用不同的运行级别。

2.2.1 概述

对于 Linux 系统的运行来说，init 程序是最基本的程序之一。一个好的 Linux 发行版本通常随带有一个 init 的配置，这个配置适合于绝大多数系统的工作，在这样一些系统上不需要对 init 做任何事。通常，只有在碰到诸如串行终端挂住了、拨入（不是拨出）调制解调器，或者希望改变缺省的运行级别时才需要关心 init。当内核启动了自己之后（已被装入内存、已经开始运行、已经初始化了所有的设备驱动程序和数据结构等），通过启动用户级程序 init 来完成引导进程的内核部分。因此，init 总是第一个进程（它的进程号总是 1）。

内核在几个位置上来查寻 init，这几个位置以前常用来放置 init，但是 init 最适当的位置（在 Linux 系统上）是/sbin/init。如果内核没有找到 init，它就会试着运行/bin/sh，如果还是失败了，那么系统的启动就宣告失败。init 开始运行时，它通过执行一些管理任务来结束引导进程，例如检查文件系统、清理/tmp、启动各种服务以及为每个终端和虚拟控制台启动 getty，在这些地方用户将登录系统。

在内核源代码的 init/目录中只有一个 main.c 文件。系统在执行完 boot/目录中的 head.s 程序后就会将执行权交给 main.c。该程序虽然不长，但却包括了内核初始化的所有工作。因此在阅读该程序的代码时需要参照很多其他程序中的初始化部分。如果能完全理解这里调用的所有程序，那么看完这章内容后应该对 Linux 内核有了大致的了解。

在本章中将接触大量的 C 程序代码，因此读者最好具有一定的 C 语言基础。在注释 C 语言程序时，为了与程序中原有的注释相区别，我们使用“//”作为注释语句的开始。有关原有注释的翻译则采用与其一样的注释标志。对于程序中包含的头文件（*.h），仅作概要含义的解释，具体详细注释内容将在注释相应头文件的章节中给出。

2.2.2 main.c 文件

main.c 文件用于对内存变量指明主内存的开始地址、系统所拥有的内存容量和作为高速缓冲区内存的末端地址。

1. 功能描述

main.c 文件首先利用前面 setup.s 程序取得的系统参数设置系统的根文件设备号以及一些内

存全局变量。这些内存变量指明了主内存的开始地址、系统所拥有的内存容量和作为高速缓冲区内存的末端地址。如果还定义了虚拟盘（RAMDISK），则主内存将适当减少。其中，高速缓冲部分还要扣除被显存和 ROM BIOS 占用的部分。高速缓冲区是用于磁盘等块设备临时存放数据的地方，以 1K（1024）字节为一个数据块单位。

主内存区域的内存是由内存管理模块 mm 通过分页机制进行管理分配，以 4K 字节为一个内存页单位。内核程序可以自由访问高速缓冲中的数据，但需要通过 mm 才能使用分配到的内存页面。然后，内核进行所有方面的硬件初始化工作。包括陷阱门、块设备、字符设备和 tty，以及人工设置第一个任务（task 0）。待所有初始化工作完成后就设置中断允许标志以开启中断，main()也切换到了任务 0 中运行。在阅读这些初始化子程序时，最好是跟着被调用的程序深入进去看，如果实在看不下去了，就暂时先放一放，继续看下一个初始化调用。在有些理解之后再继续研究没有看完的地方。

在整个内核完成初始化后，内核将执行权切换到了用户模式（任务 0），也即 CPU 从 0 特权级切换到了第 3 特权级。此时 main.c 的主程序就工作在任务 0 中。然后系统第一次调用进程创建函数 fork()，创建出一个用于运行 init()的子进程。

2．代码注释

main.c 文件的详细代码注释如下：


```
1.  #define __LIBRARY__ // 定义该变量是为了包括定义在 unistd.h 中的内嵌汇编代码等信息
2.  #include <unistd.h> // *.h 头文件所在的默认目录是 include/，则在代码中就不用明确指明位置
3.  #include <time.h> // 时间类型头文件。其中最主要定义了 tm 结构和一些有关时间的函数原形
4.  // 本程序将会在移动到用户模式（切换到任务 0）后才执行 fork()，因此避免了在内核空间写时
    复制问题
5.  // 在执行了 moveto_user_mode()之后，本程序就以任务 0 的身份在运行了。而任务 0 是所有将
    创建的子
6.  // 进程的父进程。当创建第一个子进程时，任务 0 的堆栈也会被复制。因此希望在 main.c 运
    行在任务 0
7.  // 的环境下时不要有对堆栈的任何操作，以免弄乱堆栈，从而也不会弄乱所有子进程的堆栈
8.  static inline _syscall0(int,fork)
9.  static inline _syscall0(int,pause) // int pause()系统调用：暂停进程的执行，直到收到一个信号
10. static inline _syscall1(int,setup,void *,BIOS)      // int setup(void * BIOS)系统调用，仅用于 linux
                                                        // 初始化（仅在这个程序中被调用）
11. static inline _syscall0(int,sync) // int sync()系统调用：更新文件系统
12. #include <linux/tty.h>      // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数
13. #include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、第 1 个初始任务的数
    // 据。还有一些以宏的形式定义的有关描述符参数设置和获取的嵌入式汇编函数程序
14. #include <linux/head.h>    // head 头文件，定义了段描述符的简单结构和几个选择符常量
15. #include <asm/system.h>    // 系统头文件。以宏的形式定义了许多有关设置或修改描述符/中断门
                              // 等的嵌入式汇编子程序
16. #include <asm/io.h>        // io 头文件。以宏的嵌入汇编程序形式定义对 io 端口操作的函数
17. #include <stddef.h>        // 标准定义头文件。定义了 NULL, offsetof(TYPE, MEMBER)
18. #include <stdarg.h>        // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了 1 个
19. // 类型(va_list)和 3 个宏(va_start, va_arg 和 va_end)，vsprintf、
20. // vprintf、vfprintf
21. #include <unistd.h>
22. #include <fcntl.h>         // 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义
23. #include <sys/types.h>     // 类型头文件。定义了基本的系统数据类型
24. #include <linux/fs.h>      // 文件系统头文件。定义文件表结构（file,buffer_head,m_inode 等）
```

```
25.  static char printbuf[1024];                          // 静态字符串数组，用作内核显示信息的缓存
26.  extern int vsprintf();                               // 送格式化输出到一字符串中
27.  extern void init(void);                              // 初始化函数
28.  extern void blk_dev_init(void);                      // 块设备初始化子程序
29.  extern void chr_dev_init(void);                      // 字符设备初始化
30.  extern void hd_init(void);                           // 硬盘初始化程序
31.  extern void floppy_init(void);                       // 软驱初始化程序
32.  extern void mem_init(long start, long end);          // 内存管理初始化
33.  extern long rd_init(long mem_start, int length);     // 虚拟盘初始化
34.  extern long kernel_mktime(struct tm * tm);           // 计算系统开机启动时间（秒）
35.  extern long startup_time;                            // 内核启动时间（开机时间）（秒）
36.  #define EXT_MEM_K (*(unsigned short *)0x90002) // 1M 以后的扩展内存大小（KB）
37.  #define DRIVE_INFO (*(struct drive_info *)0x90080)       // 硬盘参数表基址
38.  #define ORIG_ROOT_DEV (*(unsigned short *)0x901FC)   // 根文件系统所在设备号
39.  #define CMOS_READ(addr) ({ \                        // 这段宏读取 CMOS 实时时钟信息
40.  outb_p(0x80|addr,0x70); \ // 0x70 是写端口号，0x80|addr 是要读取的 CMOS 内存地址
41.  inb_p(0x71); \                                      // 0x71 是读端口号
42.  })
43.  // 定义宏。将 BCD 码转换成二进制数值
44.  #define BCD_TO_BIN(val) ((val)=((val)&15) + ((val)>>4)*10)
45.  // 该子程序取 CMOS 时钟，并设置开机时间 startup_time（秒）。参见后面 CMOS 内存列表
46.  static void time_init(void)
47.  {
48.  struct tm time; // 时间结构 tm 定义在 include/time.h 中
49.  // CMOS 的访问速度很慢。为了减小时间误差，在读取了下面循环中所有数值后，若此时 CMOS
     // 中秒值
50.  // 发生了变化，那么就重新读取所有值。这样内核就能把与 CMOS 的时间误差控制在 1 秒之内
51.  do {
52.  time.tm_sec = CMOS_READ(0);                          // 当前时间秒值（均是 BCD 码值）
53.  time.tm_min = CMOS_READ(2);                          // 当前分钟值
54.  time.tm_hour = CMOS_READ(4);                         // 当前小时值
55.  time.tm_mday = CMOS_READ(7);                         // 一月中的当天日期
56.  time.tm_mon = CMOS_READ(8);                          // 当前月份（1～12）
57.  time.tm_year = CMOS_READ(9);                         // 当前年份
58.  } while (time.tm_sec != CMOS_READ(0));
59.  BCD_TO_BIN(time.tm_sec);                             // 转换成二进制数值
60.  BCD_TO_BIN(time.tm_min);
61.  BCD_TO_BIN(time.tm_hour);
62.  BCD_TO_BIN(time.tm_mday);
63.  BCD_TO_BIN(time.tm_mon);
64.  BCD_TO_BIN(time.tm_year);
65.  time.tm_mon--;                                       // tm_mon 中月份范围是 0—11
66.  // 调用 kernel/mktime.c 中函数，计算从 1970 年 1 月 1 日 0 时起到开机当日经过的秒数，作
     // 为开机时间
67.  startup_time = kernel_mktime(&time);
68.  }
69.  static long memory_end = 0;                          // 机器具有的物理内存容量（字节数）
70.  static long buffer_memory_end = 0;                   // 高速缓冲区末端地址
71.  static long main_memory_start = 0;                   // 主内存（将用于分页）开始的位置
72.  struct drive_info { char dummy[32]; } drive_info; // 用于存放硬盘参数表信息
```

```
73.     void main(void)
74.     ROOT_DEV = ORIG_ROOT_DEV;
75.     drive_info = DRIVE_INFO;                        // 复制 0x90080 处的硬盘参数表
76.     memory_end = (1<<20) + (EXT_MEM_K<<10);         // 内存大小=1Mb 字节+扩展内存（k）*1024 字节
77.     memory_end &= 0xfffff000;                       // 忽略不到 4Kb（1 页）的内存数
78.     if (memory_end > 16*1024*1024)                  // 如果内存超过 16Mb，则按 16Mb 计
79.     memory_end = 16*1024*1024;
80.     if (memory_end > 12*1024*1024)                  // 如果内存>12Mb，则设置缓冲区末端=4Mb
81.     buffer_memory_end = 4*1024*1024;
82.     else if (memory_end > 6*1024*1024)              // 否则如果内存>6Mb，则设置缓冲区末端=2Mb
83.     buffer_memory_end = 2*1024*1024;
84.     else
85.     buffer_memory_end = 1*1024*1024;                // 否则则设置缓冲区末端=1Mb
86.     main_memory_start = buffer_memory_end;          // 主内存起始位置=缓冲区末端；
87.     // 如果定义了内存虚拟盘，则初始化虚拟盘。此时主内存将减少。
88.     #ifdef RAMDISK
89.     main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
90.     #endif
91.     // 以下是内核进行所有方面的初始化工作。阅读时最好跟着调用的程序深入进去看，若实在看
92.     // 不下去了，就先放一放，继续看下一个初始化调用
93.       mem_init(main_memory_start,memory_end);
94.       trap_init();                                  // 陷阱门（硬件中断向量）初始化函数
95.       blk_dev_init();                               // 块设备初始化函数
96.       chr_dev_init();                               // 字符设备初始化函数
97.       tty_init();                                   // tty 初始化函数
98.       time_init();                                  // 设置开机启动时间 startup_time 的函数
99.       sched_init();                                 // 调度程序初始化（加载了任务 0 的 tr,ldtr）
100.      buffer_init(buffer_memory_end);               // 缓冲管理初始化，建内存链表等
101.      hd_init();                                    // 硬盘初始化函数
102.      floppy_init();                                // 软驱初始化函数
103.      sti();                                        // 所有初始化工作都做完了，开启中断
104.    // 下面过程通过在堆栈中设置的参数，利用中断返回指令启动任务 0 执行
105.    move_to_user_mode();                            // 移到用户模式下执行的函数
106.    If (!fork()) {
107.    init();                                         // 在新建的子进程（任务 1）中执行
108.    }
109.    // 下面代码开始以任务 0 的身份运行。
110.    for(;;) pause();
111.    }
112.    static int printf(const char *fmt, ...)
113.    {
114.    va_list args;
115.    int i;
116.    va_start(args, fmt);
117.    write(1,printbuf,i=vsprintf(printbuf, fmt, args));
118.    va_end(args);
119.    return i;
120.    }
121.    static char * argv_rc[] = { "/bin/sh", NULL };  // 调用执行程序时参数的字符串数组
122.    static char * envp_rc[] = { "HOME=/", NULL };   // 调用执行程序时的环境字符串数组
```

```
123.    static char * argv[] = { "-/bin/sh",NULL };        // 同上
124.    static char * envp[] = { "HOME=/usr/root", NULL };
125.    // 上面 123 行中 argv[0]中的字符 "-" 是传递给 shell 程序 sh 的一个标志。通过识别该标志，sh
126.    // 程序会作为登录 shell 执行。其执行过程与在 shell 提示符下执行 sh 不太一样
127.    // 在 main()中已经进行系统初始化，包括内存管理、各种硬件设备和驱动程序。init()函数运行在
128.    // 任务 0 第 1 次创建的子进程（任务 1）中。它首先对第一个将要执行的程序（shell）的环
    境进行
129.    // 初始化，然后加载该程序并执行之
130.    void init(void)
131.    {
132.    int pid,i;
133.    // 这是一个系统调用。用于读取硬盘参数包括分区表信息并加载虚拟盘（若存在的话）和安
    装根文件系统设备。该函数是用 27 行上的宏定义的，对应函数是 sys_setup()在 kernel/blk_drv/hd.c
    文件的 71 行
134.    setup((void *) &drive_info);
135.    (void) open("/dev/tty0",O_RDWR,0);
136.    (void) dup(0);                          // 复制句柄，产生句柄 1 号——stdout 标准输出设备
137.    (void) dup(0);                          // 复制句柄，产生句柄 2 号——stderr 标准出错输出设备
138.    // 下面打印缓冲区块数和总字节数，每块 1024 字节，以及主内存区空闲内存字节数
139.    printf("%d buffers = %d bytes buffer space\n\r",NR_BUFFERS,
140.    NR_BUFFERS*BLOCK_SIZE);
141.    printf("Free mem: %d bytes\n\r",memory_end-main_memory_start);
142.    if (!(pid=fork())) {
143.    close(0);
144.    if (open("/etc/rc",O_RDONLY,0))
145.    _exit(1);                               // 如果打开文件失败，则退出(lib/_exit.c,10)
146.    execve("/bin/sh",argv_rc,envp_rc);      // 替换成/bin/sh 程序并执行
147.    _exit(2);                               // 若 execve()执行失败则退出
148.    }
149.    if (pid>0)
150.    while (pid != wait(&i))
151.    /* 空循环 */
152.    while (1) {
153.    if ((pid=fork())<0) {
154.    printf("Fork failed in init\r\n");
155.    continue;
156.    }
157.    if (!pid) {                             // 判断是否是新的子进程
158.    close(0);close(1);close(2);
159.    setsid();                               // 创建一新的会话期，见后面说明
160.    (void) open("/dev/tty0",O_RDWR,0);
161.    (void) dup(0);
162.    (void) dup(0);
163.    _exit(execve("/bin/sh",argv,envp));
164.    }
165.    while (1)
166.    if (pid == wait(&i))
167.    break;
168.    printf("\n\rchild %d died with code %04x\n\r",pid,i);
169.    sync();                                 // 同步操作，刷新缓冲区
```

```
170.        }
171.        _exit(0); /* 注意！是_exit()，不是 exit() */
172.        }
```

源代码第 8-11 行包含一些内嵌语句，从内核空间创建进程（forking）将导致没有写时复制（COPY ON WRITE），直到执行一个 execve 调用。这对堆栈可能带来问题。处理的方法是在 fork()调用之后不让 main()使用任何堆栈。因此就不能有函数调用——这意味着 fork 也要使用内嵌的代码，否则我们在从 fork()退出时就要使用堆栈了。

实际上只有 pause 和 fork 需要使用内嵌方式，以保证从 main()中不会弄乱堆栈，但是我们同时还定义了其他一些函数。源代码第 112-120 行产生格式化信息并输出到标准输出设备的 printf()方法，这里是指屏幕上显示。参数'*fmt'指定输出将采用的格式，参见各种标准 C 语言书籍。该子程序正好是 vsprintf 如何使用的一个例子。该程序使用 vsprintf()将格式化的字符串放入 printbuf 缓冲区，然后用 write()将缓冲区的内容输出到标准设备（1--stdout）。

第3章 头 文 件

通常，程序在使用一个函数之前，首先应该申明该函数，具体做法是把同一类函数或数据结构以及常数的申明放在一个头文件中。头文件中也可以包括任何相关的类型定义和宏。在这些头文件中主要描述了内核所用到的所有数据结构、初始化数据、常数和宏定义，也包括少量的程序代码。

在程序源代码文件中则使用预处理指令“#include”来引用相关的头文件。在一般应用程序源代码中，头文件与开发环境中的库文件有着不可分割的紧密联系，库中的每个函数都需要在头文件中加以申明。

3.1 include/目录下的文件

Linux 0.95 版内核中共有 39 个头文件（*.h），其中 asm/子目录中含有 4 个，linux/子目录中含有 13 个，sys/子目录中含有 9 个。头文件目录中总共有 32 个.h 头文件。

asm/目录下主要用于存放与计算机体系结构密切相关的函数申明或数据结构的头文件。linux/目录下是 Linux 内核程序使用的一些头文件。sys/目录下存放着 5 个与内核资源相关头文件。

表 3-1 为 Linux/include/目录下的文件。

表 3-1 linux/include/目录下的文件

名　　称	大 小（B）	修 改 时 间
asm/	4229	2009-05-09 15:38:32
Linux/	37703	2009-05-09 15:38:32
sys/	8110	2009-05-09 15:38:32
a.out.h	6047	1991-09-17 23:10:50
ctype.h	1049	1991-11-08 01:30:48
errno.h	1382	1992-02-19 04:36:18
fcntl.h	1358	1992-02-27 19:28:08
signal.h	2248	1992-03-01 23:26:12
stdarg.h	780	1991-09-17 23:02:24
stddef.h	285	1991-12-28 11:19:06
string.h	7973	1992-03-06 02:24:28
termios.h	5293	1992-01-30 23:44:54
time.h	873	1992-02-01 01:05:46
unistd.h	7482	1992-03-04 18:56:06
utime.h	225	1991-09-17 23:03:38

3.2 a.out.h 文件

该文件包括三个数据结构和一些宏函数，用于定义被加载的可执行文件结构。

a.out.h 文件定义了 a.out 二进制执行文件格式和一些宏。要用于加载程序 fs/exec.c 中。该文件不属于标准 C 库，它是内核专用的头文件。但由于与标准库的头文件名没有冲突，因此在 Linux

系统中一般可以放在/usr/include/目录下，以供涉及相关内容的程序使用。该头文件中定义了目标文件的一种 a.out（Assembly out）格式。a.out.h 文件的详细代码注释如下：

```
1.    #ifndef _A_OUT_H
2.    #define _A_OUT_H
3.    #define __GNU_EXEC_MACROS__
4.    struct exec {// 执行头文件结构
5.    unsigned long a_magic;          /*执行文件魔数*/
6.    unsigned a_text;                /*代码长度，字节数*/
7.    unsigned a_data;                /*数据长度，字节数*/
8.    unsigned a_bss;                 /*文件中的未初始化数据区长度，字节数*/
9.    unsigned a_syms;                /*文件中的符号表长度，字节数*/
10.   unsigned a_entry;               //开始地址
11.   unsigned a_trsize;              /*代码重定位信息长度，字节数*/
12.   unsigned a_drsize;              /*数据重定位信息长度，字节数*/
13.   };
      //用于取执行结构中的魔数
14.   #ifndef N_MAGIC
15.   #define N_MAGIC(exec) ((exec).a_magic)
16.   #endif
17.   #ifndef OMAGIC
18.   /* Code indicating object file or impure executable.  */
      /* 指明为目标文件或者目标不明确的可执行文件的代码*/
19.   #define OMAGIC 0407
      //最早在 PDP-11 计算机上，魔数是八进制数 0407
20.
21.   /* 指明为纯可执行文件的代码*/
22.   #define NMAGIC 0410
23.   /* 指明为请求分页处理的可执行文件*/
24.   #define ZMAGIC 0413
25.   #endif
      //如果魔数不能被识别，则返回真
26.   #ifndef N_BADMAG
27.   #define N_BADMAG(x)
28.    (N_MAGIC(x) != OMAGIC && N_MAGIC(x) != NMAGIC
29.   && N_MAGIC(x) != ZMAGIC)
30.   #endif
31.   #define _N_BADMAG(x)
32.    (N_MAGIC(x) != OMAGIC && N_MAGIC(x) != NMAGIC
33.   && N_MAGIC(x) != ZMAGIC)
      // 程序头在内存中的偏移位置
34.   #define _N_HDROFF(x) (SEGMENT_SIZE - sizeof (struct exec))
      // 代码起始偏移值
35.   #ifndef N_TXTOFF
36.   #define N_TXTOFF(x) \
37.   (N_MAGIC(x) == ZMAGIC ? _N_HDROFF((x)) + sizeof (struct exec) : sizeof (struct exec))
38.   #endif
      //数据起始偏移值
39.   #ifndef N_DATOFF
40.   #define N_DATOFF(x) (N_TXTOFF(x) + (x).a_text)
```

```
41.  #endif
     //代码重定位信息偏移值
42.  #ifndef N_TRELOFF
43.  #define N_TRELOFF(x) (N_DATOFF(x) + (x).a_data)
44.  #endif
     // 数据重定位信息偏移值
45.  #ifndef N_DRELOFF
46.  #define N_DRELOFF(x) (N_TRELOFF(x) + (x).a_trsize)
47.  #endif
     // 符号表偏移值
48.  #ifndef N_SYMOFF
49.  #define N_SYMOFF(x) (N_DRELOFF(x) + (x).a_drsize)
50.  #endif
     // 字符串信息偏移值
51.  #ifndef N_STROFF
52.  #define N_STROFF(x) (N_SYMOFF(x) + (x).a_syms)
53.  #endif
54.  /* 代码段加载到内存中后的地址*/
55.  #ifndef N_TXTADDR
56.  #define N_TXTADDR(x) 0
57.  #endif
     /* 数据段加载到内存中后的地址。注意，对于下面没有列出名称的机器，需要你自己来定义
58.  对应的 SEGMENT_SIZE */
59.  #if defined(vax) || defined(hp300) || defined(pyr)
60.  #define SEGMENT_SIZE PAGE_SIZE
61.  #endif
62.  #ifdef          hp300
63.  #define   PAGE_SIZE     4096
64.  #endif
65.  #ifdef          sony
66.  #define   SEGMENT_SIZE      0x2000
67.  #endif
68.  #ifdef is68k
69.  #define SEGMENT_SIZE 0x20000
70.  #endif
71.  #if defined(m68k) && defined(PORTAR)
72.  #define PAGE_SIZE 0x400
73.  #define SEGMENT_SIZE PAGE_SIZE
74.  #endif
75.  #define PAGE_SIZE 4096
76.  #define SEGMENT_SIZE 1024
     // 以段为界的大小
77.  #define _N_SEGMENT_ROUND(x) (((x) + SEGMENT_SIZE - 1) & ~(SEGMENT_SIZE - 1))
     // 代码段尾地址
78.  #define _N_TXTENDADDR(x) (N_TXTADDR(x)+(x).a_text)
     // 数据开始地址
79.  #ifndef N_DATADDR
80.  #define N_DATADDR(x) \
81.  (N_MAGIC(x)==OMAGIC? (_N_TXTENDADDR(x)) \
82.  : (_N_SEGMENT_ROUND (_N_TXTENDADDR(x))))
```

```
83.    #endif
84.    /* bss 段加载到内存以后的地址*/
85.    #ifndef N_BSSADDR
86.    #define N_BSSADDR(x) (N_DATADDR(x) + (x).a_data)
87.    #endif
       //nlist 结构。符号表记录结构
88.    #ifndef N_NLIST_DECLARED
89.    struct nlist {
90.    union {
91.    char *n_name;
92.    struct nlist *n_next;
93.    long n_strx;
94.    } n_un;
95.    unsigned char n_type;
96.    char n_other;
97.    short n_desc;
98.    unsigned long n_value;
99.    };
100.   #endif
       //下面定义 exec 结构中的变量偏移值
101.   #ifndef N_UNDF
102.   #define N_UNDF 0
103.   #endif
104.   #ifndef N_ABS
105.   #define N_ABS 2
106.   #endif
107.   #ifndef N_TEXT
108.   #define N_TEXT 4
109.   #endif
110.   #ifndef N_DATA
111.   #define N_DATA 6
112.   #endif
113.   #ifndef N_BSS
114.   #define N_BSS 8
115.   #endif
116.   #ifndef N_COMM
117.   #define N_COMM 18
118.   #endif
119.   #ifndef N_FN
120.   #define N_FN 15
121.   #endif
       //以下 3 个常量定义是 nlist 结构中 n_type 变量的屏蔽码
122.   #ifndef N_EXT
123.   #define N_EXT 1
124.   #endif
125.   #ifndef N_TYPE
126.   #define N_TYPE 036
127.   #endif
128.   #ifndef N_STAB
129.   #define N_STAB 0340
```

```
130.  #endif
131.  #define N_INDR 0xa
132.  /* 以下这些符号在目标文件中是作为链接程序 LD 的输入*/
133.  #define   N_SETA   0x14        /* 绝对集合元素符号*/
134.  #define   N_SETT   0x16        /* 代码集合元素符号*/
135.  #define   N_SETD   0x18        /* 数据集合元素符号*/
136.  #define   N_SETB   0x1A        /* Bss 集合元素符号*/
137.  /* 下面是 LD 的输出*/
138.  #define N_SETV     0x1C
139.  /* 指向数据区中集合向量*/
140.  #ifndef N_RELOCATION_INFO_DECLARED
      // 重定位信息结构
141.  struct relocation_info
142.  {
143.
144.  /* 需要重定位的地址（在段内）*/
145.  int r_address;
146.  unsigned int r_symbolnum:24;
      /* 非零意味着值是一个 pc 相关的偏移值，因而在其自己地址空间
147.  以及符号或指定的节改变时，需要被重定位*/
148.  unsigned int r_pcrel:1;
149.
150.  /* 需要被重定位的字段长度（是 2 的次方）
      因此，若值是 2 则表示 1<<2 字节数。*/
151.  unsigned int r_length:2;
152.  /* 1 => 以符号的值重定位。r_symbolnum 是文件符号表中符号的索引
153.  0 => 以段的地址进行重定位。r_symbolnum 是 N_TEXT、N_DATA、N_BSS 或 N_ABS
154.  (N_EXT 比特位也可以被设置，但是毫无意义)*/
155.  unsigned int r_extern:1;
156.
157.  /* 没有使用的 4 bits，但是当进行写一个目标文件时最好将它们复位掉*/
158.  unsigned int r_pad:4;
159.  };
160.  #endif
161.  #endif
```

该文件可被相应地分成三个部分：第一部分给出并描述了目标文件执行头结构和相关的宏定义。第二部分对符号表项结构进行定义和说明。第三部分对重定位表项结构进行定义和说明。

3.3 const.h 文件

const.h 文件定义了系统中经常用到的一些常量符号。所以在内核中很多文件中都要引用该头文件。该文件定义了 i 结点中文件属性和类型 i_mode 字段所用到的一些符号。const.h 文件的详细代码注释如下：

```
1.  #ifndef _CONST_H
2.  #define _CONST_H
3.  #define BUFFER_END 0x200000                    //缓冲使用内存的末端
                                                   // i 结点数据结构中 i_mode 字段的标志位
4.  #define I_TYPE              0170000            //指明 i 结点类型
5.  #define I_DIRECTORY         0040000            //是目录文件
6.  #define I_REGULAR           0100000            //常规文件，不是目录文件或特殊文件
7.  #define I_BLOCK_SPECIAL     0060000            //块设备特殊文件
8.  #define I_CHAR_SPECIAL      0020000            //字符设备特殊文件
9.  #define I_NAMED_PIPE        0010000            //命名管道
10. #define I_SET_UID_BIT       0004000            //在执行时设置有效用户 id 类型
11. #define I_SET_GID_BIT       0002000            //在执行时设置有效组 id 类型
12. #endif
```

3.4 ctype.h 文件

ctype.h 文件是关于字符测试和处理的头文件。因此在涉及到需要处理字符操作的文件中往往需要包含该头文件。该文件定义了一些有关字符类型判断和转换的宏。ctype.h 文件的详细代码注释如下：


```
1.  #ifndef _CTYPE_H
2.  #define _CTYPE_H
3.  #define _U 0x01                                //用于大写字符[A-Z]
4.  #define _L 0x02                                //用于小写字符[a-z]
5.  #define _D 0x04                                //用于数字[0-9]
6.  #define _C 0x08                                //用于控制字符
7.  #define _P 0x10                                //用于标点字符
8.  #define _S 0x20                                //用于空白字符，如空格、\t、\n 等
9.  #define _X 0x40                                //用于十六进制数字
10. #define _SP 0x80                               //用于空格字符（0x20）
11. extern unsigned char _ctype[];  //字符特性数组（表），定义了各个字符对应上面的属性
12. extern char _ctmp;                             //一个临时字符变量
13. #define isalnum(c) ((_ctype+1)[c]&(_U|_L|_D))  // 字符或数字[A-Z]、[a-z]或[0-9]
14. #define isalpha(c) ((_ctype+1)[c]&(_U|_L))     // 字符
15. #define iscntrl(c) ((_ctype+1)[c]&(_C))        // 控制字符
16. #define isdigit(c) ((_ctype+1)[c]&(_D))        // 数字
17. #define isgraph(c) ((_ctype+1)[c]&(_P|_U|_L|_D))  // 图形字符
18. #define islower(c) ((_ctype+1)[c]&(_L))        // 小写字符
19. #define isprint(c) ((_ctype+1)[c]&(_P|_U|_L|_D|_SP))  // 可打印字符
20. #define ispunct(c) ((_ctype+1)[c]&(_P))        // 标点符号
21. #define isspace(c) ((_ctype+1)[c]&(_S))        // 空白字符如空格,\f,\n,\r,\t,\v
22. #define isupper(c) ((_ctype+1)[c]&(_U))        // 大写字符
23. #define isxdigit(c) ((_ctype+1)[c]&(_D|_X))    // 十六进制数字
24. #define isascii(c) (((unsigned) c)<=0x7f)      //ASCII 字符
25. #define toascii(c) (((unsigned) c)&0x7f)       // 转换成 ASCII 字符
26. #define tolower(c) (_ctmp=c,isupper(_ctmp)?_ctmp-('A'-'a'):_ctmp)   // 转换成对应小写字符
27. #define toupper(c) (_ctmp=c,islower(_ctmp)?_ctmp-('a'-'A'):_ctmp)   // 转换成对应大写字符
28. #endif
```

3.5 errno.h 文件

errno.h 文件定义了 Linux 系统中的一些出错码（出错号）的常量符号。在某些情况下，想知道具体的出错号，就可以通过读取 errno 的值来确定最后一次错误的出错号。errno.h 文件的详细代码注释如下：

```
1.  #ifndef _ERRNO_H
2.  #define _ERRNO_H
3.  extern int errno;
4.  #define ERROR       99          //一般错误
5.  #define EPERM       1           //操作没有许可
6.  #define ENOENT      2           //文件或目录不存在
7.  #define ESRCH       3           //指定的进程不存
8.  #define EINTR       4           //中断的函数调用
9.  #define EIO         5           //输入/输出错
10. #define ENXIO       6           //指定设备或地址不存在
11. #define E2BIG       7           //参数列表太长
12. #define ENOEXEC     8           //执行程序格式错误
13. #define EBADF       9           //文件句柄（描述符）错误
14. #define ECHILD      10          //子进程不存在
15. #define EAGAIN      11          //资源暂时不可用
16. #define ENOMEM      12          //内存不足
17. #define EACCES      13          //没有许可权限
18. #define EFAULT      14          //地址错
19. #define ENOTBLK     15          //不是块设备文件
20. #define EBUSY       16          //资源正忙
21. #define EEXIST      17          //文件已存在
22. #define EXDEV       18          //非法连接
23. #define ENODEV      19          //设备不存在
24. #define ENOTDIR     20          //不是目录文件
25. #define EISDIR      21          //是目录文件
26. #define EINVAL      22          //参数无效
27. #define ENFILE      23          //系统打开文件数太多
28. #define EMFILE      24          //打开文件数太多
29. #define ENOTTY      25          //不恰当的 IO 控制操作（没有 tty 终端）
30. #define ETXTBSY     26          //不再使用
31. #define EFBIG       27          //文件太大
32. #define ENOSPC      28          //设备已满（设备已经没有空间）
33. #define ESPIPE      29          //无效的文件指针重定位
34. #define EROFS       30          //文件系统只读
35. #define EMLINK      31          //连接太多
36. #define EPIPE       32          //道错
37. #define EDOM        33          //域（domain）出错
38. #define ERANGE      34          //结果太大
39. #define EDEADLK     35          //避免资源死锁
40. #define ENAMETOOLONG 36         //文件名太长
41. #define ENOLCK      37          //没有锁定可用
42. #define ENOSYS      38          //功能还没有实现
43. #define ENOTEMPTY   39          //目录不空
44. #define ELOOP       40          //过多符号连接
```

```
45. /* 对用户程序永远是透明的 */
46. #define ERESTARTSYS      512
47. #define ERESTARTNOINTR 513
48. #endif
```

从第 3 行知道第 47 行是系统调用以及很多库函数返回一个特殊的值以表示操作失败或出错。这个值通常选择-1 或者其他一些特定的值来表示。但是这个返回值仅说明错误发生了。如果需要知道出错的类型，就需要查看表示系统出错号的变量 errno。该变量即在 errno.h 文件中申明。在程序开始执行时该变量值被初始化为 0。

3.6 fcntl.h 文件

该文件是包含控制选项的头文件。例如表示只读的 O_RDONLY 控制选项、表示只写的 O_WRONLY 控制选项和表示读写的 O_RDWR 控制选项，这些控制选项都定义在 fcntl.h 文件中。

本文件主要定义 fcntl()和 open()函数中用到的一些参数的可选项。fcntl.h 文件详细代码注释如下：

```
1.  #ifndef _FCNTL_H
2.  #define _FCNTL_H
3.  #include <sys/types.h> // 类型头文件。定义了基本的系统数据类型
4.  /* open/fcntl - NOCTTY 和 NDELAY 现在还没有实现*/
5.  #define O_ACCMODE      00003                    //文件访问模式屏蔽码
    // 打开文件 open()和文件控制 fcntl()函数使用的文件访问模式。同时只能使用三者之一
6.  #define O_RDONLY       00                       //以只读方式打开文件
7.  #define O_WRONLY       01                       //以只写方式打开文件
8.  #define O_RDWR         02                       //以读写方式打开文件
9.  #define O_CREAT        00100                    //如果文件不存在就创建
10. #define O_EXCL         00200                    //独占使用文件标志
11. #define O_NOCTTY       00400                    //不分配控制终端
12. #define O_TRUNC        01000
    //若文件已存在且是写操作，则长度截为 0
13. #define O_APPEND       02000                    //以添加方式打开，文件指针置为文件尾
14. #define O_NONBLOCK     04000                    //非阻塞方式打开和操作文件
15. #define O_NDELAY       O_NONBLOCK               //非阻塞方式打开和操作文件
16. #define F_DUPFD        0                        //拷贝文件句柄为最小数值的句柄
17. #define F_GETFD        1                        //取文件句柄标志
18. #define F_SETFD        2                        //设置文件句柄标志
19. #define F_GETFL        3                        //取文件状态标志和访问模式
20. #define F_SETFL        4                        //设置文件状态标志和访问模式
21. #define F_GETLK        5                        //返回阻止锁定的 flock 结构
22. #define F_SETLK        6          //设置（F_RDLCK 或 F_WRLCK）或清除（F_UNLCK）锁定
23. #define F_SETLKW       7          //等待设置或清除锁定
    /* 用于 F_GETFL 或 F_SETFL */
24. #define FD_CLOEXEC     1
    /* 实际上只要低位为 1 即可*/
25. #define F_RDLCK        0          //共享或读文件锁定
26. #define F_WRLCK        1          //独占或写文件锁定
27. #define F_UNLCK        2          //文件解锁
    //文件锁定操作数据结构。描述了受影响文件段的类型（l_type）、开始偏移（l_whence）、
    //相对偏移（l_start）、锁定长度（l_len）和实施锁定的进程 id
```

```
28.  struct flock {
29.  short l_type;                    //锁定类型（F_RDLCK，F_WRLCK，F_UNLCK）
30.  short l_whence;                  //开始偏移（SEEK_SET，SEEK_CUR 或 SEEK_END）
31.  off_t l_start;                   //阻塞锁定的开始处。相对偏移（字节数）
32.       off_t l_len;                //阻塞锁定的大小；如果是 0 则为到文件末尾
33.       pid_t l_pid;                //加锁的进程 id
34.  };
      //创建新文件或重写一个已存在文件
35.  extern int creat(const char * filename,mode_t mode);
      //文件句柄操作，会影响文件的打开
      //参数 fildes 是文件句柄，cmd 是操作命令
36.  extern int fcntl(int fildes,int cmd, ...);
      //打开文件。在文件与文件句柄之间建立联系。参数 filename 是欲打开文件的文件名
37.  extern int open(const char * filename, int flags, ...);
38.  #endif
```

3.7 signal.h 文件

信号文件提供了一种处理异步事件的方法。信号也被称为是一种软中断。在 Linux 内核中用使用信函安装函数时所用到的 sigaction 结构体时，就必须引用 signal.h 文件。

本文件定义了内核中使用的所有信号的名称和基本操作函数。对于使用信号机制的 Linux 内核来说，该文件是非常重要的。signal.h 文件的详细代码注释如下：

```
1.  #ifndef _SIGNAL_H
2.  #define _SIGNAL_H
3.  #include <sys/types.h>                    //类型头文件。定义了基本的系统数据类型
4.  typedef int sig_atomic_t;                 //定义信号原子操作类型
5.  typedef unsigned int sigset_t;            /* 32 bits */  // 定义信号集类型
6.  #define _NSIG             32             //定义信号种类——32 种
7.  #define NSIG              _NSIG
     // 以下这些是 Linux 0.95 内核中定义的信号
8.   #define SIGHUP           1              //挂断控制终端或进程
9.   #define SIGINT           2              //来自键盘的中断
10.  #define SIGQUIT          3              //来自键盘的退出
11.  #define SIGILL           4              //非法指令
12.  #define SIGTRAP          5              //跟踪断点
13.  #define SIGABRT          6              //异常结束
14.  #define SIGIOT           6              //异常结束
15.  #define SIGUNUSED        7              //没有使用
16.  #define SIGFPE           8              //协处理器出错
17.  #define SIGKILL          9              //强迫进程终止
18.  #define SIGUSR1          10             //用户信号 1，进程可使用
19.  #define SIGSEGV          11             //无效内存引用
20.  #define SIGUSR2          12             //用户信号 2，进程可使用
21.  #define SIGPIPE          13             //管道写出错，无读者
22.  #define SIGALRM          14             //实时定时器报警
23.  #define SIGTERM          15             //进程终止
24.  #define SIGSTKFLT        16             //栈出错（协处理器）
25.  #define SIGCHLD          17             //子进程停止或被终止
26.  #define SIGCONT          18             //恢复进程继续执行
```

```
27.   #define SIGSTOP          19        //停止进程的执行
28.    #define SIGTSTP         20        // tty 发出停止进程，可忽略
29.    #define SIGTTIN         21        //后台进程请求输入
30.    #define SIGTTOU         22        //后台进程请求输出
31.  #define SIGIO             23        //描述符上可以进行 I/O
32.   #define SIGPOLL          SIGIO     //可查询的事件发生
33.   #define SIGXCPU          24        //CPU 时限超时
34.   #define SIGXFSZ          25        //文件长度过长
35.   #define SIGVTALRM        26        //虚拟计时器到时
36.   #define SIGPROF          27        //统计分布图计时器到时
37.  */
38.    #define SIGWINCH         28        //窗口大小改变
39.  /*
40.    #define SIGLOST          29        //文件锁丢失
41.  */
42.   #define SA_NOCLDSTOP 1              //当子进程处于停止状态，就不对 SIGCHLD 处理
43.   #define SA_INTERRUPT  0x20000000
44.   #define SA_NOMASK     0x40000000    //不阻止在指定的信号处理程序（信号句柄）中再
                                          收到该信号
45.   #define SA_ONESHOT    0x80000000    //信号句柄一旦被调用过就恢复到默认处理句柄
46.   #define SIG_BLOCK          0
      // 在阻塞信号集中加上给定的信号集
47.    #define SIG_UNBLOCK       1
      // 从阻塞信号集中删除指定的信号集
48.   #define SIG_SETMASK        2
      // 设置阻塞信号集（信号屏蔽码）
49.   #define SIG_DFL            ((void (*)(int))0)
      // 默认的信号处理程序（信号句柄）
50.    #define SIG_IGN           ((void (*)(int))1)
      // 忽略信号的处理程序
51.    #define SIG_ERR           ((void (*)(int))-1)
      //从信号中返回错误
52.  #ifdef notdef
53.  #define sigemptyset(mask) ((*(mask) = 0), 1)
54.  #define sigfillset(mask) ((*(mask) = ~0), 1)
55.  #endif
56.  struct sigaction {
57.  void (*sa_handler)(int);// 对应某信号指定要采取的行动
58.  sigset_t sa_mask;//给出了对信号的屏蔽码，在信号程序执行时将阻塞对这些信号的处理
59.  int sa_flags;// 指定改变信号处理过程的信号集
60.  void (*sa_restorer)(void);// 是恢复函数指针
61.  };
     //该函数用于为信号_sig 安装一新的信号处理程序（信号句柄）
62.   void (*signal(int _sig, void (*_func)(int)))(int);
      //用于向当前进程自身发送信号。其作用等价于 kill(getpid(),sig)
63.   int raise(int sig);
      //kill()用于向任何进程或进程组发送任何信号
64.   int kill(pid_t pid, int sig);
      // sigaddset()和 sigdelset()用于对信号集中的信号进行增、删修改。sigaddset()用于向 mask 指
      //向的信号集中增加指定的信号 signo，sigdelset 则反之
```

```
65.  int sigaddset(sigset_t *mask, int signo);
66.  int sigdelset(sigset_t *mask, int signo);
     //用于清空屏蔽的所有信号，也即响应所有的信号
67.  int sigemptyset(sigset_t *mask);
     //向信号集中置入所有信号，即屏蔽所有信号
68.  int sigfillset(sigset_t *mask);
     //用于测试一个指定信号是否在信号集中（1——是，0——不是，-1——出错）
69.  int sigismember(sigset_t *mask, int signo); /* 1 - is, 0 - not, -1 error */
     //对 set 中的信号进行检测，看是否有挂起的信号。在 set 中返回进程中当前被阻塞的信号集
70.  int sigpending(sigset_t *set);
     //下面函数用于改变进程目前被阻塞的信号集（信号屏蔽码）。若 oldset 不是 NULL，则通过
     //其返回进程当前屏蔽信号集
71.  int sigprocmask(int how, sigset_t *set, sigset_t *oldset);
72.  int sigsuspend(sigset_t *sigmask);
     // sigaction()用于改变进程在收到指定信号时所采取的行动
73.  int sigaction(int sig, struct sigaction *act, struct sigaction *oldact);
74.  #endif
```

第 72 行函数用 sigmask 临时替换进程的信号屏蔽码，然后暂停该进程直到收到一个信号。若捕捉到某一信号并从该信号处理程序中返回，则该函数也返回，并且信号屏蔽码会恢复到调用前的值。

3.8 stdarg.h 文件

stdarg.h 文件以宏的形式定义变量参数列表。当调用某个函数，而函数参数又不确定时，就需要使用该文件中定义的宏。该文件主要包含了一个类型 va_list 和三个宏 va_start, va_arg 和 va_end。stdarg.h 文件的详细代码注释如下：

```
1.   #ifndef _STDARG_H
2.   #define _STDARG_H
     // 定义 va_list 是一个字符指针类型
3.   typedef char *va_list;
     //定义了取整后的 TYPE 类型的字节长度值
4.   #define __va_rounded_size(TYPE)  \
5.   (((sizeof (TYPE) + sizeof (int) - 1) / sizeof (int)) * sizeof (int))
6.   #ifndef __sparc__
     //使 AP 指向传给函数的可变参数表的第一个参数。在第一次调用 va_arg 或 va_end
     //之前，必须首先调用该函数
7.   #define va_start(AP, LASTARG)
8.   (AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
9.   #else
     // builtin_saveregs ()用于保存寄存器
10.   #define va_start(AP, LASTARG)
11.   (__builtin_saveregs (),
12.  AP = ((char *) &(LASTARG) + __va_rounded_size (LASTARG)))
13.  #endif
     // 下面该宏用于被调用函数完成一次正常返回。va_end 可以修改 AP 使其在重新调
     //用 va_start 之前不能被使用。va_end 必须在 va_arg 读完所有的参数后再被调用
14.   void va_end (va_list);
15.  /* 在 gnulib 中定义*/
```

```
16.  #define va_end(AP)
17.  #define va_arg(AP, TYPE)
18.  (AP += __va_rounded_size (TYPE),
19.  *((TYPE *) (AP - __va_rounded_size (TYPE))))
20.  #endif
```

第 17 到 20 行定义的宏用于扩展表达式，使其与下一个被传递参数具有相同的类型和值。对于缺省值，va_arg 可以用字符、无符号字符和浮点类型。第一次使用 va_arg 时，它返回表中的第一个参数，后续的每次调用都将返回表中的下一个参数。这是通过先访问 AP，然后把它增加以指向下一项来实现的。va_arg 使用 TYPE 来完成访问和定位下一项，每调用一次 va_arg，它就修改 AP 以指示表中的下一参数。

3.9 stddef.h 文件

stddef.h 文件主要用来声明一些特殊的函数。该文件里包含了标准库的一些常用定义，无论包含哪个标准头文件，stddef.h 文件都会被自动包含进来。

stddef.h 文件主要用来声明一些不与其他头文件类型相关的各种类型。stddef.h 文件的详细代码注释如下：


```
1.  #ifndef _STDDEF_H
2.  #define _STDDEF_H
3.  #ifndef _PTRDIFF_T
4.  #define _PTRDIFF_T
5.  typedef long ptrdiff_t;              //两个指针相减结果的类型
6.  #endif
7.  #ifndef _SIZE_T
8.  #define _SIZE_T
9.  typedef unsigned long size_t;        //sizeof 返回的类型
10. #endif
11. #undef NULL
12. #define NULL ((void *)0)            //空指针
13. //成员在类型中的偏移位置
14. #define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
15. #endif
```

这个文件里定义以下类型：

类型 size_t（sizeof 运算符的结果类型，是某个无符号整型）；

类型 ptrdiff_t（两个指针相减运算的结果类型，是某个有符号整型）；

符号常量 NULL（空指针值）；

其中有些定义也出现在其他头文件里（如 NULL）。

3.10 string.h 文件

该头文件中以内嵌函数的形式定义了所有字符串操作函数。所以在内核中要使用到字符串操作的文件中都需要包含该头文件。该文件中只包含相关函数的声明，具体的函数实现在标准的 C 库中。string.h 文件的详细代码注释如下：

```
1.  #ifndef _STRING_H_
2.  #define _STRING_H_
```

```
#ifndef NULL
#define NULL ((void *) 0)
#endif
#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned int size_t;
#endif
extern char * strerror(int errno);
 //将一个字符串（src）拷贝到另一个字符串（dest），直到遇到 NULL 字符后停止
extern inline char * strcpy(char * dest,const char *src)
 {
   __asm__("cld\n"                    //清方向位
      "1:\tlodsb\n\t"                 //加载 DS:[esi]处 1 字节 到 al，并更新 esi
      "stosb\n\t"                     //存储字节 al 到 ES:[edi]，并更新 edi
      "testb %%al,%%al\n\t"           //判断刚存储的字节是否为 0
      "jne 1b"                        //不是则跳转到标号 1 处，否则结束
 ::"S" (src),"D" (dest):"si","di","ax");
 return dest;                         //返回目的字符串指针
 }
 //拷贝源字符串 count 个字节到目的字符串。如果源串长度小于 count 个字节，就附加空字符
 //（NULL）到目的字符串
extern inline char * strncpy(char * dest,const char *src,size_t count)
{
   __asm__("cld\n"                    //清方向位
      "1:\tdecl %2\n\t"               //寄存器 ecx--
      "js 2f\n\t"                     //如果 count<0 则向前跳转到标号 2，结束
      "lodsb\n\t"                     //取 ds:[esi]处 1 字节到 al，并且 esi++
      "stosb\n\t"                     //存储该字节到 es:[edi]，并且 edi++
      "testb %%al,%%al\n\t"           //测试该字节是否为 0
      "jne 1b\n\t"                    //不是，则向前跳转到标号 1 处继续拷贝
      "rep\n\t"                       //否则，在目的串中存放剩余个数的空字符
 "stosb\n"
 "2:"
 ::"S" (src),"D" (dest),"c" (count):"si","di","ax","cx");
 return dest;                         //返回目的字符串指针
 }
 // 将源字符串拷贝到目的字符串的末尾处
extern inline char * strcat(char * dest,const char * src)
{
   __asm__("cld\n\t"                  //清方向位
  "repne\n\t"                         //比较 al 与 es:[edi]字节，并更新 edi++
      "scasb\n\t"                     //直到找到目的串中是 0 的字节，此时 edi 已指向后 1 字节
      "decl %1\n"                     //让 es:[edi]指向 0 值字节
      "1:\tlodsb\n\t"                 //取源字符串字节 ds:[esi] 到 al，并 esi++
      "stosb\n\t"                     //将该字节存到 es:[edi]，并 edi++
      "testb %%al,%%al\n\t"           //测试该字节是否为 0
      "jne 1b"                        //不是，则向后跳转到标号 1 处继续拷贝，否则结束
::"S" (src),"D" (dest),"a" (0),"c" (0xffffffff):"si","di","ax","cx");
   return dest;                       // 返回目的字符串指针
}
```

```
    //将源字符串的 count 个字节复制到目的字符串的末尾处，最后添加一空字符
49. extern inline char * strncat(char * dest,const char * src,size_t count)
50. {
51.     __asm__("cld\n\t"                    //清方向位
52.       "repne\n\t"                        //比较 al 与 es:[edi]字节，edi++
53.       "scasb\n\t"                        //直到找到目的串的末端 0 值字节
54.       "decl %1\n\t"                      //edi 指向该 0 值字节
55.       "movl %4,%3\n"                     //欲复制字节数 ecx
56.       "1:\tdecl %3\n\t"                  //ecx--
57.       "js 2f\n\t" //ecx <0 ?,           //是则向前跳转到标号 2 处
58.       "lodsb\n\t"                        //否则取 ds:[esi]处的字节到 al，esi++
59.       "stosb\n\t"                        //存储到 es:[edi]处，edi++
60.       "testb %%al,%%al\n\t"              //测试该字节是否为 0
61.       "jne 1b\n"                         //不是则向后跳转到标号 1 处，继续复制
62.       "2:\txorl %2,%2\n\t"               //将 al 清零
63.       "stosb"                            //存到 es:[edi]处
64. ::"S" (src),"D" (dest),"a" (0),"c" (0xffffffff),"g" (count)
65. :"si","di","ax","cx");
66.     return dest;                         //返回目的字符串指针
67. }
68. //将一个字符串与另一个字符串进行比较
69. extern inline int strcmp(const char * cs,const char * ct)
70. {
71.     register int __res __asm__("ax"); // __res 是寄存器变量（eax）
72.     __asm__("cld\n"                      //清方向位
73.       "1:\tlodsb\n\t"                    //取源字符串字节 ds:[esi] 到 al，并 esi++
74.       "scasb\n\t"                        //al 与字符串 1 的字节 es:[edi]作比较，并且 edi++。
75.       "jne 2f\n\t"                       //如果不相等，则向前跳转到标号 2
76.       "testb %%al,%%al\n\t"              //测试该字节是否为 0
77.       "jne 1b\n\t"                       //不是，则向后跳转到标号 1，继续比较
78.       "xorl %%eax,%%eax\n\t"             //是，则返回值 eax 清零
79.       "jmp 3f\n"                         //向前跳转到标号 3，结束
80.       "2:\tmovl $1,%%eax\n\t"            //eax 中置 1
81.       "jl 3f\n\t"                        //若前面比较中串 2 字符<串 1 字符，则返回正值，结束
82.       "negl %%eax\n"                     //否则 eax = -eax，返回负值，结束
83. "3:"
84. :"=a" (__res):"D" (cs),"S" (ct):"si","di");
85.     return __res;                        //返回比较结果
86. }
87. //将一个字符串与另一个字符串前 count 个字符进行比较
88. extern inline int strncmp(const char * cs,const char * ct,size_t count)
89. {
90.     register int __res __asm__("ax");// __res 是寄存器变量（eax）
91.     __asm__("cld\n"                      //清方向位
92.       "1:\tdecl %3\n\t"                  // count--
93.       "js 2f\n\t"                        //如果不相等，则向前跳转到标号 2
94.       "lodsb\n\t"                        //取串 2 的字符 ds:[esi] 到 al，并且 esi++
95.       "scasb\n\t"                        //比较 al 与串 1 的字符 es:[edi]，并且 edi++
96.       "jne 3f\n\t"                       //如果不相等，则向前跳转到标号 3
97.       "testb %%al,%%al\n\t"              //测试该字节是否为 0
```

```
98.         "jne 1b\n"                          //不是，则向后跳转到标号 1，继续比较
99.         "2:\txorl %%eax,%%eax\n\t"          //是 NULL 字符，则 eax 清零（返回值）
100.        "jmp 4f\n"                          //向前跳转到标号 4，结束
101.        "3:\tmovl $1,%%eax\n\t"             // eax 中置 1
102.        "jl 4f\n\t"                         //如果前面比较中串 2 字符<串 2 字符，则返回 1，结束
103.        "negl %%eax\n"                      //否则 eax = -eax，返回负值，结束
104.    "4:"
105.    :"=a" (__res):"D" (cs),"S" (ct),"c" (count):"si","di","cx");
106.        return __res;                       //返回比较结果
107.    }
108.    // 在字符串中寻找第一个匹配的字符
109.    extern inline char * strchr(const char * s,char c)
110.    {
111.     register char * __res __asm__("ax");  // __res 是寄存器变量（eax）
112.     __asm__("cld\n\t"                      //清方向位
113.         "movb %%al,%%ah\n"                 //将欲比较字符移到 ah
114.         "1:\tlodsb\n\t"                    //取源字符串字节 ds:[esi] 到 al，并 esi++
115.         "cmpb %%ah,%%al\n\t"               //字符串中字符 al 与指定字符 ah 相比较
116.         "je 2f\n\t"                        //若相等，则向前跳转到标号 2 处
117.         "testb %%al,%%al\n\t"              //测试该字节是否为 0
118.         "jne 1b\n\t"                       //若不是，则向后跳转到标号 1，继续比较
119.         "movl $1,%1\n"                     //是，则说明没有找到匹配字符，esi 置 1
120.         "2:\tmovl %1,%0\n\t"               //将指向匹配字符后一个字节处的指针值放入 eax
121.         "decl %0"                          //将指针调整为指向匹配的字符
122.    :"=a" (__res):"S" (s),"0" (c):"si");
123.        return __res;                       //返回指针
124.    }
        //找字符串中指定字符最后一次出现的地方，若找到返字符串中最后一次出现匹配字符的指针，
        //否则返空指针
125.    extern inline char * strrchr(const char * s,char c)
126.    {
127.    register char * __res __asm__("dx");   //__res 是寄存器变量（edx）
128.        __asm__("cld\n\t"                   //清方向位
129.         "movb %%al,%%ah\n"                 //将欲比较字符移到 ah
130.         "1:\tlodsb\n\t"                    //取源字符串字节 ds:[esi] 到 al，并 esi++
131.         "cmpb %%ah,%%al\n\t"               //字符串中字符 al 与指定字符 ah 相比较
132.         "jne 2f\n\t"                       //若相等，则向前跳转到标号 2 处
133.    "movl %%esi,%0\n\t"                     // 将字符指针保存到 edx 中
134.         "decl %0\n"                        //将指针调整为指向匹配的字符
135.         "2:\ttestb %%al,%%al\n\t"          //测试该字节是否为 0
136.         "jne 1b"                           // 若不是，则向后跳转到标号 1，继续比较
137.    :"=d" (__res):"0" (0),"S" (s),"a" (c):"ax","si");
138.        return __res;                       //返回指针
139.    }
        //在 cs 指向的字符串 1 中寻找第 1 个字符序列，
        //该字符序列中的任何字符都包含在 ct 指向的字符串 2 中
140.    extern inline size_t strspn(const char * cs, const char * ct)
141.    {
142.     register char * __res __asm__("si");  //__res 是寄存器变量（esi）
143.     __asm__("cld\n\t"                      //清方向位
```

```
144.        "movl %4,%%edi\n\t"                    //首先计算串 2 的长度。串 2 指针放入 edi 中
145.        "repne\n\t"                            //比较 al(0)与串 2 中的字符（es:[edi]），并 edi++
146.        "scasb\n\t"                            //如果不相等就继续比较（ecx 逐步递减）
147.        "notl %%ecx\n\t"                       //ecx 中每位取反
148.        "decl %%ecx\n\t"                       // ecx--，得串 2 的长度值
149.        "movl %%ecx,%%edx\n"                   //将串 2 的长度值暂放入 edx 中
150.        "1:\tlodsb\n\t"                        //取源字符串字节 ds:[esi] 到 al，并 esi++
151.        "testb %%al,%%al\n\t"                  //测试该字节是否为 0
152.        "je 2f\n\t"                            //若相等，则向前跳转到标号 2 处
153.        "movl %4,%%edi\n\t"                    //取串 2 头指针放入 edi 中
154.        "movl %%edx,%%ecx\n\t"                 //再将串 2 的长度值放入 ecx 中
155.        "repne\n\t"                            //比较 al 与串 2 中字符 es:[edi]，并且 edi++
156.        "scasb\n\t"                            //如果不相等就继续比较
157.        "je 1b\n"                              //如果相等，则向后跳转到标号 1 处
158.        "2:\tdecl %0"                          // esi--，指向最后一个包含在串 2 中的字符
159.    :"=S" (__res):"a" (0),"c" (0xffffffff),"0" (cs),"g" (ct)
160.    :"ax","cx","dx","di");
161.        return __res-cs;                       //返回字符序列的长度值
162.    }
        //在 cs 指向的字符串 1 中寻找不包含在 ct 指向的字符串 2 中的任何字符的首个字符序列
163.    extern inline size_t strcspn(const char * cs, const char * ct)
164.    {
165.        register char * __res __asm__("si");   //__res 是寄存器变量（esi）
166.    __asm__("cld\n\t"                          //清方向位
167.        "movl %4,%%edi\n\t"                    //首先计算串 2 的长度。串 2 指针放入 edi 中
168.        "repne\n\t"                            //比较 al 与串 2 中字符 es:[edi]，并且 edi++
169.        "scasb\n\t"                            //如果不相等就继续比较
170.        "notl %%ecx\n\t"                       //ecx 中每位取反
171.        "decl %%ecx\n\t"                       // ecx--，得串 2 的长度值
172.        "movl %%ecx,%%edx\n"                   //串 2 指针放入 edi 中
173.        "1:\tlodsb\n\t"                        //取源字符串字节 ds:[esi] 到 al，并 esi++
174.        "testb %%al,%%al\n\t"                  //测试该字节是否为 0
175.        "je 2f\n\t"                            //若相等，则向前跳转到标号 2 处
176.        "movl %4,%%edi\n\t"                    //取串 2 头指针放入 edi 中
177.        "movl %%edx,%%ecx\n\t"                 // 再将串 2 的长度值放入 ecx 中
178.        "repne\n\t"                            //比较 al 与串 2 中字符 es:[edi]，并且 edi++
179.        "scasb\n\t"                            //如果不相等就继续比较
180.        "jne 1b\n"                             //如果相等，则向后跳转到标号 1 处
181.        "2:\tdecl %0"                          // esi--，指向最后一个包含在串 2 中的字符
182.    :"=S" (__res):"a" (0),"c" (0xffffffff),"0" (cs),"g" (ct)
183.    :"ax","cx","dx","di");
         //返回字符序列的长度值
184.    return __res-cs;
185.    }
        //在 cs 指向的字符串 1 中寻找首个包含在 ct 指向的字符串 2 中的任何字符
186.    extern inline char * strpbrk(const char * cs,const char * ct)
187.    {
188.    register char * __res __asm__("si");       //__res 是寄存器变量（esi）
189.    __asm__("cld\n\t"                          //清方向位
190.        "movl %4,%%edi\n\t"                    //首先计算串 2 的长度。串 2 指针放入 edi 中
```

```
191.        "repne\n\t"                          //比较 al 与串 2 中字符 es:[edi]，并且 edi++
192.        "scasb\n\t"                          //如果不相等就继续比较
193.        "notl %%ecx\n\t"                     //ecx 中每位取反
194.        "decl %%ecx\n\t"                     // ecx--，得串 2 的长度值
195.        "movl %%ecx,%%edx\n"                 //串 2 的长度值放入 edx 中
196.        "1:\tlodsb\n\t"                      //取 x 源字符串字节 ds:[esi] 到 al，并 esi++
197.        "testb %%al,%%al\n\t"                //测试该字节是否为 0
198.        "je 2f\n\t"                          //若相等，则向前跳转到标号 2 处
199.        "movl %4,%%edi\n\t"                  //取串 2 头指针放入 edi 中
200.        "movl %%edx,%%ecx\n\t"               //再将串 2 的长度值放入 ecx 中
201.        "repne\n\t"                          //比较 al 与串 2 中字符 es:[edi]，并且 edi++
202.        "scasb\n\t"                          //如果不相等就继续比较
203.        "jne 1b\n\t"                         //若不相等，则向后跳转到标号 1，继续比较
204.        "decl %0\n\t"                        //esi--，指向一个包含在串 2 中的字符
205.        "jmp 3f\n"                           //向前跳转到标号 3 处
206.        "2:\txorl %0,%0\n"                   //没有找到符合条件的，将返回值为 NULL
207.    "3:"
208.    :"=S" (__res):"a" (0),"c" (0xffffffff),"0" (cs),"g" (ct)
209.    :"ax","cx","dx","di");
210.        return __res;                        //返回指针值
211.    }
        //在字符串 1 中寻找首个匹配整个字符串 2 的字符串
212.    extern inline char * strstr(const char * cs,const char * ct)
213.    {
214.    register char * __res __asm__("ax");// __res 是寄存器变量（eax）
215.        __asm__("cld\n\t" \                  //清方向位
216.            "movl %4,%%edi\n\t"              //首先计算串 2 的长度。串 2 指针放入 edi 中
217.        "repne\n\t"                          //比较 al 与串 2 中字符 es:[edi]，并且 edi++
218.        "scasb\n\t"                          //如果不相等就继续比较
219.        "notl %%ecx\n\t"                     //ecx 中每位取反
220.        "decl %%ecx\n\t"
            /* 注意！如果搜索串为空，将设置 Z 标志*/
221.        "movl %%ecx,%%edx\n"                 //取串 2 长度值放入 edx 中
222.        "1:\tmovl %4,%%edi\n\t"              //取串 2 头指针放入 edi 中
223.        "movl %%esi,%%eax\n\t"               //取串 1 头指针放入 eax 中
224.        "movl %%edx,%%ecx\n\t"               //再将串 2 的长度值放入 ecx 中
225.        "repe\n\t"                           //比较串 1 和串 2 字符（ds:[esi],es:[edi]），esi++,edi++
226.        "cmpsb\n\t"                          //若对应字符相等就一直比较下去
227.        "je 2f\n\t"
            /* 对空串同样有效，见上面*/
228.        "xchgl %%eax,%%esi\n\t"              //串 1 头指针放到 esi，比较结果的串 1 指针到放 eax
229.        "incl %%esi\n\t"                     //串 1 头指针指向下一个字符
230.        "cmpb $0,-1(%%eax)\n\t"              //串 1 指针（eax-1）所指字节是否为 0
231.        "jne 1b\n\t"                         //不是则跳转到标号 1，继续从串 1 的第 2 个字符开始比较
232.        "xorl %%eax,%%eax\n\t"               //清 eax，表示没有找到匹配
233.    "2:"
234.    :"=a" (__res):"0" (0),"c" (0xffffffff),"S" (cs),"g" (ct)
235.    :"cx","dx","di","si");
236.        return __res;                        //返回比较结果
237.    }
```

```
        //计算字符串长度
238.    extern inline size_t strlen(const char * s)
239.    {
240.        register int __res __asm__("cx");   //__res 是寄存器变量（ecx）
241.        __asm__("cld\n\t"                   //清方向位
242.        "repne\n\t"                         //al(0)与字符串中字符 es:[edi]比较
243.        "scasb\n\t"                         //若不相等就一直比较
244.        "notl %0\n\t"                       //ecx 取反
245.        "decl %0"                           //ecx--，得字符串得长度值
246.        :"=c" (__res):"D" (s),"a" (0),"0" (0xffffffff):"di");
247.        return __res;                       //返回字符串长度值
248.    }
249.    //用于临时存放指向下面被分析字符串 1(s)的指针
250.    extern char * ___strtok;
251.    extern inline char * strtok(char * s,const char * ct)
252.    {
253.        register char * __res __asm__("si"); //__res 是寄存器变量（esi）
254.        __asm__("testl %1,%1\n\t"           //首先测试 esi 是否是 NULL
255.        "jne 1f\n\t"                        //如果不是，则表明是首次调用本函数，跳转标号 1
256.        "testl %0,%0\n\t"                   //如果是 NULL，则表示此次是后续调用 ebx(__strtok)
257.        "je 8f\n\t"                         //如果 ebx 指针是 NULL，则不能处理，跳转结束
258.        "movl %0,%1\n"                      //将 ebx 指针复制到 esi
259.        "1:\txorl %0,%0\n\t"                //清 ebx 指针
260.        "movl $-1,%%ecx\n\t"                //置 ecx = 0xffffffff
261.        "xorl %%eax,%%eax\n\t"              //清零 eax
262.        "cld\n\t"                           //清方向位
263.        "movl %4,%%edi\n\t"                 //求字符串 2 的长度。edi 指向字符串 2
264.        "repne\n\t"                         //将 al(0)与 es:[edi]比较，并且 edi++
265.        "scasb\n\t"                         //直到找到字符串 2 的结束 null 字符，或计数 ecx= =0
266.        "notl %%ecx\n\t"                    //将 ecx 取反
267.        "decl %%ecx\n\t"                    //ecx--，得到字符串 2 的长度值
268.        "je 7f\n\t"
            /* 分割符字符串空*/
269.        "movl %%ecx,%%edx\n"                //将串 2 长度暂存入 edx
270.        "2:\tlodsb\n\t"                     //取串 1 的字符 ds:[esi]到 al，并且 esi++
271.        "testb %%al,%%al\n\t"               //判断该字符是否为 0
272.        "je 7f\n\t"                         //如果是，则跳转标号 7
273.        "movl %4,%%edi\n\t"//edi            //再次指向串 2 首
274.        "movl %%edx,%%ecx\n\t"              //取串 2 的长度值置入计数器 ecx
275.        "repne\n\t"                         //将 al 中串 1 的字符与串 2 中所有字符比较
276.        "scasb\n\t"                         //判断该字符是否为分割符
277.        "je 2b\n\t"                         //若能在串 2 中找到相同字符（分割符），则跳转标号 2
278.        "decl %1\n\t"                       //若不是分割符，则串 1 指针 esi 指向此时的该字符
279.        "cmpb $0,(%1)\n\t"                  //该字符是否是 NULL
280.        "je 7f\n\t"                         //若是，则跳转标号 7 处
281.        "movl %1,%0\n"                      //将该字符的指针 esi 存放在 ebx
282.        "3:\tlodsb\n\t"                     //取串 1 下一个字符 ds:[esi]到 al，并且 esi++
283.        "testb %%al,%%al\n\t"               //该字符是否是 NULL
284.        "je 5f\n\t"                         //若是，表示串 1 结束，跳转到标号 5
285.        "movl %4,%%edi\n\t"edi              //再次指向串 2 首
```

```
286.        "movl %%edx,%%ecx\n\t"          //串 2 长度值置入计数器 ecx
287.        "repne\n\t"                     //将 al 中串 1 的字符与串 2 中每个字符比较
288.        "scasb\n\t"                     //测试 al 字符是否是分割符
289.        "jne 3b\n\t"                    //若不是分割符则跳转标号 3，检测串 1 中下一个字符
290.        "decl %1\n\t"                   //若是分割符，则 esi--，指向该分割符字符
291.        "cmpb $0,(%1)\n\t"              //该分割符是否是 NULL
292.        "je 5f\n\t"                     //若是，则跳转到标号 5
293.        "movb $0,(%1)\n\t"              //若不是，则将该分割符用 NULL 字符替换掉
294.        "incl %1\n\t"esi                //指向串 1 中下一个字符，也即剩余串首
295.        "jmp 6f\n"                      //跳转标号 6 处
296.        "5:\txorl %1,%1\n"esi           //清零
297.        "6:\tcmpb $0,(%0)\n\t"          // ebx 指针是否指向 NULL
298.        "jne 7f\n\t"                    //若不是，则跳转标号 7
299.        "xorl %0,%0\n"                  //若是，则让 ebx=NULL
300.        "7:\ttestl %0,%0\n\t"           // ebx 指针是否为 NULL
301.        "jne 8f\n\t"                    //若不是则跳转 8，结束汇编代码
302.        "movl %0,%1\n"                  //将 esi 置为 NULL
303.    "8:"
304.    #if __GNUC__ == 2
305.    :"=r" (__res)
306.    #else
307.    :"=b" (__res)
308.    #endif
309.    ,"=S" (___strtok)
310.    :"0" (___strtok),"1" (s),"g" (ct)
311.    :"ax","cx","dx","di");
        // 返回指向新 token 的指针
312.    return __res;
313.    }
        // 内存块复制。从源地址 src 处开始复制 n 个字节到目的地址 dest 处
314.    extern inline void * memcpy(void * dest,const void * src, size_t n)
315.    {
316.        __asm__("cld\n\t"                   //清方向位
317.        "rep\n\t"                           //重复执行复制 ecx 个字节
318.    "movsb"                                 //从 ds:[esi]到 es:[edi]，esi++，edi++
319.    ::"c" (n),"S" (src),"D" (dest)
320.    :"cx","si","di");
321.        return dest;                        //返回目的地址
322.    }
        // 内存块移动。同内存块复制，但考虑移动的方向
323.    extern inline void * memmove(void * dest,const void * src, size_t n)
324.    {
325.    if (dest<src)
326.        __asm__("cld\n\t"                   //清方向位
327.        "rep\n\t"                           //从 ds:[esi]到 es:[edi]，并且 esi++，edi++
328.        "movsb"                             //重复执行复制 ecx 字节
329.    ::"c" (n),"S" (src),"D" (dest)
330.    :"cx","si","di");
331.    else
332.        __asm__("std\n\t"                   //清方向位，从末端开始复制
```

```
333.    "rep\n\t"                                   // 从 ds:[esi]到 es:[edi]，并且 esi--，edi--
334.        "movsb\n\t"                             //复制 ecx 个字节
335.    "cld"
336.    ::"c" (n),"S" (src+n-1),"D" (dest+n-1)
337.    :"cx","si","di");
338.    return dest;
339.    }
         //比较 n 个字节的两块内存（两个字符串），即使遇上 NULL 字节也不停止比较
340.    extern inline int memcmp(const void * cs,const void * ct,size_t count)
341.    {
342.        register int __res __asm__("ax");       //__res 是寄存器变量（eax）
343.        __asm__("cld\n\t"                       //清方向位
344.         "repe\n\t"                             //如果相等则重复
345.         "cmpsb\n\t"                            //比较 ds:[esi]与 es:[edi]的内容，并且 esi++，edi++
346.         "je 1f\n\t"                            //如果都相同，则跳转到标号 1，返回 0(eax)值
347.         "movl $1,%%eax\n\t"                    //否则 eax 置 1
348.         "jl 1f\n\t"                            //若内存块 2 内容的值<内存块 1，则跳转标号 1
349.         "negl %%eax\n"                         //否则 eax = -eax
350.    "1:"
351.    :"=a" (__res):"0" (0),"D" (cs),"S" (ct),"c" (count)
352.    :"si","di","cx");
353.        return __res;                           //返回比较结果
354.    }
         //在 n 字节大小的内存块（字符串）中寻找指定字符
355.    extern inline void * memchr(const void * cs,char c,size_t count)
356.    {
357.        register void * __res __asm__("di");    //__res 是寄存器变量(edi)
358.    if (!count)
359.         return NULL;                           //如果内存块长度==0，则返回 NULL，没有找到
360.        __asm__("cld\n\t"                       //清方向位
361.         "repne\n\t"                            //如果不相等则重复执行下面语句
362.         "scasb\n\t"                            //al 中字符与 es:[edi]字符作比较，并且 edi++
363.         "je 1f\n\t"                            //如果相等则向前跳转到标号 1 处
364.         "movl $1,%0\n"                         //否则 edi 中置 1
365.         "1:\tdecl %0"                          //让 edi 指向找到的字符（或是 NULL）
366.    :"=D" (__res):"a" (c),"D" (cs),"c" (count)
367.    :"cx");
368.        return __res;                           //返回字符指针
369.    }
         // 用字符填写指定长度内存块
         // 用字符 c 填写 s 指向的内存区域，共填 count 字节
370.    extern inline void * memset(void * s,char c,size_t count)
371.    {
372.        __asm__("cld\n\t"                       //清方向位
373.         "rep\n\t"                              //重复执行 ecx 指定的次数
374.         "stosb"                                //将 al 中字符存入 es:[edi]中，并且 edi++
375.    ::"a" (c),"D" (s),"c" (count)
376.    :"cx","di");
377.    return s;
378.    }
379.    #endif
```

3.11 termios.h 文件

该文件含有终端 I/O 接口定义。包括 termios 数据结构和一些对通用终端接口设置的函数原型。

该头文件定义的函数用来读取或设置终端的属性、线路控制、读取或设置波特率以及读取或设置终端前端进程的组 id。termios.h 文件的详细代码注释如下：

```
1.  #ifndef _TERMIOS_H
2.  #define _TERMIOS_H
3.  #include <sys/types.h>
4.  #define TTY_BUF_SIZE //1024 tty 中的缓冲区长度
5.  /* 0x54 只是一个魔数，目的是为了使这些常数唯一('T') */
      // 下面名称 TC[*]的含义是 tty 控制命令
      // 取相应终端 termios 结构中的信息
6.  #define TCGETS          0x5401
      // 设置相应终端 termios 结构中的信息
7.  #define TCSETS          0x5402
      // 在设置终端 termios 的信息之前，需要先等待输出队列中所有数据处理完（耗尽）
      //对于修改参数会影响输出的情况，就需要使用这种形式
8.  #define TCSETSW         0x5403
      // 在设置 termios 的信息之前，需要先等待输出队列中所有数据处理完，并且刷新（清空）输
      //入队列再设置
9.  #define TCSETSF         0x5404
      // 取相应终端 termio 结构中的信息
10. #define TCGETA          0x5405
    // 设置相应终端 termio 结构中的信息
11. #define TCSETA          0x5406
    // 在设置终端 termio 的信息之前，需要先等待输出队列中所有数据处理完（耗尽）
12. #define TCSETAW         0x5407
    // 在设置 termio 的信息之前，需要先等待输出队列中所有数据处理完，并且刷新（清空）
    //输入队列
13. #define TCSETAF         0x5408
    // 等待输出队列处理完毕（空），如果参数值是 0，则发送一个 break
14. #define TCSBRK          0x5409
    // 开始/停止控制。如果参数值是 0，则挂起输出；如果是 1，则重新开启挂起的输出；如果是 2，
    //则挂起输入；如果是 3，则重新开启挂起的输入
15. #define TCXONC          0x540A
    //刷新已写输出但还没发送或已收但还没有读数据。如果参数是 0，则刷新（清空）输入队列；
    //如果是 1，则刷新输出队列；如果是 2，则刷新输入和输出队列
16. #define TCFLSH          0x540B
    // 下面名称 TIOC[*]的含义是 tty 输入输出控制命令
    // 设置终端串行线路专用模式
17. #define TIOCEXCL        0x540C
    // 复位终端串行线路专用模式
18. #define TIOCNXCL        0x540D
    // 设置 tty 为控制终端
19. #define TIOCSCTTY       0x540E
    // 读取指定终端设备进程的组 id
20. #define TIOCGPGRP       0x540F
    // 设置指定终端设备进程的组 id
```

```
21. #define TIOCSPGRP       0x5410
    // 返回输出队列中还未送出的字符数
22. #define TIOCOUTQ        0x5411
    // 模拟终端输入。该命令以一个指向字符的指针作为参数，并假装该字符是在终端上输入的
    //用户必须在该控制终端上具有超级用户权限或具有读许可权限
23.  #define TIOCSTI        0x5412
    // 读取终端设备窗口大小信息
24.  #define TIOCGWINSZ     0x5413
    // 设置终端设备窗口大小信息
25.  #define TIOCSWINSZ     0x5414
    // 返回 modem 状态控制引线的当前状态比特位标志集
26.  #define TIOCMGET       0x5415
    // 设置单个 modem 状态控制引线的状态（true 或 false）
27.  #define TIOCMBIS       0x5416
    // 复位单个 modem 状态控制引线的状态（true 或 false）
28.  #define TIOCMBIC       0x5417
    // 设置 modem 状态引线的状态。如果某一比特位置位，则 modem 对应的状态引线将置
    //为有效
29.  #define TIOCMSET       0x5418
    // 读取软件载波检测标志（1-开启；0-关闭）
    // 对于本地连接的终端或其他设备，软件载波标志是开启的，对于使用 modem 线路的终端
    // 或设备则是关闭的
30.  #define TIOCGSOFTCAR   0x5419
    // 设置软件载波检测标志（1-开启；0-关闭）
31.  #define TIOCSSOFTCAR   0x541A
    // 返回输入队列中还未取走字符的数目
32.  #define FIONREAD       0x541B
33.  #define TIOCINQ        FIONREAD
34.  #define TIOCLINUX      0x541C
     // 窗口大小（Window size）属性结构。在窗口环境中可用于基于屏幕的应用程序
35.  struct winsize {
36.  unsigned short ws_rcw;          //窗口字符行数
37.  unsigned short ws_col;          //窗口字符列数
38.  unsigned short ws_xpixel;       //窗口宽度，像素值
39.  unsigned short ws_ypixel;       // 窗口高度，像素值
40. };
     // AT&T 系统 V 的 termio 结构
     //termio 结构中控制字符数组的长度
41.  #define NCC 8
42. struct termio {
43.  unsigned short c_iflag;         /* 输入模式标志*/
44.  unsigned short c_oflag;         /* 输出模式标志*/
45.  unsigned short c_cflag;         /* 控制模式标志*/
46.  unsigned short c_lflag;         /* 本地模式标志*/
47.  unsigned char c_line;           /* 线路规程*/
48.   unsigned char c_cc[NCC];       /* 控制字符数组*/
49.  };
    // POSIX 的 termios 结构
    //termios 结构中控制字符数组的长度
50.  #define NCCS 17
```

```
51.  struct termios {
52.  tcflag_t c_iflag;                    /* 输入模式标志*/
53.  tcflag_t c_oflag;                    /* 输出模式标志*/
54.  tcflag_t c_cflag;                    /* 控制模式标志*/
55.  tcflag_t c_lflag;                    /* 本地模式标志*/
56.  cc_t c_line;                         /* 线路规程*/
57.   cc_t c_cc[NCCS];                    /* 控制字符数组*/
58.  };
59.  /* c_cc 数组中的字符*/
60.  // 以下是 c_cc 数组对应字符的索引值。
61.   #define VINTR 0                    // c_cc[VINTR] = INTR (^C)，\003，中断字符
62.   #define VQUIT 1                    // c_cc[VQUIT] = QUIT (^\)，\034，退出字符
63.   #define VERASE 2                   // c_cc[VERASE] = ERASE (^H)，\177，擦出字符
64.   #define VKILL 3                    //c_cc[VKILL] = KILL (^U)，\025，终止字符
65.   #define VEOF 4                     // c_cc[VEOF] = EOF (^D)，\004，文件结束字符
66.   #define VTIME 5                    //c_cc[VTIME] = TIME (\0)，\0，定时器值
67.   #define VMIN 6                     // c_cc[VMIN] = MIN (\1)，\1，定时器值
68.   #define VSWTC 7                    // c_cc[VSWTC] = SWTC (\0)，\0，交换字符
69.   #define VSTART 8                   // c_cc[VSTART] = START (^Q)，\021，开始字符
70.   #define VSTOP 9                    //c_cc[VSTOP] = STOP (^S)，\023，停止字符
71.   #define VSUSP 10                   //c_cc[VSUSP] = SUSP (^Z)，\032，挂起字符
72.   #define VEOL 11                    //c_cc[VEOL] = EOL (\0)，\0，行结束字符
73.   #define VREPRINT 12                //c_cc[VREPRINT] = REPRINT (^R)，\022，重显示字符
74.   #define VDISCARD 13                // c_cc[VDISCARD] = DISCARD (^O)，\017，丢弃字符
75.   #define VWERASE 14                 //c_cc[VWERASE] = WERASE (^W)，\027，单词擦除字符
76.   #define VLNEXT 15                  // c_cc[VLNEXT] = LNEXT (^V)，\026，下一行字符
77.   #define VEOL2 16                   //c_cc[VEOL2] = EOL2 (\0)，\0，行结束字符
78.    /* c_iflag 比特位*/
        // termios 结构输入模式字段 c_iflag 各种标志的符号常数
79.   #define IGNBRK      0000001        //输入时忽略 BREAK 条件
80.   #define BRKINT      0000002        //在 BREAK 时产生 SIGINT 信号
81.   #define IGNPAR      0000004        //忽略奇偶校验出错的字符
82.   #define PARMRK      0000010        //标记奇偶校验错
83.   #define INPCK 0000020              //允许输入奇偶校验
84.   #define ISTRIP       0000040       //屏蔽字符第 8 位
85.   #define INLCR        0000100       //输入时将换行符 NL 映射成回车符 CR
86.   #define IGNCR        0000200       //忽略回车符 CR
87.   #define ICRNL        0000400       //在输入时将回车符 CR 映射成换行符 NL
88.   #define IUCLC        0001000       //在输入时将大写字符转换成小写字符
89.   #define IXON         0002000       //允许开始/停止（XON/XOFF）输出控制
90.   #define IXANY       0004000        //允许任何字符重启输出
91.   #define IXOFF        0010000       //允许开始/停止（XON/XOFF）输入控制
92.   #define IMAXBEL     0020000        //输入队列满时响铃
93.    /* c_oflag 比特位*/
        // termios 结构中输出模式字段 c_oflag 各种标志的符号常数
94.    #define OPOST       0000001       //执行输出处理
95.    #define OLCUC       0000002       //在输出时将小写字符转换成大写字符
96.    #define ONLCR       0000004       //在输出时将换行符 NL 映射成回车-换行符 CR-NL
97.    #define OCRNL       0000010       //在输出时将回车符 CR 映射成换行符 NL
98.    #define ONOCR       0000020       //在 0 列不输出回车符 CR
```

```
99.   #define ONLRET    0000040     //换行符 NL 执行回车符的功能
100.  #define OFILL     0000100     //延迟时使用填充字符而不使用时间延迟
101.  #define OFDEL     0000200     //填充字符是 ASCII 码 DEL。如果未设置，则使用 ASCII NULL
102.  #define NLDLY     0000400     //选择换行延迟
103.  #define   NL      00000000    //换行延迟类型 0
104.  #define   NL      10000400    //换行延迟类型 1
105.  #define CRDLY     0003000     //选择回车延迟
106.  #define   CR0     0000000     //回车延迟类型 0
107.  #define   CR1     0001000     //回车延迟类型 1
108.  #define   CR2     0002000     //回车延迟类型 2
109.  #define   CR3     0003000     //回车延迟类型 3
110.  #define TABDLY    0014000     //选择水平制表延迟
111.  #define   TAB0    0000000     //水平制表延迟类型 0
112.  #define   TAB1    0004000     //水平制表延迟类型 1
113.  #define   TAB2    0010000     //水平制表延迟类型 2
114.  #define   TAB3    0014000     //水平制表延迟类型 3
115.  #define   XTABS   0014000     //将制表符 Tab 换成空格，该值表示空格数
116.  #define BSDLY     0020000     //选择退格延迟
117.  #define   BS0     0000000     //退格延迟类型 0
118.  #define   BS1     0020000     //退格延迟类型 1
119.  #define VTDLY     0040000     //纵向制表延迟
120.  #define   VT      00000000    //纵向制表延迟类型 0
121.  #define   VT      10040000    //纵向制表延迟类型 1
122.  #define FFDLY     0040000     //选择换页延迟
123.  #define   FF      00000000    //换页延迟类型 0
124.  #define   FF      10040000    //换页延迟类型 1
      /* c_cflag 比特位的含义*/
      // termios 结构中控制模式标志字段 c_cflag 标志的符号常数（8 进制数）
125.  #define CBAUD     0000017     //传输速率位屏蔽码
126.  #define   B0      0000000     /* 挂断线路*/
127.  #define   B50     0000001     //波特率 50
128.  #define   B75     0000002     //波特率 75
129.  #define   B110    0000003     //波特率 110
130.  #define   B134    0000004     //波特率 134
131.  #define   B150    0000005     //波特率 150
132.  #define   B200    0000006     //波特率 200
133.  #define   B300    0000007     //波特率 300
134.  #define   B600    0000010     //波特率 600
135.  #define   B1200   0000011     //波特率 1200
136.  #define   B1800   0000012     //波特率 1800
137.  #define   B2400   0000013     //波特率 2400
138.  #define   B4800   0000014     //波特率 4800
139.  #define   B9600   0000015     //波特率 9600
140.  #define   B19200  0000016     //波特率 19200
141.  #define   B38400  0000017     //波特率 38400
142.  #define   EXTA    B19200      //扩展波特率 A
143.  #define   EXTB    B38400      //扩展波特率 B
144.  #define   CSIZE   0000060     //字符位宽度屏蔽码
145.  #define    CS5    0000000     //每字符 5 比特位
146.  #define    CS6    0000020     //每字符 6 比特位
```

```
147.    #define     CS7       0000040          //每字符 7 比特位
148.    #define     CS8       0000060          //每字符 8 比特位
149.    #define CSTOPB        0000100          //设置两个停止位，而不是 1 个
150.    #define CREAD         0000200          //允许接收
151.    #define PARENB        0000400          //开启输出时产生奇偶位、输入时进行奇偶校验
152.    #define PARODD        0001000          //输入/输入校验是奇校验
153.    #define HUPCL         0002000          //最后进程关闭后挂断
154.    #define CLOCAL        0004000          //忽略调制解调器（modem）控制线路
155.    #define CIBAUD        03600000
156.    #define CRTSCTS       020000000000     /* 流控制*/
157.    /*c_lflag 比特位*/
        // termios 结构中本地模式标志字段 c_lflag 的符号常数
158.    #define ISIG          0000001          //当收到字符 INTR、QUIT、SUSP 或 DSUSP，产生相应
                                                 的信号
159.    #define ICANON        0000002          //开启规范模式（熟模式）
160.    #define XCASE         0000004          //若设置了 ICANON，则终端是大写字符
161.    #define ECHO          0000010          //回显输入字符
162.    #define ECHOE         0000020          //若设置了 ICANON，则 ERASE/WERASE 将擦除前一字
                                                 符/单词
163.    #define ECHOK         0000040          //若设置了 ICANON，则 KILL 字符将擦除当前行
164.    #define ECHONL        0000100          //如设置了 ICANON，则即使 ECHO 没有开启也回显 NL
                                                 字符
165.    #define NOFLSH        0000200          //当生成 SIGINT 和 SIGQUIT 信号时不刷新输入列，
                                               //当生成 SIGSUSP 信号时，刷新输入队列
166.    #define TOSTOP        0000400          //发送 SIGTTOU 信号到后台进程的进程组，该后台进程
                                                 试图写自己的控制终端
167.    #define ECHOCTL  0001000               //若设置了 ECHO，则除 TAB、NL、START 和 STOP 以外
                                               //的 ASCII 控制信号将被回显成^X 式样，X 值是控制符+0x40
168.     #define ECHOPRT      0002000          //若设置了 ICANON 和 IECHO，则字符在擦除时将显示
169.    #define ECHOKE        0004000          //若设置了 ICANON，则 KILL 通过擦除行上的所有字符被
                                                 回显
170.    #define FLUSHO        0010000          //输出被刷新。通过输入 DISCARD 字符，该标志被翻转
171.    #define PENDIN        0040000          //当下一个字符是读时，输入队列中的所有字符将被重显
172.    #define IEXTEN        0100000          //开启实现时定义的输入处理
        /* modem 线路信号符号常数*/
173.    #define TIOCM_LE      0x001            //线路允许
174.    #define TIOCM_DTR     0x002            //数据终端就绪
175.    #define TIOCM_RTS     0x004            //请求发送
176.    #define TIOCM_ST      0x008            //串行数据发送
177.    #define TIOCM_SR      0x010            //串行数据接收
178.    #define TIOCM_CTS     0x020            //清除发送
179.    #define TIOCM_CAR     0x040            //载波监测
180.    #define TIOCM_RNG     0x080            //响铃指示
181.    #define TIOCM_DSR     0x100            //数据设备就绪
182.    #define TIOCM_CDTIOCM_CAR
183.    #define TIOCM_RI  TIOCM_RNG
184.
185.    /* tcflow()和 TCXONC 使用这些符号常数*/
186.    #define   TCOOFF          0            //挂起输出
187.    #define   TCOON           1            //重启被挂起的输出
```

```
188.    #define  TCIOFF          2          //系统传输一个 STOP 字符，使设备停止向系统传输数据
189.    #define  TCION           3          //系统传输一个 START 字符，使设备开始向系统传输数据
190.     /*tcflush()和 TCFLSH 使用这些符号常数*/
191.    #define  TCIFLUSH        0          //清接收到的数据但不读
192.    #define  TCOFLUSH        1          //清已写的数据但不传送
193.    #define  TCIOFLUSH       2          //清接收到的数据但不读，清已写的数据但不传送
194.    /* tcsetattr()使用这些符号常数*/
195.    #define  TCSANOW         0          //改变立即发生
196.    #define  TCSADRAIN       1          //改变在所有已写的输出被传输之后发生
197.    #define  TCSAFLUSH       2          //改变在所有已写的输出被传输之后并且在所有接收到但
                                            //还没有读取的数据被丢弃之后发生
        //返回 termios_p 所指 termios 结构中的接收波特率
198.   extern speed_t cfgetispeed(struct termios *termios_p);
        //返回 termios_p 所指 termios 结构中的发送波特率
199.    extern speed_t cfgetospeed(struct termios *termios_p);
       //将 termios_p 所指 termios 结构中的接收波特率设置为 speed
200.   extern int cfsetispeed(struct termios *termios_p, speed_t speed);
       //将 termios_p 所指 termios 结构中的发送波特率设置为 speed
201.   extern int cfsetospeed(struct termios *termios_p, speed_t speed);
       //等待 fildes 所指对象已写输出数据被传送出去
202.   extern int tcdrain(int fildes);
       //挂起/重启 fildes 所指对象数据的接收和发送
203.   extern int tcflow(int fildes, int action);
       //丢弃 fildes 指定对象所有已写但还没传送以及所有已收到但还没有读取的数据
204.   extern int tcflush(int fildes, int queue_selector);
       //获取与句柄 fildes 对应对象的参数，并将其保存在 termios_p 所指的地方
205.   extern int tcgetattr(int fildes, struct termios *termios_p);
        //如果终端使用异步串行数据传输，则在一定时间内连续传输一系列 0 值比特位
206.   extern int tcsendbreak(int fildes, int duration);
        //使用 termios 结构指针 termios_p 所指的数据，设置与终端相关的参数
207.   extern int tcsetattr(int fildes, int optional_actions,
208.   struct termios *termios_p);
209.   #endif
```

3.12 time.h 文件

该头文件用于涉及处理时间的函数。所以内核中所有使用到处理时间的文件中都需要包含该头文件。操作系统中时间的处理较为复杂，涉及格林威治标准时间、本地时间或其他时间等。时间是从 GMT 1970 年 1 月 1 日开始的。time.h 文件的详细代码注释如下：

```
1.  #ifndef _TIME_H
2.  #define _TIME_H
3.  #ifndef _TIME_T
4.  #define _TIME_T
     // 从 GMT 1970 年 1 月 1 日开始的以秒计数的时间（日历时间）
5.  typedef long time_t;
6.  #endif
7.  #ifndef _SIZE_T
8.  #define _SIZE_T
9.  typedef unsigned int size_t;
```

```
10. #endif
11. #ifndef NULL
12. #define NULL ((void *) 0)
13. #endif
14. #define CLOCKS_PER_SEC 100          // 系统时钟滴答频率，100Hz
15. typedef long clock_t;                // 从进程开始系统经过的时钟滴答数
16. struct tm {
17. int tm_sec;                         // 秒数[0，59]
18. int tm_min;                         // 分钟数[ 0，59]
19. int tm_hour;                        // 小时数[0，23]
20. int tm_mday;                        // 1 个月的天数[0，31]
21. int tm_mon;                         // 1 年中月份[0，11]
22. int tm_year;                        // 从 1900 年开始的年数
23. int tm_wday;                        // 1 星期中的某天[0，6]（星期天=0）
24. int tm_yday;                        // 1 年中的某天[0，365]
25. int tm_isdst;                       // 夏令时标志
26. };
    //判断是否为闰年
27.  #define     __isleap(year)
28. ((year) % 4 == 0 && ((year) % 100 != 0 || (year) % 400 == 0))
    // 以下是有关时间操作的函数原型
    // 确定处理器使用时间。返回程序所用处理器时间（滴答数）的近似值
29. clock_t clock(void);
    // 取时间（秒数）。返回从 1970.1.1:0:0:0 开始的秒数（称为日历时间）
30. time_t time(time_t * tp);
    // 计算时间差。返回时间 time2 与 time1 之间经过的秒数
31. double difftime(time_t time2, time_t time1);
    // 将 tm 结构表示的时间转换成日历时间
32. time_t mktime(struct tm * tp);
    // 将 tm 结构表示的时间转换成一个字符串，返回指向该串的指针
33. char * asctime(const struct tm * tp);
    // 将日历时间转换成一个字符串形式，如“Wed Jun 30 21:49:08:1993\n”
34. char * ctime(const time_t * tp);
    // 将日历时间转换成 tm 结构表示的 UTC 时间（UTC-世界时间代码 Universal Time Code）
35. struct tm * gmtime(const time_t *tp);
    // 将日历时间转换成 tm 结构表示的指定时间区（timezone）的时间
36. struct tm *localtime(const time_t * tp);
    // 将 tm 结构表示的时间利用格式字符串 fmt 转换成最大长度为
    // smax 的字符串并将结果存储在 s 中
37. size_t strftime(char * s, size_t smax, const char * fmt, const struct tm * tp);
    // 初始化时间转换信息，使用环境变量 TZ，对 zname 变量进行初始化
    // 在与时间区相关的时间转换函数中将自动调用该函数
38. void tzset(void);
39. #endif
```

3.13 unistd.h 文件

unistd.h 是 Linux 标准符号常数和类型头文件。其中包含很多 Linux 系统服务的函数原型，是 Linux 内核的核心文件之一。该文件中定义了很多各种各样的常数和类型，以及一些系统调用

函数声明。unistd.h 文件的详细代码注释如下：

```
1.  #ifndef _UNISTD_H
2.  #define _UNISTD_H
3.  // 下面符号常数指出符合 IEEE 标准 1003.1 实现的版本号，是一个整数值
4.  #define _POSIX_VERSION 198808L
    // chown()和 fchown()的使用受限于进程的权限
5.  #define _POSIX_CHOWN_RESTRICTED   1    /* 只有超级用户可以执行 chown*/
    // 长于（NAME_MAX）的路径名将产生错误，而不会自动截断
6.  #define _POSIX_NO_TRUNC                1  /* 路径名不截断（但是请看内核代码）*/
    // 下面这个符号将定义成字符值，该值将禁止终端对其的处理
7.  #define _POSIX_VDISABLE                '\0'   /* 禁止像^C 这样的字符*/
    // 系统实现支持作业控制
8.  #define _POSIX_JOB_CONTROL     1
9.  // 每个进程都有一保存的 set-user-ID 和一保存的 set-group-ID
10. #define _POSIX_SAVED_IDS    1          /* 无论如何，已经实现了它*/
11. #define STDIN_FILENO        0          // 标准输入文件句柄（描述符）号
12. #define STDOUT_FILENO       1          // 标准输出文件句柄号
13. #define STDERR_FILENO       2          // 标准出错文件句柄号
14. #ifndef NULL
15. #define NULL      ((void *)0)          // 定义空指针
16. #endif
17. //* 文件访问*/
    // 以下定义的符号常数用于 access()函数
18. #define F_OK    0                      // 检测文件是否存在
19. #define X_OK    1                      // 检测是否可执行（搜索）
20. #define W_OK    2                      // 检测是否可写
21. #define R_OK    4                      // 检测是否可读
22. /* lseek *//* 文件指针重定位*/
23. #define SEEK_SET    0                  // 将文件读写指针设置为偏移值
24. #define SEEK_CUR    1                  // 将文件读写指针设置为当前值加上偏移值
25. #define SEEK_END    2                  // 将文件读写指针设置为文件长度加上偏移值
26.
27. /* _SC 表示系统配置，我们很少使用*/
    // 下面的符号常数用于 sysconf()函数
28. #define _SC_ARG_MAX         1          // 最大变量数
29. #define _SC_CHILD_MAX       2          // 子进程最大数
30. #define _SC_CLOCKS_PER_SEC 3           // 每秒滴答数
31. #define _SC_NGROUPS_MAX    4           // 最大组数
32. #define _SC_OPEN_MAX        5          // 最大打开文件数
33. #define _SC_JOB_CONTROL     6          // 作业控制
34. #define _SC_SAVED_IDS       7          // 保存的标识符
35. #define _SC_VERSION         8          // 版本
36.
37. /* 更多的（可能的）可配置参数——现在用于路径名*/
    // 下面的符号常数用于 pathconf()函数。
38. #define _PC_LINK_MAX        1          // 连接最大数
39. #define _PC_MAX_CANON       2          // 最大常规文件数
40. #define _PC_MAX_INPUT       3          // 最大输入长度
```

```
41.  #define _PC_NAME_MAX        4          // 名称最大长度
42.  #define _PC_PATH_MAX        5          // 路径最大长度
43.  #define _PC_PIPE_BUF        6          // 管道缓冲大小
44.  #define _PC_NO_TRUNC        7          // 文件名不截断
45.  #define _PC_VDISABLE        8          // 禁止系统维护的终端特殊字符
46.  #define _PC_CHOWN_RESTRICTED 9         // 改变宿主受限
47.  #include <sys/stat.h>                  // 文件状态头文件，含有文件或文件系统状态结构
                                            // stat{}和常量
48.  #include <sys/time.h>                  // 系统的时间日期头文件
49.  #include <sys/times.h>                 // 定义了进程中运行时间结构 tms 以及 times()函数原型
50.  #include <sys/utsname.h>               // 系统名称结构头文件
51.  #include <sys/resource.h>              // 为资源操作提供定义
52.  #include <utime.h>                     // 用户时间头文件
     //定义了访问和修改时间结构以及 utime()原型
53.  #ifdef __LIBRARY__
     // 以下是内核实现的系统调用符号常数，用作系统调用函数表中的索引值
54.  #define __NR_setup   0
55.  /* __NR_setup 仅用于初始化，以启动系统*/
56.  #define __NR_exit            1
57.  #define __NR_fork            2
58.  #define __NR_read            3
59.  #define __NR_write           4
60.  #define __NR_open            5
61.  #define __NR_close           6
62.  #define __NR_waitpid         7
63.  #define __NR_creat           8
64.  #define __NR_link            9
65.  #define __NR_unlink          10
66.  #define __NR_execve          11
67.  #define __NR_chdir           12
68.  #define __NR_time            13
69.  #define __NR_mknod           14
70.  #define __NR_chmod           15
71.    #define __NR_chown         16
72.    #define __NR_break         17
73.    #define __NR_stat          18
74.    #define __NR_lseek         19
75.    #define __NR_getpid        20
76.    #define __NR_mount         21
77.    #define __NR_umount        22
78.    #define __NR_setuid        23
79.    #define __NR_getuid        24
80.    #define __NR_stime         25
81.    #define __NR_ptrace        26
82.    #define __NR_alarm         27
83.    #define __NR_fstat         28
84.    #define __NR_pause         29
85.    #define __NR_utime         30
86.    #define __NR_stty          31
87.    #define __NR_gtty          32
```

```
88.    #define __NR_access          33
89.    #define __NR_nice            34
90.    #define __NR_ftime           35
91.    #define __NR_sync            36
92.    #define __NR_kill            37
93.    #define __NR_rename          38
94.    #define __NR_mkdir           39
95.    #define __NR_rmdir           40
96.    #define __NR_dup             41
97.    #define __NR_pipe            42
98.    #define __NR_times           43
99.    #define __NR_prof            44
100.   #define __NR_brk             45
101.   #define __NR_setgid          46
102.   #define __NR_getgid          47
103.   #define __NR_signal          48
104.   #define __NR_geteuid         49
105.   #define __NR_getegid         50
106.   #define __NR_acct            51
107.   #define __NR_phys            52
108.   #define __NR_lock            53
109.   #define __NR_ioctl           54
110.     #define __NR_fcntl         55
111.     #define __NR_mpx           56
112.     #define __NR_setpgid       57
113.   #define __NR_ulimit          58
114.   #define __NR_uname           59
115.   #define __NR_umask           60
116.   #define __NR_chroot          61
117.   #define __NR_ustat           62
118.   #define __NR_dup2            63
119.   #define __NR_getppid         64
120.   #define __NR_getpgrp         65
121.   #define __NR_setsid          66
122.   #define __NR_sigaction       67
123.   #define __NR_sgetmask        68
124.   #define __NR_ssetmask        69
125.   #define __NR_setreuid        70
126.   #define __NR_setregid        71
127.   #define __NR_sigsuspend      72
128.   #define __NR_sigpending      73
129.   #define __NR_sethostname     74
130.   #define __NR_setrlimit       75
131.   #define __NR_getrlimit       76
132.   #define __NR_getrusage       77
133.   #define __NR_gettimeofday    78
134.   #define __NR_settimeofday    79
135.   #define __NR_getgroups       80
136.   #define __NR_setgroups       81
137.   #define __NR_select          82
```

ÓÉ Foxit PDF Editor ±à¼-
°æÈ¨ËùÓÐ (c) by Foxit Software Company, 2004
½öÓÃÓÚÆÀ¹À¡£

- android与iphone及ipad开发书籍 ----------持续不断更新中.....
- c、c++、c#语言pdf书籍及vip视频教程 c、c++、c#、vc等-------持续不断更新中.....
- delphi《书籍》及《视频》教程 ----------持续不断更新中.....
- E网情深VIP系列视频教程 黑客破解菜鸟修练班，VB编程学习班，仿站学习培训，免杀培训，个人系统攻防系列教程，服务器搭建学习班，PHOTOSHOP平面设计班，基础制作论坛（论坛网站搭建），网赚系列教程，网站建设教程，网站漏洞基础，远程控制教程，软件破解班，脚本漏洞提权班
- IT9网络学院VIP系列视频教程 免杀培训班，VMware虚拟机，零基础学习C语言，网游外挂开发精品系列语音教程（外挂教程学习必备研修31课全），VB语言教程30课全，Delphi编程到精通，远程控制软件，加密解密班，网络安全与黑客攻防培训，从入门到精通完整系统化学习C++编程，从入门到精通零基础学习汇编，wordpress教程(个人博客系统49课全)，外行人做易语言盗号和钓鱼程序语音教程 网址：WLSAM168.400GB.COM
- Java书籍 ----------持续不断更新中.....
- photoshop、CorelDRAW、AutocAD等图像处理书籍及vip视频教程 -------持续不断更新中.....
- powerbuilder书籍大全
- Visual Basic语言vip视频教程及pdf书籍 ----------持续不断更新中.....
- windows、linux系统开发、系统封装等pdf书籍及VIP视频教程 ----------持续不断更新中.....
- 《3DS Max》pdf书籍
- 《汇编语言》、《反汇编》及《调试》pdf书籍及vip视频教程 ----------持续不断更新中.....
- 《电子书、电子书、还是电子书》pdf专题库 编程开发，家居美食，儿童益智，人物传记，增强记忆，快速阅读
- 信息系统项目管理师、网络工程师、系统分析师等软考类书籍
- 华中红客系列vip视频教程 脚本攻防培训班，源码免杀培训班，Css语言培训班，C语言，Dreamweaver网页设计，html网页设计培训班，PC安全班， php脚本语言培训班，VMWare虚拟机专题，webshell提权培训班，防站教程，零基础免杀培训班，刷钻速成班，脱壳破解班，外挂编写班，网络赚钱培训班，网站入侵培训班
- 外挂、驱动、逆向及封包视频教程 郁金香、独立团、夜猫论坛、天都吧、看流星论坛、一切从零开始等等
- 安全中国系列vip视频教程 易语言软件编程培训班，ASP.net网站开发项目实战培训班
- 我的收藏
- 按键精灵及TC脚本开发软件视频教程 ----------持续不断更新中.....

当前位置：/《电子书、电子书、还是电子书》pdf专题库

文件名

PDF电子书专题库，内容详尽，每天不断更新！！

- 办公类软件使用指南
- 医学
- 历史人物传记
- 哲学宗教
- 外语资料（除英语外） （除英语外）
- 官场类小说
- 建筑工程类
- 情感生活类小说
- 政治军事
- 教育学习科普大全
- 文学理论
- 智力开发、增强记忆、快速阅读技巧大全
- 社会生活
- 科学技术
- 程序编程类
- 经济管理
- 网络安全及管理
- 网赚系列
- 美食小吃烹饪煲汤大全
- 课外读物

本网盘内容太多，持续不断更新，发布各类视频教程、pdf书籍，包括破解、加解密、外挂辅助制作，易语言培训教程、编程语言、网页制作等等，教程及书籍仅用于学习,如用于商业或非法用途的后果自负！

网址：WLSAM168.400GB.COM

VIP培训教程，易语言黑月VIP视频教程，天都网易语言系列培训教程(100集全)，集中营易语言学习视频(80集)

- 棉猴系列vip视频教程 gh0st远控控制源码讲解教程，套接字编程，DLL程序编写，键盘监听驱动程序编写，驱动基础教程，AsyncSelect模型QQ程序教程，C++语言入门基础，NB5.5源码分析教程
- 游戏开发pdf书籍 ----------持续不断更新中.....
- 炒股投资pdf书籍及视频教程 短线高手系列，短线天王系列，操盘论道系列，翻倍黑马，看盘快速入门，庄家手法大曝光等等。 网址：WLSAM168.400GB.COM
- 热门小说集中营 傲世九重天，网游之三国时代，武动乾坤
- 甲壳虫VIP教程全集 asp教程，Delphi培训班，FLASH培训班，Java培训班，linux培训班，PHP培训班，源码免杀班，甲壳虫C++，脚本功防班，免杀班初、中、高级班，破解班，源码免杀班，脱壳班，易语言培训班，无特征码免杀，网站架构培训班，外挂高级班，外挂初级班第1、2部
- 破解、免杀、入侵、脱壳、攻防及漏洞分析系列VIP视频教程（80多部） 天草、黑客动画吧等等--------持续不断更新中....
- 网站建设相关的pdf书籍及各种vip视频教程 ----------持续不断更新中.....
- 网赚、淘宝系列vip视频教程 网赚30天新人魔鬼训练，屠龙网赚团队vip课程，站长大学网赚视频（50课全），图腾团队日赚1000元竞价行销教程，屠龙团队淘宝宝贝卖疯系列，站群网赚系列，淘宝开店视频，红星挂机日赚10元，百万流量系列，漂流瓶圣手全自动挂机引，贴吧邮件定向营销疯狂成交量月入万元
- 英语学习资料百科大全 不断更新。。。
- 饭客论坛系列VIP视频教程 脚本入侵班，黑客之免杀教程，易语言教程，无线网络攻防教程，入侵教程，delphi系列教程，黑客基础入门
- 黑客书籍 有关黑客、安全、加解密技术等等----------持续不断更新中.....
- 黑手安全网VIP系列视频教程 DIV+CSS网页布局，Dreamweaver教程，flsah动画教程，photoshop教程，跟我一起学C++课程，抓鸡
- 黑鹰、黑基、黑防、黑盾vip系列视频教程 破解提高班66课全，SQL注入，ASP注入教程，完完全全学会抓肉鸡，脱壳破解教程50课全，提权班，C语言特训班26讲全，黑客脚本特训班，黑客工具特训班，dedecms仿站教程，VC编写远控30课全，网页美工特训班，木马免杀特训班，驱动开发技术VIP培训班，外挂破解等等。

- [电脑世界的通关密语：电脑编程基础].(杉浦贤).滕永红.扫描版.pdf
- [程序语言的奥妙：算法解读（四色全彩）].(杉浦贤).李克秋.扫描版.pdf
- [差错：软件错误的致命影响].(帕伯斯).邝宇恒等.扫描版.pdf
- [算法之道（第2版）].邹恒明.扫描版.pdf
- [O'Reilly：深入学习MongoDB].(霍多罗夫).巨成等.扫描版.pdf
- [深入浅出WPF].刘铁猛.扫描版.pdf
- [Go语言·云动力（云计算时代的新型编程语言）].樊虹剑.扫描版.pdf
- [精通.NET互操作：P/ Invoke、C++ Interop和COM Interop].黄际洲等.扫描版.pdf
- [编程的奥秘：.NET软件技术学习与实践].金旭亮.扫描版.pdf
- [O'Reilly：学习OpenCV（中文版）].(布拉德斯基等).于仕琪等.扫描版.pdf
- [Go语言编程].许式伟等.扫描版.pdf 网址：WLSAM168.400GB.COM
- [MySQL技术内幕：SQL编程].姜承尧.扫描版.pdf
- [Tomcat权威指南（第2版）].(布里泰恩等).吴豪等.扫描版.pdf
- [Ext江湖].大漠穷秋.扫描版.pdf
- [IT名人堂·Oracle DBA突击：帮你赢得一份DBA职位].张晓明.扫描版.pdf

Total: 77 1 2 3 4 5 6 ›

HTTP://WLSAM168.400GB.COM

```
138.    #define __NR_symlink        83
139.    #define __NR_lstat          84
140.    #define __NR_readlink       85
141.    #define __NR_uselib         86
142.    #define __NR_swapon         87
143.    #define __NR_reboot         88
        // 以下定义系统调用嵌入式汇编宏函数
        // 不带参数的系统调用宏函数
144.  #define _syscall0(type,name)
145.  type name(void)
146.  {
147.  long __res;
      // 调用系统中断 0x80
148.  __asm__ volatile ("int $0x80"
149.  : "=a" (__res) // 返回值到 eax(__res)。
       // 输入为系统中断调用号__NR_name
150.  : "0" (__NR_##name));
       // 如果返回值>=0，则直接返回该值
151.  if (__res >= 0)
152.  return (type) __res;
       // 否则置出错号，并返回-1
153.  errno = -__res;
154.  return -1;
155.  }
      // 有 1 个参数的系统调用宏函数，type name(atype a)
156.  #define _syscall1(type,name,atype,a)
157.  type name(atype a)
158.  {
159.  long __res;
160.  __asm__ volatile ("movl %2,%%ebx\n\t"
161.  "int $0x80"
162.  : "=a" (__res)
163.  : "0" (__NR_##name),"g" ((long)(a)):"bx");
164.  if (__res >= 0)
165.  return (type) __res;
166.  errno = -__res;
167.  return -1;
168.  }
      // 有 2 个参数的系统调用宏函数
169.  #define _syscall2(type,name,atype,a,btype,b)
170.  type name(atype a,btype b)
171.  {
172.  long __res;
173.  __asm__ volatile ("movl %2,%%ebx\n\t"
174.  "int $0x80"
175.  : "=a" (__res)
176.  : "0" (__NR_##name),"g" ((long)(a)),"c" ((long)(b)):"bx");
177.  if (__res >= 0)
178.  return (type) __res;
179.  errno = -__res;
```

```
180.    return -1;
181.    }
        // 有 3 个参数的系统调用宏函数
182.    #define _syscall3(type,name,atype,a,btype,b,ctype,c)
183.    type name(atype a,btype b,ctype c)
184.    {
185.    long __res;
186.    __asm__ volatile ("movl %2,%%ebx\n\t"
187.    "int $0x80"
188.    : "=a" (__res)
189.    : "0" (__NR_##name),"g" ((long)(a)),"c" ((long)(b)),"d" ((long)(c)):"bx");
190.    if (__res>=0)
191.    return (type) __res;
192.    errno=-__res;
193.    return -1;
194.    }
195.    #endif
196.    extern int errno; //出错号，全局变量
        //对应各系统调用的函数原型定义
197.    int access(const char * filename, mode_t mode);
198.    int acct(const char * filename);
199.    int alarm(int sec);
200.    int brk(void * end_data_segment);
201.    void * sbrk(ptrdiff_t increment);
202.    int chdir(const char * filename);
203.    int chmod(const char * filename, mode_t mode);
204.    int chown(const char * filename, uid_t owner, gid_t group);
205.    int chroot(const char * filename);
206.    int close(int fildes);
207.    int creat(const char * filename, mode_t mode);
208.    int dup(int fildes);
209.    int execve(const char * filename, char ** argv, char ** envp);
210.    int execv(const char * pathname, char ** argv);
211.    int execvp(const char * file, char ** argv);
212.    int execl(const char * pathname, char * arg0, ...);
213.    int execlp(const char * file, char * arg0, ...);
214.    int execle(const char * pathname, char * arg0, ...);
215.    volatile void exit(int status);
216.    volatile void _exit(int status);
217.    int fcntl(int fildes, int cmd, ...);
218.    int fork(void);
219.    int getpid(void);
220.    int getuid(void);
221.    int geteuid(void);
222.    int getgid(void);
223.    int getegid(void);
224.    int ioctl(int fildes, int cmd, ...);
225.    int kill(pid_t pid, int signal);
226.    int link(const char * filename1, const char * filename2);
227.    int lseek(int fildes, off_t offset, int origin);
```

```
228.  int mknod(const char * filename, mode_t mode, dev_t dev);
229.  int mount(const char * specialfile, const char * dir, int rwflag);
230.  int nice(int val);
231.  int open(const char * filename, int flag, ...);
232.  int pause(void);
233.  int pipe(int * fildes);
234.  int read(int fildes, char * buf, off_t count);
235.  int setpgrp(void);
236.  int setpgid(pid_t pid,pid_t pgid);
237.  int setuid(uid_t uid);
238.  int setgid(gid_t gid);
239.  void (*signal(int sig, void (*fn)(int)))(int);
240.  int stat(const char * filename, struct stat * stat_buf);
241.  int fstat(int fildes, struct stat * stat_buf);
242.  int stime(time_t * tptr);
243.  int sync(void);
244.  time_t time(time_t * tloc);
245.  time_t times(struct tms * tbuf);
246.  int ulimit(int cmd, long limit);
247.  mode_t umask(mode_t mask);
248.  int umount(const char * specialfile);
249.  int uname(struct utsname * name);
250.  int unlink(const char * filename);
251.  int ustat(dev_t dev, struct ustat * ubuf);
252.  int utime(const char * filename, struct utimbuf * times);
253.  pid_t waitpid(pid_t pid,int * wait_stat,int options);
254.  pid_t wait(int * wait_stat);
255.  int write(int fildes, const char * buf, off_t count);
256.  int dup2(int oldfd, int newfd);
257.  int getppid(void);
258.  pid_t getpgrp(void);
259.  pid_t setsid(void);
260.  int sethostname(char *name, int len);
261.  int setrlimit(int resource, struct rlimit *rlp);
262.  int getrlimit(int resource, struct rlimit *rlp);
263.  int getrusage(int who, struct rusage *rusage);
264.  int gettimeofday(struct timeval *tv, struct timezone *tz);
265.  int settimeofday(struct timeval *tv, struct timezone *tz);
266.  int getgroups(int gidsetlen, gid_t *gidset);
267.  int setgroups(int gidsetlen, gid_t *gidset);
268.  int select(int width, fd_set * readfds, fd_set * writefds,
269.  fd_set * exceptfds, struct timeval * timeout);
270.  int swapon(const char * specialfile);
271.  #endif
```

3.14 utime.h 文件

该文件定义了文件访问和修改时间的结构和函数原型。主要应用在文件管理中。该文件主要定义了 utimbuf 以及 utime()原型，时间以秒为单位计算。utime.h 文件的详细代码注释如下：

```
1.  #ifndef _UTIME_H
2.  #define _UTIME_H
3.  #include <sys/types.h>
4.  struct utimbuf {
5.  time_t actime;              // 文件访问时间。从 1970.1.1:0:0:0 开始的秒数
6.  time_t modtime;             // 文件修改时间。从 1970.1.1:0:0:0 开始的秒数
7.  };
    // 设置文件访问和修改时间函数
8.  extern int utime(const char *filename, struct utimbuf *times);
```

3.15 include/asm 目录下的文件

include/asm 目录下的文件如表 3-2 所示。

表 3-2 linux/include/asm 目录下的文件

名　　称	大　小（B）	修　改　时　间
io.h	606	1992-03-08 20:04:26
memory.h	507	1992-03-08 19:51:00
segme.h	1399	1992-03-05 10:03:10
system.h	1717	1992-03-05 10:10:42

3.16 io.h 文件

io.h 是硬件 IO 控制头文件。主要用于实现对硬件 IO 端口访问控制操作。

本程序主要定义了对硬件 IO 端口访问的嵌入式汇编宏函数：outb()、inb()以及 outb_p()和 inb_p()。前两个函数与后两个的主要区别在于后者代码中使用了 jmp 指令进行了时间延迟。

io.h 文件的详细代码注释如下：

```
    // 硬件端口字节输出函数。参数：value - 欲输出字节；port - 端口
1.  static void inline outb(char value, unsigned short port)
2.  {
3.  __asm__ volatile ("outb %0,%1"::"a" (value),"d" (port));
4.  }
    // 带延迟的硬件端口字节输出函数。参数：value - 欲输出字节；port - 端口
5.  static void inline outb_p(char value, unsigned short port)
6.  {
7.  __asm__ volatile ("outb %0,%1\n\tjmp 1f\n1:\tjmp 1f\n1:"
8.  ::"a" (value),"d" (port));
9.  }
    // 硬件端口字节输入函数。参数：port - 端口。返回读取的字节
10. static unsigned char inline inb(unsigned short port)
11. {
12. unsigned char _v;
13. __asm__ volatile ("inb %1,%0":"=a" (_v):"d" (port));
14. return _v;
15. }
    // 带延迟的硬件端口字节输入函数。参数：port - 端口。返回读取的字节
16. static unsigned char inb_p(unsigned short port)
```

```
17.     {
18.         unsigned char _v;
19.         __asm__ volatile ("inb %1,%0\n"
20.         "\tjmp 1f\n"
21.         "1:\tjmp 1f\n"
22.         "1:":"=a" (_v):"d" ((unsigned short) port));
23.     return _v;
24.     }
```

3.17 memory.h 文件

memory.h 是内存操作头文件。主要完成对内存数据复制的操作。在内存管理文件中经常被包含。

本文件含有一个内存复制嵌入式汇编宏 memcpy()。与 string.h 中定义的 memcpy()相同，只是后者采用的是嵌入式汇编 C 函数形式定义的。memory.h 文件的详细代码注释如下：

```
1.  #define memcpy(dest,src,n) ({
2.  void * _res = dest;
3.  10   __asm__ ("cld;rep;movsb"          // 从 ds:[esi]复制到 es:[edi]，并且 esi++，edi++
4.  // 共复制 ecx(n)字节
5.  ::"D" ((long)(_res)),"S" ((long)(src)),"c" ((long) (n))
6.  :"di","si","cx");
7.  _res;
8.  })
```

注意：memcpy(dest,src,n)假设段寄存器 ds=es=通常数据段。在内核中使用的所有函数都基于该假设（ds=es=内核空间，fs=局部数据空间，gs=null），具有良好行为的应用程序也是这样（ds=es=用户数据空间）。如果任何用户程序随意改动了 es 寄存器而出错，则并不是由于系统程序错误造成的。该段程序主要用于内存块复制。从源地址 src 处开始复制 n 个字节到目的地址 dest 处。参数：dest-复制的目的地址，src-复制的源地址，n-复制字节数。%0-edi(目的地址 dest)，%1-esi(源地址 src)，%2-ecx(字节数 n)。

3.18 segment.h 文件

segment.h 是段寄存器操作头文件。主要完成对段寄存器相关的操作。在设备管理的相关文件中经常包含该头文件。本文件中定义了一些访问段寄存器或与段寄存器有关的内存操作的嵌入式汇编函数。segment.h 文件的详细代码注释如下：

```
1.      extern inline unsigned char get_fs_byte(const char * addr)
2.          {
3.          unsigned register char _v;
4.          __asm__ ("movb %%fs:%1,%0":"=q" (_v):"m" (*addr));
5.      return _v;
6.      }
7.          extern inline unsigned short get_fs_word(const unsigned short *addr)
8.          {
9.          unsigned short _v;
```

```
10.       __asm__ ("movw %%fs:%1,%0":"=r" (_v):"m" (*addr));
11.       return _v;
12.          }
13.          extern inline unsigned long get_fs_long(const unsigned long *addr)
14.          {
15.          unsigned long _v;
16.       __asm__ ("movl %%fs:%1,%0":"=r" (_v):"m" (*addr));
17.       return _v;
18.          }
19.          extern inline void put_fs_byte(char val,char *addr)
20.          {
21.          __asm__ ("movb %0,%%fs:%1"::"q" (val),"m" (*addr));
22.    }
23.       extern inline void put_fs_word(short val,short * addr)
24.          {
25.          __asm__ ("movw %0,%%fs:%1"::"r" (val),"m" (*addr));
26.       }
27.       extern inline void put_fs_long(unsigned long val,unsigned long * addr)
28.          {
29.          __asm__ ("movl %0,%%fs:%1"::"r" (val),"m" (*addr));
30.       }
          // 取 fs 段寄存器值（选择符）。返回：fs 段寄存器值
31.       extern inline unsigned long get_fs()
32.          {
33.          unsigned short _v;
34.       __asm__("mov %%fs,%0":"=r" (_v):);
35.       return _v;
36.          }
          //  取 ds 段寄存器值。返回：ds 段寄存器值
37.          extern inline unsigned long get_ds()
38.          {
39.          unsigned short _v;
40.       __asm__("mov %%ds,%0":"=r" (_v):);
41.       return _v;
42.          }
          // 设置 fs 段寄存器。参数：val - 段值（选择符）
43.          extern inline void set_fs(unsigned long val)
44.       {
45.       __asm__("mov %0,%%fs"::"r" ((unsigned short) val));
46.          }
```

代码中第 1-6 行是读取 fs 段中指定地址处的字节。

参数：addr-指定的内存地址。%0-（返回的字节_v）；%1-（内存地址 addr）。返回：返回内存 fs:[addr]处的字节。代码中第 7～12 行是读取 fs 段中指定地址处的字。

参数：addr-指定的内存地址。%0-（返回的字_v）；%1-（内存地址 addr）。返回：返回内存 fs:[addr]处的字。代码中第 13～18 行是读取 fs 段中指定地址处的长字（4 字节）。

参数：addr-指定的内存地址。%0-（返回的长字_v）；%1-（内存地址 addr）。返回：返回内存 fs:[addr]处的长字。代码中第 19～22 行是将一字节存放在 fs 段中指定内存地址处。

参数：val-字节值；addr-内存地址。%0-寄存器（字节值 val）；%1-（内存地址 addr）。代码

中第 23～26 行是将一字存放在 fs 段中指定内存地址处。

参数：val-字值；addr-内存地址。%0-寄存器（字值 val）；%1-（内存地址 addr）。代码中第 27～30 行是将一长字存放在 fs 段中指定内存地址处。

参数：val-长字值；addr-内存地址。%0-寄存器（长字值 val）；%1-（内存地址 addr）。

3.19 system.h 文件

system.h 是系统函数头文件。在其中主要定义了设置或修改描述符/中断门等的嵌入式汇编宏。

本程序中重要函数 move_to_user_mode()是用于内核在初始化结束时人工切换到初始进程（任务 0）去执行。所使用的方法是模拟中断调用返回过程，也即利用指令 iret 运行初始任务 0。

任务 0 是一个特殊进程，它的数据段和代码段直接映射到内核代码和数据空间，即从物理地址 0 开始的 640K 内存空间，其堆栈地址即是内核代码所使用的堆栈。因此图中堆栈中的原 SS 和原 ESP 是将现有内核堆栈指针直接压入堆栈的。system.h 文件的详细代码注释如下：

```
// 切换到用户模式运行。该函数利用 iret 指令实现从内核模式切换到用户模式（初始任务 0）
1.      #define move_to_user_mode()
2.          __asm__ ("movl %%esp,%%eax\n\t"                    // 保存堆栈指针 esp 到 eax 寄存器中
3.                  "pushl $0x17\n\t"                          // 首先将堆栈段选择符（SS）入栈
4.          "pushl %%eax\n\t"                                  // 然后将保存的堆栈指针值（esp）入栈
5.          "pushfl\n\t"                                       // 将标志寄存器（eflags）内容入栈
6.                  "pushl $0x0f\n\t"                          // 将内核代码段选择符（cs）入栈
7.                  "pushl $1f\n\t"                            // 将下面标号 1 的偏移地址（eip）入栈
8.          "iret\n"                                           // 执行中断返回指令，则会跳转到下面标号 1 处
9.          "1:\tmovl $0x17,%%eax\n\t"                         // 此时开始执行任务 0
10.                 "mov %%ax,%%ds\n\t"                        // 初始化段寄存器指向本局部表的数据段
11.         "mov %%ax,%%es\n\t"
12.     "mov %%ax,%%fs\n\t"
13.     "mov %%ax,%%gs"
14.         :::"ax")
15.             #define sti() __asm__ ("sti"::)                // 开中断嵌入汇编宏函数
16.             #define cli() __asm__ ("cli"::)                // 关中断
17.         #define nop() __asm__ ("nop"::)                    // 空操作
18.         #define iret() __asm__ ("iret"::)                  // 中断返回
19.         #define _set_gate(gate_addr,type,dpl,addr)
20.         __asm__ ("movw %%dx,%%ax\n\t" \                    // 将偏移地址低字与选择符组合成描述符低 4
                                                               //字节(eax)
21.     "movw %0,%%dx\n\t"                                     // 将类型标志字与偏移高字组合成描述符高 4
                                                               //字节(edx)
22.     "movl %%eax,%1\n\t"                                    // 分别设置门描述符的低 4 字节和高 4 字节
23.         "movl %%edx,%2"
24.         :: "i" ((short) (0x8000+(dpl<<13)+(type<<8))),
25.     "m" (*((char *) (gate_addr))),
26.     "m" (*(4+(char *) (gate_addr))),
27.         "d" ((char *) (addr)),"a" (0x00080000)
28.         :"ax","dx")
29.         #define set_intr_gate(n,addr)
30.     _set_gate(&idt[n],14,0,addr)
    // 设置陷阱门函数
```

```
    // 参数：n-中断号；addr-中断程序偏移地址
    // &idt[n]对应中断号在中断描述符表中的偏移值；中断描述符的类型是 15，特权级是 0
31.   #define set_trap_gate(n,addr)
32.   _set_gate(&idt[n],15,0,addr)
33.     #define set_system_gate(n,addr)
34.     _set_gate(&idt[n],15,3,addr)
35.     #define _set_seg_desc(gate_addr,type,dpl,base,limit) {
36.   *(gate_addr) = ((base) & 0xff000000) |
37.   (((base) & 0x00ff0000)>>16) |
38.     ((limit) & 0xf0000) |
39.     ((dpl)<<13) |
40.     (0x00408000) |
41.     ((type)<<8);
42.   *((gate_addr)+1) = (((base) & 0x0000ffff)<<16) |
43.   ((limit) & 0x0ffff); }
    // %0-eax（地址 addr）；%1-（描述符项 n 的地址）；%2-（描述符项 n 的地址偏移 2 处）
    // %3-（描述符项 n 的地址偏移 4 处）；%4-（描述符项 n 的地址偏移 5 处）
    // %5-（描述符项 n 的地址偏移 6 处）；%6-（描述符项 n 的地址偏移 7 处）
44.   #define _set_tssldt_desc(n,addr,type)
45.   __asm__ ("movw $104,%1\n\t"
46.     "movw %%ax,%2\n\t"
47.     "rorl $16,%%eax\n\t"
48.     "movb %%al,%3\n\t"
49.     "movb $" type ",%4\n\t"
50.   "movb $0x00,%5\n\t"
51.   "movb %%ah,%6\n\t"
52.     "rorl $16,%%eax"
53.     ::"a" (addr), "m" (*(n)), "m" (*(n+2)), "m" (*(n+4)),
54.   "m" (*(n+5)), "m" (*(n+6)), "m" (*(n+7))
55.   )
56.     #define set_tss_desc(n,addr) _set_tssldt_desc(((char *) (n)),addr,"0x89")
57.     #define set_ldt_desc(n,addr) _set_tssldt_desc(((char *) (n)),addr,"0x82")
```

代码中第 20～28 行是设置门描述符宏函数。参数：

gate_addr-描述符地址；

type-描述符中类型域值；

dpl-描述符特权层值；

addr-偏移地址。%0-（由 dpl,type 组合成的类型标志字）；%1-（描述符低 4 字节地址）；%2-（描述符高 4 字节地址）；%3-edx（程序偏移地址 addr）；%4-eax（高字中含有段选择符）。

代码中第 29～30 行是设置中断门函数。参数：n-中断号；addr-中断程序偏移地址。&idt[n]对应中断号在中断描述符表中的偏移值；中断描述符的类型是 14，特权级是 0。

代码中第 33～34 行是设置系统调用门函数。参数：n-中断号；addr-中断程序偏移地址。&idt[n]对应中断号在中断描述符表中的偏移值；中断描述符的类型是 15，特权级是 3。

代码中第 35～43 设置段描述符函数。 参数：gate_addr-描述符地址；type-描述符中类型域值；dpl-描述符特权层值；base-段的基地址；limit-段限长。（参见段描述符的格式）

代码中第 44～45 行是在全局表中设置任务状态段/局部表描述符。参数：n-在全局表中描述符项 n 所对应的地址；addr-状态段/局部表所在内存的基地址。type-描述符中的标志类型字节。

代码中第 56 行是在全局表中设置任务状态段描述符。n-是该描述符的指针；addr-是描述符中的基地址值。任务状态段描述符的类型是 0x89。代码中第 57 行是在全局表中设置局部表描述符。n-是该描述符的指针；addr-是描述符中的基地址值。局部表描述符的类型是 0x82。

3.20 include/linux/目录下的文件

include/linux/该目录下含有的头文件见表 3-3。

表 3-3 include/linux/目录下的文件

名 称	大 小（B）	最 后 修 改 时 间
config.h	1775	1992-03-06 10:19:58
fdreg.h	2466	1991-11-02 18:48:44
fs.h	6425	1992-03-06 20:26:12
hdreg.h	2076	1992-02-01 22:04:36
head.h	304	1991-06-20 03:24:14
kernel.h	812	1992-01-27 23:38:58
mm.h	1210	1992-03-06 08:57:32
sched.h	7646	1992-03-08 20:05:00
sys.h	3476	1992-02-28 00:24:26
tty.h	2938	1992-03-01 21:55:46
minix_fs.h	2808	1992-03-03 05:52:28
math_emu.h	4980	1992-02-18 04:56:32
timer.h	787	1992-2-26 07:02:10

3.21 config.h 文件

config.h 是内核配置头文件。主要定义了键盘语言类型和硬盘类型（HD_TYPE）可选项。

本程序中 Linux 能够在启动时从 BIOS 中获取驱动器的参数，但是若由于未知原因而没有得到这些参数时，会使程序束手无策。对于这种情况，可以定义 HD_TYPE，其中包括硬盘的所有信息。

config.h 文件的详细代码注释如下：

```
1.    #ifndef _CONFIG_H
2.    #define _CONFIG_H
3.  /*
4.  * 根文件系统设备已不再是硬编码的了。通过修改 boot/bootsect.s 文件中
5.  * ROOT_DEV = XXX，可以改变根设备的默认设置值
6.  */
7.  /*
8.  * 在这里定义键盘类型：
9.  * KBD_FINNISH 是芬兰键盘
10. * KBD_US 是美式键盘
11. * KBD_GR 是德式键盘
12. * KBD_FR 是法式键盘
```

```
13.  */
14.   通常，Linux 能够在启动时从 BIOS 中获取驱动器德参数，但是若由于未知原因而没有得到这
     些参数时，会使程序束手无策。对于这种情况，可以定义 HD_TYPE，其中包括硬盘的所有信息
15.  * HD_TYPE 宏应该像这样的形式:#define HD_TYPE { head, sect, cyl, wpcom, lzone, ctl}对于有两
     个硬盘的情况，参数信息需用逗号分开：  #define HD_TYPE { h,s,c,wpcom,lz,ctl },
     {h,s,c,wpcom,lz,ctl }
16.   /* 下面是一个例子，两个硬盘，第 1 个是类型 2，第 2 个是类型 3：
17.  * #define HD_TYPE { 4,17,615,300,615,8 }, {6,17,615,300,615,0 }
18.  *注意：对应所有硬盘，若其磁头数<=8，则 ctl 等于 0，若磁头数多于 8 个，
19.  *  则 ctl=8
20.  *如果你想让 BIOS 给出硬盘的类型，那么只需不定义 HD_TYPE。这是默认操作
21.  */
22.    #endif
```

代码中 KBD_FINNISH 是芬兰键盘；KBD_US 是美式键盘； KBD_GR 是德式键盘；KBD_FR 是法式键盘。通常 Linux 能够在启动时从 BIOS 中获取驱动器德参数，但是若由于未知原因而没有得到这些参数时，会使程序束手无策。对于这种情况，你可以定义 HD_TYPE，其中包括硬盘的所有信息。 HD_TYPE 宏应该象下面这样的形式：#define HD_TYPE { head, sect, cyl, wpcom, lzone, ctl}对于有两个硬盘的情况，参数信息需用逗号分开： #define HD_TYPE { h,s,c,wpcom,lz,ctl }, {h,s,c,wpcom,lz,ctl }。

3.22 fdreg.h 头文件

fdreg.h 是软盘系统头文件。主要定义了软盘系统常用到的一些参数以及所使用的 I/O 端口。

3.22.1 功能描述

该头文件用以说明软盘系统常用到的一些参数以及所使用的 I/O 端口。由于软盘驱动器的控制比较烦琐，命令也多，因此在阅读代码之前，最好先参考有关微型计算机控制接口原理的书籍，了解软盘控制器（FDC）的工作原理，然后你就会觉得这里的定义还是比较合理有序的。在编程时需要访问 4 个端口，分别对应一个或多个寄存器。

数字输出端口（数字控制端口）是一个 8 位寄存器，它控制驱动器马达开启、驱动器选择、启动/复位 FDC 以及允许/禁止 DMA 及中断请求。FDC 的主状态寄存器也是一个 8 位寄存器，用于反映软盘控制器 FDC 和软盘驱动器 FDD 的基本状态。

通常，在 CPU 向 FDC 发送命令之前或从 FDC 获取操作结果之前，都要读取主状态寄存器的状态位，以判别当前 FDC 数据寄存器是否就绪，以及确定数据传送的方向。FDC 的数据端口对应多个寄存器（只写型命令寄存器和参数寄存器、只读型结果寄存器），但任一时刻只能有一个寄存器出现在数据端口 0x3f5。在访问只写型寄存器时，主状态控制的 DIO 方向位必须为 0，访问只读型寄存器时则反之。

在读取结果时只有在 FDC 不忙之后才算读完结果，通常结果数据最多有 7 个字节。软盘控制器共可以接受 15 条命令。每个命令均经历三个阶段：命令阶段、执行阶段和结果阶段。命令阶段是 CPU 向 FDC 发送命令字节和参数字节。每条命令的第一个字节总是命令字节（命令码）。其后跟着 0～8 字节的参数。执行阶段是 FDC 执行命令规定的操作。在执行阶段 CPU 是不加干预的，一般是通过 FDC 发出中断请求获知命令执行的结束。

如果 CPU 发出的 FDC 命令是传送数据，则 FDC 可以以中断方式或 DMA 方式进行。中断

方式每次传送 1 字节。DMA 方式是在 DMA 控制器管理下，FDC 与内存进行数据的传输直至全部数据传送完。此时 DMA 控制器会将传输字节计数终止信号通知 FDC，最后由 FDC 发出中断请求信号告知 CPU 执行阶段结束。结果阶段是由 CPU 读取 FDC 数据寄存器返回值，从而获得 FDC 命令执行的结果。返回结果数据的长度为 0～7 字节。对于没有返回结果数据的命令，则应向 FDC 发送检测中断状态命令获得操作的状态。

3.22.2 代码注释

fdreg.h 文件的详细代码注释如下：

```
1.      #ifndef _FDREG_H // 该定义用来排除代码中重复包含此头文件
2.      #define _FDREG_H
      // 一些软盘类型函数的原型说明
3.      extern int ticks_to_floppy_on(unsigned int nr);
4.      extern void floppy_on(unsigned int nr);
5.      extern void floppy_off(unsigned int nr);
6.      extern void floppy_select(unsigned int nr);
7.      extern void floppy_deselect(unsigned int nr);
      // 下面是有关软盘控制器一些端口和符号的定义
        /* 软盘控制器(FDC)寄存器端口 */
8.      #define FD_STATUS 0x3f4          // 主状态寄存器端口
9.      #define FD_DATA 0x3f5            // 数据端口
10.     #define FD_DOR 0x3f2
      // 数字输出寄存器（也称为数字控制寄存器）
11.     #define FD_DIR 0x3f7
      // 数字输入寄存器
12.     #define FD_DCR 0x3f7
      // 数据传输率控制寄存器
      /* 主状态寄存器各比特位的含义 */
13.     #define STATUS_BUSYMASK 0x0F
      // 驱动器忙位（每位对应一个驱动器）
14.     #define STATUS_BUSY 0x10
      // 软盘控制器忙
15.     #define STATUS_DMA 0x20
      // 0 -为 DMA 数据传输模式，1 -为非 DMA 模式
16.     #define STATUS_DIR 0x40
      // 传输方向：0 - CPU ?? fdc，1 -相反
17.     #define STATUS_READY 0x80
      // 数据寄存器就绪位
      /*状态字节 0（ST0）各比特位的含义 */
18.     #define ST0_DS 0x03
      // 驱动器选择号（发生中断时驱动器号）
19.     #define ST0_HA 0x04
      // 磁头号
20.     #define ST0_NR 0x08
      // 磁盘驱动器未准备好
21.     #define ST0_ECE 0x10
      // 设备检测出错（零磁道校准出错）
22.     #define ST0_SE 0x20
      // 寻道或重新校正操作执行结束
```

```
23.     #define ST0_INTR 0xC0
      // 中断代码位（中断原因），00-命令正常结束；
      // 01-命令异常结束；10-命令无效；11-FDD 就绪状态改变
      /*状态字节 1（ST1）各比特位的含义 */
24.     #define ST1_MAM 0x01
      // 未找到地址标志（ID AM）
25.     #define ST1_WP 0x02
      // 写保护
26.     #define ST1_ND 0x04
      // 未找到指定的扇区
27.     #define ST1_OR 0x10
      // 数据传输超时（DMA 控制器故障）
28.     #define ST1_CRC 0x20
      // CRC 检验出错
29.     #define ST1_EOC 0x80
      // 访问超过一个磁道上的最大扇区号
      /*状态字节 2（ST2）各比特位的含义 */
30.     #define ST2_MAM 0x01
      // 未找到数据地址标志
31.     #define ST2_BC 0x02
      // 磁道坏
32.     #define ST2_SNS 0x04
      // 检索（扫描）条件不满足
33.     #define ST2_SEH 0x08
      // 检索条件满足
34.     #define ST2_WC 0x10
      // 磁道（柱面）号不符
35.     #define ST2_CRC 0x20
      // 数据场 CRC 校验错
36.     #define ST2_CM 0x40
      // 读数据遇到删除标志
      /*状态字节 3（ST3）各比特位的含义 */
37.     #define ST3_HA 0x04
      // 磁头号
38.     #define ST3_TZ 0x10
      // 零磁道信号
39.     #define ST3_WP 0x40
      // 写保护
      /* 软盘命令码 */
40.     #define FD_RECALIBRATE 0x07
      // 重新校正（磁头退到零磁道）
41.     #define FD_SEEK 0x0F
      // 磁头寻道。
42.     #define FD_READ 0xE6
      // 读数据（MT 多磁道操作，MFM 格式，跳过删除数据）
43.     #define FD_WRITE 0xC5
      // 写数据（MT，MFM）
44.     #define FD_SENSEI 0x08
      // 检测中断状态
45.     #define FD_SPECIFY 0x03
```

```
        // 设定驱动器参数（步进速率、磁头卸载时间等）
        /* DMA 命令 */
46.     #define DMA_READ 0x46 // DMA 读盘，DMA 方式字（送 DMA 端口 12，11）
47.     #define DMA_WRITE 0x4A // DMA 写盘，DMA 方式字
48.     #endif
```

代码中第 3～7 行是一些软盘类型函数的原型说明。其中包括有关软盘控制器一些端口和符号的定义；软盘控制器（FDC）寄存器端口；主状态寄存器端口；数据端口；数字输出寄存器；数字输入寄存器；数据传输率控制寄存器；设定驱动器参数等。

3.23 fs.h 文件

fs.h 是文件系统的头文件，主要描述了文件操作的一些常量、高速缓冲块结构以及 Minix 文件系统 1.0 版的结构。该头文件用以定义文件系统基本的系统数据类型。方便后续程序对文件系统的使用。所以在内核文件管理的文件中都需要包含该头文件。

fs.h 文件的详细代码注释如下：

```
1.      #ifndef _FS_H
2.      #define _FS_H
3.      #include <sys/types.h> // 类型头文件，定义了基本的系统数据类型
4.      /*
5.      * 系统所含的设备如下：（与 Minix 系统的一样，所以我们可以使用 Minix 的
6.      * 文件系统，以下这些是主设备号）
7.      * 0 -没有用到（nodev）
8.      * 1 - /dev/mem 内存设备
9.      * 2 - /dev/fd 软盘设备
10.     * 3 - /dev/hd 硬盘设备
11.     * 4 - /dev/ttyx tty 串行终端设备
12.     * 5 - /dev/tty tty 终端设备
13.     * 6 - /dev/lp 打印设备
14.     * 7 - unnamed pipes 没有命名的管道
15.     */
16.     #define IS_SEEKABLE(x) ((x)>=1 && (x)<=3)     // 是否是可以寻找定位的设备
17.     #define READ 0
18.     #define WRITE 1
19.     #define READA 2
20.     #define WRITEA 3
21.     void buffer_init(long buffer_end);
22.     #define MAJOR(a) (((unsigned)(a))>>8) // 取高字节（主设备号）
23.     #define MINOR(a) ((a)&0xff) // 取低字节（次设备号）
24.     #define NAME_LEN 14                   // 名字长度值
25.     #define ROOT_INO 1                    // 根 i 结点
26.     #define I_MAP_SLOTS 8                 //i 结点位图槽数
27.     #define Z_MAP_SLOTS 8                 // 逻辑块（区段块）位图槽数
28.     #define SUPER_MAGIC 0x137F            // 文件系统魔数
29.     #define NR_OPEN 20                    // 打开文件数
30.     #define NR_INODE 32
31.     #define NR_FILE 64
32.     #define NR_SUPER 8
33.     #define NR_HASH 307
```

```
34.     #define NR_BUFFERS nr_buffers
35.     #define BLOCK_SIZE 1024                 // 数据块长度
36.     #define BLOCK_SIZE_BITS 10              // 数据块长度所占比特位数
37.     #ifndef NULL
38.     #define NULL ((void *) 0)
39.     #endif
    // 每个逻辑块可存放的 i 结点数
40.     #define INODES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct d_inode)))
    // 每个逻辑块可存放的目录项数
41.     #define DIR_ENTRIES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct dir_entry)))
    // 管道头、管道尾、管道大小、管道空、管道满、管道头指针递增的宏定义
42.     #define PIPE_HEAD(inode) ((inode).i_zone[0])
43.     #define PIPE_TAIL(inode) ((inode).i_zone[1])
44.     #define PIPE_SIZE(inode) ((PIPE_HEAD(inode)-PIPE_TAIL(inode))&(PAGE_SIZE-1))
45.     #define PIPE_EMPTY(inode) (PIPE_HEAD(inode)==PIPE_TAIL(inode))
46.     #define PIPE_FULL(inode) (PIPE_SIZE(inode)==(PAGE_SIZE-1))
47.     #define INC_PIPE(head)
48.     __asm__("incl %0\n\tandl $4095,%0"::"m" (head))
49.     typedef char buffer_block[BLOCK_SIZE]; // 块缓冲区
    // 缓冲区头数据结构。(极为重要!)
    // 在程序中常用 bh 来表示 buffer_head 类型的缩写
50.     struct buffer_head {
51.     char * b_data;                          // 指针
52.     unsigned long b_blocknr;                // 块号
53.     unsigned short b_dev;                   // 数据源的设备号
54.     unsigned char b_uptodate;               // 更新标志：表示数据是否已更新
55.     unsigned char b_dirt;                   // 修改标志：0，未修改；1，已修改
56.     unsigned char b_count;                  // 使用的用户数。
57.     unsigned char b_lock;                   // 缓冲区是否被锁定：0，未锁定；1：锁定
58.     struct task_struct * b_wait;            // 指向等待该缓冲区解锁的任务
59.     struct buffer_head * b_prev;            // hash 队列上前一块（这四个指针用于缓
                                                //冲区的管理）
60.     struct buffer_head * b_next;            // hash 队列上下一块
61.     struct buffer_head * b_prev_free;       // 空闲表上前一块
62.     struct buffer_head * b_next_free;       // 空闲表上下一块
63.     };
    // 磁盘上的索引结点（i 结点）数据结构
64.     struct d_inode {
65.     unsigned short i_mode;          // 文件类型和属性（rwx 位）
66.     unsigned short i_uid;           // 用户 id（文件拥有者标识符）
67.     unsigned long i_size;           // 文件大小（字节数）
68.     unsigned long i_time;           // 修改时间（自 1970.1.1:0 算起，秒）
69.     unsigned char i_gid;            // 组 id（文件拥有者所在的组）
70.     unsigned char i_nlinks;         // 链接数（多少个文件目录项指向该 i 结点）
71.     unsigned short i_zone[9];       // 直接（0-6）、间接（7）或双重间接（8）逻辑块号
    // zone 是区的意思，可译成区段或逻辑块
72.     };
    // 这是在内存中的 i 结点结构。前 7 项与 d_inode 完全一样
73.     struct m_inode {
74.     unsigned short i_mode;          // 文件类型和属性（rwx 位）
```

```
75.         unsigned short i_uid;              // 用户 id（文件拥有者标识符）
76.         unsigned long i_size;              // 文件大小（字节数）
77.         unsigned long i_mtime;             // 修改时间（自 1970.1.1:0 算起，秒）
78.         unsigned char i_gid;               // 组 id（文件拥有者所在的组）
79.         unsigned char i_nlinks;            // 文件目录项链接数
80.         unsigned short i_zone[9];          // 直接（0-6）、间接（7）或双重间接（8）逻辑块号
81.
82.         struct task_struct * i_wait;       // 等待该 i 结点的进程
83.         unsigned long i_atime;             // 最后访问时间
84.         unsigned long i_ctime;             //i 结点自身修改时间
85.         unsigned short i_dev;              //i 结点所在的设备号
86.         unsigned short i_num;              //i 结点号
87.         unsigned short i_count;            //i 结点被使用的次数，0 表示该 i 结点空闲
88.         unsigned char i_lock;              // 锁定标志
89.         unsigned char i_dirt;              // 已修改（脏）标志
90.         unsigned char i_pipe;              // 管道标志
91.         unsigned char i_mount;             // 安装标志
92.         unsigned char i_seek;              // 搜寻标志（lseek 时）
93.         unsigned char i_update;            // 更新标志
94.         };
    // 文件结构（用于在文件句柄与 i 结点之间建立关系）
95.         struct file {
96.         unsigned short f_mode;             // 文件操作模式（RW 位）
97.         unsigned short f_flags;            // 文件打开和控制的标志
98.         unsigned short f_count;            // 对应文件句柄（文件描述符）数
99.         struct m_inode * f_inode;          // 指向对应 i 结点
100.        off_t f_pos;                       // 文件位置（读写偏移值）
101.        };
    // 内存中磁盘超级块结构
102.        struct super_block {
103.        unsigned short s_ninodes;          // 结点数
104.        unsigned short s_nzones;           // 逻辑块数
105.        unsigned short s_imap_blocks;      //i 结点位图所占用的数据块数
106.        unsigned short s_zmap_blocks;      // 逻辑块位图所占用的数据块数
107.        unsigned short s_firstdatazone;    // 第一个数据逻辑块号
108.        unsigned short s_log_zone_size;    //log（数据块数/逻辑块）（以 2 为底）
109.        unsigned long s_max_size;          // 文件最大长度
110.        unsigned short s_magic;            // 文件系统魔数
111.
112.        struct buffer_head * s_imap[8];    //i 结点位图缓冲块指针数组（占用 8 块，可表示 64M）
113.        struct buffer_head * s_zmap[8];    // 逻辑块位图缓冲块指针数组（占用 8 块）
114.        unsigned short s_dev;              // 超级块所在的设备号
115.        struct m_inode * s_isup;           // 被安装的文件系统根目录的 i 结点（isup-super i）
116.        struct m_inode * s_imount;         // 被安装到的 i 结点
117.        unsigned long s_time;              // 修改时间
118.        struct task_struct * s_wait;       // 等待该超级块的进程
119.        unsigned char s_lock;              // 被锁定标志
120.        unsigned char s_rd_only;           // 只读标志
121.        unsigned char s_dirt;              // 已修改（脏）标志
122.        };
```

```
    // 磁盘上超级块结构。上面 102～110 行完全一样
123.        struct d_super_block {
124.        unsigned short s_ninodes;          // 结点数
125.        unsigned short s_nzones;           // 逻辑块数
126.        unsigned short s_imap_blocks;      //i 结点位图所占用的数据块数
127.        unsigned short s_zmap_blocks;      // 逻辑块位图所占用的数据块数
128.        unsigned short s_firstdatazone;    // 第一个数据逻辑块
129.        unsigned short s_log_zone_size;    //log（数据块数/逻辑块）（以 2 为底）
130.        unsigned long s_max_size;          // 文件最大长度
131.        unsigned short s_magic;            // 文件系统魔数
132.        };
    // 文件目录项结构
133.        struct dir_entry {
134.        unsigned short inode;              //i 结点
135.        char name[NAME_LEN];                                    // 文件名
136.        };
137.        extern struct m_inode inode_table[NR_INODE];    // 定义 i 结点表数组（32 项）
138.        extern struct file file_table[NR_FILE];                 // 文件表数组（64 项）
139.        extern struct super_block super_block[NR_SUPER]; // 超级块数组（8 项）
140.        extern struct buffer_head * start_buffer;               // 缓冲区起始内存位置
141.        extern int nr_buffers;                                  // 缓冲块数
    // 磁盘操作函数原型
    // 检测驱动器中软盘是否改变
142.        extern void check_disk_change(int dev);
    // 检测指定软驱中软盘更换情况，如果软盘更换了则返回 1，否则返回 0
143.        extern int floppy_change(unsigned int nr);
    // 设置启动指定驱动器所需等待的时间（设置等待定时器）
144.        extern int ticks_to_floppy_on(unsigned int dev);
    // 启动指定驱动器
145.        extern void floppy_on(unsigned int dev);
    // 关闭指定的软盘驱动器
146.        extern void floppy_off(unsigned int dev);
    // 以下是文件系统操作管理用的函数原型
    // 将 i 结点指定的文件截为 0
147.        extern void truncate(struct m_inode * inode);
    // 刷新 i 结点信息
148.        extern void sync_inodes(void);
    // 等待指定的 i 结点
149.        extern void wait_on(struct m_inode * inode);
    // 逻辑块（区段，磁盘块）位图操作，取数据块 block 在设备上对应的逻辑块号
150.        extern int bmap(struct m_inode * inode,int block);
    // 创建数据块 block 在设备上对应的逻辑块，并返回在设备上的逻辑块号
151.        extern int create_block(struct m_inode * inode,int block);
    // 获取指定路径名的 i 结点号
152.        extern struct m_inode * namei(const char * pathname);
    // 根据路径名为打开文件操作做准备
153.        extern int open_namei(const char * pathname, int flag, int mode,
154.        struct m_inode ** res_inode);
    // 释放一个 i 结点（回写入设备）
155.        extern void iput(struct m_inode * inode);
```

```
    // 从设备读取指定结点号的一个 i 结点
156.        extern struct m_inode * iget(int dev,int nr);
    // 从 i 结点表（inode_table）中获取一个空闲 i 结点项
157.        extern struct m_inode * get_empty_inode(void);
    // 获取（申请一）管道结点，返回为 i 结点指针（如果是 NULL 则失败）
158.        extern struct m_inode * get_pipe_inode(void);
    // 在哈希表中查找指定的数据块，返回找到块的缓冲头指针
159.        extern struct buffer_head * get_hash_table(int dev, int block);
    // 从设备读取指定块（首先会在 hash 表中查找）
160.        extern struct buffer_head * getblk(int dev, int block);
    // 读/写数据块
161.        extern void ll_rw_block(int rw, struct buffer_head * bh);
    // 释放指定缓冲块
162.        extern void brelse(struct buffer_head * buf);
    // 读取指定的数据块
163.        extern struct buffer_head * bread(int dev,int block);
    // 读 4 块缓冲区到指定地址的内存中
164.        extern void bread_page(unsigned long addr,int dev,int b[4]);
    // 读取头一个指定的数据块，并标记后续将要读的块
165.        extern struct buffer_head * breada(int dev,int block,...);
    // 向设备 dev 申请一个磁盘块（区段、逻辑块），返回逻辑块号
166.        extern int new_block(int dev);
    // 释放设备数据区中的逻辑块（区段、磁盘块）block，复位指定逻辑块 block 的逻辑块位图比特位
167.        extern void free_block(int dev, int block);
    // 为设备 dev 建立一个新 i 结点，返回 i 结点号
168.        extern struct m_inode * new_inode(int dev);
    // 释放一个 i 结点（删除文件时）
169.        extern void free_inode(struct m_inode * inode);
    // 刷新指定设备缓冲区
170.        extern int sync_dev(int dev);
    // 读取指定设备的超级块
171.        extern struct super_block * get_super(int dev);
172.        extern int ROOT_DEV;
    // 安装根文件系统
173.        extern void mount_root(void);
174.        #endif
```

代码中第 7～14 行是定义了基本的系统数据类型，系统所含的设备信息等。其中包括内存设备；软盘设备；硬盘设备；串行终端设备；终端设备；打印设备；没有命名的管道；数据块长度；数据块长度所占比特位数；每个逻辑块可存放的 i 结点数；每个逻辑块可存放的目录项数等信息。

3.24 hdreg.h 文件

hdreg.h 是硬盘控制头文件，主要描述了硬盘控制器寄存器端口和硬盘状态寄存器各位的定义。

该文件中主要定义了对硬盘控制器进行编程的一些常量符号。其中包括控制器端口、硬盘状态寄存器各位的状态、控制器命令以及出错状态常量符号。另外还给出了硬盘分区表数据结构。

hdreg.h 文件的详细代码注释如下：

```
1.          #ifndef _HDREG_H
2.          #define _HDREG_H
```

```
/* 硬盘控制器寄存器端口，参见：IBM AT Bios 程序 */
#define HD_DATA 0x1f0
#define HD_ERROR 0x1f1
#define HD_NSECTOR 0x1f2
#define HD_SECTOR 0x1f3
#define HD_LCYL 0x1f4
#define HD_HCYL 0x1f5
#define HD_CURRENT 0x1f6
#define HD_STATUS 0x1f7
#define HD_PRECOMP HD_ERROR
#define HD_COMMAND HD_STATUS
#define HD_CMD 0x3f6                    // 控制寄存器端口
/* 硬盘状态寄存器各位的定义(HD_STATUS) */
#define ERR_STAT 0x01                   // 命令执行错误
#define INDEX_STAT 0x02                 // 收到索引
#define ECC_STAT 0x04 /* Corrected error */ // ECC 校验错
#define DRQ_STAT 0x08                   // 请求服务
#define SEEK_STAT 0x10                  // 寻道结束
#define WRERR_STAT 0x20                 // 驱动器故障
#define READY_STAT 0x40                 // 驱动器准备好（就绪）
#define BUSY_STAT 0x80                  // 控制器忙碌
/* 硬盘命令值（HD_CMD） */
#define WIN_RESTORE 0x10                // 驱动器重新校正（驱动器复位）
#define WIN_READ 0x20                   // 读扇区
#define WIN_WRITE 0x30                  // 写扇区
#define WIN_VERIFY 0x40                 // 扇区检验
#define WIN_FORMAT 0x50                 // 格式化磁道
#define WIN_INIT 0x60                   // 控制器初始化
#define WIN_SEEK 0x70                   // 寻道
#define WIN_DIAGNOSE 0x90               // 控制器诊断
#define WIN_SPECIFY 0x91                // 建立驱动器参数
#define MARK_ERR 0x01
#define TRK0_ERR 0x02
#define ABRT_ERR 0x04
#define ID_ERR 0x10
#define ECC_ERR 0x40
#define BBD_ERR 0x80
// 硬盘分区表结构
struct partition {
unsigned char boot_ind;
unsigned char head;
unsigned char sector;
unsigned char cyl;
unsigned char sys_ind;
unsigned char end_head;
unsigned char end_sector;
unsigned char end_cyl;
unsigned int start_sect;
unsigned int nr_sects;
```

```
48.         };
49.         #endif
```

为了实现多个操作系统共享硬盘资源，硬盘可以在逻辑上分为 1～4 个分区。每个分区之间的扇区号是邻接的。分区表由 4 个表项组成，每个表项由 16 字节组成，对应一个分区的信息，存放有分区的大小和起止的柱面号、磁道号和扇区号。分区表存放在硬盘的 0 柱面 0 头第 1 个扇区的 0x1BE～0x1FD 处。

3.25 head.h 文件

head.h 是段描述符头文件，主要描述定义了段描述符的数据结构和几个选择符常量。

该头文件定义了 Intel CPU 中描述符的简单结构，并且指定描述符的项号。head.h 文件的详细代码注释如下：

```
1.      #ifndef _HEAD_H
2.      #define _HEAD_H
3.      typedef struct desc_struct {          // 定义了段描述符的数据结构，该结构仅说明每个描述
4.      unsigned long a,b;                    // 符是由 8 个字节构成，每个描述符表共有 256 项
5.      } desc_table[256];
6.      extern unsigned long pg_dir[1024];    //内存页目录数组，每个目录项为 4 字节，从物理地址 0 开始
7.      extern desc_table idt,gdt;            // 中断描述符表，全局描述符表
8.      #define GDT_NUL 0                     // 全局描述符表的第 0 项，不用
9.      #define GDT_CODE 1                    // 第 1 项，是内核代码段描述符项
10.     #define GDT_DATA 2                    // 第 2 项，是内核数据段描述符项
11.     #define GDT_TMP 3                     // 第 3 项，系统段描述符，Linux 没有使用
12.     #define LDT_NUL 0                     // 每个局部描述符表的第 0 项，不用
13.     #define LDT_CODE 1                    // 第 1 项，是用户程序代码段描述符项
14.     #define LDT_DATA 2                    // 第 2 项，是用户程序数据段描述符项
15.     #endif
```

代码中第 3～5 行定义了段描述符的数据结构。其中包括中断描述符表、全局描述符表、用户程序代码段描述符项、内核代码段描述符项、内核数据段描述符项。

3.26 kernel.h 文件

kernel.h 是内核头文件，在该文件中主要定义了一些内核常用的函数原型等，所以在内核很多文件中都需要引入该头文件。

3.26.1 功能描述

该头文件定义了一些常用函数的原型，用于验证给定地址开始的内存块是否超限以及显示内核出错信息等一系列功能。

3.26.2 代码注释

kernel.h 文件的详细代码注释如下：

```
/*'kernel.h'定义了一些常用函数的原型等
*/
  // 验证给定地址开始的内存块是否超限，若超限则追加内存（kernel/fork.c)
```

```
1.      void verify_area(void * addr,int count);
    // 显示内核出错信息，然后进入死循环（kernel/panic.c)
2.      volatile void panic(const char * str);
    // 标准打印（显示）函数（init/main.c)
3.      int printf(const char * fmt, ...);
    // 内核专用的打印信息函数，功能与 printf()相同（kernel/printk.c)
4.      int printk(const char * fmt, ...);
    // 往 tty 上写指定长度的字符串（kernel/chr_drv/tty_io.c)
5.      int tty_write(unsigned ch,char * buf,int count);
    // 通用内核内存分配函数（lib/malloc.c)
6.      void * malloc(unsigned int size);
    // 释放指定对象占用的内存（lib/malloc.c)
7.      void free_s(void * obj, int size);
8.      #define free(x) free_s((x), 0)
```

代码中第 1～7 行定义了一些常用函数的原型。其中包括验证给定地址开始的内存块是否超限；显示内核出错信息；标准打印（显示）函数；内核专用的打印信息函数；往 tty 上写指定长度的字符串；通用内核内存分配函数；释放指定对象占用的内存。

3.27 mm.h 文件

mm.h 是内存头文件，含有页面大小定义和一些页面释放函数原型的定义。所以在内核内存管理的很多页面中都需要引入该头文件。mm.h 头文件是内存管理的核心文件之一，详细代码注释如下：

```
1.      #ifndef _MM_H
2.      #define _MM_H
3.      #define PAGE_SIZE 4096                          //定义内存页面大小为 4096 字节
4.      #include <linux/fs.h>                           //文件系统头文件，定义了文件系统常用的函数原型
5.      #include <linux/kernel.h>                       //内核头文件，定义了一些内核常用函数的原形
6.      #include <signal.h>                             //信号头文件，定义了内核中使用的所有信号的名称
                                                        //和基本操作函数
7.      extern unsigned int swap_device;                //定义交换块设备
8.      extern struct inode * swap_file;                //定义交换文件
9.      extern void rw_swap_page(int rw, unsigned int nr, char * buf);      //swap.c 中的读写函数
10.     #define read_swap_page(nr,buf)
11.     rw_swap_page(READ,(nr),(buf))
12.     #define write_swap_page(nr,buf)
13.     rw_swap_page(WRITE,(nr),(buf))
14.     extern unsigned long get_free_page(void); //扫描页面映射数组 mem_map[]取空闲页面函数
15.     extern unsigned long put_dirty_page(unsigned long page,unsigned long address);//在页目录和页表
                                                                //中指定位置处放置指定页面信息
16.     extern void free_page(unsigned long addr); //释放物理地址 addr 开始的页面内存，并修改引用页
                                                   //面次数
17.     void swap_free(int page_nr);               //释放交换设备的页
18.     void swap_in(unsigned long *table_ptr);    //将交换设备中的页读入物理内存函数
19.                                                //显示内存已用完出错信息，并退出
20.     extern inline volatile void oom(void)
21.     {
22.     printk("out of memory\n\r");
```

```
23.     do_exit(SIGSEGV);// do_exit()应该使用退出代码，这里用了信号值 SIGSEGV
24.     }
25.     #define invalidate()
26.     __asm__("movl %%eax,%%cr3"::"a" (0))
27.
28.     #define LOW_MEM 0x100000                    // 内存低端（1MB）
29.     extern unsigned long HIGH_MEMORY;           //全局变量，放实际物理内存最高端地址
30.     #define PAGING_MEMORY (15*1024*1024)   //分页内存 15MB，主内存区最多 15MB
31.     #define PAGING_PAGES (PAGING_MEMORY>>12)//分页后的物理内存页数
32.     #define MAP_NR(addr) (((addr)-LOW_MEM)>>12)  //指定物理内存地址映射为页号
33.     #define USED 100                                    //页面被占用标志
34.   // 内存映射字节图，每个页面对应的字节，用于标志页面当前被占用次数
35.
36.     extern unsigned char mem_map [ PAGING_PAGES ];
37.     #define PAGE_DIRTY    0x40//若置位，表明该页面的内容自从上次该位被清除后已被修改过
38.     #define PAGE_ACCESSED    0x20//若置位，表示该页面最近曾被访问过
39.     #define PAGE_USER        0x04//置位表示某页面是用户空间页面，清空表示为内核空间页面
40.     #define PAGE_RW          0x02//页是可读写的，置 0 只是可读的，置 1 表示可读可写的
41.     #define PAGE_PRESENT     0x01//当前页，如果置位，则该物理页在 RAM 中
42.     #endif
```

mm.h 头文件内存管理，含有页面大小定义和一些页面释放函数原型的定义。

3.28 sched.h 文件

sched.h 是调度头文件，在其中定义了任务结构 task_struct 以及初始任务 0 的数据。

该头文件主要用于一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。sched.h 文件的详细代码注释如下：

```
1.    #ifndef _SCHED_H
2.    #define _SCHED_H
3.    #define NR_TASKS 64        // 系统中同时最多任务（进程）数
4.    #define HZ 100             // 定义系统时钟滴答频率（100Hz，每个滴答 10ms）
5.    #define FIRST_TASK task[0]      // 任务 0 比较特殊，所以特意给它单独定义一个符号
6.    #define LAST_TASK task[NR_TASKS-1] // 任务数组中的最后一项任务
7.    #include <linux/head.h>            // head 头文件，定义了段描述符的简单结构和几个选择符常量
8.    #include <linux/fs.h>              // 文件系统头文件，定义文件表结构（file,buffer_head,m_inode 等）
9.    #include <linux/mm.h>              // 内存管理头文件。含有页面大小定义和一些页面释放函数原型
10.   #include <signal.h>          // 信号头文件，定义信号符号常量，信号结构及信号操作函数原型
11.   #if (NR_OPEN > 32)
12.   #error "Currently the close-on-exec-flags are in one word, max 32 files/proc"
13.      #endif
14.   // 这里定义了进程运行可能处于的状态
15.   #define TASK_RUNNING 0         // 进程正在运行或已准备就绪
16.   #define TASK_INTERRUPTIBLE 1 // 进程处于可中断等待状态
17.      #define TASK_UNINTERRUPTIBLE 2 // 进程处于不可中断等待状态，主要用于 I/O 操作等待
18.   #define TASK_ZOMBIE 3          // 进程处于僵死状态，已经停止运行，但父进程还没发信号
19.   #define TASK_STOPPED 4         // 进程已停止
20.      #ifndef NULL
21.      #define NULL ((void *) 0)      // 定义 NULL 为空指针
22.      #endif
```

```
23.// 复制进程的页目录页表。Linus 认为这是内核中最复杂的函数之一（mm/memory.c）
24.    extern int copy_page_tables(unsigned long from, unsigned long to, long size);
25.    // 释放页表所指定的内存块及页表本身（mm/memory.c）
26.    extern int free_page_tables(unsigned long from, unsigned long size);
27.// 调度程序的初始化函数（kernel/sched.c）
28.        extern void sched_init(void);
29.    // 进程调度函数（kernel/sched.c）
30.    extern void schedule(void);
31.// 异常（陷阱）中断处理初始化函数，设置中断调用门并允许中断请求信号（kernel/traps.c）
32.        extern void trap_init(void);
33.    // 显示内核出错信息，然后进入死循环（kernel/panic.c）
34.    extern void panic(const char * str);
35.// 往 tty 上写指定长度的字符串（kernel/chr_drv/tty_io.c）
36.        extern int tty_write(unsigned minor,char * buf,int count);
37.    typedef int (*fn_ptr)(); // 定义函数指针类型
38.// 下面是数学协处理器使用的结构，主要用于保存进程切换时 i387 的执行状态信息
39.        struct i387_struct {
40.        long cwd;                    // 控制字（Control word）
41.        long swd;                    // 状态字（Status word）
42.        long twd;                    // 标记字（Tag word）
43.        long fip;                    // 协处理器代码指针
44.        long fcs;                    // 协处理器代码段寄存器
45.        long foo;                    // 内存操作数的偏移位置
46.        long fos;                        // 内存操作数的段值
47.        long st_space[20];           // 8 个 10 字节的协处理器累加器
48.        };
49.    // 任务状态段数据结构
50.    struct tss_struct {
51.    long back_link;
52.        long esp0;
53.        long ss0;
54.        long esp1;
55.        long ss1;
56.        long esp2;
57.        long ss2;
58.        long cr3;
59.        long eip;
60.        long eflags;
61.        long eax,ecx,edx,ebx;
62.        long esp;
63.        long ebp;
64.        long esi;
65.        long edi;
66.        long es;
67.        long cs;
68.        long ss;
69.        long ds;
70.        long fs;
71.        long gs;
72.        long ldt;
```

```
73.      long trace_bitmap;
74.      struct i387_struct i387;
75.      };
76.      struct task_struct {
77.
78.      long state;
79.       long counter;
80.       long priority;
81.       long signal;
82.      struct sigaction sigaction[32];
83.      long blocked;
84.
85.       int exit_code;
86.       unsigned long start_code,end_code,end_data,brk,start_stack;
87.      long pid,father,pgrp,session,leader;
88.      unsigned short uid,euid,suid;
89.       unsigned short gid,egid,sgid;
90.       long alarm;
91.       long utime,stime,cutime,cstime,start_time;
92.      unsigned short used_math;
93.
94.       int tty;
95.       unsigned short umask;
96.      struct m_inode * pwd;
97.      struct m_inode * root;
98.      struct m_inode * executable;
99.      unsigned long close_on_exec;
100.     struct file * filp[NR_OPEN];
101.
102.    struct desc_struct ldt[3];
103.
104.     struct tss_struct tss;
105.     };
106.
107.
108.
109.
110. // 对应上面任务结构的第 1 个任务的信息
111.     #define INIT_TASK
112.     /* state etc */ { 0,15,15,                    // state, counter, priority
113.     /* signals */ 0,{{},},0,                     // signal, sigaction[32], blocked
114.     /* ec,brk... */ 0,0,0,0,0,0,                 // exit_code,start_code,end_code,end_data,brk,start_stack
115.      /* pid etc.. */ 0,-1,0,0,0,                 // pid, father, pgrp, session, leader
116.     /* uid etc */ 0,0,0,0,0,0,0,                 // uid, euid, suid, gid, egid, sgid
117.     /* alarm */ 0,0,0,0,0,0,0,                   // alarm, utime, stime, cutime, cstime, start_time
118.      /* math */ 0,                               // used_math
119.     /* fs info */ -1,0022,NULL,NULL,NULL,0, \ // tty,umask,pwd,root,executable,close_on_exec
120.     /* filp */ {NULL,},                          // filp[20]
121.      {                                          // ldt[3]
122.      {0,0},
```

```
123.     /* ldt */ {0x9f,0xc0fa00},        // 代码长 640K，基址 0x0，G=1，D=1，DPL=3，P=1 TYPE=0x0a
124.     {0x9f,0xc0f200},                  // 数据长 640K，基址 0x0，G=1，D=1，DPL=3，P=1 TYPE=0x02
125.     },
126.     /*tss*/ {0,PAGE_SIZE+(long)&init_task,0x10,0,0,0,0,(long)&pg_dir, // tss
127.     0,0,0,0,0,0,0,0,
128.     0,0,0x17,0x17,0x17,0x17,0x17,0x17,
129.     _LDT(0),0x80000000,
130.     {}
131.     }
132.     }
133.     extern struct task_struct *task[NR_TASKS];      // 任务数组
134.     extern struct task_struct *last_task_used_math;  // 上一个使用过协处理器的进程
135.     extern struct task_struct *current;              // 当前进程结构指针变量
136.     extern long volatile jiffies;                    // 从开机开始算起的滴答数（10ms/滴答）
137.     extern long startup_time;                // 开机时间，从 1970:0:0:0 开始计时的秒数
138.     #define CURRENT_TIME (startup_time+jiffies/HZ) // 当前时间（秒数）
139.  // 添加定时器函数（定时时间 jiffies 滴答数，定时到时调用函数*fn()）（kernel/sched.c）
140.     extern void add_timer(long jiffies, void (*fn)(void));
141.  // 不可中断的等待睡眠（kernel/sched.c）
142.     extern void sleep_on(struct task_struct ** p);
143.  // 可中断的等待睡眠（kernel/sched.c）
144.     extern void interruptible_sleep_on(struct task_struct ** p);
145.  // 明确唤醒睡眠的进程（kernel/sched.c）
146.     extern void wake_up(struct task_struct ** p);
147.  // 全局表中第 1 个任务状态段（TSS）描述符的选择符索引号
148.     #define FIRST_TSS_ENTRY 4
149.  // 全局表中第 1 个局部描述符表（LDT）描述符的选择符索引号
150.     #define FIRST_LDT_ENTRY (FIRST_TSS_ENTRY+1)
151.  // 宏定义，计算在全局表中第 n 个任务的 TSS 描述符的索引号（选择符）
152.     #define _TSS(n) ((((unsigned long) n)<<4)+(FIRST_TSS_ENTRY<<3))
153.  // 宏定义，计算在全局表中第 n 个任务的 LDT 描述符的索引号
154.     #define _LDT(n) ((((unsigned long) n)<<4)+(FIRST_LDT_ENTRY<<3))
155.  // 宏定义，加载第 n 个任务的任务寄存器 tr
156.     #define ltr(n) __asm__("ltr %%ax"::"a" (_TSS(n)))
157.  // 宏定义，加载第 n 个任务的局部描述符表寄存器 ldt
158.     #define lldt(n) __asm__("lldt %%ax"::"a" (_LDT(n)))
159.  // 取当前运行任务的任务号（是任务数组中的索引值，与进程号 pid 不同）
160.  // 返回：n-当前任务号。用于（kernel/traps.c）
161.     #define str(n)
162.     __asm__("str %%ax\n\t"                  // 将任务寄存器中 TSS 段的有效地址??ax
163.     "subl %2,%%eax\n\t"                     // (eax - FIRST_TSS_ENTRY*8)??eax
164.     "shrl $4,%%eax"                         // (eax/16)??eax = 当前任务号
165.     :"=a" (n)
166.     :"a" (0),"i" (FIRST_TSS_ENTRY<<3))
167.     #define switch_to(n) {
168.     struct {long a,b;} __tmp;
169.     __asm__("cmpl %%ecx,_current\n\t" // 任务 n 是当前任务吗
170.     "je 1f\n\t"                             // 是，则什么都不做，退出
171.     "movw %%dx,%1\n\t"                      // 将新任务 16 位选择符存入__tmp.b 中
172.     "xchgl %%ecx,_current\n\t"              // current = task[n]；ecx = 被切换出的任务
```

```
173.    "ljmp %0\n\t"                           // 执行长跳转至*&__tmp，造成任务切换
174.  // 在任务切换回来后才会继续执行下面的语句
175.    "cmpl %%ecx,_last_task_used_math\n\t"       // 新任务上次使用过协处理器吗
176.    "jne 1f\n\t"                            // 没有则跳转，退出
177.    "clts\n"                                // 新任务上次使用过协处理器，则清 cr0 的 TS 标志
178.    "1:"
179.    ::"m" (*&__tmp.a),"m" (*&__tmp.b),
180.    "d" (_TSS(n)),"c" ((long) task[n]));
181.    }
182.  // 页面地址对准（在内核代码中没有任何地方引用!）
183.    #define PAGE_ALIGN(n) (((n)+0xfff)&0xfffff000)
184.
185.  // 设置位于地址 addr 处描述符中的各基地址字段（基地址是 base）
186.  // %0 - 地址 addr 偏移 2；%1 - 地址 addr 偏移 4；%2 - 地址 addr 偏移 7；edx - 基地址 base
187.    #define _set_base(addr,base)
188.    __asm__("movw %%dx,%0\n\t"              // 基址 base 低 16 位(位 15-0)??[addr+2]
189.    "rorl $16,%%edx\n\t"                    // edx 中基址高 16 位(位 31-16)??dx
190.    "movb %%dl,%1\n\t"                      // 基址高 16 位中的低 8 位(位 23-16)??[addr+4]
191.    "movb %%dh,%2"                          // 基址高 16 位中的高 8 位(位 31-24)??[addr+7]
192.    ::"m" (*((addr)+2)),
193.    "m" (*((addr)+4)),
194.    "m" (*((addr)+7)),
195.    "d" (base)
196.    :"dx")
197.    #define _set_limit(addr,limit)
198.    __asm__("movw %%dx,%0\n\t"              // 段长 limit 低 16 位(位 15-0)??[addr]
199.    "rorl $16,%%edx\n\t"                    // edx 中的段长高 4 位(位 19-16)??dl
200.    "movb %1,%%dh\n\t"                      // 取原[addr+6]字节??dh，其中高 4 位是些标志
201.    "andb $0xf0,%%dh\n\t"                   // 清 dh 的低 4 位（将存放段长的位 19-16）
202.    "orb %%dh,%%dl\n\t"                     // 将原高 4 位标志和段长的高 4 位（位 19-16）合成 1 字节，
203.    "movb %%dl,%1"                          // 并放回[addr+6]处
204.    ::"m" (*(addr)),
205.    "m" (*((addr)+6)),
206.    "d" (limit)
207.    :"dx")
208.  // 设置局部描述符表中 ldt 描述符的基地址字段
209.    #define set_base(ldt,base) _set_base( ((char *)&(ldt)) , base )
210.  // 设置局部描述符表中 ldt 描述符的段长字段
211.    #define set_limit(ldt,limit) _set_limit( ((char *)&(ldt)) , (limit-1)>>12 )
212.  // 从地址 addr 处描述符中取段基地址。功能与_set_base()正好相反
213.  // edx - 存放基地址(__base)；%1 - 地址 addr 偏移 2；%2 - 地址 addr 偏移 4；%3 - addr 偏移 7
214.    #define _get_base(addr) ({
215.    unsigned long __base;
216.    __asm__("movb %3,%%dh\n\t"              // 取[addr+7]处基址高 16 位的高 8 位(位 31-24)??dh
217.    "movb %2,%%dl\n\t"                      // 取[addr+4]处基址高 16 位的低 8 位(位 23-16)??dl
218.    "shll $16,%%edx\n\t"                    // 基地址高 16 位移到 edx 中高 16 位处
219.    "movw %1,%%dx"                          // 取[addr+2]处基址低 16 位(位 15-0)??dx
220.    :"=d" (__base)                          // 从而 edx 中含有 32 位的段基地址
221.    :"m" (*((addr)+2)),
222.    "m" (*((addr)+4)),
```

```
223.      "m" (*((addr)+7)));
224.      __base;})
225.  // 取局部描述符表中 ldt 所指段描述符中的基地址
226.     #define get_base(ldt) _get_base( ((char *)&(ldt)) )
227.     #define get_limit(segment) ({
228.      unsigned long __limit;
229.      __asm__("lsll %1,%0\n\tincl %0":"=r" (__limit):"r" (segment));
230.      __limit;})
231.     #endif
```

CPU 管理任务需要的所有信息被存储于一个特殊类型的段中，即任务状态段（task state segment，TSS）。TSS 中的字段可以分为两类：

（1）CPU 在进行任务切换时更新的动态信息集。这些字段有：

通用寄存器（EAX，ECX，EDX，EBX，ESP，EBP，ESI，EDI）；

段寄存器（ES, CS, SS, DS, FS, GS）；

标志寄存器（EFLAGS）；

指令指针（EIP）；

前一个执行任务的 TSS 的选择符（仅当返回时才更新）。

（2）CPU 读取但不会更改的静态信息集。这些字段有：任务的 LDT 的选择符；含有任务页目录基地址的寄存器（PDBR）；特权级 0～2 的堆栈指针；当任务进行切换时导致 CPU 产生一个调试（debug）异常的 T-比特位（调试跟踪位）；I/O 比特位图基地址（其长度上限就是 TSS 的长度上限，在 TSS 描述符中说明）。任务状态段可以存放在线形空间的任何地方。与其他各类段相似，任务状态段也是由描述符来定义的。当前正在执行任务的 TSS 是由任务寄存器（TR）来指示的。指令 LTR 和 STR 用来修改和读取任务寄存器中的选择符（任务寄存器的可见部分）。I/O 比特位图中的每 1 比特对应 1 个 I/O 端口。比如端口 41 的比特位就是 I/O 位图基地址+5，位偏移 1 处。在保护模式中，当遇到 1 个 I/O 指令时（IN, INS, OUT, OUTS），CPU 首先就会检查当前特权级是否小于标志寄存器的 IOPL，如果这个条件满足，就执行该 I/O 操作。如果不满足，那么 CPU 就会检查 TSS 中的 I/O 比特位图。如果相应比特位是置位的，就会产生一般保护性异常，否则就会执行该 I/O 操作。

3.29 sys.h 文件

sys.h 是系统头文件，定义了内核中所有系统调用函数的原型。所以在很多内核文件中都需要引用该头文件。sys.h 头文件列出了内核中所有系统调用函数的原型，以及系统调用函数指针表。

sys.h 文件的详细代码注释如下：

```
1.    extern int sys_setup();          // 系统启动初始化设置函数（kernel/blk_drv/hd.c）
2.    extern int sys_exit();           // 程序退出（kernel/exit.c）
3.    extern int sys_fork();           // 创建进程（kernel/system_call.s）
4.    extern int sys_read();           // 读文件（fs/read_write.c）
5.    extern int sys_write();          // 写文件（fs/read_write.c）
6.    extern int sys_open();           // 打开文件（fs/open.c）
7.    extern int sys_close();          // 关闭文件（fs/open.c）
8.    extern int sys_waitpid();        // 等待进程终止（kernel/exit.c）
9.    extern int sys_creat();          // 创建文件（fs/open.c）
10.   extern int sys_link();           // 创建一个文件的硬连接（fs/namei.c）
```

```
11.     extern int sys_unlink();        // 删除一个文件名（或删除文件）（fs/namei.c）
12.     extern int sys_execve();        // 执行程序（kernel/system_call.s）
13.     extern int sys_chdir();         // 更改当前目录（fs/open.c）
14.     extern int sys_time();          // 取当前时间（kernel/sys.c）
15.     extern int sys_mknod();         // 建立块/字符特殊文件（fs/namei.c）
16.     extern int sys_chmod();         // 修改文件属性（fs/open.c）
17.     extern int sys_chown();         // 修改文件宿主和所属组（fs/open.c）
18.     extern int sys_break();         //（-kernel/sys.c）
19.     extern int sys_stat();          // 使用路径名取文件的状态信息（fs/stat.c）
20.     extern int sys_lseek();         // 重新定位读/写文件偏移（fs/read_write.c）
21.     extern int sys_getpid();        // 取进程 id（kernel/sched.c）
22.     extern int sys_mount();         // 安装文件系统（fs/super.c）
23.     extern int sys_umount();        // 卸载文件系统（fs/super.c）
24.     extern int sys_setuid();        // 设置进程用户 id（kernel/sys.c）
25.     extern int sys_getuid();        // 取进程用户 id（kernel/sched.c）
26.     extern int sys_stime();         // 设置系统时间日期（-kernel/sys.c）
27.     extern int sys_ptrace();        // 程序调试（-kernel/sys.c）
28.     extern int sys_alarm();         // 设置报警（kernel/sched.c）
29.     extern int sys_fstat();         // 使用文件句柄取文件的状态信息（fs/stat.c）
30.     extern int sys_pause();         // 暂停进程运行（kernel/sched.c）
31.     extern int sys_utime();         // 改变文件的访问和修改时间（fs/open.c）
32.     extern int sys_stty();          // 修改终端行设置（-kernel/sys.c）
33.     extern int sys_gtty();          // 取终端行设置信息（-kernel/sys.c）
34.     extern int sys_access();        // 检查用户对一个文件的访问权限（fs/open.c）
35.     extern int sys_nice();          // 设置进程执行优先权（kernel/sched.c）
36.     extern int sys_ftime();         // 取日期和时间（-kernel/sys c）
37.     extern int sys_sync();          // 同步高速缓冲与设备中数据（fs/buffer.c）
38.     extern int sys_kill();          // 终止一个进程（kernel/exit.c）
39.     extern int sys_rename();        // 更改文件名（kernel/sys.c）
40.     extern int sys_mkdir();         // 创建目录（fs/namei.c）
41.     extern int sys_rmdir();         // 删除目录（fs/namei.c）
42.     extern int sys_dup();           // 复制文件句柄（fs/fcntl.c）
43.     extern int sys_pipe();          // 创建管道（fs/pipe.c）
44.     extern int sys_times();         // 取运行时间（kernel/sys.c）
45.     extern int sys_prof();          // 程序执行时间区域（-kernel/sys.c）
46.     extern int sys_brk();           // 修改数据段长度（kernel/sys.c）
47.     extern int sys_setgid();        // 设置进程组 id（kernel/sys.c）
48.     extern int sys_getgid();        // 取进程组 id（kernel/sched.c）
49.     extern int sys_signal();        // 信号处理（kernel/signal.c）
50.     extern int sys_geteuid();       // 取进程有效用户 id（kenrl/sched.c）
51.     extern int sys_getegid();       // 取进程有效组 id（kenrl/sched.c）
52.     extern int sys_acct();          // 进程记帐（-kernel/sys.c）
53.     extern int sys_phys();          //（-kernel/sys.c）
54.     extern int sys_lock();          //（-kernel/sys.c）
55.     extern int sys_ioctl();         // 设备控制（fs/ioctl.c）
56.     extern int sys_fcntl();         // 文件句柄操作（fs/fcntl.c）
57.     extern int sys_mpx();           //（-kernel/sys.c）
58.     extern int sys_setpgid();       // 设置进程组 id（kernel/sys.c）
59.     extern int sys_ulimit();        //（-kernel/sys.c）
60.     extern int sys_uname();         // 显示系统信息（kernel/sys.c）
```

```
61.     extern int sys_umask();          // 取默认文件创建属性码（kernel/sys.c）
62.     extern int sys_chroot();         // 改变根系统（fs/open.c）
63.     extern int sys_ustat();          // 取文件系统信息（fs/open.c）
64.     extern int sys_dup2();           // 复制文件句柄（fs/fcntl.c）
65.     extern int sys_getppid();        // 取父进程 id（kernel/sched.c）
66.     extern int sys_getpgrp();        // 取进程组 id，等于 getpgid(0)（kernel/sys.c）
67.     extern int sys_setsid();         // 在新会话中运行程序（kernel/sys.c）
68.     extern int sys_sigaction();      // 改变信号处理过程（kernel/signal.c）
69.     extern int sys_sgetmask();       // 取信号屏蔽码（kernel/signal.c）
70.     extern int sys_ssetmask();       // 设置信号屏蔽码（kernel/signal.c）
71.     extern int sys_setreuid();       // 设置真实与/或有效用户 id（kernel/sys.c）
72.     extern int sys_setregid();       // 设置真实与/或有效组 id（kernel/sys.c）
73.
74.  // 系统调用函数指针表。用于系统调用中断处理程序（int 0x80），作为跳转表
75.     fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
76.     sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
77.     sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
78.     sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
79.     sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
80.     sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
81.     sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
82.     sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
83.     sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
84.     sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
85.     sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
86.     sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
87.     sys_setreuid,sys_setregid };
```

代码中第 75～87 行是定义了系统调用函数指针表。用于系统调用中断处理程序，作为跳转表。其中包括系统启动初始化设置函数；程序退出；创建进程；读文件；写文件；打开文件；关闭文件；等待进程终止；创建文件；创建一个文件的硬连接；删除一个文件名；执行程序；更改当前目录；取当前时间；建立块/字符特殊文件；修改文件属性；修改文件宿主和所属组；使用路径名取文件的状态信息；重新定位读/写文件偏移；取进程 id 等。

3.30 tty.h 文件

tty.h 是终端头文件，定义了终端以及串行通信的数据结构和常量。所以在设备管理的文件中经常需要引入该头文件。该文件中包含 tty 类型终端的数据结构和常量定义，通过这些数据结构和常量可以定义各种类型的终端。tty.h 文件的详细代码注释如下：

```
1.      /*
2.       * 'tty.h'中定义了 tty_io.c 程序使用的某些结构和其他一些定义
3.       * 注意！在修改这里的定义时，一定要保证 rs_io.s 或 con_io.s 程序中不会出现问题
4.    * 在系统中有些常量是直接写在程序中的（主要是一些 tty_queue 中的偏移值）
5.    */
6.          #ifndef _TTY_H
7.          #define _TTY_H
8.          #include <termios.h>   // 终端输入输出函数头文件。主要定义控制异步通信口的终端接口
9.          #define TTY_BUF_SIZE 1024 // tty 缓冲区（缓冲队列）大小
10.   // tty 等待队列数据结构。用于 tty_struc 结构中的读、写和辅助（规范）缓冲队列
```

```
11.     struct tty_queue {
12.         unsigned long data;          // 队列缓冲区中含有字符行数值（不是当前字符数）
13.                                      // 对于串口终端，则存放串行端口地址
14.         unsigned long head;          // 缓冲区中数据头指针
15.     unsigned long tail;              // 缓冲区中数据尾指针
16.     struct task_struct * proc_list;  // 等待进程列表
17.         char buf[TTY_BUF_SIZE]; // 队列的缓冲区
18.         };
19.     // 以下定义了 tty 等待队列中缓冲区操作宏函数。（tail 在前，head 在后，参见 tty_io.c 文件）
20.  // a 缓冲区指针前移 1 字节，若已超出缓冲区右侧，则指针循环
21.     #define INC(a) ((a) = ((a)+1) & (TTY_BUF_SIZE-1))
22.     // a 缓冲区指针后退 1 字节，并循环。
23.         #define DEC(a) ((a) = ((a)-1) & (TTY_BUF_SIZE-1))
24.  // 清空指定队列的缓冲区
25.     #define EMPTY(a) ((a).head == (a).tail)
26.     // 缓冲区还可存放字符的长度（空闲区长度）
27.         #define LEFT(a) (((a).tail-(a).head-1)&(TTY_BUF_SIZE-1))
28.  // 缓冲区中最后一个位置
29.     #define LAST(a) ((a).buf[(TTY_BUF_SIZE-1)&((a).head-1)])
30.     // 缓冲区满（如果为 1 的话）
31.         #define FULL(a) (!LEFT(a))
32.  // 缓冲区中已存放字符的长度
33.     #define CHARS(a) (((a).head-(a).tail)&(TTY_BUF_SIZE-1))
34.     // 从 queue 队列项缓冲区中取一字符（从 tail 处，并且 tail+=1）
35.     #define GETCH(queue,c)
36.     (void)({c=(queue).buf[(queue).tail];INC((queue).tail);})
37.     // 往 queue 队列项缓冲区中放置一字符（在 head 处，并且 head+=1）
38.     #define PUTCH(c,queue)
39.     (void)({(queue).buf[(queue).head]=(c);INC((queue).head);})
40.     // 判断终端键盘字符类型
41.         #define INTR_CHAR(tty) ((tty)->termios.c_cc[VINTR])        // 中断符
42.     #define QUIT_CHAR(tty) ((tty)->termios.c_cc[VQUIT])            // 退出符
43.     #define ERASE_CHAR(tty) ((tty)->termios.c_cc[VERASE])          // 消除符
44.         #define KILL_CHAR(tty) ((tty)->termios.c_cc[VKILL])        // 终止符
45.     #define EOF_CHAR(tty) ((tty)->termios.c_cc[VEOF])              // 文件结束符
46.     #define START_CHAR(tty) ((tty)->termios.c_cc[VSTART])          // 开始符
47.         #define STOP_CHAR(tty) ((tty)->termios.c_cc[VSTOP])        // 停止符
48.     #define SUSPEND_CHAR(tty) ((tty)->termios.c_cc[VSUSP])         // 挂起符
49.  // tty 数据结构
50.         struct tty_struct {
51.         struct termios termios;                  // 终端 io 属性和控制字符数据结构
52.         int pgrp;                                // 所属进程组
53.     int stopped;                                 // 停止标志
54.     void (*write)(struct tty_struct * tty);      // tty 写函数指针
55.         struct tty_queue read_q;                 // tty 读队列
56.         struct tty_queue write_q;                // tty 写队列
57.     struct tty_queue secondary;                  // tty 辅助队列（存放规范模式字符序列）
58.     };                                           // 可称为规范(熟)模式队列
59.         extern struct tty_struct tty_table[];    // tty 结构数组
60.         /* intr=^C quit=^| erase=del kill=^U
```

```
61.      eof=^D vtime=\0 vmin=\1 sxtc=\0
62.      start=^Q stop=^S susp=^Z eol=\0
63.         reprint=^R discard=^U werase=^W lnext=^V
64.         eol2=\0
65.         */
66.      /* 中断 intr=^C 退出 quit=^| 删除 erase=del 终止 kill=^U
67.   * 文件结束 eof=^D vtime=\0 vmin=\1 sxtc=\0
68.   * 开始 start=^Q 停止 stop=^S 挂起 susp=^Z 行结束 eol=\0
69.      * 重显 reprint=^R 丢弃 discard=^U werase=^W lnext=^V
70.      * 行结束 eol2=\0
71.   */
72.   // 控制字符对应的 ASCII 码值。[8 进制]
73.      #define INIT_C_CC "\003\034\177\025\004\0\1\0\021\023\032\0\022\017\027\026\0"
74.      void rs_init(void);                                    // 异步串行通信初始化（kernel/chr_drv/serial.c）
75.      void con_init(void);                                   // 控制终端初始化（kernel/chr_drv/console.c）
76.      void tty_init(void);                                // tty 初始化（kernel/chr_drv/tty_io.c）
77.      int tty_read(unsigned c, char * buf, int n);           // （kernel/chr_drv/tty_io.c）
78.      int tty_write(unsigned c, char * buf, int n);                // （kernel/chr_drv/tty_io.c）
79.      void rs_write(struct tty_struct * tty);             // （kernel/chr_drv/serial.c）
80.      void con_write(struct tty_struct * tty);               // （kernel/chr_drv/console.c）
81.      void copy_to_cooked(struct tty_struct * tty);       // （kernel/chr_drv/tty_io.c）
82.      #endif
```


代码中第 17～24 行是定义终端输入输出函数头文件。主要定义控制异步通信口的终端接口和缓冲区（缓冲队列）大小。其中包括队列缓冲区中含有字符行数值；串行端口地址；缓冲区中数据头指针；缓冲区中数据尾指针；等待进程列表；队列的缓冲区等信息。

3.31 math_emu.h 文件

该文件是数学仿真头文件，在其中定义了异常中断处理过程入栈的寄存器。

3.31.1 功能描述

math_emulate.c 程序中的所有函数可分为 3 部分：第一类是设备不存在异常处理程序接口函数 math_emulate()，只有这一个函数；第二类是浮点指令仿真处理主函数 do_emu()，也只有一个函数；另外所有函数都是仿真运算辅助类函数，包括其余几个 C 语言程序中的函数。

在一台不包含 80387 协处理器芯片的 PC 中，如果内核初始化时在 CR0 中设置了仿真标志 EM = 1，那么当 CPU 遇到一条浮点指令时就会引起 CPU 产生异常中断 int 7，并且在该中断处理过程中调用本程序中的 math_emulate(long _false)函数。

在 math_emulate()函数中，若判断出当前进程还没有使用过仿真的协处理运算时就会对仿真的 80387 控制字、状态字和特征字（Tag Word）进行初始化操作，设置控制字中所有 6 种协处理器异常屏蔽位并复位状态字和特征字。然后调用仿真处理主函数 do_emu()。使用的参数是作为 info 结构的中断处理过程中调用 math_emulate()函数的返回地址指针。info 结构实际上就是栈中自从 CPU 产生中断 int7 后逐渐入栈的一些数据构成的一个结构，因此它与系统调用时内核栈中数据的分布情况基本相同。

3.21.2 代码注释

math_emu.h 文件的详细代码注释如下：

```
1.  /*
2.   * linux/include/linux/math_emu.h
3.  #ifndef _LINUX_MATH_EMU_H
4.  #define _LINUX_MATH_EMU_H
5.  #include <linux/sched.h>
6.  struct info {
7.   long ___orig_eip;                  // math_emulate()调用者（int 7）返回地址
8.   long ___ret_from_system_call;      // 临时保存原 EIP 的地方
9.   long ___ebx;                       // 异常中断 int 7 处理过程入栈的寄存器
10.  long ___ecx;
11.  long ___edx;
12.  long ___esi;                       // 中断 int 7 返回时将去执行系统调用的返回处理代码
13.  long ___edi;                       // 以下部分（13～25）与系统调用时栈中结构相同
14.  long ___ebp;
15.  long ___eax;
16.  long ___ds;
17.  long ___es;                        // 如不是系统调用而是其他中断时，该值为-1
18.  long ___fs;
19.  long ___gs;
20.  long ___orig_eax;
21.  long ___eip;                       // 21～26 由 CPU 自动入栈
22.  long ___cs;
23.  long ___eflags;
24.  long ___esp;
25.  long ___ss;
26.  };
27.  #define EAX (info->___eax)
28.  #define EBX (info->___ebx)
29.  #define ECX (info->___ecx)
30.  #define EDX (info->___edx)
31.  #define ESI (info->___esi)
32.  #define EDI (info->___edi)
33.  #define EBP (info->___ebp)
34.  #define ESP (info->___esp)
35.  #define EIP (info->___eip)
36.  #define ORIG_EIP (info->___orig_eip)
37.  #define EFLAGS (info->___eflags)
38.  #define DS (*(unsigned short *) &(info->___ds))
39.  #define ES (*(unsigned short *) &(info->___es))
40.  #define FS (*(unsigned short *) &(info->___fs))
41.  #define CS (*(unsigned short *) &(info->___cs))
42.  #define SS (*(unsigned short *) &(info->___ss))
43.  void __math_abort(struct info *, unsigned int);
44.  #define math_abort(x,y) \
45.  (((volatile void (*)(struct info *,unsigned int)) __math_abort)((x),(y)))
46.  typedef struct {
47.   long a,b;
48.   short exponent;
49.  } temp_real;
50.  typedef struct {
```

```
51.         short m0,m1,m2,m3;
52.         short exponent;
53.         } temp_real_unaligned;
54.         #define real_to_real(a,b) \
55.         ((*(long long *) (b) = *(long long *) (a)),((b)->exponent = (a)->exponent))
56.         typedef struct {
57.          long a,b;
58.         } long_real;
59.         typedef long short_real;
60.         typedef struct {
61.          long a,b;
62.          short sign;
63.         } temp_int;
64.         struct swd {
65.          int ie:1;
66.          int de:1;
67.          int ze:1;
68.          int oe:1;
69.          int ue:1;
70.          int pe:1;
71.          int sf:1;
72.          int ir:1;
73.          int c0:1;
74.          int c1:1;
75.          int c2:1;
76.          int top:3;
77.          int c3:1;
78.          int b:1;
79.         };
80.         #define I387 (current->tss.i387)
81.         #define SWD (*(struct swd *) &I387.swd)
82.         #define ROUNDING ((I387.cwd >> 10) & 3)
83.         #define PRECISION ((I387.cwd >> 8) & 3)
84.         #define BITS24   0
85.         #define BITS53   2
86.         #define BITS64   3
87.         #define ROUND_NEAREST0
88.         #define ROUND_DOWN     1
89.         #define ROUND_UP   2
90.         #define ROUND_0            3
91.         #define CONSTZ     (temp_real_unaligned) {0x0000,0x0000,0x0000,0x0000,0x0000}
92.         #define CONST1     (temp_real_unaligned) {0x0000,0x0000,0x0000,0x8000,0x3FFF}
93.         #define CONSTPI    (temp_real_unaligned) {0xC235,0x2168,0xDAA2,0xC90F,0x4000}
94.         #define CONSTLN2 (temp_real_unaligned) {0x79AC,0xD1CF,0x17F7,0xB172,0x3FFE}
95.         #define CONSTLG2 (temp_real_unaligned) {0xF799,0xFBCF,0x9A84,0x9A20,0x3FFD}
96.         #define CONSTL2E (temp_real_unaligned) {0xF0BC,0x5C17,0x3B29,0xB8AA,0x3FFF}
97.         #define CONSTL2T (temp_real_unaligned) {0x8AFE,0xCD1B,0x784B,0xD49A,0x4000}
98.         #define set_IE() (I387.swd |= 1)
99.         #define set_DE() (I387.swd |= 2)
100.        #define set_ZE() (I387.swd |= 4)
```

```
101.    #define set_OE() (I387.swd |= 8)
102.    #define set_UE() (I387.swd |= 16)
103.    #define set_PE() (I387.swd |= 32)
104.    #define set_C0() (I387.swd |= 0x0100)
105.    #define set_C1() (I387.swd |= 0x0200)
106.    #define set_C2() (I387.swd |= 0x0400)
107.    #define set_C3() (I387.swd |= 0x4000)
108.
109.    char * ea(struct info * __info, unsigned short __code);
110.
111.    void frndint(const temp_real * __a, temp_real * __b);
112.    void short_to_temp(const short_real * __a, temp_real * __b);
113.    void long_to_temp(const long_real * __a, temp_real * __b);
114.    void temp_to_short(const temp_real * __a, short_real * __b);
115.    void temp_to_long(const temp_real * __a, long_real * __b);
116.    void real_to_int(const temp_real * __a, temp_int * __b);
117.    void int_to_real(const temp_int * __a, temp_real * __b);
118.
119.    void get_short_real(temp_real *, struct info *, unsigned short);
120.    void get_long_real(temp_real *, struct info *, unsigned short);
121.    void get_temp_real(temp_real *, struct info *, unsigned short);
122.    void get_short_int(temp_real *, struct info *, unsigned short);
123.    void get_long_int(temp_real *, struct info *, unsigned short);
124.    void get_longlong_int(temp_real *, struct info *, unsigned short);
125.    void get_BCD(temp_real *, struct info *, unsigned short);
126.    void put_short_real(const temp_real *, struct info *, unsigned short);
127.    void put_long_real(const temp_real *, struct info *, unsigned short);
128.    void put_temp_real(const temp_real *, struct info *, unsigned short);
129.    void put_short_int(const temp_real *, struct info *, unsigned short);
130.    void put_long_int(const temp_real *, struct info *, unsigned short);
131.    void put_longlong_int(const temp_real *, struct info *, unsigned short);
132.    void put_BCD(const temp_real *, struct info *, unsigned short);
133.
134.    void fadd(const temp_real *, const temp_real *, temp_real *);
135.
136.    void fmul(const temp_real *, const temp_real *, temp_real *);
137.
138.    void fdiv(const temp_real *, const temp_real *, temp_real *);
139.
140.    void fcom(const temp_real *, const temp_real *);
141.    void fucom(const temp_real *, const temp_real *);
142.    void ftst(const temp_real *);
143.    #endif
```

代码中第 7～17 行是定义调用者（int 7）返回地址。其中包括临时保存原 EIP 的地方；异常中断 int 7 处理过程入栈的寄存器；中断 int 7 返回时将去执行系统调用的返回处理代码等信息。

3.32 minix_fs.h 文件

该文件是 Minix 文件系统头文件，在其中定义了一些内核常用的文件操作的函数原型等。所以在内核的文件管理的相关文件中经常需要引用该头文件。

在该文件中定义了一些内核文件管理中常用的函数原型等。在具体文件管理的 C 函数文件中，将对应不同的文件格式来实现这些函数。minix_fs.h 文件的详细代码注释如下：

```
1.  #ifndef _MINIX_FS_H
2.  #define _MINIX_FS_H
3.  #include <sys/types.h>
4.  #define MINIX_NAME_LEN 14 //MINIX 名字长度
5.  #define MINIX_ROOT_INO 1 //MINIX 根结点
6.  #define MINIX_I_MAP_SLOTS 8 //MINIX MAP
7.  #define MINIX_Z_MAP_SLOTS 8
8.  #define MINIX_SUPER_MAGIC 0x137F
9.  #define MINIX_INODES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct minix_inode)))
10. #define MINIX_DIR_ENTRIES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct minix_dir_ entry)))
11. struct minix_inode {                    //MINIX i 结点的定义
12.  unsigned short i_mode;
13.  unsigned short i_uid;
14.  unsigned long i_size;
15.  unsigned long i_time;
16.  unsigned char i_gid;
17.  unsigned char i_nlinks;
18.  unsigned short i_zone[9];
19. };
20. struct minix_super_block {              //MINIX 超级块的定义
21.  unsigned short s_ninodes;
22.  unsigned short s_nzones;
23.  unsigned short s_imap_blocks;
24.  unsigned short s_zmap_blocks;
25.  unsigned short s_firstdatazone;
26.  unsigned short s_log_zone_size;
27.  unsigned long s_max_size;
28.  unsigned short s_magic;
29. };
30. struct minix_dir_entry {                 //MINIX 目录的定义
31.  unsigned short inode;
32.  char name[MINIX_NAME_LEN];
33. };
34. extern int minix_open(struct inode * inode, struct file * filp);
35. extern void minix_release(struct inode * inode, struct file * filp);
36. extern struct inode * minix_follow_link(struct inode * dir, struct inode * inode);
37. extern int minix_lookup(struct inode * dir,const char * name, int len,
38.  struct inode ** result);
39. extern int minix_create(struct inode * dir,const char * name, int len, int mode,
40.  struct inode ** result);
41. extern int minix_mkdir(struct inode * dir, const char * name, int len, int mode);
42. extern int minix_rmdir(struct inode * dir, const char * name, int len);
43. extern int minix_unlink(struct inode * dir, const char * name, int len);
```

```
44.         extern int minix_symlink(struct inode * inode, const char * name, int len,
45.          const char * symname);
46.         extern int minix_link(struct inode * oldinode, struct inode * dir, const char * name, int len);
47.         extern int minix_mknod(struct inode * dir, const char * name, int len, int mode, int rdev);
48.         extern int minix_rename(struct inode * old_dir, const char * old_name, int old_len,
49.          struct inode * new_dir, const char * new_name, int new_len);
50.         extern int minix_readlink(struct inode * inode, char * buffer, int buflen);
51.         extern struct inode * minix_new_inode(int dev);
52.         extern void minix_free_inode(struct inode * inode);
53.         extern int minix_new_block(int dev);
54.         extern int minix_free_block(int dev, int block);
55.         extern int minix_create_block(struct inode * inode, int block);
56.         extern int minix_bmap(struct inode * inode,int block);
57.         extern int minix_lseek(struct inode * inode, struct file * filp, off_t offset, int origin);
58.         extern int minix_read(struct inode * inode, struct file * filp, char * buf, int count);
59.         extern int minix_write(struct inode * inode, struct file * filp, char * buf, int count);
60.         extern struct inode_operations minix_inode_operations;
61.         extern struct file_operations minix_file_operations;
62.         #endif
```

代码中第 3～12 行是定义 MINIX 文件系统。

3.33 timer.h 文件

timer.h 是定时器头文件，在文件中定义了数据结构 timer_list 来描述一个内核定时器。

在该文件中定义了一个表示内核定时器的结构体，在该结构体中定义了时钟到期的时间以及时钟到期后所要执行的对应的函数。timer.h 文件的代码如下：

```
1.          #ifndef _TIMER_H
2.          #define _TIMER_H
3.          #define BLANK_TIMER     0
4.          #define BEEP_TIMER 1
5.          #define SER1_TIMER 2
6.          #define SER2_TIMER 3
7.          #define SER1_TIMEOUT    8
8.          #define SER2_TIMEOUT    9
9.          #define HD_TIMER    16
10.         #define FLOPPY_TIMER    17
11.         struct timer_struct {
12.          unsigned long expires;
13.          void (*fn)(void);
14.         };
15.         extern unsigned long timer_active;
16.         extern struct timer_struct timer_table[32];
17.         #endif
```

该文件主要是对系统定时器的一些设置。结构体中各数据成员的含义如下：

expires：指定定时器到期的时间，这个时间被表示成自系统启动以来的时钟滴答计数（也即时钟节拍数）。当一个定时器的 expires 值小于或等于 jiffies 变量时，我们就说这个定时器已经超时或到期了。在初始化一个定时器后，通常把它的 expires 域设置成当前 expires 变量的当前值加上某个时间间隔值（以时钟滴答次数计）。

函数指针 fn：指向一个可执行函数。当定时器到期时，内核就执行 fn 所指定的函数。而 data 域则被内核用作 fn 函数的调用参数。

3.34 include/sys/目录下的文件

include/sys/目录下的文件如表 3-4 所示。

表 3-4 include/sys/目录下的文件

名　　称	大小（B）	最 后 修 改 时 间
param.h	196	1992-01-07 05:10:22
ptrace.h	874	1992-02-18 03:54:12
resource.h	1809	1992-01-04 02:52:56
stat.h	1376	1992-02-20 22:45:18
time.h	1838	1992-02-21 07:07:22
times.h	200	1992-09-17 23:03:06
types.h	952	1992-02-29 05:34:58
utsname.h	272	1992-01-04 23:05:42
wait.h	593	1992-12-22 23:08:02

3.35 param.h 文件

该文件定义了操作系统中一些与系统设置相关的系统参数。

param.h 头文件中定义了操作系统的一些参数值，例如，时钟中断频率、hostname 的最大长度等，这些参数都是系统参数，是与操作系统性能紧密相关的。param.h 文件的详细代码注释如下：

```
1.    #ifndef _SYS_PARAM_H
2.    #define _SYS_PARAM_H
3.
4.    #define HZ 100                          //设定时钟中断频率为 100
5.    #define EXEC_PAGESIZE 4096              //定义执行页面大小为 4MB
6.
7.    #define NGROUPS        32               //定义每个用户最大组数量为 32
8.
9.    #define NOGROUP        -1               //定义无用户组为-1
10.
11.   #define MAXHOSTNAMELEN 8                //hostname 的最大长度定义为 8
12.
13.   #endif
```

3.36 ptrace.h 文件

该文件定义了与 ptrace 系统调用相关的参数。Linux 中的 ptrace 系统调用是监视进程间通信的关键。ptrace 为用户提供了一种观察和控制其他进程的方法。利用 ptrace，用户可以截获正在运行的进程的所有的系统调用。ptrace.h 头文件中定义了进程跟踪调试系统调用 ptrace 的相关结构体和宏定义。

ptrace.h 文件的详细代码注释如下：

```
1.          #ifndef _SYS_PTRACE_H
2.            #define _SYS_PTRACE_H
3.
        //相关寄存器定义。
4.            #define EBX 0
5.            #define ECX 1
6.            #define EDX 2
7.            #define ESI 3
8.            #define EDI 4
9.            #define EBP 5
10.           #define EAX 6
11.           #define DS 7
12.           #define ES 8
13.           #define FS 9
14.           #define GS 10
15.           #define ORIG_EAX 11
16.           #define EIP 12
17.           #define CS    13
18.           #define EFL 14
19.           #define UESP 15
20.           #define SS      16
21.           //寄存器值存储结构体
22.         struct pt_regs {
23.           long ebx;
24.           long ecx;
25.           long edx;
26.           long esi;
27.           long edi;
28.           long ebp;
29.           long eax;
30.           long ds;
31.           long es;
32.           long fs;
33.           long gs;
34.           long orig_eax;
35.           long eip;
36.           long cs;
37.           long eflags;
38.           long esp;
39.           long ss;
40.         };
41.
42.         #endif /* _SYS_PTRACE_H */
```

源代码中第 1～2 行说明该文件的主要内容为帮助用户使用 ptrace 系统调用的结构体定义和宏定义。第 4～20 行说明了使用系统调用 ptrace（3 or 6, pid, PT_EXCL, data）去读或者写进程寄存器。第 22～40 行解释了寄存器值存储结构体的主要功能为定义了在使用系统调用期间，寄存器值存储在堆栈中的方法。

3.37 resource.h 文件

运行在 Linux 系统上的程序是有资源限制的。这些也许是硬件引起的限制（例如，内存），也许由系统策略引起的限制（例如，允许的 CPU 时间），或者是实现的限制（例如，整数的尺寸或是文件名允许的最大字符个数）。Linux 规范定义了一些可以由程序确定的限制。resource.h 文件中就定义了这些资源限制管理的相关参数。

resource.h 头文件主要包含了 Linux 资源限制管理的相关结构体和宏定义。resource.h 文件的详细代码注释如下：

```
1.  #ifndef _SYS_RESOURCE_H
2.      #define _SYS_RESOURCE_H
3.      #define   RUSAGE_SELF 0
4.      #define   RUSAGE_CHILDREN     -1
5.  //进程资源结构体
6.  struct    rusage {
7.          struct timeval ru_utime;          //在用户模式中的执行时间
8.          struct timeval ru_stime;          //系统模式中的执行时间
9.      long  ru_maxrss;                      //常驻集内存使用数
10.     long  ru_ixrss;
                                              //文本段（text segment）使用的内存数乘以执行滴答数
11.     long ru_idrss;                        //进程所使用的私有内存数乘以执行滴答数
12.     long  ru_isrss;                       //栈使用的内存数(KB 为单位)乘以执行滴答数
13.     long  ru_minflt;                      //不需要 I/O 的页缺失数
14.     long  ru_majflt;                      //需要 I/O 的页缺失数
15.     long  ru_nswap;                       //一个进程将要调出内存的次数
16.     long  ru_inblock;                     //文件系统需要为一个读请求执行输入操作的次数
17.     long  ru_oublock;                     //文件系统需要为一个写入请求执行输出操作的次数
18.     long  ru_msgsnd;                      //发送的 IPC 信息总数
19.     long  ru_msgrcv;                      //收到的 IPC 信息总数
20.     long  ru_nsignals;                    //进程收到的信号总数
21.     long  ru_nvcsw;                       //一个进程主动上下文且混总数
22.     long  ru_nivcsw;                      //因高优先级进程运行导致的上下文切换总数
23. };
24.
25.      #define RLIMIT_CPU          0    //CPU 毫秒级时间
26.      #define RLIMIT_FSIZE        1    //最大文件长度
27.      #define RLIMIT_DATA         2    //最大数据长度
28.     #define RLIMIT_STACK             3    //栈大小的上界
29.     #define RLIMIT_CORE              4    //Core 文件长度的上界
30.     #define RLIMIT_RSS          5    //驻留进程集的上限
31.
32.      #ifdef notdef
33.      #define RLIMIT_MEMLOCK      6    //可锁住的内存的上限
34.     #define RLIMIT_NPROC             7    //进程数目的上限
35.     #define RLIMIT_OFILE             8    //打开文件数目的上限
36.     #endif
37.
38.     #define RLIM_NLIMITS        6    //受控资源数
39.
40.     #define RLIM_INFINITY  0x7fffffff  //非受控资源数
```

```
41.
      //资源限制结构体
42.   struct rlimit {
43.           int    rlim_cur;          //软限制：当前限制
44.           int    rlim_max;          //硬限制：rlimcur 的最大值
45.   };
46.   #endif /* _SYS_RESOURCE_H */
```

源代码中第 1～4 行说明该文件为 linux 的资源管理头文件。第 6～23 行对结构进行了解释：从 BSD 4.3 Reno 获取的结构定义，我们现在并不支持所有的这些，但我们以后可能会支持。否则，每次我们加入新的项，依赖于这个结构体的程序将失效，这样可以减少类似事件的发生。第 38～40 行声明一下的宏定义为资源的限制。

3.38 stat.h 文件

该文件中定义了读取文件状态信息返回的相关参数。因此在 stat.c 文件中需要引入该头文件。

stat.h 头文件说明了函数 stat()返回的数据及其结构类型，以及一些属性操作测试宏、函数原型等。stat.h 文件的详细代码注释如下：

```
1.    #ifndef _SYS_STAT_H
2.      #define _SYS_STAT_H
3.      #include <sys/types.h>                    //类型头文件，定义了基本的数据类型
4.    struct stat {
5.          dev_t st_dev;                       //文件的设备号
6.          ino_t st_ino;                       //文件 i 结点号
7.      umode_t   st_mode;                      //文件类型和属性
8.      nlink_t     st_nlink;                   //文件的连接数
9.          uid_t st_uid;                       //文件的用户号
10.     gid_t st_gid;                           //文件的组号
11.     dev_t st_rdev;                          //设备号
12.         off_t st_size;                      //文件大小
13.         time_t      st_atime;               //文件最后访问时间
14.     time_t      st_mtime;                   //最后修改时间
15.     time_t      st_ctime;                   //最后结点修改时间
16.       };
      //文件类型
17.     #define S_IFMT   00170000              //文件类型
18.     #define S_IFLNK   0120000                   //符号连接文件
19.     #define S_IFREG   0100000                   //常规文件
20.     #define S_IFBLK   0060000              //块设备文件
21.     #define S_IFDIR   0040000                   //目录文件
22.     #define S_IFCHR   0020000                   //字符设备文件
23.     #define S_IFIFO   0010000                   //FIFO 特殊文件
      //文件属性
24.     #define S_ISUID   0004000                   //执行时设置用户 ID
25.     #define S_ISGID   0002000                   //执行时设置组 ID
26.     #define S_ISVTX   0001000                   //对于目录的受限删除标志
27.     #define S_ISLNK(m)   (((m) & S_IFMT) == S_IFLNK)  //测试是否为符号连接文件
28.     #define S_ISREG(m)   (((m) & S_IFMT) == S_IFREG)  //测试是否为常规文件
29.     #define S_ISDIR(m)   (((m) & S_IFMT) == S_IFDIR)  //测试是否为目录文件
```

```
30.     #define S_ISCHR(m)   (((m) & S_IFMT) == S_IFCHR)  //测试是否为字符设备文件
31.     #define S_ISBLK(m)   (((m) & S_IFMT) == S_IFBLK)  //测试是否为块设备文件
32.     #define S_ISFIFO(m)  (((m) & S_IFMT) == S_IFIFO)  //测试是否为 FIFO 特殊文件
        //文件访问权限
33.     #define S_IRWXU 00700                             //宿主可以读、写、执行/搜索
34.     #define S_IRUSR 00400                             //宿主读许可
35.     #define S_IWUSR 00200                             //宿主写许可
36.     #define S_IXUSR 00100                             //宿主执行/搜索许可
37.     #define S_IRWXG 00070                             //组成员可以读、写、执行/搜索
38.     #define S_IRGRP 00040                             //组成员读许可
39.     #define S_IWGRP 00020                             //组成员写许可
40.     #define S_IXGRP 00010                             //组成员执行/搜索许可
41.     #define S_IRWXO 00007                             //其他人读、写、执行/搜索许可
42.     #define S_IROTH 00004                             //其他人读许可
43.     #define S_IWOTH 00002                             //其他人写许可
44.     #define S_IXOTH 00001                             //其他人执行/搜索许可
45.     extern int chmod(const char *_path, mode_t mode);         //修改文件属性
46.     extern int fstat(int fildes, struct stat *stat_buf);      //取指定文件句柄的文件状态信息
47.     extern int mkdir(const char *_path, mode_t mode);         //创建目录
48.     extern int mkfifo(const char *_path, mode_t mode);        //创建管道文件
49.     extern int stat(const char *filename, struct stat *stat_buf); //取指定文件名的文件状态信息
50.     extern mode_t umask(mode_t mask);                         //设置属性屏蔽码
51.     #endif
```


3.39 time.h 文件

该文件定义了与系统中时间相关的参数。在需要获取系统时间的文件中都需要引入该头文件。

time.h 是 Linux 系统的日期时间头文件，定义了时间相关的宏和操作函数原型等。time.h 文件的详细代码注释如下：

```
1.  #ifndef _SYS_TIME_H
2.  #define _SYS_TIME_H
3.      struct timeval {
4.          long  tv_sec;.                          //秒
5.          long  tv_usec;                          //微秒
6.  };
7.  struct timezone {
8.          int   tz_minuteswest; //和格林威治时间的分钟差
9.          int   tz_dsttime; //日光节约时间的状态
10.     };
11. //tz_dsttime 所代表的状态。
12. #define  DST_NONE    0                          //不使用
13.  #define  DST_USA     1                          //美国
14.  #define  DST_AUST    2                          //澳洲
15. #define  DST_WET     3                          //西欧
16. #define  DST_MET     4                          //中欧
17.  #define  DST_EET     5                          //东欧
18.  #define  DST_CAN     6                          //加拿大
19.  #define  DST_GB      7                          //大不列颠
```

```
20.   #define   DST_RUM      8                                  //罗马尼亚
21.   #define   DST_TUR      9                                  //土耳其
22.   #define   DST_AUSTALT 10                                  //澳洲（1986 以后）
23.   #define FD_SETSIZE            (8*sizeof(fd_set))          //一个进程文件描述符的最大数目
24.   #define FD_SET(fd,fdsetp)(*(fdsetp) |= (1 << (fd)))       //在文件描述符集合中增加一个新的文件
                                                                //描述符
25.  #define FD_CLR(fd,fdsetp)(*(fdsetp)&=~(1<< (fd)))         //在文件描述符集合中删除一个文件描述符
26.   #define FD_ISSET(fd,fdsetp)   ((*(fdsetp) >> fd) & 1)//测试指定的文件描述符是否在该集合中
27.   #define FD_ZERO(fdsetp)       (*(fdsetp) = 0)           //将指定的文件描述符集清空
28.   #define   timerisset(tvp)       ((tvp)->tv_sec || (tvp)->tv_usec)   //测试时钟是否为零
29.   #define   timercmp(tvp, uvp, cmp)                       //比较两个时钟
30.       ((tvp)->tv_sec cmp (uvp)->tv_sec ||
31.        (tvp)->tv_sec == (uvp)->tv_sec && (tvp)->tv_usec cmp (uvp)->tv_usec)
32.        #define   timerclear(tvp)        ((tvp)->tv_sec = (tvp)->tv_usec = 0)   //将时钟清零
33.         #define   ITIMER_REAL 0                           //定时器按照真实时钟递减
34.        #define   ITIMER_VIRTUAL   1 //定时器按照进程虚拟时钟递减，仅当进程执行中运行
35.  #define   ITIMER_PROF  2            //定时器在进程虚拟时钟中递减
36.  struct      itimerval {
37.              struct timeval it_interval;                   /*计时器值*/
38.              struct timeval it_value;                      /*现在的值*/
39.  };
40.     #include <time.h>                                      //时间类型头文件，其中最主要定义了 tm
                                                                //结构和一些有关时间的函数原型
41.      #include <sys/types.h>                                //类型头文件，定义了基本的数据类型
42.  int gettimeofday(struct timeval * tp, struct timezone * tz); //获取当前时间和时区信息函数
43.  int select(int width, fd_set * readfds, fd_set * writefds,  //确定一个或多个套接口的状态函数
44.              fd_set * exceptfds, struct timeval * timeout);
45.      #endif /*_SYS_TIME_H*/
```

源代码中第 3 行声明结构体 timeval 是由 gettimofday 返回的。第 32～35 行说明 timevals 的操作不能用于>=或者<=。第 40～45 行声明以下定义为计时器和结构的名称、定义计时器设置。

3.40 times.h 文件

该文件主要定义了代表进程时间的结构体 tms。times.h 头文件定义了文件访问和修改时间结构体 tms 和一个函数原型 times()。times.h 文件的详细代码注释如下：

```
1.      #ifndef _TIMES_H
2.      #define _TIMES_H
3.        #include <sys/types.h>                                //类型头文件，定义了基本的数据类型
4.        struct tms {
5.         time_t tms_utime;                                   //用户使用的 CPU 时间
6.         time_t tms_stime;                                   //系统 CPU 时间
7.         time_t tms_cutime;                                  //已终止的子进程使用的用户 CPU 时间
8.         time_t tms_cstime;                                  //已终止的子进程使用的系统 CPU 时间
9.      };
10.       extern time_t times(struct tms * tp);
11.       #endif
```

3.41 types.h 文件

types.h 头文件中定义了操作系统中的一些基本数据类型。在内核中使用这些基本数据类型的文件中就必须引用该头文件。types.h 头文件定义了操作系统中的基本的数据类型。所有的类型定义为适当的数学类型长度。types.h 文件的详细代码注释如下：

```
1.    #ifndef _SYS_TYPES_H
2.    #define _SYS_TYPES_H
3.    #ifndef _SIZE_T
4.    #define _SIZE_T
5.    typedef unsigned int size_t;                    //用于对象的大小
6.    #endif
7.
8.    #ifndef _TIME_T
9.    #define _TIME_T
10.   typedef long time_t;                            //用于时间
11.   #endif
12.     #ifndef _PTRDIFF_T
13.   #define _PTRDIFF_T
14.   typedef long ptrdiff_t;
15.   #endif
16.     #ifndef NULL
17.   #define NULL ((void *) 0)
18.   #endif
19.   typedef int pid_t;                              //用于进程号和进程组号
20.   typedef unsigned short uid_t;                   //用于用户号
21.   typedef unsigned short gid_t;                   //用于组号
22.   typedef unsigned short dev_t;                   //用于设备号
23.   typedef unsigned short ino_t;                   //用于文件序列号
24.   typedef unsigned short mode_t;                  //用于某些文件属性
25.   typedef unsigned short umode_t;
26.   typedef unsigned short nlink_t;                 //用于连接计数
27.   typedef int daddr_t;
28.   typedef long off_t;                             //用于文件长度
29.   typedef unsigned char u_char;                   //无符号字符类型
30.   typedef unsigned short ushort;                  //无符号短整数类型
31.   typedef char *caddr_t;
32.   typedef unsigned char cc_t;
33.   typedef unsigned int speed_t;
34.   typedef unsigned long tcflag_t;
35.   typedef unsigned long fd_set;                   //用于文件描述设置
36.   typedef struct { int quot,rem; } div_t;         //用于 DIV 操作
37.   typedef struct { long quot,rem; } ldiv_t;       //用于长 DIV 操作
38.   //文件系统参数结构。
39.   struct ustat {
40.       daddr_t f_tfree;                            //系统总空闲块数
41.       ino_t f_tinode;                             //总空闲 i 结点数
42.       char f_fname[6];                            //文件系统名
43.       char f_fpack[6];                            //文件系统压缩名
44.   };
45.     #endif
```

3.42 utsname.h 文件

在该文件中定义了与系统名称相关的参数。这些参数包括当前操作系统的名称、版本信息以及运行操作系统的计算机名称等。

utsname.h 头文件是系统名称结构头文件。定义了结构体 utsname 以及函数原型 uname()。utsname.h 文件的详细代码注释如下：

```
1.    #ifndef _SYS_UTSNAME_H
2.    #define _SYS_UTSNAME_H
3.    #include <sys/types.h>                          //类型头文件，定义了基本的数据类型
4.    #include <sys/param.h>                          //系统参数头文件
5.    struct utsname {
6.        char sysname[9];                            //本版本操作系统的名称
7.        char nodename[MAXHOSTNAMELEN+1];            //与实现相关的网络结点名称
8.        char release[9];                            //本实现的当前发行级别
9.        char version[9];                            //本次发行的版本级别
10.       char machine[9];                            //系统运行的硬件类型名称
11.   };
12.   extern int uname(struct utsname * utsbuf);
13.   #endif
```


3.43 wait.h 文件

在 Linux 内核中，通常为了使父子进程同步，在子进程执行时，父进程需要处于等待状态，直到子进程结束。该文件中定义了与进程等待相关的参数。wait.h 头文件描述了进程等待时的信息。包括一些符号常数和函数原型声明。wait.h 文件的详细代码注释如下：

```
1.    #ifndef _SYS_WAIT_H
2.    #define _SYS_WAIT_H
3.    #include <sys/types.h>                          //类型头文件，定义了基本的数据类型
4.    #define _LOW(v) ( (v) & 0377)                   //取低字节
5.    #define _HIGH(v) ( ((v) >> 8) & 0377)           //取高字节
6.    #define WNOHANG        1                        //如果没有状态也不挂起，并立刻返回
7.    #define WUNTRACED      2                        //报告停止执行的子进程状态
      //判断 waitpid()函数返回的状态子含义的宏定义。
8.    #define WIFEXITED(s) (!((s)&0xFF))              //如果子进程正常退出，则为真
9.    #define WIFSTOPPED(s) (((s)&0xFF)==0x7F)         //如果子进程正停止着，则为真
10.   #define WEXITSTATUS(s) (((s)>>8)&0xFF)           //返回退出状态
11.   #define WTERMSIG(s) ((s)&0x7F)                   //返回导致进程终止的信号量
12.   #define WCOREDUMP(s) ((s)&0x80)                  //如果子进程导致内核转存返回真
13.   #define WSTOPSIG(s) (((s)>>8)&0xFF)              //返回导致进程停止的信号量
14.   #define WIFSIGNALED(s) (((unsigned int)(s)-1 & 0xFFFF) < 0xFF)    //如果由于未捕捉到信号而
                                                                        //导致子进程退出则为真
15.   pid_t wait(int *stat_loc);
16.   pid_t waitpid(pid_t pid, int *stat_loc, int options);
17.   #endif
```

源代码中第 8 行声明以下定义了 waitpid 的选项，并且其中 WUNTRACED 未被支持。

第 4 章　中断处理与系统调用

本章主要讨论系统调用与中断处理的原理及功能。本章分析的内容主要包括：set_gate()函数、traps.c、sys.c、asm.s、sys_call.s、unistd.h、sys.h 和 ptrace.c 文件。

- set_gate 函数主要负责中断描述符表的初始化工作；
- traps.c 主要实现了硬件异常中断处理程序；
- sys.c 主要完成了系统调用程序的实现；
- asm.s 汇编文件主要实现了大部分的硬件故障（或出错）处理程序；
- sys_call.s 主要实现了系统调用中断（int 0x80）的入口处理以及信号检测处理程序；
- unistd.h 定义了各种符号常数和类型；
- sys.h 主要声明了内核中所有的系统函数，并构建系统调用表 sys_call_table；
- ptrace.c 主要实现了用于跟踪子进程的系统调用程序，即 sys_ptrace。通过 ptrace，使得父进程可以对自己的子进程进行监督与控制：包括读写子进程用户空间的程序、数据或 user 结构中的变量，软中断的发送以及自我终止等。

所涉及的源代码文件见表 4-1 所示。

表 4-1　源代码文件 目录

名　　称	大　小（B）	修　改　时　间
sys.h	3476	1992-02-28 10:24:26
unistd.h	7482	1992-03-04 18:56:06
trap.c	4806	1992-03-06 14:27:06
sys.c	12996	1992-02-28 10: 42:10
sys_call.s	5941	1992-03-05 18:56:08
asm.s	2378	1992-03-06 00:15:20
ptrace.c	8640	1992-02-18 04:37:58

4.1　总体功能

中断和系统调用是操作系统内核中必不可少的功能之一。中断是操作系统处理异常事件的一种机制。若在计算机执行期间系统发生了非预期的异常事件时，当前执行任务被中止并且相应的事件处理程序被执行，在处理完毕后又返回到原来被中断处继续执行。这一过程被称之为中断。中断按照中断源来划分可分为内部中断、软中断以及 I/O 中断（外部中断）三类。系统调用是应用程序和内核之间的交互接口，它是由内核提供并且可供用户调用的函数集合。

4.1.1　中断处理

中断是系统处理内部或外部事件的一种机制。当在系统正常运行过程中有异常事件发生时（如 IO 请求），计算机系统是采用中断机制进行异常事件的处理。中断处理过程一般分成以下几个步骤：

1. 保护被中断进程的上下文环境

处理机状态字 PSW 和程序计数器 PC 中的内容保存在中断保留区，中断进程的 CPU 现场信息（包括通用寄存器、段寄存器等 CPU 寄存器信息）保存在中断栈中。

2. 分析中断原因，转入相应的中断处理程序

确定引起本次中断的原因，然后将相应的中断处理程序入口地址装入程序计数器中，以便处理器转向中断处理程序。

3. 中断处理

不同的中断有不同的处理过程。例如设备中断处理程序先从设备控制器中读出设备状态，以识别本次中断是正常完成中断还是异常结束中断。若是正常完成中断，中断程序便做结束处理；若是异常结束中断，则根据异常发生的原因做相应的处理。

4. 恢复被中断进程的上下文环境

当中断处理完成后，可将在中断栈中的被中断进程的现场信息取出，并装入到相应的寄存器中，其中包括该程序下一次要执行的指令的地址、处理器的状态字以及通用寄存器和段寄存器等寄存器的内容。

5. 返回被中断进程继续执行

处理器根据程序计数器转向被中断的程序处开始执行。

中断可分为软中断和硬中断两种。软中断是由内核机制的触发事件引起的（例如进程运行超时）。硬中断是由硬件发过来的中断信号引起的。当硬中断收到相应处理的中断信号后，硬中断会根据设备的状态寄存器进行相应的处理。

4.1.2 系统调用

在 Linux 操作系统中，系统调用是必不可少的系统功能。操作系统的主要功能是为应用程序的运行创建良好的环境。操作系统提供给应用程序一系列可调用的内核函数。这些内核函数通过一组称为系统调用的（system call）的接口呈现给用户。系统调用是应用程序和内核之间的交互接口。

在系统中真正被所有进程都使用的内核通信方式是系统调用。系统调用把应用程序的请求传给内核，调用相应的内核函数完成所需的处理，将处理结果返回给应用程序。如果没有系统调用和内核函数，用户将不能编写大型应用程序。

系统调用在 Linux 系统中发挥着巨大的作用，缺少系统调用，应用程序就会失去了内核的支持而无法正常工作。常见的如 fork、open 等这些功能都是通过系统调用来实现的，例如下面这个程序：

```
int main()
{
 fork();
 exit(0);
}
```

这里用到了 fork 和 exit 两个函数，这两个函数都是 glibc 中的函数。fork 和 exit 采用软中断的方式陷入到内核并通过系统调用实现其功能。系统调用是用户接口在内核中的实现，不通过系统调用，用户就无法利用内核功能。

4.2 重要数据结构

idt_descr 是 Linux 操作系统中的中断描述符表。中断描述符表(IDT)的创建代码在 boot/head.s 中，与全局描述符表的创建类似，内核执行 lidt idt_descr 指令完成中断描述符表的创建。其定义如下：

```
 idt_descr:
  .word 256*8-1 # idt contains 256 entries
        .long _idt
.align 2
.word 0
```

IDT 描述符表包含 256 表项，lidt 指令为 6 字节操作数，它将_idt 的地址加载进 idtr 寄存器。

4.3 sys.h 文件

sys.h 中定义了系统中常用的系统调用，因此会被内核中很多文件包含。该文件位于 Linux 0.95 内核的 kernel/include 目录下，这个文件主要包含两部分代码内容：一部分是系统调用接口（以 sys_开头的函数）；另一部分是系统调用表（sys_call_table 数组）。

1. 功能描述

sys.h 文件包含了内核中所有的系统调用接口的声明（以 sys_开头的函数）和系统调用表 sys_call_table 的定义。sys_call_table（系统调用表）建立了系统调用号与系统调用的映射关系。linux 0.95 内核定义了 89 个系统调用，这些系统调用都存放在 sys_call_table 这个指针数组里。可利用 sys_call_table（系统调用号）调用相应的系统调用，如 sys_call_table[1]是调用中止进程的系统调用。

2. 代码分析

sys.h 文件的详细代码注释如下：

```
1.      extern int  sys_setup() ;          //仅用于初始化，以启动系统
2.      extern int  sys_exit ();           //中止进程
3.      extern int  sys_fork ();           //创建一个新进程
4.      extern int  sys_read ();           //读文件
5.      extern int  sys_write ();          //写文件
6.      extern int  sys_open ();           //打开文件
7.      extern int  sys_close ();          //关闭句柄指向的文件
8.      extern int  sys_waitpid ();        //等待子进程终止
9.      extern int  sys_creat ();          //创建新文件
10.     extern int  sys_link ();           //创建链接
11.     extern int  sys_unlink ();         //删除链接
12.     extern int  sys_execve ();         //运行可执行文件
13.     extern int  sys_chdir ();          //改变当前工作目录
14.     extern int  sys_time ();           //取得系统时间
15.     extern int  sys_mknod ();          //创建索引结点
16.     extern int  sys_chmod ();          //改变文件方式
17.     extern int  sys_lchown ();         //改变文件的属主或用户组
18.     extern int  sys_break ();          //终止
19.     extern int  sys_oldstat ();        //取文件状态信息
20.     extern int  sys_lseek ();          //移动指针
```

```
21.     extern int   sys_getpid ();          //获取进程标识号
22.     extern int   sys_mount ();           //安装文件系统
23.     extern int   sys_umount ();          //卸下文件系统
24.     extern int   sys_setuid ();          //设置用户标志号
25.     extern int   sys_getuid ();          //获取用户标识号
26.     extern int   sys_stime ();           //设置系统时间
27.     extern int   sys_ptrace ();          //进程跟踪
28.     extern int   sys_alarm ();           //设置进程的闹钟
29.     extern int   sys_oldfstat ();        //获取文件状态信息
30.     extern int   sys_pause ();           //挂起进程
31.     extern int   sys_utime ();           //改变文件的访改时间
32.     extern int   sys_stty ();            //修改终端行设置
33.     extern int   sys_gtty ();            //取终端行设置信息
34.     extern int   sys_access ();          //确定文件的可性
35.     extern int   sys_nice ();            //改变分时进程优先级
36.     extern int   sys_ftime ();           //获取系统当前时间
37.     extern int   sys_sync ();            //将内存缓冲区数据写回硬盘
38.     extern int   sys_kill ();            //向进程或进程组发信号
39.     extern int   sys_rename ();          //文件改名
40.     extern int   sys_mkdir ();           //创建目录
41.     extern int   sys_rmdir ();           //删除目录
42.     extern int   sys_dup ();             //复制句柄指向的文件
43.     extern int   sys_pipe ();            //创建管道
44.     extern int   sys_times ();           //取运行时间
45.     extern int   sys_prof ();            //程序执行时间区域
46.     extern int   sys_brk ();             //改变数据段空间的分配
47.     extern int   sys_setgid ();          //设置组标志号
48.     extern int   sys_getgid ();          //获取组标识号
49.     extern int   sys_signal ();          //信号控制
50.     extern int   sys_geteuid ();         //获取有效用户标识号
51.     extern int   sys_getegid ();         //获取有效组标识号
52.     extern int   sys_acct ();            //启用或禁止进程记账
53.     extern int   sys_umount2 ();         //卸下文件系统
54.     extern int   sys_lock ();            //锁定一个信元
55.     extern int   sys_ioctl ();           // I/O 设备控制
56.     extern int   sys_fcntl ();           //文件控制
57.     extern int   sys_mpx ();             //多指针
58.     extern int   sys_setpgid ();         //设置组号
59.     extern int   sys_ulimit ();          //资源限制
60.     extern int   sys_oldolduname ();     //不常用，可用来替换
61.     extern int   sys_umask ();           //设置文件权限掩码
62.     extern int   sys_chroot ();          //改变参考目录
63.     extern int   sys_ustat   ();         //获取文件系统信息
64.     extern int   sys_dup2 ();            //按指定条件复制文件
65.     extern int   sys_getppid ();         //获得父进程 pid
66.     extern int   sys_getpgrp ();         //获得进程组标识号
67.     extern int   sys_setsid ();          //设置会晤标识号
68.     extern int   sys_sigaction ();       //设置对指定信号的处理方法
69.     extern int   sys_sgetmask ();        //取得现有阻塞信号掩码
70.     extern int   sys_ssetmask ();        //信号处理函数，作用类似 sigaction
```

```
71.     extern int   sys_setreuid ();                    //设置真实和有效的用户标识号
72.     extern int   sys_setregid ();                    //设置真实和有效的组标识号
73.     extern int   sys_sigsuspend ();                  //挂起进程等待特定信号
74.     extern int   sys_sigpending ();                  //为指定的被阻塞信号设置队列
75.     extern int   sys_sethostname ();                 //设置主机名称
76.     extern int   sys_setrlimit ();                   //设置系统资源上限
77.     extern int   sys_getrlimit ();                   //获取系统资源上限
78.     extern int   sys_getrusage ();                   //获取系统资源使用情况
79.     extern int   sys_gettimeofday ();                //取时间和时区
80.     extern int   sys_settimeofday ();                //设置时间和时区
81.     extern int   sys_getgroups ();                   //获取后补组标志清单
82.     extern int   sys_setgroups ();                   //设置后补组标志清单
83.     extern int   sys_select ();                      //对多路同步 I/O 进行轮询
84.     extern int   sys_symlink ();                     //创建符号链接
85.     extern int   sys_oldlstat ();                    //读取文件状态
86.     extern int   sys_readlink ();                    //读符号链接的值
87.     extern int   sys_uselib ();                      //选择要使用的二进制函数库
88.     extern int   sys_swapon ();                      //打开交换文件和设备
89.     extern int   sys_reboot ();                      //重新启动
90.     fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
91.     sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
92.     sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
93.     sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
94.     sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
95.     sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
96.     sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
97.     sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
98.     sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
99.     sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
100.    sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
101.    sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
102.    sys_setreuid,sys_setregid, sys_sigsuspend, sys_sigpending, sys_sethostname,
103.    sys_setrlimit, sys_getrlimit, sys_getrusage, sys_gettimeofday,
104.    sys_settimeofday, sys_getgroups, sys_setgroups, sys_select, sys_symlink,
105.    sys_lstat, sys_readlink, sys_uselib, sys_swapon, sys_reboot };
106.    int NR_syscalls = sizeof(sys_call_table)/sizeof(fn_ptr);     //计算系统调用号
```

上述代码段中的 90～105 行定义的 sys_call_table 是系统调用表。其中 fn_ptr 是一个系统调用函数指针，其详细定义参见 sched.h。

4.4 unistd.h 文件

unistd.h 是位于 include 目录下的 Linux 标准头文件，此代码文件其主要包含：系统调用号、系统调用汇编函数和系统调用函数原型。Linux 下开发的 C 程序都需要该头文件。

1. 功能描述

unistd.h 包含 Linux 0.95 内核中各种符号常数和类型、系统调用号以及内嵌汇编_syscall 等定义，尤其是系统调用的函数原型的声明。这里着重介绍 unistd.h 中有关系统调用的部分功能。同 Linux 0.12 一样，Linux 0.95 在 include/unistd.h 中为每个系统调用定义了一个唯一的编号。用户

在调用一个系统调用时，系统调用号作为参数传递给中断 0x80，该标号实际上是后面将要提到的系统调用表（sys_call_table）的下标，通过该标号执行相应的系统调用。下面对 unistd.h 代码进行详细分析。

2. 代码注释

unistd.h 文件的详细代码注释如下：

```
1.  #define STDIN_FILENO        0          //0 为标准输入文件句柄号
2.  #define STDOUT_FILENO       1          //1 为标准输出文件句柄号
3.  #define STDERR_FILENO       2          //2 为标准出错文件句柄号
4.  #ifndef NULL
5.  #define NULL    ((void *)0)            //定义空类型指针
6.  #endif
7.                                         //定义文件访问符号常数
8.  #define F_OK                0          //检测文件是否存在
9.  #define X_OK                1          //检测是否可执行（搜索）
10. #define W_OK                2          //检测是否可写
11. #define R_OK                4          //检测是否可读
12.                                        //定义文件指针重定位符号常数
13. #define SEEK_SET            0          //将读写指针设置为偏移值
14. #define SEEK_CUR            1          //将文件指针设置为当前值加上偏移
15. #define SEEK_END            2          //将文件指针设置为文件长度加上偏移
16. #define _SC_ARG_MAX         1          //最大变量数
17. #define _SC_CHILD_MAX       2          //最多子进程数
18. #define _SC_CLOCKS_PER_SEC  3          //每秒时间数（滴答）
19. #define _SC_NGROUPS_MAX     4          //最大组数
20. #define _SC_OPEN_MAX        5          //最大打开文件数
21. #define _SC_JOB_CONTROL     6          //作业控制常数
22. #define _SC_SAVED_IDS       7          //保存的标识符
23. #define _SC_VERSION         8          //版本
24. #define _PC_LINK_MAX 1                 //最大连接数
25. #define _PC_MAX_CANON 2                //最大常规文件数
26. #define _PC_MAX_INPUT 3                //最大输入长度
27. #define _PC_NAME_MAX 4                 //最大名称长度
28. #define _PC_PATH_MAX 5                 //最大路径长度
29. #define _PC_PIPE_BUF 6                 //管道缓冲大小
30. #define _PC_NO_TRUNC 7                 //文件名不截断
31. #define _PC_VDISABLE 8                 //禁止功能
32. #define _PC_CHOWN_RESTRICTED 9         //改变宿主受限
33. #include <sys/stat.h>
34. #include <sys/time.h>
35. #include <sys/times.h>
36. #include <sys/utsname.h>
37. #include <sys/resource.h>
38. #include <utime.h>                      //包含头文件
39. #ifdef __LIBRARY__
40. #define __NR_setup   0                 //仅用于初始化，以启动系统
41. #define __NR_exit        1             //终止进程
42. #define __NR_fork        2             //创建一个新进程
43. #define __NR_read        3             //读文件
44. #define __NR_write       4             //写文件
```

```
45.  #define __NR_open          4          //打开文件
46.  #define __NR_close         6          //关闭句柄指向的文件
47.  #define __NR_waitpid       7          //等待子进程终止
48.  #define __NR_creat         8          /创建新文件
49.  #define __NR_link          9          /创建链接
50.  #define __NR_unlink        10         //删除链接
51.  #define __NR_execve        11         //运行可执行文件
52.  #define __NR_chdir         12         //改变当前工作目录
53.  #define __NR_time          13         //取得系统时间
54.  #define __NR_mknod         14         //创建索引结点
55.  #define __NR_chmod         15         //改变文件方式
56.  #define __NR_lchown        16         //改变文件的属主或用户组
57.  #define __NR_break         17         //终止
58.  #define __NR_oldstat       18         //取文件状态信息
59.  #define __NR_lseek         19         //移动文件指针
60.  #define __NR_getpid        20         //获取进程标识号
61.  #define __NR_mount         21         //安装文件系统
62.  #define __NR_umount        22          //卸下文件系统
63.  #define __NR_setuid        23         //设置用户标志号
64.  #define __NR_getuid        24         //获取用户标识号
65.  #define __NR_stime         25         //设置系统时间
66.  #define __NR_ptrace        26         //进程跟踪
67.  #define __NR_alarm         27         //设置进程的闹钟
68.  #define __NR_oldfstat      28         //取文件状态信息
69.  #define __NR_pause         28         //挂起进程
70.  #define __NR_utime         30         //改变文件的访问修改时间
71.  #define __NR_stty          31         //修改终端行设置
72.  #define __NR_gtty          32         //取终端行设置信息
73.  #define __NR_access        33         //确定文件的可存取性
74.  #define __NR_nice          34         //改变分时进程的优先级
75.  #define __NR_ftime         35         //获取系统当前时间
76.  #define __NR_sync          36         //将内存缓冲区数据写回硬盘
77.  #define __NR_kill          37         //向进程或进程组发信号
78.  #define __NR_rename        38         //文件改名
79.  #define __NR_mkdir         39         //创建目录
80.  #define __NR_rmdir         40         //删除目录
81.  #define __NR_dup           41         //复制句柄指向的文件
82.  #define __NR_pipe          42         //创建管道
83.  #define __NR_times         43         //取运行时间
84.  #define __NR_prof          44         //程序执行时间区域
85.  #define __NR_brk           45         //改变数据段空间的分配
86.  #define __NR_setgid        46         //设置组标志号
87.  #define __NR_getgid        47         //获取组标识号
88.  #define __NR_signal        48         //信号
89.  #define __NR_geteuid       49         //获取有效用户标识号
90.  #define __NR_getegid       50         //获取有效组标识号
91.  #define __NR_acct          51         //启用或禁止进程记账
92.  #define __NR_umount2       52         //卸下文件系统
93.  #define __NR_lock          53         //锁定一个信元
94.  #define __NR_ioctl         54         //I/O 设备控制
```

```
95.  #define __NR_fcntl                55      //文件控制
96.  #define __NR_mpx                  56      //多指针
97.  #define __NR_setpgid              57      //设置组号
98.  #define __NR_ulimit               58      //资源限制
99.  #define __NR_oldolduname          59      //不常用，可用来替换
100. #define __NR_umask                60      //设置文件权限掩码
101. #define __NR_chroot               61      //改变参考目录
102. #define __NR_ustat                62      //获取文件系统信息
103. #define __NR_dup2                 63      //按指定条件复制文件描述字
104. #define __NR_getppid              64      //获得父进程 pid
105. #define __NR_getpgrp              65      //获得进程组标识号
106. #define __NR_setsid               66      //设置会晤标识号
107. #define __NR_sigaction            67      //设置对指定信号的处理方法
108. #define __NR_sgetmask             68      //取得现有阻塞信号掩码
109. #define __NR_ssetmask             69      //信号处理函数，作用类似 sigaction
110. #define __NR_setreuid             70      //设置真实和有效的用户标识号
111. #define __NR_setregid             71      //设置真实和有效的组标识号
112. #define __NR_sigsuspend           72      //挂起进程等待特定信号
113. #define __NR_sigpending           73      //为指定的被阻塞信号设置队列
114. #define __NR_sethostname          74      //设置主机名称
115. #define __NR_setrlimit            75      //设置系统资源上限
116. #define __NR_getrlimit            76      //获取系统资源上限
117. #define __NR_getrusage            77      //获取系统资源使用情况
118. #define __NR_gettimeofday         78      //获取时间和时区
119. #define __NR_settimeofday         79      //设置时间和时区
120. #define __NR_getgroups            80      //获取后补组标志清单
121. #define __NR_setgroups            81      //设置后补组标志清单
122. #define __NR_select               82      //对多路同步 I/O 进行轮询
123. #define __NR_symlink              83      //创建符号链接
124. #define __NR_oldlstat             84      //读取文件状态
125. #define __NR_readlink             85      //读符号链接的值
126. #define __NR_uselib               86      //选择要使用的二进制函数库
127. #define __NR_swapon               87      //打开交换文件和设备
128. #define __NR_reboot               88      //重新启动
129. #define _syscall0(type,name)              //不包含参数的系统调用汇编函数
130. type name(void)
131. {
132. long __res;                               //定义存储变量
133. __asm__ volatile ("int $0x80"             //系统中断 0x80
134. : "=a" (__res)                           //返回值给 eax(__res)
135. : "0" (__NR_##name));
136. if (__res >= 0)                          //如果返回值大于等于 0
137. return (type) __res;                     //则返回该值
138. errno = -__res;                          //否则置出错号  errno
139. return -1;                               //并返回-1
140. }
141. #define _syscall1(type,name,atype,a)     //包含 1 个参数的系统调用汇编宏函数
142. type name(atype a)
143. {
144. long __res;                               //定义存储变量
```

```
145.  __asm__ volatile ("movl %2,%%ebx\n\t"                    //系统中断 0x80
146.  "int $0x80"
147.  : "=a" (__res)                                           //返回值给 eax(__res)
148.  : "0" (__NR_##name),"g" ((long)(a)):"bx");
149.  if (__res >= 0)                                          //如果返回值大于等于 0
150.  return (type) __res;                                     //则返回该值
151.  errno = -__res;                                          //否则置出错号 errno
152.  return -1;                                               //并返回-1
153.  }
154.  #define _syscall2(type,name,atype,a,btype,b)            //包含 2 个参数的系统调用汇编宏
155.  type name(atype a,btype b)
156.  {
157.  long __res;                                             //定义存储变量
158.  __asm__ volatile ("movl %2,%%ebx\n\t"
159.  "int $0x80"                                             //系统中断 0x80
160.  : "=a" (__res)
161.  : "0" (__NR_##name),"g" ((long)(a)),"c" ((long)(b)):"bx");
162.    if (__res >= 0)                                       //如果返回值大于等于 0
163.  return (type) __res;                                     //则返回该值
164.  errno = -__res;                                          //否则置出错号 errno
165.  return -1;                                               //并返回-1
166.  }
167.  #define _syscall3(type,name,atype,a,btype,b,ctype,c)    //包含 3 个参数的系统调用汇编宏函数
168.  type name(atype a,btype b,ctype c)
169.  {
170.  long __res;                                             //定义存储变量
171.  __asm__ volatile ("movl %2,%%ebx\n\t"
172.  "int $0x80"                                             //系统中断 0x80
173.  : "=a" (__res)
174.  : "0" (__NR_##name),"g" ((long)(a)),"c" ((long)(b)),"d" ((long)(c)):"bx");
175.  if (__res >= 0)                                         //如果返回值大于等于 0
176.  return (type) __res;                                     //则返回该值
177.  errno = -__res;                                          //否则置出错号 errno
178.  return -1;                                               //并返回-1
179.  }
180.  #endif                                                  //条件编译结束
181.  extern int errno;                                       //extern errno（错误类型）全局可见
182.  int access(const char * filename, mode_t mode);         //确定文件的可存取性
183.  int acct(const char * filename);                        //启用或禁止进程记账
184.  int alarm(int sec);                                     //设置进程的闹钟
185.  int brk(void * end_data_segment);                       //改变数据段空间的分配
186.  void * sbrk(ptrdiff_t increment);                       //和 brk 类似，增减可用数据空间
187.  int chdir(const char * filename);                       //改变当前工作目录
188.  int chmod(const char * filename, mode_t mode);          //改变文件方式
189.  int chown(const char * filename, uid_t owner, gid_t group); //改变文件属主
190.  int chroot(const char * filename);                      //改变当前工作目录
191.  int close(int fildes);                                  //关闭句柄指向的文件
192.  int creat(const char * filename, mode_t mode);          //创建文件名为 filename 的文件
193.  int dup(int fildes);                                    //复制句柄指向的文件
194.  int execve(const char * filename, char ** argv, char ** envp);   //194～199 详见段后分析
```

```
195. int execv(const char * pathname, char ** argv);
196. int execvp(const char * file, char ** argv);
197. int execl(const char * pathname, char * arg0, ...);
198. int execlp(const char * file, char * arg0, ...);
199. int execle(const char * pathname, char * arg0, ...);
200. volatile void exit(int status);                                    //200～201 详见段后分析
201. volatile void _exit(int status);
202. int fcntl(int fildes, int cmd, ...);                              //文件控制
203. int fork(void);                                                   //创建进程
204. int getpid(void);                                                 //获取进程标识
205. int getuid(void);                                                 //获取用户标识
206. int geteuid(void);                                                /获取有效用户标识
207. int getgid(void);                                                 //获取进程组号
208. int getegid(void);                                                //获取进程有效组号
209. int ioctl(int fildes, int cmd, ...);                              //设备控制
210. int kill(pid_t pid, int signal);                                  //向进程或进程组发信号
211. int link(const char * filename1, const char * filename2);         //创建符号链接
212. int lseek(int fildes, off_t offset, int origin);                  //移动文件指针
213. int mknod(const char * filename, mode_t mode, dev_t dev);         //创建索引结点
214. int mount(const char * specialfile, const char * dir, int rwflag); //卸载文件系统
215. int nice(int val);                                                //改变分时进程的优先级
216. int open(const char * filename, int flag, ...);                   //打开文件
217. int pause(void);                                                  //挂起进程
218. int pipe(int * fildes);                                           //创建管道
219. int read(int fildes, char * buf, off_t count);                    //从 fildes 中读取 nbytes 的 Byte 到 buf
220. int setpgrp(void);                                //将目前进程所属的组识别码设为目前进程的进程识别码
221. int setpgid(pid_t pid,pid_t pgid);                //将参数 pid 指定进程所属组识别码设为参数 pgid 指定
222. int setuid(uid_t uid);                                            //设置用户标识
223. int setgid(gid_t gid);                                            //设置组标识
224. void (*signal(int sig, void (*fn)(int)))(int); //信号表进行修改，程序捕捉信号后进行相应处理
225. int stat(const char * filename, struct stat * stat_buf);         //提供文件名字，获取文件属性
226. int fstat(int fildes, struct stat * stat_buf);                    //通过文件描述符获取文件属性
227. int stime(time_t * tptr);                                         //设置系统时间
228. int sync(void);                                                   //将内存缓冲区数据写回硬盘
229. time_t time(time_t * tloc);                                       //获取系统当前时间
230. time_t times(struct tms * tbuf);                                  //获取进程运行时间
231. int ulimit(int cmd, long limit);                                  //资源限制
232. mode_t umask(mode_t mask);                                        //设置文件权限掩码
233. int umount(const char * specialfile);                            //卸下文件系统
234. int uname(struct utsname * name);                                //显示当前操作系统名称
235. int unlink(const char * filename);                               //删除参数 filename 指定的文件
236. int ustat(dev_t dev, struct ustat * ubuf);                       //获取文件系统信息.
237. int utime(const char * filename, struct utimbuf * times);       //改变文件的访问修改时间
238. pid_t waitpid(pid_t pid,int * wait_stat,int options);           //235～239 详见段后分析
239. pid_t wait(int * wait_stat);
240. int write(int fildes, const char * buf, off_t count);           // write 从 buf 里面读去 nbyes 的字节
241. int dup2(int oldfd, int newfd);
242. int getppid(void);                                              //获得父进程 pid
```

```
243. pid_t getpgrp(void);                                    //获得进程组标识号
244. pid_t setsid(void);                                     //设置会晤标识号
245. int sethostname(char *name, int len);                   //设置主机名称
246. int setrlimit(int resource, struct rlimit *rlp);        //设置系统资源上限
247. int getrlimit(int resource, struct rlimit *rlp);        //获取系统资源上限
248. int getrusage(int who, struct rusage *rusage);          //获取系统资源使用情况
249. int gettimeofday(struct timeval *tv, struct timezone *tz); //获取时间和时区
250. int settimeofday(struct timeval *tv, struct timezone *tz); //设置时间和时区
251. int getgroups(int gidsetlen, gid_t *gidset);           //获取后补组标志清单
252. int setgroups(int gidsetlen, gid_t *gidset);           //设置后补组标志清单
253. int select(int width, fd_set * readfds, fd_set * writefds,
254. fd_set * exceptfds, struct timeval * timeout);        //对多路同步 I/O 进行轮询
255. int swapon(const char * specialfile);                 //打开交换文件和设备
256. #endif                                                 //文件结束
```

代码段中 16～32 行定义了系统的配置参数和关于路径名的配置参数，这些参数是 sysconf()和 pathconf()中用到的符号常数。

代码段中 129～180 行定义了 4 种系统调用嵌入式汇编宏函数（_syscall0～_syscall3），它们的作用是形成相应的系统调用函数原型供程序调用，这些函数原型包括_syscall0、_sys_syscall1、_sys_syscall2、_sys_syscall3。其中_sys_syscallN 后面的数字 N 代表系统调用的参数个数，对应各系统调用的函数原型定义详见 include/linux/sys.h。系统调用根据自己的参数个数，调用相应的_sys_syscallN。

代码段中 2～199 行定义了 exec 函数族，其作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容。这里的可执行文件可以是二进制文件，也可以是可执行的脚本文件。execve()用来执行参数 filename 字符串所代表的文件路径，参数 argv 是传递给执行文件的参数的指针数组，envp 是传递给执行文件的新环境变量数组。 execv()用来执行参数 path 字符串所代表的文件路径，与 execl()不同的地方在于 execve()只需两个参数，第二个参数利用数组指针传递参数给执行文件。execvp()会从 PATH 环境变量所指的目录中查找符合参数 file 的文件名，若找到则执行该文件；然后将第二个参数 argv 传给该欲执行的文件。execl()用来执行参数 path 字符串所代表的文件路径，接下来的参数代表执行该文件时传递过去的 argv(0)、argv[1]……，最后一个参数必须用空指针（NULL）作结束标志。execlp()会从 PATH 环境变量所指的目录中查找符合参数 file 的文件名，找到后便执行该文件，然后将第二个以后的参数当做该文件的 argv[0]、argv[1]……，最后一个参数必须用空指针（NULL）作结束标志。 execle()用来传递环境变量到新进程中，可实现新的子进程的加载和执行。

代码段中 200～201 行定义了 exit()和_exit()。它们都能够完成终止进程的功能，但 exit()在进程结束时，文件描述符号会被关闭并且做一些清理工作，仅保留进程返回状态的信息；然而_exit 在进程结束时，进程直接退出，不会做一些清理工作，也不会被关闭文件描述符。

代码段中 238～239 行定义了 waitpid 和 wait 系统调用。它们都是用于等待子进程中断或结束。系统调用 waitpidwait 比 wait 多出了两个可由用户控制的参数 pid 和 options，从而为程序员编程提供了另一种更灵活的方式。代码段 241 和 193 行定义了 dup2 和 dup。dup2 函数跟 dup 函数都是能够将某个特定的文件描述字重定向输出。但 dup2 函数允许调用者规定一个有效描述符和目标描述符的 id。dup2 函数成功返回时，目标描述符（dup2 函数的第二个参数）将变成源描述符（dup2 函数的第一个参数）的复制品。

4.5 IDT 初始化——_set_gate

汇编函数_set_gate 位于 include/asm/system.h 文件中，用于设置门描述符。_set_gate 通过设置中断向量表，即将中断和 idt_table 联系在一起。在 Linux 操作系统中，有两个地方设置了 idt：一是在系统初始化中，二是在 start_kernel()中。系统初始设置 idt 是通过调用 head.S 中的 set_idt 完成的。

1. 功能描述

汇编函数_set_gate 共输入 5 个数据，%1 与%2 结合分别同 gte_addr 与 (gate_addr+4)相联，gate_addr 是内存单元地址。gte_addr 低 16 位移入%%ax 的低 16 位，这样在%eax 中就形成了所需要的中断门的第一个长整数，其高 16 位为_KERNEL_CS，而低 16 位为 addr 的低 16 位。

中断描述符表的初始化流程如下：第一个输入立即数(0x8000+(dpl<<13)+(type<<8)为%%edx 的低 16 位。这样，%%edx 中高 16 位存储了 ddr 的高 16 位数值，而低 16 位的 P 位为 1（因为是 0x8000)；而 D 位加上类型位段则为 type（因为 type<<8)，其余各位皆为 0，这就形成了中断门中的第 2 个长整数；然后，将%eax 写入 gate_addr，而%%edx 写入*(gate_addr+4)地址中。

2. 代码注释

_set_gate 函数的详细代码注释如下：

```
1.    #define _set_gate(gate_addr,type,dpl,addr)
//这里参数 gate_addr 表示相应 IDT 项的地址。type 表示 IDT 项的 TYPE 字段（15 表示系统门，
//14 表示中断门)。dpl 表示 IDT 项对应的 DPL 值，addr 表示中断处理程序地址
2.    __asm__ ("movw %%dx,%%ax\n\t"                    //%%dx 的低 16 位移入%%ax 的低 16 位
3.    "movw %0,%%dx\n\t"
4.    "movl %%eax,%1\n\t"
5.    "movl %%edx,%2"
6.    :: "i" ((short) (0x8000+(dpl<<13)+(type<<8))),
7.    "m" (*((char *) (gate_addr))),
8.    "m" (*(4+(char *) (gate_addr))),
9.    "d" ((char *) (addr)),"a" (0x00080000)
10.   :"ax","dx")
11.   /* set_intr_gate 用来设置中断门函数，特权级 0，类型 386 中断门*/
12.   #define set_intr_gate(n,addr)
13.   _set_gate(&idt[n],14,0,addr)
14.   /*这个函数设置陷阱门函数，特权级 0，类型 386 陷阱门*/
15.   #define set_trap_gate(n,addr)
16.   _set_gate(&idt[n],15,0,addr)
17.   /*这个函数设置系统调用函数，特权级 3，类型 386 陷阱门*/
18.   #define set_system_gate(n,addr)
19.  _set_gate(&idt[n],15,3,addr)
```

4.6 traps.c 文件

Linux 通过 asm.s 和 traps.c 这两个文件来提供中断处理功能。其中 asm.s 实现了大部分硬件异常所引起的中断处理程序；traps.c 文件位于 kernel 目录下，它为 asm.s 中断处理过程提供了可调用的程序。

1. 功能描述

traps.c 定义和实现了硬件异常中断处理程序，这些处理程序提供给 asm.s 以完成中断处理。

trap.c 中的中断处理需要通过中断描述符对某些中断进行初始化。在了解 trap.c 前，需要了解中断描述符表以及 linux 的中断类型。

中断描述符表（Interrupt Descriptor Table，IDT）将每个异常或中断向量分别与它们的处理过程联系起来。与 GDT 和 LDT 表类似，IDT 也是由 8 字节长描述符组成的一个数组。与 GDT 不同的是，表中第 1 项可以包含描述符。为了构成 IDT 表中的一个索引值，处理器把异常或中断的向量号乘以 8。因为最多只有 256 个中断或异常向量，所以 IDT 无需包含多于 256 个描述符。因为只有可能发生的异常或中断才需要描述符，IDT 中可以含有少于 256 个描述符，并且 IDT 中所有空描述符项应该设置其存在位（标志）为 0。

Linux 中主要中断描述符是：

（1）中断门

中断门包含了一个中断或异常处理程序所在段的选择符和段内偏移量。所有的中断处理程序都由中断门激活。用户态的进程不能访问的一个中断门（特权级为 0），所有的中断都是通过中断门激活的，并且全部在内核态。set_intr_gate 负责在 IDT 表项中插入一个中断门。

（2）系统门

系统门是 Linux 内核特别设置的，用来让用户态的进程能够访问陷阱门，因此，门描述符的 DPL 为 3。用户态的进程可以通过系统门来激活 Linux 异常处理程序，从而实现一个陷阱门（特权级为 3）的访问。set_system_gate 负责在 IDT 表项中插入一个系统门。

（3）陷阱门

陷阱门与中断门类似。陷阱门与中断门唯一的区别在于控制权通过陷阱门进入处理程序时维持 IF 标志位不变，即不关中断。set_trap_gate 负责在 IDT 表项中插入一个陷阱门。

2. 代码注释

traps.c 文件的详细代码注释如下：

```
1.    #include <string.h>                                    //包含字符串文件
2.    #include <linux/head.h>                                //包含头文件
3.    #include <linux/sched.h>                               //包含调度头文件
4.    #include <linux/kernel.h>                              //包含内核头文件
5.    #include <asm/system.h>                                //包含系统文件
6.    #include <asm/segment.h>                               //包含段头文件
7.    #include <asm/io.h>                                    //包含 io 头文件
8.    #define get_seg_byte(seg,addr) ({
9.    register char __res;                                   //定义寄存器变量，用于存储函数返回值
10.   __asm__("push %%fs;mov %%ax,%%fs;movb %%fs:%2,%%al;pop %%fs"
                                                             //将 fs 的内容入栈；然后将 ax 中的值赋
                                                             //给 fs，接着把 fs:(*(addr)所指定的字节
                                                             //放入 al 中。然后恢复 fs 寄存器原内容
11.   :"=a" (__res):"0" (seg),"m" (*(addr)));               //定义输出寄存器和输入寄存器
12.   __res;})                                               //将输出结果赋给__res
13.   #define get_seg_long(seg,addr) ({
14.   register unsigned long __res;                          //定义寄存器变量，存储函数返回值
15.   __asm__("push %%fs;mov %%ax,%%fs;movl %%fs:%2,%%eax;pop %%fs"
                                                             //将 fs 的内容入栈；然后将 ax 中的值赋
                                                             //给 fs，接着把 fs:(*(addr)所指定的字节
                                                             //放入 al 中。然后恢复 fs 寄存器原内容
16.   :"=a" (__res):"0" (seg),"m" (*(addr)));               //定义输出寄存器和输入寄存器
```

```
17.     __res;})                                        //将输出结果赋给__res
18.      /*宏函数_fs 用于读取 fs 段寄存器中的 2byte 的内容*/
19.     #define _fs() ({
20.     register unsigned short __res;                  //定义一个寄存器变量，存储函数返回值
    __asm__("mov %%fs,%%ax":"=a" (__res):);             //将 fs 中的值赋给 ax 段寄存器
21.      __res;})                                       //将输出结果赋给__res
        /*下面是中断处理的函数声明，函数定义详见 kernel/asm.s 或 system_call.s。*/
22.     void page_exception(void);                      //页异常由内存管理程序处理
23.     void divide_error(void);                        //int 0 除法错误
24.     void debug(void);                               //int1  调试
25.     void nmi(void);                                 //int2 Non Maskable Interrupt
26.     void int3(void);                                //int 3    软中断
27.     void overflow(void);                            //int4  溢出
28.     void bounds(void);                              //int5  越界
29.     void invalid_op(void);                          //int6  无效操作指令
30.     void device_not_available(void);                //int7  设备不存在
31.     void double_fault(void);                        //int8  双出错故障
32.     void coprocessor_segment_overrun(void);         //int9  协处理器段溢出
33.     void invalid_TSS(void);                         //int10  无效 TSS
34.     void segment_not_present(void);                 //int11  缺段中断
35.     void stack_segment(void);                       //int12  堆栈异常
36.     void general_protection(void);                  //int13  一般保护异常
37.     void page_fault(void); /                        //int14  页故障
38.     void coprocessor_error(void);                   //int16 协处理器错误
39.     void reserved(void);                            //int15  保留
40.     void parallel_interrupt(void);                  //int39 并行口中断
41.     void irq13(void);                               //int45 协处理器中断
42.     void alignment_check(void);                     //int17 对其检查中断
43.     int send_sig(long, struct task_struct *, int);  //用于进程通信
44.     static void die(char * str,long esp_ptr,long nr)        //die()用来打印出错中断的名称
45.     {
46.     long * esp = (long *) esp_ptr;                  //取堆栈段堆栈指针寄存器的值
47.     int i;
48.     printk("%s: %04x\n\r",str,nr&0xffff);           //打印错误信息，其中 str 是错误信息、nr
                                                        //是错误码，nr&0xffff 保证错误码是 2byte
49.     printk("EIP:      %04x:%p\nEFLAGS: %p\n", 0xffff & esp[1],esp[0],esp[2]);
                                                        //打印执行中断处理程序之前的寄存器
50.     if ((0xffff & esp[1]) == 0xf)
51.     printk("ESP:      %04x:%p\n",0xffff & esp[4],esp[3]);
                                                        //如果 cs 为 0xf, 则打印 ss 和 esp 寄存器
52.     printk("fs: %04x\n",_fs());                     //打印 fs 寄存器的值
53.     printk("base: %p, limit: %p\n",get_base(current->ldt[1]),get_limit(0x17));
    //打印描述符中段限长值和中断描述符表的段基地址。0x17 表示局部描述符表中，特权级为 3，
   //索引值为 2 的段选择符，即数据段
54.     if ((0xffff & esp[1]) == 0xf) {
55.     printk("Stack: ");
56.     for (i=0;i<4;i++)
57.     printk("%p ",get_seg_long(0x17,i+(long *)esp[3]));
58.     printk("\n");
59.     }                                   //如果 cs 寄存器值为 0xf, 则打印从该堆栈开始的 16bytes 的内容
```

```
60.     str(i);                                                          //str()用于获取取当前任务号
61.     printk("Pid: %d, process nr: %d\n\r",current->pid,0xffff & i);   //打印当前 Pid 和任务号
62.     for(i=0;i<10;i++)
63.     printk("%02x ",0xff & get_seg_byte(esp[1],(i+(char *)esp[0])));  //打印代码段前 10bytes
64.     printk("\n\r");
65.     do_exit(11);                                                     //程序退出处理
66.     }
/*以下是各个中断处理操作的函数实现，这些函数以 do_开头，其后是相应的中断名，如
double_fault，并被 asm.s 中的中断处理函数被调用。*/
     /* do_double_fault 双出错故障中断处理*/
67.     void do_double_fault(long esp, long error_code)
68.     {
69.     die("double fault",esp,error_code);                              //打印双出错故障信息
70.     }
     /* do_general_protection 一般保护异常中断处理*/
71.     void do_general_protection(long esp, long error_code)
72.     {
73.     die("general protection",esp,error_code);                        //打印一般保护异常信息
74.     }
     /* do_alignment_check 对齐检查中断处理*/
75.     void do_alignment_check(long esp, long error_code)
76.     {
77.     die("alignment check",esp,error_code);                           //打印对齐检查异常信息
78.     }
     /* do_divide_error 除法错误中断处理*/
79.     void do_divide_error(long esp, long error_code)
80.     {
81.     die("divide error",esp,error_code);                              //打印除法故障信息
82.     }
     /* do_int3 软中断陷阱处理*/
83.     void do_int3(long esp, long error_code)
84.     {
85.     send_sig(SIGTRAP, current, 0);                                   //给当前进程发送 SIGTRAP 信号
86.     }
     /* do_nmi 不可屏蔽中断处理*/
87.     void do_nmi(long esp, long error_code)
88.     {
89.             die("nmi",esp,error_code);                               //打印不可屏蔽故障信息
90.     }
     /* do_debug 调试中断处理*/
91.     void do_debug(long esp, long error_code)
92.     {
93.     send_sig(SIGTRAP, current, 0);                                   //打印调试故障中断信息
94.     }
     /* do_overflow 溢出陷阱中断处理*/
95.     void do_overflow(long esp, long error_code)
96.     {
97.     die("overflow",esp,error_code);                                  //打印溢出陷阱中断信息
98.     }
     /* do_bounds 访问越界故障中断处理*/
```

```
99.     void do_bounds(long esp, long error_code)
100.    {
101.    die("bounds",esp,error_code);                          //打印访问越界故障中断信息
102.    }
        /* do_invalid_op 无效指令故障中断处理*/
103.    void do_invalid_op(long esp, long error_code)
104.    {
105.    die("invalid operand",esp,error_code);                 //打印无效指令故障中断信息
106.    }
        /* do_device_not_available 设备不存在中断处理*/
107.    void do_device_not_available(long esp, long error_code)
108.    {
109.    die("device not available",esp,error_code);            //打印设备不存在故障中断信息
110.    }
         /* do_coprocessor_segment_overrun 协处理器段溢出故障中断处理*/
111.    void do_coprocessor_segment_overrun(long esp, long error_code)
112.    {
113.    die("coprocessor segment overrun",esp,error_code);     //打印协处理器段溢出故障中断信息
114.    }
         /* do_invalid_TSS 无效 TSS 故障中断处理*/
115.    void do_invalid_TSS(long esp,long error_code)
116.    {
117.    die("invalid TSS",esp,error_code);
118.    }
         /* do_segment_not_present 缺段中断故障中断处理*/
119.    void do_segment_not_present(long esp,long error_code)
120.    {
121.    die("segment not present",esp,error_code);             //打印无效 TSS 故障中断信息
122.    }
         /* do_stack_segment 堆栈故障中断处理*/
123.    void do_stack_segment(long esp,long error_code)
124.    {
125.    die("stack segment",esp,error_code);                   //打印堆栈故障中断信息
126.    }
        /* do_coprocessor_error 协处理器故障中断处理*/
127.    void do_coprocessor_error(long esp, long error_code)
128.    {
129.    if (last_task_used_math != current)
130.    return;
131.    die("coprocessor error",esp,error_code);               //打印协处理器故障中断信息
132.    }
        /* do_reserved  保留中断处理*/
133.    void do_reserved(long esp, long error_code)
134.    {
135.    die("reserved (15,17-47) error",esp,error_code);
136.    }
   /*trap_init 为初始化中断处理程序，用于设置中断调用门，在 main.c 中被调用*/
137.    void trap_init(void)
138.    {
139.    int i;
```

```
140.    set_trap_gate(0,&divide_error);                    //设置 int0 为除法中断
141.    set_trap_gate(1,&debug);                           //设置 int1 为调试中断
142.    set_trap_gate(2,&nmi);
143.    set_system_gate(3,&int3);
144.    set_system_gate(4,&overflow);
145.    set_system_gate(5,&bounds);
146.    set_trap_gate(6,&invalid_op);                      //以下类似
147.    set_trap_gate(7,&device_not_available);
148.    set_trap_gate(8,&double_fault);
149.    set_trap_gate(9,&coprocessor_segment_overrun);
150.    set_trap_gate(10,&invalid_TSS);
151.    set_trap_gate(11,&segment_not_present);
152.    set_trap_gate(12,&stack_segment);
153.    set_trap_gate(13,&general_protection);
154.    set_trap_gate(14,&page_fault);
155.    set_trap_gate(15,&reserved);
156.    set_trap_gate(16,&coprocessor_error);
157.    set_trap_gate(17,&alignment_check);
158.    for (i=18;i<48;i++)
159.    set_trap_gate(i,&reserved);
160.    set_trap_gate(45,&irq13);                          //设置 int45 为协处理器中断陷阱门
161.    outb_p(inb_p(0x21)&0xfb,0x21);                     //允许主 8259A 芯片 IRQ2 中断请求
162.    outb(inb_p(0xA1)&0xdf,0xA1);                       //允许从 8259A 芯片 IRQ13 中断请求
163.    set_trap_gate(39,&parallel_interrupt);             //设置 int39 为并行口中断
164.    }
```


traps.c 用来处理硬件陷阱和故障，目前主要用于调试。代码段中 8～12 定义了汇编函数 get_seg_byte，它用于获得段寄存器中 addr 地址 1 byte 长度的数据。其执行的汇编过程是：首先将 fs 段寄存器的内容入栈；然后将 ax 中的值给 fs；接着把 fs:(*(addr)所指定的字节放入 al 寄存器中并恢复 fs 寄存器原内容；汇编语句执行完后，输出寄存器 eax 的值将被放入__res，作为该宏函数的返回值。代码段中 13～17 行定义了 get_seg_long 函数，它的执行过程和 get_seg_byte 类似，但是 get_seg_long 在执行完后会返回寄存器中 addr 地址的 4 byte 长度的数据。

代码段 67～136 行，在实现的 do_XX 函数里调用了 die 函数。该函数是用来打印出错中断的名称、出错号、调用程序的 EIP、EFLAGS、ESP、fs 值、段基址、段长度、pid、任务号、10 byte 指令码。如果堆栈在用户数据段，则还打印 16 byte 的堆栈内容。代码段中 143～145 行定义了 int3～int5 系统陷阱门。异常处理函数插入 IDT 的表项是由 set_trap_gate()和 set_system_gate()函数来完成的。用户态的进程不能访问的一个陷阱门（特权级为 0），大部分 Linux 异常处理程序通过陷阱门激活并由 set_trap_gate ()函数负责在 IDT 表项中插入一个陷阱门。与陷阱门不同，系统陷阱门的特权级是 3，即系统陷阱门设置的中断处理过程能够被所有进程调用（如单步调试、溢出出错和超出边界出错等）。set_trap_gate()与 set_system_gate()的主要区别在于设置的特权级不同（前者特权级为 0，后者特权级为 3）。因此断点陷阱中断 int3、溢出中断 overflow 和边界出错中断 bounds 可由任何程序产生。

4.7 sys.c 文件

sys.c 文件位于 linux/kernel 目录下，其包含了在 sys.h 头文件中所声名的系统调用功能函数的具体实现。

sys.c 位于 linux/kernel 目录下，主要实现系统调用函数功能。在 linux 0.95 内核中，并没有真正的实现全部的系统调用的功能。对于未实现的系统调用，系统除了返回 ENOSYS 之外，不做任何其他的工作。这些可方便扩展对未实现的系统调用的功能。sys.c 文件的详细代码注释如下：

```
1.      #include <errno.h>                                  //包含错误头文件
2.      #include <linux/sched.h>                            //包含调度头文件
3.      #include <linux/tty.h>                              //包含 tty 头文件
4.      #include <linux/kernel.h>                           //包含内核头文件
5.      #include <linux/config.h>                           //包含配置头文件
6.      #include <asm/segment.h>                            //包含内段头文件
7.      #include <sys/times.h>                              //包含时间头文件
8.      #include <sys/utsname.h>                            //包含系统名称结构头文件
9.      #include <sys/param.h>                              //包含参数头文件
10.     #include <sys/resource.h>                           //包含资源头文件
11.     #include <string.h>                                 //包含字符串头文件
12.     static int C_A_D = 1;                               //是否可以用 ctrl-alt-del 重启，默认的情况
                                                              是可以的
13.     struct timezone sys_tz = { 0, 0};                   //系统所在地方的时区
14.     extern int session_of_pgrp(int pgrp);
          /* 未实现 sys_ftime、sys_break、sys_stty、sys_gtty、sys_prof 的系统调用*/
15.     int sys_ftime()
16.     {
17.     return -ENOSYS;
18.     }
19.     int sys_break()
20.     {
21.     return -ENOSYS;
22.     }
23.     int sys_stty()
24.     {
25.     return -ENOSYS;
26.     }
27.     int sys_gtty()
28.     {
29.     return -ENOSYS;
30.     }
31.     int sys_prof()
32.     {
33.     return -ENOSYS;
34.     }
35.     extern void hard_reset_now(void);
        /* sys_reboot   系统重启调用实现*/
36.     int sys_reboot(int magic, int magic_too, int flag)
37.     {
38.     if (!suser())                                       //若非超级用户（根用户）
39.     return -EPERM;                                      //则返回操作没有许可标志（EPERM）
40.     if (magic != 0xfee1dead || magic_too != 672274793)  //根据 magic 和 magic_too 值判断
41.     return -EINVAL;                                     //则返回参数无效标志（EINVAL）
42.     if (flag == 0x01234567)                             //若 flag=0x01234567
```

```
43.     hard_reset_now();                          //调用硬重启函数
44.     else if (flag == 0x89ABCDEF)               //若 flag == 0x89ABCDEF
45.     C_A_D = 1;                                 //则设置 C_A_D 标志
46.     else if (!flag)                            //若 flag == 0
47.     C_A_D = 0;                                 //则复位 C_A_D 标志
48.     else                                       //若 flag 为其他值
49.     return -EINVAL;                            //则返回参数无效标志（EINVAL）
50.     return (0);
51.     }
52.     void ctrl_alt_del(void)
53.     {
54.     if (C_A_D)                                 //根据 C_A_D 标志判断是否立即重启
55.     hard_reset_now();
56.     }
        /* sys_setregid   设置当前任务的实际或者有效组 ID（gid）*/
57.     int sys_setregid(int rgid, int egid)
58.     {
59.     if (rgid>0) {                              //若实际 gid 有效
60.     if ((current->gid == rgid) ||              //若当前任务的 gid 同实际 gid 相同
61.     suser())                                   //超级用户特权
62.     current->gid = rgid;                       //则设置当前 gid 为实际 gid
63.     else                                       //否则返回操作没有许可标志
64.     return(-EPERM);
65.     }
66.     if (egid>0) {                              //若有效组 gid 有效
67.     if ((current->gid == egid) ||              //若 current->gid == egid 并 current->egid ==
                                                   //egid 并且有超级用户特权
68.     (current->egid == egid) ||
69.     suser()) {
70.     current->egid = egid;                      //则设置当前有效 gid 和保留 gid
71.     current->sgid = egid;
72.     } else                                     //否则返回操作没有许可标志
73.     return(-EPERM);
74.     }
75.     return 0;
76.     }
    /* sys_setgid  设置进程组号（gid）*/
77.     int sys_setgid(int gid)
78.     {
79.       if (suser())                             //如果任务有超级用户特权
80.     current->gid = current->egid = current->sgid = gid;
                                                   //则实际 gid、有效 gid 和保留 gid 都被
                                                   //设置成参数指定的 gid
81.     else if ((gid == current->gid) || (gid == current->sgid)) //若 gid 同当前 gid 或同当前保留 gid 相等
82.     current->egid = gid;                       //则将参数 gid 置成有效 gid
83.     else
84.     return -EPERM;                             //否则返回操作没有许可标志
85.     return 0;
86.     }
        /* sys_acct、sys_phys、sys_lock、sys_mpx、sys_ulimit 的系统调用未实现 */
```

```
87.     int sys_acct()
88.     {
89.     return -ENOSYS;
90.     }
91.     int sys_phys()
92.     {
93.     return -ENOSYS;
94.     }
95.     int sys_lock()
96.     {
97.     return -ENOSYS;
98.     }
99.     int sys_mpx()
100.    {
101.    return -ENOSYS;
102.    }
103.    int sys_ulimit()
104.    {
105.    return -ENOSYS;
106.    }
        /* sys_time   计算从 1970 年 1 月 1 日 00:00:00 GMT 开始计时的时间值（秒）*/
107.    int sys_time(long * tloc)
108.    {
109.    int i;
110.    i = CURRENT_TIME;
111.    if (tloc) {
112.    verify_area(tloc,4);                    //验证是否有足够内存（4 字节）
113.    put_fs_long(i,(unsigned long *)tloc);   //将时间值存储在 tlocz 中
114.    }
115.    return i;                               //返回时间值
116.    }
    /* sys_setreuid 设置任务的实际或者有效用户 ID（uid）*/
117.    int sys_setreuid(int ruid, int euid)
118.    {
119.    int old_ruid = current->uid;
120.    if (ruid>0) {
121.    if ((current->euid==ruid) ||
122.    (old_ruid == ruid) ||                   //若 ruid 是当前任务的有效 uid 或者与当前任
123.    suser())                                //务的 uid 相同或任务具有超级用户的特权
124.    current->uid = ruid;                    //则设置当前任务的 uid 为 ruid
125.    else                                    //否则返回操作没有许可标志
126.    return(-EPERM);
127.    }
128.    if (euid>0) {
129.    if ((old_ruid == euid) ||
130.    (current->euid == euid) ||              //若 euid 是当前任务的 uid 或者与当前任务
131.    suser()) {                              //务的 euid 相同或任务具有超级用户的特权
132.    current->euid = euid;                   //则任务的 euid 和 suid 设置同值
133.    current->suid = euid;
134.    } else {                                //否则
```

```
135.    current->uid = old_ruid;                          //则设置任务 uid 为 old_ruid
136.    return(-EPERM);
137.    }
138.    }
139.    return 0;
140.    }
        /* sys_setuid   设置用户 id*/
141.    int sys_setuid(int uid)
142.    {
143.    if (suser())                                      //若是超级用户，可根据参数设置 uid、euid 以及 suid
144.    current->uid = current->euid = current->suid = uid;
145.    else if ((uid == current->uid) || (uid == current->suid))
                                                          //若没有超级用户的权限并且 uid 是当前任务的 uid 或者 suid
146.    current->euid = uid;                              //则将 uid 设置成当前任务的有效 uid
147.    else                                              //否则返回操作没有许可标志
148.    return -EPERM;
149.    return(0);
150.    }
        /* sys_stime   设置系统时间和日期*/
151.    int sys_stime(long * tptr)
152.    {
153.    if (!suser())                                     //只有超级用户才有权限设置时间
154.    return -EPERM;
155.    startup_time = get_fs_long((unsigned long *)tptr) - jiffies/HZ;//计算开始时间
156.    jiffies_offset = 0;
157.    return 0;
158.    }
    /* sys_times   获取当前任务时间*/
159.    int sys_times(struct tms * tbuf)
160.    {
161.    if (tbuf) {
162.    verify_area(tbuf,sizeof *tbuf);                   //验证内存的容量
163.    put_fs_long(current->utime,(unsigned long *)&tbuf->tms_utime);   //获取用户时间
164.    put_fs_long(current->stime,(unsigned long *)&tbuf->tms_stime);   //获取系统时间
165.    put_fs_long(current->cutime,(unsigned long *)&tbuf->tms_cutime); //获取子进程用户时间
166.    put_fs_long(current->cstime,(unsigned long *)&tbuf->tms_cstime); //获取子进程系统时间
167.    }
168.    return jiffies;
169.    }
    /* sys_brk   修改数据段长度*/
170.    int sys_brk(unsigned long end_data_seg)
171.    {
172.    if (end_data_seg >= current->end_code &&          //若进程没有超过其代码结尾并且
173.    end_data_seg < current->start_stack - 16384)      //有足够的内存容量，
174.    current->brk = end_data_seg;                      //设置数据段末尾为 end_data_seg
175.    return current->brk;
176.    }
     /* sys_setpgid   设置进程的组 id（与 uid 不同，此为进程 id）*/
177.    int sys_setpgid(int pid, int pgid)
178.    {
```

```
179.    int i;
180.    if (!pid)                                   //进程 id 为 0 时，置 pid 为 current->pid
181.    pid = current->pid;
182.    if (!pgid)                                  //进程组 id 为 0 时，置 pgid 为 current->pid
183.    pgid = current->pid;
184.    if (pgid < 0)                               //若进程组 id 为负数，返回无效参数标志
185.    return -EINVAL;
186.    for (i=0 ; i<NR_TASKS ; i++)                //遍历任务数组，查找指定进程号的任务
187.    if (task[i] && (task[i]->pid == pid) &&
188.    ((task[i]->p_pptr == current) ||
189.    (task[i] == current))) {
190.    if (task[i]->leader)                        //若任务 leader 标志不为 0（已加入组），
191.    return -EPERM;                              //则返回 EPERM
192.    if ((task[i]->session != current->session) ||
193.    ((pgid != pid) &&
194.    (session_of_pgrp(pgid) != current->session)))
195.    return -EPERM;
196.    task[i]->pgrp = pgid;                       //设置进程组
197.    return 0;
198.    }
199.    return -ESRCH;                              //否则返回进程不存在标志（ESRCH）
200.    }
        /* sys_getpgrp   获取当前进程的进程组 id*/
201.    int sys_getpgrp(void)
202.    {
203.    return current->pgrp;
204.    }
        /* sys_setsid  设置当前进程的会话 id*/
205.    int sys_setsid(void)
206.    {
207.    if (current->leader && !suser())            //若当前进程已是会话首领且非超级用户，
208.    return -EPERM;                              //返回操作没有许可标志
209.    current->leader = 1;                        //设置当前进程为会话首领
210.    current->session = current->pgrp = current->pid;   //设置该进程是会话首领和进程组首领
211.    current->tty = -1;                          //失去控制终端
212.    return current->pgrp;                       //返回当前进程的进程组号
213.    }
        /* sys_getgroups  取当前进程组号，放到 grouplist 中*/
214.    int sys_getgroups(int gidsetsize, gid_t *grouplist)
215.    {
216.    int i;
217.    if (gidsetsize)                             //若存在进程组（用户组个数大于 0），
218.    verify_area(grouplist, sizeof(gid_t) * gidsetsize);   //验证组空间是否有 gidsetsize 的容量
219.    for (i = 0; (i < NGROUPS) && (current->groups[i] != NOGROUP);
220.    i++, grouplist++) {                         //遍历进程组，查找当前进程组
221.    if (gidsetsize) {
222.    if (i >= gidsetsize)                        //若进程组个数小于 i，
223.    return -EINVAL;                             //则返回参数无效标志
224.    put_fs_word(current->groups[i], (short *) grouplist);   //输出组号到组列表（grouplist）
225.    }
```

```
226.    }
227.    return(i);
228.    }
        /* sys_setgroups 与 sys_getgroups 相反，用来设置当前任务进程组号*/
229.    int sys_setgroups(int gidsetsize, gid_t *grouplist)
230.    {
231.    int i;
232.    if (!suser())                                    //只有超级用户权限才能这是进程组
233.    return -EPERM;
234.    if (gidsetsize > NGROUPS)                        //若进程组的个数超出最大限制，
235.    return -EINVAL;                                  //则返回参数无效标志
236.    for (i = 0; i < gidsetsize; i++, grouplist++) {  //遍历组列表并设置当前进程的组号
237.    current->groups[i] = get_fs_word((unsigned short *) grouplist);
238.    }
239.    if (i < NGROUPS)                                 //将当前进程中没有设置组号的表项
240.    current->groups[i] = NOGROUP;                    //（groups[i]）设置成 NOGROUP 标志
241.    return 0;
242.    }
    /* in_group_p(gid_t grp) 检查当前进程是否在指定的用户组 grp 中*/
243.    int in_group_p(gid_t grp)
244.    {
245.    int i;
246.    if (grp == current->egid)                        //若当前进程的有效组号和 grp 相同，则
247.    return 1;                                        //说明该进程是属于 grp 组
248.    for (i = 0; i < NGROUPS; i++) {                  //遍历该进程的所有组号
249.    if (current->groups[i] == NOGROUP)               //若遍历到无组号的表项，则说明该进程
250.    break;                                           //不包含在指定的进程组中。退出循环
251.    if (current->groups[i] == grp)                   //若找到，则返回 1（当前任务属于 grp）
252.    return 1;
253.    }
254.    return 0;
255.    }
256.    static struct utsname thisname = {
257.    UTS_SYSNAME, UTS_NODENAME, UTS_RELEASE, UTS_VERSION,   UTS_MACHINE
258.    };
    //定义系统名：当前操作系统的名称，主机名，发行级别，版本号，硬件类型名称（详见 config.h）
        /* sys_uname 获取系统信息*/
259.    int sys_uname(struct utsname * name)
260.    {
261.    int i;
262.    if (!name) return -ERROR;                        //若参数为空（未分配内存），返回错误
263.    verify_area(name,sizeof *name);
264.    for(i=0;i<sizeof *name;i++)
265.    put_fs_byte(((char *) &thisname)[i],i+(char *) name); //获取系统信息放到 name 中
266.    return 0;
267.    }
        /* sys_sethostname 设置主机名 */
268.    int sys_sethostname(char *name, int len)
269.    {
270.    int i;
```

```
271.    if (!suser())                                   //只有超级用户才有权限设置主机名
272.    return -EPERM;
273.    if (len > MAXHOSTNAMELEN)                       //若欲设置的主机名过长（大于最大限制）
274.    return -EINVAL;                                 //则返回错误
275.    for (i=0; i < len; i++) {                       //根据提供的参数设置系统的主机名
276.    if ((thisname.nodename[i] = get_fs_byte(name+i)) == 0)
277.    break;
278.    }
279.    if (thisname.nodename[i]) {                     //在主机名的字符串中加结束标志
280.    thisname.nodename[i>MAXHOSTNAMELEN ? MAXHOSTNAMELEN : i] = 0;
281.    }
282.    return 0;
283.    }
          /* sys_getrlimit  取当前进程指定资源限制*/
284.    int sys_getrlimit(int resource, struct rlimit *rlim)
285.    {
286.    if (resource >= RLIM_NLIMITS)                   // RLIM_NLIMITS 在 resource.h 中定义
287.    return -EINVAL;                                 //若资源数超出限制，则返回错误
288.    verify_area(rlim,sizeof *rlim);
289.    put_fs_long(current->rlim[resource].rlim_cur,   //取当前进程资源的软限制（rlim_cur）
290.    (unsigned long *) rlim);
291.    put_fs_long(current->rlim[resource].rlim_max,   //取当前进程资源的硬限制（rlim_max）
292.    ((unsigned long *) rlim)+1);
293.    return 0;
294.    }
      /* sys_setrlimit  设置当前进程指定资源的限制*/
295.    int sys_setrlimit(int resource, struct rlimit *rlim)
296.    {
297.    struct rlimit new, *old;
298.    if (resource >= RLIM_NLIMITS)                   //若资源数超出限制，则返回错误
299.    return -EINVAL;
300.    old = current->rlim + resource;
301.    new.rlim_cur = get_fs_long((unsigned long *) rlim);   //设置指定资源的软界限
302.    new.rlim_max = get_fs_long(((unsigned long *) rlim)+1); //设置指定资源的硬界限
303.    if (((new.rlim_cur > old->rlim_max) ||
304.    (new.rlim_max > old->rlim_max)) &&
305.    !suser())                                       //若指定资源软硬界限超出当前任务资源
306.    return -EPERM;                                  //界限或者权限不够，则错误返回
307.    *old = new;
308.    return 0;
309.    }
          /* sys_getrusage  获取指定进程的资源利用信息*/
310.    int sys_getrusage(int who, struct rusage *ru)
311.    {
312.    struct rusage r;
313.    unsigned long *lp, *lpend, *dest;
314.    if (who != RUSAGE_SELF && who != RUSAGE_CHILDREN)
      // RUSAGE_SELF  和 RUSAGE_CHILDREN 见 resource.h。
315.    return -EINVAL;
316.    verify_area(ru, sizeof *ru);
```

```
317.    memset((char *) &r, 0, sizeof(r));
318.    if (who == RUSAGE_SELF) {                          //复制当前进程资源利用信息
319.    r.ru_utime.tv_sec = CT_TO_SECS(current->utime);
320.    r.ru_utime.tv_usec = CT_TO_USECS(current->utime);
321.    r.ru_stime.tv_sec = CT_TO_SECS(current->stime);
322.    r.ru_stime.tv_usec = CT_TO_USECS(current->stime);
323.    } else {                                           //复制当前进程的子进程资源利用信息
324.    r.ru_utime.tv_sec = CT_TO_SECS(current->cutime);
325.    r.ru_utime.tv_usec = CT_TO_USECS(current->cutime);
326.    r.ru_stime.tv_sec = CT_TO_SECS(current->cstime);
327.    r.ru_stime.tv_usec = CT_TO_USECS(current->cstime);
328.    }
329.    lp = (unsigned long *) &r;
330.    lpend = (unsigned long *) (&r+1);
331.    dest = (unsigned long *) ru;
332.    for (; lp < lpend; lp++, dest++)
333.    put_fs_long(*lp, dest);
334.    return(0);
335.    }
       /* sys_gettimeofday 获取指定进程的资源利用信息*/
336.    int sys_gettimeofday(struct timeval *tv, struct timezone *tz)
337.    {
338.    if (tv) {
339.    verify_area(tv, sizeof *tv);
340.    put_fs_long(startup_time + CT_TO_SECS(jiffies+jiffies_offset),
341.    (unsigned long *) tv);                             //获取由凌晨开始算起的秒数
342.    put_fs_long(CT_TO_USECS(jiffies+jiffies_offset),
343.    ((unsigned long *) tv)+1);                         //获取由凌晨开始算起的微秒数
344.    }
345.    if (tz) {
346.    verify_area(tz, sizeof *tz);
347.    put_fs_long(sys_tz.tz_minuteswest, (unsigned long *) tz); //取同 Greenwich 的时差（差多少分钟）
348.    put_fs_long(sys_tz.tz_dsttime, ((unsigned long *) tz)+1); //取日光节约时间的状态
349.    }
350.    return 0;
351.    }
       /* sys_settimeofday 设置指定进程的资源利用信息*/
352.    int sys_settimeofday(struct timeval *tv, struct timezone *tz)
353.    {
354.    static int firsttime = 1;
355.    void        adjust_clock();
356.    if (!suser())                                      //必须具有超级用户权限，才能设置时间
357.    return -EPERM;
358.    if (tz) {
359.    sys_tz.tz_minuteswest = get_fs_long((unsigned long *) tz);  //设置同 Greenwich 的时差
360.    sys_tz.tz_dsttime = get_fs_long(((unsigned long *) tz)+1);  //设置日光节约时间的状态
361.    if (firsttime) {                                   //如果是第一次设置，
362.    firsttime = 0;                                     //则调整时间为 GMT 时间
363.    if (!tv)
364.    adjust_clock();
```

```
365.      }
366.      }
367.      if (tv) {
368.      int sec, usec;
369.      sec = get_fs_long((unsigned long *)tv);                    //设置由凌晨开始算起的秒数
370.      usec = get_fs_long(((unsigned long *)tv)+1);               //设置由凌晨开始算起的微秒数
371.      startup_time = sec - jiffies/HZ;
372.      jiffies_offset = usec * HZ / 1000000 - jiffies%HZ;
373.      }
374.      return 0;
375.      }
      /* adjust_clock  调整系统启动时间为以 GMT 为标准的时间*/
376.      void adjust_clock()
377.      {
378.      startup_time += sys_tz.tz_minuteswest*60;
379.      }
      /* sys_umask  设置当前进程创建文件属性屏蔽码*/
380.      int sys_umask(int mask)
381.      {
382.      int old = current->umask;
383.      current->umask = mask & 0777;                              //设置屏蔽码为 mask & 0777
384.      return (old);
385.      }
```


代码段中 36～51 行是用于重启系统调用的程序。只有 root 用户才有权限去调用该程序。并且 root 用户需要设置寄存器中的一些特殊的字段，以便某些错误将不会使机器重启。重启不是同步操作，需要在调用前完成自己的工作。

代码段中 52～56 行定义的 ctrl_alt_del 函数是对键盘 ctrl-alt-del 的中断响应。若它在一个中断中被调用，则这个函数不是同步的，这时系统要么立即重启，要么忽略掉这个中断。

代码段中 57～76 行定义的程序是根据 BSD 的风格实现的。它没有考虑保留的 gid，但是如果设置了有效的 gid，保留的 gid 也被设置成了同值。这会使 setgid 程序完全放弃自己的权限，有利于对进程做安全审核。

代码段中 117～140 行定义了 sys_setreuid。通过 sys_setreuid，无特权的用户可将 ruid 改成 effective uid。

代码段中 141～150 行定义了 sys_setuid 用来设置用户 id。setuid()按照 SysV w/ SAVED_IDS 的方式实现的。需要注意，SAVED_ID's 同 sendmail 一样，它无法设置根用户的用户 id。因为根用户在设置 uid 的同时也更改了保留的 uid，故它不可能将自己设置成普通用户，然后再更改成根用户。需要注意， BSD 风格的 setreuid 允许根用户修改自己的权限，并且可通过实际的 uid 和有效的 uid 重新获得原来的权限。

代码段 310～335 行定义了 sys_getrusage 函数。该函数将 rusuage 数据结构放到任务数据结构中，因为这个数据结构会使 task_struct 的尺寸变得非常大。这使得任务数据结构的其他信息变得更简单。

代码段 352～375 行定义了 sys_settimeofday 函数，其用于时间设置。在第一次设置时区时，为了时钟运行 GMT 时间而不是本地的时间，需要 sys_settimeofday 进行时间调整。在设置时区时，程序会知道系统所在时区。这个工作应在启动/etc/rc 脚本的时候完成，要尽可能的快速，以便时

间能够被设置正确，否则，不同的程序会因调整时间出现混乱。

代码段中 376～379 定义了 adjust_clock，用于调整系统启动时间。以 CMOS 而不是本地的时间为基准进行时间的调整。否则我们要么需要编写一个在/etc/rc 下运行的程序（如果程序出现混乱，很难准确调正时钟），要么将时区信息编译至内核。

4.8 system_call.s 文件

system_call.s 文件位于 kernel 目录下。它是用汇编语言实现的代码文件。system_call.s 负责系统调用（system_call）中断 int 0x80 的入口处理过程以及信号检测处理，提供给系统调用底层处理的程序实现。

1. 功能描述

Linux 是通过 int 0x80 中断来实现内核调用的功能。实现系统调用的相关文件包括 system_call.s、fork.c、signal.c、sys.c 和 exit.c 文件。system_call.s 位于 kernel 目录下，主要实现系统调用中断 int 0x80 的入口处理过程以及信号检测处理。system_call.s 文件包含系统调用底层处理子程序。由于有些代码比较类似，所以 system_call.s 也包括时钟中断处理（timer-interrupt）句柄，硬盘和软盘的中断处理程序也在这里。

2. 代码分析

sys_call.s 文件的详细代码注释如下：

```
1.      SIG_CHLD    = 17                                    //定义子进程停止或结束信号
2.      /* 3～19 行定义用于 Stack 中保存寄存器的值的位置*/
3.      EBX             = 0x00
4.      ECX             = 0x04
5.      EDX             = 0x08
6.      ESI             = 0x0C
7.      EDI             = 0x10
8.      EBP             = 0x14
9.      EAX             = 0x18
10.     DS              = 0x1C
11.     ES              = 0x20
12.     FS              = 0x24
13.     GS              = 0x28
14.     ORIG_EAX    = 0x2C
15.     EIP             = 0x30
16.     CS              = 0x34
17.     EFLAGS      = 0x38
18.     OLDESP      = 0x3C
19.     OLDSS       = 0x40
20.     /*21～26 行是任务结构(task_struct)中变量的偏移值*/
21.     state     = 0          # these are offsets into the task-struct.   /进程状态码
22.     counter  = 4                                                       //任务运行时间计数（滴答数）
23.     priority = 8                                                       //进程优先级
24.     signal    = 12                                                     //信号位图，每个比特位代表一种信号
25.     sigaction = 16# MUST be 16 (=len of sigaction)                     // sigaction 长度必须是 16 字节
26.     blocked = (33*16)                                                  //受阻塞信号位图的偏移量
    /*28～32 行定义各数据在 sigaction 结构中的偏移量*/
27.     # offsets within sigaction
```

```
28.     sa_handler = 0                                          //信号处理过程的句柄
29.     sa_mask = 4                                             //信号屏蔽码
30.     sa_flags = 8                                            //信号集
31.     sa_restorer = 12                                        //恢复函数指针，参见 kernel/signal.c
32.     nr_system_calls = 82
    // Linux 0.95 和 Linux 0.12 内核中的系统调用总数是 82，而 Linux 0.11 内核的总数是 72
33.     ENOSYS = 38                                             //系统错误号
34.
35.     /* 36～38 行 .globl 定义入口点*/
36.     .globl _system_call,_timer_interrupt,_sys_execve
37.     .globl _hd_interrupt,_floppy_interrupt,_parallel_interrupt
38.     .globl _device_not_available, _coprocessor_error
39.     .align 2                                                //4 字节对齐
    /* bad_sys_call 错误系统调用处理程序*/
40.     bad_sys_call:
41.     pushl $-ENOSYS                                          //将错误号压栈
42.     jmp ret_from_sys_call                                   //跳转到 ret_from_sys_call
43.     .align 2                                                //4 字节对齐
44.     /* reschedule 重新调度处理程序*/
45.     reschedule:
46.     pushl $ret_from_sys_call
        //将 ret_from_sys_call 地址压栈，以便返回从 ret_from_sys_call 处执行
47.     jmp _schedule                                           //跳到_schedule
48.     .align 2                                                //4 字节对齐
    /*   _system_call 系统调用处理程序*/
49.     _system_call:
50.     cld                                                     //清除方向标志
    /*51～62 行由于在系统调用中，涉及到一些寄存器，需要保存相应寄存器中的值*/
51.     pushl %eax          # save orig_eax                     //将 eax 寄存器的值压栈，保存 eax 值
52.     push %gs                                                //将 gs 寄存器的值压栈
53.     push %fs                                                //将 fs 寄存器的值压栈
54.     push %es                                                //将 es 寄存器的值压栈
55.     push %ds                                                //将 ds 寄存器的值压栈
56.     pushl %eax          # save eax.  The return value will be put here.
57.     pushl %ebp                                              //将 ebp 寄存器的值压栈
58.     pushl %edi                                              //将 edi 寄存器的值压栈
59.     pushl %esi                                              //将 esi 寄存器的值压栈
60.     pushl %edx                                              //将 edx 寄存器的值压栈
61.     pushl %ecx          # push %ebx,%ecx,%edx as parameters
62.     pushl %ebx          # to the system call
63.     movl $0x10,%edx     # set up ds,es to kernel space
64.     mov %dx,%ds
65.     mov %dx,%es                     //设置 ds,es 指向内核空间（0x10 是内核数据段的段选择符）
66.     movl $0x17,%edx     # fs points to local data space//设置 fs 指向用户空间（0x17 是用户数据段描述符）
67.     mov %dx,%fs
68.     cmpl _NR_syscalls,%eax                                  //若调用号超出范围，
69.     jae bad_sys_call                                        //则跳转到 bad_sys_call
70.     call _sys_call_table(,%eax,4)                           //查表进行系统调用
71.     movl %eax,EAX(%esp)   # save the return value           //保存返回值
72.     2: movl _current,%eax                                   //将当前任务的指针输出到 eax 中
```

```
73.     cmpl $0,state(%eax)         # state          //若当前任务的状态不是就绪态（0），
74.     jne reschedule                             //则跳转到 reschedule，重新调度
75.     cmpl $0,counter(%eax)       # counter        //若任务的时间片用完，
76.     je reschedule                              //则跳转到 reschedule，重新调度
    /* ret_from_sys_call  系统调用返回处* /
77.     ret_from_sys_call:
78.     movl _current,%eax                         //将当前任务的指针输出到 eax 中
79.     cmpl _task,%eax             # task[0] cannot have signals //若当前任务是 task0，
80.     je 3f                                      //则跳转到 3（99 行）执行
81.     cmpw $0x0f,CS(%esp)         # was old code segment supervisor ?//若原调用程序不是内核任务
                                                   //（0x0f 是用户代码段描述符）
82.     jne 3f                                     //则跳转到 3（99 行）执行
83.     cmpw $0x17,OLDSS(%esp)          # was stack segment = 0x17 ? //若原堆栈不在用户数据段中
84.     jne 3f                                     //则跳转到 3 执行
85.     movl signal(%eax),%ebx                     //将当前任务的信号位图放入 ebx
86.     movl blocked(%eax),%ecx                    //将受阻塞的信号位图放入 ecx
87.     notl %ecx                                  //将 ecx 的值每位取反
88.     andl %ebx,%ecx                             //从当前任务的信号位图中清除受阻塞的信号集合
89.     bsfl %ecx,%ecx                             /选择受阻塞信号的最小值放入 ecx
90.     je 3f                                      //如果没有信号需要处理，结束中断处理
91.     btrl %ecx,%ebx                             //从信号位图中复位该信号
92.     movl %ebx,signal(%eax)                     //重置信号位图
93.     incl %ecx                                  //将要处理的信号+1
94.     pushl %ecx                                 //将要处理的信号压栈
95.     call _do_signal                            //调用信号处理函数（signal.c）
96.     popl %ecx                                  //返回值存入 ecx
97.     testl %eax, %eax                           //若 eax 中没有值
98.     jne 2b          # see if we need to switch tasks, or do more signals //则跳转到 2 处
    /*3 结束中断处理程序。将保存在栈中的寄存器值返回给相应的寄存器*/
99.     3:
100.    popl %ebx                                  //弹出相应的值到相应的寄存器
101.    popl %ecx
102.    popl %edx
103.    popl %esi
104.    popl %edi
105.    popl %ebp
106.    popl %eax
107.    pop %ds
108.    pop %es
109.    pop %fs
110.    pop %gs
111.    addl $4,%esp            # skip the orig_eax
112.    iret                                       //中断返回
113.    .align 2                                   //对齐方式
    /*_coprocessor_error 协处理器错误处理程序。*/
114.    _coprocessor_error:
115.    cld                                        //清除方向标志
    /*116-127 压栈操作，用于保存相应寄存器中的值*/
116.    pushl $-1               # mark this as an int.   //117-128 将寄存器压入栈
117.    push %gs
```

```
118.    push %fs
119.    push %es
120.    push %ds
121.    pushl %eax              # save eax.
122.    pushl %ebp
123.    pushl %edi
124.    pushl %esi
125.    pushl %edx
126.    pushl %ecx
127.    pushl %ebx
128.    movl $0x10,%eax                                  //es，ds 段指向内核数据段
129.    mov %ax,%ds
130.    mov %ax,%es
131.    movl $0x17,%eax                                  //fs 段指向用户数据段
132.    mov %ax,%fs
133.    pushl $ret_from_sys_call                         //将 ret_from_sys_call 地址压栈
134.    jmp _math_error                                  //执行 math_error
135.    .align 2
    /*_device_not_available 设备不存在处理程序*/
136.    _device_not_available:
137.    cld                                              //清除方向标志
    /*138～149 压栈操作，用于保存相应寄存器中的值*/
138.    pushl $-1               # mark this as an int    //139～149 将寄存器压入栈
139.    push %gs
140.    push %fs
141.    push %es
142.    push %ds
143.    pushl %eax
144.    pushl %ebp
145.    pushl %edi
146.    pushl %esi
147.    pushl %edx
148.    pushl %ecx
149.    pushl %ebx
150.    movl $0x10,%eax                                  //es，ds 段指向内核数据段
151.    mov %ax,%ds
152.    mov %ax,%es
153.    movl $0x17,%eax                                  //fs 段指向用户数据段
154.    mov %ax,%fs
155.    pushl $ret_from_sys_call                         //将 ret_from_sys_call 地址压栈
156.    clts                    # clear TS so that we can use math    //清除任务切换标志
157.    movl %cr0,%eax    // CR0 包含系统的控制标志，用于控制处理器的操作模式和状态
158.    testl $0x4,%eax   # EM (math emulation bit)  //如果不是 EM 引起的中断，则恢复新任
                                                     //务协处理器状态
159.    je _math_state_restore                           //则执行函数 math_state_restore()
160.    pushl $0        # temporary storage for ORIG_EIP
161.    call _math_emulate                               //如果是 EM 引起的中断，则软件模拟
162.    addl $4,%esp                                     //恢复堆栈
163.    ret                                              //跳转到 ret_from_sys_call
164.
```

```
165.    .align 2                                          //4 字节对齐方式
    /*_timer_interrupt 定时中断处理程序*/
166.    _timer_interrupt:
167.    cld                                               //清除方向标志
168.    pushl $-1           # mark this as an int         //168～179 保存相应寄存器中的值
169.    push %gs
170.    push %fs
171.    push %es
172.    push %ds
173.    pushl %eax
174.    pushl %ebp
175.    pushl %edi
176.    pushl %esi
177.    pushl %edx
178.    pushl %ecx
179.    pushl %ebx
180.    movl $0x10,%eax                                   //es，ds 段指向内核数据段
181.    mov %ax,%ds
182.    mov %ax,%es
183.    movl $0x17,%eax                                   //fs 段指向用户数据段
184.    mov %ax,%fs
185.    incl _jiffies                                     //滴答数加 1
    /*因初始化中断控制芯片时没有采用自动 EOI，所以需要发指令结束该硬件中断*/
186.    movb $0x20,%al    # EOI to interrupt controller #1
187.      outb %al,$0x20                                  //发 EOI 到主 8295A（从 8295A 是 0xA0）
    /*188～190 是从选择符中取当前特权级别(0 或 3)并压入堆栈，作为 do_timer 的参数*/
188.    movl CS(%esp),%eax
189.    andl $3,%eax # %eax is CPL (0 or 3, 0=supervisor)
190.    pushl %eax
191.    call _do_timer# 'do_timer(long CPL)' does everything from //执行 do_timer 函数
192.    addl $4,%esp # task switching to accounting ...
193.    jmp ret_from_sys_call                             //执行 ret_from_sys_call
194.
195.    .align 2
196.    /*_sys_execve 系统调用 */
197.    _sys_execve:
198.    lea (EIP+4)(%esp),%eax    # don't forget about the return address.
                                                          //取中断调用前的指令指针寄存器的值，存入 eax
199.    pushl %eax                                        //该地址作为_do_execve 函数的参数
200.    call _do_execve                                   //执行函数 do_execve
201.    addl $4,%esp                                      //恢复堆栈
202.    ret
    /*_hd_interrupt 硬盘中断处理 */
203.    _hd_interrupt:
204.    cld                                               //清除方向标志
205.    pushl %eax                                        //205～210 保存相应寄存器中的值
206.    pushl %ecx
207.    pushl %edx
208.    push %ds
209.    push %es
```

```
210.    push %fs
211.    movl $0x10,%eax                          //es，ds 段指向内核数据段
212.    mov %ax,%ds
213.    mov %ax,%es
214.    movl $0x17,%eax                          //fs 段指向用户数据段
215.    mov %ax,%fs
    /*因初始化中断控制芯片时没有采用自动 EOI，所以需要发指令结束该硬件中断*/
216.    movb $0x20,%al
217.    outb %al,$0xA0     # EOI to interrupt controller #1      //发 EOI 到从 8295A（0xA0）
218.    jmp 1f             # give port chance to breathe         //两个入口 1， 起到延时作用
219.    1: jmp 1f
220.    1: outb %al,$0x20                        //发数据到主 8295A
221.    andl $0xfffeffff,_timer_active
222.    xorl %edx,%edx                           //置 edx 为空
223.    xchgl _do_hd,%edx                        //置_do_hd 为空
224.    testl %edx,%edx                          //测试函数指针是否为 Null
225.    jne 1f                                   //不为空则跳转
226.    movl $_unexpected_hd_interrupt,%edx      //若空，则使指针指向 C 函数
227.    #unexpected_hd_interrupt()。
228.    1: call *%edx # "interesting" way of handling intr.   //处理中断的有趣反方式
229.    pop %fs                                  //229～234 恢复寄存器内容
230.    pop %es
231.    pop %ds
232.    popl %edx
233.    popl %ecx
234.    popl %eax
235.    iret                                     //执行中断返回
    /_floppy_interrupt: 软盘中断调用 */
236.    _floppy_interrupt:
237.    cld                                      //清除方向标志
238.    pushl %eax                               //238～263 保存相应寄存器中的值
239.    pushl %ecx
240.    pushl %edx
241.    push %ds
242.    push %es
243.    push %fs
244.    movl $0x10,%eax                          //es，ds 段指向内核数据段
245.    mov %ax,%ds
246.    mov %ax,%es
247.    movl $0x17,%eax                          //fs 段指向用户数据段
248.    mov %ax,%fs
249.    movb $0x20,%al
250.    outb %al,$0x20          # EOI to interrupt controller #1     #发 EOI 到主 8295A
251.    xorl %eax,%eax                           //置 eax 为空
252.    xchgl _do_floppy,%eax                    //置_do_floppy 为空
253.    testl %eax,%eax                          //测试函数指针是否=NULL
254.    jne 1f                                   //不为空则跳转
255.    movl $_unexpected_floppy_interrupt,%eax  //若空，则使指针指向 C 函数
256.    #unexpected_hd_interrupt()。
257.    305    1: call *%eax # "interesting" way of handling intr.
```

```
258.    pop %fs                                  //258～263 恢复寄存器内容
259.    pop %es
260.    pop %ds
261.    popl %edx
262.    popl %ecx
263.    popl %eax
264.    iret                                     //中断返回
265.    /_parallel_interrupt 并行口中断处理*/
266.    _parallel_interrupt:
267.    cld                                      //清除方向标志
268.    pushl %eax                               //将 eax 压栈
269.    movb $0x20,%al
270.    outb %al,$0x20                           //发 EOI 到主 8295A
271.    popl %eax                                //恢复寄存器内容
272.       iret                                  //中断返回
```

4.9 asm.s 文件

asm.s 是一个位于 kernel 目录下的汇编文件。asm.s 实现了大部分的硬件故障（或出错）处理的底层代码：其中包括除零中断处理、错误时的中断处理以及调试中断，所以在内核被引导和启动加载时经常被调用。

1. 功能描述

asm.s 是位于 kernel 目录下的汇编文件。它主要实现了包括大部分的硬件故障（或出错）处理的底层次代码，其中有除零中断处理、错误时的中断处理、调试中断、不可屏蔽中断、断点中断、溢出中断、越界、无效操作符、协处理器段溢出、保留中断、协处理器中断、双出错在、无效 TSS、堆栈段中断、保护中断、对齐检查、页故障中断。页异常是由内存管理程序 mm 处理的，所以 asm.s 程序中没有实现页异常的中断处理。此程序还处理由于 TS 位而造成的异常，因为这种必须正确地进行保存/恢复处理。

该程序中的中断函数的调用分成 2 类：一类是带返回错误码的中断调用，对应的符号为 error_code；另一类是不带返回错误码的中断调用，对应的符号为 no_error_code。

2. 代码分析

asm.s 文件的详细代码注释如下：

```
        /*1-6 程序入口点。*/
1.          .globl _divide_error,_debug,_nmi,_int3,_overflow,_bounds,_invalid_op
2.          .globl _double_fault,_coprocessor_segment_overrun
3.          .globl _invalid_TSS,_segment_not_present,_stack_segment
4.          .globl _general_protection,_coprocessor_error,_irq13,_reserved
5.          .globl _alignment_check
6.          .globl _page_fault
7.
         /* divide_error   除零中断处理*/
8.          _divide_error:
9.          pushl $0                                  //不带返回错误码的中断调用
10.         pushl $_do_divide_error                   // do_divide_error 函数地址压栈
        /* error_code 错误时的处理程序*/
```

```
11.     error_code:
12.     push %fs                                        // 12-21 保存寄存器
13.     push %es
14.     push %ds
15.     pushl %eax
16.     pushl %ebp
17.     pushl %edi
18.     pushl %esi
19.     pushl %edx
20.     pushl %ecx
21.     pushl %ebx
22.     cld                                             //清除方向标志
23.     movl $-1, %eax
24.     xchgl %eax, 0x2c(%esp)  # orig_eax (get the error code. )   //eax 取得错误代码
25.     xorl %ebx,%ebx                                  //将 ebx 清零
26.     mov %gs,%bx                                     //获取 gs 低位
27.     xchgl %ebx, 0x28(%esp)                          //存储 gs
28.     pushl %eax                                      //错误号压栈
29.     lea 52(%esp),%edx
30.     pushl %edx
31.     movl $0x10,%edx                                 //设置 ds,es 指向内核空间
32.     mov %dx,%ds
33.     mov %dx,%es
34.     mov %dx,%fs
35.     call *%ebx                                      //调用相应的 C 函数，其参数已入栈
36.     addl $8,%esp                                    //esp 重新指向寄存器 fs 入栈处
37.     popl %ebx                                       //恢复各个寄存器
38.     popl %ecx
39.     popl %edx
40.     popl %esi
41.     popl %edi
42.     popl %ebp
43.     popl %eax
44.     pop %ds
45.     pop %es
46.     pop %fs
47.     pop %gs
48.     addl $4,%esp
49.     iret                                            //中断返回
      /*_debug 调试中断处理*/
50.     _debug:
51.     pushl $0
52.     pushl $_do_int3          # _do_debug            //do_int3 函数指针压栈
53.     jmp error_code                                  //跳转到_error_code
    /*_nmi 不可屏蔽中断处理*/
54.     _nmi:
55.     pushl $0
56.     pushl $_do_nmi                                  //do_nmi 函数指针压栈
57.     jmp error_code                                  //跳转到_error_code
     /*_ int3 断点中断处理*/
```

```
58.       _int3:
59.       pushl $0
60.       pushl $_do_int3                                    //do_int3 函数指针压栈
61.       jmp error_code                                     //跳转到_error_code
      /*_overflow 溢出中断处理*/
62.       _overflow:
63.       pushl $0
64.       pushl $_do_overflow                                //do_overflow 函数指针压栈
65.       jmp error_code                                     //跳转到_error_code
      /*_bounds 越界中断处理*/
66.       _bounds:
67.       pushl $0
68.       pushl $_do_bounds                                  //do_bounds 函数指针压栈
69.       jmp error_code                                     //跳转到_error_code
      /*_invalid_op 无效操作符中断处理*/
70.       _invalid_op:
71.       pushl $0
72.       pushl $_do_invalid_op                              //do_invalid_op 函数指针压栈
73.       jmp error_code                                     //跳转到_error_code
      /*_c oprocessor_segment_overrun 协处理器段溢出中断处理*/
74.       _coprocessor_segment_overrun:
75.       pushl $0
76.       pushl $_do_coprocessor_segment_overrun             //函数指针压栈
77.       jmp error_code                                     //跳转到_error_code
      /*_reserved 保留中断处理*/
78.       _reserved:
79.       pushl $0
80.       pushl $_do_reserved                                //do_reserved 函数指针压栈
81.       jmp error_code                                     //跳转到_error_code
      /*_irq13 协处理器中断处理*/
82.       _irq13:
83.       pushl %eax                                         //保存 eax
84.       xorb %al,%al                                       //al 清零
85.       outb %al,$0xF0     //为了确保本中断被响应。通过写 0xF0 端口，消除 CPU 的 BUSY 延续信号
86.       movb $0x20,%al
87.       outb %al,$0x20                                     //向主 8259 发送 EOI 信号
88.       jmp 1f
89.       1: jmp 1f
90.       1: outb %al,$0xA0                                  //向从 8259 发送 EOI 信号
91.       popl %eax                                          //恢复 eax 的值
92.       jmp _coprocessor_error                             //跳转到_coprocessor_error(syste,_ call.s)
      /*_double_faul 双出错中断处理*/
93.       _double_fault:
94.       pushl $_do_double_fault                            //do_double_fault 函数指针压栈
95.       jmp error_code                                     //跳转到_error_code
      /*_invalid_TSS 无效 TSS 中断处理*/
96.       _invalid_TSS:
97.       pushl $_do_invalid_TSS                             // do_invalid_TSS 函数指针压栈
98.       jmp error_code                                     //跳转到_error_code
      /*_invalid_TSS 段不存在中断处理*/
```

```
99.       _segment_not_present:
100.      pushl $_do_segment_not_present                    //函数指针压栈
101.      jmp error_code                                    //跳转到_error_code
        /*_stack_segmen 堆栈段中断处理*/
102.      _stack_segment:
103.      pushl $_do_stack_segment                          // do_stack_segment 函数指针压栈
104.      jmp error_code                                    //跳转到_error_code
        /*_general_protection 一般保护性中断处理*/
105.      _general_protection:
106.      pushl $_do_general_protection                     // do_general_protection 函数指针压栈
107.      jmp error_code                                    //跳转到_error_code
        /*_alignment_check 对齐检查中断处理*/
108.      _alignment_check:
109.      pushl $_do_alignment_check                        // do_alignment_check 函数指针压栈
110.      jmp error_code                                    //跳转到_error_code
        /*_page_fault 页故障中断处理*/
111.      _page_fault:
112.      pushl $_do_page_fault                             // do_page_fault 函数指针压栈
113.      jmp error_code                                    //跳转到_error_code
```

4.10 ptrace.c 文件

ptrace.c 文件位于 kernel 目录下，主要提供了进程跟踪系统调用 sys_ptrace（在 linux0.95 系统中，其对应的系统调用号为 26）。值得注意的是从 linux0.95 以后才有的 ptrace.c 程序文件，linux0.12 内核中没有这个文件。在该文件中，涉及下面几个宏：

```
EPERM          //操作不被允许
ESRCH          //不存在这样的进程
EIO            //I/O 错误。
```

1. 功能描述

在多任务的嵌入式系统中，常常同时运行着相当多的进程。这些进程之间频繁的进行着大量的通信动作。进程的运行状态与这些不断发生的通信有着直接和紧密的联系。通过对进程间通信的监视，开发人员可以掌控系统内部运转的状态。发现错误时，利用获取到的进程间通信的信息，调试工程师更容易发现问题之所在。而 Linux 中的 ptrace 系统提供了监视进程运行信息以及进程间通信情况的解决方案。可通过 ptrace 观察和控制其他进程、截获正在运行进程的所有的系统调用。例如，通过 ptrace 一旦接管了被监视进程的 sendto 和 recvfrom 系统调用，将使我们能够截获到使用这两个系统调用进行通信的数据。

ptrace 提供了进程的跟踪功能。例如父进程可对子进程进行跟踪，使得父进程可以对自己的子进程进行监督与控制。具体包括读写子进程用户空间的程序、数据或 user 结构中的变量，向它们发送软中断，以及命令它们自我终止等。系统提供了两种系统调用 waitpid()和 ptrace()来实现跟踪功能。这种父子进程间的关联可由子进程发 ptrace(0)请求，也可由父进程发请求来实现。

2. 代码分析

ptrace.c 文件的详细代码注释如下：

```
1.      #include <sys/ptrace.h>
2.      /* 1 = access 0 = no access */
      /*当子进程死掉时，在 exit.c or in signal.c 还未捕获发送的信号。决定用户可以操作的方式，1 代表
```

```
        允许，0 代表不允许*/
3.      #define FLAG_MASK 0x00000dd9                          //定义跟踪操作掩码
4.
5.      /* set's the trap flag. */
6.      #define TRAP_FLAG 0x100                               //设置陷阱标志
7.
8.      /* check's for granularity. */
9.      #define GRANULARITY 0x00800000                        //检查粒度
10.
11.     #define MAGICNUMBER 68                  //这个数字是减去栈顶的值，目的是找到本地的框架
12.
        /*do_no_page 为内存缺页中断处理函数*/
13.     void do_no_page (unsigned long, unsigned long, struct task_struct *);
        /* write_verify 为内存页面验证函数，用来判断是否有页面共享的情况存在，如果有，就进行页面的写时复制操作*/
14.     void write_verify (unsigned long);
15.
16.     /* get_task 为内联函数，作用是根据进程标识获取该进程任务数据结构的索引位置*/
17.     static inline int get_task(int pid)
18.     {
19.     int i;
20.     for (i =0; i < NR_TASKS; i++)                         //遍历进程任务数据结构列表
21.     {
22.     if (task[i] != NULL && (task[i]->pid == pid)) return (i);     //若找到进程的任务数据结构
                                                              //则返回进程在任务数组中的位置
23.     }
24.     return (-1);                                          //否则（例如进程号非法）则返回-1
                                                              //表示没有该进程号对应的进程
25.     }
        /* get_stack_long 获取参数 task 指定的进程偏移量，为 offset 的栈数据，这个函数用来从一个进程的特权栈取数据，其中 offset 表示离基地址的偏移量，函数假设特权栈在我们的数据空间里*/
26.     static inline int
27.     get_stack_long(struct task_struct *task, int offset)
28.     {
29.     unsigned char *stack;
30.     stack = (unsigned char *)task->tss.esp0;              //取栈顶位置
31.     stack += offset;                                      //指向栈中偏移量为 offset 的位置
32.     return (*((int *)stack));                             //返回栈数据
33.     }
        /* put_stack_long 放数据 data 到参数 task 指定的进程偏移量为 offset 的栈中，这个函数用来往一个进程的特权栈放数据，其中 offset 表示离基地址偏移量，函数假设特权栈在我们的数据空间里*/
34.     static inline int
35.     put_stack_long(struct task_struct *task, int offset, unsigned short data)
36.     {
37.     unsigned char *stack;
38.     stack = (unsigned char *)task->tss.esp0;              //取栈顶位置
39.     stack += offset;                                      //指向栈中偏移量为 offset 的位置
40.     *(int *)stack = data;                                 //对该栈对应的位置赋值
41.     return (0);
42.     }
```

```
    /* get_long 用来从任意一个任务的特权栈取数据，它使用任务索引而不是任务指针来定位任务*/
43.     static inline int get_long (int tsk, long addr, unsigned seg, int *data)
44.     {
45.     int i;
46.     int limit;
47.     int cur;
48.     unsigned long address;
49.     unsigned long page;
50.     unsigned oldfs;
51.                                                     //查找当前任务的任务索引*/
52.     for (i = 0; i < NR_TASKS ; i ++)
53.     {
54.     if (task[i] == current) break;                  //遍历任务数组，若找到则退出循环
55.     }
56.     if (i == NR_TASKS)                              //若不存在 current 指定的任务
57.     {
58.     panic ("PTRACE: Can't find current task\n");    //则停机
59.     }
60.     cur = i;                                        //cur 记录对应的任务索引
61.     seg++;                                          //为了避免段访问失败，需要检查段的可读
                                                          性和这个字节的数据
62.     limit=(task[tsk]->ldt[seg].a) & 0xffff;
63.     if (task[tsk]->ldt[seg].b & GRANULARITY) limit = limit << 12;
                                                        //在所有的段里，这应该是个常量。但是无
                                                        //论如何最好检查
64.     if (limit <= addr+4) return (-EIO);
65.     address = ((task[tsk]->ldt[seg].a & 0xffff000) >> 8) |
66.     ((task[tsk]->ldt[seg].b & 0xff) << 16 ) |
67.     (task[tsk]->ldt[seg].b & 0xff000000);           //计算地址，确保有这个地址*/
68.     page = *((unsigned long*) ((address >> 20) & 0xffc));
69.     if (! (page & PAGE_PRESENT))                    //确定是不是当前页，若不是，则缺页处理
70.     {
71.     do_no_page (0, address, task[tsk]);
72.     }
    /* get_fs()防止用户空间程序蓄意破坏内核空间；系统调用默认是提供给用户空间的程序访
    问的，它会认为来自用户空间，为了保护内核空间，一般会用 get_fs()得到的值来和
    USER_DS 进行比较*/
73.     oldfs=get_fs();
74.     /* now convert seg to the right format. */
75.     seg = seg << 3 | 0x4;
76.     cli();                                          //关中断
77.     lldt(tsk);                                      //若将改变局部描述符表，应先关中断
    /*set_fs(KERNEL_DS);将其能访问的空间限制扩大到 KERNEL_DS，这样就可以在内核
    顺利使用系统调用了*/
78.     set_fs(seg);
79.     *data = get_fs_long((void *)addr); /* we are assuming kernel space
80.     is in the gdt here. */
81.     lldt(cur);                                      //加载局部描述符
82.     set_fs(oldfs);                                  //设置 fs
83.     sti();                                          //开中断
```

```
84.     return (0);
85.     }
        /* put_long 函数用来往任意一个任务的特权栈放数据。它使用任务索引而不是任务指针来定位任务*/
86.     static inline int put_long (int tsk, long addr, int data, unsigned seg)
87.     {
88.     int i;
89.     int limit;
90.     unsigned oldfs;
91.     unsigned long address;
92.     unsigned long page;
93.     int cur;
94.                                                     //查找当前任务的任务索引
95.     for (i = 0; i < NR_TASKS ; i ++)                //遍历任务数组，若找到则退出循环
96.     {
97.     if (task[i] == current) break;
98.     }
99.     if (i == NR_TASKS)
100.    {
101.    panic ("PTRACE: Can't find current task\n");    //若不存在 current 指定的任务，则停机
102.    }
103.    cur = i;                                        //cur 记录对应的任务索引
104.    seg++;                                          //为了避免段访问失败，需要检查段的可读
                                                        //性和这个字节的数据
105.    limit=(task[tsk]->ldt[seg].a) & 0xffff;
106.    if (task[tsk]->ldt[seg].b & GRANULARITY) limit = limit << 12;
                                                        //在所有的段里，这应该是个常量。但是无论如何最好检查
107.    if (limit <= addr+4) return (-EIO);
108.
109.    address = ((task[tsk]->ldt[seg].a & 0xffff000) >> 8)
110.    ((task[tsk]->ldt[seg].b & 0xff) << 16 )
111.    (task[tsk]->ldt[seg].b & 0xff000000);
112.
113.    page = *((unsigned long*) ((address >> 20) & 0xffc));
114.    if (! (page & PAGE_PRESENT))                    //确定是不是当前页。若不是当前页，则缺页处理
115.    {
116.    do_no_page (0, address, task[tsk]);
117.    }
118.    write_verify (address);                         //内存页面验证来判断页面共享的情况
       /* get_fs()防止用户空间程序蓄意破坏内核空间；系统调用默认是提供给用户空间的程序访问的，它
       会认为来自用户空间，为了保护内核空间，一般会用 get_fs()得到的值来和 USER_DS 进行比较*/
119.    oldfs=get_fs();                                 //取 fs 的值
120.
121.    seg = seg << 3 | 0x4;                           //做移位运算计算 seg 的值
122.
123.    cli();                                          //若将改变局部描述符表，应先关中断
124.    lldt(tsk);                                      //加载局部描述符
125.    set_fs(seg);
126.    put_fs_long(data,(void *)addr);
127.    lldt(cur);                                      //加载局部描述符
128.    set_fs(oldfs);
```

```
129.    sti();                                          //开中断
130.    return (0);
131.    }
        /* sys_ptrace()进程跟踪系统调用。ptrace()中涉及下面几个宏：
        EPERM           操作不被允许
        ESRCH           不存在这样的进程
        EIO             I/O 错误   */
132.    int sys_ptrace( unsigned long *buffer)
133.
134.    {
135.    long request, pid, data;
136.    long addr;
137.    struct task_struct *child;
138.    int childno;
139.    request = get_fs_long(buffer++);                //取 fs 段中指定处的长字，作为 request
140.    pid = get_fs_long(buffer++);                    //取进程号
141.    addr = get_fs_long(buffer++);                   //取地址
142.    data = get_fs_long(buffer++);                   //取数据
143.    if (request == 0)                               //若请求父进程跟踪
144.    {
145.
146.    current->flags |= PF_PTRACED;                   //则设置该进程的跟踪标志
147.    return (0);                                     //返回
148.    }
149.    childno=get_task(pid);                          //获取进程 pid 对应在任务数组中的位置
150.    if (childno < 0)                                //若不存在这样的进程
151.    return (-ESRCH);                                //返回相应的标志
152.    else                                            //否则获取相应的进程数据结构
153.    child = task[childno];
154.    if (child->p_pptr != current ||
155.    !(child->flags & PF_PTRACED) || child->state != TASK_STOPPED)
                                                        //若当前进程不是被跟踪进程（child）的父进
                                                        //程或 child 允许跟踪或 child 没有被暂停
156.    return (-ESRCH);                                //返回相应的错误标志
157.    switch (request)                                //处理其他的请求
158.    {
159.
160.    case 1:                                         //request=1 或者 2,则在指定的位置读一个字
161.    case 2: {
162.    int tmp;                                        //定义临时变量
163.    int res;
164.    res = get_long(childno, addr, 1, &tmp);         //获取数据
165.    if (res < 0)
166.    return res;
167.    verify_area(data, 4);                           //验证给定地址开始的内存块是否超限
168.    put_fs_long( tmp, (unsigned long *)data);       //将数据 tmp 数据放到 data 中
169.    return 0;
170.    }
171.    case 3:                                         // request=3 在 USER 指定的位置读一个字
```

```
172.    {
173.    int tmp;
174.    addr = addr >> 2; /* temporary hack. */
175.    if (addr < 0 || addr >= 17)                          //若 addr < 0 或者 addr >= 17
176.    return (-EIO);                                       //则返回 EIO 标志
177.    verify_area(data, 4);                                //验证给定地址开始的内存块是否超限
178.    tmp = get_stack_long (child, 4*addr-MAGICNUMBER);    //从栈中读取长字
179.    put_fs_long(tmp,(unsigned long *)data);              //放到 fs 指定的地址中
180.    return (0);
181.    }
182.    case 4:                                              // request=4 或者 5，在指定的位置写一个字
183.    case 5:
184.
185.    if (put_long(childno, addr, data, 1)) return (-EIO);//若失败。返回 EIO 标志
186.    return (0);
187.    case 6:
                                                             // request=6，在 USER 指定的位置写一个字
188.    addr = addr >> 2;
189.    if (addr < 0 || addr >= 17) return (-EIO);
190.    if (addr == ORIG_EAX) return (-EIO);
191.    if (addr == EFL)
192.    {
193.    data &= FLAG_MASK;
194.    data |= get_stack_long(child, EFL*4-MAGICNUMBER)   & ~FLAG_MASK;
195.    }
196.
197.    if (put_stack_long(child, 4*addr-MAGICNUMBER, data)) return (-EIO);
198.    return (0);
199.    case 7:                                              // request=7 子进程接受信号后，重启
200.    {
201.    long tmp;
202.    child->signal=0;                                     //子进程信号清零
203.    if (data > 0 && data <= NSIG)                        //若 data 大于 0 或者比 NSIG 小
204.    child->signal = 1<<(data-1);                         //则根据 data 设置信号
205.    child->state = 0;                                    //子进程状态清零
206.                                                         //确保单步位没有被置位
207.    tmp = get_stack_long (child, 4*EFL-MAGICNUMBER) & ~TRAP_FLAG;  //从栈中取数据
208.    put_stack_long(child, 4*EFL-MAGICNUMBER,tmp);
209.    return (0);
210.    }
211.    case 8:                                              // request=8，终止子进程
212.
213.
214.    {
215.    long tmp;
216.    child->state = 0;                                    //子进程信号清零
217.    child->signal = 1 << (SIGKILL -1 );                  //设置子进程信号
218.
```

```
219.    tmp = get_stack_long (child, 4*EFL-MAGICNUMBER) & ~TRAP_FLAG;//从堆栈中取一个长字数据
220.    put_stack_long(child, 4*EFL-MAGICNUMBER,tmp);//放入一个长字数据
221.    return (0);
222.    }
223.    case 9:                                          // request=9，置 TRAP 标志
224.    {
225.    long tmp;
226.    tmp = get_stack_long (child, 4*EFL-MAGICNUMBER) | TRAP_FLAG;
227.    put_stack_long(child, 4*EFL-MAGICNUMBER,tmp);//从栈中取数据
228.    child->state = 0;                                //修改子进程的状态
229.    child->signal=0;                                 //信号清零
230.    if (data > 0 && data <NSIG)                      //若 data 大于 0 或者比 NSIG 小
231.    child->signal= 1<<(data-1);                      //则设置信号
232.
233.    return (0);
234.    }
235.    default:                                         // request 为其他值时，返回错误标志
236.    return (-EIO);
237.    }
238.    }
```

第5章 进 程 控 制

在 Linux 0.95 版内核中，与进程控制相关的文件主要位于 linux/kernel/目录下，此目录下共有 13 个代码文件，一个 Makefile 文件和三个文件夹子目录，三个子目录文件分别为块设备驱动程序子目录 blk_drv、字符设备驱动程序子目录 chr_drv、协处理器操作程序子目录 math。本章主要对 kernel/目录下 13 个代码文件中与进程有关的 7 个文件进行分析，表 5-1 为本章中将要介绍的文件列表。

表 5-1 linux/kernel 目录(与进程相关部分)

名　　称	大　小（B）	修 改 时 间
fork.c	4255	1992-3-4 20:11:24
sched.c	9773	1992-3-4 20:07:42
signal.c	5805	1992-3-1 23:27:36
exit.c	10981	1992-2-21 7:20:12
vsprintf.c	4800	1991-10-2 22:16:30
printk.c	537	1992-1-11 7:14:00
panic.c	448	1991-10-17 22:22:02

5.1 总体概述

程序是一个静态的可执行文件，是为实现特定目标或解决特定问题而用计算机语言编写的命令序列的集合。而进程（Process）是最初定义在 UNIX 等多用户、多任务操作系统环境下用于表示应用程序在内存环境中基本执行单元的概念。程序是静态的，而进程是动态的。

在 Linux 内核中，系统最多可有 64 个进程同时存在。一般来说，Linux 下的进程包含以下几个关键要素：一段可执行程序；专用的系统堆栈空间；内核中它的控制块（即进程控制块，描述进程所占用的资源，这样进程才能接受内核的调度）；独立的存储空间。

系统的第一个进程由系统初始化生成，创建者被称为父进程，被创建者被称为子进程，其他进程都使用 fork()系统调用生成。内核通过一个唯一的进程标识号 PID 来标识每个进程。在任意时刻，CPU 只能运行一个进程。Linux 内核用数据结构 task_struct 表示一个进程，包括进程的属性和执行状态等。task_struct 结构中的指向前后进程的指针形成了 task 数组。当新进程生成时，Linux 为新进程分配一个 task_struct 结构，然后将指针保存在 task 数组中。数组大小由 NR_TASKS 保存。进程的调度通过调度函数实现。在内核要切换到另一个进程时，需要保存当前进程的“上下文”，即任务数据结构中的数据。

在 Linux 中，进程的生存状态包括五种进程状态（进程状态被保存在任务数据结构的 state 字段中），即 TASK_RUNNING、TASK_INTERRUPTIBLE、TASK_UNINTERRUPTIBLE、TASK_STOPPED、TASK_ZOMBIE 五种状态。

（1）TASK_RUNNING：运行状态。

此状态表示进程正处于执行状态，或者该进程正在就绪队列中等待执行。即在 Linux 中，就绪态也包括在运行态中，并且进程可处于内核态运行状态，也可处于用户态运行状态。

（2）TASK_INTERRUPTIBLE：可中断睡眠状态。

此状态表示进程由于等待某个特定事件的发生而处于睡眠状态。此状态的进程可以是特定的事件发生而将其唤醒，也可以因为接受到信号而被唤醒，然后进程将转到运行状态。

（3）TASK_UNINTERRUPTIBLE：不可中断睡眠状态。

此状态的进程同可中断睡眠状态类似，不同之处在于，它不可以因为接受到信号而被唤醒转到运行状态，它只有被 wake_up()函数明确唤醒时才能转到运行状态。

（4）TASK_STOPPED：停止状态。

此状态中的进程既没有处于运行状态，也不可以将其投入运行。当进程收到信号 SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU 时就会进入此状态，可以通过向他发送 SIGCONT 信号让进程转换到运行状态。

（5）TASK_ZOMBIE：僵死状态。

此状态中的进程已经结束了运行，但是它的父进程还未询问其状态，为了让它的父进程能够获得它结束的消息，该进程的描述符会一直保留，直到父进程调用系统调用获得其进程状态为止。

这几种状态之间的转换关系如图 5-1 所示。

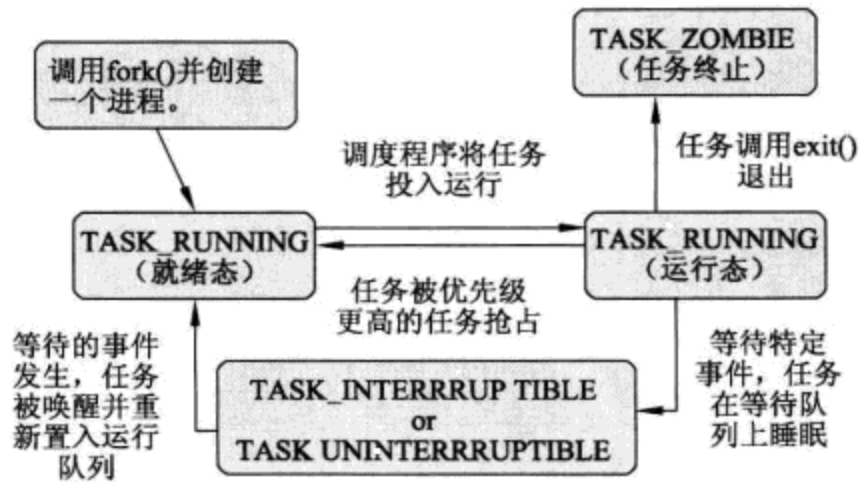


图 5-1　Linux 进程状态转换图

在创建新进程时，系统首先在任务数组中找出一个还没有被任何进程使用的空槽。如果系统已经有最大个数的进程在运行，则 fork()系统调用会返回-1 来表示因为任务数组中没有可用空槽而出错，否则父进程 fork 返回子进程的 ID，而对于 fork()子进程返回 0。我们通过这个返回值来区分父子进程。同时在创建过程中应该将新进程的状态置为不可中断睡眠状态，主要是为了防止未处理完成的新进程被调度函数调度。

Linux 是一个抢占式的多任务操作系统，而 Linux 的调度基于分时技术。所谓的分时是指 CPU 的时间被分成“片”，给每个可运行进程分配一片。当分配给正在执行进程的时间片结束的时候，调度程度就会执行，重新进行进程调度。时间片的分配，是根据进程的优先级决定的。在 Linux 中，进程的优先级是动态的，调度程序会动态地调整它们的优先级。

Linux 里面的进程可以被分为交互式进程、批处理进程和实时进程三类，初期的 Linux 内核的调度方式只有一种，即分时调度策略。之后，随着内核版本的提高，针对实时进程有 SCHED_FIFO 和 SCHED_RR 两种调度策略，针对普通进程有 SCHED_OTHER 策略。在进程调度时，schedule()函数首先扫描任务数组，通过比较运行态任务的运行时间递减滴答记数 counter 值来选中当前那个进程运行的时间最少的进程，并使用任务切换函数切换到该进程运行。进程切换由宏 switch_to()完成。在切换之前，要完成的任务为：检查要切换到的进程是否为当前进程，若是，则直接退出，若不是，则把当前进程指针 current 置为新任务的指针，然后跳转到新任务

的任务状态段组成的地址处，造成 CPU 执行任务切换。对于进程中止时，内核需要通过函数 exit() 释放该进程所占用的系统资源。

一个大型的系统，需要众多进程协作，进程间通信的重要性就显而易见，进程间通信就是信息在不同进程之间传播或交换。Linux 环境下的进程间通信手段，主要包括管道、系统 IPC（包括消息队列、信号量、共享存储）、SOCKET。而管道包括普通管道、流管道、命名管道三种。但在 0.95 版内核中，还未完全实现这些手段。其中，值得一提的是信号通信方式，信号是用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身。Linux 除了支持 UNIX 早期信号语义函数 sigal 外，还支持语义符合 Posix.1 标准的信号函数 sigaction。

在本章中，主要介绍与进程创建相关的 fork.c 程序，与进程调度相关的 sched.c 程序，与信号处理相关的 signal.c 程序等。

5.2 重要数据结构

本节主要内容为介绍与进程相关部分的数据结构，对各个数据结构的代码进行分析。本节介绍的数据结构列表如表 5-2 所示：

表 5-2 进程相关数据结构列表

名 称	功 能 简 单 描 述
task_struct	进程描述符
tss_struct	任务状态段数据结构
struct i387_struct	数学协处理器使用的结构
task_union	一个任务联合体
sigaction	信号处理相关的结构体
timer_list	定时器链表结构

5.2.1 task_struct

为了对操作系统中的进程进行管理，操作系统必须对每个进程在系统中所做的事情进行清楚地描述，为此，操作系统使用数据结构 task_struct 来代表处理不同的进程。这是对进程控制的最有效的手段。

```
1.  struct task_struct {                                    // 进程描述符，位于 linux/include/linux/sched.h 中
2.  long state;  /* -1 unrunnable, 0 runnable, >0 stopped */// 任务的运行状态
3.  long counter;                                           // 任务运行时间片
4.  long priority;                                          // 任务优先数
5.  long signal;                                            // 信号位图
6.  struct sigaction sigaction[32];                         // 包括信号将要执行的操作和标志
7.  long blocked;                                           // 进程信号屏蔽码
8.  int exit_code;                                          // 任务执行停止的退出码，由父进程取
9.  unsigned long start_code,end_code,end_data,brk,start_stack;
10. long pid,pgrp,session,leader;                           // 定义了进程号，进程组号，会话号，会话首领
11. int    groups[NGROUPS];                                 // 定义用户数组，最大数为 32
12. struct task_struct *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
// 分别定义了指向父进程，最新子进程，比自己后创建的相邻进程，比自己早创建的相邻进程的指针
13. unsigned short uid,euid,suid;                           // 定义了用户 ID，有效用户 ID，保存的用户 ID
14. unsigned short gid,egid,sgid;                           // 定义了组 ID，有效组 ID，保存的组 ID
15. unsigned long timeout,alarm;                            // 定义了内核定时超时值以及报警定时器
```

```
16.   long utime,stime,cutime,cstime,start_time;
// 定义了用户态运行时间，系统运行时间，子进程用户态运行时间，
// 子进程系统态运行时间以及进程开始运行时刻
17.   struct rlimit rlim[RLIM_NLIMITS];        // 定义了进程使用统计数组
18.   unsigned int flags;                       // 定义了每个进程的标志
19.   unsigned short used_math;                 // 标志是否使用了协处理器
20.   int link_count;                           // 定义了链接数
21.   int tty;                                  // 定义了进程使用 tty 的子设备号
22.   unsigned short umask;                     // 进程建新文件 16 位属性屏蔽字
23.   struct inode * pwd;                       // 当前工作目录 i 结点结构
24.   struct inode * root;                      // 根目录 i 结点结构
25.   struct inode * executable;                // 执行文件 i 结点结构
26.   struct inode * library;                   // 库文件 i 结点结构
27.   unsigned long close_on_exec;              // 关闭文件句柄位图标志
28.   struct file * filp[NR_OPEN];              // 进程使用的文件表结构
29.   struct desc_struct ldt[3];                // 本进程的局部表描述符
30.   struct tss_struct tss;                    // 本进程的任务状态结构
31.   };
```

5.2.2 tss_struct

任务状态段用于恢复任务执行而所要保存的处理器状态信息。在任务状态段中包含了两大类：动态字段和静态字段。动态字段即当任务切换而被挂起时，处理器会更新动态字段的内容。


```
1.    struct tss_struct {                   // 任务状态段数据结构，位于 linux/include/linux/sched.h 中
2.    long  back_link;                      // 前任务链接字段
3.    long  esp0;                           // 特权级 0 堆栈栈中偏移量
4.    long  ss0;                            // 特权级 0 堆栈段选择符
5.    long  esp1;                           // 特权级 1 堆栈栈中偏移量
6.    long  ss1;                            // 特权级 1 堆栈段选择符
7.    long  esp2;                           // 特权级 2 堆栈栈中偏移量
8.    long  ss2;                            // 特权级 2 堆栈段选择符
9.    long  cr3;                            // 页目录基地址寄存器
10.   long  eip;                            // 指令指针
11.   long  eflags;                         // 标志寄存器
12.   long  eax,ecx,edx,ebx;                // 通用寄存器保存字段
13.   long  esp;                            // 保存通用寄存器 esp 内容
14.   long  ebp;                            // 保存通用寄存器 ebp 内容
15.   long  esi;                            // 保存通用寄存器 esi 内容
16.   long  edi;                            // 保存通用寄存器 edi 内容
17.   long  es;                             // es 段选择符保存字段
18.   long  cs;                             // cs 段选择符保存字段
19.   long  ss;                             // ss 段选择符保存字段
20.   long  ds;                             // ds 段选择符保存字段
21.   long  fs;                             // fs 段选择符保存字段
22.   long  gs;                             // gs 段选择符保存字段
23.   long  ldt;                            // 任务 LDT 段选择符
24.   long  trace_bitmap;                   // I/O 位图基地址字段
25.   struct i387_struct i387;              // 数学协处理器结构。保存切换时的执行状态信息
26.   };
```

5.2.3 i387_struct

i387_struct 是数学协处理器使用的数据结构，主要用于保存进程切换时 i387 协处理器的执行状态信息。

```
1.    struct i387_struct {                          // 数学协处理器使用的结构
// 用于保存进程切换时的 i387 的执行状态信息，位于 linux/include/linux/sched.h 中
2.    long  cwd;                                    // 控制字
3.    long  swd;                                    // 状态字
4.    long  twd;                                    // 标记字
5.    long  fip;                                    // 协处理器代码指针
6.    long  fcs;                                    // 协处理器代码段寄存器
7.    long  foo;                                    // 内存操作数的偏移值
8.    long  fos;                                    // 内存操作数的段值
9.    long  st_space[20];                           // 8 个 10 字节的协处理器累加器
10.   };
```

5.2.4 task_union, init_task, current

这三个数据结构均位于 linux/kernel/sched.c 中，用来表示操作系统中的任务联合体。

```
//union task_union 结构体
1.    union task_union {
// 一个任务联合体，该联合体既可以用字节形式表示，也可以用任务结构表示
2.    struct task_struct task;                      // 进程描述符
3.    char stack[PAGE_SIZE];                        // 4KB 大小的数组
4.    };
// init_task 结构体
5.    static union task_union init_task = {INIT_TASK,};
// 定义初始化任务数组，设置第一个任务的信息
// INIT_TASK 定义在 include\linux\sched.h 中
// current 结构体
6.    struct task_struct *current = &(init_task.task);
// 初始化当前任务指针
```

5.2.5 sigaction

结构体 sigaction 主要在 sigaction 信号安装和 sigqueue 信号发送时会用到。它指定了对特定信号的处理，信号所传递的信息，信号处理函数执行过程中应屏蔽掉哪些函数等。

```
1.    struct sigaction {                            // 位于 linux/include/signal.h 中
2.    void (*sa_handler)(int);                      // 用户指定的信号处理函数
3.    sigset_t sa_mask;                             // 指定在信号处理程序执行过程中，哪些信号被阻塞
4.    int sa_flags;                                 // 包含许多标志位，指定改变信号处理过程的信号集
5.    void (*sa_restorer)(void);                    // 恢复函数指针，清理用户态堆栈
6.    };
```

5.2.6 timer_list

操作系统应该能够在将来某个时刻准时调度某个任务，所以需要一种能够保证任务准时调度运行的机制。因此操作系统必须包含一个可周期性中断它的定时器，这个定时器就被称为系统时钟嘀嗒，而 timer_list 数据结构就定义了这么一组定时器。

```
1.  static struct timer_list {          // 定时器链表结构，位于 linux/kernel/sched.c 中
2.  long jiffies;                       // 定时滴答数
3.  void (*fn)();                       // 定时器处理程序
4.  struct timer_list * next;           // 指向下一个定时器
5.  };
```

5.3 fork.c 文件

fork.c 主要用于在系统中创建子进程。除了在进程初始化过程中的第 0 号进程外，其他进程均由 fork.c 中的系统调用 fork()来完成。

5.3.1 功能描述

fork.c 文件主要用于 Linux 的进程管理，包括 verify_area()、copy_mem()、find_empty_process()和 sys_fork()四个函数。

verify_area()是进程空间区域写前验证函数。用于对当前进程的进程空间以页为单位执行写操作前的检测操作。如果页面是只读的，则执行共享检验和复制页面操作。

copy_mem()用于设置新任务的代码和数据段基址、限长并复制页表。

find_empty_process()用于为新进程取得不重复的进程号 last_pid，并返回在任务数组中的任务号。

sys_fork()是主要的用于产生进程的函数。它复制系统进程信息（task[n]）并且设置必要的寄存器，它还整个地复制数据段。

5.3.2 代码注释

fork.c 文件的详细代码注释如下：

```
1.  #include <errno.h>          // 错误号头文件，包含系统中各种出错号
2.  #include <linux/sched.h>// 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
     // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句
3.  #include <linux/kernel.h>  // 内核头文件，含有一些内核常用函数的原形定义
4.  #include <asm/segment.h> // 段操作头文件，定义了有关段寄存器操作的嵌入式汇编函数
5.  #include <asm/system.h>  // 系统头文件，定义了设置或修改描述符/中断门等的嵌入式汇编宏
6.  extern void write_verify(unsigned long address);// 写页面验证，若页面不可写，则复制页面
7.  long last_pid=0;
8.  void verify_area(void * addr,int size)
9.  {
10.   unsigned long start;
11.   start = (unsigned long) addr;
     // 将起始地址 start 赋值为其所在页的左边界开始位置，同时相应地调整验证区域大小
     // 此时，start 是当前进程空间中的线性地址
12.   size += start & 0xfff;
13.   start &= 0xfffff000;
14.   start += get_base(current->ldt[2]);
     // 此时，start 变成系统整个线性空间中的地址位置
15.   while (size>0) {
16.     size -= 4096;                          // 写页面验证。如果页面不可写，则复制页面
17.     write_verify(start);
18.     start += 4096;
```

```
19.       }
20.     }
        // 设置新任务的代码和数据段基址、限长并复制页表
        // nr 为新任务号；p 是新任务数据结构指针
21.     int copy_mem(int nr,struct task_struct * p)
22.     {
23.       unsigned long old_data_base,new_data_base,data_limit;
24.       unsigned long old_code_base,new_code_base,code_limit;
25.       code_limit=get_limit(0x0f);                    // 将局部描述符表中代码段描述符项中的段限
                                                         // 长赋给 code_limit
26.       data_limit=get_limit(0x17);                    // 将局部描述符表中数据段描述符项中的段限
                                                         // 长赋给 data_limit
27.       old_code_base = get_base(current->ldt[1]);     // 取源代码段基址
28.       old_data_base = get_base(current->ldt[2]);     // 取原数据段基址
29.       if (old_data_base != old_code_base) {      // 如果代码段和数据段的基址不重合，死机
30.         printk("ldt[0]: %08x %08x\n",current->ldt[0].a,current->ldt[0].b);
31.         printk("ldt[1]: %08x %08x\n",current->ldt[1].a,current->ldt[1].b);
32.         printk("ldt[2]: %08x %08x\n",current->ldt[2].a,current->ldt[2].b);
33.         panic("We don't support separate I&D");
34.       }
35.       if (data_limit < code_limit)                 // 如果数据段的段限长小于代码段的段限长，死机
36.         panic("Bad data_limit");
37.       new_data_base = new_code_base = nr * TASK_SIZE;
38.       p->start_code = new_code_base;               // 为新进程的代码段和数据段基地址赋值
39.       set_base(p->ldt[1],new_code_base);
40.       set_base(p->ldt[2],new_data_base);           // 为新进程的局部描述符表设置代码段和数据段的值
        // 为新进程设置页目录表项和页表项。将新进程的线性地址内存页对应到实际物理地址内存页上
        // 这个时候新进程与其父进程共享所有的物理页面，如果出错则释放申请的内存
41.       if (copy_page_tables(old_data_base,new_data_base,data_limit)) {
42.         free_page_tables(new_data_base,data_limit);
43.         return -ENOMEM;
44.       }
45.       return 0;
46.     }
        // 复制代码和数据段，如果出错则释放申请的内存
        /* 为新进程取得不重复的进程号 last_pid，并返回在任务数组中的任务号 */
47.     static int find_empty_process(void)
48.     {
49.       int i;
50.       repeat:
51.         if ((++last_pid)<0) last_pid=1;     //检测进程号是否超过正数范围，超过则从 1 重新开始
52.         for(i=0 ; i<NR_TASKS ; i++)
        // 检测指针是否为空，不为空则检测该任务的进程号是否和得到的进程号相等
53.           if (task[i] && ((task[i]->pid == last_pid) ||
54.                           (task[i]->pgrp == last_pid)))
55.             goto repeat;                          // 该进程号已被使用，重新寻找新的进程号
56.       for(i=1 ; i<NR_TASKS ; i++)                 // 遍历任务数组
57.         if (!task[i])                             // 检测当前任务指针是否为空
58.           return i;
59.       return -EAGAIN;
```

```
60.     }
61.     int sys_fork(long ebx,long ecx,long edx,
62.         long esi, long edi, long ebp, long eax, long ds,
63.         long es, long fs, long gs, long orig_eax,
64.         long eip,long cs,long eflags,long esp,long ss)  // 复制进程
65.     {
66.       struct task_struct *p;
67.       int i,nr;
68.       struct file *f;
69.       p = (struct task_struct *) get_free_page();          // 为新任务数据结构分配内存空间
70.       if (!p)
71.         return -EAGAIN;                                   // 如果内存分配出错，则返回出错码并退出
72.       nr = find_empty_process();                          // 否则将任务号赋给 nr
73.       if (nr < 0) {
74.         free_page((unsigned long) p);
75.         return nr;
76.       }
77.       task[nr] = p;                                       // 将新任务结构指针放入任务数组中
78.       *p = *current;
          /* 注意！这样做不会复制超级用户的堆栈 */
          // 只复制当前进程内容
79.       p->state = TASK_UNINTERRUPTIBLE;                    // 将新进程的状态置为不可中断等待状态
80.       p->pid = last_pid;                                  // 新进程号，由前面 find_empty_process()得到
81.       p->counter = p->priority;
82.       p->signal = 0;                                      // 信号位图置 0
83.       p->alarm = 0;                                       // 报警标志置 0
84.       p->leader = 0;
          /* 进程的领导权是不能继承的 */
85.       p->utime = p->stime = 0;                            // 初始化用户态时间和核心态时间
86.       p->cutime = p->cstime = 0;                          // 初始化子进程用户态和核心态时间
87.       p->start_time = jiffies;                            // 当前滴答数时间
          // 以下设置任务状态段 TSS 所需的数据
88.       p->tss.back_link = 0;
89.       p->tss.esp0 = PAGE_SIZE + (long) p;                 // 堆栈指针，ss0:esp0 是在内核态执行时的堆栈
90.       p->tss.ss0 = 0x10;                                  // 堆栈段选择符
91.       p->tss.eip = eip;                                   // 指令代码指针
92.       p->tss.eflags = eflags;                             // 标志寄存器
93.       p->tss.eax = 0;
94.       p->tss.ecx = ecx;
95.       p->tss.edx = edx;
96.       p->tss.ebx = ebx;
97.       p->tss.esp = esp;
98.       p->tss.ebp = ebp;
99.       p->tss.esi = esi;
100.      p->tss.edi = edi;
101.      p->tss.es = es & 0xffff;                            // 段寄存器仅 16 位有效
102.      p->tss.cs = cs & 0xffff;
103.      p->tss.ss = ss & 0xffff;
104.      p->tss.ds = ds & 0xffff;
105.      p->tss.fs = fs & 0xffff;
```

```
106.    p->tss.gs = gs & 0xffff;
107.    p->tss.ldt = _LDT(nr);                          // 该新任务 nr 的局部描述符表选择符
108.    p->tss.trace_bitmap = 0x80000000;
109.    if (last_task_used_math == current)  // 如果当前任务使用了协处理器，就保存其上下文
110.        __asm__("clts ; fnsave %0 ; frstor %0"::"m" (p->tss.i387));
        // 设置新任务的代码和数据段基址、限长并复制页表。如果返回值不是 0，表示出错，则
        复位任务数组中相应项并释放为该新任务分配的内存
111.    if (copy_mem(nr,p)) {
112.        task[nr] = NULL;
113.        free_page((long) p);
114.        return -EAGAIN;
115.    }
116.    for (i=0; i<NR_OPEN;i++)// 如果父进程中有文件是打开的，则将对应文件的打开次数加 1
117.        if (f=p->filp[i])
118.            f->f_count++;
119.    if (current->pwd)     // 将当前进程的 pwd, root, executable 和 library 引用次数均加 1
120.        current->pwd->i_count++;
121.    if (current->root)
122.        current->root->i_count++;
123.    if (current->executable)
124.        current->executable->i_count++;
125.    if (current->library)
126.        current->library->i_count++;
127.    set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
128.    set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
        // 在 GDT 中设置新任务的 TSS 和 LDT 描述符项，数据从 task 结构中取，在任务切换时，
        // 任务寄存器 tr 由 CPU 自动加载
129.    p->p_pptr = current;
130.    p->p_cptr = 0;
131.    p->p_ysptr = 0;
132.    p->p_osptr = current->p_cptr;
133.    if (p->p_osptr)
134.        p->p_osptr->p_ysptr = p;
135.    current->p_cptr = p;
136.    p->state = TASK_RUNNING;
        /* 最后再将新任务设置成可运行状态，以防万一 */
137.    return p->pid;                                  // 返回新进程号
138. }
```


代码中第 8～20 行为：进程空间区域写前验证函数。对当前进程地址 addr 到 addr+size 的进程空间执行写操作前的检测操作，以页为单位。如果页面只读，则执行共享检验和复制页面操作。代码中第 61-138 行为 sys_fork 函数，是主要的 fork 子程序。

5.4 sched.c 文件

内核中与任务调度相关的函数都被定义在 sched.c 文件中，它定义了进程切换时需要的所有功能。它是内核最核心的程序，它负责整个内核的调度，从开机到整个系统的启动，它一直对各个子系统的事件进行处理。

5.4.1 功能描述

此文件主要包括 show_task()、show_state()、math_state_restore()、schedule()、sys_pause()、_sleep_on()、interruptible_sleep_on()、sleep_on()、wake_up()、ticks_to_floppy_on()、floppy_on()、floppy_off()、do_floppy_timer()、add_timer()、do_timer()、sys_alarm()、sched_init()等函数。其中 ticks_to_floppy_on()、floppy_on()、floppy_off()、do_floppy_timer()、add_timer()、do_timer()、sys_alarm()这些函数是一些有关软盘的子程序，将它们放在内核的主要部分里，是因为软驱需要一个时钟，而放在这里是最方便的办法。这里不做详细分析。

show_task()用于显示任务号 nr 的进程号、进程状态和内核堆栈空闲字节数。

show_state()用于显示所有任务的任务号、进程号、进程状态和内核堆栈空闲字节数。

math_state_restore()用于将当前协处理器内容保存到老协处理器状态数组中，并将当前任务的协处理器内容加载进协处理器。

schedule()是主要的进程调度函数，它首先对所有进程进行检测，唤醒任何一个已经得到信号的任务。随后根据进程的时间片和优先权调度机制，来选择随后要执行的任务。通过 switch_to()函数切换到被选择的任务。

sys_pause()，该系统调用将导致进程进入睡眠状态，并重新调度。

_sleep_on()用于把当前任务置为 state 的状态，并让队列头的指针指向当前任务。

interruptible_sleep_on()用于将当前任务置为可中断的等待状态，并放入*p 指定的等待队列中。

sleep_on()用于把当前任务置为不可中断睡眠状态，并让睡眠队列头的指针指向当前任务。

wake_up()用于唤醒指定任务。

sched_init()是调度程序的初始化子程序。

5.4.2 代码注释

sched.c 文件的详细代码注释如下：

```
1.  #include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、初始任务 1 的数据
    // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句
2.  #include <linux/timer.h>  // 内核定时器头文件，包含各种类型定时器号
3.  #include <linux/kernel.h> // 内核头文件，含有一些内核常用函数的原形定义
4.  #include <linux/sys.h>    // 系统调用头文件，含有 72 个系统调用 C 函数处理程序
5.  #include <linux/fdreg.h>  // 软驱头文件，含有软盘控制器参数的一些定义
6.  #include <asm/system.h>  // 系统头文件，定义了设置或修改描述符/中断门等的嵌入式汇编宏
7.  #include <asm/io.h>       // io 头文件，定义硬件端口输入/输出宏汇编语句
8.  #include <asm/segment.h>// 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数
9.  #include <signal.h>                          // 信号头文件。定义信号符号常量，信号结构以及信
                                                // 号操作函数原型
10. #include <errno.h>                           // 错误号头文件。包含系统中各种出错号
11. #define _S(nr) (1<<((nr)-1))                // 取信号 nr 在信号位图中对应位的二进制数值
12. #define _BLOCKABLE (~(_S(SIGKILL) | _S(SIGSTOP)))
    // 除了 SIGKILL 和 SIGSTOP 信号以外其他都是可阻塞的
13. // 显示任务号 nr 的进程号、进程状态和内核堆栈空闲字节数
14. void show_task(int nr,struct task_struct * p)
15. {
16.     int i,j = 4096-sizeof(struct task_struct);
17.     printk("%d: pid=%d, state=%d, father=%d, child=%d, ",nr,p->pid,33 p->state, p->p_pptr->pid,
        p->p_cptr ? p->p_cptr->pid : -1);
```

```
18.         i=0;
19.         while (i<j && !((char *)(p+1))[i])        // 检测指定任务数据结构以后等于 0 的字节数
20.             i++;
21.         printk("%d/%d chars free in kstack\n\r",i,j);
22.         printk("     PC=%08X.", *(1019 + (unsigned long *) p));
23.         if (p->p_ysptr || p->p_osptr)
24.             printk("     Younger sib=%d, older sib=%d\n\r",41 p->p_ysptr ? p->p_ysptr->pid : -1,
25.                 p->p_osptr ? p->p_osptr->pid : -1);
26.         else
27.             printk("\n\r");
28.     }
        //  显示所有任务的任务号、进程号、进程状态和内核堆栈空闲字节数
29.     void show_state(void)
30.     {
31.         int i;
32.         printk("\rTask-info:\n\r");
33.         for (i=0;i<NR_TASKS;i++)              // NR_TASKS 是系统能容纳的最大进程数量（64 个）
34.             if (task[i])
35.                 show_task(i,task[i]);
36.     }
37.     #define LATCH (1193180/HZ)                // 定义每个时间片的滴答数
38.     extern void mem_use(void);
39.     extern int timer_interrupt(void);         // 时钟中断处理程序
40.     extern int system_call(void);             // 系统调用中断处理程序
41.     union task_union {                        // 定义任务联合体（任务结构成员和 stack 字符数组
                                                  // 程序成员）
42.         struct task_struct task;
43.         char stack[PAGE_SIZE];
44.     } ;
        //  因为一个任务数据结构与其堆栈放在同一内存页中，
        //  所以从堆栈段寄存器 ss 可以获得其数据段选择符
45.     static union task_union init_task = {INIT_TASK,};     // 定义初始任务的数据
46.     unsigned long volatile jiffies=0;         // 从开机开始算起的滴答数值
47.     unsigned long startup_time=0;             // 开机时间，从 1970:0:0:0 开始计时的秒数
48.     int jiffies_offset = 0;
49.
50.     struct task_struct *current = &(init_task.task);      // 将当前任务指针初始化为初始任务
51.     struct task_struct *last_task_used_math = NULL;       // 使用过协处理器任务的指针
52.     struct task_struct * task[NR_TASKS] = {&(init_task.task), }; // 定义任务指针数组
53.     long user_stack [ PAGE_SIZE>>2 ] ;        // 定义堆栈指针，4KB，指针指在最后一项
54.     struct {
55.         long * a;
56.         short b;
57.         } stack_start = { & user_stack [PAGE_SIZE>>2] , 0x10 };
        //  该结构用于设置堆栈 ss:esp（数据段选择符，指针）
58.     void math_state_restore()
59.     {
60.         if (last_task_used_math == current)   // 如果上一个任务就是当前任务，则返回
61.             return;
62.         __asm__("fwait");                     // 在发送协处理器命令之前要先发 WAIT 指令
```

```
63.         if (last_task_used_math) {
64.             __asm__("fnsave %0"::"m" (last_task_used_math->tss.i387));
65.         }                                          // 如果上个任务使用了协处理器，则保存其状态
66.         last_task_used_math=current;
    // last_task_used_math 指向当前任务，以备当前任务被交换出去时使用
67.         if (current->used_math) {                  // 如果当前任务用过协处理器，恢复其状态
68.             __asm__("frstor %0"::"m" (current->tss.i387));
69.         } else {                                   // 否则
70.             __asm__("fninit"::);                   // 向协处理器发初始化命令
71.             current->used_math=1;                  // 并设置使用了协处理器标志
72.         }
73.     }
74.     void schedule(void)
75.     {
76.         int i,next,c;
77.         struct task_struct ** p;                   // 任务结构指针的指针
78.
79.
    // 从任务数组中最后一个任务开始检测 timeout 以及 alarm
80.         for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
81.             if (*p) {
82.                 if ((*p)->timeout && (*p)->timeout < jiffies) {
83.                     (*p)->timeout = 0;
84.                     if ((*p)->state == TASK_INTERRUPTIBLE)
85.                         (*p)->state = TASK_RUNNING;
86.                 }
    // 如果任务的 alarm 时间已经过期，则在信号位图中置 SIGALRM 信号，然后，清 alarm
87.                 if ((*p)->alarm && (*p)->alarm < jiffies) {
88.                     (*p)->signal |= (1<<(SIGALRM-1));
89.                     (*p)->alarm = 0;
90.                 }
    // 如果信号位图中除被阻塞的信号外还有其他信号，且任务处于可中断状态，置任务为就绪状态
91.                 if (((*p)->signal & ~(*p)->blocked) &&
92.                 (*p)->state==TASK_INTERRUPTIBLE)
93.                     (*p)->state=TASK_RUNNING;          // 置为就绪（可执行）状态
94.             }
95.         while (1) {
96.             c = -1;
97.             next = 0;
98.             i = NR_TASKS;
99.             p = &task[NR_TASKS];
100.            while (--i) {
101.                if (!*--p)
102.                    continue;
103.                if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
104.                    c = (*p)->counter, next = i;
105.            }
    //  如果比较得出有 counter 值大于 0 的结果，则退出 while 循环，执行任务切换
106.            if (c) break;
    // 否则，根据每个任务的优先权值，更新每一个任务的 counter 值，然后回到 while 循环重新比较
```

```
107.                for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
108.                        if (*p)
109.                                (*p)->counter = ((*p)->counter >> 1) +
110.                                        (*p)->priority;
111.        }
112.        switch_to(next);                                    // 切换到任务号为 next 的任务，并运行
113.    }
114.    int sys_pause(void)
115.    {
116.            unsigned long old_blocked;
117.            unsigned long mask;
118.            struct sigaction * sa = current->sigaction;
119.        old_blocked = current->blocked;
120.        for (mask=1 ; mask ; sa++,mask += mask)
121.                if (sa->sa_handler == SIG_IGN)
122.                        current->blocked |= mask;
123.        current->state = TASK_INTERRUPTIBLE;
124.        schedule();
125.        current->blocked = old_blocked;
126.        return -EINTR;
127.    }
128.    static inline void __sleep_on(struct task_struct **p, int state)
129.    {
130.            struct task_struct *tmp;
131.            unsigned int flags;
132.        if (!p)                                             // 若指针无效，则退出
133.                return;
134.        if (current == &(init_task.task))                   // 如果当前任务是任务 0，则死机
135.                panic("task[0] trying to sleep");
136.        __asm__("pushfl ; popl %0":"=r" (flags));
137.        tmp = *p;                                           // 让 tmp 指向已经在等待队列上的任务
138.        *p = current;                                       // 将睡眠队列头的等待指针指向当前任务
139.        current->state = state;                             // 将当前任务置为不可中断的等待状态
140.    /* 确定中断可用，此时应该没有进程调度冲突 */
141.        sti();                                              // 打开中断
142.    repeat:    schedule();                                  // 调度，上下文切换
143.        if (*p && *p != current) {
144.                current->state = TASK_UNINTERRUPTIBLE;
145.                (**p).state = 0;
146.                goto repeat;
147.        }
148.        if (*p = tmp)                                       // 若还存在等待的任务，则也将其置为就绪状态
149.                tmp->state=0;
150.        __asm__("pushl %0 ; popfl"::"r" (flags));
151.    }
        // 将当前任务置为可中断的等待状态，并放入*p 指定的等待队列中
152.    void interruptible_sleep_on(struct task_struct **p)
153.    {
154.        __sleep_on(p,TASK_INTERRUPTIBLE);
155.    }
```

```
        // 把当前任务置为不可中断睡眠状态，并让睡眠队列头的指针指向当前任务
156.    void sleep_on(struct task_struct **p)
157.    {
158.        __sleep_on(p,TASK_UNINTERRUPTIBLE);
159.    }
        // 唤醒指定任务*p
160.    void wake_up(struct task_struct **p)
161.    {
162.        if (p && *p) {
163.            if ((**p).state == TASK_STOPPED)
164.                printk("wake_up: TASK_STOPPED");
165.            if ((**p).state == TASK_ZOMBIE)
166.                printk("wake_up: TASK_ZOMBIE");
167.            (**p).state=0;                          // 置为就绪状态
168.        }
169.    }
170.    static struct task_struct * wait_motor[4] = {NULL,NULL,NULL,NULL};
171.    static int    mon_timer[4]={0,0,0,0};
172.    static int moff_timer[4]={0,0,0,0};
173.    unsigned char current_DOR = 0x0C;               // 数字输出寄存器
174.     // 指定软盘到正常运转状态所需延迟滴答数。nr-软驱号（0～3），返回值为滴答数
175.    int ticks_to_floppy_on(unsigned int nr)
176.    {
177.        extern unsigned char selected;              // 当前选中的软盘号
178.        unsigned char mask = 0x10 << nr;            //所选软驱对应数字输出寄存器中启动马达比特位
179.        if (nr>3)                                   // 最多 4 个软驱
180.            panic("floppy_on: nr>3");
181.        moff_timer[nr]=10000;
182.        cli();
183.        mask |= current_DOR;
        // 如果不是当前软驱，则首先复位其他软驱的选择位，然后置对应软驱选择位
184.        if (!selected) {
185.            mask &= 0xFC;
186.            mask |= nr;
187.        }
188.        if (mask != current_DOR) {
189.            outb(mask,FD_DOR);
190.            if ((mask ^ current_DOR) & 0xf0)
191.                mon_timer[nr] = HZ/2;
192.            else if (mon_timer[nr] < 2)
193.                mon_timer[nr] = 2;
194.            current_DOR = mask;
195.        }
196.        sti();
197.        return mon_timer[nr];
198.    }
        //  等待指定软驱马达启动所需时间
199.    void floppy_on(unsigned int nr)
200.    {
201.        cli();                                      // 关中断
```

ÓÉ Foxit PDF Editor ±à¼-
°æÈ¨ËùÓÐ (c) by Foxit Software Company, 2004
½öÓÃÓÚÆÀ¹À¡£

- android与iphone及ipad开发书籍 ----------持续不断更新中.....
- c、c++、c#语言pdf书籍及vip视频教程 c、c++、c#、vc等-------持续不断更新中.....
- delphi《书籍》及《视频》教程 ----------持续不断更新中.....
- E网情深VIP系列视频教程 黑客破解菜鸟修练班，VB编程学习班，仿站学习培训，免杀培训，个人系统攻防系列教程，服务器搭建学习班，PHOTOSHOP平面设计班，基础制作论坛（论坛网站搭建），网赚系列教程，网站建设教程，网站漏洞基础，远程控制教程，软件破解班，脚本漏洞提权班
- IT9网络学院VIP系列视频教程 免杀培训班，VMware虚拟机，零基础学习C语言，网游外挂开发精品系列语音教程（外挂教程学习必备研修31课全），VB语言教程30课全，Delphi编程到精通，远程控制软件，加密解密班，网络安全与黑客攻防培训，从入门到精通完整系统化学习C++编程，从入门到精通零基础学习汇编，wordpress教程(个人博客系统49课全)，外行人做易语言盗号和钓鱼程序语音教程 网址：WLSAM168.400GB.COM
- Java书籍 ----------持续不断更新中.....
- photoshop、CorelDRAW、AutocAD等图像处理书籍及vip视频教程 -------持续不断更新中.....
- powerbuilder书籍大全
- Visual Basic语言vip视频教程及pdf书籍 ----------持续不断更新中.....
- windows、linux系统开发、系统封装等pdf书籍及VIP视频教程 ----------持续不断更新中.....
- 《3DS Max》pdf书籍
- 《汇编语言》、《反汇编》及《调试》pdf书籍及vip视频教程 ----------持续不断更新中.....
- 《电子书、电子书、还是电子书》pdf专题库 编程开发，家居美食，儿童益智，人物传记，增强记忆，快速阅读
- 信息系统项目管理师、网络工程师、系统分析师等软考类书籍
- 华中红客系列vip视频教程 脚本攻防培训班，源码免杀培训班，Css语言培训班，C语言，Dreamweaver网页设计，html网页设计培训班，PC安全班， php脚本语言培训班，VMWare虚拟机专题，webshell提权培训班，防站教程，零基础免杀培训班，刷钻速成班，脱壳破解班，外挂编写班，网络赚钱培训班，网站入侵培训班
- 外挂、驱动、逆向及封包视频教程 郁金香、独立团、夜猫论坛、天都吧、看流星论坛、一切从零开始等等
- 安全中国系列vip视频教程 易语言软件编程培训班，ASP.net网站开发项目实战培训班
- 我的收藏
- 按键精灵及TC脚本开发软件视频教程 ----------持续不断更新中.....

当前位置：/《电子书、电子书、还是电子书》pdf专题库

文件名

PDF电子书专题库，内容详尽，每天不断更新！！

- 办公类软件使用指南
- 医学
- 历史人物传记
- 哲学宗教
- 外语资料（除英语外） （除英语外）
- 官场类小说
- 建筑工程类
- 情感生活类小说
- 政治军事
- 教育学习科普大全
- 文学理论
- 智力开发、增强记忆、快速阅读技巧大全
- 社会生活
- 科学技术
- 程序编程类
- 经济管理
- 网络安全及管理
- 网赚系列
- 美食小吃烹饪煲汤大全
- 课外读物

本网盘内容太多，持续不断更新，发布各类视频教程、pdf书籍，包括破解、加解密、外挂辅助制作，易语言培训教程、编程语言、网页制作等等，教程及书籍仅用于学习,如用于商业或非法用途的后果自负！

网址：WLSAM168.400GB.COM

VIP培训教程，易语言黑月VIP视频教程，天都网易语言系列培训教程(100集全)，集中营易语言学习视频(80集)

- 棉猴系列vip视频教程 gh0st远控控制源码讲解教程，套接字编程，DLL程序编写，键盘监听驱动程序编写，驱动基础教程，AsyncSelect模型QQ程序教程，C++语言入门基础，NB5.5源码分析教程
- 游戏开发pdf书籍 ----------持续不断更新中.....
- 炒股投资pdf书籍及视频教程 短线高手系列，短线天王系列，操盘论道系列，翻倍黑马，看盘快速入门，庄家手法大曝光等等。 网址：WLSAM168.400GB.COM
- 热门小说集中营 傲世九重天，网游之三国时代，武动乾坤
- 甲壳虫VIP教程全集 asp教程，Delphi培训班，FLASH培训班，Java培训班，linux培训班，PHP培训班，源码免杀班，甲壳虫C++，脚本功防班，免杀班初、中、高级班，破解班，源码免杀班，脱壳班，易语言培训班，无特征码免杀，网站架构培训班，外挂高级班，外挂初级班第1、2部
- 破解、免杀、入侵、脱壳、攻防及漏洞分析系列VIP视频教程（80多部） 天草、黑客动画吧等等--------持续不断更新中....
- 网站建设相关的pdf书籍及各种vip视频教程 ----------持续不断更新中.....
- 网赚、淘宝系列vip视频教程 网赚30天新人魔鬼训练，屠龙网赚团队vip课程，站长大学网赚视频（50课全），图腾团队日赚1000元竞价行销教程，屠龙团队淘宝宝贝卖疯系列，站群网赚系列，淘宝开店视频，红星挂机日赚10元，百万流量系列，漂流瓶圣手全自动挂机引，贴吧邮件定向营销疯狂成交量月入万元
- 英语学习资料百科大全 不断更新。。。
- 饭客论坛系列VIP视频教程 脚本入侵班，黑客之免杀教程，易语言教程，无线网络攻防教程，入侵教程，delphi系列教程，黑客基础入门
- 黑客书籍 有关黑客、安全、加解密技术等等----------持续不断更新中.....
- 黑手安全网VIP系列视频教程 DIV+CSS网页布局，Dreamweaver教程，flsah动画教程，photoshop教程，跟我一起学C++课程，抓鸡
- 黑鹰、黑基、黑防、黑盾vip系列视频教程 破解提高班66课全，SQL注入，ASP注入教程，完完全全学会抓肉鸡，脱壳破解教程50课全，提权班，C语言特训班26讲全，黑客脚本特训班，黑客工具特训班，dedecms仿站教程，VC编写远控30课全，网页美工特训班，木马免杀特训班，驱动开发技术VIP培训班，外挂破解等等。

- [电脑世界的通关密语：电脑编程基础].(杉浦贤).滕永红.扫描版.pdf
- [程序语言的奥妙：算法解读（四色全彩）].(杉浦贤).李克秋.扫描版.pdf
- [差错：软件错误的致命影响].(帕伯斯).邝宇恒等.扫描版.pdf
- [算法之道（第2版）].邹恒明.扫描版.pdf
- [O'Reilly：深入学习MongoDB].(霍多罗夫).巨成等.扫描版.pdf
- [深入浅出WPF].刘铁猛.扫描版.pdf
- [Go语言·云动力（云计算时代的新型编程语言）].樊虹剑.扫描版.pdf
- [精通.NET互操作：P/ Invoke、C++ Interop和COM Interop].黄际洲等.扫描版.pdf
- [编程的奥秘：.NET软件技术学习与实践].金旭亮.扫描版.pdf
- [O'Reilly：学习OpenCV（中文版）].(布拉德斯基等).于仕琪等.扫描版.pdf
- [Go语言编程].许式伟等.扫描版.pdf 网址：WLSAM168.400GB.COM
- [MySQL技术内幕：SQL编程].姜承尧.扫描版.pdf
- [Tomcat权威指南（第2版）].(布里泰恩等).吴豪等.扫描版.pdf
- [Ext江湖].大漠穷秋.扫描版.pdf
- [IT名人堂·Oracle DBA突击：帮你赢得一份DBA职位].张晓明.扫描版.pdf

Total: 77 1 2 3 4 5 6 ›

HTTP://WLSAM168.400GB.COM

```
202.        while (ticks_to_floppy_on(nr))
203.                sleep_on(nr+wait_motor);
        // 如果马达启动定时还没到，就一直把当前进程置为不可中断睡眠状态，并放入等待马达运行
        // 的队列中
204.        sti();                                          // 开中断
205.    }
206.     // 置关闭相应软驱马达停转定时器
207.    void floppy_off(unsigned int nr)
208.    {
209.        moff_timer[nr]=3*HZ;
210.    }
        // 软盘定时处理子程序
211.    void do_floppy_timer(void)
212.    {
213.        int i;
214.        unsigned char mask = 0x10;
215.        for (i=0 ; i<4 ; i++,mask <<= 1) {
216.                if (!(mask & current_DOR))              // 如果不是 DOR 指定的马达，则跳过
217.                        continue;
218.                if (mon_timer[i]) {
219.                        if (!--mon_timer[i])
220.                                wake_up(i+wait_motor);  // 如果马达启动定时到，则唤醒进程
221.                } else if (!moff_timer[i]) {            // 如果马达停转定时到
222.                        current_DOR &= ~mask;           // 复位相应马达启动位
223.                        outb(current_DOR,FD_DOR);       // 更新数字输出寄存器
224.                } else
225.                        moff_timer[i]--;                // 马达停转计时递减
226.        }
227.    }
228.    #define TIME_REQUESTS 64
        // 定时器链表结构和定时器数组
229.    static struct timer_list {
230.        long jiffies;                                   // 定时滴答数
231.        void (*fn)();                                   // 定时处理程序
232.        struct timer_list * next;                       // 下一个定时器
233.    } timer_list[TIME_REQUESTS], * next_timer = NULL;
234.    // 添加定时器。输入参数为指定的定时值（滴答数）和相应的处理程序指针
235.    void add_timer(long jiffies, void (*fn)(void))
236.    {
237.        struct timer_list * p;
        // 如果定时处理程序指针为空，则退出
238.        if (!fn)
239.                return;
240.        cli();
241.        if (jiffies <= 0)   // 如果定时值<=0，则立刻调用其处理程序。并且该定时器不加入链表中
242.                (fn)();
243.        else {                                          // 从定时器数组中找一个空闲项
244.                for (p = timer_list ; p < timer_list + TIME_REQUESTS ; p++)
245.                        if (!p->fn)
246.                                break;
```

```
247.                if (p >= timer_list + TIME_REQUESTS)// 如果已经用完了定时器数组，则系统崩溃
248.                    panic("No more time requests free");    // 向定时器数据结构填入相应信息
249.                p->fn = fn;
250.                p->jiffies = jiffies;
251.                p->next = next_timer;
252.                next_timer = p;
        // 链表项按定时值从小到大排序。在排序时减去排在前面需要的滴答数，
        // 这样在处理定时器时只要查看链表头的第一项的定时是否到期即可
253.                while (p->next && p->next->jiffies < p->jiffies) {
254.                    p->jiffies -= p->next->jiffies;
255.                    fn = p->fn;
256.                    p->fn = p->next->fn;
257.                    p->next->fn = fn;
258.                    jiffies = p->jiffies;
259.                    p->jiffies = p->next->jiffies;
260.                    p->next->jiffies = jiffies;
261.                    p = p->next;
262.                }
263.        }
264.        sti();
265.    }
266.    unsigned long timer_active = 0;
267.    struct timer_struct timer_table[32];
268.    void do_timer(long cpl)
269.    {
270.        unsigned long mask;
271.        struct timer_struct *tp = timer_table+0;
272.        for (mask = 1 ; mask ; tp++,mask += mask) {
273.            if (mask > timer_active)
274.                break;
275.            if (!(mask & timer_active))
276.                continue;
277.            if (tp->expires > jiffies)
278.                continue;
279.            timer_active &= ~mask;
280.            tp->fn();
281.        }
282.    // 如果当前特权级（cpl）为 0（最高，表示是内核程序在工作），则将超级用户运行时间 stime 递增；
        // 如果 cpl > 0，则表示是一般用户程序在工作，则增加 utime
283.        if (cpl)
284.            current->utime++;
285.        else
286.            current->stime++;
287.        if (next_timer) {                                   // next_timer 是定时器链表的头指针
288.            next_timer->jiffies--;
289.            while (next_timer && next_timer->jiffies <= 0) {
290.                void (*fn)(void);                           // 这里插入了一个函数指针定义
291.                fn = next_timer->fn;
292.                next_timer->fn = NULL;
293.                next_timer = next_timer->next;
```

```
294.                    (fn)();                                 // 调用处理函数
295.                }
296.        }
        // 如果当前软盘控制器 FDC 的数字输出寄存器中马达启动位有置位的，执行软盘定时程序
297.        if (current_DOR & 0xf0)
298.                do_floppy_timer();
299.        if ((--current->counter)>0) return;                 // 如果进程运行时间还没完，则退出
300.        current->counter=0;
301.        if (!cpl) return;                                   // 对于超级用户程序，不依赖 counter 值进行调度
302.        schedule();
303.    }
        // 系统调用功能，设置报警定时时间值（秒）。如果已经设置过 alarm 值，则返回旧值，否则返回 0
304.    int sys_alarm(long seconds)
305.    {
306.        int old = current->alarm;
307.        if (old)
308.                old = (old - jiffies) / HZ;
309.        current->alarm = (seconds>0)?(jiffies+HZ*seconds):0;
310.        return (old);
311.    }
        //  取当前进程号 pid
312.    int sys_getpid(void)
313.    {
314.        return current->pid;
315.    }
        // 取父进程号 ppid
316.    int sys_getppid(void)
317.    {
318.        return current->p_pptr->pid;
319.    }
        // 取用户号 uid
320.    int sys_getuid(void)
321.    {
322.        return current->uid;
323.    }
        // 取有效用户号 euid
324.    int sys_geteuid(void)
325.    {
326.        return current->euid;
327.    }
        // 取组号 gid
328.    int sys_getgid(void)
329.    {
330.        return current->gid;
331.    }
        // 取有效组号 egid
332.    int sys_getegid(void)
333.    {
334.        return current->egid;
335.    }
```

```
    // 系统调用功能，降低对 CPU 的使用优先权
336. int sys_nice(long increment)
337. {
338.     if (increment < 0 && !suser())
339.         return -EPERM;
340.     if (increment > current->priority)
341.         increment = current->priority-1;
342.     current->priority -= increment;
343.     return 0;
344. }
    //  调度程序的初始化子程序
345. void sched_init(void)
346. {
347.     int i;
348.     struct desc_struct * p;                    // 描述符表结构指针
349.     if (sizeof(struct sigaction) != 16)        // sigaction 存放有关信号状态
350.         panic("Struct sigaction MUST be 16 bytes");
    // 设置初始任务的任务状态段描述符和局部数据表描述符
351.     set_tss_desc(gdt+FIRST_TSS_ENTRY,&(init_task.task.tss));
352.     set_ldt_desc(gdt+FIRST_LDT_ENTRY,&(init_task.task.ldt));
    // 清任务数组和描述符表项
353.     p = gdt+2+FIRST_TSS_ENTRY;
354.     for(i=1 ; i<NR_TASKS ; i++) {
355.         task[i] = NULL;
356.         p->a=p->b=0;
357.         p++;
358.         p->a=p->b=0;
359.         p++;
360.     }
361.     __asm__("pushfl ; andl $0xffffbfff,(%esp) ; popfl");      // 复位 NT 标志
362.     ltr(0);                                    // 将任务 0 的 TSS 加载到任务寄存器 tr
363.     lldt(0);                                   // 将局部描述符表加载到局部描述符表寄存器
    // 初始化 8253 定时器
364.     outb_p(0x36,0x43);
365.     outb_p(LATCH & 0xff , 0x40);               // 定时值低字节
366.     outb(LATCH >> 8 , 0x40);                   // 定时值高字节
    // 设置时钟中断处理程序句柄
367.     set_intr_gate(0x20,&timer_interrupt);
    // 修改中断控制器屏蔽码，允许时钟中断
368.     outb(inb_p(0x21)&~0x01,0x21);
    // 设置系统调用中断门
369.     set_system_gate(0x80,&system_call);
370. }
```

该文件中 58～73 行为 math_state_restore()函数，它将当前协处理器内容保存到老协处理器状态数组中，并将当前任务的协处理器内容加载进协处理器。当任务被调度交换过以后，该函数用以保存原任务的协处理器状态（上下文），并恢复新调度进来的当前任务的协处理器执行状态。

该文件中 74～113 为 schedule()函数，它是调度函数。这是个很好的代码，没有任何理由对它进行修改，因为它可以在所有的环境下工作。只有一件事值得留意，那就是这里的信号处理代

码。注意，任务 0 是闲置任务，只有当没有其他任务可以运行时才调用它。它不能被杀死，也不能睡眠。任务 0 中的状态信息 state 是从来不用的。

该文件中 114～127 行为：pause()系统调用。转换当前任务的状态为可中断的等待状态，并重新调度。该系统调用将导致进程进入睡眠状态，直到收到一个信号。该信号用于终止进程或者使进程调用一个信号捕获函数。只有当捕获了一个信号，并且信号捕获处理函数返回，pause()才会返回。此时 pause()返回值应该是-1。

该文件 128～151 行函数的作用为：把当前任务置为 state 的状态，并让队列头的指针指向当前任务。只有明确地唤醒时才会返回。该函数提供了进程与中断处理程序之间的同步机制。函数参数*p 是任务的队列头指针。该文件中 175～265 行是有关软盘的子程序，本不应该放在内核的主要部分中，将它们放在那里是因为软驱需要一个时钟，而放在那里是最方便的办法。

5.5 signal.c 文件

信号是 Linux 系统中一个重要的进程控制方法，像中断一样，可通过向进程发送不同的信号临时中断程序的正常运行而进入信号处理程序或执行默认的信号响应，如重新更新程序配置、终止进程等。在用户空间中，信号处理接口是通过一系列系统调用来实现的，进程可通过这些接口定义进程的信号处理函数。如果要向其他进程发信号，可通过 kill 系统调用实现。在内核中对信号的处理程序主要定义在 signal.c 文件中。

5.5.1 功能描述

signal.c 文件定义了有关信号处理的一些系统调用以及一个在对应的中断处理程序中处理信号的函数 do_signal()。该文件主要包括 sys_sgetmask()、sys_ssetmask()、sys_sigpending()、sys_sigsuspend()、save_old()、get_new()、sys_signal()、sys_sigaction()、core_dump()、do_signal()等函数。

sys_sgetmask()用于获取当前进程信号屏蔽位图。

sys_ssetmask()用于设置新的信号屏蔽位图。

sys_sigpending()是进程阻塞函数，检测 set 所指向页面的写验证，将当前进程放在 set 所指向的内存处。

sys_sigsuspend()，在新的信号屏蔽码中的原子级交换，并且等待一个信号。

save_old()用于复制 sigaction 数据到 fs 数据段 to 处。

get_new()用于把 sigaction 数据从 fs 数据段 from 位置复制到 to 处。

sys_signal()是 signal()系统调用。与 sigaction()相似，为指定的信号安装新的信号句柄。

sys_sigaction()与 signal()类似，不同之处在于 signal()会返回原信号的处理句柄，并且在新句柄被调用一次后句柄就会恢复到默认值。而 sys_sigaction()可以进行自由的设置。

core_dump()，内核发生故障时的转储函数，目前还未完全实现。

do_signal()是内核系统调用中断处理程序中信号的预处理程序。它将信号的处理句柄插入到用户程序堆栈中。并在本系统调用结束返回后立刻执行信号句柄程序，然后继续执行用户的程序。

5.5.2 代码注释

signal.c 文件的详细代码注释如下：

```
1.  #include <linux/sched.h>                    // 调度程序头文件
2.  #include <linux/kernel.h>                   // 内核头文件，含有一些内核常用函数的原形定义
3.  #include <asm/segment.h>                    // 段操作头文件
4.  #include <signal.h>                         // 信号头文件，定义信号符号常量、信号结构以及信
                                                // 号操作函数原型
5.  #include <sys/wait.h>                       // 等待头文件，包括进程等待信息的一些宏
6.  #include <errno.h>                          // 错误号头文件，包含系统各种出错号
7.  int send_sig (int, struct task_struct *, int);
    // 获取当前任务信号屏蔽位图
8.  int sys_sgetmask()
9.  {
10.         return current->blocked;
11.     }
    // 设置新的信号屏蔽位图。SIGKILL 是不可以被屏蔽的。函数的返回值是原信号的屏蔽位图
12.     int sys_ssetmask(int newmask)
13.     {
14.         int old=current->blocked;
15.         current->blocked = newmask & ~(1<<(SIGKILL-1)) & ~(1<<(SIGSTOP-1));
16.         return old;
17.     }
    // 进程阻塞函数，检测 set 所指向页面的写验证，将当前进程放在 set 所指向的内存处
18.     int sys_sigpending(sigset_t *set)
19.     {
20.         verify_area(set,4);
21.         put_fs_long(current->blocked & current->signal, (unsigned long *)set);
22.         return 0;
23.     }
24.     int sys_sigsuspend(int restart, unsigned long old_mask, unsigned long set)
25.     {
26.         extern int sys_pause(void);
27.         if (restart) {
28.                             /*  重启  */
29.         current->blocked = old_mask;
30.         return -EINTR;
31.         }
32.
33.         *(&restart) = 1;
34.         *(&old_mask) = current->blocked;
35.         current->blocked = set;
36.         (void) sys_pause();                    /* 信号到达时返回 */
37.         return -ERESTARTNOINTR;
38.     }
    //  复制 sigaction 数据到 fs 数据段 to 处
39.     static inline void save_old(char * from,char * to)
40.     {
41.         int i;
42.         verify_area(to, sizeof(struct sigaction)); // 验证 to 处的内存是否足够
43.         for (i=0 ; i< sizeof(struct sigaction) ; i++) {
44.             put_fs_byte(*from,to);             // 复制指定内容到 to 所指向的内存地址处
45.             from++;
```

```
46.                 to++;
47.             }
48.         }
        //  把 sigaction 数据从 fs 数据段 from 位置复制到 to 处
49.         static inline void get_new(char * from,char * to)
50.         {
51.             int i;
52.             for (i=0 ; i< sizeof(struct sigaction) ; i++)
53.                 *(to++) = get_fs_byte(from++);
54.         }
55.         int sys_signal(int signum, long handler, long restorer)
56.         {
57.             struct sigaction tmp;
58.             if (signum<1 || signum>32 || signum==SIGKILL || signum==SIGSTOP)
        // 信号值要在（1～2）范围内，并且不得是 SIGKILL
59.                 return -EINVAL;
60.             tmp.sa_handler = (void (*)(int)) handler;              // 指定的信号处理句柄
61.             tmp.sa_mask = 0;                                       // 执行时的信号屏蔽码
62.             tmp.sa_flags = SA_ONESHOT | SA_NOMASK;
        //  该句柄在使用 1 次后，恢复到默认值，并允许信号在自己的处理句柄中收到
63.             tmp.sa_restorer = (void (*)(void)) restorer;           // 保存返回地址
64.             handler = (long) current->sigaction[signum-1].sa_handler;
65.             current->sigaction[signum-1] = tmp;
66.             return handler;
67.         }
68.         int sys_sigaction(int signum, const struct sigaction * action,
69.             struct sigaction * oldaction)
70.         {
71.             struct sigaction tmp;
        // 信号值要在（1～32）范围内，并且信号 SIGKILL 的处理句柄不能被改变
72.             if (signum<1 || signum>32 || signum==SIGKILL || signum==SIGSTOP)
73.                 return -EINVAL;
74.             tmp = current->sigaction[signum-1];                    // 在信号的 sigaction 结构中设置新的操作
75.             get_new((char *) action,
76.                 (char *) (signum-1+current->sigaction));
        // 如果 oldaction 指针不为空，则将原操作指针保存到 oldaction 所指的位置
77.             if (oldaction)
78.                 save_old((char *) &tmp,(char *) oldaction);
        // 如果允许信号在自己的信号句柄中收到，则令屏蔽码为 0，否则设置屏蔽本信号
79.             if (current->sigaction[signum-1].sa_flags & SA_NOMASK)
80.                 current->sigaction[signum-1].sa_mask = 0;
81.             else
82.                 current->sigaction[signum-1].sa_mask |= (1<<(signum-1));
83.             return 0;
84.         }
85.
86.
87.
        /*  在当前目录写了一个处理 core dump 的函数，但该函数目前没有被实现 */
88.         int core_dump(long signr)
```

```
89.     {
90.         return(0);                  /* 没有进行转储 */
91.     }
92.     extern int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options);
        // 该段代码是将信号的处理句柄插入到用户程序堆栈中，并在本系统调用结束，返回后立刻执
        // 行信号句柄程序，然后继续执行用户的程序
93.     int do_signal(long signr,long ebx, long ecx, long edx,
94.                   long esi, long edi, long ebp, long eax,
95.                   long ds, long es, long fs, long gs,
96.                   long orig_eax,
97.         long eip, long cs, long eflags,
98.         unsigned long * esp, long ss)
99.     {
100.          unsigned long sa_handler;
101.          long old_eip=eip;
102.          struct sigaction * sa = current->sigaction + signr - 1;
103.          int longs;
104.        unsigned long * tmp_esp;
105.    #ifdef notdef                       // 如果定义了 notdef，则输出当前进程号，以及相关寄存器值
106.        printk("pid: %d, signr: %x, eax=%d, oeax = %d, int=%d\n",
107.                current->pid, signr, eax. orig_eax,
108.                sa->sa_flags & SA_INTERRUPT);
109.    #endif
110.        if ((orig_eax != -1) &&
111.            ((eax == -ERESTARTSYS) || (eax == -ERESTARTNOINTR))) {
112.            if ((eax == -ERESTARTSYS) && ((sa->sa_flags & SA_INTERRUPT) ||
113.                signr < SIGCONT || signr > SIGTTOU))
114.                *(&eax) = -EINTR;
115.            else {
116.                *(&eax) = orig_eax;
117.                *(&eip) = old_eip -= 2;
118.            }
119.        }
120.        sa_handler = (unsigned long) sa->sa_handler;
121.        if (sa_handler==1) {                // 如果信号句柄为 SIG_IGN（忽略）
122.
123.            if (signr == SIGCHLD)       // 确认是否是子进程停止或被中止
124.                while (sys_waitpid(-1,NULL,WNOHANG) > 0)
125.
126.            return(1);
127.         }
128.        if (!sa_handler) {              // 如果句柄为 SIG_DFL(默认处理)
129.            switch (signr) {
130.            case SIGCONT:
131.            case SIGCHLD:
132.            case SIGWINCH:      // 否则如果信号是 SIGCONT,SIGCHLD,SIGWINCH，则返回 1
133.                return(1);
134.            case SIGSTOP:
135.            case SIGTSTP:
136.            case SIGTTIN:
```

```
137.            case SIGTTOU:                   // 如果信号是 SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU,
138.                    current->state = TASK_STOPPED;                          // 中止当前进程
139.                    current->exit_code = signr;
140.                    if (!(current->p_pptr->sigaction[SIGCHLD-1].sa_flags &  // 当子进程未停止
141.                                    SA_NOCLDSTOP))
142.                      send_sig(SIGCHLD, current->p_pptr, 1);// 将终止子进程信号赋给父进程
143.    /*                      current->p_pptr->signal |= (1<<(SIGCHLD-1));*/
144.                    return(1);
145.            case SIGQUIT:
146.            case SIGILL:
147.            case SIGTRAP:
148.            case SIGIOT:
149.            case SIGFPE:
150.            case SIGSEGV:
        // 如果信号是 SIGQUIT,SIGILL,SIGTRAP,SIGIOT,SIGFPE,SIGSEGV，则判断以下情况
151.                    if (core_dump(signr))
152.                            do_exit(signr|0x80);
153.
154.            default:
155.                    do_exit(signr);         // 否则终止进程的执行
156.            }
157.    }
158.
159.
        /*  调用一个句柄 */
        //  如果该信号句柄只使用一次，则将其置空
160.    if (sa->sa_flags & SA_ONESHOT)
161.            sa->sa_handler = NULL;
        // 将该信号处理句柄替换内核中用户调用系统调用的代码指针
162.    *(&eip) = sa_handler;
        // 若允许信号自己的处理句柄收到信号自己，则将进程的阻塞码压入堆栈
163.    longs = (sa->sa_flags & SA_NOMASK)?7:8;
164.    *(&esp) -= longs;
165.    verify_area(esp,longs*4);                       // 检查内存使用情况
        // 将 sa_restorer，信号 signr，屏蔽码 blocked、eax、ecx、edx、eflags，用户程序源代码指针 old_eip
        // 在用户堆栈中依次存放
166.    tmp_esp=esp;
167.    put_fs_long((long) sa->sa_restorer,tmp_esp++);
168.    put_fs_long(signr,tmp_esp++);
169.    if (!(sa->sa_flags & SA_NOMASK))
170.            put_fs_long(current->blocked,tmp_esp++);
171.    put_fs_long(eax,tmp_esp++);
172.    put_fs_long(ecx,tmp_esp++);
173.    put_fs_long(edx,tmp_esp++);
174.    put_fs_long(eflags,tmp_esp++);
175.    put_fs_long(old_eip,tmp_esp++);
176.    current->blocked |= sa->sa_mask;
177.
        /* 强制在监督模式下的信号句柄的页面调入来减少竞争 */
178.    __asm__("testb $0,%%fs:%0"::"m" (*(char *) sa_handler));
```

```
179.        return(0);
180. }
```

该文件中 55～67 行为 signal()系统调用。与 sigaction()相似。为指定的信号安装新的信号句柄。信号句柄可以是用户指定的函数，也可以是 SIG_DFL（默认句柄）或 SIG_IGN（忽略）。函数返回原信号句柄。该文件中 68～84 行为 sigaction()系统调用。在收到一个信号时，改变当前进程的操作。signum 是除了 SIGKILL 以外的任何信号。如果新操作不为空，则新操作被安装。如果 oldaction 指针不为空，则原操作被保留到 oldaction。成功，返回 0，否则为-1。

5.6 exit.c 文件

exit.c 文件主要描述了进程终止和退出的处理事宜。主要包含进程释放、会话终止和程序退出处理函数以及杀死进程、终止进程、挂起进程等系统调用函数。

5.6.1 功能描述

该文件中的主要函数为 do_exit()，它是程序退出处理程序。在系统调用的中断处理程序中被调用。其他各函数的功能简单介绍如下：

send_sig()函数用于发送信号给指定任务。

release()函数用于释放除自身外的指定进程。

bad_task_ptr()函数主要用于检查一下任务结构指针是否存在于 task 数组中，如果找到了，返回 0，否则，返回 1。

audit_ptree()函数扫描进程树，仅用于调试。

session_of_pgrp()函数用于返回进程组的会话号，如果进程的 pid 号同指定的参数 pgrp 一样，也返回这单个进程的会话号。

kill_pg()函数用于向进程组发送信号，成功返回 0，否则返回错误号。

kill_proc()函数用来指定某个特定进程的操作，这个函数将会被 sys_kill 调用。

sys_kill()系统调用可用于向指定进程或进程组发送指定信号。

is_orphaned_pgrp()函数判断是否是孤儿进程组，如果不是返回 0，是返回 1。

has_stopped_jobs()函数检查进程组中是否有 TASK_STOPPED 标记的进程。如果有返回 1，否则返回 0。

sys_waitpid()用于挂起当前进程，直到 pid 指定的子进程退出（终止）或者收到要求终止该进程的信号，或者是需要调用一个信号句柄。

5.6.2 代码注释

exit.c 文件的详细代码注释如下：

```
1.  #define DEBUG_PROC_TREE
2.  #include <errno.h>          // 错误号头文件，包含系统中各种出错号
3.  #include <signal.h>         // 信号头文件，定义信号符号常量，信号结构以及信号操作函数原型
4.  #include <sys/wait.h>       // 等待调用头文件，定义系统调用 wait()及相关常数符号
5.  #include <linux/sched.h>
      // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据等
6.  #include <linux/kernel.h>   // 内核头文件，含有一些内核常用函数的原形定义
7.  #include <linux/tty.h>      // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数
8.  #include <asm/segment.h>    // 段操作头文件，定义有关段寄存器操作的嵌入式汇编函数
```

```
9.  int sys_close(int fd);
    //  向指定任务（*p）发送信号（sig），权限为 priv
10. inline int send_sig(long sig,struct task_struct * p,int priv)
11. {
12.         if (!p || (sig < 0) || (sig > 32))        // 若指针为空任务或者信号不正确，则出错退出
13.                 return -EINVAL;
    // 若无权并且进程有效用户标识符（euid）不是指定进程的 euid 并且不是超级用户，则出错退出
14.         if (!priv && (current->euid!=p->euid) && !suser())
15.                 return -EPERM;
    // 如果不是信号，返回 0
16.         if (!sig)
17.                 return 0;
    // 如果信号为 SIGKILL 或者 SIGCONT，则相应修改任务状态等信息
18.         if ((sig == SIGKILL) || (sig == SIGCONT)) {
19.                 if (p->state == TASK_STOPPED)
20.                         p->state = TASK_RUNNING;
21.                 p->exit_code = 0;
22.                 p->signal &= ~( (1<<(SIGSTOP-1)) | (1<<(SIGTSTP-1)) |
23.                                 (1<<(SIGTTIN-1)) | (1<<(SIGTTOU-1)) );
24.         }
25.         if ((sig >= SIGSTOP) && (sig <= SIGTTOU))
26.                 p->signal &= ~(1<<(SIGCONT-1));
27.
     /* 实际上是传送信号 */
28.         p->signal |= (1<<(sig-1));
29.         if (p->flags & PF_PTRACED) {
30.                 /* save the signal number for wait. */                    /* 保存信号，以便等待 */
31.                 p->exit_code = sig;
32.
33.                 /*必须确保父进程是就绪的*/
34.                 if (p->p_pptr != NULL && p->p_pptr->state == TASK_INTERRUPTIBLE)
35.                         p->p_pptr->state = TASK_RUNNING;
36.                 /*必须确保进程停止*/
37.                 if (p->state == TASK_INTERRUPTIBLE || p->state == TASK_RUNNING)
38.                         p->state = TASK_STOPPED;
39.                 if (p == current)                                  // 如果 P 为当前进程，重新调度
40.                         schedule();
41.         }
42.         return 0;
43. }
    //  释放指定进程（任务），但是任务不能释放任务本身
44. void release(struct task_struct * p)
45. {
46.         int i;
47.         if (!p)
48.                 return;
49.         if (p == current) {                          // 若 p 为当前任务，则返回
50.                 printk("task releasing itself\n\r");
51.                 return;
52.         }
```

```
53.         for (i=1 ; i<NR_TASKS ; i++)                    // 扫描任务数组，寻找指定任务
54.                 if (task[i]==p) {
55.                         task[i]=NULL;                   // 置空该任务项并释放相关内存页
56.                         /* 更新链接 */
57.                         if (p->p_osptr)
58.                                 p->p_osptr->p_ysptr = p->p_ysptr;
59.                         if (p->p_ysptr)
60.                                 p->p_ysptr->p_osptr = p->p_osptr;
61.                         else
62.                                 p->p_pptr->p_cptr = p->p_osptr;
63.                         free_page((long)p);
64.                         schedule();                     // 重新调度
65.                         return;
66.                 }
67.         panic("trying to release non-existent task");   // 指定任务若不存在则死机
68. }
69.  #ifdef DEBUG_PROC_TREE
    // 检查任务结构指针是否存在于 task 数组中，如果找到了，返回 0，否则，返回 1
70.  int bad_task_ptr(struct task_struct *p)
71.  {
72.          int     i;
73.          if (!p)
74.                  return 0;
75.          for (i=0 ; i<NR_TASKS ; i++)
76.                  if (task[i] == p)
77.                          return 0;
78.          return 1;
79.  }
80.  void audit_ptree()
81.  {
82.          int     i;
83.          for (i=1 ; i<NR_TASKS ; i++) {
84.                  if (!task[i])
85.                          continue;
86.                  if (bad_task_ptr(task[i]->p_pptr))
87.                          printk("Warning, pid %d's parent link is bad\n",
88.                                  task[i]->pid);
89.                  if (bad_task_ptr(task[i]->p_cptr))
90.                          printk("Warning, pid %d's child link is bad\n",
91.                                  task[i]->pid);
92.                  if (bad_task_ptr(task[i]->p_ysptr))
93.                          printk("Warning, pid %d's ys link is bad\n",
94.                                  task[i]->pid);
95.                  if (bad_task_ptr(task[i]->p_osptr))
96.                          printk("Warning, pid %d's os link is bad\n",
97.                                  task[i]->pid);
98.                  if (task[i]->p_pptr == task[i])
99.                          printk("Warning, pid %d parent link points to self\n");
100.                 if (task[i]->p_cptr == task[i])
101.                         printk("Warning, pid %d child link points to self\n");
```

```
102.            if (task[i]->p_ysptr == task[i])
103.                    printk("Warning, pid %d ys link points to self\n");
104.            if (task[i]->p_osptr == task[i])
105.                    printk("Warning, pid %d os link points to self\n");
106.            if (task[i]->p_osptr) {
107.                    if (task[i]->p_pptr != task[i]->p_osptr->p_pptr)
108.                            printk(
109.                    "Warning, pid %d older sibling %d parent is %d\n",
110.                            task[i]->pid, task[i]->p_osptr->pid,
111.                            task[i]->p_osptr->p_pptr->pid);
112.                    if (task[i]->p_osptr->p_ysptr != task[i])
113.                            printk(
114.            "Warning, pid %d older sibling %d has mismatched ys link\n",
115.                            task[i]->pid, task[i]->p_osptr->pid);
116.            }
117.            if (task[i]->p_ysptr) {
118.                    if (task[i]->p_pptr != task[i]->p_ysptr->p_pptr)
119.                            printk(
120.                    "Warning, pid %d younger sibling %d parent is %d\n",
121.                            task[i]->pid, task[i]->p_osptr->pid,
122.                            task[i]->p_osptr->p_pptr->pid);
123.                    if (task[i]->p_ysptr->p_osptr != task[i])
124.                            printk(
125.            "Warning, pid %d younger sibling %d has mismatched os link\n",
126.                            task[i]->pid, task[i]->p_ysptr->pid);
127.            }
128.            if (task[i]->p_cptr) {
129.                    if (task[i]->p_cptr->p_pptr != task[i])
130.                            printk(
131.                    "Warning, pid %d youngest child %d has mismatched parent link\n",
132.                            task[i]->pid, task[i]->p_cptr->pid);
133.                    if (task[i]->p_cptr->p_ysptr)
134.                            printk(
135.                    "Warning, pid %d youngest child %d has non-NULL ys link\n",
136.                            task[i]->pid, task[i]->p_cptr->pid);
137.            }
138.    }
139. }
140. #endif
    // 返回进程组的会话号，如果进程的 pid 号同指定参数 pgrp 一样，也返回这单个进程的会话号
141. int session_of_pgrp(int pgrp)
142. {
143.     struct task_struct **p;
144.            for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
145.            if ((*p)->pgrp == pgrp)
146.                    return((*p)->session);
147.     return -1;
148. }
    // 向进程组发送信号，成功返回 0，否则返回错误号，这些常常在终端中使用^C, ^Z 等来发送
149. int kill_pg(int pgrp, int sig, int priv)
```

```
150.    {
151.        struct task_struct **p;
152.        int err,retval = -ESRCH;
153.        int found = 0;
154.        if (sig<0 || sig>32 || pgrp<=0)
155.            return -EINVAL;
156.        for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
157.            if ((*p)->pgrp == pgrp) {
158.                if (sig && (err = send_sig(sig,*p,priv)))
159.                    retval = err;
160.                else
161.                    found++;
162.            }
163.        return(found ? 0 : retval);
164.    }
        // 针对于 pid>0 的用来指定某个特定进程的操作，这个函数将会被 sys_kill 调用
165.    int kill_proc(int pid, int sig, int priv)
166.    {
167.        struct task_struct **p;
168.        if (sig<0 || sig>32)
169.            return -EINVAL;
170.        for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
171.            if ((*p)->pid == pid)
172.                return(sig ? send_sig(sig,*p,priv) : 0);
173.        return(-ESRCH);
174.    }
175.    int sys_kill(int pid,int sig)
176.    {
177.        struct task_struct **p = NR_TASKS + task;
178.        int err, retval = 0;
179.        if (!pid)                                    // 如果 pid==0，就向同组的所有进程发送信号
180.            return(kill_pg(current->pid,sig,0));
181.        if (pid == -1) {                             // 向所有的进程发送信号，进程 1 除外
182.            while (--p > &FIRST_TASK)
183.                if (err = send_sig(sig,*p,0))
184.                    retval = err;
185.            return(retval);
186.        }
187.        if (pid < 0)                                 // 向指定的进程组发送信号
188.            return(kill_pg(-pid,sig,0));
189.
190.        return(kill_proc(pid,sig,0));                // pid>0，向指定的 pid 发送信号
191.    }
192.    int is_orphaned_pgrp(int pgrp)
193.    {
194.        struct task_struct **p;
195.        for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
196.            if (!(*p) ||
197.                ((*p)->pgrp != pgrp) ||
198.                ((*p)->state == TASK_ZOMBIE) ||
```

```
199.                    ((*p)->p_pptr->pid == 1))
200.                    continue;
201.            if (((*p)->p_pptr->pgrp != pgrp) &&
202.                    ((*p)->p_pptr->session == (*p)->session))
203.                    return 0;
204.    }
205.    return(1);
206. }
207.    // 检查进程组中是否有 TASK_STOPPED 标记的进程，如果有返回 1，否则返回 0
208. static int has_stopped_jobs(int pgrp)
209. {
210.     struct task_struct ** p;
211.    for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
212.            if ((*p)->pgrp != pgrp)
213.                    continue;
214.            if ((*p)->state == TASK_STOPPED)
215.                    return(1);
216.    }
217.    return(0);
218. }
219.    // 程序退出处理程序，在系统调用的中断处理程序中被调用
220. volatile void do_exit(long code)                    // code 是错误码
221. {
222.    struct task_struct *p;
223.    int i;
224.    // 释放当前进程代码段和数据段所占的内存页
225.    free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
226.    free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
    // 关闭当前进程打开的所有文件
227.    for (i=0 ; i<NR_OPEN ; i++)
228.            if (current->filp[i])
229.                    sys_close(i);
230.    iput(current->pwd);
231.    current->pwd = NULL;
232.    iput(current->root);
233.    current->root = NULL;
234.    iput(current->executable);
235.    current->executable = NULL;
236.    iput(current->library);
237.    current->library = NULL;
238.    current->state = TASK_ZOMBIE;
239.    current->exit_code = code;
240.    if ((current->p_pptr->pgrp != current->pgrp) &&
241.            (current->p_pptr->session == current->session) &&
242.            is_orphaned_pgrp(current->pgrp) &&
243.            has_stopped_jobs(current->pgrp)) {
244.             kill_pg(current->pgrp,SIGHUP,1);
245.             kill_pg(current->pgrp,SIGCONT,1);
246.    }
247.    /* 让父进程知道我们已死 */
```

```
248.        send_sig (SIGCHLD, current->p_pptr, 1);
249.        if (p = current->p_cptr) {
250.            while (1) {
251.                p->flags &= ~PF_PTRACED;
252.                p->p_pptr = task[1];
253.                if (p->state == TASK_ZOMBIE)
254.                    task[1]->signal |= (1<<(SIGCHLD-1));
255.                if ((p->pgrp != current->pgrp) &&
256.                    (p->session == current->session) &&
257.                    is_orphaned_pgrp(p->pgrp) &&
258.                    has_stopped_jobs(p->pgrp)) {
259.                    kill_pg(p->pgrp,SIGHUP,1);
260.                    kill_pg(p->pgrp,SIGCONT,1);
261.                }
262.                if (p->p_osptr) {
263.                    p = p->p_osptr;
264.                    continue;
265.                }
266.
267.
268.
        /* 把所有的都链接为 init 进程的孩子并且退出*/
269.                p->p_osptr = task[1]->p_cptr;
270.                task[1]->p_cptr->p_ysptr = p;
271.                task[1]->p_cptr = current->p_cptr;
272.                current->p_cptr = 0;
273.                break;
274.            }
275.        }
276.        if (current->leader) {                          // leader 为进程组首领
277.            struct task_struct **p;
278.            struct tty_struct *tty;
279.            if (current->tty >= 0) {                    // 如果当前进程有控制的终端，则释放该终端
280.                tty = TTY_TABLE(current->tty);
281.                if (tty->pgrp>0)
282.                    kill_pg(tty->pgrp, SIGHUP, 1);
283.                tty->pgrp = 0;
284.                tty->session = 0;
285.            }
286.            for (p = &LAST_TASK ; p > &FIRST_TASK ; --p)
287.                if ((*p)->session == current->session)
288.                    (*p)->tty = -1;
289.        }
290.        if (last_task_used_math == current)
291.            last_task_used_math = NULL;                 // 进程结束，不保存协处理器信息了
292.    #ifdef DEBUG_PROC_TREE
293.        audit_ptree();                                  // 调试
294.    #endif
295.        schedule();                                     // 重新调度进程的运行
296.    }
```

```
297.    int sys_exit(int error_code)                    // 系统调用 exit()，终止进程
298.    {
299.        do_exit((error_code&0xff)<<8);
300.    }
301.    int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
302.    {
303.        int flag;
304.        struct task_struct *p;
305.        unsigned long oldblocked;
306.        if (stat_addr)                              // 用来保存终止进程的状态
307.            verify_area(stat_addr,4);
308.    repeat:
309.        flag=0;
310.        for (p = current->p_cptr ; p ; p = p->p_osptr) {
311.            if (pid>0) {                            // 指定进程
312.                if (p->pid != pid)
313.                    continue;
314.            } else if (!pid) {                      // 当前进程组
315.                if (p->pgrp != current->pgrp)
316.                    continue;
317.            } else if (pid != -1) {                 // 指定进程组
318.                if (p->pgrp != -pid)
319.                    continue;
320.            }
321.            switch (p->state) {
322.                case TASK_STOPPED:                  // 进程被挂起
323.                    if (!(options & WUNTRACED) ||   // 如果不被追踪
324.                        !p->exit_code)              // 或者收到了停止类的信号
325.                        continue;
326.                    if (stat_addr)                  // 拷贝信息
327.                        put_fs_long((p->exit_code << 8) | 0x7f,
328.                            stat_addr);
329.                    p->exit_code = 0;               // 清除退出代码
330.                    return p->pid;
331.                case TASK_ZOMBIE:                   //进程僵死，时间统计并加入到当前进程中
332.                    current->cutime += p->utime;
333.                    current->cstime += p->stime;
334.                    flag = p->pid;
335.                    if (stat_addr)
336.                        put_fs_long(p->exit_code, stat_addr);
337.                    release(p);                     // 释放资源
338.    #ifdef DEBUG_PROC_TREE
339.                    audit_ptree();
340.    #endif
341.                    return flag;
342.                default:
343.                    flag=1;
344.                    continue;
345.            }
346.        }
```

```
347.        if (flag) {                    // 有满足条件的进程，但是进程的状态不是停止或者僵死的状态时
348.            if (options & WNOHANG)
349.                return 0;                          // 如果没有子进程处于退出或者停止状态，就立即返回
350.            current->state=TASK_INTERRUPTIBLE;
351.            oldblocked = current->blocked;
352.            current->blocked &= ~(1<<(SIGCHLD-1));
353.            schedule();
354.            current->blocked = oldblocked;
355.            if (current->signal & ~(current->blocked | (1<<(SIGCHLD-1))))
356.                return -ERESTARTSYS;
357.            else
358.                goto repeat;        // 如果该进程再次运行，说明是被别的信号唤醒，则重新扫描
359.        }
360.        return -ECHILD;
361.    }
```

该文件中 80～139 行的作用为检查进程树。仅用于调试。该文件中 192～206 行的作用为决定一个进程组是不是孤儿进程组，通过 POSIX 的定义。孤儿进程组不会受到终端产生的停止信号的影响。新产生的孤儿进程组将收到一个 SIGHUP 或者 SIGCONT。判断是否是孤儿进程组，如果不是返回 0，是返回 1。该文件中 301～361 行为系统调用 waitpid()。挂起当前进程，直到 pid 指定的子进程退出（终止）或者收到要求终止该进程的信号，或者是需要调用一个信号句柄。如果 pid 所指的子进程早已退出，则本调用将立刻返回。子进程使用的所有资源将释放。

5.7 vsprintf.c 文件

该程序主要描述了如何产生格式化输出的方法定义。作为通用函数的定义，可以在内核中所有需要输出字符串的情况下被调用。该文件主要包括 vsprintf()函数，该函数用于对参数产生格式化输出。该函数的作用类似于与 C 语言 stdio.h 中定义的 printf()函数。

vsprintf.c 文件的详细代码注释如下：

```
1.      #include <stdarg.h>                    // 标准参数头文件，以宏的形式定义变量参数列表
2.      #include <string.h>                    // 字符串头文件，主要定义了一些有关字符串操作的函数
3.      #define is_digit(c)    ((c) >= '0' && (c) <= '9')      // 判断字符是否是数字字符
4.      // 该函数将字符数字串转换成整数。输入是数字串指针的指针，返回是结果数值
5.      static int skip_atoi(const char **s)
6.      {
7.          int i=0;
8.          while (is_digit(**s))
9.              i = i*10 + *((*s)++) - '0';
10.         return i;
11.     }
        // 这里定义转换类型的各种符号常数。
12.     #define ZEROPAD   1                    /* 填充零*/
13.     #define SIGN   2                       /* 无符号/符号长整数*/
14.     #define PLUS   4                       /* 显示加*/
15.     #define SPACE8                         /* 如果是加，则置空格*/
16.     #define LEFT   16                      /* 左调整*/
17.     #define SPECIAL    32                  /* 0x */
18.     #define SMALL      64
```

```
        /* 使用小写字母 */
19.     // 除操作。输入：n 为被除数，base 为除数；结果：n 为商，函数返回值为余数
20.     #define do_div(n,base) ({
21.     int __res;
22.     __asm__("divl %4":"=a" (n),"=d" (__res):"0" (n),"1" (0),"r" (base));
23.     __res; })
24.     static char * number(char * str, int num, int base, int size, int precision
25.         ,int type)
26.     {
27.         char c,sign,tmp[36];
28.         const char *digits="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
29.         int i;
30.         if (type&SMALL) digits="0123456789abcdefghijklmnopqrstuvwxyz";
31.         if (type&LEFT) type &= ~ZEROPAD;
32.         if (base<2 || base>36)
33.             return 0;
34.         c = (type & ZEROPAD) ? '0' : ' ';
35.         if (type&SIGN && num<0) {
36.             sign='-';
37.             num = -num;
38.         } else
39.             sign=(type&PLUS) ? '+' : ((type&SPACE) ? ' ' : 0);
        // 若带符号，则宽度值减 1。若类型指出是特殊转换，则对于十六进制宽度再减 2 位，
        // 对于八进制宽度减 1
40.         if (sign) size--;
41.         if (type&SPECIAL)
42.             if (base==16) size -= 2;
43.             else if (base==8) size--;
        // 如果数值 num 为 0，则临时字符串为'0'；否则根据给定的基数将数值 num 转换为字符形式
44.         i=0;
45.         if (num==0)
46.             tmp[i++]='0';
47.         else while (num!=0)
48.             tmp[i++]=digits[do_div(num,base)];
        // 若数值字符个数大于精度值，则精度值扩展为数字个数值
        // 宽度值 size 减去用于存放数值字符的个数
49.         if (i>precision) precision=i;
50.         size -= precision;
51.         if (!(type&(ZEROPAD+LEFT)))
52.             while(size-->0)
53.                 *str++ = ' ';
54.         if (sign)
55.             *str++ = sign;
        // 若类型指出是特殊转换，则对于八进制转换结果头一位放置一个'0'；而十六进制则存放'0x'
56.         if (type&SPECIAL)
57.             if (base==8)
58.                 *str++ = '0';
59.             else if (base==16) {
60.                 *str++ = '0';
61.                 *str++ = digits[33];
```

```
62.                 }
        // 若类型中没有左调整（左靠齐）标志，则在剩余宽度中存放 c 字符（'0'或空格）
63.         if (!(type&LEFT))
64.                 while(size-->0)
65.                         *str++ = c;
        // 此时 i 存有数值 num 的数字个数。若数字个数小于精度值，则 str 中放入精度值-i 个'0'
66.         while(i<precision--)
67.                 *str++ = '0';
        // 将转换好的数字字符填入 str 中
68.         while(i-->0)
69.                 *str++ = tmp[i];
        // 若宽度值仍大于零，则表示类型标志中有左靠齐标志，则在剩余宽度中放入空格
70.         while(size-->0)
71.                 *str++ = ' ';
72.         return str;
73.     }
        // 下面函数是送格式化输出到字符串中。Linus 在内核中实现了使用格式化的输出的函数
        // 其中参数 fmt 是格式字符串；args 是个数变化的值；buf 是输出字符串缓冲区
74.     int vsprintf(char *buf, const char *fmt, va_list args)
75.     {
76.         int len;
77.         int i;
78.         char * str;
79.         char *s;
80.         int *ip;
81.         int flags;          /* number()函数使用的标志 */
82.
83.         int field_width;  /* 输出字段宽度*/
84.         int precision;      /* min 整数数字个数；max 字符串中字符个数 */
85.
86.         int qualifier;      /* 'h', 'l',或'L'用于整数字段 */
87.     for (str=buf ; *fmt ; ++fmt) {
88.                 if (*fmt != '%') {
89.                         *str++ = *fmt;
90.                         continue;
91.                 }
        // 下面取得格式指示字符串中的标志域，并将标志常量放入 flags 变量中
92.
93.                 flags = 0;
94.                 repeat:
95.                         ++fmt;
96.                         switch (*fmt) {
97.                                 case '-': flags |= LEFT; goto repeat;
98.                                 case '+': flags |= PLUS; goto repeat;
99.                                 case ' ': flags |= SPACE; goto repeat;
100.                                case '#': flags |= SPECIAL; goto repeat;
101.                                case '0': flags |= ZEROPAD; goto repeat;
102.                                }
103.
104.                field_width = -1;
```

```
105.            if (is_digit(*fmt))
106.                    field_width = skip_atoi(&fmt);
107.            else if (*fmt == '*') {
108.                    /* it's the next argument */
109.                    field_width = va_arg(args, int);
110.                    if (field_width < 0) {
111.                            field_width = -field_width;
112.                            flags |= LEFT;
113.                    }
114.            }
115.
116.            precision = -1;
117.            if (*fmt == '.') {
118.                    ++fmt;
119.                    if (is_digit(*fmt))
120.                            precision = skip_atoi(&fmt);
121.                    else if (*fmt == '*') {
122.
123.                            precision = va_arg(args, int);
124.                    }
125.                    if (precision < 0)
126.                            precision = 0;
127.            }
        // 下面这段代码分析长度修饰符，并将其存入 qualifer 变量
128.
129.            qualifier = -1;
130.            if (*fmt == 'h' || *fmt == 'l' || *fmt == 'L') {
131.                    qualifier = *fmt;
132.                    ++fmt;
133.            }
        // 下面分析转换指示符
134.            switch (*fmt) {
135.            case 'c':
136.                    if (!(flags & LEFT))
137.                            while (--field_width > 0)
138.                                    *str++ = ' ';
139.                    *str++ = (unsigned char) va_arg(args, int);
140.                    while (--field_width > 0)
141.                            *str++ = ' ';
142.                    break;
143.            case 's':
144.                    s = va_arg(args, char *);
145.                    len = strlen(s);
146.                    if (precision < 0)
147.                            precision = len;
148.                    else if (len > precision)
149.                            len = precision;
150.                    if (!(flags & LEFT))
151.                            while (len < field_width--)
152.                                    *str++ = ' ';
```

```
153.                    for (i = 0; i < len; ++i)
154.                        *str++ = *s++;
155.                    while (len < field_width--)
156.                        *str++ = ' ';
157.                    break;
        // 如果是'o'，表示需将对应的参数转换成八进制数的字符串。调用 number()函数处理
158.            case 'o':
159.                    str = number(str, va_arg(args, unsigned long), 8,
160.                        field_width, precision, flags);
161.                    break;
162.    // 如果是'p'，表示对应参数的一个指针类型。此时若该参数没有设置宽度域，则默认宽度为 8，
        // 并且需要添零。然后调用 number()函数进行处理
163.            case 'p':
164.                    if (field_width == -1) {
165.                        field_width = 8;
166.                        flags |= ZEROPAD;
167.                    }
168.                    str = number(str,
169.                        (unsigned long) va_arg(args, void *), 16,
170.                        field_width, precision, flags);
171.                    break;
172.    // 若是'x'或'X'，则表示对应参数需要打印成十六进制数输出。'x'表示用小写字母表示
173.            case 'x':
174.                    flags |= SMALL;
175.            case 'X':
176.                    str = number(str, va_arg(args, unsigned long), 16,
177.                        field_width, precision, flags);
178.                    break;
        // 如果是'd','i'或'u'，则表示对应参数是整数，'d', 'i'代表符号整数，因此需要加上
        // 带符号标志。'u'代表无符号整数
179.            case 'd':
180.            case 'i':
181.                    flags |= SIGN;
182.            case 'u':
183.                    str = number(str, va_arg(args, unsigned long), 10,
184.                        field_width, precision, flags);
185.                    break;
186.    // 若是'n'，则表示要把到目前为止转换输出的字符数保存到对应参数指针指定的位置中
        // 首先利用 va_arg()取得该参数指针，然后将已经转换好的字符数存入该指针所指的位置
187.            case 'n':
188.                    ip = va_arg(args, int *);
189.                    *ip = (str - buf);
190.                    break;
191.            default:
192.                    if (*fmt != '%')
193.                        *str++ = '%';
194.                    if (*fmt)
195.                        *str++ = *fmt;
196.                    else
197.                        --fmt;
```

```
198.                    break;
199.                }
200.        }
201.        *str = '\0';                    // 最后在转换好的字符串结尾处添上 null
202.        return str-buf;                 // 返回转换好的字符串长度值
203.    }
```

该文件中第 24 行开始的 char * number 函数的作用是将整数转换为指定进制的字符串。输入为 num-整数，base-进制，size-字符串长度，precision-数字长度（精度），type-类型选项。输出为 str 字符串指针。该文件中从第 50 行开始是真正开始形成所需要的转换结果，并暂时放在字符串 str 中。若类型中没有填零（ZEROPAD）和左靠齐（左调整）标志，则在 str 中首先填放剩余宽度值指出的空格数。若需带符号位，则存入符号。

文件中 83～91 行为：取当前参数字段宽度域值，放入 field_width 中。如果宽度域中是数值，则直接取其为宽度值。如果宽度域中是字符“*”，表示下一个参数指定宽度。因此调用 va_arg 取宽度值。若此时宽度值小于 0，则该负数表示其带有标志域“-”标志（左靠齐），因此还需在标志变量中添入该标志，并将字段宽度值取为其绝对值。

文件中 116～127 行为取格式转换串的精度域，并放入 precision 变量中。精度域开始的标志是“.”。其处理过程与上面宽度域的类似。如果精度域中是数值则直接取其为精度值。如果精度域中是字符“*”，表示下一个参数指定精度。因此调用 va_arg 取精度值。若此时宽度值小于 0，则将字段精度值取为其绝对值。

5.8 printk.c 文件

printk.c 文件主要包含了 printk()方法，该方法为 Linux 内核中定义的用于显示的方法，可以作为 Linux 提供的标准库函数使用。

printk()是内核中使用的显示函数，功能与 C 标准函数库中的 print()相同。重新编写这样一个函数的原因是在内核中不能使用专用于用户模式的 fs 段寄存器，需要首先保存它。在保存 fs 的前提下定义一个 printf()函数，这样所有问题就解决了。printk.c 文件的详细代码注释如下：

```
1.      #include <stdarg.h>
        // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了一个类型（va_list）
        // 和三个宏（va_start, va_arg 和 va_end）
2.      #include <stddef.h>                     // 标准定义头文件
3.      #include <linux/kernel.h>               // 内核头文件。含有一些内核常用函数的原形定义
4.      static char buf[1024];
        // 下面函数是送格式化输出到字符串中
5.      extern int vsprintf(char * buf, const char * fmt, va_list args);
6.      // 内核使用的显示函数
7.      int printk(const char *fmt, ...)
8.      {
9.          va_list args;                       // va_list 实际上是一个字符指针类型
10.         int i;
11.         va_start(args, fmt);                // 参数处理开始函数
12.         i=vsprintf(buf,fmt,args);           // 使用格式串 fmt 将参数列表 args 输出到 buf 中
13.         va_end(args);                       // 参数处理结束函数
14.         console_print(buf);                 // 控制台显示函数，仅用于内核显示函数
```

```
15.         return i;                          // 返回字符串长度 i
16.     }
```

5.9 panic.c 文件

panic.c 文件主要包含了 panic()方法，该方法用于显示内核中出现的重大错误信息。当系统发现无法继续运行下去的故障时将调用它，这将会导致程序中止，然后由系统显示错误号。

panic()函数用来显示内核中出现的重大错误信息。当内核程序出错时，调用函数 panic()，显示错误信息并使系统进入死循环。在内核中，有许多地方都调用了这个函数，包括头文件、内存管理 mm 和文件系统 fs 等，用于指出主要的问题。panic.c 文件的详细代码注释如下：

```
1.    #include <linux/kernel.h>                  // 内核头文件。含有内核常用函数的原形定义
2.    #include <linux/sched.h>                   // 调度程序头文件。定义了任务结构 task_struct 等
3.    void sys_sync(void);                       /* 实际上是整型 */
4.    volatile void panic(const char * s)
5.    {
6.        printk("Kernel panic: %s\n\r",s);
7.        if (current == task[0])
8.            printk("In swapper task - not syncing\n\r");
9.        else
10.           sys_sync();
11.       for(;;);
12.   }
```

第 6 章　字符设备驱动程序

本章主要讨论字符设备驱动程序。0.95 版的 Linux 内核中字符设备主要包括控制终端设备和串行终端设备。所涉及的源代码文件如表 6-1 所示。

表 6-1　linux/kernel/chr_drv 目录

名　　称	大小（B）	修 改 时 间
console.c	28 000	1992-03-05 23:03:52
keyboard.S	16 896	1992-03-05 08:58:08
pty.c	1186	1992-01-11 07:56:46
rs_io.s	2763	1992-03-05 08:57:26
serial.c	2633	1992-03-04 00:50:36
tty_io.c	12 065	1992-03-04 19:22:40
tty_ioctl.c	8568	1992-03-08 19:18:38

6.1　总体功能

本章的程序主要分成四部分。第一部分是关于 RS-232 串行线路驱动程序，包括程序 rs_io.s 和 serial.c；第二部分是涉及控制台驱动程序，这包括键盘中断驱动程序 keyboard.s 和控制台显示驱动程序 console.c；第三部分是终端驱动程序与上层接口部分，包括终端输入输出程序 tty_io.c 和终端控制程序 tty_ioctl.c。第四部分是伪终端设备驱动程序，包括 pty.c。

6.1.1　Linux 支持的终端设备类型

终端是一种字符型设备，它有多种类型。通常使用 tty 来简称各种类型的终端设备。tty 是 Teletype 的缩写，Teletype 是一种由 Teletype 公司生产的最早出现的终端设备，外观很像电传打字机。这些终端设备文件可以分为以下几种类型：

1. 串行端口终端（/dev/ttySn）

串行端口终端是使用计算机串行端口连接的终端设备。计算机把每个串行端口都看作是一个字符设备。有段时间这些串行端口设备通常被称为终端设备，因为那时它的最大用途就是用来连接终端。这些串行端口所对应的设备文件名是/dev/ttyS0、/dev/ttySl 等，设备号分别是（4，64）、（4，65）等，分别对应于 DOS 系统下的 COMl、COM2。若要向一个端口发送数据，只要在命令行上把标准输出重定向到这些特殊文件名上即可。例如，在命令行提示符下输人：echo test>/dev/ttySl，就会把单词“test”发送到连接在 ttySl 端口的设备上。

2. 伪终端(/ 'dev / ptyp、 / 'din / ttyp)

伪终端（Pseudo Terminals，或 Pseudo-TTY，简称为 PTY）是一种功能类似于一般终端的设备，但是这种设备并不与任何终端硬件相关。伪终端设备用于为其他程序提供类似于终端式样的接口，主要应用于通过网络登录主机时为网络服务器和登录 slaell 程序之前提供一个终端接口，或为运行于 x Window 窗口中的终端程序提供终端样式的接口。当然，我们也可以利用伪终端在任何两个使用终端接口的程序之间建立数据读写通道。为了分别为两个应用程序或进程提供终端

样式接口，伪终端均配对使用。一个称为主伪终端（Master PTY）或伪终端主设备，另一个称为从伪终端（Slave PTY）或伪终端从设备。对于像 ptypl 和 ttypl 这样成对的伪终端逻辑设备来讲，ptypl 是主设备或者是控制终端，而 ttypl 则是从设备。往其中任意一个伪终端写入的数据会通过内核直接由配对的伪终端接收到。例如对于主设备/devr/ptyp3 和从设备/dev/ttyp3，如果一个程序把 ttyp3 看作是一个串行端口设备，则它对该端口的读写操作会反映在对应的另一个逻辑终端设备 ptyp3 上面。而 ptyp3 则会是另一个程序用于读写操作的逻辑设备。这样，两个程序就可以通过过这种逻辑设备进行互相交流，而其中一个使用从设备 ttyp3 的程序则认为自己正在与一个串行端口进行通信。这很像是逻辑设备对之间的管道操作。对于伪终端从设备，任何一个设计成使用串行端口设备的程序都可以使用该逻辑设备。但对于使用主设备的程序来讲，就需要专门设计来使用伪终端主设备。

例如，如果某人在网上使用 telnet 程序连接到你的计算机，那么 telnet 程序就可能会开始连接到伪终端主设备 ptyp2 上。此时一个 getty 程序就应该运行在对应的 ttyp2 端口上。当 telnet 从远端获取了一个字符时，该字符就会通过 ptyp2、ttyp2 传递给 getty 程序，而 getty 程序则会通过 ttyp2、ptyp2 和 telnet 程序往网络上送出“login：”字符串信息。这样，登录程序与 telnet 程序就通过“伪终端”进行通信。通过使用适当的软件，我们就可以把两个甚至多个伪终端设备连接到同一个物理端口上。

以前的 Linux 系统最多只有 16 个成对的 ttyp（ttyp0～ttypf）设备文件名。但现在的 Linux 系统上通常都使用“主伪终端（ptm—pty master）”命名方式，例如/dev/ptrn3。它的对应端则会被自动创建成/dev/pts/3。这样就可以在需要时动态提供一个 pty 伪终端。现在的 Linux 系统上目录/dev/pts 是一个 devpts 类型的文件系统。虽然“文件”/dev/pts/3 看上去是设备文件系统中的一项，但其实它完全是一种不同的文件系统。

3. 控制终端（/dev/tty）

字符设备文件/dev/tty 是进程控制终端（Controlling Terminal）的别名，其主设备号是 5，次设备号是 0。如果当前进程有控制终端，那么/dev/tty 就是当前进程控制终端的设备文件。我们可以使用命令“ps-ax”来查看进程与哪个控制终端相连。对于登录 shell 来讲，/dev/tty 就是我们使用的终端，其设备号是（5，0）。我们可以使用命令“tty”来查看它具体对应哪个实际终端设备。实际上/dev/tty 有些类似于连接到实际终端设备的一个链接。

如果一个终端用户执行了一个程序，但不想要控制终端（例如一个后台服务器程序），那么进程可以先试着打开/dev/tty 文件。如果打开成功，则说明进程有控制终端。此时可以使用 TIOCNOTTY(Terminal IO Control NO TTY)参数的 ioctl()调用来放弃控制终端。

4. 控制台（/dev/ttyn，/dev/console）

在 Linux 系统中，计算机显示器通常被称为控制台终端或控制台（Console）。它模仿了 VT200 或 Linux 类型终端（TERM=Linux），并且有一些字符设备文件与之关联：tty0、ttyl、tty2 等。当在控制台上登录时，使用的就是 ttyl。另外，使用【Alt+F1】～【Alt+F6】组合键就可以切换到 tt2、tty3 等上面去。ttyl～tty6 被称为虚拟终端，而 tty0 则是当前所使用虚拟终端的一个别名。Linux 系统所产生的信息都会发送到 tty0 上。因此不管当前正在使用哪个虚拟终端，系统信息都会发送到屏幕上。

可以登录到不同的虚拟终端上去，因而可以让系统同时有几个不同的会话存在。但只有系统或超级用户 root 可以向/dev/tty0 执行写操作。而且有时/dev/console 也会连接至该终端设备上。

5. 其他类型

现在的 Linux 系统中还针对很多不同的字符设备建有很多其他种类的终端设备特殊文件。例如针对 ISIDN 设备的/dev/ttyIn 终端设备等。这里不再赘述。

6.1.2 Linux 字符设备驱动程序

字符设备是指那些无需缓冲区可直接读写的设备，如鼠标、键盘、串口设备等，对它们读写数据直接通过读写寄存器就可实现。字符终端（/dev/console）和串口（/dev/ttyS0 以及类似设备）就是字符设备的两个例子，它们能够用流抽象很好地表示。字符设备是能够像字节流（类似文件）一样被访问的设备，由字符设备驱动程序来实现这种特性。字符设备通常可以直接读写，没有缓冲区。字符驱动程序是以单个字符为数据访问的最小单位。字符驱动程序的结构要比块设备驱动程序的结构简单得多，也容易理解。因此可以首先以字符驱动程序入手来分析驱动程序的最基本的结构。

Linux 系统是靠主次设备号来联系驱动程序和设备文件结点的。系统依靠主设备号标识不同的驱动程序，因此，在同一个系统中，一类设备的主设备号是惟一的。同一个驱动程序可以管理多个设备，它们依靠次设备号来区别。次设备号只在驱动程序内部使用，系统内核直接把次设备号传递给驱动程序，由驱动程序管理。正如前面所说，Linux 下的字符设备正是由一个主设备号和一个次设备号来标识。主设备号标识设备对应的驱动程序，内核利用主设备号将设备与相应的驱动程序对应起来。次设备号只由设备驱动程序使用，内核的其他部分不使用它，仅将它传递给驱动程序。

字符设备驱动程序是 Linux 系统最基本、最常用的驱动程序结构。0.95 版内核中字符设备主要包括控制终端设备和串行终端设备，有关终端驱动程序的工作原理读者可以参考相关书籍。

6.1.3 字符设备驱动程序功能

0.95 版的 Linux 内核字符设备程序代码划分为四部分： RS-232 串行驱动程序，包括 rs_io.s 和 serial.c；控制台驱动程序，包括 keyboard.s 和 console.c；控制台驱动程序和终端驱动程序与上层接口部分，包括 tty_io.c 和 tty_ioctl.c；伪终端设备程序 pty.c。

伪终端设备程序 pty.c，主要用于完成伪终端设备的读写操作，主要包括主伪终端的写函数 mpty_write()、从伪终端的写函数 spty_write()。

串口程序 serial.c，主要是串行设备 boot-time 初始化部分，此部分主要包括了 7 个函数，主要是：串口初始设置函数 early_serical_setup，串口控制台写函数 serial_console_ write 和汇编程序 rs_io.s 等，完成主要功能是实现 RS-232 串行通信中断处理过程。主要是根据引发中断的 4 种原因分别进行处理。

键盘中断处理程序 keyboard.s 用于处理用户输入的字符，把它们放入读缓冲队列 read_q 中，并调用 copy_to_cooked()函数读取 read_q 中的字符，经转换后放入辅助缓冲队列 secon dary。

屏幕显示处理程序 console.c 是内核中最长的程序之一，但是实现的功能单一，都是为了实现终端屏幕写函数 con_write()和终端初始化工作。主要包括键盘中断处理函数 do_keyboard_ interrupt()、终端屏幕初始化函数 con_init()、终端屏幕写函数 con_write() 等。

tty_io.c 程序包括字符设备的上层接口函数。主要含有终端读/写函数 tty_read()和 tty_write()。

tty_ioctl.c 实现了函数（系统调用）tty_ioctl()。程序通过使用该函数可以修改指定终端 termios 结构中的设置标志等信息。

6.2 重要数据结构

本节主要内容为介绍与字符设备驱动相关的数据结构，对各个数据结构的代码进行分析。本节介绍的数据结构列表如表 6-2 所示：

表 6-2 进程相关数据结构列表

名　　称	功能简单描述
tty_struct	终端的数据结构体
tty_queue	字符缓冲队列结构体
termios	终端 io 属性和控制字符结构体

1. tty_struct

tty 是 TeleTYpe 的一个缩写，为用户输入提供不同控制台的设备驱动程序。tty_struct 结构体里面存放着关于这个 tty 的所有信息，该结构体中定义了通用或高层 tty 的属性（例如宽度和高度等）。它被 tty 用来保持当前的特定 tty 端口的状态，被几乎所有的 tty 中的文件所使用。

```
struct tty_struct {
    struct termios termios;                    // 终端 IO 属性和控制字符数据结构
    int pgrp;                                  // 所属进程组
    int session;                               // 时间间隔
    int stopped;                               // 停止标志
    struct winsize winsize;                    // 窗口大小（Window size）属性
    void (*write)(struct tty_struct * tty);    // tty 写函数指针
    struct tty_queue *read_q;                  // tty 读队列
    struct tty_queue *write_q;                 // tty 写队列
    struct tty_queue *secondary;               // tty 辅助队列
};
extern struct tty_struct tty_table[];          // tty 结构数组
```

2. tty_queue

tty_queue 结构体表示字符缓冲队列。所有出去等待状态的字符设备文件都将被存放在系统中的该队列里面。

```
struct tty_queue {
    unsigned long data;                        // 等待队列缓冲区中当前字符行数
    unsigned long head;                        // 缓冲区中数据头指针
    unsigned long tail;                        // 缓冲区中数据尾指针
    struct task_struct * proc_list;            // 等待进程列表
    char buf[TTY_BUF_SIZE];                    // 队列缓冲区
};
```

3. termios

termios 结构体表示一个常规的字符终端接口，用于控制非同步通信端口。

```
struct termios {
    tcflag_t c_iflag;                          // 输入模式标志
    tcflag_t c_oflag;                          // 输出模式标志
    tcflag_t c_cflag;                          // 控制模式标志
    tcflag_t c_lflag;                          // 本地模式标志
    cc_t c_line;                               // 线路规程
    cc_t c_cc[NCCS];                           // 控制字符数组
};
```

6.3 console.c 文件

该文件主要实现了一个控制台的功能，在其中主要定义了用于进行终端初始化操作和屏幕的写操作的函数。

本程序中的所有子程序都是为了实现终端屏幕写函数 con_write()以及进行终端初始化操作。函数 con_write()是整个程序中非常重要的函数，用于对控制台屏幕进行写操作。它是根据系统初始化时获得信息来设置屏幕相关的一些参数。首先它从写缓冲队列 tty 中取字符，然后根据字符的状态性质，把字符显示在屏幕上或进行光标移动等操作。

console.c 文件的详细代码注释如下：

```
1.      #include <linux/sched.h>                    //调度程序头文件
2.        #include <linux/timer.h>                  //定时器头文件
3.        #include <linux/tty.h>                    // tty 头文件
4.      #include <linux/config.h>                   //内核配置头文件
5.      #include <linux/kernel.h>                   //内核头文件
6.        #include <asm/io.h>                       // io 头文件
7.        #include <asm/system.h>                   //系统头文件
8.      #include <asm/segment.h>                    //段操作头文件
9.      #include <string.h>                         //字符串头文件
10.       #include <errno.h>                        //错误号头文件
11.       #define DEF_TERMIOS
12.       (struct termios) {
13.     ICRNL,
14.     OPOST | ONLCR,
15.       0,
16.       IXON | ISIG | ICANON | ECHO | ECHOCTL | ECHOKE,
17.       0,
18.       INIT_C_CC
19.     }
20.     static void blank_screen(void);
21.       static void unblank_screen(void);
      //这些是设置子程序 setup 在引导启动系统时设置的参数
22.     #define ORIG_X                  (*(unsigned char *)0x90000)                 // 光标列号
23.     #define ORIG_Y                  (*(unsigned char *)0x90001)                 // 光标行号
24.       #define ORIG_VIDEO_PAGE           (*(unsigned short *)0x90004)      // 显示页面
25.     #define ORIG_VIDEO_MODE           ((*(unsigned short *)0x90006) & 0xff)   // 显示模式
26.     #define ORIG_VIDEO_COLS (((*(unsigned short *)0x90006) & 0xff00) >> 8) // 字符列数
27.       #define ORIG_VIDEO_LINES    ((*(unsigned short *)0x9000e) & 0xff)   // 显示行数
28.     #define ORIG_VIDEO_EGA_AX   (*(unsigned short *)0x90008)
29.     #define ORIG_VIDEO_EGA_BX   (*(unsigned short *)0x9000a)// 显示内存大小和色彩模式
30.       #define ORIG_VIDEO_EGA_CX       (*(unsigned short *)0x9000c)      // 显卡特性参数
    // 以下定义显示器单色/彩色显示模式类型符号常数
31.     #define VIDEO_TYPE_MDA     0x10                                         /* 单色文本 */
32.     #define VIDEO_TYPE_CGA       0x11                                       /* CGA 显示器 */
33.      #define VIDEO_TYPE_EGAM 0x20                                          /* EGA/VGA 单色 */
34.     #define VIDEO_TYPE_EGAC  0x21                                          /* EGA/VGA 彩色 */
35.     #define NPAR 16
36.       int NR_CONSOLES = 0;
37.       extern void keyboard_interrupt(void);                 // 键盘中断处理程序
```

```
38.     extern void set_leds(void);
39.     extern unsigned char kapplic;
40.      extern unsigned char kleds;
41.      extern unsigned char kmode;
42.      unsigned long    video_num_columns;
43.     unsigned long    video_num_lines;
44.     static unsigned char    video_type;                /* 使用的显示类型 */
45.     static unsigned long    video_mem_base;            /* 显示内存基地址 */
46.     static unsigned long    video_mem_term;            /* 显示内存结束地址 */
47.     static unsigned long    video_size_row;            /* 每行使用的字节数 */
48.     static unsigned char    video_page;                /* 初始显示页面 */
49.     static unsigned short   video_port_reg;            /* 显示控制索引寄存器端口 */
50.     static unsigned short   video_port_val;            /* 显示控制数据寄存器端口 */
51.     static int can_do_colour = 0;
52.     // 虚拟控制台结构体
53.      static struct {
54.       unsigned short  vc_video_erase_char;
55.       unsigned char   vc_attr;
56.       unsigned char   vc_def_attr;
57.      int vc_bold_attr;
58.       unsigned long   vc_ques;
59.       unsigned long   vc_state;
60.      char * vc_restate;
61.       unsigned long   vc_checkin;
62.       unsigned long   vc_origin;                       // 滚屏起始内存地址
63.       unsigned long   vc_scr_end;                      // 滚屏末端内存地址
64.       unsigned long   vc_pos;                          // 当前光标对应的显示内存位置
65.       unsigned long   vc_x,vc_y;                       // 当前光标位置
66.       unsigned long   vc_top,vc_bottom;                // 滚动时顶行行号；底行行号
67.       unsigned long   vc_npar,vc_par[NPAR];            // ANSI 转义字符序列参数个数和参数数组
68.       unsigned long   vc_video_mem_start;
69.       unsigned long   vc_video_mem_end;
70.       unsigned int    vc_saved_x;
71.       unsigned int    vc_saved_y;
72.       unsigned int    vc_iscolor;
73.       unsigned char   vc_kbdapplic;
74.       unsigned char     vc_kbdleds;
75.       unsigned char   vc_kbdmode;
76.       char * vc_translate;
77.     } vc_cons [MAX_CONSOLES];
78.       #define MEM_BUFFER_SIZE (2*80*50*8)
79.       unsigned short *vc_scrbuf[MAX_CONSOLES];
80.       unsigned short vc_scrmembuf[MEM_BUFFER_SIZE/2];
81.       static int console_blanked = 0;
82.       #define origin          (vc_cons[currcons].vc_origin)
83.       #define scr_end         (vc_cons[currcons].vc_scr_end)
84.       #define pos             (vc_cons[currcons].vc_pos)
85.       #define top             (vc_cons[currcons].vc_top)
86.       #define bottom          (vc_cons[currcons].vc_bottom)
87.       #define x         (vc_cons[currcons].vc_x)
```

```
88.         #define y          (vc_cons[currcons].vc_y)
89.         #define state          (vc_cons[currcons].vc_state)
90.         #define restate        (vc_cons[currcons].vc_restate)
91.         #define checkin        (vc_cons[currcons].vc_checkin)
92.         #define npar           (vc_cons[currcons].vc_npar)
93.         #define par            (vc_cons[currcons].vc_par)
94.         #define ques           (vc_cons[currcons].vc_ques)
95.         #define attr           (vc_cons[currcons].vc_attr)
96.         #define saved_x        (vc_cons[currcons].vc_saved_x)
97.         #define saved_y        (vc_cons[currcons].vc_saved_y)
98.         #define translate(vc_cons[currcons].vc_translate)
99.         #define video_mem_start    (vc_cons[currcons].vc_video_mem_start)
100.        #define video_mem_end      (vc_cons[currcons].vc_video_mem_end)
101.        #define def_attr (vc_cons[currcons].vc_def_attr)
102.        #define video_erase_char   (vc_cons[currcons].vc_video_erase_char)
103.        #define iscolor        (vc_cons[currcons].vc_iscolor)
104.        #define kbdapplic      (vc_cons[currcons].vc_kbdapplic)
105.        #define kbdmode            (vc_cons[currcons].vc_kbdmode)
106.        #define kbdleds        (vc_cons[currcons].vc_kbdleds)
107.    int blankinterval = 5*60*HZ;
108.    static int screen_size = 0;
109.        static void sysbeep(void);          // 系统蜂鸣函数
        //下面是终端回应 ESC-Z 或 csi0c 请求的应答
110.    #define RESPONSE "\033[?1;2c"   // csi - 控制序列引导码(Control Sequence Introducer)
111.    static char * translations[] = {
112.
113.        " !\"#$%&'()*+,-./0123456789:;<=>?"
114.    "@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_"
115.    "`abcdefghijklmnopqrstuvwxyz{|}~ ",
116.
117.        " !\"#$%&'()*+,-./\333123456789:;<=>?"
118.    "@ABCDEFGH\017JKLMNOPQRSTUVWXYZ[\\]^ "
119.      /* '   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o */
120.       "\004\261\007\007\007\007\370\361\040\007\331\277\332\300\305\007"
121.      /* p   q   r   s   t   u   v   w   x   y   z   {   |   }   ~       */
122.    "\007\304\007\007\303\264\301\302\263\007\007\007\007\007\234 "
123.        };
124.        #define NORM_TRANS (translations[0])
125.    #define GRAF_TRANS (translations[1])
       /* 注意！ gotoxy 函数认为 x==video_num_columns，这是正确的 */
      // 跟踪光标当前位置
126. static inline void gotoxy(int currcons, int new_x,unsigned int new_y)
127.    {
      // 如果输入的光标行号超出显示器列数，或者光标行号超出显示的最大行数，则退出
128.        if (new_x > video_num_columns || new_y >= video_num_lines)
129.    return;
      // 更新当前光标变量；更新光标位置对应的在显示内存中位置变量 pos
130.    x = new_x;
131.    y = new_y;
132.    pos = origin + y*video_size_row + (x<<1);
```

```
133.      }
      // 设置滚屏起始显示内存地址
134.      static inline void set_origin(int currcons)
135.          {
136.          if (video_type != VIDEO_TYPE_EGAC && video_type != VIDEO_TYPE_EGAM)
137.          return;
138.          if (currcons != fg_console)
139.          return;
140.          cli();
      // 首先选择显示控制数据寄存器 r12，然后写入卷屏起始地址高字节。向右移动 9 位，
      // 表示向右移动 8 位，再除以 2（2 字节代表屏幕上 1 字符）。是相对于默认显示内存操作的
141.      outb_p(12, video_port_reg);
142.      outb_p(0xff&((origin-video_mem_base)>>9), video_port_val);
      // 再选择显示控制数据寄存器 r13，然后写入卷屏起始地址底字节。向右移动 1 位表示除以 2
143.      outb_p(13, video_port_reg);
144.      outb_p(0xff&((origin-video_mem_base)>>1), video_port_val);
145.        sti();
146.          }
      // 向上卷动一行
147.          static void scrup(int currcons)
148.      {
149.      unsigned int oldbottom, oldtop;
150.          oldbottom = bottom;
151.          oldtop = top;
152.          if (y < top) {
153.          top = 0;
154.      bottom = y + 1;
155.      } else if (y > bottom) {
156.          bottom = video_num_lines;
157.          top = y;
158.          }
159.          if (top > video_num_lines)
160.      top = 0;
161.      if (bottom > video_num_lines)
162.          bottom = video_num_lines;
163.          if (bottom <= top) {
164.          bottom = oldbottom;
165.      top = oldtop;
166.      return;
167.          }
       // 如果显示类型是 EGA，则执行以下操作
168.          if (video_type == VIDEO_TYPE_EGAC || video_type == VIDEO_TYPE_EGAM)
169.      {
      // 如果移动起始行 top=0，移动最底行 bottom=video_num_lines=25，则表示整屏窗口向下移动
170.      if (!top && bottom == video_num_lines) {
      // 调整屏幕显示对应内存的起始位置指针 origin 为向下移一行屏幕字符对应的内存位置，
      // 同时也调整当前光标对应的内存位置以及屏幕末行末端字符指针 scr_end 的位置
171.      origin += video_size_row;
172.      pos += video_size_row;
173.      scr_end += video_size_row;
```

```
        // 如果屏幕末端最后一个显示字符所对应的显示内存指针 scr_end 超出了实际显示内存的
        // 末端，则将屏幕内容内存数据移动到显示内存的起始位置 video_mem_start 处，并在出现的
        // 新行上填入空格字符
174.    if (scr_end > video_mem_end) {
175.                    __asm__("cld\n\t"                          // 清方向位
176.                        "rep\n\t"                              // 重复操作
177.                        "movsl\n\t"                            // 移动到显示内存起始处
178.                        "movl _video_num_columns,%1\n\t"// 行字符数 ecx = 1
179.                        "rep\n\t"                              // 在新行上填入空格字符
180.        "stosw"
181.        ::"a" (video_erase_char),
182.        "c" ((video_num_lines-1)*video_num_columns>>1),
183.        "D" (video_mem_start),
184.        "S" (origin)
185.        :"cx","di","si");
        // 根据屏幕内存数据移动后的情况，重新调整当前屏幕对应内存的起始指针、光标位置指针和
        // 屏幕末端对应内存指针 scr_end
186.    scr_end -= origin-video_mem_start;
187.    pos -= origin-video_mem_start;
188.        origin = video_mem_start;
189.        } else {
        // 如果调整后的屏幕末端对应的内存指针 scr_end 没有超出显示内存的末端 video_mem_end，
        // 则只需在新行上填入擦除字符(空格字符)
190.                    __asm__("cld\n\t"                          // 清方向位
191.                        "rep\n\t"                              // 重复操作，在新出现行上
192.                        "stosw"                                // 填入擦除字符（空格字符）
193.        ::"a" (video_erase_char),
194.        "c" (video_num_columns),
195.        "D" (scr_end-video_size_row)
196.        :"cx","di");
197.        }
        // 向显示控制器中写入新的屏幕内容对应的内存起始位置值
198.        set_origin(currcons);
199.            } else {            // 否则表示不是整屏移动
200.                __asm__("cld\n\t"                              // 清方向位
201.                    "rep\n\t"                                  // 循环操作，将 top+1 到 bottom 行
202.                    "movsl\n\t"                                // 所对应的内存块移到 top 行开始处
203.                    "movl _video_num_columns,%%ecx\n\t"// 行字符数 ecx = 1
204.                    "rep\n\t"                                  // 在新行上填入擦除字符
205.    "stosw"
206.        ::"a" (video_erase_char),
207.        "c" ((bottom-top-1)*video_num_columns>>1),
208.        "D" (origin+video_size_row*top),
209.        "S" (origin+video_size_row*(top+1))
210.        :"cx","di","si");
211.        }
212.        }
        // 如果显示类型不是 EGA(是 MDA)，则执行下面移动操作
213.        else
214.        {
```

```
215.          __asm__("cld\n\t"
216.          "rep\n\t"
217.          "movsl\n\t"
218.          "movl _video_num_columns,%%ecx\n\t"
219.          "rep\n\t"
220.          "stosw"
221.          ::"a" (video_erase_char),
222.          "c" ((bottom-top-1)*video_num_columns>>1),
223.          "D" (origin+video_size_row*top),
224.          "S" (origin+video_size_row*(top+1))
225.          :"cx","di","si");
226.          }
227.      bottom = oldbottom;
228.      top = oldtop;
229.          }
      // 向下卷动一行
230.          static void scrdown(int currcons)
231.          {
232.          unsigned int oldbottom, oldtop;
233.          301
234.          oldbottom = bottom;
235.          oldtop = top;
236.          if (y < top) {
237.          top = 0;
238.          bottom = y + 1;
239.          } else if (y > bottom) {
240.          bottom = video_num_lines;
241.          top = y;
242.          }
243.          if (top > video_num_lines)
244.          top = 0;
245.          if (bottom > video_num_lines)
246.          bottom = video_num_lines;
247.          if (bottom<=top) {
248.          bottom = oldbottom;
249.          top = oldtop;
250.          return;
251.          }
252.           __asm__("std\n\t"                          // 置方向位
253.                "rep\n\t"                             // 重复操作，向下移动从 top 行到 bottom-1 行
254.                "movsl\n\t"                           // 对应的内存数据
255.                "addl $2,%%edi\n\t"                   /* %edi 已经减 4，因为也是方向填擦除字符 */
256.                "movl _video_num_columns,%%ecx\n\t"   // 置 ecx=1 行字符数
257.                "rep\n\t"                             // 将擦除字符填入上方新行中
258.          "stosw\n\t"
259.          "cld"
260.          ::"a" (video_erase_char),
261.          "c" ((bottom-top-1)*video_num_columns>>1),
262.          "D" (origin+video_size_row*bottom-4),
```

```
263.          "S" (origin+video_size_row*(bottom-1)-4)
264.          :"ax","cx","di","si");
265.          bottom = oldbottom;
266.          top = oldtop;
267.          }
          // 光标位置下移一行
268.          static void lf(int currcons)
269.      {
          // 如果光标没有位于倒数第 2 行之后，则直接修改光标当前行变量 y++，并调整光标对应显示
          //   内存位置 pos
270.      if (y+1<bottom) {
271.          y++;
272.          pos += video_size_row;
273.      return;
274.          } else                    // 否则需要将屏幕内容上移一行
275.          scrup(currcons);
276.          }
          // 光标上移一行
277.          static void ri(int currcons)
278.          {
          // 如果光标不在第 1 行，则直接修改光标当前行标量 y--，并调整光标对应显示内存位置 pos
279.      if (y>top) {
280.      y--;
281.          pos -= video_size_row;
282.          return;
283.          } else                                    // 否则需要将屏幕内容下移一行
284.          scrdown(currcons);
285.          }
          // 光标回到第 1 列（0 列）左端
286.      static void cr(int currcons)
287.      {
288.          pos -= x<<1;                              // 光标所在的列号*2
289.          x=0;
290.          }
          // 擦除光标前一字符
291.          static void del(int currcons)
292.          {
          // 如果光标没有在 0 列，则将光标对应内存位置指针 pos 后退 2 字节，然后将当前光标变量
          // 列值 x 减 1，并将光标所在位置字符擦除
293.      if (x) {
294.      pos -= 2;
295.          x--;
296.          *(unsigned short *)pos = video_erase_char;
297.          }
298.          }
          // 删除屏幕上与光标位置相关的部分
299.          static void csi_J(int currcons, int vpar)
300.      {
301.      long count;
302.          long start;
```

```
303.          switch (vpar) {// 根据以下三种情况分别设置需要删除的字符数和删除开始的显示内存位置
304.                case 0:                                          /* 擦除光标到屏幕底端 */
305.      count = (scr_end-pos)>>1;
306.      start = pos;
307.          break;
308.          case 1:                                              /* 删除从屏幕开始到光标处的字符 */
309.          count = (pos-origin)>>1;
310.      start = origin;
311.      break;
312.          case 2:                                              /* 删除整个屏幕上的字符 */
313.          count = video_num_columns * video_num_lines;
314.          start = origin;
315.          break;
316.          default:
317.          return;
318.          }
        // 使用擦除字符填写删除字符的地方
319.          __asm__("cld\n\t"
320.          "rep\n\t"
321.          "stosw\n\t"
322.          ::"c" (count),
323.          "D" (start),"a" (video_erase_char)
324.          :"cx","di");
325.          }
        // 删除行内与光标位置相关的部分
326.          static void csi_K(int currcons, int vpar)
327.      {
328.          long count;
329.          long start;
330.          switch (vpar) { // 根据三种情况分别设置需要删除的字符数和删除开始的显示内存位置
331.                case 0:                                          /* 删除光标到行尾字符 */
332.                   if (x>=video_num_columns)
333.                   return;
334.                      count = video_num_columns-x;
335.                      start = pos;
336.                      break;
337.                case 1:                                          /* 删除从行开始到光标处 */
338.      start = pos - (x<<1);
339.      count = (x<video_num_columns)?x:video_num_columns;
340.          break;
341.                case 2:                                          /* 将整行字符全删除 */
342.                   start = pos - (x<<1);
343.                   count = video_num_columns;
344.          break;
345.          default:
346.          return;
347.          }
        // 使用擦除字符填写删除字符的地方
348.          __asm__("cld\n\t"
349.      "rep\n\t"
```

```
350.        "stosw\n\t"
351.          ::"c" (count),
352.            "D" (start),"a" (video_erase_char)
353.            :"cx","di");
354.            }
          // 允许重新设置字符属性，比如加粗、加下画线、闪烁、反显等
355.        void csi_m(int currcons )
356.        {
357.            int i;
358.            static int conv_table[8] = { 0, 4, 2, 6, 1, 5, 3, 7 };
359.            for (i=0;i<=npar;i++)
360.            switch (par[i]) {
361.        case 0: attr=def_attr;break;
362.        case 1: attr=(iscolor?attr|0x08:attr|0x0f);break;
363.
364.        case 4: /* bold */
365.        if (!iscolor)
366.            attr |= 0x01;
367.            else
368.
369.            if (vc_cons[currcons].vc_bold_attr != -1)
370.                attr = (vc_cons[currcons].vc_bold_attr&0x0f)|(0xf0&(attr));
371.            else
372.            { short newattr = (attr&0xf0)|(0xf&(~attr));
373.            attr = ((newattr&0xf)==((attr>>4)&0xf)?
374.            (attr&0xf0)|((((attr&0xf)+1)%0xf):
375.            newattr);
376.            }
377.            }
378.            break;
379.            case 5: attr=attr|0x80;break;
380.            case 7: attr=(attr<<4)|(attr>>4);break;
381.            case 22: attr=attr&0xf7;break;
382.            case 24: attr=attr&0xfe;break;
383.            case 25: attr=attr&0x7f;break;
384.            case 27: attr=def_attr;break;
385.            case 39: attr=(attr & 0xf0)|(def_attr & 0x0f); break;
386.            case 49: attr=(attr & 0x0f)|(def_attr & 0xf0); break;
387.        default:
388.        if (!can_do_colour)
389.            break;
390.            iscolor = 1;
391.            if ((par[i]>=30) && (par[i]<=37))
392.            attr = (attr & 0xf0) | conv_table[par[i]-30];
393.        else
394.            if ((par[i]>=40) && (par[i]<=47))
395.                attr = (attr & 0x0f) | (conv_table[par[i]-40]<<4);
396.                else
397.        break;
398.        }
```

```
399.          }
      // 根据设置显示光标
400.          static inline void set_cursor(int currcons)
401.          {
402.          if (currcons != fg_console)
403.      return;
404.      cli();
      // 首先使用索引寄存器端口选择显示控制数据寄存器 r14，然后写入光标当前位置高字节
405.          outb_p(14, video_port_reg);
406.          outb_p(0xff&((pos-video_mem_base)>>9), video_port_val);
      // 再使用索引寄存器选择 r15，并将光标当前位置低字节写入其中
407.          outb_p(15, video_port_reg);
408.          outb_p(0xff&((pos-video_mem_base)>>1), video_port_val);
409.          sti();
410.          }
       // 隐藏光标
411.          static inline void hide_cursor(int currcons)
412.      {
413.          outb_p(14, video_port_reg);
414.          outb_p(0xff&((scr_end-video_mem_base)>>9), video_port_val);
415.          outb_p(15, video_port_reg);
416.          outb_p(0xff&((scr_end-video_mem_base)>>1), video_port_val);
417.      }
      // 将响应序列放入读缓冲队列中
418.          static void respond(int currcons, struct tty_struct * tty)
419.          {
420.          char * p = RESPONSE;
421.          cli();                                  // 关中断
422.      while (*p) {
     // 将字符序列放入写队列
423.          PUTCH(*p,tty->read_q);
424.          p++;
425.          }
426.          sti();                                  // 开中断
427.          copy_to_cooked(tty);                    // 转换成规范模式
428.      }
      // 在光标处插入一空格字符
429.      static void insert_char(int currcons)
430.          {
431.          int i=x;
432.          unsigned short tmp, old = video_erase_char;
433.          unsigned short * p = (unsigned short *) pos;
434.          while (i++<video_num_columns) {
     // 光标开始的所有字符右移一格，并将擦除字符插入在光标所在处
435.          tmp=*p;
436.          *p=old;
437.          old=tmp;
438.          p++;
439.      }
440.          }
```

```
        // 在光标处插入一行
441.    static void insert_line(int currcons)
442.    {
443.    int oldtop,oldbottom;
444.    oldtop=top;                          // 保存原 top、bottom 值
445.    oldbottom=bottom;
446.    top=y;                               // 设置屏幕卷动开始行
447.    bottom = video_num_lines;            // 设置屏幕卷动最后行
448.    scrdown(currcons);                   // 从光标开始处，屏幕内容向下滚动一行
449.    top=oldtop;                          // 恢复原 top，bottom 值
450.    bottom=oldbottom;
451.    }
        // 删除光标处的一个字符
452.    static void delete_char(int currcons)
453.    {
454.    int i;
455.    unsigned short * p = (unsigned short *) pos;
456.    if (x>=video_num_columns)            // 如果光标超出屏幕最右列，则返回
457.    return;
        // 从光标右一个字符开始到行末所有字符左移一格
458.    i = x;
459.    while (++i < video_num_columns) {
460.    *p = *(p+1);
461.     p++;
462.    }
        // 最后一个字符处填入擦除字符
463.    *p = video_erase_char;
464.    }
        // 删除光标所在行
465.    static void delete_line(int currcons)
466.    {
467.    int oldtop,oldbottom;
468.    oldtop=top;                          // 保存原 top，bottom 值
469.    oldbottom=bottom;
470.    top=y;                               // 设置屏幕卷动开始行
471.    bottom = video_num_lines;            // 设置屏幕卷动最后行
472.    scrup(currcons);                     // 从光标开始处，屏幕内容向上滚动一行
473.    top=oldtop;                          // 恢复原 top，bottom 值
474.    bottom=oldbottom;
475.    }
        // 在光标处插入 nr 个字符
476.    static void csi_at(int currcons, unsigned int nr)
477.    {
        // 如果插入的字符数大于一行字符数，则截为一行字符数
478.    if (nr > video_num_columns)
479.    nr = video_num_columns;
480.    else if (!nr)                        // 若插入字符数 nr 为 0，则插入 1 个字符
481.    nr = 1;
482.    while (nr--)                         // 循环插入指定的字符数
483.    insert_char(currcons);
```

```
484.      }
        // 在光标位置处插入 nr 行
485.      static void csi_L(int currcons, unsigned int nr)
486.      {
        // 如果插入的行数大于屏幕最多行数，则截为屏幕显示行数
487.      if (nr > video_num_lines)
488.      nr = video_num_lines;
489.      else if (!nr)                    // 若插入行数 nr 为 0，则插入 1 行
490.      nr = 1;
491.      while (nr--)                     // 循环插入指定行数 nr
492.      insert_line(currcons);
493.      }
        // 删除光标处的 nr 个字符
494.      static void csi_P(int currcons, unsigned int nr)
495.      {
        // 如果删除的字符数大于 1 行字符数，则截为一行字符数
496.      if (nr > video_num_columns)
497.      nr = video_num_columns;
498.      else if (!nr)                    // 若删除字符数 nr 为 0，则删除 1 个字符
499.      nr = 1;
500.      while (nr--)                     // 循环删除指定字符数 nr
501.      delete_char(currcons);
502.      }
        // 删除光标处的 nr 行
503.      static void csi_M(int currcons, unsigned int nr)
504.      {
        // 如果删除的行数大于屏幕最多行数，则截为屏幕显示行数
505.      if (nr > video_num_lines)
506.      nr = video_num_lines;
507.      else if (!nr)                    // 若删除的行数 nr 为 0，则删除 1 行
508.      nr=1;
509.      while (nr--)                     // 循环删除指定行数 nr
510.      delete_line(currcons);
511.      }
        // 保存当前光标位置
512.      static void save_cur(int currcons)
513.      {
514.      saved_x=x;
515.      saved_y=y;
516.      }
        // 恢复保存的光标位置
517.      static void restore_cur(int currcons)
518.      {
519.      gotoxy(currcons,saved_x, saved_y);
520.      }
521.      enum { ESnormal, ESesc, ESsquare, ESgetpars, ESgotpars, ESfunckey,
522.      ESsetterm, ESsetgraph, ESgraph, ESgresc, ESignore };
        // 控制台写函数，从终端对应的 tty 写缓冲队列中取字符，针对每个字符进行分析
523.      void con_write(struct tty_struct * tty)
524.      {
```

```
525.            int nr;
526.            char c;
527.            int currcons;
528.            wake_up(&tty->write_q->proc_list);
529.            currcons = tty - tty_table;
530.            if ((currcons>=MAX_CONSOLES) || (currcons<0)) {
531.          printk("con_write: illegal tty\n\r");
532.          return;
533.            }
              // 首先取得写缓冲队列中现有字符数 nr，然后针对每个字符进行处理
534.            nr = CHARS(tty->write_q);
535.            while (nr--) {
536.            if (tty->stopped)
537.            break;
              // 从写队列中取一字符 c，根据前面所处理字符的状态 state 分别处理
538.          GETCH(tty->write_q,c);
539.            if (c == 24 || c == 26)
540.            state = ESnormal;
541.            switch(state) {
542.            case ESnormal:
543.            if (c>31 && c<127) {
              // 如果字符不是控制字符（c>31），并且也不是扩展字符（c<127），则
544.            if (x>=video_num_columns) {
              // 若当前光标处在行末端或末端以外，则将光标移到下行头列，
              // 并调整光标位置对应的内存指针 pos
545.          x -= video_num_columns;
546.          pos -= video_size_row;
547.            lf(currcons);
548.            }
549.            *(char *) pos = translate[c-32];
550.            pos++;
551.            *(char *) pos = attr;
552.            pos++;
553.            x++;
554.            } else if (c==27)
              // 如果字符 c 是转义字符 ESC，则转换状态 state 到 ESesc
555.            state = ESesc;
              // 如果字符 c 是换行符(10)，或是垂直制表符 VT(11)，或者是换页符 FF(12)，
              // 则移动光标到下一行
556.          else if (c==10 || c==11 || c==12)
557.          lf(currcons);
              // 如果字符 c 是回车符 CR(13)，则将光标移动到头列（0 列）
558.            else if (c==13)
559.            cr(currcons);
              // 如果字符 c 是 DEL(127)，则将光标右边一字符擦除，并将光标移到被擦除位置
560.            else if (c==127)
561.            del(currcons);
            // 如果字符 c 是 BS(backspace,8)，则将光标右移 1 格，并相应调整光标对应内存位置指针 pos
562.          else if (c==8) {
563.            if (x) {
```

```
564.            x--;
565.            pos -= 2;
566.        }
567.        } else if (c==9) {
        // 如果字符 c 是水平制表符 TAB(9)，则将光标移到 8 的倍数列上
568.        c = 8-(x&7);
569.        x += c;
570.        pos += c<<1;
571.        if (x>video_num_columns) {
572.        x -= video_num_columns;
573.        pos -= video_size_row;
574.        lf(currcons);
575.    }
576.        c=9;
577.        } else if (c==7)
        // 如果字符 c 是响铃符 BEL(7)，则调用蜂鸣函数，是扬声器发声
578.        sysbeep();
579.        else if (c == 14) {
580.        checkin = 1;
581.        translate = restate;
582.        } else if (c == 15) {
583.        translate = NORM_TRANS;
584.        checkin = 0;
585.        }
586.        break;
        // 如果原状态是 ESesc，则转到状态 ESnormal 处理
587.        case ESesc:
588.        state = ESnormal;
589.        switch (c) {
        // 如果字符 c 是'['，则将状态 state 转到 ESsquare
590.        case '[':
591.        state = ESsquare;
592.      break;
        // 如果字符 c 是'E'，则光标移到下一行开始处（0 列）
593.        case 'E':
594.        gotoxy(currcons,0,y+1);
595.        break;
        // 如果字符 c 是'M'，则光标上移一行
596.        case 'M':
597.        ri(currcons);
598.        break;
        // 如果字符 c 是'D'，则光标下移一行
599.        case 'D':
600.        lf(currcons);
601.        break;
        // 如果字符 c 是'Z'，则发送终端应答字符序列
602.        case 'Z':
603.        respond(currcons,tty);
604.        break;
        // 如果字符 c 是'7'，则保存当前光标位置
```

```
605.            case '7':
606.            save_cur(currcons);
607.            break;
        // 如果字符 c 是'8'，则恢复到原保存的光标位置
608.            case '8':
609.            restore_cur(currcons);
610.            break;
        // 如果字符 c 是'('和')'，则转到状态 ESsetgraph 处理
611.            case '(':
612.            case ')':
613.            state = ESsetgraph;
614.            break;
         // 如果字符 c 是'P'，则转到状态 ESsetterm 处理
615.            case 'P':
616.            state = ESsetterm;
617.            break;
        // 如果字符 c 是'#'，设置状态为 state 值为-1
618.            case '#':
619.            state = -1;
620.            break;
        // 如果字符 c 是'c'，则转到状态 ESnormal 处理
621.            case 'c':
622.            tty->termios = DEF_TERMIOS;
623.            state = ESnormal;
624.            restate = NORM_TRANS;
625.            checkin = 0;
626.            top = 0;
627.            bottom = video_num_lines;
628.            translate = NORM_TRANS;
        // 如果字符 c 是'>'，设置 kbdapplic 值为 0
629.            case '>':
630.            kbdapplic = 0;
631.            if (currcons == fg_console)
632.            kapplic = 0;
633.            break;
        // 如果字符 c 是'='，设置 kbdapplic 值为 1
634.            case '=':   /* Appl. keypad */
635.            kbdapplic = 1;
636.            if (currcons == fg_console)
637.            kapplic = 1;
638.            break;
639.            }
640.            break;
         // 如果原状态是 ESesc，则首先对 ESC 转义字符序列参数使用的处理数组 par[]清零，
         // 索引变量 npar 指向首项，并且设置状态为 ESgetpars
641.            case ESsquare:
642.            for(npar = 0 ; npar < NPAR ; npar++)
643.            par[npar]=0;
644.            npar = 0;
645.            state = ESgetpars;
```

```
646.          if (c == '[') { /* Function key */
647.          state=ESfunckey;
648.          break;
649.          }
650.          if (ques=(c=='?'))
651.          break;
652.        case ESgetpars:
            // 如果字符 c 是分号';'，并且数组 par 未满，则索引值加 1
653.          if (c==';' && npar<NPAR-1) {
654.             npar++;
655.          break;
          //如果字符 c 是数字字符'0'～'9'，则将该字符转换成数值并与 npar 所索引的项组成 10 进制数
656.          } else if (c>='0' && c<='9') {
657.             par[npar]=10*par[npar]+c-'0';
658.             break;
659.          } else state=ESgotpars;
        // 如果原状态是 ESgotpars，则转到 ESnormal 处理
660.          case ESgotpars:
661.          state = ESnormal;
662.          if (ques) {
663.             ques = 0;
664.          break;
665.          }
666.          switch(c) {
        // 如果字符 c 是'G'或'`'，则 par[]中第一个参数代表列号。若列号不为零，则将光标右移一格
667.          case 'G': case '`':
668.          if (par[0]) par[0]--;
669.             gotoxy(currcons,par[0],y);
670.          break;
        // 如果字符 c 是'A'，则第一个参数代表光标上移的行数
671.          case 'A':
672.          if (!par[0]) par[0]++;
673.          gotoxy(currcons,x,y-par[0]);
674.             break;
        // 如果字符 c 是'B'或'e'，则第一个参数代表光标下移的行数
675.          case 'B': case 'e':
676.          if (!par[0]) par[0]++;
677.             gotoxy(currcons,x,y+par[0]);
678.          break;
        // 如果字符 c 是'C'或'a'，则第一个参数代表光标右移的格数
679.          case 'C': case 'a':
680.          if (!par[0]) par[0]++;
681.          gotoxy(currcons,x+par[0],y);
682.          break;
        // 如果字符 c 是'D'，则第一个参数代表光标左移的格数
683.          case 'D':
684.          if (!par[0]) par[0]++;
685.          gotoxy(currcons,x-par[0],y);
686.          break;
        // 如果字符 c 是'E'，则第一个参数代表光标向下移动的行数，并回到 0 列
```

```
687.            case 'E':
688.            if (!par[0]) par[0]++;
689.            gotoxy(currcons,0,y+par[0]);
690.            break;
          // 如果字符 c 是'F'，则第一个参数代表光标向上移动的行数，并回到 0 列
691.            case 'F':
692.            if (!par[0]) par[0]++;
693.            gotoxy(currcons,0,y-par[0]);
694.            break;
          // 如果字符 c 是'd'，则第一个参数代表光标所需在的行号
695.            case 'd':
696.            if (par[0]) par[0]--;
697.            gotoxy(currcons,x,par[0]);
698.            break;
          // 如果字符 c 是'H'或'f'，则第一个参数代表光标移到的行号，第二个参数代表光标移到的列号
699.            case 'H': case 'f':
700.            if (par[0]) par[0]--;
701.            if (par[1]) par[1]--;
702.            gotoxy(currcons,par[1],par[0]);
703.            break;
          // 如果字符 c 是'J'，则第一个参数代表以光标所处位置清屏的方式
704.            case 'J':
705.            csi_J(currcons,par[0]);
706.            break;
          // 如果字符 c 是'K'，则第一个参数代表以光标所在位置对行中字符进行删除处理的方式
707.            case 'K':
708.            csi_K(currcons,par[0]);
709.            break;
          // 如果字符 c 是'L'，表示在光标位置处插入 n 行
710.            case 'L':
711.            csi_L(currcons,par[0]);
712.            break;
          // 如果字符 c 是'M'，表示在光标位置处删除 n 行
713.            case 'M':
714.            csi_M(currcons,pai[0]);
715.            break;
          // 如果字符 c 是'P'，表示在光标位置处删除 n 个字符
716.          case 'P':
717.          csi_P(currcons,par[0]);
718.            break;
          // 如果字符 c 是'@'，表示在光标位置处插入 n 个字符
719.            case '@':
720.            csi_at(currcons,par[0]);
721.            break;
          // 如果字符 c 是'm'，表示改变光标处字符的显示属性
722.            case 'm':
723.            csi_m(currcons);
724.            break;
          // 如果字符 c 是'r'，则表示用两个参数设置滚屏的起始行号和终止行号
725.          case 'r':
```

```
726.      if (par[0]) par[0]--;
727.          if (!par[1]) par[1] = video_num_lines;
728.          if (par[0] < par[1] &&
729.          par[1] <= video_num_lines) {
730.          top=par[0];
731.          bottom=par[1];
732.          }
733.          break;
        // 如果字符 c 是's'，则表示保存当前光标所在位置
734.          case 's':
735.          save_cur(currcons);
736.          break;
        // 如果字符 c 是'u'，则表示恢复光标到原保存的位置处
737.          case 'u':
738.          restore_cur(currcons);
739.          break;
740.          case 'l':
741.          case 'b':
742.      if (!((npar >= 2) &&
743.          ((par[1]-13) == par[0]) &&
744.          ((par[2]-17) == par[0])))
745.          break;
746.          if ((c=='l') && (par[0]<=60)) {
747.          blankinterval = HZ*60*par[0];
748.      }
749.          if (c=='b')
750.          vc_cons[currcons].vc_bold_attr
751.          = par[0];
752.          }
753.          break;
        // 如果原状态是 ESfunckey，则转到 ESnormal 处理
754.          case ESfunckey:
755.          state = ESnormal;
756.          break;
        // 如果原状态是 ESsetterm，则转到 ESnormal 处理
757.          case ESsetterm:
758.          state = ESnormal;
759.          if (c == 'S') {
760.          def_attr = attr;
761.          video_erase_char = (video_erase_char&0x0ff) | (def_attr<<8);
762.          } else if (c == 'L')
763.
764.          else if (c == 'l')
765.
766.          break;
        // 如果原状态是 ESsetgraph，则转到 ESnormal 处理
767.          case ESsetgraph:
768.          if (c == '0') {
769.          if (checkin)
770.          translate = GRAF_TRANS;
```

```
771.            restate = GRAF_TRANS;
772.            } else if (c == 'B')
773.            translate = restate = NORM_TRANS;
774.            state = ESnormal;
775.            break;
776.            default:
777.            state = ESnormal;
778.            }
779.            }
780.            set_cursor(currcons);
781.            timer_active &= ~(1<<BLANK_TIMER);
782.            if (console_blanked) {
783.            timer_table[BLANK_TIMER].expires = 0;
784.            timer_active |= 1<<BLANK_TIMER;
785.            } else if (blankinterval) {
786.            timer_table[BLANK_TIMER].expires = jiffies + blankinterval;
787.            timer_active |= 1<<BLANK_TIMER;
788.            }
789.            }
        // 键盘中断处理函数
790.            void do_keyboard_interrupt(void)
791.            {
792.            copy_to_cooked(TTY_TABLE(0));
793.            timer_active &= ~(1<<BLANK_TIMER);
794.            if (console_blanked) {
795.            timer_table[BLANK_TIMER].expires = 0;
796.            timer_active |= 1<<BLANK_TIMER;
797.        } else if (blankinterval) {
798.            timer_table[BLANK_TIMER].expires = jiffies + blankinterval;
799.            timer_active |= 1<<BLANK_TIMER;
800.        }
801.        }
802.            void * memsetw(void * s,unsigned short c,int count)
803.            {
804.            __asm__("cld\n\t"
805.            "rep\n\t"
806.            "stosw"
807.            ::"a" (c),"D" (s),"c" (count)
808.            :"cx","di");
809.            return s;
810.            }
811.            void con_init(void)
812.            {
813.            register unsigned char a;
814.            char *display_desc = "????";
815.            char *display_ptr;
816.            int currcons = 0;
817.            long base, term;
818.            long video_memory;
819.            long saveterm, savebase;
```

```
820.        video_num_columns = ORIG_VIDEO_COLS;          // 显示器显示字符列数
821.        video_size_row = video_num_columns * 2;       // 每行需使用字节数
822.        video_num_lines = ORIG_VIDEO_LINES;           // 显示器显示字符行数
823.        video_page = ORIG_VIDEO_PAGE;                 // 当前显示页面
824.        video_erase_char = 0x0720;                    // 擦除字符
825.        timer_table[BLANK_TIMER].fn = blank_screen;
826.        timer_table[BLANK_TIMER].expires = 0;
827.        if (blankinterval) {
828.        timer_table[BLANK_TIMER].expires = jiffies+blankinterval;
829.        timer_active |= 1<<BLANK_TIMER;
830.        }
831.      // 如果原始显示模式等于 7，则表示是单色显示器
832.     if (ORIG_VIDEO_MODE == 7)
833.        {
834.              video_mem_base = 0xb0000;               // 设置单显映象内存起始地址
835.              video_port_reg = 0x3b4;                 // 设置单显索引寄存器端口
836.              video_port_val = 0x3b5;                 // 设置单显数据寄存器端口
            // 根据 BIOS 中断 int 0x10 功能获得的显示模式信息
            // 判断显卡是单色显卡还是彩色显卡
837.              if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)    // EGA 卡
838.     {
839.                    video_type = VIDEO_TYPE_EGAM;     // 设置显示类型（EGA 单色）
840.                    video_mem_term = 0xb8000;         // 设置显示内存末端地址
841.                    display_desc = "EGAm";            // 设置显示描述字符串
842.     }
      // 如果 BX 寄存器的值等于 0x10，则说明是单色显示卡 MDA
843.        else
844.        {
845.                    video_type = VIDEO_TYPE_MDA;      // 设置显示类型（MDA 单色）
846.                    video_mem_term = 0xb2000;         // 设置显示内存末端地址
847.                    display_desc = "*MDA";            // 设置显示描述字符串
848.        }
849.        }
850.        else
      // 如果显示模式不为 7，则为彩色模式
851.     {
852.              can_do_colour = 1;
853.              video_mem_base = 0xb8000;               // 显示内存起始地址
854.              video_port_reg   = 0x3d4;               // 设置彩色显示索引寄存器端口
855.              video_port_val   = 0x3d5;               // 设置彩色显示数据寄存器端口
      //  如果 BX 不等于 0x10，则说明是 EGA 显卡
856.      if ((ORIG_VIDEO_EGA_BX & 0xff) != 0x10)
857.        {
858.                    video_type = VIDEO_TYPE_EGAC;     // 设置显示类型（EGA 彩色）
859.                    video_mem_term = 0xc0000;         // 设置显示内存末端地址
860.                    display_desc = "EGAc";            // 设置显示描述字符串
861.        }
       // 如果 BX 寄存器的值等于 0x10，则说明是 CGA 显卡
862.        else
863.        {
```

```
                video_type = VIDEO_TYPE_CGA;              // 设置显示类型（CGA）
                video_mem_term = 0xba000;                 // 设置显示内存末端地址
                display_desc = "*CGA";                    // 设置显示描述字符串
        }
        }
        display_ptr = ((char *)video_mem_base) + video_size_row - 8;
        while (*display_desc)
    {
        *display_ptr++ = *display_desc++;
        display_ptr++;
        }
        savebase = video_mem_base;
        saveterm = video_mem_term;
        memsetw(vc_scrmembuf,video_erase_char,MEM_BUFFER_SIZE/2);
        video_mem_base = (long)vc_scrmembuf;
        video_mem_term = (long)&(vc_scrmembuf[MEM_BUFFER_SIZE/2]);
        video_memory = video_mem_term - video_mem_base;
        screen_size = (video_num_lines * video_size_row);
        NR_CONSOLES = video_memory / screen_size;
        if (NR_CONSOLES > MAX_CONSOLES)
        NR_CONSOLES = MAX_CONSOLES;
        if (!NR_CONSOLES)
        NR_CONSOLES = 1;
        video_memory = screen_size;
    /* 初始化用于滚屏的变量（主要用于 EGA/VGA） */
        base = origin = video_mem_start = video_mem_base;
        term = video_mem_end = base + video_memory;
        scr_end    = video_mem_start + screen_size;
        top  = 0;
        bottom     = video_num_lines;
        attr = 0x07;
        def_attr = 0x07;
        restate = NORM_TRANS;
        state = ESnormal;
        checkin = 0;
        ques = 0;
        iscolor = 0;
        translate = NORM_TRANS;
        kbdleds = 2;
        kbdmode = 0;
        kbdapplic = 0;
        vc_cons[0].vc_bold_attr = -1;
        gotoxy(currcons,ORIG_X,ORIG_Y);                   // 跟踪光标当前位置
        for (currcons = 1; currcons<NR_CONSOLES; currcons++) {
        vc_cons[currcons] = vc_cons[0];
        origin = video_mem_start = (base += video_memory);
        scr_end = origin + video_num_lines * video_size_row;
        video_mem_end = (term += video_memory);
        gotoxy(currcons,0,0);
        }
```

```
913.        for (currcons = 0; currcons<NR_CONSOLES; currcons++)
914.        vc_scrbuf[currcons] = (unsigned short *)origin;
915.        currcons = 0;
916.        video_mem_base = savebase;
917.        video_mem_term = saveterm;
918.        video_mem_start = video_mem_base;
919.        video_mem_end = video_mem_term;
920.        origin      = video_mem_start;                    // 滚屏起始显示内存地址
921.        scr_end     = video_mem_start + video_num_lines * video_size_row; //滚屏结束内存地址
922.        top  = 0;                                         // 顶端行号
923.        bottom      = video_num_lines;                    // 底端行号
924.        pos=origin + y*video_size_row + (x<<1);           // 设置内存位置 pos
925.        update_screen(fg_console);                        // 更新屏幕
926.        set_trap_gate(0x21,&keyboard_interrupt);          // 设置键盘中断陷阱门
927.        outb_p(inb_p(0x21)&0xfd,0x21);// 取消键盘中断的屏蔽，允许 IRQ1
928.        a=inb_p(0x61);                                    // 延迟读取键盘端口 0x61
929.        outb_p(a|0x80,0x61);                              // 设置禁止键盘工作
930.        outb_p(a,0x61);                                   // 再允许键盘工作，用以复位键盘操作
931.        }
    // 用于保存状态参数
932.        void kbdsave(int new_console)
933.    {
934.        int currcons = fg_console;
935.        kbdmode = kmode;
936.        kbdleds = kleds;
937.        kbdapplic = kapplic;
938.        currcons = new_console;
939.        kmode = (kmode & 0x3F) | (kbdmode & 0xC0);
940.        kleds = kbdleds;
941.        kapplic = kbdapplic;
942.        set_leds();
943.        }
    // 获取屏幕显示内存信息
944.        static void get_scrmem(int currcons)
945.    {
946.        memcpy((void *)vc_scrbuf[fg_console],(void *)origin, screen_size);
947.        video_mem_start = (unsigned long)vc_scrbuf[fg_console];
948.        origin = video_mem_start;
949.        scr_end = video_mem_end = video_mem_start+screen_size;
950.        top = 0;
951.        pos = origin + y*video_size_row + (x<<1);
952.        bottom = video_num_lines;
953.    }
    // 设置屏幕显示内存
954.        static void set_scrmem(int currcons)
955.        {
956.        video_mem_start = video_mem_base;
957.        video_mem_end = video_mem_term;
958.        origin      = video_mem_start;
959.        scr_end     = video_mem_start + screen_size;
```

```
960.          top  = 0;
961.          bottom     = video_num_lines;
962.          pos = origin + y*video_size_row + (x<<1);
963.          memcpy((void *)video_mem_base, (void *)vc_scrbuf[fg_console], screen_size);
964.      }
      // 黑屏处理，在某段时间无任何操作时就让屏幕黑屏
965.          void blank_screen(void)
966.          {
967.          timer_table[BLANK_TIMER].fn = unblank_screen;
968.          get_scrmem(fg_console);
969.          hide_cursor(fg_console);
970.          console_blanked = 1;
971.          memsetw((void *)video_mem_base, 0x0020, video_mem_term-video_mem_base );
972.      }
     // 恢复屏幕
973.      void unblank_screen(void)
974.          {
975.          timer_table[BLANK_TIMER].fn = blank_screen;
976.          if (blankinterval) {
977.              timer_table[BLANK_TIMER].expires = jiffies + blankinterval;
978.              timer_active |= 1<<BLANK_TIMER;
979.          }
980.          console_blanked = 0;
981.          set_scrmem(fg_console);
982.          set_origin(fg_console);
983.          set_cursor(fg_console);
984.      }
     // 更新当前控制台
985.          void update_screen(int new_console)
986.          {
987.          static int lock = 0;
988.          if (new_console == fg_console || lock)
989.          return;
990.          lock = 1;
991.          kbdsave(new_console);
992.          get_scrmem(fg_console);
993.          fg_console = new_console;
994.          set_scrmem(fg_console);
995.          set_origin(fg_console);
996.          set_cursor(new_console);
997.          lock = 0;
998.          }
999.          // 停止蜂鸣
1000.         static void sysbeepstop(void)
1001.     {
1002.       /* 禁止定时器 2 */
1003.         outb(inb_p(0x61)&0xFC, 0x61);
1004.         }
        // 开通蜂鸣
1005.         static void sysbeep(void)
```

```
1006.    {
1007.      /* 开启定时器 2 */
1008.       outb_p(inb_p(0x61)|3, 0x61);
1009.      /* 发送设置定时器 2 命令 */
1010.    outb_p(0xB6, 0x43);
1011.      /* 设置频率为 750Hz，发送定时值 0x637 */
1012.       outb_p(0x37, 0x42);
1013.       outb(0x06, 0x42);
1014.      /* 蜂鸣时间为 1/8 秒 */
1015.       timer_table[BEEP_TIMER].expires = jiffies + HZ/8;
1016.       timer_table[BEEP_TIMER].fn = sysbeepstop;
1017.       timer_active |= 1<<BEEP_TIMER;
1018.    }
       // 拷贝屏幕，把屏幕内容复制到指定的用户缓冲区 arg 中
1019.       int do_screendump(int arg)
1020.       {
1021.       char *sptr, *buf = (char *)arg;
1022.       int currcons, l;
1023.       verify_area(buf,2+video_num_columns*video_num_lines);
1024.       currcons = get_fs_byte(buf+1);
1025.       if ((currcons<0) || (currcons>NR_CONSOLES))
1026.       return -EIO;
1027.       put_fs_byte((char)(video_num_lines),buf++);
1028.       put_fs_byte((char)(video_num_columns),buf++);
1029.       currcons = (currcons ? currcons-1 : fg_console);
1030.       sptr = (char *) origin;
1031.       for (l=video_num_lines*video_num_columns; l>0 ; l--, sptr++)
1032.       put_fs_byte(*sptr++,buf++);
1033.       return(0);
1034.    }
       // 控制台显示函数，仅用于内核显示函数
1035.    void console_print(const char * b)
1036.       {
1037.       int currcons = fg_console;
1038.       char c;
1039.       if (currcons<0 || currcons>=NR_CONSOLES)
1040.          currcons = 0;
1041.       while (c = *(b++)) {
1042.       if (c == 10) {
1043.       cr(currcons);
1044.       lf(currcons);
1045.       continue;
1046.    }
1047.    if (c == 13) {
1048.       cr(currcons);
1049.       continue;
1050.       }
1051.       if (x>=video_num_columns) {
1052.          x -= video_num_columns;
1053.          pos -= video_size_row;
```

```
1054.            lf(currcons);
1055.        }
1056.        *(char *) pos = c;
1057.            pos++;
1058.        *(char *) pos = attr;
1059.            pos++;
1060.            x++;
1061.        }
1062.        set_cursor(currcons);
1063.    }
```

代码中第 811～931 行定义的函数 con_init 用于初始化控制台中断，其他什么都不做。如果想让屏幕干净，就使用适当的转义字符序列调用 tty_write()函数。读取 setup.s 程序保存的信息，用以确定当前显示器类型，并且设置所有相关参数。

第 869～874 行是让用户知道正在使用哪一类显示驱动程序，它在屏幕的右上角显示描述字符串。程序首先将显示指针 display_ptr 指到屏幕第一行右端差 4 个字符处，然后循环复制字符串中的字符，并且每复制一个字符都空开一个属性字节。

6.4 keyboard.s 文件

该键盘驱动汇编程序主要包括键盘中断处理程序。在英文惯用法中，键被按下用 make 表示，键被松开用 break 表示。

本文件中定义的是键盘驱动的汇编程序，它主要包括键盘中断的处理程序。并支持美语、德语、法语、丹麦语等类型的键盘。keyboard.s 文件的详细代码注释如下：

```
1.          #define KBD_FINNISH
2.          .text    .globl _hard_reset_now
3.          .globl _keyboard_interrupt
4.          .globl _kapplic
5.          .globl _kmode
6.          .globl _kleds
7.          .globl _set_leds
          // 以下这些是用于键盘读操作
          // size 是键盘缓冲区的长度
8.          size    = 1024
9.          head = 4                        // 缓冲区中头指针字段偏移
10.         tail = 8                        // 缓冲区中尾指针字段偏移
11.         proc_list = 12                  // 等待该缓冲队列的进程字段偏移
12.         buf = 16                        // 缓冲区字段偏移
13.         _kapplic:    .byte 0
14.         _kmode:      .byte 0
15.         _kleds:      .byte 2
16.         e0:     .byte 0
          //con_int 是实际的中断处理子程序，用于读键盘扫描码并将其转换成相应的 ascii 字符
          // 键盘中断处理程序入口点
17.         _keyboard_interrupt:
18.         cld
19.         pushl %eax
20.         pushl %ebx
```

```
21.     pushl %ecx
22.     pushl %edx
23.     push %ds
24.     push %es
25.     movl $0x10,%eax                 // 将 ds、es 段寄存器置为内核数据段
26.     mov %ax,%ds
27.     mov %ax,%es
28.     xorl %eax,%eax                  /* eax 中是扫描码 */
29.     inb $0x60,%al                   // 读取扫描码
30.     pushl %eax
31.     inb $0x61,%al                   // 取 PPI 端口 B 状态，其位 7 用于允许/禁止(0/1)键盘
32.     jmp 1f                          // 延迟一会
33.     1:    jmp 1f
34.     1:    orb $0x80,%al             //al 位 7 置位（禁止键盘工作）
35.     jmp 1f                          // 再延迟一会
36.     1:    jmp 1f
37.     1:    outb %al,$0x61            // 使 PPI PB7 位置位
38.     jmp 1f                          // 延迟一会
39.     1:    jmp 1f
40.     1:    andb $0x7F,%al            //al 位 7 复位
41.     outb %al,$0x61                  //PPI PB7 位复位
42.     jmp 1f                          // 延迟一会
43.     1:    jmp 1f
44.     1:    movb $0x20,%al
45.     outb %al,$0x20
46.     popl %eax
47.     movl $1,%ebx
48.     cmpb $0xE0,%al        // 该扫描码是 0xE0 吗？如果是则跳转到设置 end_intr 标志代码处
49.     je end_intr
50.     movl $2,%ebx
51.     cmpb $0xE1,%al        // 扫描码是 0xE1 吗？如果是则跳转到设置 end_intr 标志代码处
52.     je end_intr
53.     sti
54.      call key_table(,%eax,4)        // 调用键处理程序
55.     call _do_keyboard_interrupt
56.     movl $0,%ebx
57.     end_intr:
58.     movb %bl,e0
59.     pop %es
60.     pop %ds
61.     popl %edx
62.     popl %ecx
63.     popl %ebx
64.     popl %eax
65.     iret
66.     put_queue:
67.       pushl %ecx                    // 保存 ecx，edx 内容
68.     pushl %edx                      // 取控制台 tty 结构中读缓冲队列指针
69.     movl _table_list,%edx # read-queue for console
70.     movl head(%edx),%ecx            // 取缓冲队列中头指针
```

```
71.         1:movb %al,buf(%edx,%ecx)        // 将 al 中的字符放入缓冲队列头指针位置处
72.         incl %ecx                        // 头指针前移 1 字节
73.         andl $size-1,%ecx                // 以缓冲区大小调整头指针
74.         cmpl tail(%edx),%ecx
75.         je 3f                            // 如果缓冲区已满，则后面未放入的字符全部丢弃
76.         shrdl $8,%ebx,%eax               // 将 ebx 中 8 位比特位右移 8 位到 eax 中，但 ebx 不变
77.           je 2f                          //若没有字符则跳转
78.           shrl $8,%ebx                   // 将 ebx 中比特位右移 8 位，并跳转到标号 1 继续操作
79.           jmp 1b
80.         2:movl %ecx,head(%edx)           // 若已将所有字符都加入到队列，则保存头指针
81.           movl proc_list(%edx),%ecx
82.           testl %ecx,%ecx                // 检测任务结构指针是否为空
83.           je 3f                          // 无，则跳转
84.           movl $0,(%ecx)                 // 有，则置该进程为可运行就绪状态
85.         3:popl %edx                      // 弹出保留的寄存器并返回
86.         popl %ecx
87.         ret
      // 下面这段代码根据【Ctrl】或【Alt】的扫描码，分别设置模式标志中相应位
88.       ctrl:movb $0x04,%al                // 模式标志_kmode 中左【Ctrl】键对应的比特位为 0x4
89.       jmp 1f
90.         alt: movb $0x10,%al              // 模式标志_kmode 中左【Alt】键对应的比特位为 0x10
91.         1: cmpb $0,e0                    // e0 标志是否置位?
92.           je 2f                          // 不是，则跳转
93.           addb %al,%al                   // 是，则改成置相应右键的标志位
94.         2: orb %al,_kmode                // 设置模式标志_kmode 中对应的比特位
95.           ret
      // 这段代码处理【Ctrl】或【Alt】键松开的扫描码，对应复位模式标志 mode 中的比特位
96.           unctrl:   movb $0x04,%al       // 模式标志_kmode 中左【Ctrl】键对应的比特位为 0x4
97.           jmp 1f
98.         unalt: movb $0x10,%al            // 模式标志_kmode 中左【Alt】键对应的比特位为 0x10
99.         1:cmpb $0,e0                     //  判断 e0 标志是否置位
100.          je 2f                          // 不是，则跳转
101.          addb %al,%al                   // 是，则改成置相应右键的标志位
102.        2:notb %al                       // 复位模式标志_kmode 中对应的比特位
103.        andb %al,_kmode
104.        ret
105.        lshift:
106.        orb $0x01,_kmode                 // 是左【Shift】键按下，设置_kmode 中对应的标志位
107.        ret
108.        unlshift:
109.        andb $0xfe,_kmode                // 是左【Shift】键松开，复位_kmode 中对应的标志位
110.        ret
111.        rshift:
112.        orb $0x02,_kmode                 // 是右【Shift】键按下，设置_kmode 中对应的标志位
113.        ret
114.        unrshift:
115.        andb $0xfd,_kmode                // 是右【shift】键松开，复位_kmode 中对应的标志位
116.        ret
117.        old_leds:
118.        .byte 2
```

```
119.      caps: testb $0x80,_kmode          // 测试模式标志_kmode 中位 7 是否已经置位
120.      jne 1f                            // 若已处于按下状态，则返回
121.      xorb $4,_kleds                    // 翻转_kleds 标志中【Caps Lock】比特位
122.      xorb $0x40,_kmode                 // 翻转_kmode 标志中【Caps】键按下的比特位
123.      orb $0x80,_kmode                  // 设置_kmode 标志中【Caps】键已按下标志位
        // 这段代码根据 leds 标志，开启或关闭 LED 指示器
124.      _set_leds:
125.      movb _kleds,%al
126.      cmpb old_leds,%al
127.      je 1f
128.      movb %al,old_leds
129.      call kb_wait
130.      movb $0xed,%al
131.      outb %al,$0x60
132.      call kb_wait
133.      movb _kleds,%al
134.      outb %al,$0x60
135.      1: ret
136.      uncaps: andb $0x7f,_kmode         //【Caps】键松开，则复位模式标志_kmode 中的对应位（位 7）
137.      ret
138.      scroll:
139.        testb $0x03,_kmode              // 测试模式标志_kmode 中位 2 是否已经置位
140.      je 1f                             // 若已处于按下状态，则返回
141.      call _show_mem
142.      jmp 2f
143.      1: call _show_state
144.      2: xorb $1,_kleds                 //【Scroll】键按下，则翻转_kleds 标志中的对应位（位 0）
145.      jmp _set_leds                     // 根据 leds 标志重新开启或关闭 LED 指示器
146.      num: cmpb $0x01,_kapplic
147.      jne notappl
148.      movw $0x0050,%ax
149.      applkey:
150.      shll $16,%eax
151.      movw $0x4f1b,%ax
152.      xorl %ebx,%ebx
153.      jmp put_queue
154.      notappl:
155.      xorb $2,_kleds
156.      jmp _set_leds
        //这里处理方向键/数字小键盘方向键，检测数字小键盘等
157.      cursor:
158.        subb $0x47,%al                  // 检查扫描码是否是小数字键盘上的键（其扫描码>=0x47）发出的
159.        jb 1f                           // 如果小于则不处理，返回
160.        cmpb $12,%al
161.        ja 1f                           // 扫描码值超过 0x53，不处理，返回
162.        jne cur2                        /* 检查是否按下【Ctrl+Alt+Del】*/
163.        testb $0x0c,_kmode              // 有【Ctrl】键按下吗
164.        je cur2                         // 无，则跳转
165.        testb $0x30,_kmode              // 有【Alt】键按下吗
166.        jne _ctrl_alt_del               // 有，则跳转到_ctrl_alt_del 处理
```

```
167.        cur2:cmpb $0x01,e0              /* e0 置位表示光标移动 */
168.        je cur                          // e0 标志已置位，则跳转光标移动处理处 cur
169.        testb $0x03,_kmode              /* 【Shift】键也使光标移动 */
        // 测试模式标志_kmode 中【Shift】按下标志
170.      jne cur                           // 有，则跳转到 cur 处理
171.      cmpb $0x01,_kapplic
172.      jne notcappl
173.      movb appl_table(%eax),%al
174.      jmp applkey
175.      notcappl:
176.      testb $0x02,_kleds                /* 【Num Lock】键则不许 */
        // 测试_kleds 中标志【Num Lock】键标志是否置位
177.      je cur                            // 如果没有置位，则也进行光标移动处理
178.      xorl %ebx,%ebx                    // 否则查询扫数字表，取对应键的数字 ASCII 码
179.      movb num_table(%eax),%al// 以 eax 作为索引值，取对应数字字符%al
180.      jmp put_queue                     // 将该字符放入缓冲队列中
181.      1:    ret
        // 这段代码处理光标的移动
182.      cur:   movb cur_tablc(%eax),%al   // 取光标字符表中相应键的代表字符%al
183.      cmpb $'9,%al                      // 若该字符<='9'，说明是上一页、下一页、插入或删除键
184.    ja ok_cur                           // 则功能字符序列中要添入字符'~'
185.      movb $'~,%ah
186.      ok_cur: shll $16,%eax             // 将 ax 中内容移到 eax 高字中
187.      movw $0x5b1b,%ax                  // 在 ax 中放入'esc ['字符，与 eax 高字中字符组成移动序列
188.      xorl %ebx,%ebx
189.      cmpb $0x01,_kapplic
190.      jne put_queue
191.      movb $0x4f,%ah
192.      jmp put_queue                     // 将该字符放入缓冲队列中
193.
194.      #if defined(KBD_FR)
195.      num_table:
196.      .ascii "789-456+1230."            // 数字小键盘上键对应的数字 ASCII 码表
197.      #else
198.      num_table:
199.      .ascii "789-456+1230,"
200.      #endif
201.      cur_table:
202.      .ascii "HA5-DGC+YB623"   // 数字小键盘上方向键或插入删除键对应的移动表示字符表
203.      appl_table:
204.    .   ascii "wxyStuvlqrspn"
        // 下面子程序处理功能键
205.    func:
206.     subb $0x3B,%al                    // 功能键'F1'的扫描码是 0x3B
207.       jb end_func                     // 如果扫描码小于 0x3B，则不处理，返回
208.       cmpb $9,%al                     // 判断功能键是不是 F1～F10
209.    jbe ok_func                        // 如果是，则跳转
210.     subb $18,%al                      // 判断是不是功能键 F11，F12
211.       cmpb $10,%al                    // 判断是不是功能键 F11
212.       jb end_func                     // 判断不是，则不处理，直接返回
```

```
213.      cmpb $11,%al                    // 判断是不是功能键 F12
214.      ja end_func                     // 不是，则不处理，直接返回
215.       ok_func:
216.       testb $0x10,_kmode
217.       jne alt_func
218.         cmpl $4,%ecx                  /* 检查是否有足够空间*/
219.      jl end_func                     // 需要放入 4 个字符序列，如果放不下，则返回
220.      movl func_table(,%eax,4),%eax   // 取功能键对应字符序列
221.       xorl %ebx,%ebx
222.         jmp put_queue                 // 放入缓冲队列中
223.      alt_func:
224.      pushl %eax
225.       call _change_console
226.       popl %eax
227.       end_func:
228.       ret
        // 功能键发送的扫描码，F1 键为：'esc [ [ A'，  F2 键为：'esc [ [ B'等
229.      func_table:
230.      .long 0x415b5b1b,0x425b5b1b,0x435b5b1b,0x445b5b1b
231.      .long 0x455b5b1b,0x465b5b1b,0x475b5b1b,0x485b5b1b
232.      .long 0x495b5b1b,0x4a5b5b1b,0x4b5b5b1b,0x4c5b5b1b
        // 扫描码-ASCII 字符映射表
233.      #if defined(KBD_FINNISH)
        // 以下是芬兰语键盘的扫描码映射表
234.       key_map:
235.       .byte 0,27                     // 扫描码 0x00,0x01 对应的 ASCII 码
236.       .ascii "1234567890+'"          // 扫描码 0x02,...0x0c,0x0d 对应的 ASCII 码，以下类似
237.       .byte 127,9
238.       .ascii "qwertyuiop}"
239.       .byte 0,13,0
240.       .ascii "asdfghjkl|{"
241.       .byte 0,0
242.       .ascii "'zxcvbnm,.-"
243.       .byte 0,'*,0,32                /* 扫描码 0x36～0x39 对应的 ASCII 码 */
244.       .fill 16,1,0                   /* 扫描码 0x3A～0x49 对应的 ASCII 码 */
245.       .byte '-,0,0,0,'+              /* 扫描码 0x4A～0x4E 对应的 ASCII 码 */
246.       .byte 0,0,0,0,0,0,0            /* 扫描码 0x4F～0x55 对应的 ASCII 码 */
247.       .byte '<
248.       .fill 10,1,0
        //【Shift】键同时按下时的映射表
249.       shift_map:
250.       .byte 0,27
251.       .ascii "!\"#$%&/()=?`"
252.       .byte 127,9
253.       .ascii "QWERTYUIOP]^"
254.       .byte 13,0
255.       .ascii "ASDFGHJKL\\["
256.       .byte 0,0
257.       .ascii "*ZXCVBNM;:_"
258.       .byte 0,'*,0,32
```

```
259.        .fill 16,1,0
260.            .byte '-,0,0,0,'+
261.            .byte 0,0,0,0,0,0,0
262.            .byte '>
263.        .fill 10,1,0
            // 【Alt】键同时按下时的映射表
264.        alt_map:
265.        .byte 0,0
266.        .ascii "\0@\0$\0\0{[]}\\\0"
267.        .byte 0,0
268.        .byte 0,0,0,0,0,0,0,0,0,0,0
269.        .byte '~,13,0
270.        .byte 0,0,0,0,0,0,0,0,0,0,0
271.        .byte 0,0
272.        .byte 0,0,0,0,0,0,0,0,0,0,0
273.        .byte 0,0,0,0
274.        .fill 16,1,0
275.        .byte 0,0,0,0,0
276.        .byte 0,0,0,0,0,0,0
277.        .byte '|
278.        .fill 10,1,0
            #elif defined(KBD_US)
            // 以下是美式键盘的扫描码映射表
279.        key_map:
280.        .byte 0,27                          // 扫描码 0x00,0x01 对应的 ASCII 码
281.        .ascii "1234567890-="               // 扫描码 0x02,...0x0c,0x0d 对应的 ASCII 码，以下类似
282.        .byte 127,9
283.        .ascii "qwertyuiop[]"
284.        .byte 13,0
285.        .ascii "asdfghjkl;'"
286.        .byte `,0
287.        .ascii "\\zxcvbnm,./"
288.        .byte 0,'*,0,32                     /* 扫描码 0x36～0x39 对应的 ASCII 码 */
289.        .fill 16,1,0                        /* 扫描码 0x3A～0x49 对应的 ASCII 码 */
290.        .byte '-,0,0,0,'+                   /* 扫描码 0x4A～0x4E 对应的 ASCII 码 */
291.        .byte 0,0,0,0,0,0,0                 /* 扫描码 0x4F～0x55 对应的 ASCII 码 */
292.        .byte '<
293.        .fill 10,1,0
294.        shift_map:
295.        .byte 0,27
296.        .ascii "!@#$%^&*()_+"
297.        .byte 127,9
298.        .ascii "QWERTYUIOP{}"
299.        .byte 13,0
300.        .ascii "ASDFGHJKL:\""
301.        .byte '~,0
302.        .ascii "|ZXCVBNM<>?"
303.        .byte 0,'*,0,32
304.        .fill 16,1,0
305.        .byte '-,0,0,0,'+
```

```
306.        .byte 0,0,0,0,0,0,0
307.        .byte '>
308.        .fill 10,1,0
309.        alt_map:
310.        .byte 0,0
311.        .ascii "\0@\0$\0\0{[]}\\\0"
312.        .byte 0,0
313.        .byte 0,0,0,0,0,0,0,0,0,0
314.        .byte '~,13,0
315.        .byte 0,0,0,0,0,0,0,0,0,0
316.        .byte 0,0
317.        .byte 0,0,0,0,0,0,0,0,0,0
318.        .byte 0,0,0,0
319.        .fill 16,1,0
320.        .byte 0,0,0,0,0
321.        .byte 0,0,0,0,0,0,0
322.        .byte '|
323.        .fill 10,1,0
324.        #elif defined(KBD_UK)
        // 英式扩展键盘的扫描码映射表
325.        key_map:
326.        .byte 0,27
327.        .ascii "1234567890-="
328.        .byte 127,9
329.        .ascii "qwertyuiop[]"
330.        .byte 13,0
331.        .ascii "asdfgh;kl;'"
332.        .byte `,0
333.        .ascii "#zxcvbnm,./"
334.        .byte 0,'*,0,32
335.        .fill 16,1,0
336.        .byte '-,0,0,0,'+
337.        .byte 0,0,0,0,0,0,0
338.        .ascii "\\"
339.        .fill 10,1,0
340.        shift_map:
341.        .byte 0,27
342.        .ascii "!\"#$%%&*()_+"
343.        .byte 127,9
344.        .ascii "QWERTYUIOP{}"
345.        .byte 13,0
346.        .ascii "ASDFGHJKL:@"
347.        .byte '~,0
348.        .ascii "~ZXCVBNM<>?"
349.        .byte 0,'*,0,32
350.        .fill 16,1,0
351.        .byte '-,0,0,0,'+
352.        .byte 0,0,0,0,0,0,0
353.        .byte '|
354.        .fill 10,1,0
```

```
355.        alt_map:
356.        .byte 0,0
357.        .ascii "\0@\0$\0\0{[]}\\\0"
358.        .byte 0,0
359.        .byte 0,0,0,0,0,0,0,0,0,0
360.        .byte '~,13,0
361.        .byte 0,0,0,0,0,0,0,0,0,0
362.        .byte 0,0
363.        .byte 0,0,0,0,0,0,0,0,0,0
364.        .byte 0,0,0,0
365.        .fill 16,1,0
366.        .byte 0,0,0,0,0
367.        .byte 0,0,0,0,0,0,0
368.        .byte '|
369.        .fill 10,1,0
370.        #elif defined(KBD_GR)
        // 以下是德语键盘的扫描码映射表
371.        key_map:
372.        .byte 0,27
373.        .ascii "1234567890\\'"
374.        .byte 127,9
375.        .ascii "qwertzuiop@+"
376.        .byte 13,0
377.        .ascii "asdfghjkl[]^"
378.        .byte 0,'#
379.        .ascii "yxcvbnm,.-"
380.        .byte 0,'*,0,32
381.        .fill 16,1,0
382.        .byte '-,0,0,0,'+
383.        .byte 0,0,0,0,0,0,0
384.        .byte '<
385.        .fill 10,1,0
386.        shift_map:
387.        .byte 0,27
388.        .ascii "!\"#$%&/()=?`"
389.        .byte 127,9
390.        .ascii "QWERTZUIOP\\*"
391.        .byte 13,0
392.        .ascii "ASDFGHJKL{}~"
393.        .byte 0,"
394.        .ascii "YXCVBNM;:_"
395.        .byte 0,'*,0,32
396.        .fill 16,1,0
397.        .byte '-,0,0,0,'+
398.        .byte 0,0,0,0,0,0,0
399.        .byte '>
400.        .fill 10,1,0
401.        alt_map:
402.        .byte 0,0
403.        .byte '@,0,0,0,0,0,0,0,0,0,0
```

```
404.        .ascii "\0@\0$\0\0{[]}\\\0"
405.        .byte 0,0
406.        .byte '~,13,0
407.        .byte 0,0,0,0,0,0,0,0,0,0,0
408.        .byte 0,0
409.        .byte 0,0,0,0,0,0,0,0,0,0,0
410.        .byte 0,0,0,0               /* 36-39 */
411.        .fill 16,1,0          /* 3A-49 */
412.        .byte 0,0,0,0,0             /* 4A-4E */
413.        .byte 0,0,0,0,0,0,0         /* 4F-55 */
414.        .byte '|
415.        .fill 10,1,0
416.        #elif defined(KBD_FR)
        // 以下是法语键盘的扫描码映射表
417.        key_map:
418.        .byte 0,27
419.        .ascii "&{\"'(-}_/@)="
420.        .byte 127,9
421.        .ascii "azertyuiop^$"
422.        .byte 13,0
423.        .ascii "qsdfghjklm|"
424.        .byte `,0,42
425.        .ascii "wxcvbn,;:!"
426.        .byte 0,'*,0,32
427.        .fill 16,1,0
428.        .byte '-,0,0,0,'+
429.        .byte 0,0,0,0,0,0,0
430.        .byte '<
431.        .fill 10,1,0
432.        shift_map:
433.        .byte 0,27
434.        .ascii "1234567890]+"
435.        .byte 127,9
436.        .ascii "AZERTYUIOP<>"
437.        .byte 13,0
438.        .ascii "QSDFGHJKLM%"
439.        .byte '~,0,'#
440.        .ascii "WXCVBN?./\\"
441.        .byte 0,'*,0,32
442.        .fill 16,1,0
443.        .byte '-,0,0,0,'+
444.        .byte 0,0,0,0,0,0,0
445.        .byte '>
446.        .fill 10,1,0
447.        alt_map:
448.        .byte 0,0
449.        .ascii "\0~#{[|`\\^@]}"
450.        .byte 0,0
451.        .byte '@,0,0,0,0,0,0,0,0,0,0
452.        .byte '~,13,0
```

```
453.        .byte 0,0,0,0,0,0,0,0,0,0,0,0
454.        .byte 0,0
455.        .byte 0,0,0,0,0,0,0,0,0,0,0,0
456.        .byte 0,0,0,0
457.        .fill 16,1,0
458.        .byte 0,0,0,0,0
459.        .byte 0,0,0,0,0,0,0,0
460.        .byte '|
461.        .fill 10,1,0
462.        #elif defined(KBD_DK)
            // 以下是丹麦语键盘的扫描码映射表
463.        key_map:
464.        .byte 0,27
465.        .ascii "1234567890+'"
466.        .byte 127,9
467.        .ascii "qwertyuiop"
468.        .byte 134,0,13,0
469.        .ascii "asdfghjkl"
470.        .byte 145,155,0,0
471.        .ascii "'zxcvbnm,.-"
472.        .byte 0,'*,0,32
473.        .fill 16,1,0
474.        .byte '-,0,0,0,'+
475.        .byte 0,0,0,0,0,0,0,0
476.        .byte '<
477.        .fill 10,1,0
478.        shift_map:
479.        .byte 0,27
480.        .ascii "!\"#$%&/()=?`"
481.        .byte 127,9
482.        .ascii "QWERTYUIOP"
483.        .byte 143,94,13,0
484.        .ascii "ASDFGHJKL"
485.        .byte 146,157,0,0
486.        .ascii "*ZXCVBNM;:_"
487.        .byte 0,'*,0,32
488.        .fill 16,1,0
489.        .byte '-,0,0,0,'+
490.        .byte 0,0,0,0,0,0,0,0
491.        .byte '>
492.        .fill 10,1,0
493.        alt_map:
494.        .byte 0,0
495.        .ascii "\0@\0$\0\0{[]}\0"
496.        .byte '|,0,0
497.        .byte 0,0,0,0,0,0,0,0,0,0,0,0
498.        .byte '~,13,0
499.        .byte 0,0,0,0,0,0,0,0,0,0,0,0
500.        .byte 0,0
501.        .byte 0,0,0,0,0,0,0,0,0,0,0,0
```

```
502.        .byte 0,0,0,0
503.        .fill 16,1,0
504.        .byte 0,0,0,0,0
505.        .byte 0,0,0,0,0,0,0
506.        .ascii "\\"
507.        .fill 10,1,0
508.        #else
509.        #error "KBD-type not defined"
510.        #endif
          //do_self 用于处理“普通”键，也即含义没有变化，并且只有一个字符返回的键
511.        do_self:
512.       lea alt_map,%ebx                  // 根据_kmode  标志选择alt_map,shift_map,key_map映射表之一
513.       testb $0x20,_kmode
514.        jne 1f
515.        lea shift_map,%ebx
516.        testb $0x03,_kmode
517.        jne 1f
518.        lea key_map,%ebx
519.       1:movb (%ebx,%eax),%al        // 根据扫描码取映射表中对应的 ASCII 字符
520.       orb %al,%al
521.        je none
522.          testb $0x4c,_kmode          /* 控制键已按下或大写锁定亮？ */
523.        je 2f
524.        cmpb $'a,%al
525.       jb 2f
526.       cmpb $'},%al
527.        ja 2f
528.        subb $32,%al
529.        2: testb $0x0c,_kmode         /*【Ctrl】键同时按下了吗？ */
530.        je 3f
531.       cmpb $64,%al
532.       jb 3f
533.        cmpb $64+32,%al
534.        jae 3f
535.        subb $64,%al
536.        3: testb $0x10,_kmode         /* 左【Alt】键同时按下？ */
537.       je 4f
538.       orb $0x80,%al
539.        4: andl $0xff,%eax            // 清 eax 的高字和 ah
540.          xorl %ebx,%ebx              // 清 ebx
541.          call put_queue              // 将字符放入缓冲队列中
542.          none: ret
          //斜线和星号都有自己的规则，正如斜线前面的扫描码'E0h'意味着这里是数字键盘的斜线
543.       slash:   cmpb $1,e0
544.        jne do_self
545.        cmpb $1,_kapplic
546.       jne notmapplic
547.       movw $'Q,%ax
548.        jmp applkey
549.        notmapplic:
```

```
550.        movl $'/,%eax
551.        xorl %ebx,%ebx
552.        jmp put_queue
553.        star:   cmpb $1,_kapplic
554.      jne do_self
555.      movw $'R,%ax
556.        jmp applkey
557.        notsapplic:
558.        movl $'*,%eax
559.        xorl %ebx,%ebx
560.        jmp put_queue
561.        enter: cmpb $1,e0
562.      jne do_self
563.      cmpb $1,_kapplic
564.        jne do_self
565.        movw $'M,%ax
566.        jmp applkey
567.        minus:      cmpb $1,_kapplic
568.        jne do_self
569.      movw $'S,%ax
570.      jmp applkey
571.        plus:  cmpb $1,_kapplic
572.        jne do_self
573.        movw $'l,%ax
574.        jmp applkey
575.        key_table:
576.        .long none,do_self,do_self,do_self
577.        .long do_self,do_self,do_self,do_self
578.        .long do_self,do_self,do_self,do_self
579.        .long do_self,do_self,do_self,do_self
580.        .long do_self,do_self,do_self,do_self
581.        .long do_self,do_self,do_self,do_self
582.        .long do_self,do_self,do_self,do_self
583.        .long enter,ctrl,do_self,do_self
584.        .long do_self,do_self,do_self,do_self
585.        .long do_self,do_self,do_self,do_self
586.        .long do_self,do_self,lshift,do_self
587.        .long do_self,do_self,do_self,do_self
588.        .long do_self,do_self,do_self,do_self
589.        .long do_self,slash,rshift,star
590.        .long alt,do_self,caps,func
591.        .long func,func,func,func
592.        .long func,func,func,func
593.        .long func,num,scroll,cursor
594.        .long cursor,cursor,minus,cursor
595.        .long cursor,cursor,plus,cursor
596.        .long cursor,cursor,cursor,cursor
597.        .long none,none,do_self,func
598.        .long func,none,none,none
599.        .long none,none,none,none
```

```
600.        .long none,none,none,none
601.        .long none,none,none,none
602.        .long none,none,none,none
603.        .long none,none,none,none
604.        .long none,none,none,none
605.        .long none,none,none,none
606.        .long none,none,none,none
607.        .long none,none,none,none
608.        .long none,none,none,none
609.        .long none,none,none,none
610.        .long none,none,none,none
611.        .long none,none,none,none
612.        .long none,none,none,none
613.        .long none,none,none,none
614.        .long none,none,none,none
615.        .long none,unctrl,none,none
616.        .long none,none,none,none
617.        .long none,none,none,none
618.        .long none,none,unlshift,none
619.        .long none,none,none,none
620.        .long none,none,none,none
621.        .long none,none,unrshift,none
622.        .long unalt,none,uncaps,none
623.        .long none,none,none,none
624.        .long none,none,none,none
625.        .long none,none,none,none
626.        .long none,none,none,none
627.        .long none,none,none,none
628.        .long none,none,none,none
629.        .long none,none,none,none
630.        .long none,none,none,none
631.        .long none,none,none,none
632.        .long none,none,none,none
633.        .long none,none,none,none
634.        .long none,none,none,none
635.        .long none,none,none,none
636.        .long none,none,none,none
637.        .long none,none,none,none
638.        .long none,none,none,none
639.        .long none,none,none,none
        //子程序 kb_wait 用于等待键盘控制器缓冲空
640.        kb_wait:
641.        pushl %eax
642.        pushl %ebx
643.        movl $10000,%ebx
644.        1: inb $0x64,%al                    // 读键盘控制器状态
645.          testb $0x02,%al                   // 测试输入缓冲器是否为空
646.       je 2f
647.       decl %ebx
648.         jne 1b
```

```
649.        2: popl %ebx
650.        popl %eax
651.        ret
652.        _hard_reset_now:
653.        sti
654.        movl $100,%ebx
655.       1:call kb_wait                    // 等待键盘控制器输入缓冲器空
656.        movw $0x1234,0x472
657.         movb $0xfe,%al
658.           outb %al,$0x64                // 向系统复位和 A20 线输出负脉冲
659.       decl %ebx
660.       jne 1b
661.        lidt no_idt
662.       jmp _hard_reset_now
```

代码中第 66～70 行的子程序是把 ebx:eax 中的最多 8 个字符添入缓冲队列中。edx 是所写入字符的顺序是 al,ah,eal,eah,bl,bh 直到 eax 等于 0。

第 576～639 行是一张子程序地址跳转表。当取得扫描码后就根据此表调用相应的扫描码处理子程序。大多数调用的子程序是 do_self，或者是 none，这起决于是按键（make）还是释放键（break）。

第 652～662 行的子程序通过设置键盘控制器，向复位线输出负脉冲，使系统复位重启（reboot）。

6.5 pty.c 文件

pty 是伪终端设备，在远程登录等需要以终端方式进行连接，但又并非真实终端的应用程序中必须使用这种设备，如 telnet 或 xterm 等程序。pty.c 文件主要用于完成伪终端设备的读写操作函数的定义。

本文件主要包括主伪终端的写函数 mpty_write()和从伪终端的写函数 spty_write()的定义。使用这些函数可以实现对伪终端的读写操作。pty.c 文件的详细代码注释如下：

```
1.    #include <linux/tty.h>                    // tty 头文件
2.    #include <linux/sched.h>                  // 调度程序头文件
3.    #include <asm/system.h>                   // 系统头文件
4.    #include <asm/io.h>                       // io 头文件
       //用于复制指定长度队列数据为规范式字符序列
5.     static inline void pty_copy(struct tty_struct * from, struct tty_struct * to)
6.     {
7.           char c;
8.           while (!from->stopped && !EMPTY(from->write_q)) {
9.           if (FULL(to->read_q)) {
10.          if (FULL(to->secondary))
11.        break;
12.        copy_to_cooked(to);                 // 复制为规范式字符序列
13.          continue;
14.          }
15.          GETCH(from->write_q,c);
16.          PUTCH(c,to->read_q);
17.      if (current->signal & ~current->blocked)
18.      break;
```

```
19.             }
20.             copy_to_cooked(to);
21.             wake_up(&from->write_q->proc_list);
22.             }
        //下面函数用于主伪终端的写函数
23.             void mpty_write(struct tty_struct * tty)
24.      {
25.             int nr = tty - tty_table;
26.             if ((nr >> 6) != 2)
27.             printk("bad mpty\n\r");
28.             else
29.             pty_copy(tty,tty+64);
30.      }
        //下面函数用于从伪终端的写函数
31.      void spty_write(struct tty_struct * tty)
32.             {
33.             int nr = tty - tty_table;
34.             if ((nr >> 6) != 3)
35.             printk("bad spty\n\r");
36.             else
37.             pty_copy(tty,tty-64);
38.             }
```


pty.c 这个模块是对伪终端 pty 设备功能的补充。代码中第 23～30 行该函数是在 tty_write 函数向 write_queue 缓冲队列写入时被调用的，它将输入复制到输出队列中。

6.6 rs_io.s 文件

该汇编文件用于实现 RS232 串行通信输入输出的中断处理过程。当使用 RS232 串口进行通信时，通信双方的程序需要调用中断处理函数，而这些中断具体的处理函数就是在文件 rs_io.s 文件中被定义的。

本程序的终端处理过程是在进行字符的传输和存储过程中使用，是对终端的读、写缓冲队列进行操作。它把从串行线路上接收到的字符存入串行终端的读缓冲队列 read_q 中，或把写缓冲队列 write_q 中需要发送出去的字符通过串行线路发送给远端的串行终端设备。

rs_io.s 文件的详细代码注释如下：

```
1.      .text
2.      .globl _rs1_interrupt,_rs2_interrupt
        // size 是读写队列缓冲区的字节长度，必须是 2 的次方并且需要与 tty_io.c 中的值匹配
3.      size = 1024                     /* 以下这些是读写缓冲结构中的偏移量 */
4.      rs_addr = 0                     // 串行端口号字段偏移
5.      head = 4                        // 缓冲区中头指针字段偏移
6.      tail = 8                        // 缓冲区中尾指针字段偏移
7.      proc_list = 12                  // 等待该缓冲的进程字段偏移
8.      buf = 16                        // 缓冲区字段偏移
9.
10.      startup = 256
         //这些是实际的中断程序。程序首先检查中断的来源，然后执行相应的处理
11.     .align 2
```

```
12.     _rs1_interrupt:                 // 串行端口 1 中断处理程序入口点
13.     pushl $_table_list+8
14.     jmp rs_int
15.     .align 2
16.     _rs2_interrupt:                 // 串行端口 2 中断处理程序入口点
17.     pushl $_table_list+16
18.     rs_int:
19.     cld
20.     pushl %edx
21.     pushl %ecx
22.     pushl %ebx
23.     pushl %eax
24.     push %es
25.     push %ds
26.     pushl $0x10
27.     pop %ds
28.     pushl $0x10
29.     pop %es
30.     movl 24(%esp),%edx
31.     movl (%edx),%edx
32.     movl rs_addr(%edx),%edx
33.     addl $2,%edx
34.     rep_int:
35.     xorl %eax,%eax
36.     inb %dx,%al
37.     testb $1,%al
38.     jne end
39.     cmpb $6,%al
40.     ja end
41.     movl 24(%esp),%ecx
42.     pushl %edx
43.     subl $2,%edx
44.     call jmp_table(,%eax,2)
45.     popl %edx
46.     jmp rep_int
47.     end: movb $0x20,%al
48.     outb %al,$0x20
49.     pop %ds
50.     pop %es
51.     popl %eax
52.     popl %ebx
53.     popl %ecx
54.     popl %edx
55.     addl $4,%esp
56.     iret
57.     jmp_table:
58.     .long modem_status,write_char,read_char,line_status
59.     .align 2
60.     modem_status:
61.     addl $6,%edx
```

```
62.     inb %dx,%al             /* 通过读 modem 状态寄存器进行复位（0x3fe）*/
63.     ret
64.     .align 2
65.     line_status:
66.         addl $5,%edx
67.         inb %dx,%al         /* 通过读线路状态寄存器进行复位（0x3fd）*/
68.     ret
69.     .align 2
70.     read_char:
71.         inb %dx,%al                         // 读取字符%al
72.     movl %ecx,%edx                          // 当前串口缓冲队列指针地址%edx
73.         subl $_table_list,%edx              // 缓冲队列指针表首址
74.     shrl $3,%edx                            // 差值/8
75.         movl (%ecx),%ecx                    // 取读缓冲队列结构地址%ecx
76.         movl head(%ecx),%ebx                // 取读队列中缓冲头指针%ebx
77.         movb %al,buf(%ecx,%ebx)             // 将字符放在缓冲区中头指针所指的位置
78.         incl %ebx                           // 将头指针前移一字节
79.     andl $size-1,%ebx
80.         cmpl tail(%ecx),%ebx                // 缓冲区头指针与尾指针相比较
81.         je 1f                               // 若相等，表示缓冲区满，跳转到标号 1 处
82.         movl %ebx,head(%ecx)                // 保存修改过的头指针
83.     1:  movl mask_table(,%edx,4),%edx
84.     orl %edx,_timer_active
85.     ret
86.     .align 2
87.     mask_table:
88.     .long 0,4,8
89.     .align 2
90.     write_char:
91.         movl 4(%ecx),%ecx                   // 取写缓冲队列结构地址%ecx
92.         movl head(%ecx),%ebx                // 取写队列头指针%ebx
93.     subl tail(%ecx),%ebx
94.         andl $size-1,%ebx                   // 对指针取模运算
95.         je write_buffer_empty               // 如果头指针 = 尾指针，说明写队列无字
                                                // 符，跳转处理
96.         cmpl $startup,%ebx                  // 检查队列中字符数是否超过 256 个
97.         ja 1f                               // 超过，则跳转处理
98.         movl proc_list(%ecx),%ebx           // 唤醒等待的进程
99.         testl %ebx,%ebx                     // 是否有等待的进程
100.        je 1f                               // 无，则向前跳转到标号 1 处
101.        movl $0,(%ebx)                      // 否则唤醒进程
102.    1:  movl tail(%ecx),%ebx                // 取尾指针
103.        movb buf(%ecx,%ebx),%al             // 从缓冲中尾指针处取一字符%al
104.        outb %al,%dx                        // 向端口 0x3f8(0x2f8)送出到保持寄存器中
105.        incl %ebx                           // 尾指针向前移
106.        andl $size-1,%ebx                   // 尾指针若到缓冲区末端，则返回
107.        movl %ebx,tail(%ecx)                // 保存已修改过的尾指针
108.        cmpl head(%ecx),%ebx                // 比较尾指针与头指针
109.        je write_buffer_empty               // 若相等，表示队列已空，则跳转
110.    ret
```

```
111.    .align 2
112.    write_buffer_empty:
113.        movl proc_list(%ecx),%ebx          // 唤醒等待的进程
114.        testl %ebx,%ebx                    // 有等待的进程吗?
115.        je 1f                              // 无，则向前跳转到标号 1 处
116.        movl $0,(%ebx)                     // 否则将进程置为可运行状态
117.    1:  incl %edx
118.        inb %dx,%al                        // 读取中断允许寄存器
119.        jmp 1f                             // 延迟一会
120.    1:  jmp 1f
121.    1:  andb $0xd,%al
122.        outb %al,%dx                       /* 屏蔽发送保持寄存器空中断*/
123.    ret
```

该模块是实现 RS232 输入输出中断处理的程序。代码中第 56～111 行为各中断类型处理程序地址跳转表，共有 4 种中断来源：modem 状态变化中断、写字符中断、读字符中断、线路状态有问题中断。

6.7 serial.c 文件

本文件实现系统串行端口初始化，为使用串行终端设备作好准备工作。当串行端口初始化完成之后，就可以调用 rs_io.s 中定义的串行化中断处理函数进行串行化通信了。

本程序中函数 rs_init()用于完成初始化操作，它设置了默认的串行通信参数，并设置串行端口的中断陷阱门（中断向量）。rs_write()函数用于把串行终端设备写缓冲队列中的字符通过串行线路发送给远端的终端设备。

rs_write()将在文件系统中用于操作字符设备文件时被调用。当一个程序往串行设备/dev/tty64 文件执行写操作时，就会执行系统调用 sys_write()（在 fs/read_write.c 中），而这个系统调用在判别出所读文件是一个字符设备文件时，即会调用 rw_char()函数（在 fs/char_dev.c 中），该函数则会根据所读设备的子设备号等信息，由字符设备读写函数表（设备开关表）调用 rw_tty()，最终调用到这里的串行终端写操作函数 rs_write()。rs_write()函数实际上只是开启串行发送保持寄存器已空中断标志，在 UART 将数据发送出去后允许发中断信号。具体发送操作是在 rs_io.s 程序中完成。serial.c 文件的详细代码注释如下：

```
1.      #include <linux/tty.h>      // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数
2.      #include <linux/sched.h> // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据
3.  // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句
4.      #include <linux/timer.h> 0.95   // 新加的头函数
5.      #include <asm/system.h>         // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数
6.      #include <asm/io.h>             // io 头文件。定义硬件端口输入/输出宏汇编语句
7.      #define WAKEUP_CHARS (TTY_BUF_SIZE/4)// 当写队列中含有 WAKEUP_CHARS 个字符
                                                // 时，就开始发送
8.      extern void rs1_interrupt(void);            // 串行口 1 的中断处理程序
9.      extern void rs2_interrupt(void);            // 串行口 2 的中断处理程序
10. // 初始化串行端口
11. // port: 串口 1～0x3F8，串口 2～0x2F8
12.     static void com1_timer(void)                // 串行口 1 的定时程序
13.     {
14.         copy_to_cooked(tty_table+64);
```

```
15.     }
16.     static void com2_timer(void)                    // 串行口 2 的定时程序
17.     {
18.         copy_to_cooked(tty_table+65);
19.     }
20.     static inline void do_rs_write(unsigned int port)   //0.95 新加的串口写程序
21.     {
22.         char c;
23.         #define TTY (tty_table[64+port].write_q)
24.         #define TIMER (SER1_TIMEOUT+port)
25.         cli();
26.         if (!EMPTY(TTY)) {
27.          outb_p(inb_p(TTY->data+1)|0x02,TTY->data+1);
28.          if (inb(TTY->data+5) & 0x20) {
29.                 GETCH(TTY,c);
30.                 outb(c,TTY->data);
31.          }
32.          timer_table[TIMER].expires = jiffies + 50;
33.          timer_active |= 1 << TIMER;
34.          } else
35.          timer_active &= ~(1 << TIMER);
36.         sti();
37.         #undef TIMER
38.         #undef TTY
39.     }
40.      static void com1_timeout(void)
41.      {
42.        do_rs_write(0);
43.      }
44.      static void com2_timeout(void)        // 串行口 2 的超时程序
45.      {
46.        do_rs_write(1);
47.      }
48.      static void init(int port)
49.      {
50.      outb_p(0x80,port+3);
51.    /* 设置线路控制寄存器的 DLAB 位（位 7）*/
52.      outb_p(0x30,port);
53.    /* 发送波特率因子低字节，0x30->2400bps */
54.      outb_p(0x00,port+1);
55.    /* 发送波特率因子高字节，0x00 */
56.      outb_p(0x03,port+3);
57.    /* 复位 DLAB 位，数据位为 8 位 */
58.      outb_p(0x0b,port+4);
59.    /* 设置 DTR，RTS，辅助用户输出 2 */
60.      outb_p(0x0d,port+1);
61.    /* 除了写（写保持空）以外，允许所有中断源中断 */
62.      (void)inb(port);
63.    /* 读数据口，以进行复位操作*/
64.      }
```

```
65.  // 初始化串行中断程序和串行接口
66.     void rs_init(void)
67.     {
68.       timer_table[SER1_TIMER].fn = com1_timer;
69.       timer_table[SER1_TIMER].expires = 0;
70.       timer_table[SER2_TIMER].fn = com2_timer;
71.       timer_table[SER2_TIMER].expires = 0;
72.
73.  /* SERx_TIMEOUT 写定时器: 防止串口死锁 */
74.       timer_table[SER1_TIMEOUT].fn = com1_timeout;
75.       timer_table[SER1_TIMEOUT].expires = 0;
76.       timer_table[SER2_TIMEOUT].fn = com2_timeout;
77.       timer_table[SER2_TIMEOUT].expires = 0;
78.       set_intr_gate(0x24,rs1_interrupt);    // 设置串行口 1 的中断门向量（硬件 IRQ4 信号）
79.       set_intr_gate(0x23,rs2_interrupt);    // 设置串行口 2 的中断门向量（硬件 IRQ3 信号）
80.       init(tty_table[1].read_q.data);       // 初始化串行口 1（.data 是端口号）
81.       init(tty_table[2].read_q.data);       // 初始化串行口 2
82.       outb(inb_p(0x21)&0xE7,0x21);          // 允许主 8259A 芯片的 IRQ3，IRQ4 中断信号请求
83.     }
84.     void rs_write(struct tty_struct * tty)
85.     {
86.     cli(); // 关中断。
87.  // 如果写队列不空，则从 0x3f9（或 0x2f9）首先读取中断允许寄存器内容，添上发送保持寄存器中断
88.  // 允许标志（位 1）后，再写回该寄存器。这样就会让串行设备由于要写（发送）字符而引发中断
89.     if (!EMPTY(tty->write_q))
90.       outb(inb_p(tty->write_q.data+1)|0x02,tty->write_q.data+1);
91.       sti(); // 开中断
92.     }
```


serial.c 程序是用于实现 rs232 的输入输出功能以及与传输 IO 有关系的所有中断处理程序。代码中第 50～62 行 SERx_TIMER 接收定时器：超时归 0。第 84～92 行定义的函数 rs_write 是在函数 tty_write 已将数据放入输出（写）队列时才调用。首先检查写队列是否为空，并相应设置中断寄存器。实际上只是开启串行发送保持寄存器已空中断标志，在 UART 将数据发送出去后允许发中断信号。第 89～91 行代码用来判断当写队列不空时，则从 0x3f9（或 0x2f9）首先读取中断允许寄存器内容，添上发送保持寄存器中断允许标志（位 1）后，再写回该寄存器。这样就会让串行设备由于要写（发送）字符而引发中断。

6.8 tty_io.c 文件

每个 tty 设备有 3 个缓冲队列，分别是读缓冲队列（read_q）、写缓冲队列（write_q）和辅助缓冲队列（secondary），定义在 tty_struct 结构中（include/linux/tty.h）。对于每个缓冲队列，读操作是从缓冲队列的左端取字符，并且把缓冲队列尾（tail）指针向右移动。而写操作则是往缓冲队列的右端添加字符，并且也把头（head）指针向右移动。这两个指针中，任何一个若移动到超出了缓冲队列的末端，则折回到左端重新开始。

6.8.1 功能描述

本程序包括字符设备的上层接口函数。主要含有终端读/写函数 tty_read()和 tty_write()。读操

作的行规则函数 copy_to_cooked()也在这里实现。tty_read()和 tty_write()将在文件系统中用于操作字符设备文件时被调用。例如当一个程序读/dev/tty 文件时，就会执行系统调用 sys_read()（在 fs/read_write.c 中），而这个系统调用在判别出所读文件是一个字符设备文件时，即会调用 rw_char()函数（在 fs/char_dev.c 中），该函数则会根据所读设备的子设备号等信息，由字符设备读写函数表（设备开关表）调用 rw_tty()，最终调用到这里的终端读操作函数 tty_read()。

copy_to_cooked()函数由键盘中断过程调用（通过 do_tty_interrupt()），用于根据终端 termios 结构中设置的字符输入/输出标志（例如 INLCR、OUCLC）对 read_q 队列中的字符进行处理，把字符转换成以字符行为单位的规范模式字符序列，并保存在辅助字符缓冲队列（规范模式缓冲队列）（secondary）中，供上述 tty_read()读取。在转换处理期间，若终端的回显标志 L_ECHO 置位，则还会把字符放入写队列 write_q 中，并调用终端写函数把该字符显示在屏幕上。如果是串行终端，那么写函数将是 rs_write()。rs_write()会把串行终端写队列中的字符通过串行线路发送给串行终端，并显示在串行终端的屏幕上。copy_to_cooked()函数最后还将唤醒等待着辅助缓冲队列的进程。函数实现的步骤如下所示：

① 如果读队列空或者辅助队列已经满，则跳转到最后一步（第10 步），否则执行以下操作；

② 从读队列 read_q 的尾指针处取一字符，并且尾指针前移一字符位置；

③ 若是回车（CR）或换行（NL）字符，则根据终端 termios 结构中输入标志（ICRNL、INLCR、INOCR）的状态，对该字符作相应转换。例如，如果读取的是一个回车字符并且 ICRNL 标志是置位的，则把它替换成换行字符；

④ 若大写转小写标志 IUCLC 是置位的，则把字符替换成对应的小写字符；

⑤ 若规范模式标志 ICANON 是置位的，则对该字符进行规范模式处理：

- 若是删行字符（^U），则删除 secondary 中的一行字符（队列头指针后退，直到遇到回车或换行或队列已空为止）；
- 若是擦除字符（^H），则删除 secondary 中头指针处的一个字符，头指针后退一个字符位置；
- 若是停止字符（^S），则设置终端的停止标志 stopped=1；
- 若是开始字符（^Q），则复位终端的停止标志。

⑥ 如果接收键盘信号标志 ISIG 是置位的，则为进程生成对应输入控制字符的信号；

⑦ 如果是行结束字符（例如 NL 或^D），则辅助队列 secondary 的行数统计值 data 增 1；

⑧ 如果本地回显标志是置位的，则把字符也放入写队列 write_q 中，并调用终端写函数在屏幕上显示该字符；

⑨ 把该字符放入辅助队列 secondary 中，返回上面第 1 步继续循环处理读队列中其他字符；

⑩ 最后唤醒睡眠在辅助队列上的进程。在阅读下面程序时不免首先查看一下 include/linux/tty.h 头文件。在该头文件定义了 tty 字符缓冲队列的数据结构以及一些宏操作定义。另外还定义了控制字符的 ASCII 码值。

6.8.2 代码注释

tty_io.c 文件的详细代码注释如下：

```
1.          #include <ctype.h>    // 字符类型头文件。定义了一些有关字符类型判断和转换的宏
2.          #include <errno.h>    // 错误号头文件。包含系统中各种出错号（Linus 从 Minix 中引进的）
3.          #include <signal.h>   // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型
4.        // 下面给出相应信号在信号位图中的对应比特位
5.          #define ALRMMASK (1<<(SIGALRM-1))  // 警告（alarm）信号屏蔽位
```

```
6.      #define KILLMASK (1<<(SIGKILL-1))     // 终止（kill）信号屏蔽位
7.      #define INTMASK (1<<(SIGINT-1))       // 键盘中断（int）信号屏蔽位
8.      #define QUITMASK (1<<(SIGQUIT-1))     // 键盘退出（quit）信号屏蔽位
9.      #define TSTPMASK (1<<(SIGTSTP-1))     // tty 发出的停止进程（tty stop）信号屏蔽位
10.     #include <linux/sched.h>               // 调度程序头文件，定义了任务结构 task_struct、
                                              // 初始任务 0 的数据，还有一些有关描述符参数
11.                                           // 设置和获取的嵌入式汇编函数宏语句
12.     #include <linux/tty.h>        // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数
13.     #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数
14.     #include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏
15.     #define _L_FLAG(tty,f) ((tty)->termios.c_lflag & f)     // 取 termios 结构中的本地模式标志
16.     #define _I_FLAG(tty,f) ((tty)->termios.c_iflag & f)     // 取 termios 结构中的输入模式标志
17.     #define _O_FLAG(tty,f) ((tty)->termios.c_oflag & f)     // 取 termios 结构中的输出模式标志
18.   // 取 termios 结构中本地模式标志集中的一个标志位
19.     #define L_CANON(tty) _L_FLAG((tty),ICANON)      // 取本地模式标志集中规范（熟）
                                                        // 模式标志位
20.     #define L_ISIG(tty) _L_FLAG((tty),ISIG) /       // 取信号标志位
21.     #define L_ECHO(tty) _L_FLAG((tty),ECHO)         // 取回显字符标志位
22.     #define L_ECHOE(tty) _L_FLAG((tty),ECHOE)       // 规范模式时，取回显擦出标志位
23.     #define L_ECHOK(tty) _L_FLAG((tty),ECHOK)       // 规范模式时，取 KILL 擦除当前行
                                                        // 标志位
24.     #define L_ECHOCTL(tty) _L_FLAG((tty),ECHOCTL) // 取回显控制字符标志位
25.     #define L_ECHOKE(tty) _L_FLAG((tty),ECHOKE)     // 规范模式时，取 KILL 擦除行并回
                                                        // 显标志位
26.   // 取 termios 结构中输入模式标志中的一个标志位
27.     #define I_UCLC(tty) _I_FLAG((tty),IUCLC)        // 取输入模式标志集中大写到小写
                                                        // 转换标志位
28.     #define I_NLCR(tty) _I_FLAG((tty),INLCR)        // 取换行符 NL 转回车符 CR 标志位
29.     #define I_CRNL(tty) _I_FLAG((tty),ICRNL)        // 取回车符 CR 转换行符 NL 标志位
30.     #define I_NOCR(tty) _I_FLAG((tty),IGNCR)        // 取忽略回车符 CR 标志位
31.   // 取 termios 结构中输出模式标志中的一个标志位
32.     #define O_POST(tty) _O_FLAG((tty),OPOST)        // 取输出模式标志集中执行输出处
                                                        // 理标志
33.     #define O_NLCR(tty) _O_FLAG((tty),ONLCR)        // 取换行符 NL 转回车换行符 CR-
                                                        // NL 标志
34.     #define O_CRNL(tty) _O_FLAG((tty),OCRNL)        // 取回车符 CR 转换行符 NL 标志
35.     #define O_NLRET(tty) _O_FLAG((tty),ONLRET)      // 取换行符 NL 执行回车功能的标志
36.     #define O_LCUC(tty) _O_FLAG((tty),OLCUC)        // 取小写转大写字符标志
37.   // tty 数据结构的 tty_table 数组。其中包含三个初始化项数据，分别对应控制台、串口终端 1 和
38.   // 串口终端 2 的初始化数据
39.     struct tty_struct tty_table[] = {
40.     {
41.     {ICRNL,                                         /* 将输入的 CR 转换为 NL */
42.     OPOST|ONLCR,                                    /* 将输出的 NL 转 CRNL */
43.     0,                                              // 控制模式标志初始化为 0
44.     ISIG | ICANON | ECHO | ECHOCTL | ECHOKE,        // 本地模式标志
45.     0,                                              // 控制台 termio
46.     INIT_C_CC},                                     // 控制字符数组
47.     0,                                              // 所属初始进程组
48.     0,                                              // 初始停止标志
```

```
49.         con_write, // tty 写函数指针
50.         {0,0,0,0,""},                   // tty 控制台读队列
51.         {0,0,0,0,""},                   // tty 控制台写队列
52.         {0,0,0,0,""}                    // tty 控制台辅助（第二）队列
53.         },{
54.         {0,                             // 输入模式标志。0，无须转换
55.         0,                              // 输出模式标志。0，无须转换
56.         B2400 | CS8,                    // 控制模式标志。波特率 2400bps，8
                                            // 位数据位
57.         0,                              // 本地模式标志 0
58.         0,                              // 行规程 0
59.         INIT_C_CC},                     // 控制字符数组
60.         0,                              // 所属初始进程组
61.         0,                              // 初始停止标志
62.         rs_write,                       // 串口 1 tty 写函数指针
63.         {0x3f8,0,0,0,""},               // 串行终端 1 读缓冲队列
64.         {0x3f8,0,0,0,""},               // 串行终端 1 写缓冲队列
65.         {0,0,0,0,""}                    // 串行终端 1 辅助缓冲队列
66.         },{
67.         {0, /* no translation */        // 输入模式标志。0，无须转换
68.         0, /* no translation */         // 输出模式标志。0，无须转换
69.         B2400 | CS8,                    // 控制模式标志。波特率 2400bps，8 位数据位
70.         0,                              // 本地模式标志 0
71.         0,                              // 行规程 0
72.         INIT_C_CC},                     // 控制字符数组
73.         0,                              // 所属初始进程组
74.         0,                              // 初始停止标志
75.         rs_write,                       // 串口 2 tty 写函数指针
76.         {0x2f8,0,0,0,""}, /* rs 2 */    // 串行终端 2 读缓冲队列
77.         {0x2f8,0,0,0,""},               // 串行终端 2 写缓冲队列
78.         {0,0,0,0,""}                    // 串行终端 2 辅助缓冲队列
79.         }
80.         };
81.         struct tty_queue * table_list[]={
82.         &tty_table[0].read_q, &tty_table[0].write_q,     // 控制台终端读、写缓冲队列地址
83.         &tty_table[1].read_q, &tty_table[1].write_q,     // 串行口 1 终端读、写缓冲队列地址
84.         &tty_table[2].read_q, &tty_table[2].write_q      // 串行口 2 终端读、写缓冲队列地址
85.         };
86.       // tty 终端初始化函数
87.       // 初始化串口终端和控制台终端
88.         void tty_init(void)
89.         {
90.         rs_init();                      // 初始化串行中断程序和串行接口 1
                                            // 和 2
91.         con_init();                     // 初始化控制台终端
92.         }
93.         void tty_intr(struct tty_struct * tty, int mask)
94.         {
95.         int i;
96.         if (tty->pgrp <= 0)
```

```
97.         return;
98.       // 扫描任务数组，向 tty 指明的进程组（前台进程组）中所有进程发送指定的信号
99.         for (i=0;i<NR_TASKS;i++)
100.      // 如果该项任务指针不为空，并且其组号等于 tty 组号，则设置（发送）该任务指定的信号 mask
101.        if (task[i] && task[i]->pgrp==tty->pgrp)
102.        task[i]->signal |= mask;
103.        }
104.        static void sleep_if_empty(struct tty_queue * queue)
105.        {
106.        cli(); // 关中断。
107.      // 若当前进程没有信号要处理并且指定的队列缓冲区空，则让进程进入可中断睡眠状态，并
          // 让队列的进程等待指针指向该进程
108.
109.        while (!current->signal && EMPTY(*queue))
110.        interruptible_sleep_on(&queue->proc_list);
111.        sti(); // 开中断
112.        }
113.      // 若队列缓冲区满则让进程进入可中断的睡眠状态
114.      // 参数：queue - 指定队列的指针
115.      // 进程在往队列缓冲区中写入时调用此函数
116.        static void sleep_if_full(struct tty_queue * queue)
117.        {
118.      // 若队列缓冲区不满，则返回退出
119.        if (!FULL(*queue))
120.        return;
121.        cli(); // 关中断
122.        while (!current->signal && LEFT(*queue)<128)
123.        interruptible_sleep_on(&queue->proc_list);
124.        sti(); // 开中断
125.        }
126.      // 等待按键
127.      // 如果控制台的读队列缓冲区空则让进程进入可中断的睡眠状态
128.        void wait_for_keypress(void)
129.        {
130.        sleep_if_empty(&tty_table[0].secondary);
131.        }
132.        void copy_to_cooked(struct tty_struct * tty)
133.        {
134.        signed char c;
135.      // 如果 tty 的读队列缓冲区不空并且辅助队列缓冲区为空，则循环执行下列代码
136.        while (!EMPTY(tty->read_q) && !FULL(tty->secondary)) {
137.      // 从读队列尾处取一字符到 c，并前移尾指针
138.        GETCH(tty->read_q,c);
139.        if (c==13)
140.        if (I_CRNL(tty))
141.        c=10;
142.        else if (I_NOCR(tty))
143.        continue;
144.        else ;
145.      // 如果该字符是换行符 NL(10)并且换行转回车标志 NLCR 置位，则将其转换为回车符 CR(13)
```

```
146.        else if (c==10 && I_NLCR(tty))
147.        c=13;
148.      // 如果大写转小写标志 UCLC 置位，则将该字符转换为小写字符
149.        if (I_UCLC(tty))
150.        c=tolower(c);
151.      // 如果本地模式标志集中规范（熟）模式标志 CANON 置位，则进行以下处理
152.        if (L_CANON(tty)) {
153.      // 如果该字符是键盘终止控制字符 KILL(^U)，则进行删除输入行上所有字符的处理
154.        if (c==KILL_CHAR(tty)) {
155.        while(!(EMPTY(tty->secondary) ||
156.        (c=LAST(tty->secondary))==10 ||
157.        c==EOF_CHAR(tty))) {
158.        if (L_ECHO(tty)) {
159.        if (c<32)
160.        PUTCH(127,tty->write_q);
161.        PUTCH(127,tty->write_q);
162.        tty->write(tty);
163.        }
164.      // 将 tty 辅助队列头指针后退 1 字节
165.        DEC(tty->secondary.head);
166.        }
167.        continue; // 继续读取并处理其他字符
168.        }
169.      // 判断该字符是否删除控制字符 ERASE(^H)
170.        if (c==ERASE_CHAR(tty)) {
171.        if (EMPTY(tty->secondary) ||
172.        (c=LAST(tty->secondary))==10 ||
173.        c==EOF_CHAR(tty))
174.        continue;
175.        if (L_ECHO(tty)) {
176.        if (c<32)
177.        PUTCH(127,tty->write_q);
178.        PUTCH(127,tty->write_q);
179.        tty->write(tty);
180.        }
181.      // 将 tty 辅助队列头指针后退 1 字节，继续处理其他字符
182.        DEC(tty->secondary.head);
183.        continue;
184.        }
185.      //如果该字符是停止字符（^S），则置 tty 停止标志，继续处理其他字符
186.        if (c==STOP_CHAR(tty)) {
187.        tty->stopped=1;
188.        continue;
189.        }
190.      // 如果该字符是开始字符（^Q），则复位 tty 停止标志，继续处理其他字符
191.        if (c==START_CHAR(tty)) {
192.        tty->stopped=0;
193.        continue;
194.        }
195.        }
```

```
196.        if (L_ISIG(tty)) {
197.      // 如果该字符是键盘中断符（^C），则向当前进程发送键盘中断信号，并继续处理下一字符
198.        if (c==INTR_CHAR(tty)) {
199.        tty_intr(tty,INTMASK);
200.        continue;
201.        }
202.      // 如果该字符是退出符（^\），则向当前进程发送键盘退出信号，并继续处理下一字符
203.        if (c==QUIT_CHAR(tty)) {
204.        tty_intr(tty,QUITMASK);
205.        continue;
206.        }
207.        }
208.      // 如果该字符是换行符 NL(10)，或者是文件结束符 EOF(4，^D)，表示一行字符已处理完，则把
          // 辅助缓冲队列中含有字符行数值增 1。在 tty_read()中若取走一行字符，就会将其值减 1
209.
210.        if (c==10 || c==EOF_CHAR(tty))
211.        tty->secondary.data++;
212.        if (L_ECHO(tty)) {
213.        if (c==10) {
214.        PUTCH(10,tty->write_q);
215.        PUTCH(13,tty->write_q);
216.        } else if (c<32) {
217.        if (L_ECHOCTL(tty)) {
218.        PUTCH('^',tty->write_q);
219.        PUTCH(c+64,tty->write_q);
220.        }
221.        } else
222.        PUTCH(c,tty->write_q);
223.        tty->write(tty);
224.        }
225.      // 将该字符放入辅助队列中
226.        PUTCH(c,tty->secondary);
227.        }
228.      // 唤醒等待该辅助缓冲队列的进程（如果有的话）
229.        wake_up(&tty->secondary.proc_list);
230.        }
231.      // tty 读函数，从终端辅助缓冲队列中读取指定数量的字符，放到用户指定的缓冲区中
232.      // 参数：channel - 子设备号；buf - 用户缓冲区指针；nr - 欲读字节数
233.      // 返回已读字节数
234.        int tty_read(unsigned channel, char * buf, int nr)
235.        {
236.        struct tty_struct * tty;
237.        char c, * b=buf;
238.        int minimum,time,flag=0;
239.        long oldalarm;
240.
241.      // 本版本 Linux 内核的终端只有 3 个子设备，分别是控制台（0）、串口终端 1（1）和串口终端 2（2）
242.      // 所以任何大于 2 的子设备号都是非法的。读的字节数当然也不能小于 0 的
243.        if (channel>2 || nr<0) return -1;
244.      // tty 指针指向子设备号对应 ttb_table 表中的 tty 结构
```

```
245.        tty = &tty_table[channel];
246.        oldalarm = current->alarm;
247.      // 设置读操作超时定时值 time 和需要最少读取的字符个数 minimum
248.        time = 10L*tty->termios.c_cc[VTIME];
249.        minimum = tty->termios.c_cc[VMIN];
250.      // 如果设置了读超时定时值 time 但没有设置最少读取个数 minimum，那么在读到至少一个
          // 字符或者定时超时后读操作将立刻返回。所以这里置 minimum=1
251.
252.        if (time && !minimum) {
253.        minimum=1;
254.      // 如果进程原定时值是 0 或者 time+当前系统时间值小于进程原定时值的话，则置重新设置
          // 进程定时值为 time+当前系统时间，并置 flag 标志
255.
256.        if (flag=(!oldalarm || time+jiffies<oldalarm))
257.        current->alarm = time+jiffies;
258.        }
259.      // 如果设置的最少读取字符数大于欲读的字符数，则令其等于此次欲读取的字符数
260.        if (minimum>nr)
261.        minimum=nr;
262.      // 当欲读的字节数大于 0，则循环执行以下操作
263.        while (nr>0) {
264.      // 如果 flag 不为 0（也即进程原定时值是 0 或者 time+当前系统时间值小于进程原定时值）
          // 并且进程此时以收到定时信号 SIGALRM，表明这里设置的定时时间已到，则复位进程的
          // 定时信号 SIGALRM，并中断循环
265.
266.
267.        if (flag && (current->signal & ALRMMASK)) {
268.        current->signal &= ~ALRMMASK;
269.        break;
270.        }
271.      // 如果 flag 没有置位，或者已置位但当前进程有其他信号要处理，则退出，返回 0
272.        if (current->signal)
273.        break;
274.      // 如果辅助缓冲队列（规范模式队列）为空，或者设置了规范模式标志并且辅助队列中字符
          // 数为 0 以及辅助模式缓冲队列空闲空间大于 20，则进入可中断睡眠状态，返回后继续处理
275.
276.        if (EMPTY(tty->secondary) || (L_CANON(tty) &&
277.        !tty->secondary.data && LEFT(tty->secondary)>20)) {
278.        sleep_if_empty(&tty->secondary);
279.        continue;
280.        }
281.      // 执行以下取字符操作，需读字符数 nr 依次递减，直到 nr=0 或者辅助缓冲队列为空
282.        do {
283.      // 取辅助缓冲队列字符 c，并且缓冲队列 secondary->tail 指针向右移动一个字符位置（tail++）
284.        GETCH(tty->secondary,c);
285.      // 如果该字符是文件结束符（^D）或者是换行符 NL（10），则把辅助缓冲队列中含有字符行数
          // 值减 1
286.        if (c==EOF_CHAR(tty) || c==10)
287.        tty->secondary.data--;
288.      // 如果该字符是文件结束符（^D）并且规范模式标志置位，则返回已读字符数，并退出
```

```
289.        if (c==EOF_CHAR(tty) && L_CANON(tty))
290.        return (b-buf);
291.      // 否则说明是原始模式（非规范模式）操作，于是将该字符直接放入用户数据段缓冲区 buf
          // 中，并把欲读字符数减 1。此时如果欲读字符数已为 0，则中断循环
292.
293.        else {
294.        put_fs_byte(c,b++);
295.        if (!--nr)
296.        break;
297.        }
298.        } while (nr>0 && !EMPTY(tty->secondary));
299.      // 如果超时定时值 time 不为 0 并且规范模式标志没有置位(非规范模式)，那么：
300.        if (time && !L_CANON(tty))
301.        if (flag=(!oldalarm || time+jiffies<oldalarm))
302.        current->alarm = time+jiffies;
303.        else
304.        current->alarm = oldalarm;
305.      // 如果规范模式标志置位，那么若已读到起码一个字符则中断循环。否则若已读取数大于或
          // 等于最少要求读取的字符数，则也中断循环
306.
307.        If (L_CANON(tty)) {
308.        if (b-buf)
309.        break;
310.        } else if (b-buf >= minimum)
311.        break;
312.        }
313.      // 读取 tty 字符循环操作结束，让进程的定时值恢复值
314.        current->alarm = oldalarm;
315.      // 如果进程有信号并且没有读取到任何字符，则返回出错号（被中断）
316.        if (current->signal && !(b-buf))
317.        return -EINTR;
318.        return (b-buf); // 返回已读取的字符数
319.        }
320.      //// tty 写函数。把用户缓冲区中的字符写入 tty 的写队列中
321.      // 参数：channel - 子设备号；buf - 缓冲区指针；nr - 写字节数
322.      // 返回已写字节数
323.        int tty_write(unsigned channel, char * buf, int nr)
324.        {
325.        static cr_flag=0;
326.        struct tty_struct * tty;
327.        char c, *b=buf;
328.      // 本版本 Linux 内核的终端只有 3 个子设备，分别是控制台（0）、串口终端 1（1）和串口
          // 终端 2（2）所以任何大于 2 的子设备号都是非法的。写的字节数当然也不能小于 0
329.
330.        if (channel>2 || nr<0) return -1;
331.      // tty 指针指向子设备号对应 ttb_table 表中的 tty 结构
332.        tty = channel + tty_table;
333.      // 字符设备是一个一个字符进行处理的，所以这里对于 nr 大于 0 时对每个字符进行循环处理
334.        while (nr>0) {
335.      // 如果此时 tty 的写队列已满，则当前进程进入可中断的睡眠状态
```

```
336.        sleep_if_full(&tty->write_q);
337.      // 如果当前进程有信号要处理，则退出，返回 0
338.        if (current->signal)
339.        break;
340.      // 当要写的字节数>0 并且 tty 的写队列不满时，循环执行以下操作
341.        while (nr>0 && !FULL(tty->write_q)) {
342.      // 从用户数据段内存中取一字节 c
343.        c=get_fs_byte(b);
344.      // 如果终端输出模式标志集中的执行输出处理标志 OPOST 置位，则执行下列输出时处理过程
345.        if (O_POST(tty)) {
346.      // 如果该字符是回车符'\r'(CR，13)并且回车符转换行符标志 OCRNL 置位，则将该字符换成
          // 换行符'\n'(NL，10)；否则如果该字符是换行符'\n'(NL，10)并且换行转回车功能标志 ONLRET
          // 置位的话，则将该字符换成回车符'\r'(CR，13)
347.
348.
349.        if (c=='\r' && O_CRNL(tty))
350.        c='\n';
351.        else if (c=='\n' && O_NLRET(tty))
352.        c='\r';
353.      // 如果该字符是换行符'\n'并且回车标志 cr_flag 没有置位，换行转回车-换行标志 ONLCR 置
      位的话，则将 cr_flag 置位，并将一回车符放入写队列中。然后继续处理下一个字符
354.      //
355.        if (c=='\n' && !cr_flag && O_NLCR(tty)) {
356.        cr_flag = 1;
357.        PUTCH(13,tty->write_q);
358.        continue;
359.        }
360.      // 如果小写转大写标志 OLCUC 置位的话，就将该字符转成大写字符
361.        if (O_LCUC(tty))
362.        c=toupper(c);
363.        }
364.      // 用户数据缓冲指针 b 前进 1 字节；欲写字节数减 1 字节；复位 cr_flag 标志，并将该字节
      放入 tty 写队列中
365.      //
366.        b++; nr--;
367.        cr_flag = 0;
368.        PUTCH(c,tty->write_q);
369.        }
370.      // 若字节全部写完，或者写队列已满，则程序执行到这里。调用对应 tty 的写函数，若还有
      字节要写，则等待写队列不满，所以调用调度程序，先去执行其他任务
371.      //
372.        tty->write(tty);
373.        if (nr>0)
374.        schedule();
375.        }
376.        return (b-buf); // 返回写入的字节数
377.        }
378.        void do_tty_interrupt(int tty)
379.        {
380.        copy_to_cooked(tty_table+tty);
```

```
381.        }
382.    //// 字符设备初始化函数。空，为以后扩展做准备
383.        void chr_dev_init(void)
384.        {
385.        }
```

源代码第 23～24 行修改了传输速率，其中参数：tty - 终端对应的 tty 数据结构。

源代码第 38～39 行描述了刷新 tty 缓冲队列，其中参数：gueue - 指定的缓冲队列指针，它令缓冲队列的头指针等于尾指针，从而达到清空缓冲区(零字符)的目的。

源代码第 55～56 行描述了取终端 termios 结构信息，其中参数：tty - 指定终端的 tty 结构指针；termios - 用户数据区 termios 结构缓冲区指针，其返回值为 0。

6.9 tty_ioctl.c 文件

该文件主要用于完成字符设备的控制操作，实现了系统调用函数 tty_ioctl()。通过使用该函数可以修改指定终端 termios 结构中的设置标志等信息。

本程序实现了 tty 的 I/O 控制接口函数 tty_ioctl()以及对 termios 终端 I/O 结构的读写函数，并会在实现系统调用 sys_ioctl()的 fs/ioctl.c 程序中被调用。

tty_ioctl.c 文件的详细代码注释如下：

```
1.      #include <errno.h>          // 错误号头文件。包含系统中各种出错号（Linus 从 Minix 中引进的）
2.      #include <termios.h>        // 终端输入输出函数头文件。主要定义控制异步通信口的终端接口
3.      #include <linux/sched.h>    // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                                    // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句
4.      #include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原形定义
5.      #include <linux/tty.h>      // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数
6.      #include <asm/io.h>         // io 头文件，定义硬件端口输入/输出宏汇编语句
7.      #include <asm/segment.h>    // 段操作头文件，定义了有关段寄存器操作的嵌入式汇编函数
8.      #include <asm/system.h>     // 系统头文件，定义了设置或修改描述符/中断门等的嵌入式汇编宏
    // 这是波特率因子数组（或称为除数数组）
9.      static unsigned short quotient[] = {
10.     0, 2304, 1536, 1047, 857,
11.     768, 576, 384, 192, 96,
12.     64, 48, 24, 12, 6, 3
13.     };
14.     static void change_speed(struct tty_struct * tty)
15.     {
16.     unsigned short port,quot;
    // 对于串口终端，其 tty 结构的读缓冲队列 data 字段存放的是串行端口号（0x3f8 或 0x2f8）
17.     if (!(port = tty->read_q.data))
18.     return;
    // 从 tty 的 termios 结构控制模式标志集中取得设置的波特率索引号，据此从波特率因子数组中
    // 取得对应的波特率因子值。CBAUD 是控制模式标志集中波特率位屏蔽码
19.     quot = quotient[tty->termios.c_cflag & CBAUD];
20.     cli(); // 关中断
21.     outb_p(0x80,port+3);                    // 首先设置除数锁定标志 DLAB
22.     outb_p(quot & 0xff,port);               // 输出因子低字节
23.     outb_p(quot >> 8,port+1);               // 输出因子高字节
24.     outb(0x03,port+3);                      // 复位 DLAB
```

```
25.     sti(); // 开中断。
26.     }
27.     static void flush(struct tty_queue * queue)
28.     {
29.     cli();
30.     queue->head = queue->tail;
31.     sti();
32.     }
    // 等待字符发送出去
33.     static void wait_until_sent(struct tty_struct * tty)
34.     {
35.     /* 什么都没做 - 还未实现 */
36.     }
    // 发送 BREAK 控制符
37.     static void send_break(struct tty_struct * tty)
38.     {
39.     /* 什么都没做 - 还未实现 */
40.     }
41.     static int do_get_ps_info(int arg)              // 取得每秒钟的串口传输信息
42.     {
43.         struct tstruct {                            // 0.95 新加的数据结构 tstruct
44.                 int flag;
45.                 int present[NR_TASKS];
46.                 struct task_struct tasks[NR_TASKS];
47.         };
48.         struct tstruct *ts = (struct tstruct *)arg;
49.         struct task_struct **p;
50.         char *c, *d;
51.         int i, n = 0;
52.
53.         verify_area((void *)arg, sizeof(struct tstruct));
54.
55.         for (p = &FIRST_TASK ; p <= &LAST_TASK ; p++, n++)
56.                 if (*p)
57.                 {
58.                         c = (char *)(*p);
59.                         d = (char *)(ts->tasks+n);
60.                         for (i=0 ; i<sizeof(struct task_struct) ; i++)
61.                                 put_fs_byte(*c++, d++);
62.                         put_fs_long(1, (unsigned long *)(ts->present+n));
63.                 }
64.                 else
65.                         put_fs_long(0, (unsigned long *)(ts->present+n));
66.         return(0);
67.     }
68.     static int get_termios(struct tty_struct * tty, struct termios * termios)
69.     {
70.     int i;
    // 首先验证一下用户的缓冲区指针所指内存区是否足够，如不够则分配内存
71.     verify_area(termios, sizeof (*termios));
```

```
    // 复制指定 tty 结构中的 termios 结构信息到用户 termios 结构缓冲区
72.     for (i=0 ; i< (sizeof (*termios)) ; i++)
73.     put_fs_byte( ((char *)&tty->termios)[i] , i+(char *)termios );
74.     return 0;
75.     }
    // 设置终端 termios 结构信息
    // 参数：tty - 指定终端的 tty 结构指针；termios - 用户数据区 termios 结构指针
    // 返回 0
76.     static int set_termios(struct tty_struct * tty, struct termios * termios)
77.     {
78.     int i;
    // 首先复制用户数据区中 termios 结构信息到指定 tty 结构中
79.     for (i=0 ; i< (sizeof (*termios)) ; i++)
80.     ((char *)&tty->termios)[i]=get_fs_byte(i+(char *)termios);
    // 用户有可能已修改了 tty 的串行口传输波特率，所以根据 termios 结构中的控制模式标志
81.   // c_cflag 修改串行芯片 UART 的传输波特率
82.     change_speed(tty);
83.     return 0;
84.     }
    // 读取 termio 结构中的信息
    // 参数：tty - 指定终端的 tty 结构指针；termio - 用户数据区 termio 结构缓冲区指针
    // 返回 0
85.     static int get_termio(struct tty_struct * tty, struct termio * termio)
86.     {
87.     int i;
88.     struct termio tmp_termio;
    // 首先验证一下用户的缓冲区指针所指内存区是否足够，如不够则分配内存
89.     verify_area(termio, sizeof (*termio));
    // 将 termios 结构的信息复制到 termio 结构中。目的是为了其中模式标志集的类型进行转换，
    // 也即从 termios 的长整数类型转换为 termio 的短整数类型
90.     tmp_termio.c_iflag = tty->termios.c_iflag;
91.     tmp_termio.c_oflag = tty->termios.c_oflag;
92.     tmp_termio.c_cflag = tty->termios.c_cflag;
93.     tmp_termio.c_lflag = tty->termios.c_lflag;
    // 两种结构的 c_line 和 c_cc[]字段是完全相同的
94.     tmp_termio.c_line = tty->termios.c_line;
95.     for(i=0 ; i < NCC ; i++)
96.     tmp_termio.c_cc[i] = tty->termios.c_cc[i];
    // 最后复制指定 tty 结构中的 termio 结构信息到用户 termio 结构缓冲区
97.     for (i=0 ; i< (sizeof (*termio)) ; i++)
98.     put_fs_byte( ((char *)&tmp_termio)[i] , i+(char *)termio );
99.     return 0;
100.    }
    /*
     * 下面的 termio 设置函数仅在 386 低字节在前的方式下可用
     */
    // 设置终端 termio 结构信息
    // 参数：tty - 指定终端的 tty 结构指针；termio - 用户数据区 termio 结构指针
    // 将用户缓冲区 termio 的信息复制到终端的 termios 结构中。返回 0
101.    static int set_termio(struct tty_struct * tty, struct termio * termio)
```

```
102.    {
103.    int i;
104.    struct termio tmp_termio;
    // 首先复制用户数据区中 termio 结构信息到临时 termio 结构中
105.    for (i=0 ; i< (sizeof (*termio)) ; i++)
106.    ((char *)&tmp_termio)[i]=get_fs_byte(i+(char *)termio);
    //再将 termio 结构的信息复制到 tty 的 termios 结构中。目的是为了其中模式标志集的类型进行转换
    // 也即从 termio 的短整数类型转换成 termios 的长整数类型
107.    *(unsigned short *)&tty->termios.c_iflag = tmp_termio.c_iflag;
108.    *(unsigned short *)&tty->termios.c_oflag = tmp_termio.c_oflag;
109.    *(unsigned short *)&tty->termios.c_cflag = tmp_termio.c_cflag;
110.    *(unsigned short *)&tty->termios.c_lflag = tmp_termio.c_lflag;
    // 两种结构的 c_line 和 c_cc[]字段是完全相同的。
111.    tty->termios.c_line = tmp_termio.c_line;
112.    for(i=0 ; i < NCC ; i++)
113.    tty->termios.c_cc[i] = tmp_termio.c_cc[i];
    // 用户可能已修改了 tty 的串行口传输波特率，所以根据 termios 结构中的控制模式标志集 c_cflag
    // 修改串行芯片 UART 的传输波特率
114.    change_speed(tty);
115.    return 0;
116.    }
    // tty 终端设备的 ioctl 函数
    // 参数：dev - 设备号；cmd - ioctl 命令；arg - 操作参数指针
117.    static int set_window_size(struct tty_struct * tty, struct winsize * ws)
    // 设置窗体大小
118.    {
119.        int i,changed;
120.        char c, * tmp;
121.        if (!ws)
122.            return -EINVAL;
123.        tmp = (char *) &tty->winsize;
124.        changed = 0;
125.        for (i = 0; i < sizeof (*ws) ; i++,tmp++) {
126.            c = get_fs_byte(i + (char *) ws);
127.            if (c == *tmp)
128.                continue;
129.            changed = 1;
130.            *tmp = c;
131.        }
132.        if (changed)
133.            kill_pg(tty->pgrp, SIGWINCH, 1);
134.        return 0;
135.    }
136.    static int get_window_size(struct tty_struct * tty, struct winsize * ws)
    // 取得得窗体的大小
137.    {
138.        int i;
139.        char * tmp;
140.        if (!ws)
141.            return -EINVAL;
```

```
142.        verify_area(ws, sizeof (*ws));
143.        tmp = (char *) ws;
144.        for (i = 0; i < sizeof (struct winsize) ; i++,tmp++)
145.                put_fs_byte(((char *) &tty->winsize)[i], tmp);
146.        return 0;
147.    }
148.    int tty_ioctl(int dev, int cmd, int arg)
149.    {
150.    struct tty_struct * tty;
      // 首先取 tty 的子设备号。如果主设备号是 5(tty 终端)，则进程的 tty 字段即是子设备号；如果
      // 进程的 tty 子设备号是负数，表明该进程没有控制终端，也即不能发出该 ioctl 调用，出错死机
151.    if (MAJOR(dev) == 5) {
152.    dev=current->tty;
153.    if (dev<0)
154.    panic("tty_ioctl: dev<0");
      // 否则直接从设备号中取出子设备号
155.    } else
156.    dev=MINOR(dev);
      // 子设备号可以是 0（控制台终端）、1（串口 1 终端）、2（串口 2 终端）
      // 让 tty 指向对应子设备号的 tty 结构
157.    tty = dev + tty_table;
      // 根据 tty 的 ioctl 命令进行分别处理
158.    switch (cmd) {
159.    case TCGETS:
     //取相应终端 termios 结构中的信息
160.    return get_termios(tty,(struct termios *) arg);
161.    case TCSETSF:
     // 在设置 termios 的信息之前，需要先等待输出队列中所有数据处理完，并且刷新（清空）输入
     // 队列再设置
162.    flush(&tty->read_q);
163.    case TCSETSW:
     // 在设置终端 termios 的信息之前，需要先等待输出队列中所有数据处理完（耗尽）。对于修改参数
     // 会影响输出的情况，就需要使用这种形式
164.    wait_until_sent(tty);
165.    case TCSETS:
     // 设置相应终端 termios 结构中的信息
166.    return set_termios(tty,(struct termios *) arg);
167.    case TCGETA:
     // 取相应终端 termio 结构中的信息
168.    return get_termio(tty,(struct termio *) arg);
169.    case TCSETAF:
     // 在设置 termio 的信息之前，需要先等待输出队列中所有数据处理完，并且刷新（清空）输入
     // 队列再设置
170.    flush(&tty->read_q);
171.    case TCSETAW:
     // 在设置终端 termio 的信息之前，需要先等待输出队列中所有数据处理完（耗尽）。对于修改参
     // 数会影响输出的情况，就需要使用这种形式
172.    wait_until_sent(tty); /* 继续执行 */
173.    case TCSETA:
     // 设置相应终端 termio 结构中的信息
```

```
174.      return set_termio(tty,(struct termio *) arg);
175.      case TCSBRK:
      // 等待输出队列处理完毕（空），如果参数值是 0，则发送一个 break
176.      if (!arg) {
177.      wait_until_sent(tty);
178.      send_break(tty);
179.      }
180.      return 0;
181.      case TCXONC:
      // 开始/停止控制。如果参数值是 0，则挂起输出；如果是 1，则重新开启挂起的输出；如果是 2，
      // 则挂起输入；如果是 3，则重新开启挂起的输入
182.
183.      return -EINVAL; /* 未实现 */
184.      case TCFLSH:
      // 刷新已写输出但还没发送，或已收但还没有读数据。如果参数是 0，则刷新（清空）输入队列；
      // 如果是 1，则刷新输出队列；如果是 2，则刷新输入和输出队列
185.      if (arg==0)
186.      flush(&tty->read_q);
187.      else if (arg==1)
188.      flush(&tty->write_q);
189.      else if (arg==2) {
190.      flush(&tty->read_q);
191.      flush(&tty->write_q);
192.      } else
193.      return -EINVAL;
194.      return 0;
195.      case TIOCEXCL:
      // 设置终端串行线路专用模式
196.      return -EINVAL; /* 未实现 */
197.      case TIOCNXCL:
      // 复位终端串行线路专用模式。
198.      return -EINVAL; /* 未实现 */
199.      case TIOCSCTTY:
      // 设置 tty 为控制终端。（TIOCNOTTY - 禁止 tty 为控制终端）
200.      return -EINVAL; /* 设置控制终端 NI */
201.      case TIOCGPGRP: // NI - Not Implemented。
      // 读取指定终端设备进程的组 id。首先验证用户缓冲区长度，然后复制 tty 的 pgrp 字段到用户缓冲区
202.      verify_area((void *) arg,4);
203.      put_fs_long(tty->pgrp,(unsigned long *) arg);
204.      return 0;
205.      case TIOCSPGRP:
      // 设置指定终端设备进程的组 id
206.      tty->pgrp=get_fs_long((unsigned long *) arg);
207.      return 0;
208.      case TIOCOUTQ:
      // 返回输出队列中还未送出的字符数。首先验证用户缓冲区长度，然后复制队列中字符数给用户
209.      verify_area((void *) arg,4);
210.      put_fs_long(CHARS(tty->write_q),(unsigned long *) arg);
211.      return 0;
212.      case TIOCINQ:
```

```
        // 返回输入队列中还未读取的字符数。首先验证用户缓冲区长度，然后复制队列中字符数给用户
213.    verify_area((void *) arg,4);
214.    put_fs_long(CHARS(tty->secondary),
215.    (unsigned long *) arg);
216.    return 0;
217.    case TIOCSTI:
        // 模拟终端输入。该命令以一个指向字符的指针作为参数，并假装该字符是在终端上输入的。用
        // 户必须在该控制终端上具有超级用户权限或具有读许可权限
218.    return -EINVAL; /* 未实现 */
219.    case TIOCGWINSZ:
        // 读取终端设备窗口大小信息（参见 termios.h 中的 winsize 结构）
220.    return -EINVAL; /* 未实现 */
221.    case TIOCSWINSZ:
222.    // 设置终端设备窗口大小信息（参见 winsize 结构）
223.    return -EINVAL; /* 未实现 */
224.    case TIOCMGET:
        // 返回 modem 状态控制引线的当前状态比特位标志集
225.    return -EINVAL; /* 未实现 */
226.    case TIOCMBIS:
        // 设置单个 modem 状态控制引线的状态（true 或 false）
227.    return -EINVAL; /* 未实现 */
228.    case TIOCMBIC:
        // 复位单个 modem 状态控制引线的状态
229.    return -EINVAL; /* 未实现 */
230.    case TIOCMSET:
        // 设置 modem 状态引线的状态。如果某一比特位置位，则 modem 对应的状态引线将置为有效
231.    return -EINVAL; /* 未实现 */
232.    case TIOCGSOFTCAR:
        // 读取软件载波检测标志（1 - 开启；0 - 关闭）
233.    return -EINVAL; /* 未实现 */
234.    case TIOCSSOFTCAR:
        // 设置软件载波检测标志(1 - 开启；0 - 关闭)
235.    return -EINVAL; /* 未实现 */
236.            case TIOCLINUX:
237.                switch (get_fs_byte((char *)arg))
238.                {
239.                    case 0:
240.                        return do_screendump(arg);
241.                    case 1:
242.                        return do_get_ps_info(arg);
243.                    default:
244.                        return -EINVAL;
245.                }
246.    default:
247.    return -EINVAL;
248.    }
249.    }
```

源代码第 9 行修改了传输速率，其中参数：tty - 终端对应的 tty 数据结构。源代码第 27～31 行描述了刷新 tty 缓冲队列，其中参数：gueue - 指定的缓冲队列指针，它令缓冲队列的头指

针等于尾指针，从而达到清空缓冲区（零字符）的目的。源代码第 68～75 行描述了取终端 termios 结构信息，其中参数：tty - 指定终端的 tty 结构指针；termios - 用户数据区 termios 结构缓冲区指针，其返回值为 0。

第 7 章　块设备驱动程序

本章主要讨论的是块设备（如各种磁盘）的驱动程序，在 0.95 版的 Linux 内核中主要支持的块设备有硬盘和软盘驱动器两种。其中所涉及的源代码文件如表 7-1 所示。

表 7-1　linux/kernel/blk_drv 目录

名　　称	大　小（B）	修 改 时 间
blk.h	4033	1992-03-04 12:31:58
floppy.c	13191	1992-03-08 07:08:32
hd.c	8829	1992-03-05 17:55:50
ll_rw_blk.c	5694	1992-03-06 09:04:44
Makefile	2783	1992-03-04 12:31:58
ramdisk.c	2782	1992-02-20 19:15:48

7.1　总体功能

在 Linux 内核中，主要是通过中断处理程序来实现对块设备数据的读写操作。每次读写数据都是以块为单位，在处理过程中，使用读写请求项等待队列来对多个逻辑块进行操作。

当程序读取硬盘上的一个逻辑块时，首先会向缓冲区管理程序提出申请。缓冲区管理程序将查看缓冲区中以前是否已经读取过这块数据。如果已经有了，就直接将对应的缓冲区块头指针返回给程序并唤醒该程序进程。若不存在所要求的数据块，则缓冲管理程序就会调用函数 ll_rw_block()，向相应的块设备驱动程序发出一个读数据块的操作请求。当对应的块设备的请求项队列空时，表明此刻该块设备不忙。于是内核就会立刻向该块设备的控制器发出读数据命令。当块设备的控制器将数据读入到指定的缓冲块中后，就会发出中断请求信号，并调用相应的读命令后处理函数，处理继续读扇区操作或者结束本次请求项的过程。

7.1.1　Linux 设备管理

Linux 内核中，管理硬件设备控制器的代码并没有放在每个应用程序中，而是由内核统一管理。这些处理和管理硬件控制器的软件就是设备驱动程序。Linux 把外部设备主要分为两类：字符设备和块设备。

字符设备指那些无需缓冲区可直接读写的设备，如鼠标、键盘、串口设备等，对它们读写数据直接通过读写寄存器就可实现。字符终端（/dev/console）和串口（/dev/ttyS0 以及类似设备）就是字符设备的两个例子，它们能够用流抽象很好地表示。字符设备可以通过文件系统结点（如 /dev/tyl 和/dev/lp0 等）来访问，它和普通文件之间的唯一差别在于，对普通文件的访问可以前后移动访问的指针，而大多数字符设备只能顺序访问数据。

块设备则仅能以块为单位对它们进行读写，如硬盘、光驱、软驱等。典型的块大小为 512 字节或 1024 字节。块设备的存取是通过 buffercache 来完成的，并且可以进行随机访问，即不管块位于设备的何处都可以对其进行读写。和字符设备一样，块设备也是通过/dev 目录下的文件系统结点被访问的。块设备（例如磁盘）上能够容纳文件系统。在大多数 Linux 系统中，块设备包含整数个块，而每块包含 1KB 或 2 的几次方字节的数据。

Linux 允许应用程序像字符设备那样读写块设备，可以一次传送任意多字节的数据。因而，块设备和字符设备的区别仅仅在于内核内部管理数据的方式，也就是内核和驱动程序的接口不同。像字符设备一样，块设备也是通过文件系统的结点被访问的，它们之间的差异对用户来讲是透明的。块设备驱动除了给内核提供和字符设备一样的接口以外，还提供了专门面向块设备的接口，不过这些接口对于那些从/dev 目录下某个目录项打开块设备的用户应用程序都是不可见的。另外，块设备的接口必须支持挂接（mount）文件系统。

为了便于扩展，Linux 把块设备表建成了一个以主设备号为索引的数组。在 Linux 0.95 中，主设备号有 7 种，如表 7-2 所示。其中，主设备号 1、2 和 3 分别对应块设备：虚拟盘、软盘和硬盘。在块设备数组中其他各项都被默认地置成 NULL。

表 7-2　Linux 0.11 内核中的主设备号

主 设 备 号	类　　型	说　　明
0	无	无
1	块/字符	虚拟 ram，内存设备
2	块	软驱设备 fd
3	块	硬盘设备 hd
4	字符	ttyx 设备
5	字符	tty 设备
6	字符	lp 打印机设备

Linux 内核与设备驱动之间定义由一个标准方式进行互操作的接口。每一类设备驱动都提供了通用接口以便在需要时为内核提供服务。这种通用接口使得内核可以以相同的方式来对待不同的设备及设备驱动。设备驱动的一个基本特征是设备处理的抽象概念。所有硬件设备都被看成普通文件，可以通过和操作普通文件相同的标准系统调用来打开、关闭、读取和写入设备。系统中每个设备都用一种特殊的设备相关文件来表示，例如系统中的一个 IDE 硬盘被表示成/dev/had。

块设备和字符设备的设备文件可以通过 mknod 命令来创建，并使用主从设备号来描述次设备。由同一个设备驱动控制的所有设备具有相同的主设备号。从设备号则被用来区分具有相同主设备号并且由相同设备驱动控制的不同设备。例如主 IDE 硬盘的每个分区的从设备号都不相同。如/dev/hda2 表示主 IDE 硬盘的主设备号为 3 而从设备号为 2。Linux 通过使用主从设备号将包含在系统调用中的设备相关文件映射到设备的设备驱动以及大量系统表格中。

7.1.2　Linux 块设备驱动程序

块设备是指系统中能够随机（不需要按顺序）访问固定大小数据片的设备，这些数据片就称为块。块设备的特点是对设备的读写是以块为单位的（块的大小由具体的设备决定，并不一定是 Linux 定义的 1024 字节），并且对设备的访问是随机的。为了使高速的 CPU 同低速块设备能够协调工作，提高读写效率，操作系统设计了缓冲机制。当进行读写的时候，首先对缓冲区读写，只有缓冲区中没有需要读的数据，或是需要写的数据没有地方写时，才真正启动设备控制器去控制设备本身进行数据交换。而对于设备本身的数据交换同样也是同缓冲区打交道。

最常见的块设备是硬盘，在嵌入式系统中最典型的块设备就是 Flash 设备。内核管理块设备要比管理其他设备细致得多，需要考虑的问题和完成的工作相对来说也要复杂得多，这是因为块设备访问的随机性，它要保证设备访问的位置能够在介质的不同区间前后移动，同时块设备对系统的执行性能影响很大，因为系统代码和数据都存在在块设备上，运行时需要从块设备中读取，

所以块设备驱动设计得好坏直接影响到系统的工作效率。

块设备中最小的可寻址单元是扇区。扇区大小一般是 2 的整数倍，最常见的大小是 512 个字节。扇区的大小是设备的物理属性。注意，应把扇区作为数据传送的基本单元，不允许传送少于一个扇区的数据，尽管大部分磁盘设备都可以同时传送几个相邻的扇区。

在基于块设备的软件中，能够使用的最小逻辑可寻址单元就是块。它是 VFS 和文件系统传送数据的基本单位。在 Linux 中，块的大小必须是 2 的幂，而且不能超过一个页框。同时它也必须是扇区大小的整数倍，因为每个块必须包含整数个扇区。因此在 8086 体系结构中，允许块的大小为 512、1024、2048 和 4096 字节。每个块都需要自己的块缓冲区，它是内核用来存放块内容的 RAM 内存区。当内核从磁盘读出一个块时，就用从硬件设备中获得的值来填充相应的块缓冲区；同样当内核向磁盘中写入一个块时，就用相关块缓冲区的实际值来更新硬件设备上相应的一组相邻字节。块缓冲区的大小通常要与相应块的大小相匹配。

块设备驱动程序是 Linux 块子系统中的最底层组件。它们从 I/O 调度程序中获得请求，然后按要求处理这些请求。通常一个块设备驱动程序可能处理几个块设备。例如，IDE 设备驱动程序就可以处理多个 IDE 硬盘，其中每一个都是单独的块设备。并且每个磁盘通常是被分区的。每一个分区都被理解为一个独立的逻辑块设备。显然，块设备驱动程序必须处理在块设备对应的设备文件发出的所有 VFS 系统调用。

7.1.3 块设备驱动程序功能

0.95 版的 Linux 内核块设备程序代码划分为四部分：硬盘驱动程序实现，软盘驱动程序实现，内存虚拟盘驱动程序实现和块设备访问接口程序。

块设备驱动从功能上可以划分为两个部分。第一部分是对应各块设备的驱动程序，包含有 floppy.c、hd.c 和 ramdisk.c。第二部分是内核中其他程序访问块设备的接口程序 ll_rw_blk.c。

块设备专用头文件 blk.h 为上面三种块设备与 ll_rw_blk.c 程序交互提供了一个统一的设置方式和相同的设备请求开始程序。其中定义了请求等待队列中项的数据结构 request，用宏语句定义了电梯搜索算法，对内核目前支持的虚拟盘、硬盘和软盘三种块设备，根据它们各自的主设备号分别对应了常数值。

软盘驱动程序 flopp.c 主要包括释放软盘函数 floppy_deselect()、检测指定软驱中软盘更换情况函数 floppy_change()、初始化软盘 DMA 通道函数 setup_DMA()、软盘操作出错中断调用函数 bad_flp_intr()、寻道处理中断调用函数 seek_interrupt()、读写数据传输函数 transfer()、复位软盘控制器函数 reset_floppy()、软盘读写请求项处理函数 do_fd_request()以及软盘系统初始化函数 floppy_init()等。

硬盘驱动程序 hd.c 主要包括硬盘请求项的操作函数 do_hd_request()、读操作中断调用函数 read_intr()、写扇区中断调用函数 write_intr()、设置硬盘分区结构并加载 RAM 虚拟盘和根文件系统函数 sys_setup()、硬盘系统初始化函数 hd_init()、判断并循环等待驱动器就绪函数 controler_ready()、等待硬盘就绪函数 drive_busy()、检测硬盘执行命令后的状态函数 win_result()、向硬盘控制器发送命令块函数 hd_out()和复位检测的硬盘控制器函数 reset_controler()等。

块设备访问接口程序 ll_rw_blk.c 用于实现处理低层块设备的读写操作，主要包括低层页面读写操作函数 ll_rw_page()、低层读写数据块函数 ll_rw_block()、低层读写文件交换函数 ll_rw_swap_file()以及块设备初始化函数 blk_dev_init()等。

内存虚拟盘驱动程序 ramdisk.c 主要包括虚拟盘初始化函数 rd_init()、虚拟盘设备的请求项操作函数 do_rd_request()以及虚拟盘根文件加载函数 rd_load()等。

7.2 重要数据结构

本节主要内容为介绍与块设备驱动相关的数据结构，对各个数据结构的代码进行分析。本节介绍的数据结构列表如表 7-3 所示。

表 7-3 进程相关数据结构列表

名 称	功 能 简 单 描 述
blk_dev_struct	块结构体
request	请求项结构体
floppy_struct	软盘结构体
hd_i_struct	硬盘参数及类型结构体
struct hd_struct	硬盘分区结构体

1. blk_dev_struct

该结构体包括一个请求例程的地址和一个指针。

```
struct blk_dev_struct {
    void (*request_fn)(void);                    //请求操作对应的函数指针
    struct request * current_request;            //指向当前正在处理的块设备请求项结构
};
extern struct blk_dev_struct blk_dev[NR_BLK_DEV]; //块设备数组（NR_BLK_DEV =7）
```

2. request

该结构体用来表示等待进行的 I/O 请求

```
struct request
{
   int dev;                                      //使用的设备号（如果为-1，表示该项没有被使用）
   int cmd;                                      //命令字（READ 或 WRITE）
   int errors;                                   //操作时产生的错误次数
   unsigned long sector;                         //起始扇区
   unsigned long nr_sectors;                     //读/写扇区数
   char *buffer;                                 //数据缓冲区
      struct task_struct *waiting;               //任务等待操作执行完成
   struct buffer_head *bh;                       //缓冲区头指针
struct request *next;                            //指向下一请求项
};
extern struct request request[NR_REQUEST];       //请求队列数组（NR_REQUEST=32）
```

3. floppy_struct

该结构体定义了不同的软盘类型。对某些类型的软盘（例如在 1.2MB 驱动器中的 360KB 软盘等），stretch 属性用于检测磁道是否需要特殊处理。

```
struct floppy_struct {
      unsigned int size, sect, head, track, stretch;   //size 大小（扇区数）；sect 每磁道扇区数
                                                       //head 磁头数；track 磁道数
                                                       //stretch 对磁道是否要特殊处理（标志）
unsigned char gap,rate,spec1;                          //gap 扇区间隙长度（字节数）
```

```
                                                    //rate 数据传输速率
                                                    //spec1 参数（高 4 位步进速率，低四位磁头卸载时间）
};
```

4. hd_i_struct

该结构体定义了硬盘参数及类型。

```
struct hd_i_struct {
        unsigned int head,sect,cyl,wpcom,lzone,ctl;    //head 磁头数；sect 每磁道扇区数
                                                       //cyl 柱面数；wpcom 写前预补偿柱面号
                                                       //lzone 磁头着陆区柱面号；ctl 控制字节
};
```

5. struct hd_struct

该数据结构经常用来填充 gendisk 数据结构，gendisk 含有一个硬盘的信息。当系统初始化时，该数据结构用来发现硬盘和检测分区参数。

```
static struct hd_struct {
   long start_sect;                                    //物理起始扇区号
   long nr_sects;                                      //分区扇区总数
} hd[MAX_HD<<6]={{0,0},};
```


7.3 blk.h 文件

blk.h 是硬盘块设备专用头文件，用于存放硬盘块设备的相关参数。经常与对应的 C 文件存放在一起。

blk.h 文件的详细代码注释如下：

```
1.  #ifndef _BLK_H
2.  #define _BLK_H
3.  #define NR_BLK_DEV 7                              //块设备的数量
4.  #define NR_REQUEST 32                             //请求队列包含的项数为 32
5.  struct request {
6.  int dev;                                          //使用的设备号
7.  int cmd;                                          //使用的命令字
8.  int errors;                                       //产生错误的个数
9.  unsigned long sector;                             //起始扇区
10. unsigned long nr_sectors;                         //读写扇区数
11.  char * buffer;                                   //数据缓冲区
12.  struct task_struct * waiting;                     //任务等待操作执行完成
13.  struct buffer_head * bh;                          //缓冲区的头指针
14.  struct request * next;                            //指向下一个请求项
15.  };
16.  #define IN_ORDER(s1,s2)
17.  ((s1)->cmd<(s2)->cmd || ((s1)->cmd==(s2)->cmd &&
18.  ((s1)->dev < (s2)->dev || (((s1)->dev == (s2)->dev &&
19.  (s1)->sector < (s2)->sector)))))
20.  struct blk_dev_struct {
21.  void (*request_fn)(void);                         //请求操作对应的函数指针
22.  struct request * current_request;                 //指向当前正在处理的块设备请求项结构
23.  };
24.  extern struct blk_dev_struct blk_dev[NR_BLK_DEV];  //块设备数组
```

```
25.     extern struct request request[NR_REQUEST];          //请求队列数组
26.     extern struct task_struct * wait_for_request;       //等待空闲请求的任务结构队列头指针
27.     extern int * blk_size[NR_BLK_DEV];                  //块设备总尺寸
28.     #ifdef MAJOR_NR                                     //主设备号
        //需要增加条目，目前块设备仅支持硬盘和软盘（还有虚拟磁盘）
29.     #if (MAJOR_NR == 1)                                 //RAM 盘主设备号为 1
30.                                                         //内存虚拟磁盘
31.     #define DEVICE_NAME "ramdisk"                       //设备名称 ramdisk
32.     #define DEVICE_REQUEST do_rd_request                //设备请求函数 do_rd_request
33.     #define DEVICE_NR(device) ((device) & 7)            //设备号（0～7）
34.     #define DEVICE_ON(device)                           //打开设备
35.     #define DEVICE_OFF(device)                          //关闭设备
36.     #elif (MAJOR_NR == 2)                               //软盘的主设备号为 2
37.
38.     #define DEVICE_NAME "floppy"                        //设备名称 floppy
39.     #define DEVICE_INTR do_floppy                       //设备的中断处理函数 do_floppy
40.     #define DEVICE_REQUEST do_fd_request                //设备请求函数 do_fd_request
41.     #define DEVICE_NR(device) ((device) & 3)            //设备号（0～3）
42.     #define DEVICE_ON(device) floppy_on(DEVICE_NR(device))      //设备打开函数
43.     #define DEVICE_OFF(device) floppy_off(DEVICE_NR(device))    //设备关闭函数
44.     #elif (MAJOR_NR == 3)                               //硬盘的主设备号为 3
45.
46.     #define DEVICE_NAME "harddisk"                      //设备名称 harddisk
47.     #define DEVICE_INTR do_hd                           //设备中断处理函数 do_hd
48.     #define DEVICE_TIMEOUT HD_TIMER                     //硬盘定时器
49.     #define DEVICE_REQUEST do_hd_request                //设备请求函数 do_hd_request
50.     #define DEVICE_NR(device) (MINOR(device)>>6) //设备号（0～1）
51.     #define DEVICE_ON(device)                           //设备打开函数
52.     #define DEVICE_OFF(device)                          //设备关闭函数
53.     #elif
54.                                                         //未知的块设备
55.     #error "unknown blk device"
56.     #endif
57.     #define CURRENT (blk_dev[MAJOR_NR].current_request)    //主设备的当前请求项
58.     #define CURRENT_DEV DEVICE_NR(CURRENT->dev)      // CURRENT 的设备号
59.     #ifdef DEVICE_INTR
60.     void (*DEVICE_INTR)(void) = NULL;
61.     #endif
        //定义硬盘定时器操作
62.     #ifdef DEVICE_TIMEOUT
63.     #define SET_INTR(x) (DEVICE_INTR = (x),
64.     timer_table[DEVICE_TIMEOUT].expires = jiffies + 200,
65.     timer_active |= 1<<DEVICE_TIMEOUT)
66.     #else
67.     #define SET_INTR(x) (DEVICE_INTR = (x))
68.     #endif
69.     static void (DEVICE_REQUEST)(void);
        //释放锁定的缓冲区
70.     extern inline void unlock_buffer(struct buffer_head * bh)
71.     {
```

```
72.         if (!bh->b_lock)                                    //如果缓冲区 bh 没被上锁，就打印警告信息
73.         printk(DEVICE_NAME ": free buffer being unlocked\n");
74.         bh->b_lock=0;                                       //否则给缓冲区加锁
75.         wake_up(&bh->b_wait);                               //唤醒等待该缓冲区的进程
76.         }
            //请求队列结束操作
77.         extern inline void end_request(int uptodate)
78.         {
79.         DEVICE_OFF(CURRENT->dev);                           //关闭当前设备
80.         if (CURRENT->bh) {                                  //当前主设备请求结构
81.         CURRENT->bh->b_uptodate = uptodate;                 //更新设备标志为 uptodate
82.         unlock_buffer(CURRENT->bh);                         //释放锁定的缓冲区
83.         }
84.         if (!uptodate) {                                    //如果更新标志 uptodate 为 0 则输出设备错误信息
85.         printk(DEVICE_NAME " I/O error\n\r");
86.         printk("dev %04x, block %d\n\r",CURRENT->dev,
87.         CURRENT->bh->b_blocknr);
88.         }
89.         wake_up(&CURRENT->waiting);                         //唤醒等待该请求项的进程
90.         wake_up(&wait_for_request);                         //唤醒等待请求的进程
91.         CURRENT->dev = -1;                                  //释放该请求项
92.         CURRENT = CURRENT->next;                            //删除当前请求项，并将指针指向下一项
             }
            // 下面是对硬盘定时器清理操作的宏定义
93.         #ifdef DEVICE_TIMEOUT
94.         #define CLEAR_DEVICE_TIMEOUT timer_active &= ~(1<<DEVICE_TIMEOUT);
95.         #else
96.         #define CLEAR_DEVICE_TIMEOUT
97.         #endif
98.         #ifdef DEVICE_INTR
99.         #define CLEAR_DEVICE_INTR DEVICE_INTR = 0;
100.        #else
101.        #define CLEAR_DEVICE_INTR
102.        #endif
            //下面是对初始化请求操作的宏定义
103.        #define INIT_REQUEST
104.        repeat:
105.        if (!CURRENT) {                                     //如果当前请求结构指针为 NULL
106.        CLEAR_DEVICE_INTR                                   //清空设备中断标志位
107.        CLEAR_DEVICE_TIMEOUT                                //清空设备定时器
108.        return;
109.        }
110.        if (MAJOR(CURRENT->dev) != MAJOR_NR)     //如果当前主设备号不匹配则死机
111.        panic(DEVICE_NAME ": request list destroyed");
112.        if (CURRENT->bh) {
113.        if (!CURRENT->bh->b_lock)                           //如果在进行请求操作时缓冲区没加锁则死机
114.        panic(DEVICE_NAME ": block not locked");
115.        }
116.        #endif
117.        #endif
```

代码中第 4 行中的 NR_REQUEST 是请求队列中的包含的项数。要注意在写操作时仅使用其中低端的 2/3，而读操作是优先处理请求队列项，设置为 32，看起来是个合理的数字：它足够满足电梯算法的要求，同时当一些缓冲区在请求队列中被锁住时这个数字看起来不是很大。64 看起来就很大（当进行大量写/同步操作，在读取时会造成长时间的暂停）。第 5～15 行给出 request 的扩展结构形式，目的是为了当在分页请求时同样可以使用下面这个结构体。在分页处理中，'bh' 字段设置为 NULL，同时'waiting'字段值用于等待读/写的完成。第 16～19 是对电梯算法的描述：要注意读操作要在写操作之前。这是很正常的：读操作对时间的要求要比写操作更严格。

7.4 floppy.c 文件

floppy.c 是软盘控制器驱动程序文件，主要用于处理对软盘控制器块设备的读写和初始化工作。

本程序中使用请求项和请求项链表结构来处理所有对软盘的读写操作。在实际对驱动器中的软盘进行读写之前，都需要有一个等待时间，即等待马达启动并达到正常工作速度。同样，对磁盘读写操作完成后，驱动器不是立即停止转动，这也需要一个等待时间。当一个磁盘的读写操作发生错误，或某些其他情况导致一个驱动器的马达没有被关闭。此时也需要让系统在一定时间之后自动将其关闭。因此在软盘驱动器操作过程中需要用到很多定时操作，在代码中都有相应的定时处理函数。另外，程序中涉及到很多控制器的执行状态和标志，阅读时读者可以查阅相关书籍。

floppy.c 文件的详细代码注释如下：

```
1.      #include <linux/sched.h>                    //调度程序头文件
2.      #include <linux/fs.h>                       //文件系统头文件
3.      #include <linux/kernel.h>                   //内核头文件
4.      #include <linux/fdreg.h>                    //软盘驱动头文件
5.      #include <asm/system.h>                     //系统头文件
6.      #include <asm/io.h>                         //IO 头文件
7.      #include <asm/segment.h>                    //段操作头文件
8.      #define MAJOR_NR 2                          //软驱的主设备号为 2
        // 块设备头文件。定义请求数据结构、块设备数据结构和宏函数等信息
9.      #include "blk.h"
        // 下面定义三个标志：重新校正 recalibrate；复位操作 reset；寻道 seek
10.     static int recalibrate = 0;
11.     static int reset = 0;
12.     static int seek = 0;
13.     extern unsigned char current_DOR;           //数字输出寄存器 DOR 定义在 shed.c 文件
14.     #define immoutb_p(val,port) \               //字节直接输出
15.     __asm__("outb %0,%1\n\tjmp 1f\n1:\tjmp 1f\n1:"::"a" ((char) (val)),"i" (port))
      //这两个定义用于计算软驱的设备号
16.     #define TYPE(x) ((x)>>2)                    //软驱类型
17.     #define DRIVE(x) ((x)&0x03)                 //软驱序号
18.     #define MAX_ERRORS 12
      // 下面是函数 result()使用的全局变量
      // 这些状态字节中各比特位的含义定义在 include/linux/fdreg.h 头文件
19.     #define MAX_REPLIES 7                       //FDC 最多返回 7 字节的结果信息
20.     static unsigned char reply_buffer[MAX_REPLIES];     //存放 FDC 的返回结果信息
21.     #define ST0 (reply_buffer[0])               //返回结果的状态字节 0
```

```
22.     #define ST1 (reply_buffer[1])                    /返回结果的状态字节 1
23.     #define ST2 (reply_buffer[2])                    //返回结果的状态字节 2
24.     #define ST3 (reply_buffer[3])                    //返回结果的状态字节 3
        //下面是软盘结构体，结构体中的参数有：
        //size    扇区数
        //sect    每个磁道的扇区数
        //head    磁头数
        //track   磁道数
        //stretch 磁道是否需要特殊处理标志位
        //gap     扇区间隙的长度
        //rate    数据传输速率
        //specl   参数规格（高 4 位步进速率，低 4 位磁头卸载时间）
25.     struct floppy_struct {
26.     unsigned int size, sect, head, track, stretch;
27.     unsigned char gap,rate,spec1;
28.     };
29.     static struct floppy_struct floppy_type[] = {
30.     {       0, 0,0, 0,0,0x00,0x00,0x00 },
31.     {   720, 9,2,40,0,0x2A,0x02,0xDF },
32.     { 2400,15,2,80,0,0x1B,0x00,0xDF },
33.     {   720, 9,2,40,1,0x2A,0x02,0xDF },
34.     { 1440, 9,2,80,0,0x2A,0x02,0xDF },
35.     {   720, 9,2,40,1,0x23,0x01,0xDF },
36.     { 1440, 9,2,80,0,0x23,0x01,0xDF },
37.     { 2880,18,2,80,0,0x1B,0x00,0xCF },
38.     };
39.     extern void floppy_interrupt(void);
40.     extern char tmp_floppy_area[1024];
41.     extern char floppy_track_buffer[512*2*18];
        //下面定义一些全局变量，因为这是把信息传递给中断程序最简单的方式
        //它们是用于请求项的数据
42.     static int read_track = 0;
43.     static int buffer_track = -1;
44.     static int buffer_drive = -1;
45.     static int cur_spec1 = -1;
46.     static int cur_rate = -1;
47.     static struct floppy_struct * floppy = floppy_type;
48.     static unsigned char current_drive = 0;
49.     static unsigned char sector = 0;
50.     static unsigned char head = 0;
51.     static unsigned char track = 0;
52.     static unsigned char seek_track = 0;
53.     static unsigned char current_track = 255;
54.     static unsigned char command = 0;
55.     unsigned char selected = 0;
56.     struct task_struct * wait_on_floppy_select = NULL;
        //取消选定软驱
57.     void floppy_deselect(unsigned int nr)
58.     {
59.     if (nr != (current_DOR & 3))
```

ÓÉ Foxit PDF Editor ±à¼-
°æÈ¨ËùÓÐ (c) by Foxit Software Company, 2004
½öÓÃÓÚÆÀ¹À¡£

- android与iphone及ipad开发书籍 ----------持续不断更新中.....
- c、c++、c#语言pdf书籍及vip视频教程 c、c++、c#、vc等-------持续不断更新中.....
- delphi《书籍》及《视频》教程 ----------持续不断更新中.....
- E网情深VIP系列视频教程 黑客破解菜鸟修练班，VB编程学习班，仿站学习培训，免杀培训，个人系统攻防系列教程，服务器搭建学习班，PHOTOSHOP平面设计班，基础制作论坛（论坛网站搭建），网赚系列教程，网站建设教程，网站漏洞基础，远程控制教程，软件破解班，脚本漏洞提权班
- IT9网络学院VIP系列视频教程 免杀培训班，VMware虚拟机，零基础学习C语言，网游外挂开发精品系列语音教程（外挂教程学习必备研修31课全），VB语言教程30课全，Delphi编程到精通，远程控制软件，加密解密班，网络安全与黑客攻防培训，从入门到精通完整系统化学习C++编程，从入门到精通零基础学习汇编，wordpress教程(个人博客系统49课全)，外行人做易语言盗号和钓鱼程序语音教程 网址：WLSAM168.400GB.COM
- Java书籍 ----------持续不断更新中.....
- photoshop、CorelDRAW、AutocAD等图像处理书籍及vip视频教程 -------持续不断更新中.....
- powerbuilder书籍大全
- Visual Basic语言vip视频教程及pdf书籍 ----------持续不断更新中.....
- windows、linux系统开发、系统封装等pdf书籍及VIP视频教程 ----------持续不断更新中.....
- 《3DS Max》pdf书籍
- 《汇编语言》、《反汇编》及《调试》pdf书籍及vip视频教程 ----------持续不断更新中.....
- 《电子书、电子书、还是电子书》pdf专题库 编程开发，家居美食，儿童益智，人物传记，增强记忆，快速阅读
- 信息系统项目管理师、网络工程师、系统分析师等软考类书籍
- 华中红客系列vip视频教程 脚本攻防培训班，源码免杀培训班，Css语言培训班，C语言，Dreamweaver网页设计，html网页设计培训班，PC安全班， php脚本语言培训班，VMWare虚拟机专题，webshell提权培训班，防站教程，零基础免杀培训班，刷钻速成班，脱壳破解班，外挂编写班，网络赚钱培训班，网站入侵培训班
- 外挂、驱动、逆向及封包视频教程 郁金香、独立团、夜猫论坛、天都吧、看流星论坛、一切从零开始等等
- 安全中国系列vip视频教程 易语言软件编程培训班，ASP.net网站开发项目实战培训班
- 我的收藏
- 按键精灵及TC脚本开发软件视频教程 ----------持续不断更新中.....

当前位置：/《电子书、电子书、还是电子书》pdf专题库

文件名

PDF电子书专题库，内容详尽，每天不断更新！！

- 办公类软件使用指南
- 医学
- 历史人物传记
- 哲学宗教
- 外语资料（除英语外） （除英语外）
- 官场类小说
- 建筑工程类
- 情感生活类小说
- 政治军事
- 教育学习科普大全
- 文学理论
- 智力开发、增强记忆、快速阅读技巧大全
- 社会生活
- 科学技术
- 程序编程类
- 经济管理
- 网络安全及管理
- 网赚系列
- 美食小吃烹饪煲汤大全
- 课外读物

本网盘内容太多，持续不断更新，发布各类视频教程、pdf书籍，包括破解、加解密、外挂辅助制作，易语言培训教程、编程语言、网页制作等等，教程及书籍仅用于学习,如用于商业或非法用途的后果自负！

网址：WLSAM168.400GB.COM

VIP培训教程，易语言黑月VIP视频教程，天都网易语言系列培训教程(100集全)，集中营易语言学习视频(80集)

- 棉猴系列vip视频教程 gh0st远控控制源码讲解教程，套接字编程，DLL程序编写，键盘监听驱动程序编写，驱动基础教程，AsyncSelect模型QQ程序教程，C++语言入门基础，NB5.5源码分析教程
- 游戏开发pdf书籍 ----------持续不断更新中.....
- 炒股投资pdf书籍及视频教程 短线高手系列，短线天王系列，操盘论道系列，翻倍黑马，看盘快速入门，庄家手法大曝光等等。 网址：WLSAM168.400GB.COM
- 热门小说集中营 傲世九重天，网游之三国时代，武动乾坤
- 甲壳虫VIP教程全集 asp教程，Delphi培训班，FLASH培训班，Java培训班，linux培训班，PHP培训班，源码免杀班，甲壳虫C++，脚本功防班，免杀班初、中、高级班，破解班，源码免杀班，脱壳班，易语言培训班，无特征码免杀，网站架构培训班，外挂高级班，外挂初级班第1、2部
- 破解、免杀、入侵、脱壳、攻防及漏洞分析系列VIP视频教程（80多部） 天草、黑客动画吧等等--------持续不断更新中....
- 网站建设相关的pdf书籍及各种vip视频教程 ----------持续不断更新中.....
- 网赚、淘宝系列vip视频教程 网赚30天新人魔鬼训练，屠龙网赚团队vip课程，站长大学网赚视频（50课全），图腾团队日赚1000元竞价行销教程，屠龙团队淘宝宝贝卖疯系列，站群网赚系列，淘宝开店视频，红星挂机日赚10元，百万流量系列，漂流瓶圣手全自动挂机引，贴吧邮件定向营销疯狂成交量月入万元
- 英语学习资料百科大全 不断更新。。。
- 饭客论坛系列VIP视频教程 脚本入侵班，黑客之免杀教程，易语言教程，无线网络攻防教程，入侵教程，delphi系列教程，黑客基础入门
- 黑客书籍 有关黑客、安全、加解密技术等等----------持续不断更新中.....
- 黑手安全网VIP系列视频教程 DIV+CSS网页布局，Dreamweaver教程，flsah动画教程，photoshop教程，跟我一起学C++课程，抓鸡
- 黑鹰、黑基、黑防、黑盾vip系列视频教程 破解提高班66课全，SQL注入，ASP注入教程，完完全全学会抓肉鸡，脱壳破解教程50课全，提权班，C语言特训班26讲全，黑客脚本特训班，黑客工具特训班，dedecms仿站教程，VC编写远控30课全，网页美工特训班，木马免杀特训班，驱动开发技术VIP培训班，外挂破解等等。

- [电脑世界的通关密语：电脑编程基础].(杉浦贤).滕永红.扫描版.pdf
- [程序语言的奥妙：算法解读（四色全彩）].(杉浦贤).李克秋.扫描版.pdf
- [差错：软件错误的致命影响].(帕伯斯).邝宇恒等.扫描版.pdf
- [算法之道（第2版）].邹恒明.扫描版.pdf
- [O'Reilly：深入学习MongoDB].(霍多罗夫).巨成等.扫描版.pdf
- [深入浅出WPF].刘铁猛.扫描版.pdf
- [Go语言·云动力（云计算时代的新型编程语言）].樊虹剑.扫描版.pdf
- [精通.NET互操作：P/ Invoke、C++ Interop和COM Interop].黄际洲等.扫描版.pdf
- [编程的奥秘：.NET软件技术学习与实践].金旭亮.扫描版.pdf
- [O'Reilly：学习OpenCV（中文版）].(布拉德斯基等).于仕琪等.扫描版.pdf
- [Go语言编程].许式伟等.扫描版.pdf 网址：WLSAM168.400GB.COM
- [MySQL技术内幕：SQL编程].姜承尧.扫描版.pdf
- [Tomcat权威指南（第2版）].(布里泰恩等).吴豪等.扫描版.pdf
- [Ext江湖].大漠穷秋.扫描版.pdf
- [IT名人堂·Oracle DBA突击：帮你赢得一份DBA职位].张晓明.扫描版.pdf

Total: 77 1 2 3 4 5 6 ›

HTTP://WLSAM168.400GB.COM

```
60.         printk("floppy_deselect: drive not selected\n\r");
61.         selected = 0;
62.         wake_up(&wait_on_floppy_select);
63.         }
        //用于检测软驱中软盘是否更换，更换则返回 1，否则返回 0
64.         int floppy_change(unsigned int nr)
65.         {
66.         repeat:
67.         floppy_on(nr);                                          // 启动指定的软驱 nr
        //如果当前选择的软驱不是指定的 nr，且已经被选定了其他的软驱时，则将任务陷入等待状态
68.         while ((current_DOR & 3) != nr && selected)
69.         sleep_on(&wait_on_floppy_select);
         //如果当前没有选择其他的软驱或者任务被唤醒，当前软驱如果还不是指定的 nr 时，
         //则一直循环等待下去
70.         if ((current_DOR & 3) != nr)
71.         goto repeat;
        //查看数字输入寄存器的值，如果最高位置位，则表示软盘已经更换，此时关闭
        //软驱并且设置磁道缓存位为-1。否则表示软盘没有被更换，关闭软驱并返回 0
72.         if (inb(FD_DIR) & 0x80) {
73.         floppy_off(nr);
74.         buffer_track = -1;
75.         return 1;
76.         }
77.         floppy_off(nr);
78.         return 0;
79.         }
         //复制内存缓冲块
80.         #define copy_buffer(from,to)
81.         __asm__("cld ; rep ; movsl"
82.         ::"c" (BLOCK_SIZE/4),"S" ((long)(from)),"D" ((long)(to))
83.         :"cx","di","si")
        //初始化软盘 DMA 通道
84.         static void setup_DMA(void)
85.         {
86.         unsigned long addr = (long) CURRENT->buffer;     //当前请求项缓冲区地址
87.         unsigned long count = 1024;                      //扇区数 1024
88.         cli();
89.         if (read_track) {
90.         //用来标注坏磁道缓存，防止访问所有的磁道都出现错误。
91.
92.         buffer_drive = buffer_track = -1;                //设置对应的标志位为-1
93.         count = floppy->sect*2*512;                      //计算出坏磁道对应的扇区
94.         addr = (long) floppy_track_buffer;               //软盘坏磁道对应的地址
95.         } else if (addr >= 0x100000) {
        //否则当缓冲区地址大于 1M 时，则将 DMA 缓冲区设置在临时缓冲区域处
96.         addr = (long) tmp_floppy_area;
97.         if (command == FD_WRITE)
98.         copy_buffer(CURRENT->buffer,tmp_floppy_area);
99.         }
100.        /*屏蔽 DMA 通道 2*/
```

```
101.        immoutb_p(4|2,10);
        //汇编代码完成向 DMA 控制器端口 11 和 12 发写命令字
102.        __asm__("outb %%al,$12\n\tjmp 1f\n1:\tjmp 1f\n1:\t"
103.        "outb %%al,$11\n\tjmp 1f\n1:\tjmp 1f\n1:"::
104.        "a" ((char) ((command == FD_READ)?DMA_READ:DMA_WRITE)));
        /*地址低 0～7 位*/
105.        immoutb_p(addr,4);
106.        addr >>= 8;
        /*地址 8～15 位*/
107.        immoutb_p(addr,4);
108.        addr >>= 8;
        /*地址 16～19 位*/
109.        immoutb_p(addr,0x81);
        /*计数器低 8 位(count-1)*/
110.        count--;                                /*1024-1=0x3fff*/
111.        immoutb_p(count,5);
112.        count >>= 8;
        /*计数器的高 8 位*/
113.        immoutb_p(count,5);
114.        /*激活通道 DMA2 的请求*/
115.        immoutb_p(0|2,10);
116.        sti();
117.        }
        //向软驱控制器输出一个字节
118.        static void output_byte(char byte)
119.        {
120.        int counter;
121.        unsigned char status;
122.        if (reset)
123.        return;
        //循环读取控制器 FD_STATUS 的状态。当状态位为 STATUS_READY
        //且 STATUS_DIR=0 时，则向 FD_DATA 端口输出指定的字节
124.        for(counter = 0 ; counter < 10000 ; counter++) {
125.        status = inb_p(FD_STATUS) & (STATUS_READY | STATUS_DIR);
126.        if (status == STATUS_READY) {
127.        outb(byte,FD_DATA);
128.        return;
129.        }
130.        }
        //如果循环 1 万次还不能发送，则置复位标志，并打印错误信息
131.        reset = 1;
132.        printk("Unable to send byte to FDC\n\r");
133.        }
        /*读取 FDC 执行的结果*/
134.        static int result(void)
135.        {
136.        int i = 0, counter, status;
137.        if (reset)              /* 如果已经设置复位标志位，则退出返回-1*/
138.        return -1;
139.        for (counter = 0 ; counter < 10000 ; counter++) {
```

```
140.        status = inb_p(FD_STATUS)&(STATUS_DIR|STATUS_READY|STATUS_BUSY);
          //当循环标志位为 STATUS_READY ，表示没有数据可取了，此时返回已经读取的字节数
141.        if (status == STATUS_READY)
142.        return i;
          //判断方向标志位看是否有数据可读取，并将读取的结果存放在 reply_buffer[]数组中
          //最多可读取字节数位 MAX_REPLIES
143.        if (status == (STATUS_DIR|STATUS_READY|STATUS_BUSY)) {
144.        if (i >= MAX_REPLIES)
145.        break;
146.        reply_buffer[i++] = inb_p(FD_DATA);
147.        }
148.        }
149.        reset = 1;
150.        printk("Getstatus times out\n\r");
151.        return -1;
152.        }
          //软盘操作出错中断调用函数
153.        static void bad_flp_intr(void)
154.        {
155.        current_track = -1;                    //设置当前出错磁道标志位
156.        CURRENT->errors++;                     // 当前请求项出错次数增 1
          //如果当前请求项出错次数大于最大允许出错次数，则取消选定当前软驱，并结束该请求项
157.        if (CURRENT->errors > MAX_ERRORS) {
158.        floppy_deselect(current_drive);
159.        end_request(0);
160.        }
          //如果当前请求项出错次数大于最大允许出错次数的一半，则置复位标志
          //需对软驱进行复位操作
161.        if (CURRENT->errors > MAX_ERRORS/2)
162.        reset = 1;
163.        }
          //下面中断函数是在 DMA 读/写成功后调用，从而可以检查执行结果以及拷贝缓冲区的数据
          //软盘读写操作的中断调用函数
164.        static void rw_interrupt(void)
165.        {
166.        char * buffer_area;
          //如果返回结果的字节数不等于 7，或者 ST0、ST1、ST2 状态位出现错误
167.        if (result() != 7 || (ST0 & 0xf8) || (ST1 & 0xbf) || (ST2 & 0x73)) {
168.        if (ST1 & 0x02) {
          //若是写保护，则打印错误信息，释放当前软驱，并结束当前请求项
          //否则就执行函数 bad_flp_intr()，进行出错计数处理，然后继续执行
          //软盘的请求操作
169.        printk("Drive %d is write protected\n\r",current_drive);
170.        floppy_deselect(current_drive);
171.        end_request(0);
172.        } else
173.            ;
174.        do_fd_request();
175.        return;
176.        }
```

```
        //下面是读取某一磁道扇区数据时的操作。设置磁道缓存相应的标志位，并将数据
        //软驱缓存中的数据拷贝到请求项缓存区中
177.        if (read_track) {
178.        buffer_track = seek_track;
179.        buffer_drive = current_drive;
180.        buffer_area = floppy_track_buffer +((sector-1 + head*floppy->sect)<<9);
181.        copy_buffer(buffer_area,CURRENT->buffer);
        //如果当前请求项的缓冲区地址超过 1M ，则说明此次软盘读操作的内容还放在
        //临时缓冲区内，需要复制到请求项的缓冲区中
182.        }
183.         else if (command == FD_READ &&
184.        (unsigned long)(CURRENT->buffer) >= 0x100000)
185.        copy_buffer(tmp_floppy_area,CURRENT->buffer);
186.        floppy_deselect(current_drive);          //取消选定软驱
187.        end_request(1);                          //结束当前请求项
188.        do_fd_request();                         //执行软盘请求项
189.        }
190.        // 设置 DMA 并输出软盘操作命令和参数
191.        inline void setup_rw_floppy(void)
192.        {
193.        setup_DMA();                             //初始化 DMA 通道
194.        do_floppy = rw_interrupt;                //中断调用函数指针
195.        output_byte(command);                    //发送命令字
196.        if (read_track) {                        //当读取某一磁道数据时，发送以下参数
197.        output_byte(current_drive);
198.        output_byte(track);
199.        output_byte(0);
200.        output_byte(1);
201.        } else {                                 //当读取所有磁道扇区数据时，发送以下参数
202.        output_byte(head<<2 | current_drive);
203.        output_byte(track);
204.        output_byte(head);
205.        output_byte(sector);
206.        }
207.        output_byte(2);                          //发送参数（字节数（N=2）512 字节）
208.        output_byte(floppy->sect);               //发送参数（每磁道扇区数）
209.        output_byte(floppy->gap);                //发送参数（扇区间隔长度）
210.        output_byte(0xFF);
        //发送参数（当 N=0 时，扇区定义的字节长度）
211.        if (reset)          //若在发送命令和参数时出现错误，则继续执行下一软盘操作请求
212.        do_fd_request();
213.        }
        //寻道中断调用函数
214.        static void seek_interrupt(void)
215.        {
216.        //检测中断状态位
        //发送检测中断状态命令
217.        output_byte(FD_SENSEI);
        //如果返回结果字节数不等于 2，或者 ST0 不是寻道终止，或者磁头所在磁道（ST1）
        //不等于设定磁道，则说明发生错误，此时需要重新校正，并执行检测错误计数处理，
```

```
        //然后继续执行软盘的下一次请求项，并退出
218.        if (result() != 2 || (ST0 & 0xF8) != 0x20 || ST1 != seek_track) {
219.        recalibrate = 1;
220.        bad_flp_intr();
221.        do_fd_request();
222.        return;
223.        }
224.        current_track = ST1;                          //设置当前磁道
225.        setup_rw_floppy();                            //设置 DMA 并输出软盘操作命令和参数
226.        }
        //读写数据传输函数
227.        static void transfer(void)
228.        {
229.        //设置读写命令字
230.        read_track = (command == FD_READ) && (CURRENT->errors < 4);
        //看当前驱动器参数是否就是指定驱动器的参数
        //若不是就发送设置驱动器参数命令及相应参数
231.        if (cur_spec1 != floppy->spec1) {
232.        cur_spec1 = floppy->spec1;
233.        output_byte(FD_SPECIFY);                      // 发送设置磁盘参数命令
234.        output_byte(cur_spec1);                       //发送参数
235.        output_byte(6);
236.        }
        //判断当前数据传输速率是否与指定驱动器的速率一致，若不一致，则发送指定
        //软驱的速率值给 FD_DCR
237.        if (cur_rate != floppy->rate)
238.        outb_p(cur_rate = floppy->rate,FD_DCR);
        //若返回结果错误，则再次调用软盘请求函数，并返回
239.        if (reset) {
240.        do_fd_request();
241.        return;
242.        }
        //若寻道标志为零，则设置 DMA 并发送相应读写操作命令和参数，然后返回
        //否则进行寻道处理，并设置软盘中断处理调用函数为寻道中断函数
243.        if (!seek) {
244.        setup_rw_floppy();
245.        return;
246.        }
247.        do_floppy = seek_interrupt;
248.        output_byte(FD_SEEK);
        //如果要读取所有磁道，那么发送当前驱动器号
249.        if (read_track)
250.        output_byte(current_drive);
251.        else
252.        output_byte((head<<2) | current_drive);       //发送参数：磁头号和当前软驱号
253.        output_byte(seek_track);                      // 发送磁道号参数
        //如果已设置复位标志，则继续执行软盘请求项
254.        if (reset)
255.        do_fd_request();
256.        }
```

```
        //软驱重新校正的中断调用函数
257.    static void recal_interrupt(void)
258.    {
259.    output_byte(FD_SENSEI);                          //发送检测中断状态命令
      //如果返回结果字节数不等于 2 或命令异常结束时，则设置复位标志位为 1
260.    if (result()!=2 || (ST0 & 0xE0) == 0x60)
261.    reset = 1;
262.    else                                             //否则，设置复位重新校正标志位为 0
263.    recalibrate = 0;
264.    do_fd_request();                                 //执行软盘请求项
265.    }
        //意外软盘中断请求调用函数
266.    void unexpected_floppy_interrupt(void)
267.    {
268.    output_byte(FD_SENSEI);                          //发送检测中断状态命令
      //如果返回结果字节数不等于 2 或命令异常结束时，则设置复位标志位为 1
269.    if (result()!=2 || (ST0 & 0xE0) == 0x60)
270.    reset = 1;
271.    else                                             //否则，设置复位重新校正标志位为 1
272.    recalibrate = 1;
273.    }
      // 软盘的重新校正处理函数
274.    static void recalibrate_floppy(void)
275.    {
276.    recalibrate = 0;                                 //重新校正标志位初始为 0
277.    current_track = 0;                               //当前磁道号设置为 0
278.    do_floppy = recal_interrupt;                     //设置软盘中断调用函数指针
279.    output_byte(FD_RECALIBRATE);                     //发送重新校正命令
280.    output_byte(head<<2 | current_drive);            //发送当前驱动器号
      //如果设置了复位标志位，则继续执行软盘请求
281.    if (reset)
282.    do_fd_request();
283.    }
        //复位中断调用函数。它在软盘中断处理程序中被调用
284.    static void reset_interrupt(void)
285.    {
286.    output_byte(FD_SENSEI);                          //发送检测中断状态命令
287.    (void) result();                                 //读取命令执行结果
288.    output_byte(FD_SPECIFY);                         //发送设定软驱参数命令
289.    output_byte(cur_spec1);
290.    output_byte(6);
291.    do_fd_request();                                 // 执行软盘请求项
292.    }
        //软盘控制器复位函数
293.    static void reset_floppy(void)
294.    {
295.    int i;
296.    reset = 0;                                       //复位标志设置 0
297.    cur_spec1 = -1;
298.    cur_rate = -1;
```

```
299.      recalibrate = 1;                                    //重新校正标志置位
300.       printk("Reset-floppy called\n\r");                 //打印执行软盘复位操作信息
301.       cli();                                             //关中断
302.       do_floppy = reset_interrupt;                       //设置软盘中断处理程序中调用的函数
303.       outb_p(current_DOR & ~0x04,FD_DOR);                //对软盘控制器执行复位操作
304.       for (i=0 ; i<100 ; i++)                            //空操作，延迟作用
305.       __asm__("nop");
306.       outb(current_DOR,FD_DOR);                          //再次启动软盘控制器
307.       sti();                                             //开中断
308.       }
          //软驱启动的中断调用函数。
309.       static void floppy_on_interrupt(void)
310.       {
311.       selected = 1;                                      //置已选择当前驱动器标志
312.       //如果当前驱动器号与数字输出寄存器 DOR 不同，则需要重新设置 DOR 为
313.       //当前驱动器 current_drive
314.       if (current_drive != (current_DOR & 3)) {
315.       current_DOR &= 0xFC;
316.       current_DOR |= current_drive;
317.       outb(current_DOR,FD_DOR);                          //向数字输出寄存器输出当前 DOR
318.       add_timer(2,&transfer);                            //添加定时器并执行软盘读写传输函数 transfer
319.       } else
320.       transfer();                                        //否则，直接执行软盘读写传输函数
321.       }
322.       //软盘读写请求项处理函数
323.       void do_fd_request(void)
324.       {
325.       unsigned int block;
326.       char * buffer_area;
327.       seek = 0;
          //如果已设置复位标志位，则执行软盘复位操作，并返回
328.       if (reset) {
329.       reset_floppy();
330.       return;
331.       }
          //如果已设置重新校正标志位，则执行软盘重新校正操作，并返回
332.       if (recalibrate) {
333.       recalibrate_floppy();
334.       return;
335.       }
          //检测请求项的合法性
336.       INIT_REQUEST;
          //将请求项结构中软盘设备号中的软盘类型(MINOR(CURRENT->dev)>>2)作为
          //索引项得到软盘参数块
337.       floppy = (MINOR(CURRENT->dev)>>2) + floppy_type;
          //如果当前驱动器不是请求项中指定的驱动器，则置标志 seek 为 1，表示需要
          //进行寻道操作
338.       if (current_drive != CURRENT_DEV)
339.       seek = 1;
340.       current_drive = CURRENT_DEV;                       //设置请求项设备为当前驱动器
```

```
341.        block = CURRENT->sector;                    //设置软盘请求项中起始扇区号
          //如果 block+2 大于磁盘扇区总数，则结束本次软盘请求项
342.        if (block+2 > floppy->size) {
343.        end_request(0);
344.        goto repeat;
345.        }
          //计算对应在磁道上的扇区号、磁头号、磁道号、搜寻磁道号
346.        sector = block % floppy->sect;
347.        block /= floppy->sect;
348.        head = block % floppy->head;
349.        track = block / floppy->head;
350.        seek_track = track << floppy->stretch;
          //如果请求项中是读操作，则置软盘读命令
351.        if (CURRENT->cmd == READ)
352.        command = FD_READ;
353.        else if (CURRENT->cmd == WRITE)          //如果请求项中是写操作，则置软盘写命令码
354.        command = FD_WRITE;
355.        else {
356.        printk("do_fd_request: unknown command\n");
357.        end_request(0);                          //结束请求项操作
358.        goto repeat;
359.        }
        //如果寻道与当前缓冲区中磁道相同且当前驱动器与缓冲区中一致时
        //以 floppy_track_buffer 为索引计算磁道缓冲区大小
360.        if ((seek_track == buffer_track) &&(current_drive == buffer_drive)) {
361.        buffer_area = floppy_track_buffer +
362.        ((sector + head*floppy->sect)<<9);
          //如果请求项是读操作，则将当前缓冲区内容复制到 buffer_area
363.        if (command == FD_READ) {
364.        copy_buffer(buffer_area,CURRENT->buffer);
365.        end_request(1);
366.        goto repeat;
367.        } else
368.        copy_buffer(CURRENT->buffer,buffer_area);
369.        }
        //如果寻道与当前磁道不同，此时 seek 计数加 1，扇区号加 1
370.        if (seek_track != current_track)
371.        seek = 1;
372.        sector++;
        //添加定时器并执行定时中断函数 floppy_on_interrupt
373.        add_timer(ticks_to_floppy_on(current_drive),&floppy_on_interrupt);
374.        }
         //设置软盘尺寸
375.        static int floppy_sizes[] ={
376.        0,     0,     0,     0,
377.        360, 360 ,360, 360,
378.        1200,1200,1200,1200,
379.        360, 360, 360, 360,
380.        720, 720, 720, 720,
381.        360, 360, 360, 360,
382.        720, 720, 720, 720,
```

```
383.        1440,1440,1440,1440
384.        };
        //软盘系统初始化
385.        void floppy_init(void)
386.        {
387.        outb(current_DOR,FD_DOR);                    //启动软盘控制器
388.        blk_size[MAJOR_NR] = floppy_sizes;           //设置块尺寸为软盘大小
389.        blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;
390.        set_intr_gate(0x26,&floppy_interrupt);       //设置软盘中断门 int 0x26
        //复位软盘的中断请求屏蔽位，允许软盘控制器发送中断请求信号
391.        outb(inb_p(0x21)&~0x40,0x21);
392.        }
```

程序中为了适应复位和重新校正操作程序修改了静态变量。这使得一些事情处理起来更简便（output_byte 复位检查等），同时也意味着在出现错误时可以少一些进行跳转，因此也希望代码更方便理解。正像 hd.c 文件一样，本文件中的所有子程序都可以被中断程序调用，因此需要特别注意。硬件中断处理程序是不会休眠的，否则内核就会死机。因而我们不能直接调用“floppy-on”，但是可以通过设置一个特殊的定时中断等方法来解决。

代码中第 18 行定义 MAX_ERRORS=X 并不是意味着每次读错误最多都要 X 次——对于某些类型的错误需要将出错计数乘以 2，因此我们实际上在放弃操作之前只需尝试 X/2 次。

第 25～28 行定义了不同类型的软盘类型。和 Minix 不同，Linux 没有“搜索正确类型”类型，同时这部分代码让人感到费解和怪异。该驱动程序已让我已经遇到了很多问题。stretch 告诉我们对于某些类型的软盘（比如在 1.2M 驱动器中的 360KB 的软盘等），用于检测磁道是否需要特殊处理。而其他的就不需加以说明。速率 rate：0 表示 500kb/s，2 表示 300kbps，1 表示 250kbps；Spec1 参数表示为 0xSH，其中 S 代表步进速率（F=1ms，E=2ms，D=3ms 等）；H 代表磁头卸载时间（1=16ms，2=32ms 等）；Spec2 是（HLD<<1 | ND），其中 HLD 是磁头加载时间（1=2ms，2=4ms 等）；置 ND 位说明不使用 DMA，程序编码设置 HLD=6ms，表示使用 DMA 方式。

第 64～79 行 floppy-change()函数不是从中断程序调用，因此这里可以稍微放松一下，休眠等。要注意 floppy-on()函数试图去设置 current_DOR 对应所需要的驱动器。当同一时间存在多个软盘时不会处于休眠状态，这里就是使用循环方式。

第 214～226 行是在每次软盘控制器寻道（或者重新校正）中断后被调用。这里要注意的是对于“意外的中断”程序同样也会引起重新校正，但不是在这里说明。

第 227～256 行是在传输操作的所有信息都正确设置好后被调用的，也即软驱马达已开启并且已选择了正确的软盘。

7.5 hd.c 文件

hd.c 是硬盘控制器驱动程序文件，主要用于处理对硬盘控制器块设备的读写和硬盘的初始化工作。

7.5.1 功能描述

本程序中的函数可分为三类。一类是用于硬盘中断响应处理过程的函数，如 do_hd_request()、read_intr()和 write_intr()；第二类是用于初始化硬盘和设置硬盘所用数据结构的函数，如 sys_setup()

和 hd_init()；第三类是针对硬盘控制器操作的实用函数，如 controler_ready()、drive_busy()、win_result()、hd_out()和 reset_controler()等。下面分别针对各主要函数做简要说明：

第一类中的 do_hd_request()，它是硬盘请求项的操作函数。该函数首先判断当前请求项是否存在，若当前请求项指针为空，则说明目前硬盘块设备已经没有待处理的请求项，因此立刻退出程序。否则就继续处理当前请求项。并对当前请求项中指明的设备号和请求的盘起始扇区号的合理性进行验证。根据当前请求项提供的信息计算请求数据的磁盘磁道号、磁头号和柱面号。若复位标志（reset）已被设置，则也设置硬盘重新校正标志（recalibrate），并对硬盘执行复位操作，向控制器重新发送“建立驱动器参数”命令（WIN_SPECIFY）。

若重新校正标志被置位，就向控制器发送硬盘重新校正命令（WIN_RESTORE），并在发送之前预先设置好该命令引发的中断中需要执行的 C 函数（recal_intr()），并退出。若当前请求项指定是写操作，则首先设置硬盘控制器调用的 C 函数为 write_intr()，向控制器发送写操作的命令参数块，并循环查询控制器的状态寄存器，以判断请求服务标志（DRQ）是否置位。

若该标志置位，则表示控制器已“同意”接收数据，于是接着就把请求项所指缓冲区中的数据写入控制器的数据缓冲区中。若循环查询超时后该标志仍然没有置位，则说明此次操作失败。于是调用 bad_rw_intr()函数，根据处理当前请求项发生的出错次数来确定是放弃继续当前请求项还是需要设置复位标志，以继续重新处理当前请求项。最后如果当前请求项是读操作，则设置硬盘控制器调用的 C 函数为 read_intr()，并向控制器发送读盘操作命令。

第二类中的 sys_setup()函数是利用 boot/setup.s 程序提供的信息对系统中所含硬盘驱动器的参数进行了设置。然后读取硬盘分区表，并尝试把启动引导盘上的虚拟盘根文件系统映像文件复制到内存虚拟盘中，若成功则加载虚拟盘中的根文件系统，否则就继续执行普通根文件系统加载操作。函数 hd_init()用于在内核初始化时设置硬盘控制器中断描述符，并复位硬盘控制器中断屏蔽码，以允许硬盘控制器发送中断请求信号。

第三类中的 hd_out()，它是硬盘控制器操作命令发送函数。该函数带有一个中断过程中调用的 C 函数指针参数，在向控制器发送命令之前，它首先使用这个参数预置好中断过程中会调用的函数指针（do_hd），然后它按照规定的方式依次向硬盘控制器 0x1f0 至 0x1f7 发送命令参数块。除控制器诊断（WIN_DIAGNOSE）和建立驱动器参数（WIN_SPECIFY）两个命令以外，硬盘控制器在接收到任何其他命令并执行了命令以后，都会向 CPU 发出中断请求信号，从而引发系统去执行硬盘中断处理过程。

7.5.2 代码注释

hd.c 文件的详细代码注释如下：

```
1.      #include <linux/config.h>          //内核配置头文件
2.       #include <linux/sched.h>          //调度程序头文件
3.       #include <linux/timer.h>          //内核定时器相关定义
4.       #include <linux/fs.h>             //文件系统头文件
5.       #include <linux/kernel.h>         //内核头文件
6.       #include <linux/hdreg.h>          //硬盘参数头文件
7.       #include <asm/system.h>           //系统头文件
8.       #include <asm/io.h>               //IO 头文件
9.       #include <asm/segment.h>          //段操作头文件
10.       #define MAJOR_NR 3               //硬盘主设备号是 3
11.       #include "blk.h"                 //块设备头文件
```

```
12.     #define CMOS_READ(addr) ({          //读 CMOS 参数宏函数
13.     outb_p(0x80|addr,0x70);
14.     inb_p(0x71);
15.     })
16.
17.     #define MAX_ERRORS      7           //读/写一个扇区时最多出错次数
18.     #define MAX_HD          2           //支持的最多硬盘数
19.     static void recal_intr(void);       //硬盘中断程序在复位操作时被调用的重新校正函数
20.     static void bad_rw_intr(void);      //读写硬盘失败处理调用函数
21.     static int recalibrate = 0;         //重新校正标志
22.     static int reset = 0;               //复位标志
23.     struct hd_i_struct {
24.     unsigned int head,sect,cyl,wpcom,lzone,ctl;
25.     };
      //宏定义声明，如果已经在 include/linux/config.h 中定义了 HD_TYPE
26.     #ifdef HD_TYPE
27.     struct hd_i_struct hd_info[] = { HD_TYPE }; // 将定义好的参数作为 hd_info[]的数据
28.     #define NR_HD ((sizeof (hd_info))/(sizeof (struct hd_i_struct)))      //计算硬盘数
29.     #else
30.     struct hd_i_struct hd_info[] = { {0,0,0,0,0,0},{0,0,0,0,0,0} };       //否则，全都设为 0
31.     static int NR_HD = 0;
32.     #endif
      //定义硬盘分区结构
33.     static struct hd_struct {
34.     long start_sect;                //每个分区的物理起始扇区号
35.     long nr_sects;                  //分区扇区总数
36.     } hd[MAX_HD<<6]={{0,0},};
37.     static int hd_sizes[MAX_HD<<6] = {0, };
      //端口读写函数（读端口 port，共读 nr 字，保存在 buf 中）
38.     #define port_read(port,buf,nr) \
39.     __asm__("cld;rep;insw"::"d" (port),"D" (buf),"c" (nr):"cx","di")
      //端口写函数（写端口 port，共写 nr 字，从 buf 中取数据）
40.     #define port_write(port,buf,nr) \
41.     __asm__("cld;rep;outsw"::"d" (port),"S" (buf),"c" (nr):"cx","si")
42.     extern void hd_interrupt(void);
43.     extern void rd_load(void);
44.     static unsigned int current_minor;
      //硬盘分区检测函数
45.     static void check_partition(unsigned int dev)
46.     {
47.     int minor, i;
48.     struct buffer_head *bh;
49.     struct partition *p;
       //如果当前设备缓存内容和缓冲区中不一致，则输出错误提示信息
50.     if (!(bh = bread(dev,0))) {
51.     printk("Unable to read partition table of device %04x\n",dev);
52.     return;
53.     }
54.     minor = current_minor;
       //检测当前设备数据块，并记录开始分区及分区内字节数
```

```
55.         if (*(unsigned short *) (bh->b_data+510) == 0xAA55) {
56.         p = 0x1BE + (void *)bh->b_data;
57.         for (i=0 ; i<4 ; i++,p++) {
58.         hd[i+minor].start_sect = p->start_sect;
59.         hd[i+minor].nr_sects = p->nr_sects;
60.         }
61.         if (p->nr_sects && p->sys_ind == EXTENDED_PARTITION) {
62.         current_minor += 4;
63.         check_partition(i+minor);
64.         }
65.         } else
66.         printk("Bad partition table on dev %04x\n",dev);
67.         brelse(bh);
68.         }
69.         int sys_setup(void * BIOS)
70.         {
71.         static int callable = 1;
72.           int i,drive;
73.         unsigned char cmos_disks;
74.         if (!callable)
75.         return -1;
76.         callable = 0;
        //宏定义声明，如果没有在 config.h 中定义硬盘参数，就从 0x90080 处读入
77.         #ifndef HD_TYPE
78.         for (drive=0 ; drive<2 ; drive++) {
79.         hd_info[drive].cyl = *(unsigned short *) BIOS;          //柱面数
80.         hd_info[drive].head = *(unsigned char *) (2+BIOS);      //磁头数
81.         hd_info[drive].wpcom = *(unsigned short *) (5+BIOS);    //写前预补偿柱面号
82.         hd_info[drive].ctl = *(unsigned char *) (8+BIOS);       //控制字节
83.         hd_info[drive].lzone = *(unsigned short *) (12+BIOS);   //磁头着陆区柱面号
84.         hd_info[drive].sect = *(unsigned char *) (14+BIOS);     //每磁道扇区数
85.         BIOS += 16;                                             // BIOS 指向下一个表
86.         }
87.         if (hd_info[1].cyl)
88.         NR_HD=2;                                                //硬盘数置为 2
89.         else
90.         NR_HD=1;
91.         #endif
        //设置每个硬盘的起始扇区号和扇区总数
92.         for (i=0 ; i<NR_HD ; i++) {
93.         hd[i<<6].start_sect = 0;
94.           hd[i<<6].nr_sects = hd_info[i].head*
95.         hd_info[i].sect*hd_info[i].cyl;
96.         }
        //检测硬盘是否是 AT 控制器兼容的
97.         if ((cmos_disks = CMOS_READ(0x12)) & 0xf0)
98.         if (cmos_disks & 0x0f)
99.         NR_HD = 2;
100.        else
101.        NR_HD = 1;
```

```
102.        else
103.        NR_HD = 0;
          //若 NR_HD=0，则两个硬盘都不是 AT 控制器兼容的，硬盘数据结构清零
          //若 NR_HD=1，则将第 2 个硬盘的参数清零
104.        for (i = NR_HD ; i < 2 ; i++) {
105.        hd[i<<6].start_sect = 0;
106.        hd[i<<6].nr_sects = 0;
107.        }
        //读取每个硬盘上第 1 块数据，获取分区表的信息
108.        for (drive=0 ; drive<NR_HD ; drive++) {
109.        current_minor = 1+(drive<<6);
110.        check_partition(0x0300+(drive<<6));
111.        }
112.        for (i=0 ; i<(MAX_HD<<6) ; i++)
113.        hd_sizes[i] = hd[i].nr_sects>>1 ;
114.        blk_size[MAJOR_NR] = hd_sizes;
115.        if (NR_HD)
          //如果有硬盘存在并且已读入分区表，则打印分区表正常信息
116.        printk("Partition table%s ok.\n\r",(NR_HD>1)?"s":"");
117.        rd_load();                                  //加载 RAMDISK
118.        mount_root();                               //安装根文件系统
119.        return (0);
120.        }
          //判断并循环等待驱动器就绪
121.        static int controller_ready(void)
122.        {
123.        int retries = 100000;                       //等待循环的次数
124.
125.        while (--retries && (inb_p(HD_STATUS)&0x80))
126.          if (!retries)                             //等待循环次数为 0，驱动器处于就绪状态
127.        printk("controller_ready: status = %02x\n\r",
128.        (unsigned char) inb_p(HD_STATUS));
129.        return (retries);
130.        }
          //检测硬盘执行命令后的状态
131.        static int win_result(void)
132.        {
133.          int i=inb_p(HD_STATUS);                   //取状态信息
134.        if ((i & (BUSY_STAT | READY_STAT | WRERR_STAT | SEEK_STAT | ERR_STAT))
135.        == (READY_STAT | SEEK_STAT))
136.        return(0);
137.          if (i&1) i=inb(HD_ERROR);                 //若 ERR_STAT 置位，则读取错误寄存器
138.        return (1);
139.        static void hd_out(unsigned int drive,unsigned int nsect,unsigned int sect,
140.        unsigned int head,unsigned int cyl,unsigned int cmd,
        //硬盘中断处理程序中将调用的 C 处理函数
141.        void (*intr_addr)(void))
142.        {
143.        unsigned short port;
          //如果驱动器号>1 或磁头号>15，则程序输出写错误扇区
```

```
144.        if (drive>1 || head>15)
145.        panic("Trying to write bad sector");
          //如果重置标志已设置，或等待一段时间后仍未就绪则出错，则程序直接返回
146.        if (reset || !controller_ready()) {
147.        reset = 1;
148.        return;
149.        }
150.        SET_INTR(intr_addr);
151.          outb_p(hd_info[drive].ctl,HD_CMD);          //向控制寄存器（0x3f6）输出控制字节
152.          port=HD_DATA;                               //置 dx 为数据寄存器端口
153.          outb_p(hd_info[drive].wpcom>>2,++port);     //参数：写预补偿柱面号
154.          outb_p(nsect,++port);                       //参数：读/写扇区总数
155.          outb_p(sect,++port);                        //参数：起始扇区
156.          outb_p(cyl,++port);                         //参数：柱面号的低 8 位
157.          outb_p(cyl>>8,++port);                      //参数：柱面号的高 8 位
158.          outb_p(0xA0|(drive<<4)|head,++port);        //参数：驱动器号加磁头号
159.          outb(cmd,++port);                           //命令：硬盘控制命令
160.        }
          //等待硬盘就绪函数
161.        static int drive_busy(void)
162.        {
163.        unsigned int i;
164.        unsigned char c;
165.        for (i = 0; i < 50000; i++) {                 //循环等待就绪标志位置位
166.                c = inb_p(HD_STATUS);                 //取主控制器状态字节
          //检测忙状态位、就绪位和寻道结束位
167.          c &= (BUSY_STAT | READY_STAT | SEEK_STAT);
          //若仅有就绪或寻道结束标志，则返回 0
168.          if (c == (READY_STAT | SEEK_STAT))
169.          return 0;
170.          }
          //否则等待超时，输出显示信息。并返回 1
171.          printk("HD controller times out, c=%02x\n\r",c);
172.          return(1);
173.          }
          //复位检测的硬盘控制器函数
174.          static void reset_controller(void)
175.          {
176.          int     i;
177.            outb(4,HD_CMD);                           //向控制寄存器端口发送复位（4）控制字节
178.            for(i = 0; i < 1000; i++) nop();          //等待，空操作
          //发送正常的控制字节（不禁止重试、重读）
179.          outb(hd_info[0].ctl & 0x0f ,HD_CMD);
180.            if (drive_busy())                         //若等待硬盘就绪超时，则输出错误信息
181.          printk("HD-controller still busy\n\r");
          //取错误寄存器的值，并判断它是否为 1，不等于 1，则输出错误信息
182.          if ((i = inb(HD_ERROR)) != 1)
183.          printk("HD-controller reset failed: %02x\n\r",i);
184.          }
          //复位硬盘函数
```

```
185.    static void reset_hd(void)
186.    {
187.    static int i;
188.    repeat:
189.    if (reset) {
190.    reset = 0;
191.    i = -1;
192.            reset_controller();                    //复位硬盘控制器
193.    } else if (win_result()) {
194.    bad_rw_intr();
195.    if (reset)
196.    goto repeat;
197.    }
198.    i++;
199.      if (i < NR_HD) {                             //发送硬盘控制器命令
200.    hd_out(i,hd_info[i].sect,hd_info[i].sect,hd_info[i].head-1,
201.    hd_info[i].cyl,WIN_SPECIFY,&reset_hd);
202.            if (reset)                             //若已置复位标志位，则转入 repeat 执行
203.    goto repeat;
204.      } else                                       //否则，执行硬盘其他请求操作
205.    do_hd_request();
206.    }
        //意外硬盘中断调用函数
207.    void unexpected_hd_interrupt(void)
208.    {
209.    printk("Unexpected HD interrupt\n\r");
210.    reset = 1;
211.      do_hd_request();                             //硬盘中断处理程序中调用该函数执行请求操作
212.    }
        //读写硬盘出错时调用函数
213.    static void bad_rw_intr(void)
214.    {
        //如果读扇区时的出错次数大于或等于 7 次时，结束请求并唤醒等待
        //该请求的进程，并设置对应缓冲区更新标志复位
215.    if (++CURRENT->errors >= MAX_ERRORS)
216.    end_request(0);
        //如果读一扇区时的出错次数大于 3 次，则要执行复位操作
217.    if (CURRENT->errors > MAX_ERRORS/2)
218.    reset = 1;
219.    }
        //读操作的中断调用函数
220.    static void read_intr(void)
221.    {
        //若控制器忙、读写错或命令执行错，则进行读写硬盘失败处理
        //然后再次请求硬盘作相应（复位）处理
222.    if (win_result()) {
223.    bad_rw_intr();
224.    do_hd_request();
225.    return;
226.    }
```

```
        //将数据从数据寄存器口读到请求结构缓冲区
227.      port_read(HD_DATA,CURRENT->buffer,256);
228.      CURRENT->errors = 0;                          //出错次数清零
229.        CURRENT->buffer += 512;                     //调整缓冲区指针，指向新的空区
230.        CURRENT->sector++;                          //起始扇区号加 1
        //如果所需读出的扇区数还没有读完，则再次调用函数指针 read_intr()
231.      if (--CURRENT->nr_sectors) {
232.      SET_INTR(&read_intr);
233.      return;
234.      }
        //若全部扇区数据已经读完，则处理请求结束，执行其他硬盘请求操作
235.      end_request(1);
236.      do_hd_request();
237.      }
        //写扇区中断调用函数
238.      static void write_intr(void)
239.      {
       //如果硬盘控制器返回错误信息，则首先进行硬盘读写失败处理
        //然后再次请求硬盘作相应处理
240.      if (win_result()) {
241.      bad_rw_intr();
242.      do_hd_request();
243.      return;
244.      }
245.      if (--CURRENT->nr_sectors) {                  //若还有扇区要写
246.        CURRENT->sector++;                          //请求起始扇区号加 1
247.      CURRENT->buffer += 512;                       //调整缓冲区指针
248.        SET_INTR(&write_intr);                      //设置硬盘中断程序调用函数指针。向数据寄存
                                                      //器端口写 256 字节
249.      port_write(HD_DATA,CURRENT->buffer,256);
250.      return;
251.      }
      //若全部扇区数据已经写完，则处理请求结束，执行其他硬盘请求操作
252.      end_request(1);
253.      do_hd_request();
254.      }
      //硬盘重新校正中断调用函数
255.      static void recal_intr(void)
256.      {
257.      if (win_result())                             //若硬盘控制器返回错误信息
258.            bad_rw_intr();                          //先进行硬盘读写失败处理
259.        do_hd_request();                            //然后再请求硬盘作相应操作
260.      }
      //硬盘请求超时计数函数
261.      static void hd_times_out(void)
262.      {
263.      if (!CURRENT)
264.      return;
265.      printk("HD timeout");
266.      if (++CURRENT->errors >= MAX_ERRORS)
```

```
267.        end_request(0);
268.        SET_INTR(NULL);
269.        reset = 1;
270.        do_hd_request();
271.        }
          //执行硬盘读写请求操作
272.        void do_hd_request(void)
273.        {
274.        int i,r;
275.        unsigned int block,dev;
276.        unsigned int sec,head,cyl;
277.        unsigned int nsect;
278.        INIT_REQUEST;                          //检测请求项的合法性
279.        dev = MINOR(CURRENT->dev);             //取设备号中的子设备号
280.        block = CURRENT->sector;               //请求的起始扇区
          //如果子设备号不存在或者起始扇区大于该分区扇区数-2，则结束该请求
          //并跳转到标号 repeat 处
281.        if (dev >= (NR_HD<<6) || block+2 > hd[dev].nr_sects) {
282.        end_request(0);
283.          goto repeat;
284.        }
285.        block += hd[dev].start_sect;
286.        dev >>= 6;
          //下面代码用来从硬盘信息结构中根据起始扇区号和每磁道扇区数
          //计算在磁道中的扇区号（sec）、所在柱面号（cyl）和磁头号（head）
287.        sec = block % hd_info[dev].sect;
288.        block /= hd_info[dev].sect;
289.        head = block % hd_info[dev].head;
290.        cyl = block / hd_info[dev].head;
291.        sec++;
292.          nsect = CURRENT->nr_sectors;          //要被读/写的扇区数
          //若 reset 置 1，则执行复位操作
          //复位硬盘和控制器，置重新校正标志，返回
293.        if (reset) {
294.        recalibrate = 1;
295.        reset_hd();
296.        return;
297.        }
          //如果重新校正标志置位，则首先复位该标志
          //然后向硬盘控制器发送重新校正命令
298.        if (recalibrate) {
299.        recalibrate = 0;
300.        hd_out(dev,hd_info[CURRENT_DEV].sect,0,0,0,
301.        WIN_RESTORE,&recal_intr);
302.        if (reset)
303.        goto repeat;
304.        return;
305.        }
          //如果当前请求是写扇区操作，则发送写命令
          //循环读取状态寄存器信息并判断请求服务标志 DRQ_STAT 是否置位
```

```
306.        if (CURRENT->cmd == WRITE) {
307.        hd_out(dev,nsect,sec,head,cyl,WIN_WRITE,&write_intr);
308.        if (reset)
309.        goto repeat;
310.        for(i=0 ; i<10000 && !(r=inb_p(HD_STATUS)&DRQ_STAT) ; i++)
311.      //如果设置请求服务位则退出循环。若等到循环结束也没有置位，则此次写硬盘
          //操作失败，去处理下一个硬盘请求。否则向硬盘控制器数据寄存器端口
          // HD_DATA 写入 1 个扇区的数据
312.        if (!r) {
313.        bad_rw_intr();
314.        goto repeat;
315.        }
316.        port_write(HD_DATA,CURRENT->buffer,256);
        //否则如果当前请求是读硬盘扇区，则向硬盘控制器发送读扇区命令
317.        } else if (CURRENT->cmd == READ) {
318.        hd_out(dev,nsect,sec,head,cyl,WIN_READ,&read_intr);
319.        if (reset)
320.        goto repeat;
321.        } else
322.        panic("unknown hd-command");
323.        }
      //硬盘系统初始化函数
324.        void hd_init(void)
325.        {
326.        blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;
327.          set_trap_gate(0x2E,&hd_interrupt);          //设置硬盘中断门向量
328.          outb_p(inb_p(0x21)&0xfb,0x21);              //复位屏蔽位，允许从片发出中断请求信号
      //复位硬盘的中断请求屏蔽位，允许硬盘控制器发送中断请求信号
329.        outb(inb_p(0xA1)&0xbf,0xA1);
330.          timer_table[HD_TIMER].fn = hd_times_out; //设置硬盘超时定时器函数
331.        }
```

本程序是底层硬盘中断程序。主要用于扫描请求列表，在函数之间使用中断进行跳转。因为所有的函数都是在中断被调用的，所以它们不可以睡眠。建议这里需要特别注意。本程序由 Drew Eckhardt 修改，利用 CMOS 信息检测硬盘数。

程序中对 CMOS 关于硬盘的信息有些怀疑：可能会出现这样的情况，假如我们有一块 SCSI/ESDI/等的控制器，它是以 ST-506 方式与 BIOS 兼容的，因而会出现在 BIOS 参数表中，但却又不是与寄存器兼容的，因此这些参数在 CMOS 中又不存在。另外，我们假设 ST-506 驱动器（如果有的话）是系统中的基本驱动器，也即以驱动器 1 或 2 出现的驱动器。第 1 个驱动器参数存放在 CMOS 字节 0x12 的高半字节中，第 2 个存放在低半字节中。该 4 位字节信息可以是驱动器类型，也可能仅是 0xf。0xf 表示使用 CMOS 中 0x19 字节作为驱动器 1 的 8 位类型字节，使用 CMOS 中 0x1A 字节作为驱动器 2 的类型字节。总之，一个非零值意味着有一个 AT 控制器硬盘兼容的驱动器。

代码中第 23～25 行定义了硬盘参数及其类型，结构体中定义各字段含义分别是磁头数、每磁道扇区数、柱面数、写前预补偿柱面号、磁头着陆柱面号、控制字节。

第 69～120 行定义的函数只在初始化时被调用一次。用静态变量 callable 作为可调用标志。主要用于完成读取 CMOS 和硬盘参数表信息，设置硬盘分区结构 hd，并加载 RAM 虚拟盘和根

文件系统。

第 139～140 行是向硬盘控制器发送命令块。调用参数：drive - 硬盘号（0-1）；nsect - 读写扇区数；sect - 起始扇区；head - 磁头号；cyl - 柱面号；cmd - 命令码。

7.6 ll_rw_blk.c 文件

ll_rw_blk.c 文件主要用于处理低层块设备的读写操作，还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。它是本章所有块设备与系统其他部分的接口程序。

读写块设备中的数据是通过调用与上层其他程序的接口函数 ll_rw_block()。该函数的主要功能是为块设备创建块设备读写请求项，并插入到指定块设备请求队列中。实际的读写操作则是由设备的请求项处理函数 request_fn()完成。对于硬盘操作，该函数是 do_hd_request()；对于软盘操作，该函数是 do_fd_request()。ll_rw_blk.c 文件的详细代码注释如下：

```
1.      #include <errno.h>                                          //错误号头文件
2.      #include <linux/sched.h>                                    //调度程序头文件
3.      #include <linux/kernel.h>                                   //内核头文件
4.      #include <asm/system.h>                                     //系统头文件
5.      #include "blk.h"                                            //块设备头文件
      //该请求结构中含有加载 nr 扇区数据到内存必须的信息
6.      struct request request[NR_REQUEST];
      //当没有空闲项时，用于临时等待的请求数组
7.      struct task_struct * wait_for_request = NULL;
      //该数组使用主设备号作为索引
8.      struct blk_dev_struct blk_dev[NR_BLK_DEV] = {
9.        { NULL, NULL },                                           // 0 表示无设备
10.       { NULL, NULL },                                           // 1 表示内存
11.       { NULL, NULL },                                           // 2 表示软驱设备
12.       { NULL, NULL },                                           // 3 表示硬盘设备
13.       { NULL, NULL },                                           // 4 表示 ttyx 设备
14.       { NULL, NULL },                                           // 5 表示 tty 设备
15.       { NULL, NULL }                                            // 6 表示打印机设备
16.     };
17.     int * blk_size[NR_BLK_DEV] = { NULL, NULL, };               //包含所有块设备的尺寸
      // 锁定指定的缓冲区 bh
18.     static inline void lock_buffer(struct buffer_head * bh)
19.     {
20.       cli();                                                    //清中断
21.     while (bh->b_lock)                                          //如果缓冲区已被锁定，则睡眠，等
                                                                    //待缓冲区解锁
22.   sleep_on(&bh->b_wait);
23.     bh->b_lock=1;                                               //锁定该缓冲区
24.     sti();                                                      //开中断
25.     }
      //释放（解锁）锁定的缓冲区
26.     static inline void unlock_buffer(struct buffer_head * bh)
27.     {
28.       if (!bh->b_lock)                          //如果该缓冲区并没有被锁定，则打印出错信息
29.   printk("ll_rw_block.c: buffer not locked\n\r");
30.       bh->b_lock = 0;                           //清除锁标志
```

```
31.           wake_up(&bh->b_wait);                    //唤醒等待该缓冲区的任务
32.         }
          //函数是向链表中加入一项请求。它关闭中断，从而能安全地处理请求链表
33.         static void add_request(struct blk_dev_struct * dev, struct request * req)
34.         {
35.         struct request * tmp;
36.         req->next = NULL;
37.         cli();
38.         if (req->bh)
39.         req->bh->b_dirt = 0;
40.         //如果 dev 的当前请求字段为空，则表示目前该设备没有请求项，本次是首个请求项
41.         //因此可将块设备当前请求指针直接指向请求项，并立刻执行相应设备的请求函数
42.         if (!(tmp = dev->current_request)) {
43.         dev->current_request = req;
44.         sti();
45.         (dev->request_fn)();
46.         return;
47.         }
        //如果当前该设备已经有请求项在等待，则首先利用电梯算法搜索最佳位置，然后将
        //当前请求插入请求链表中
48.         for ( ; tmp->next ; tmp = tmp->next) {
49.         if (!req->bh)
50.         if (tmp->next->bh)
51.         break;
52.         else
53.         continue;
54.         if ((IN_ORDER(tmp,req) ||
55.         !IN_ORDER(tmp,tmp->next)) &&
56.         IN_ORDER(req,tmp->next))
57.         break;
58.         }
59.         req->next = tmp->next;
60.         tmp->next = req;
61.         sti();
62.         }
        //创建请求项并插入请求队列
63.         static void make_request(int major,int rw, struct buffer_head * bh)
64.         {
65.         struct request * req;
66.         int rw_ahead;
        // 如果指定的缓冲区正在使用，已被上锁时，就放弃预读/写请求
67.         if (rw_ahead = (rw == READA || rw == WRITEA)) {
68.         if (bh->b_lock)
69.         return;
70.         if (rw == READA)
71.         rw = READ;
72.         else
73.         rw = WRITE;
74.         }
        //如果命令不是 READ 或 WRITE 则表示程序出错，打印出错信息并死机
```

```
75.     if (rw!=READ && rw!=WRITE) {
76.     printk("Bad block dev command, must be R/W/RA/WA\n");
77.     return;
78.     }
79.     lock_buffer(bh);
80.     if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
81.     unlock_buffer(bh);
82.     return;
83.     }
84.     repeat:
      //下面请求项将从请求数组末尾开始搜索空项并填入
85.     if (rw == READ)
86.             req = request+NR_REQUEST;          //对于读请求，将队列指针指向队列尾部
87.     else
88.         req = request+((NR_REQUEST*2)/3);      //对于写请求，队列指针指向队列 2/3 处
89.     /* 搜索一个空请求项 */
90.     cli();
      //从后向前搜索，当请求结构 request 中 dev 字段值为-1 时，表示该项未被占用
91.     while (--req >= request)
92.     if (req->dev < 0)
93.     goto found;
      /* 如果没有找到空闲项，则本次新请求将睡眠：需检查是否提前读/写 */
94.     if (rw_ahead) {                            //如果是提前读/写请求，则解锁缓冲区，退出
95.     sti();
96.     unlock_buffer(bh);
97.     return;
98.     }
99.         sleep_on(&wait_for_request);           //否则本次请求将睡眠，稍等一会再检查请求队列
100.    sti();
101.    goto repeat;
102.    found:  sti();
      //向空闲请求项中填写请求信息，并将其加入队列中
      //填充请求结构中的各字段值
103.    req->dev = bh->b_dev; //设备号。
104.        req->cmd = rw;                         //命令（READ/WRITE）
105.        req->errors=0;                         //操作时产生的错误次数
106.        req->sector = bh->b_blocknr<<1;        //起始扇区
107.        req->nr_sectors = 2;                   //读写扇区数
108.        req->buffer = bh->b_data;              //数据缓冲区
109.        req->waiting = NULL;                   //任务等待操作执行完成的地方
110.        req->bh = bh;                          //缓冲区头指针
111.        req->next = NULL;                      //指向下一请求项
112.        add_request(major+blk_dev,req);        //将请求项加入队列中
113.    }
      //低层页面读写操作
114.    void ll_rw_page(int rw, int dev, int page, char * buffer)
115.    {
116.        struct request * req;                  //以页面（4K）为单位访问块设备数据，即每次
                                                   //读/写 8 个扇区
117.    unsigned int major = MAJOR(dev);
```

```
        //如果设备的主设备号不存在或者该设备的读写操作函数不存在，则打印出错信息，并返回
118.    if (major >= NR_BLK_DEV || !(blk_dev[major].request_fn)) {
119.    printk("Trying to read nonexistent block-device\n\r");
120.    return;
121.    }
        //如果设备命令不是读也不是写，显示错误的命令信息
122.    if (rw!=READ && rw!=WRITE)
123.    panic("Bad block dev command, must be R/W");
124.    cli();
125.    repeat:
126.    req = request+NR_REQUEST;
127.    while (--req >= request)
128.    if (req->dev<0)
129.    break;
130.    if (req < request) {
131.    sleep_on(&wait_for_request);
132.    goto repeat;
133.    }
134.    sti();
        //向空闲请求项中填写请求信息，并将其加入队列中，填充请求结构中的各字段值
135.    req->dev = dev;                                  //设备号
136.    req->cmd = rw;                                   //命令（READ/WRITE）
137.        req->errors = 0;                             //操作时产生的错误次数
138.        req->sector = page<<3;                       //起始扇区
139.        req->nr_sectors = 8;                         //读写扇区数
140.        req->buffer = buffer;                        //数据缓冲区
141.        req->waiting = current;                      //任务等待操作执行完成的地方
142.        req->bh = NULL;                              //缓冲区头指针
143.        req->next = NULL;                            //指向下一请求项
144.    current->state = TASK_UNINTERRUPTIBLE;           //当前请求状态项
145.    add_request(major+blk_dev,req);
146.    schedule();
147.    }
        //低层读写数据块函数
148.    void ll_rw_block(int rw, struct buffer_head * bh)
149.    {
150.        unsigned int major;                          //主设备号
        //如果设备的主设备号不存在或者该设备的读写操作函数不存在，则显示出错信息，并返回
151.    if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||
152.    !(blk_dev[major].request_fn)) {
153.    printk("ll_rw_block: Trying to read nonexistent block-device\n\r");
154.    return;
155.    }
156.    make_request(major,rw,bh);                       //创建请求项并插入请求队列
157.    }
        //块设备初始化函数，被初始化程序 main.c 调用
158.    void blk_dev_init(void)
159.    {
160.    int i;
        //初始化请求数组，将所有请求项置为空闲项(dev = -1)
```

```
161.      for (i=0 ; i<NR_REQUEST ; i++) {
162.      request[i].dev = -1;
163.      request[i].next = NULL;
164.      }
165.      }
          //低层读写文件交换函数
166.      void ll_rw_swap_file(int rw, int dev, unsigned int *b, int nb, char *buf)
167.      {
168.      int i;
169.      struct request * req;
170.      unsigned int major = MAJOR(dev);
          //如果设备的主设备号不存在或者该设备的读写操作函数不存在，则打印出错信息，并返回
171.      if (major >= NR_BLK_DEV || !(blk_dev[major].request_fn)) {
172.      printk("ll_rw_swap_file: trying to swap nonexistent block-device\n\r");
173.      return;
174.      }
          //如果设备命令不是读也不是写，显示错误的命令信息
175.      if (rw!=READ && rw!=WRITE) {
176.      printk("ll_rw_swap: bad block dev command, must be R/W");
177.      return;
178.      }
179.      for (i=0; i<nb; i++, buf += BLOCK_SIZE)
180.      {
181.      repeat:
182.      req = request+NR_REQUEST;
183.      while (--req >= request)
184.      if (req->dev<0)
185.      break;
186.      if (req < request) {
187.      sleep_on(&wait_for_request);
188.      goto repeat;
189.      }
          //向空闲请求项中填写请求信息
190.      req->dev = dev;
191.      req->cmd = rw;
192.      req->errors = 0;
193.      req->sector = b[i] << 1;
194.      req->nr_sectors = 2;
195.      req->buffer = buf;
196.      req->waiting = current;
197.      req->bh = NULL;
198.      req->next = NULL;
199.      current->state = TASK_UNINTERRUPTIBLE;
200.      add_request(major+blk_dev,req);
201.      schedule();
202.      }
203.      }
```

本程序是用于处理块设备的所有读/写操作。代码中第 67 行的 WRITEA/READA 是一特殊情况，它们并不是真正需要，因此如果缓冲区已上锁，我们就不管它而退出，否则的话就执行一般的读/写操作。

7.7 ramdisk.c 文件

ramdisk.c 是内存虚拟盘驱动程序文件。对它的读写操作和其他块设备一样，但是由于在实际操作中不涉及外部控制器的相关操作，因而实现方式简单。

对虚拟盘设备的读写访问操作原则上完全按照对普通磁盘的操作进行，也需要按照块设备的访问方式对其进行读写操作。由于在实现上不牵涉与外部控制器或设备进行同步操作，因此其实现方式比较简单。对于数据在系统与设备之间的“传送”只需执行内存数据块复制操作即可。

本程序包含 3 个函数。rd_init()会在系统初始化时被 init/main.c 程序调用，用于确定虚拟盘在物理内存中的具体位置和大小；do_rd_request()是虚拟盘设备的请求项操作函数，对当前请求项实现虚拟盘数据的访问操作；rd_load()是虚拟盘根文件加载函数。在系统初始化阶段，该函数被用于尝试从启动引导盘上指定的磁盘块位置开始处把一个根文件系统加载到虚拟盘中。在函数中，这个起始磁盘块位置被定为 256。

在进行正常的根文件系统加载之前，系统会首先执行 rd_load()函数，试图从磁盘的第 257 块中读取根文件系统超级块。若成功，就把该根文件映像文件读到内存虚拟盘中，并把根文件系统设备标志 ROOT_DEV 设置为虚拟盘设备（0x0101），否则退出 rd_load()，系统继续从别的设备上执行根文件加载操作。ramdisk.c 文件的详细代码注释如下：


```
1.      #include <string.h>                          //字符串头文件
2.      #include <linux/config.h>                    //内核配置头文件
3.      #include <linux/sched.h>                     //调度程序头文件
4.      #include <linux/minix_fs.h>                  //最小文件系统头文件
5.      #include <linux/fs.h>                        //文件系统头文件
6.      #include <linux/kernel.h>                    //内核头文件
7.      #include <asm/system.h>                      //系统头文件
8.      #include <asm/segment.h>                     //段操作头文件
9.      #include <asm/memory.h>                      //内存拷贝头文件
10.     #define MAJOR_NR 1                           //内存主设备号是 1
11.     #include "blk.h"
12.     char   *rd_start;                            //虚拟盘在内存中的起始位置
13.     int     rd_length = 0;                       //虚拟盘所占内存大小
      //执行虚拟盘读写操作
14.     void do_rd_request(void)
15.     {
16.      int    len;
17.      char  *addr;
18.      INIT_REQUEST;                               //检测请求的合法性
      //下面语句用来得到 ramdisk 的起始扇区对应的内存起始位置和内存长度
19.     addr = rd_start + (CURRENT->sector << 9);
20.     len = CURRENT->nr_sectors << 9;
      //如果子设备号不等于 1 或对应内存起始位置大于虚拟盘末尾，则结束该请求
      //并跳转到 repeat 处
21.     if ((MINOR(CURRENT->dev) != 1) || (addr+len > rd_start+rd_length)) {
22.     end_request(0);
23.     goto repeat;
24.     }
      //如果是写命令，则将请求项中长度为 len 字节的缓冲区内容复制到 addr 处
```

```
25.         if (CURRENT-> cmd == WRITE) {
26.         (void ) memcpy(addr,
27.         CURRENT->buffer,
28.         len);
          //如果是读命令，则将 addr 开始长度为 len 字节的内容复制到请求项中缓冲区中
29.         } else if (CURRENT->cmd == READ) {
30.         (void) memcpy(CURRENT->buffer,
31.         addr,
32.         len);
33.         } else                                          //否则显示命令不存在，死机
34.         panic("unknown ramdisk-command");
35.         end_request(1);
36.         goto repeat;
37.         }
38.         long rd_init(long mem_start, int length)
39.         {
40.          int     i;
41.          char  *cp;
42.         blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;
43.         rd_start = (char *) mem_start;
44.         rd_length = length;
45.         cp = rd_start;
46.         for (i=0; i < length; i++)
47.         *cp++ = '\0';
48.         return(length);
49.         }
50.         void rd_load(void)
51.         {
52.        struct buffer_head *bh;
53.         struct super_block       s;
54.         int             block = 256;
55.         int             i = 1;
56.         int             nblocks;
57.         char            *cp;
58.         if (!rd_length)                                 //如果 ramdisk 的长度是零，则返回退出
59.         return;
60.         printk("Ram disk: %d bytes, starting at 0x%x\n", rd_length,
61.                 (int) rd_start);                        //打印 ramdisk 的大小以及内存起始位置
62.                 if (MAJOR(ROOT_DEV) != 2)               //如果根文件设备不是软盘，则退出
63.         return;
64.         bh = breada(ROOT_DEV,block+1,block,block+2,-1); //读取软盘块 256+1,256,256+2 的内容
65.         if (!bh) {
66.         printk("Disk error while looking for ramdisk!\n");
67.         return;
68.         }
          //将 s 指向缓冲区中的磁盘超级块
69.         *((struct minix_super_block *) &s) = *((struct minix_super_block *) bh->b_data);
70.         brelse(bh);
71.         if (s.s_magic != MINIX_SUPER_MAGIC)
          //如果没有 ramdisk 映像文件，退出时执行通常的软盘引导
```

```
72.         return;
        //如果数据块数大于内存中虚拟盘所能容纳的块数，则不能加载，显示出错信息并返回
        //否则显示加载数据块信息
73.         nblocks = s.s_nzones << s.s_log_zone_size;
74.         if (nblocks > (rd_length >> BLOCK_SIZE_BITS)) {
75.         printk("Ram disk image too big!   (%d blocks, %d avail)\n",
76.         nblocks, rd_length >> BLOCK_SIZE_BITS);
77.         return;
78.         }
79.         printk("Loading %d bytes into ram disk... 0000k",
80.         nblocks << BLOCK_SIZE_BITS);
81.           cp = rd_start;                                    // cp 指向虚拟盘起始处
82.         while (nblocks) {
83.               if (nblocks > 2)                              //如果需读取的块数多于 3 块则采用超前预读方
                                                                //式读数据块
84.         bh = breada(ROOT_DEV, block, block+1, block+2, -1);
85.               else                                          //否则就单块读取
86.         bh = bread(ROOT_DEV, block);
87.         if (!bh) {
88.         printk("I/O error on block %d, aborting load\n",
89.         block);
90.         return;
91.         }
92.               (void) memcpy(cp, bh->b_data, BLOCK_SIZE);    //将缓冲区中的数据复制到 cp 处
93.               brelse(bh);                                   //释放缓冲区
94.         printk("\010\010\010\010\010%4dk",i);
95.               cp += BLOCK_SIZE;                             //虚拟盘指针向前移
96.         block++;
97.         nblocks--;
98.         i++;
99.         }
100.        printk("\010\010\010\010\010done \n");
101.          ROOT_DEV=0x0101;                                  //将 ROOT_DEV 设置为指向虚拟盘 ramdisk
102.        }
```

代码中第 38～49 行虚拟盘初始化函数。用来确定虚拟盘在内存中的起始地址，长度。并对整个虚拟盘区清零。返回内存虚拟盘 ramdisk 所需的内存量。第 50～102 行定义函数是用来加载根文件系统到 ramdisk，如果根文件系统设备是 ramdisk 的话，则尝试去加载它。root device 原先是指向软盘的，我们将它改成指向 ramdisk。

第8章 文件系统

本章所涉及的Linux内核代码主要包括Linux文件系统实现程序代码和块设备高速缓冲区管理程序代码（见表8-1）。其中，Minix文件系统的实现代码被独立放到一个名为minix的文件夹中（见表8-2）。

表8-1 Linux/fs目录

名　　称	大小（B）	修 改 时 间
buffer.c	10053	1992-02-21 21:09:52
block_dev.c	1933	1992-03-02 10:15:02
char_dev.c	3109	1992-03-02 10:16:20
exec.c	10116	1992-03-04 12:24:52
fcntl.c	1502	1992-02-21 02:04:34
file_table.c	122	1991-10-02 22:16:30
inode.c	4681	1992-03-06 10:39:20
ioctl.c	1131	1992-02-20 23:01:12
namei.c	9449	1992-03-07 01:04:10
open.c	3927	1992-03-03 05:52:08
pipe.c	2981	1992-03-06 21:04:16
read_write.c	2510	1992-03-03 06:16:10
select.c	6421	1992-02-20 23:03:26
stat.c	1681	1992-03-06 11:22:36
super.c	5874	1992-02-29 01:01:44
Makefile	6244	1992-03-06 05:36:44
minix	34699	1992-03-08 20:09:00

表8-2 Linux/fs/minix目录

名　　称	大　小（B）	修 改 时 间
bitmap.c	4175	1992-03-06 02:36:44
file_dev.c	2168	1992-03-06 02:56:00
Inode.c	3786	1992-03-02 07:15:18
minix_op.c	686	1992-03-03 06:18:40
namei.c	17892	1992-03-04 01:39:36
truncate.c	3043	1992-03-03 05:56:16
Makefile	2949	1992-03-06 05:36:48

8.1 总体功能

本章所注释说明的程序从内容上主要分为VFS虚拟文件系统程序和MINIX文件系统程序两大部分，其中所涉及的程序功能可分为四个部分。第一部分是实现了硬盘等设备数据高速存取函

数的高速缓冲区管理程序。第二部分重点描述了文件系统的低层通用函数。第三部分包括了对字符设备、管道、块读写文件数据访问的程序。第四部分描述了文件系统调用接口的实现。

8.1.1 VFS 虚拟文件系统

0.95 版本的 Linux 内核加入了对虚拟文件系统的支持，将 MINIX 文件系统的代码从原本的文件系统中分离出来，但此时的虚拟文件系统还并不完整，并且只支持 MINIX 文件系统，如图 8-1 所示。

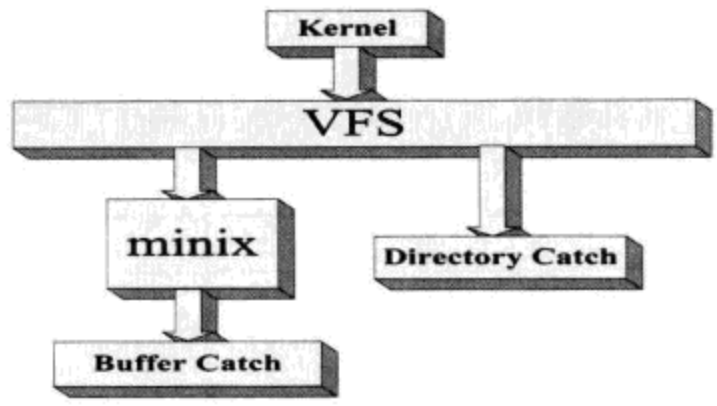


图 8-1 VFS 文件系统框架

Sun icrosystems 公司在定义网络文件系统时创造了虚拟文件系统 VFS（Virtual File Systems）。它是一种用于网络环境的分布式文件系统，是实现与操作系统使用不同文件系统的接口。

虚拟文件系统（VFS）是服务与物理文件系统之间的一个接口层，它抽象了 Linux 的每个文件系统的所有细节，使得不同的文件系统对于 Linux 核心以及系统中运行的其他进程来说都是相同的。VFS 只存在于内存中，在系统启动时建立，系统关闭时消亡，不存在于任何外存空间，并不是一种实际的文件系统。

有了 VFS，Linux 可以同时安装、支持许多不同类型的文件系统。VFS 拥有超级块、inode、文件操作函数入口等关于各种特殊文件系统的公共界面。实际文件系统的细节对系统核心和用户进程来说是透明的，都由 VFS 的公共界面来索引。下面介绍一些 VFS 的公共界面：

VFS 超级块，用来描述整个文件系统的组织结构和管理信息等，是文件系统中最重要的数据结构。VFS 超级块在具体的文件系统安装时被建立，并在文件系统卸载时被删除。超级块对象由 super_block 结构组成，该结构主要包括描述文件系统整体信息的域、用于管理超级块的域、具体文件系统相联系的域。

索引结点对象 inode，包含了文件系统处理文件所需的信息。每个文件都有唯一的索引结点，磁盘上存储的是具体文件系统的索引结点，使用的时候被调入内存，用于填写内存中 VFS 的索引结点，所以 VFS 的索引结点是动态结点。inode 结构主要包括描述文件信息的域、用于索引结点管理的域、用于索引结点操作的域等。

VFS 提供的文件操作函数入口主要包括操作超级块的函数 read_super()、put_super()、get_super()、sys_umount()及 sys_mount()等；操作文件索引结点的函数 read_inode()、write_inode()、iput ()、iget()等。

VFS 的主要功能为将设备与相应的文件系统相关联，记录文件系统的类型，文件通用操作的处理。对面向文件系统的操作进行处理时，VFS 把它们与控制文件、目录以及 inode 相关的物理文件系统相映射。

当发起一个面向文件的系统调用时，核心将调用 VFS 中处理与物理结构无关操作的函数，并且把它重定向为真实文件系统中处理那些与物理结构相关操作的函数。

8.1.2 Minix 文件系统

Minix 原来是荷兰阿姆斯特丹 Vrije 大学计算机科学系的 Andrew S. Tanenbaum 教授所发展的一个迷你版本的类 Unix 操作系统。Minix 中使用汇编语言编写了启动部分，其他部分都是使用 C

语言编写的，主要分为内核、档案管理和内存管理三部分。Linux 是受到 Minix 的影响而产生的，起初 Linux 的文件系统就使用了 Minix 的文件系统。

Minix 文件系统主要包括 6 个组成部分：引导块、超级块、逻辑块位图、i 结点位图、i 结点、数据区。

引导块是一系列执行代码和数据，在计算机加电启动时，这些代码和数据可由 ROM BIOS 自动读入。引导块使用一个盘块，逻辑块号为 0。但在不用于引导的盘片中，这一盘块可以不含任何代码，不过任何盘片中都必须要含有引导块空间，用来保持 Minix 文件系统格式的统一性。如果内核文件放在文件系统中，那么就可以在文件系统所在的设备的第一个块（引导块空间）存放实际的引导程序，并由它来取得和加载文件系统中的内核映像文件。

超级块里面装入了文件系统的布局信息，它的主要功能是给出文件系统不同部分的大小。超级块使用一个盘块，逻辑块号为 1。

i 结点位图说明了 i 结点的使用情况，每个位代表一个 i 结点。i 结点位图块最多占 8 个逻辑块，超级块字段中 s_imap_blocks 的值记录了 i 结点位图块的个数，逻辑块号从 2 开始。

逻辑块位图说明了盘上每个数据块的使用状况。逻辑块位图块最多可以有 8 个逻辑块，超级块字段中 s_zmap_blocks 的值记录了逻辑块的个数，逻辑块号紧跟 i 结点位图块号。除了第一个位（位 0）以外，逻辑块位图中每个位依次代表盘上数据区中的一个逻辑块。因此逻辑块位图的位 1 代表盘上数据区中的第一个数据盘块，而不是盘上的一个磁盘块（引导块）。当一个数据盘块被使用时，逻辑块位图中的相应位将被置位。由于当所有磁盘数据盘块都被占用时查找空闲盘块的函数会返回 0 值，因此逻辑块位图最低位（位 0）闲置不用，并且在创建文件系统时会预先将其设置为 1。

i 结点部分是文件系统中文件和目录名的索引结点集合，每个 i 结点结构中存放着对应文件的相关信息。文件中的数据是存放在磁盘块的数据区中的，数据区被分为很多个数据块，又称为数据逻辑块，一个逻辑块为一个盘块，一个盘块占 1024 个字节，也就是两个扇区。而一个文件名则通过对应的 i 结点与这些数据磁盘块相联系，这些盘块的号码就存放在 i 结点的逻辑块数组中。

另外， /dev/目录下的设备文件的文件长度为 0，并不占用数据盘块。设备文件所对应的 i 结点只保存其所定义设备的属性和设备号，其中 i 结点对应的逻辑块数组 0 中存放了设备文件所对应的设备号。

8.1.3 文件系统程序功能

0.95 版的 Linux 内核将文件系统的程序划分为两部分，一部分实现了对虚拟文件系统的支持；另一部分用于实现与 Minix 文件系统相关的操作，该部分被单独放在一个名为 Minix 的文件夹内。

虚拟文件系统从功能上可以划分为四个部分。第一部分是高速缓冲区的管理程序，该程序主要实现了高速缓冲区缓存的功能。第二部分是文件系统低层程序，主要实现了对 i 结点的分配与释放、文件路径名与 i 结点的映射、文件系统超级块的处理以及文件系统的选择处理功能。第三部分是文件数据的访问操作程序，主要实现了对字符设备、管道、块设备、正规文件的数据访问功能。第四部分是文件系统的上层操作程序，主要实现了文件的打开、关闭、创建等有关文件操作的系统调用。

实现高速缓冲区管理的程序是 buffer.c，该程序包含一系列接口函数，用于其他程序进行块的读写操作。主要包括块读取函数 bread()、块提前预读取函数 breada()、页块读取函数 bread_page()、块释放函数 brelse()等。

文件系统底层程序主要包括 inode.c 程序、namei.c 程序、super.c 程序。

inode.c 程序主要包括写 i 结点信息函数 write_inode()、i 结点信息读取函数 read_inode()、分配 i 结点函数 iget()和释放内存 i 结点函数 iput()以及取文件数据块 block 在设备上对应的逻辑块号的函数 bmap()。

namei.c 程序主要包括获取路径名对应 i 结点的函数 namei()、创建文件结点的系统调用函数 sys_mknod()、创建目录的系统调用函数 sys_mkdir()等。

super.c 程序主要包括查找设备超级块的函数 get_super()、释放设备超级块函数 put_super()、加载文件系统的函数 sys_mount()以及卸载设备文件系统的函数 sys_umount()等。

文件数据的访问操作程序主要涉及 6 个文件：block_dev.c、char_dev.c、file_dev.c、pipe.c 和 read_wirte.c。其中 read_wirte.c 是其他程序的上层访问接口，不同文件类型分别调用不同的访问函数。

block_dev.c 主要包括块设备文件数据读取函数 block_read()和块设备文件数据写函数 block_read()。

char_dev.c 主要包括控制台终端读写操作函数 rw_tty()、串口终端读写操作函数 rw_ttyx()、内存数据操作函数 rw_mem()及字符设备读写操作接口函数 rw_char()等。read_wirte.c 中的系统调用会通过函数 rw_char()来进行具体字符设备的读写操作。

pipe.c 主要包括读取管道数据的函数 pipe_read()和向管道写入数据的函数 pipe_read()。

文件系统上层操作程序主要实现了文件和目录管理的系统调用函数。主要包含 open.c 程序、exec.c 程序、fcntl.c 程序、ioctl.c 程序、stat.c 程序、select.c 程序。

open.c 程序实现了文件操作相关的系统调用。主要包括文件的创建系统调用函数 sys_creat()、文件的打开系统调用函数 sys_open()、文件的关闭系统调用函数 sys_close()、修改文件宿主系统调用 sys_chown()、改变当前工作目录系统调用 sys_chdir()等。

exec.c 程序的主要功能为加载和执行可执行文件和 shell 脚本文件。其中主要包括了 execve()系统中断调用函数 do_execve()。

fcntl.c 程序主要包括文件句柄复制系统调用函数 dup()和 dup()、文件控制系统调用函数 fcntl()。

ioctl.c 程序实现了输入和输出控制系统调用。主要包括系统输入输出控制系统调用函数 sys_ioctl()。

stat.c 程序的功能为读取文件状态信息。主要包括利用文件名读取文件状态信息的系统调用函数 sys_stat()、利用文件句柄读取文件状态信息的 sys_fstat()。

select.c 程序实现了 Linux 的选择处理功能。主要实现了选择处理系统调用函数 sys_select()。

8.2 重要数据结构

本节主要内容为介绍与文件系统相关的数据结构，对各个数据结构的代码进行分析。本节介绍的数据结构列表如表 8-3 所示。

表 8-3 文件系统相关数据结构列表

名　　称	功 能 描 述
buffer_head	缓冲头结构
inode	虚拟文件系统 i 结点结构
super_block	虚拟文件系统超级块结构
minix_inode	Minix 文件系统 i 结点结构
minix_super_block	Minix 文件系统超级块结构

1. buffer_head

缓冲区的低端被划分为一个个的缓冲头结构，用于描述对应缓冲块的属性和把所有缓冲头连接成链表。缓冲头结构如下：

```
struct buffer_head {
  char * b_data;                          //指向缓冲块的指针
  unsigned long b_blocknr;                //缓冲数据的逻辑块号
  unsigned short b_dev;                   //块设备号
  unsigned char b_uptodate;               //数据有效标志
  unsigned char b_dirt;                   //数据修改标志
  unsigned char b_count;                  //数据被使用的进程数
  unsigned char b_lock;                   //缓冲块锁定标志
  struct task_struct * b_wait;            //指向等待该缓冲块的任务指针
  struct buffer_head * b_prev;            //Hash 表前向指针
  struct buffer_head * b_next;            //Hash 表后向指针
  struct buffer_head * b_prev_free;       //空闲链表前向指针
  struct buffer_head * b_next_free;       //空闲链表后向指针
};
```

2. inode

索引结点对象 inode，包含了文件系统处理文件所需的信息。每个文件都有唯一的索引结点，磁盘上存储的是具体文件系统的索引结点，使用的时候被调入内存，用于填写内存中 VFS 的索引结点，所以 VFS 的索引结点是动态结点。

虚拟文件系统 i 结点结构如下：

```
struct inode {
  dev_t     i_dev;                        //所属设备号
  ino_t     i_ino;                        //i 结点号
  umode_t i_mode;                         //文件类型和属性
  nlink_t   i_nlink;                      //文件目录项链接数
  uid_t     i_uid;                        //文件宿主的用户 id
  gid_t     i_gid;                        //文件宿主的组 id
  dev_t     i_rdev;                       //挂载的设备号
  off_t     i_size;                       //文件长度
  time_t    i_atime;                      //访问时间
  time_t    i_mtime;                      //修改时间
  time_t    i_ctime;                      //改变时间
  unsigned long i_data[16];               //文件所占用的逻辑块号数组
  struct inode_operations * i_op;         //指向 i 结点操作结构的指针
  struct super_block * i_sb;              //所属设备的超级块指针
  struct task_struct * i_wait;            //指向等待该 i 结点的任务指针
  struct task_struct * i_wait2;           //指向等待该 i 结点的任务指针（用于管道）
  unsigned short i_count;                 //引用计数
  unsigned char i_lock;                   //锁定标志
  unsigned char i_dirt;                   //修改标志
  unsigned char i_pipe;                   //管道 i 结点标志
  unsigned char i_mount;                  //i 结点是否为其他文件系统安装点标志
  unsigned char i_seek;                   //搜寻标志
  unsigned char i_update;                 //更新标志
};
```

3. super_block

虚拟文件系统超级块，用来描述整个文件系统的组织结构和管理信息等，是文件系统中最重要的数据结构。虚拟文件系统超级块在具体的文件系统安装时被建立，并在文件系统卸载时被删除。虚拟文件系统超级块如下：

```
struct super_block {
    unsigned short s_ninodes;                    //结点数
    unsigned short s_nzones;                     //逻辑块数
    unsigned short s_imap_blocks;                //i 结点位图所占用的数据块数
    unsigned short s_zmap_blocks;                // 逻辑块位图所占用的数据块数
    unsigned short s_firstdatazone;              // 第一个数据逻辑块号
    unsigned short s_log_zone_size;              // log(数据块数/逻辑块)
    unsigned long s_max_size;                    // 文件最大长度
    unsigned short s_magic;                      // 文件系统魔数
    struct buffer_head * s_imap[8];              // i 结点位图缓冲块指针数组
    struct buffer_head * s_zmap[8];              // 逻辑块位图缓冲块指针数组
    unsigned short s_dev;                        // 超级块所在的设备号
    struct inode * s_covered;                    // 被安装的文件系统根目录的 i 结点
    struct inode * s_mounted;                    // 被安装到的 i 结点
    unsigned long s_time;                        // 修改时间
    struct task_struct * s_wait;                 // 等待该超级块的进程
    unsigned char s_lock;                        // 锁定标志
    unsigned char s_rd_only;                     // 只读标志
    unsigned char s_dirt;                        // 已修改标志
};
```

4. minix_inode

Minix 文件系统 i 结点存放着 Minix 文件系统中文件或目录的索引结点，每个 Minix 文件或目录都有一个 i 结点。每个 i 结点结构中存放着对应文件的相关信息。

Minix 文件系统 i 结点结构如下：

```
struct minix_inode {
    unsigned short i_mode;                       //文件类型和属性
    unsigned short i_uid;                        //文件宿主的用户 id
    unsigned long i_size;                        //文件大小
    unsigned long i_time;                        //修改时间
    unsigned char i_gid;                         //文件宿主的组 id
    unsigned char i_nlinks;                      //指向该 i 结点的目录项数
    unsigned short i_zone[9];                    //文件所占用的逻辑块号数组
};
```

5. minix_super_block

Minix 超级块用于存放盘设备上文件系统结构的信息，并说明各部分的大小。

Minix 超级块结构如下：

```
struct minix_super_block {
    unsigned short s_ninodes;                    // 结点数
    unsigned short s_nzones;                     // 逻辑块数
    unsigned short s_imap_blocks;                // i 结点位图所占用的数据块数
    unsigned short s_zmap_blocks;                // 逻辑块位图所占用的数据块数
    unsigned short s_firstdatazone;              // 第一个数据逻辑块
    unsigned short s_log_zone_size;              // log(数据块数/逻辑块)
```

```
    unsigned long s_max_size;                    // 文件最大长度
    unsigned short s_magic;                      // 文件系统魔数
};
```

8.3 buffer.c 文件

buffer.c 文件用于对高速缓冲区进行操作和管理。高速缓冲区在提高对块设备的访问效率和增加数据共享方面起着重要的作用。

8.3.1 功能描述

程序对块设备的读出和写入基本都需要通过高速缓冲区来进行操作。高速缓冲区在物理内存上占据内核模块和主内存区之间的内存空间，主要由缓冲块和指向缓冲块的缓冲头组成，每个缓冲块的大小和磁盘逻辑块的大小相同。缓冲头和缓冲块是在缓冲区初始化的过程中，从缓冲区的两端开始分别划分出来的，最终无法划分出缓冲块的内存空间弃之不用，划分过程如图 8-2 所示。

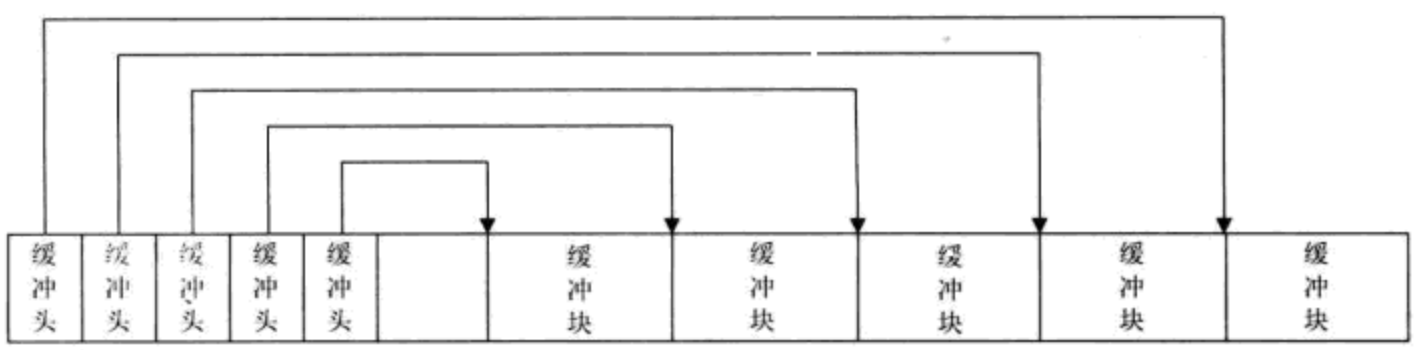


图 8-2 高速缓冲区初始化

整个缓冲区则采用散列表和空闲队列进行操作和管理，其中散列表是以设备号和逻辑块号的组合为散列函数，由缓冲头链接而成的表，空闲链表则是由缓冲头链接而成的双向链表。

本程序实现了对缓冲区高速缓存的操作功能。程序中的函数 buffer_init()实现了初始化缓冲区的功能，从缓冲区始端开始建立缓冲头，同时从缓冲区末端开始划分对应于缓冲头的缓冲块，每个缓冲块大小 1KB。建立缓冲头的同时将其组成双向链表。

函数 sync_buffers()、sys_sync()、sys_sync()实现了缓冲区一系列的同步功能。函数 sync_buffers()遍历空闲缓冲块链表，将被修改的缓冲块中的数据写入盘中，使得高速缓冲区的数据与设备的数据同步。函数 sys_sync()调用 i 结点同步函数，把内存 i 结点表中所有修改过的 i 结点写入高速缓冲，然后调用缓冲区同步函数，把高速缓冲中的数据与设备中的同步。函数 sync_dev()调用缓冲区数据同步函数，同步指定设备缓冲区与设备中的数据，然后把内存中 i 结点表数据写入高速缓冲中，之后再次调用缓冲区数据同步函数对指定设备进行写盘操作。

对缓冲块链表和 hash 队列的操作则是通过函数 remove_from_free_list()、remove_from_queues()、put_first_free()等实现的。函数 remove_from_free_list()从空闲缓冲块表中移除缓冲块。函数 remove_from_queues()从 hash 队列和空闲缓冲队列中移走缓冲块。函数 put_first_free()将指定缓冲块插入空闲缓冲块链表头。函数 put_last_free()将指定缓冲块插入空闲缓冲块链表末尾。函数 insert_into_queues()将指定缓冲块插入空闲缓冲块链表尾并放入 hash 队列中。函数 find_buffer()遍历相应的 hash 队列，寻找需要的缓冲区块并返回其指针。函数 get_hash_table()获得指定设备和指定块的缓冲块。

内核程序在使用高速缓冲区中的缓冲块时，是通过指定设备号和所要访问设备数据的逻辑块号，调用 bread()、bread_page()或 breada()函数进行操作的。这几个函数都使用了缓冲区搜索管理函数 getblk()。函数 getblk()用于获得指定设备和指定块的缓冲块，如果没有，就在高速缓冲区中建立一个对应的新缓冲块；函数 brelse()用于释放指定的缓冲块；函数 bread()用于从设备上读取指定的数据块并返回含有其数据的缓冲块；函数 bread_page()用于读取指定设备的四个缓冲块到指定的内存位置；函数 breada()可以像 bread 一样使用，但会同时额外地预读其他一些块。

8.3.2 代码注释

buffer.c 文件的详细代码注释如下：

```
1.      #include <stdarg.h>              //标准参数头文件。以宏的形式定义变量参数列表
2.      #include <linux/config.h>        //内核配置头文件。定义键盘语言和硬盘类型可选项
3.      #include <linux/sched.h>         //调度程序头文件。定义了 task_struct 和任务 0 的数据。一些关
                                         //于描述符参数设置和获取以及 switch_to()的嵌入式汇编函数宏
4.      #include <linux/kernel.h>        //内核头文件。定义了一些内核常用函数的原型
5.      #include <asm/system.h>          //系统头文件。定义了设置或修改描述符、中断门等的嵌入式汇编宏
6.      #include <asm/io.h>              //IO 头文件。定义了对硬件 IO 端口访问的嵌入式汇编宏函数：
                                         //outb()、inb()以及 outb_p()和 inb_p()
7.      extern int end;                  //变量 end 表明内核代码的末端位置，即高速缓冲区的开始位置
8.      struct buffer_head * start_buffer = (struct buffer_head *) &end;    //start_buffer 指向第一个缓冲块头
9.      struct buffer_head * hash_table[NR_HASH];          //声明哈希数组，NR_HASH 为 307
10.     static struct buffer_head * free_list;             //声明空闲缓冲块链表头指针
11.     static struct task_struct * buffer_wait = NULL;    //声明因等待空闲缓冲块而睡眠的任务队列头指针
12.     int NR_BUFFERS = 0;              //缓冲区块数，初始值为 0
      //等待指定缓冲块解锁函数
13.   static inline void wait_on_buffer(struct buffer_head * bh)
14.     {
15.         cli();                       //关中断
16.         while (bh->b_lock)           //判断缓冲块是否上锁，等待解锁
17.         sleep_on(&bh->b_wait);
18.         sti();                       //开中断
19.     }
      //缓冲区数据同步函数，同步缓冲区与设备中的数据
20.   static void sync_buffers(int dev)
21.   {
22.         int i;
23.         struct buffer_head * bh;
24.         bh = free_list;              //bh 指向空闲缓冲块链表头
25.    for (i=0 ; i<NR_BUFFERS ; i++,bh = bh->b_next_free) {
26.         #if 0
27.         if (dev && (bh->b_dev != dev))   //判断是否指定了设备，及该缓冲块是否为指定
                                             //设备的缓冲块
28.
29.             continue;                    //不是则继续判断一下缓冲块
30.         #endif
31.             wait_on_buffer(bh);          //如果缓冲块已上锁，则等待缓冲块解锁
32.         #if 0
33.         if (dev && (bh->b_dev != dev))   //在睡眠过程中该缓冲块可能被释放或者挪
                                             //为他用，所以要继续执行前再次判断该缓冲块
```

```
                                                    //是否为指定设备的缓冲块
34.                     continue;
35.                 #endif
36.                     if (bh->b_dirt)             //判断该缓冲块是否修改过
37.                         ll_rw_block(WRITE,bh);//产生写设备块请求
38.             }
39.         }
    //设备数据同步函数。首先调用 i 结点同步函数，把内存 i 结点表中所有修改过的 i 结点写入高速
      //缓中.然后调用缓冲区同步函数，把高速缓冲中的数据与设备中的同步
40.         int sys_sync(void)
41.         {
42.
43.             sync_inodes();
44.                                                 /*将 i 结点写入缓冲区中*/
45.             sync_buffers(0);                    //将高速缓冲中的数据于设备中的同步
46.             return 0;
47.         }
48.
49.         int sync_dev(int dev)
50.         {
51.             sync_buffers(dev);                  //同步指定设备 dev 的缓冲区与该设备中的数
                                                    //据。这次缓冲区同步操作使内核中许多脏块变
                                                    //干净，从而令 i 结点同步操作有更高的执行效率
52.
53.             sync_inodes();                      //将 i 结点数据写入高速缓冲
54.             sync_buffers(dev);                  //再次将指定设备 dev 的缓冲区与该设备中的数据同
                                                    //步。本次缓冲区同步操作把那些由于 i 结点同步操作
                                                    //而又变脏的缓冲块与设备中数据同步
55.
56.         return 0;
57.         }
    //使指定设备在高速缓冲区中的数据无效函数
58.         void inline invalidate_buffers(int dev)
59.         {
60.             int i;
61.             struct buffer_head * bh;
62.
63.             bh = start_buffer;
64.             for (i=0 ; i<NR_BUFFERS ; i++,bh++) {       //扫描高速缓冲区所有缓冲块
65.                 if (bh->b_dev != dev)                   //判定该缓冲块是否为指定设备缓冲块，如果不
                                                            //是，则
66.                     continue;                           //继续扫描下一块
67.                 wait_on_buffer(bh);                     //如果缓冲块已上锁，则等待缓冲块解锁
68.                 if (bh->b_dev == dev)                   //因为经过睡眠等待，所以需要再次判定该缓冲
                                                            //块是否为指定设备的缓冲块
69.
70.                 bh->b_uptodate = bh->b_dirt = 0; //复位该缓冲块的有效标志和修改标志，使缓
71.                                                  //冲块数据无效
72.         }
73.     }
```

```
        //检查磁盘更换函数，如果磁盘被更换就使该磁盘对应的高速缓冲区无效
74.         void check_disk_change(int dev)
75.         {
76.             int i;
77.
78.             if (MAJOR(dev) != 2)                    //检查设备 dev 是否为软盘，如果不是则退出
79.             return;
80.             if (!floppy_change(dev & 0x03))         //检查软盘是否被更换，如果没有则退出
81.                 return;
82.             for (i=0 ; i<NR_SUPER ; i++)            // 扫描所有超级块，寻找该设备的超级块
83.             if (super_block[i].s_dev == dev)
84.                 put_super(super_block[i].s_dev);    //释放该设备对应的 i 结点位图和逻辑块位图
85.                                                     //所占高速缓冲区
86.             invalidate_inodes(dev);                 //使该设备的 i 结点所占高速缓冲块无效
87.             invalidate_buffers(dev);                //使该设备的数据块信息所占的高速缓冲块无效
88.         }
        //hash 函数和 hash 表项的计算宏定义
89.         #define _hashfn(dev,block) (((unsigned)(dev^block))%NR_HASH)
90.         #define hash(dev,block) hash_table[_hashfn(dev,block)]
91.
        //从 hash 队列移走缓冲块函数
92.         static inline void remove_from_hash_queue(struct buffer_head * bh)
93.         {
        //从 hash 队列中移除缓冲块
94.         if (bh->b_next)
95.             bh->b_next->b_prev = bh->b_prev;
96.             if (bh->b_prev)
97.             bh->b_prev->b_next = bh->b_next;
98.             if (hash(bh->b_dev,bh->b_blocknr) == bh)
99.             hash(bh->b_dev,bh->b_blocknr) = bh->b_next;
100.
101.        bh->b_next = bh->b_prev = NULL;             //将该缓冲块完全从队列中脱离
102.     }
        //从空闲缓冲块表中移除缓冲块函数
103.        static inline void remove_from_free_list(struct buffer_head * bh)
104.        {
105.     if (!(bh->b_prev_free) || !(bh->b_next_free))  //因为空闲缓冲块队列是双向循环链表结构
106.             panic("Free block list corrupted");    //因此该缓冲块任一方向断链，则表明该链表结构
                                                       //被毁坏
        //从空闲队列中移除缓冲块
107.        bh->b_prev_free->b_next_free = bh->b_next_free;
108.        bh->b_next_free->b_prev_free = bh->b_prev_free;
109.            if (free_list == bh)                   //如果空闲链表头指向本缓冲块，则让其指向下
                                                       //一缓冲块
110.            free_list = bh->b_next_free;
111.            bh->b_next_free = bh->b_prev_free = NULL;       //将该缓冲块完全从队列中脱离
112.        }
        //从 hash 队列和空闲缓冲队列中移走缓冲块函数
113.        static inline void remove_from_queues(struct buffer_head * bh)
114.     {
```

```
115.            remove_from_hash_queue(bh);          //从 hash 队列中移走缓冲块
116.            remove_from_free_list(bh);           //从空闲缓冲队列中移走缓冲块
117.      }
        //将指定缓冲块插入空闲缓冲块链表头函数
118.      static inline void put_first_free(struct buffer_head * bh)
119.        {
120.            if (!bh || (bh == free_list))        //如果该缓冲块不存在或者已经是链表头，则退出
121.            return;
122.        remove_from_free_list(bh);               //将该缓冲块从空闲缓冲块链表中移除
        //将该缓冲块插入链表头
123.            bh->b_next_free = free_list;
124.            bh->b_prev_free = free_list->b_prev_free;
125.            free_list->b_prev_free->b_next_free = bh;
126.            free_list->b_prev_free = bh;
127.            free_list = bh;
128.      }
        //将指定缓冲块插入空闲缓冲块链表末尾
129.      static inline void put_last_free(struct buffer_head * bh)
130.        {
131.            if (!bh)                             //如果该缓冲块不存在，则返回
132.            return;
133.        if (bh == free_list) {                   //如果该缓冲块是空闲缓冲块链表头，则将链表
                                    //头指针指向该缓冲块的下一缓冲块，这样该缓冲块就变为链表尾
134.                free_list = bh->b_next_free;     //
135.            return;
136.        }
137.            remove_from_free_list(bh);           //将该缓冲块从空闲缓冲块链表中移除
        //将该缓冲块插入链表尾
138.            bh->b_next_free = free_list;
139.            bh->b_prev_free = free_list->b_prev_free;
140.            free_list->b_prev_free->b_next_free = bh;
141.            free_list->b_prev_free = bh;
142.        }
        //将指定缓冲块插入空闲缓冲块链表尾并放入 hash 队列中
143.      static inline void insert_into_queues(struct buffer_head * bh)
144.      {
        //将该缓冲块插入空闲链表尾
145.            bh->b_next_free = free_list;
146.            bh->b_prev_free = free_list->b_prev_free;
147.            free_list->b_prev_free->b_next_free = bh;
148.            free_list->b_prev_free = bh;
        //如果该缓冲块对应一个设备，则将其插入相应的 hash 队列中去
149.            bh->b_prev = NULL;
150.            bh->b_next = NULL;
151.        if (!bh->b_dev)                          //如果该缓冲块没有相应的设备，则返回
152.            return;
        //将该缓冲块插入相应的 hash 队列的队首
153.            bh->b_next = hash(bh->b_dev,bh->b_blocknr);
154.            hash(bh->b_dev,bh->b_blocknr) = bh;
155.            bh->b_next->b_prev = bh;
```

```
156.      }
      //寻找指定设备和指定块的缓冲区块
157.      static struct buffer_head * find_buffer(int dev, int block)
158.      {
159.          struct buffer_head * tmp;
      //遍历相应的 hash 队列，寻找需要的缓冲区块并返回其指针。否则，返回 NULL
160.      for (tmp = hash(dev,block) ; tmp != NULL ; tmp = tmp->b_next)
161.          if (tmp->b_dev==dev && tmp->b_blocknr==block)
162.                  return tmp;
163.                  return NULL;
164.      }
      //获得指定设备和指定块的缓冲块
165.    struct buffer_head * get_hash_table(int dev, int block)
166.    {
167.          struct buffer_head * bh;
168.
169.          for (;;) {
170.              if (!(bh=find_buffer(dev,block)))  //寻找指定设备和指定块的缓冲区块，如果没找
                                                     //到则返回 NULL
171.              return NULL;
172.              bh->b_count++;                     //对该缓冲块增加引用计数
173.              wait_on_buffer(bh);                //如果该缓冲块被上锁，则等待其解锁
174.              if (bh->b_dev == dev && bh->b_blocknr == block) { //因为经过睡眠等待，所以需要
175.                                                 //再次判定该缓冲块是否为需要的缓冲块
176.              put_last_free(bh);                 //如果该缓冲块为需要的缓冲块，则放置空闲链
                                                     //表尾
177.                  return bh;
178.              }
179.              bh->b_count--;                     //如果该缓冲块不是需要的缓冲块，则取消对它
                                                     //的引用计数，重新寻找

180.      }
181.    }
      //同时判断缓冲块的修改标志和锁定标志的宏定义，其中定义修改标志的权重要比锁定标志大
182.        #define BADNESS(bh) (((bh)->b_dirt<<1)+(bh)->b_lock)
      //取高速缓冲区中指定的缓冲块函数。获得指定设备和指定块的缓冲块，如果没有，就在高速缓
       //冲区中建立一个对应的新缓冲块
183.    struct buffer_head * getblk(int dev,int block)
184.    {
185.          struct buffer_head * bh, * tmp;
186.          int buffers;
187.
188.          repeat:
189.          if (bh = get_hash_table(dev,block))    //获取指定的缓冲块，如果成功，返回该缓冲块
                                                     //头指针
190.          return bh;
191.      buffers = NR_BUFFERS;                      //定义存储缓冲块个数的变量，为下面的循环做
                                                     //计数
192.          tmp = free_list;                       //tmp 指向空闲链表头
      //在空闲链表中取得一个缓冲块作为指定设备上的块所对应的缓冲块
```

```
193.        do {
194.                tmp = tmp->b_next_free;                  //tmp 指向下一个缓冲头
195.                    if (tmp->b_count)                    //如果该缓冲块正在被使用，则继续扫描下一个
                                                             //缓冲块
196.                    continue
197.                if (!bh || BADNESS(tmp)<BADNESS(bh)) { //如果 bh 为空或 tmp 所指的缓冲头修改
198.                        bh = tmp;                        //和锁定标志的权重小于 bh 的，则将 bh 指向 tmp
                                                             //所指向的缓冲头
199.                    if (!BADNESS(tmp))                   //如果 tmp 所指缓冲块既没有被修改也没有被锁
                                                             //定，则说明已为指定设备上的块取得相应的高
                                                             //速缓冲块，退出循环
200.                        break;
201.                    }
202.                    if (tmp->b_dirt)                     //如果 tmp 所指缓冲块被修改，则产生写设备块
                                                             //请求
203.                    ll_rw_block(WRITEA,tmp);
        //重复操作直到我们找到合适的缓冲块
204.        } while (buffers--);
205.                if (!bh) {                               //如果所有缓冲块都在被使用，则睡眠，然后重
                                                             //复上面循环再次寻找空闲缓冲块
206.                sleep_on(&buffer_wait);
207.                goto repeat;
208.                }
209.                wait_on_buffer(bh);                      //如果缓冲块被上锁，等待其解锁
210.                if (bh->b_count)                         //在睡眠过程中如果该缓冲块被其他任务使用，
                                                             //则重复以上过程
211.                goto repeat;
212.            while (bh->b_dirt) {                         //判定该缓冲块是否被修改
213.                        sync_dev(bh->b_dev);             //如果该缓冲块被修改，则同步该缓冲块所属设
                                                             //备的数据
214.                wait_on_buffer(bh);                      //如果缓冲块被上锁，等待其解锁
215.                if (bh->b_count)                         //在睡眠过程中如果该缓冲块被其他任务使用，
                                                             //则重复以上过程
216.                            goto repeat;
217.                }
218.                if (find_buffer(dev,block))              //如果指定设备和块的缓冲块已经被加入到高速
                                                             //缓冲中，则重复上述操作
219.                goto repeat;
220.                bh->b_count=1;                           //置引用计数为 1
221.                bh->b_dirt=0;                            //复位修改标志
222.                bh->b_uptodate=0;                        //复位更新标志
223.                remove_from_queues(bh);                  //从 hash 队列和空闲链表中移除该缓冲块
224.                bh->b_dev=dev;                           //置设备号为指定设备号
225.                bh->b_blocknr=block;                     //置逻辑块号为指定块号
226.                insert_into_queues(bh);                  //将该缓冲块重新插入空闲链表和所属的 hash 队列中
227.            return bh;
228.        }
        //释放指定的缓冲块
229.        void brelse(struct buffer_head * buf)
230.        {
```

```
231.            if (!buf)                           //如果缓冲头指针无效，则返回
232.            return;
233.            wait_on_buffer(buf);                //如果该缓冲块上锁，等待其解锁
234.            if (!(buf->b_count--))              //引用计数减 1
235.            panic("Trying to free free buffer");
236.            wake_up(&buffer_wait);              //唤醒等待空闲缓冲区的进程
237.        }
        //从指定的设备上读取指定的块
238.        struct buffer_head * bread(int dev,int block)
239.    {
240.     struct buffer_head * bh;
241.
242.            if (!(bh=getblk(dev,block)))        //从高速缓冲中获取指定设备指定块的缓冲块
243.            panic("bread: getblk returned NULL\n");    //返回 NULL，表明出错，死机
244.            if (bh->b_uptodate)                 //如果该缓冲块的数据是有效的，直接返回
245.                    return bh;
246.            ll_rw_block(READ,bh);               //否则，产生读设备块请求
247.            wait_on_buffer(bh);                 //如果该缓冲块上锁，等待其解锁
248.            if (bh->b_uptodate)                 //如果该缓冲块已更新，返回缓冲头指针
249.            return bh;
250.            brelse(bh);                         //否则，表明操作失败，释放该缓冲区，返回 NULL
251.            return NULL;
252.    }
        //从 from 地址复制一个内存块到 to 位置
253.        #define COPYBLK(from,to) \
254.        __asm__("cld\n\t" \
255.                "rep\n\t" \
256.                "movsl\n\t" \
257.        ::"c" (BLOCK_SIZE/4),"S" (from),"D" (to) \
258.        :"cx","di","si")
        //读取指定设备的四个缓冲块到指定的内存位置
259.        void bread_page(unsigned long address,int dev,int b[4])
260.    {
261.     struct buffer_head * bh[4];
262.            int i;
263.            for (i=0 ; i<4 ; i++)               //循环读取四个块
264.            if (b[i]) {                         //判定块号是否有效
265.                    if (bh[i] = getblk(dev,b[i]))  //有效，则获取指定设备和块的缓冲块
266.                            if (!bh[i]->b_uptodate)       //判断该缓冲块数据是否有效
267.                            ll_rw_block(READ,bh[i]);  //无效则产生读设备请求
268.            } else
269.                            bh[i] = NULL;       //块号无效，则该块所对应缓冲头指针置为 NULL
        //将 4 个缓冲块顺序复制到内存指定地址处
270.            for (i=0 ; i<4 ; i++,address += BLOCK_SIZE)
271.            if (bh[i]) {                        //如果缓冲头指针不为 NULL，则开始操作
272.                    wait_on_buffer(bh[i]);      //如果缓冲块上锁，则等待其解锁
273.                            if (bh[i]->b_uptodate) //如果该缓冲块数据有效，则进行复制
274.                            COPYBLK((unsigned long) bh[i]->b_data,address);
275.                    brelse(bh[i]);              //释放该缓冲区
276.                    }
```

```
277.        }
    //从指定设备读取一些块，但只返回第一个指定块的缓冲头指针
278.        struct buffer_head * breada(int dev,int first, ...)
279.    {
280.      va_list args;
281.            struct buffer_head * bh, *tmp;
282.
283.            va_start(args,first);                          //取参数列表中第一个块号
284.            if (!(bh=getblk(dev,first)))                   //取指定设备和指定的第一个块号的缓冲块
285.            panic("bread: getblk returned NULL\n");
286.            if (!bh->b_uptodate)                           //判定缓冲块数据是否有效
287.            ll_rw_block(READ,bh);                          //无效，则发出读设备数据块的请求
288.      while ((first=va_arg(args,int))>=0) {                //读取参数列表其他块号
289.                  tmp=getblk(dev,first);                   //取指定设备和块号的缓冲块
290.            if (tmp) {
291.                  if (!tmp->b_uptodate)                    //判定缓冲块数据是否有效
292.                        ll_rw_block(READA,bh);             //无效，则发出读设备数据块的请求。注
                                                             //意，这里存在一个错误，bh 应该替换为 tmp
293.
294.                  tmp->b_count--;                          //将引用计数减 1
295.                  }
296.            }
297.            va_end(args);                                  //可变参数列表所有参数处理完成
298.      wait_on_buffer(bh);                                  //如果读取的第一个缓冲块上锁，则等待其
                                                             //解锁
299.      if (bh->b_uptodate)                                  //判定缓冲区数据是否有效
300.                  return bh;                               //有效，则返回该缓冲头指针
301.        brelse(bh);                                        //无效，则释放该缓冲块，返回 NULL
302.        return (NULL);
303.          }
    //缓冲区初始化函数，其中 buffer_end 是缓冲区内存的末端
304.          void buffer_init(long buffer_end)
305.      {
306.        struct buffer_head * h = start_buffer;
307.              void * b;
308.              int i;
309.
310.              if (buffer_end == 1<<20)                     //如果缓冲区末端等于 1MB，则实际可用缓冲区末端
                                                             //应该是 640KB
311.              b = (void *) (640*1024);
312.        else                                               //否则，缓冲区末端就为指定的末端
313.                    b = (void *) buffer_end;
314.              while ( (b-= BLOCK_SIZE) >= ((void *) (h+1)) ) {   //h+1 指向缓冲头的末端，当
    b-BLOCK_SIZE 所指向的大于 h+1 时，表明有足够的内存来建立缓冲头和对应的缓冲块
315.                    h->b_dev = 0;                          //该缓冲块设备号置 0
316.                    h->b_dirt = 0;                         //修改标志置 0
317.                    h->b_count = 0;                        //引用计数置 0
318.                    h->b_lock = 0;                         //锁定标志置 0
319.                    h->b_uptodate = 0;                     //更新标志置 0
320.                    h->b_wait = NULL;                      //等待队列头指针置 NULL
```

```
321.                h->b_next = NULL;          //指向具有相同 hash 值的下一缓冲头指针置 NULL
322.                h->b_prev = NULL;          //指向具有相同 hash 值的上一缓冲头指针置 NULL
323.                h->b_data = (char *) b;    //指向该缓冲头对应的缓冲块
324.                h->b_prev_free = h-1;      //指向空闲链表的前一项
325.                h->b_next_free = h+1;      //指向空闲链表的下一项
326.                h++;                       //h 指向下一新缓冲头位置
327.            NR_BUFFERS++;                  //缓冲块数累计
328.                if (b == (void *) 0x100000) //如果 b 地址等于 1MB，则跳过 384KB，指向 640KB
329.                b = (void *) 0xA0000;
330.        }
331.            h--;                           //h 指向最后一个缓冲头
332.            free_list = start_buffer;      //空闲链表头指针指向第一个缓冲区
333.            free_list->b_prev_free = h;    //将链表头的前一项设置为链表尾
334.            h->b_next_free = free_list;    //将链表尾的下一项设置为链表头
335.            for (i=0;i<NR_HASH;i++)        //初始化 hash 表，置表中所有指针为 NULL
336.            hash_table[i] = NULL;
337.    }
```

源代码中第 7～12 行用来声明与实现高速缓存功能相关的变量和数据结构。

第 13～19 行定义等待指定缓冲块解锁函数，该函数用来检测软盘是否更换。但它需要使更换的软盘缓冲区失效。

第 89～102 行是计算宏定义：该宏定义检查软盘是否被更换，如果被更换就使该软驱的所有缓冲区无效。这是一个相对较慢的程序，所以尽量少的使用它。因此，仅在执行 mount 或 open 时才去调用它。

第 183～228 行定义 getblk 函数，在该函数中隐含了竞争条件，因此其中的代码大部分都很少用到（例如重复语句），所以它要比看起来的效率要高。

第 259～277 行定义的 bread_page()函数用来读取四个缓冲块到内存指定的地址。它是一个独立完整的函数，同时读取四块可以有一些速度的提升，不用等待读取一块，然后再读一块了。

第 278～303 行定义的 breada()函数可以像 bread()函数一样使用，但会同时额外地预读其他一些块。用一个负数作为参数列表的结束。

第 317～318 行为该缓冲块指定设备和块的唯一一个，它没有被使用（b_count=0），没有上锁（b_lock=0），并且没有被修改。此时可占用该缓冲块，并将其变为指定设备和块的缓冲块。

8.4 inode.c 文件

由于此时的内核加入了对虚拟文件系统的支持，将 Minix 文件系统从原文件系统中分离出来，分别形成了充当服务接口的虚拟文件系统（VFS）和物理文件系统 Minix。所以文件 i 结点也分为内存中的 VFS 文件系统 inode 和 Minix 文件系统中的 inode。本程序即是 VFS 文件系统的 inode 程序。

8.4.1 功能描述

虚拟文件系统的 inode 包含的信息主要有文件访问权限、属主、大小、组、生成时间、访问时间、最后修改时间等。它是 Linux 管理文件系统的最基本单位，也是文件系统连接任何子目录、文件的桥梁。inode 结构中的静态信息由文件系统指定的函数填写，信息内容来自物理设备上的文件系统。这些 inode 只能通过 inode 缓存进行访问。每个文件都有相对应的 inode 结点，但是只

有要使用的时候系统才会在内存中为其建立相应的 inode 数据结构，所有建立的 inode 结构将形成一个链表，我们可以通过遍历这个链表去得到需要的文件结点。inode 结构中的 struct inode_operations *i_op 为我们提供了一个 inode 操作列表，通过这个列表提供的函数可以对 VFS inode 结点进行各种操作。每个 inode 结构都有一个 i 结点号 i_ino，在同一个文件系统中每个 i 结点号是唯一的。

本程序主要实现了对 VFS 文件系统的文件结点的处理功能。函数 wait_on_inode()实现了等待指定 i 结点解锁的功能；函数 lock_inode()实现了对 i 结点上锁的功能；函数 unlock_inode()现实了对 i 结点解锁的功能；函数 write_inode()将 i 结点的信息写入缓冲区，在缓冲区刷新时会被写入设备中；函数 read_inode()读取指定 i 结点的信息；函数 bmap()读取文件数据块 block 在设备上对应的逻辑块号；函数 invalidate_inodes()使内存 i 结点中指定设备 i 结点的结点无效；函数 sync_inodes()同步内存 i 结点表中所有 i 结点；函数 iput()将 i 结点放回设备；函数 get_empty_inode()循环扫描内存 i 结点表，获取一个空闲 i 结点。函数 get_pipe_inode()获取管道结点；函数 iget()获取指定设备上指定 i 结点号的 i 结点。

8.4.2 代码注释

inode.c 文件的详细代码注释如下：

```
1.      #include <string.h>                    //字符串头文件。定义了所有字符串操作函数
2.      #include <sys/stat.h>                  //文件状态头文件。主要说明了函数 stat()返回的数据及其
                                               //结构类型，一些属性操作测试宏、函数原型
3.      #include <linux/sched.h>               //调度程序头文件。定义了 task_struct 和任务 0 的数据。一
                                               //些关于描述符参数设置和获取以及 switch_to()的嵌入式汇
                                               //编函数宏
4.
5.      #include <linux/minix_fs.h>            //MINIX 文件系统头文件。定义了与 MINIX 文件系统相关的
                                               //函数和结构体
6.      #include <linux/kernel.h>              //内核头文件。定义了一些内核常用函数的原形
7.      #include <linux/mm.h>                  //内存管理头文件。定义了内存页面的大小和页面释放函数
                                               //原型
8.      #include <asm/system.h>                //系统头文件。定义了设置或修改描述符、中断门等的嵌入
                                               //式汇编宏
9.
10.     struct inode inode_table[NR_INODE]={{0,},};          //声明并初始化内存 i 结点表
11.
12.     extern void minix_read_inode(struct inode * inode);  //声明从设备上读取指定 i 结点信息
                                                             //的函数
13.     extern void minix_write_inode(struct inode * inode); //声明将指定 i 结点的信息写入设备
                                                             //的函数
        //等待指定的 i 结点解锁
14.     static inline void wait_on_inode(struct inode * inode)
15.         {
16.             cli();                          //关中断
17.             while (inode->i_lock)           //判断 i 结点是否上锁
18.             sleep_on(&inode->i_wait);       //如果被上锁，将当前任务加到该 i 结点的等待队列
19.             sti();                          //开中断
20.     }
```

```
        //对指定的 i 结点上锁
21.     static inline void lock_inode(struct inode * inode)
22.       {
23.           cli();                              //关中断
24.           while (inode->i_lock)               //判断 i 结点是否上锁
25.           sleep_on(&inode->i_wait);           //如果已被上锁，将当前任务加到该 i 结点的等待队列
26.           inode->i_lock=1;                    //将锁定标志置位
27.       sti();                                  //开中断
28.     }
        //对指定的 i 结点解锁
29.       static inline void unlock_inode(struct inode * inode)
30.       {
31.           inode->i_lock=0;                    //将锁定标志置位
32.           wake_up(&inode->i_wait);            //唤醒在此 i 结点等待队列上的进程
33.     }
        //将 i 结点信息写入缓冲区。待缓冲区刷新时会写入设备中
34.       static void write_inode(struct inode * inode)
35.       {
36.           lock_inode(inode);                          //锁定指定 i 结点
37.       if (!inode->i_dirt || !inode->i_dev) {          //如果该 i 结点没有被修改过或者 i 结点的设备号为
38.           unlock_inode(inode);                        //0，则对指定 i 结点解锁并退出
39.                 return;
40.           }
41.           minix_write_inode(inode); //调用相应文件系统的 i 结点写入设备的函数，该内核版本只有
42.                   // Minix 文件系统，因此调用 Minix 的 i 结点写入设备函数将 i 结点信息写入
43.       unlock_inode(inode);          //对该 i 结点解锁
44.       }
        //从设备上读取指定 i 结点信息的函数
45.       static void read_inode(struct inode * inode)
46.     {
47.       lock_inode(inode);                          //锁定指定 i 结点
48.       minix_read_inode(inode);                    //调用相应文件系统的从指定设备读取 i 结点信息函
                                                      //数，该内核版本只有 Minix 文件系统，因此调用 Minix
                                                      //的相应函数
49.
50.       unlock_inode(inode);                        //对 i 结点解锁
51.       }
        //取文件数据块 block 在设备上对应的逻辑块号
52.       int bmap(struct inode * inode, int block)
53.     {
54.       return minix_bmap(inode,block); //调用 Minix 文件系统相应函数进行处理
55.       }
        //释放内存 i 结点表中指定设备的 i 结点
56.       void invalidate_inodes(int dev)
57.     {
58.       int i;
59.       struct inode * inode;
60.       inode = 0+inode_table;                      //指向内存 i 结点表指针数组首项
61.       for(i=0 ; i<NR_INODE ; i++,inode++) {//扫描整个内存 i 结点表
62.           wait_on_inode(inode);                   //等待 i 结点解锁
```

```
63.                     if (inode->i_dev == dev) {   //查看是否为指定设备 i 结点
64.                     if (inode->i_count)          //如果 i 结点仍然在使用，则出错
65.                           printk("inode in use on removed disk\n\r");
66.                           inode->i_dev = inode->i_dirt = 0;  //置修改标志和设备号为 0，释放 i 结点
67.                 }
68.           }
69.           }
      //同步内存 i 结点表中所有 i 结点
70.           void sync_inodes(void)
71.           {
72.                 int i;
73.                 struct inode * inode;
74.                 inode = 0+inode_table;              //指向内存 i 结点表指针数组首项
75.           for(i=0 ; i<NR_INODE ; i++,inode++) {//扫描整个内存 i 结点表
76.                     wait_on_inode(inode);         //等待 i 结点解锁
77.           if (inode->i_dirt && !inode->i_pipe)    //若 i 结点被修改并且不是管道结点
78.                     write_inode(inode);           //则进行写盘操作
79.                 }
80.           }
      //将指定 i 结点放回设备
81.           void iput(struct inode * inode)
82.           {
83.           if (!inode)                                   //判断 i 结点有效性，无效则退出
84.                     return;
85.                 wait_on_inode(inode);                   //等待该 i 结点解锁
86.                 if (!inode->i_count)                    //如果 i 结点引用计数为 0，内核仍然要求释放，则
87.                 panic("iput: trying to free free inode"); //出错死机
88.                 if (inode->i_pipe) {                    //如果该 i 结点是管道 i 结点
89.                 wake_up(&inode->i_wait);                //唤醒等待该 i 结点的进程
90.                 wake_up(&inode->i_wait2);               //唤醒等待该管道的进程
91.                       if (--inode->i_count)             //引用计数减 1 后如果为 0，即如果仍有其他进程在
92.                       return;                           //使用，则返回
93.                 free_page(inode->i_size);               //释放管道占用的页面
94.                       inode->i_count=0;                 //对引用计数复位
95.                 inode->i_dirt=0;                        //对修改标志复位
96.                 inode->i_pipe=0;                        //对管道标志复位
97.                       return;
98.                 }
99.                 if (!inode->i_dev) {                    //如果该 i 结点的设备号为 0
100.                inode->i_count--;                       //则应用计数减 1，返回
101.                return;
102.                }
103.                if (S_ISBLK(inode->i_mode)) {           //如果该 i 结点是块设备文件的 i 结点
104.                      sync_dev(inode->i_rdev);          //则同步该设备相应的 i 结点
105.                wait_on_inode(inode);                   //等待该 i 结点解锁
106.          }
107.           repeat:
108.                if (inode->i_count>1) {                 //如果该结点除本进程外，还有其他进程引用
109.                      inode->i_count--;                 //则引用计数减 1，返回
110.                return;
```

```
111.        }
112.              if (!inode->i_nlink) {                    //如果该 i 结点的连接数为 0，说明该文件被删除
113.              minix_truncate(inode);                    //则释放该 i 结点所有逻辑块
114.              minix_free_inode(inode);                  //复位该 i 结点在 i 结点位图中的比特位，清空内容
115.              return;
116.              }
117.              if (inode->i_dirt) {                      //如果该 i 结点已被修改
118.              write_inode(inode);                       //进程经过睡眠，需要重新判断
119.              wait_on_inode(inode);                     //等待该 i 结点解锁
120.              goto repeat;                              //重新判断
121.              }
122.              inode->i_count--;                         //引用计数减 1，返回
123.              return;
124.        }
          //获取一个空闲 i 结点
125.        struct inode * get_empty_inode(void)
126.            {
127.              struct inode * inode;
128.              static struct inode * last_inode = inode_table;    //last_inode 指向内存 i 结点表指针数组
129.          int i;                                                 //首项
130.          do {
131.                    inode = NULL;
132.                    for (i = NR_INODE; i ; i--) {                //循环扫描内存 i 结点表
133.                    if (++last_inode >= inode_table + NR_INODE)  //如果 last_node 指向内存 i
134.                          last_inode = inode_table;              //结点表最后一项之后，则重新指向首项
135.                          if (!last_inode->i_count) {            //如果该 i 结点引用计数为 0
136.                          inode = last_inode;                    //将 inode 指向该 i 结点
137.                          if (!inode->i_dirt && !inode->i_lock)  //如果该 i 结点未被修改并且未上锁
138.                                  break;                         //则说明找到空闲 i 结点，退出循环
139.                    }
140.              }
141.                    if (!inode) {                                //没有找到空闲 i 结点
142.                          for (i=0 ; i<NR_INODE ; i++)           //则打印内存 i 结点表并停机
143.                          printk("(%04x: %d (%o)) ",inode_table[i].i_dev,
144.                                inodc_table[i].i_ino,inode_table[i].i_mode);
145.                          panic("No free inodes in mem");
146.                    }
147.                    wait_on_inode(inode);                        //等待该 i 结点解锁
148.                    while (inode->i_dirt) {                      //如果该 i 结点被修改
149.                    write_inode(inode);                          //则将 i 结点写入设备
150.                          wait_on_inode(inode);                  //等待该 i 结点解锁
151.              }
152.        } while (inode->i_count);                                //如果在睡眠时，该 i 结点又被占用，则重
                                                                     //新寻找
153.              memset(inode,0,sizeof(*inode));                    //将该 i 结点清零
154.              inode->i_count = 1;                                //引用计数置为 1，返回该 i 结点指针，退出
155.              return inode;
156.        }
          //获取管道结点函数
157.        struct inode * get_pipe_inode(void)
```

```
158.        {
159.              struct inode * inode;
160.        if (!(inode = get_empty_inode()))                    //如果没有空闲 i 结点
161.                    return NULL;                             //返回 NULL
162.        if (!(inode->i_size=get_free_page())) {              //获取一页空闲内存给管道，如果没有空闲
                                                                 //内存，则复位 i 结点引用计数，返回 NULL
163.              inode->i_count = 0;
164.                    return NULL;
165.              }
166.              inode->i_count = 2;                            /*读者/写者之和*/
167.        PIPE_HEAD(*inode) = PIPE_TAIL(*inode) = 0;           //初始化管道头尾指针
168.        inode->i_pipe = 1;                                   //对管道标志置位
169.              return inode;                                  //返回管道 i 结点
170.      }
      //从指定设备上读取 nr 结点号的 i 结点
171.    struct inode * iget(int dev,int nr)
172.      {
173.              struct inode * inode, * empty;
174.              if (!dev)                                      //如果设备号无效，则出错停机
175.                    panic("iget with dev==0");
176.              empty = get_empty_inode();                     //获取一个空闲 i 结点
177.        inode = inode_table;                                 //inode 指向内存 i 结点表首项
178.        while (inode < NR_INODE+inode_table) {               //遍历内存 i 结点表
179.                    if (inode->i_dev != dev || inode->i_ino != nr) {      //如果找不到指定设备和结点
                                                                 //号的 i 结点则继续
180.                    inode++;
181.                    continue;
182.                    }
183.                    wait_on_inode(inode);                    //等待 i 结点解锁
184.                    if (inode->i_dev != dev || inode->i_ino != nr) { //由于经过睡眠，i 结点可能发生变化
185.                    inode = inode_table;                     //再次检查，如果发生变化，重新扫描内存 i 结
                                                                 //点表
186.                    continue;
187.                    }
188.                    inode->i_count++;                        //增加引用计数
189.                    if (inode->i_mount) {                    //判断该 i 结点是否为其他文件系统的安装点
190.                    int i;
191.              for (i = 0 ; i<NR_SUPER ; i++)                 //如果是，则扫描超级块表，寻找安装在该 i
192.                          if (super_block[i].s_covered==inode) //结点的超级块
193.                          break;
194.                    if (i >= NR_SUPER) {                     //如果没有找到，则释放获取的空闲结点
195.                          printk("Mounted inode hasn't got sb\n"); //打印错误信息
196.                        if (empty)                           //释放空闲 i 结点
197.                          iput(empty);
198.                          return inode;                      //返回该 i 结点指针
199.                        }
200.                        iput(inode);                         //将该 i 结点放回设备
201.              if (!(inode = super_block[i].s_mounted))//将 i 结点指向被安装文件系统的根结点
202.              printk("iget: mounted dev has no rootinode\n");//如果没有根结点，打印错误信息
203.                        else {
```

```
204.                        inode->i_count++;      //将该 i 结点引用计数加 1
205.                    wait_on_inode(inode);      //等待该 i 结点解锁
206.                }
207.                }
208.                if (empty)                     //释放申请的空闲 i 结点
209.                    iput(empty);
210.                return inode;                  //返回该 i 结点指针
211.        }
212.        if (!empty)                            //如果没有找到指定的 i 结点，又没有空闲 i 结点，
                                                   //则返回 NULL
213.                return (NULL);
214.        inode = empty;                         //如果有空闲 i 结点，则利用该空闲 i 结点创建指
                                                   //定 i 结点
215.        if (!(inode->i_sb = get_super(dev))) {  //将该 i 结点超级块指针指向指定设备超级块
216.                printk("iget: gouldn't get super-block\n\t"); //如果超级块不存在，打印错误信息
217.            iput(inode);                       //将该 i 结点放回设备，返回 NULL
218.            return NULL;
219.            }
220.            inode->i_dev = dev;                //该 i 结点的设备号设为指定设备号
221.            inode->i_ino = nr;                 //该 i 结点的结点号设为指定结点号
222.            read_inode(inode);                 //从设备上读取该 i 结点信息，返回该 i 结点指针
223.            return inode;
224.         }
```


8.5 namei.c 文件

该文件主要包含根据目录名或文件名寻找到对应 i 结点的函数，以及一些关于目录的建立和删除、目录项的建立和删除等操作函数和系统调用。

8.5.1 功能描述

本程序实现了与目录名或文件名有关的操作。例如根据指定的路径寻找指定的 i 结点，创建和删除指定目录名的目录及目录项的创建和删除，创建符号连接或硬连接等。

- 函数 permission()检查指定文件的访问权限。
- lookup()寻找指定目录和文件名的 i 结点。
- 函数 follow_link()获取指定目录下指定符号连接文件的目标文件 i 结点。
- 函数 dir_namei()获取指定路径名最顶层目录 i 结点、顶层文件名称及长度。
- 函数_namei()获取指定路径名的 i 结点。
- 函数 lnamei()获取指定路径名的符号连接文件 i 结点。
- 函数 namei()取指定路径名的非符号连接文件 i 结点。
- 函数 open_namei()是给 open()使用的 namei 函数。
- 函数 sys_mknod()创建一个文件结点。
- 函数 sys_mkdir()创建一个目录。
- 函数 sys_rmdir()删除一个目录。
- 函数 sys_unlink()用于删除一个目录项。

- 函数 sys_symlink()为一个已存在的文件创建一个符号链接。
- 函数 sys_link()为一个已存在的文件创建一个硬链接。
- 函数 sys_rename()用来为一个文件重新命名。

8.5.2 代码注释

namei.c 文件的详细代码注释如下：

```
1.      #include <linux/sched.h>          //调度程序头文件。定义了 task_struct 和任务 0 的数据。一些关
                                          //于描述符参数设置和获取以及 switch_to()的嵌入式汇编函数宏
2.      #include <linux/minix_fs.h>       //MINIX 文件系统头文件。定义了与 MINIX 文件系统
                                          //相关的函数和结构体
3.      #include <linux/kernel.h>         //内核头文件。定义了一些内核常用函数的原形
4.      #include <asm/segment.h>          //段操作头文件。定义了一些访问段寄存器或与段寄存器有关
                                          //的内存操作函数
5.      #include <string.h>               //字符串头文件。定义了所有字符串操作函数
6.      #include <fcntl.h>                //文件控制头文件。用于文件及其描述符的操作控制常数符号的
                                          //定义
7.      #include <errno.h>                //错误号头文件。包含系统中各种出错号
8.      #include <const.h>                //常数符号头文件。定义了 i 结点中文件属性和类型 i_mode 字段
                                          //所用到的一些标志位常量符号
9.      #include <sys/stat.h>             //文件状态头文件。主要说明了函数 stat()返回的数据及其结构类型
10.                                       //一些属性操作测试宏、函数原型
11.          struct inode * _namei(const char * filename, struct inode * base,
12.          int follow_links);
       //访问模式宏。根据文件访问标志 x 索引对应数值
13.    #define ACC_MODE(x) ("\004\002\006\377"[(x)&O_ACCMODE])
       //检查指定文件的访问权限
14.    int permission(struct inode * inode,int mask)
15.    {
16.            int mode = inode->i_mode;
       //特殊情况，即便是超级用户也不能读/写一个已经被删除的文件
       //如果该 i 结点有设备号同时连接数为 0，表明该 i 结点对应文件已被删除，则返回 0
17.        if (inode->i_dev && !inode->i_nlink)
18.            return 0;
       //否则，如果进程有效用户 id 与该 i 结点的用户 id 相等，则取文件宿主的用户访问权限
19.            else if (current->euid == inode->i_uid)
20.            mode >>= 6;
       //否则，如果进程有效组 id 与该 i 结点的组 id 相等，则取组用户的访问权限
21.        else if (in_group_p(inode->i_gid))
22.                mode >>= 3;
       //如果所取的用户访问权限与屏蔽码相等或者用户为超级用户，则返回 1，否则返回 0
23.        if (((mode & mask & 0007) == mask) || suser())
24.            return 1;
25.            return 0;
26.         }
       //寻找指定目录和文件名的 i 结点
27.         int lookup(struct inode * dir,const char * name, int len,
28.        struct inode ** result)
29.    {
30.            struct super_block * sb;
```

```
31.
32.             *result = NULL;
    //首先对文件名是“..”的情况进行特殊处理
33.             if (len==2 && get_fs_byte(name)=='.' && get_fs_byte(name+1)=='.') {
34.             if (dir == current->root)
35.                     len = 1;
36.                     else if ((sb = dir->i_sb) && (dir == sb->s_mounted)) {
37.                             sb = dir->i_sb;          //获取文件系统超级块
38.                     iput(dir);                       //释放指定目录 i 结点
39.                     if (dir = sb->s_covered)         //将指向指定目录 i 结点的指针重新指向文件系统被
40.                                                      //安装到的 i 结点
41.                             dir->i_count++;          //将该 i 结点的引用计数加 1
42.             }
43.             }
44.             if (!dir)                                //如果指定目录 i 结点不存在，则返回错误码
45.                     return -ENOENT;
46.         if (!permission(dir,MAY_EXEC)) {             //如果指定目录的执行权限不可执行
47.             iput(dir);                               //则释放该 i 结点，返回错误码
48.                     return -EACCES;
49.             }
50.             if (!len) {                              //如果文件名长度为 0
51.             *result = dir;                           //则在*result 中返回指定目录 i 结点指针
52.             return 0;                                //返回 0
53.             }
54.             if (!dir->i_op || !dir->i_op->lookup) {  //如果指定目录 i 结点的 i_op 字段不存在或
                                                         //者无查找函数，则释放该 i 结点，返回错误码
55.                     iput(dir);
56.                     return -ENOENT;
57.                     }
58.                     return dir->i_op->lookup(dir,name,len,result); //返回调用查找函数的结果
59.             }
     //获取指定目录下指定符号连接文件的目标文件 i 结点
60.     struct inode * follow_link(struct inode * dir, struct inode * inode)
61.     {
62.             if (!dir || !inode) {                            //如果指定目录或文件的 i 结点无效
63.             iput(dir);                                       //释放指定目录 i 结点
64.             iput(inode);                                     //释放指定文件 i 结点
65.             return NULL;                                     //返回 NULL
66.             }
67.             if (!inode->i_op || !inode->i_op->follow_link) { //如果指定文件 i 结点 i_op 字段不存在
68.             iput(dir);                                       //或者无符号连接查找函数，则释放指定目
                                                                 //录 i 结点
69.             return inode;                                    //返回指定文件 i 结点
70.             }
71.             return inode->i_op->follow_link(dir,inode);      //返回调用查找符号连接目标文件函数的
                                                                 //结果
72.     }
     //获取指定路径名最顶层目录 i 结点、顶层文件名称及长度
73.     static struct inode * dir_namei(const char * pathname,
74.             int * namelen, const char ** name, struct inode * base)
```

```
75.    {
76.            char c;
77.            const char * thisname;
78.            int len,error;
79.            struct inode * inode;
80.            if (!base) {                                    //如果指定的开始 i 结点不存在
81.            base = current->pwd;                            //则从当前工作目录 i 结点开始
82.            base->i_count++;                                //当前工作目录 i 结点引用计数加 1
83.            }
84.            if ((c=get_fs_byte(pathname))=='/') {           //如果用户指定的路径名的第一个字符是
                                                               // "/"，则说明指定的路径名为绝对路径
                                                               //应从根 i 结点开始
85.
86.            iput(base);                                     //释放已有的开始 i 结点
87.            base = current->root;                           //从根 i 结点开始
88.            pathname++;                                     //指向路径名第二个字符
89.            base->i_count++;                                //开始 i 结点引用计数加 1
90.            }
91.            while (1) {                                     //针对路径名中的各个目录名部分和文件
                                                               //名进行循环处理。
92.            thisname = pathname; //把变量 thisname 指向当前正在处理的目录名或文件名部分
93.            for(len=0;(c=get_fs_byte(pathname++))&&(c!='/');len++)
94.
95.                    if (!c)
96.                            break;
97.            base->i_count++;                                //对开始 i 结点引用计数加 1
98.                    error = lookup(base,thisname,len,&inode);   //在当前处理的目录中查找指定名称的目
99.                                                           //录 i 结点
100.                   if (error) {                            //如果未找到
101.                           iput(base);                     //释放开始 i 结点，返回 NULL
102.                                   return NULL;
103.                           }
104.                           if (!(base = follow_link(base,inode)))   //查找符号连接的目标文件
105.                           return NULL;                    //失败，则返回 NULL
106.           }
107.                   *name = thisname;                       //返回顶层文件名
108.                   *namelen = len;                         //返回顶层文件名长度
109.                   return base;                            //返回顶层目录 i 结点
110.   }
       //获取指定路径名的 i 结点
111.           struct inode * _namei(const char * pathname, struct inode * base,
112.                   int follow_links)
113.   {
114.           const char * basename;
115.                   int namelen,error;
116.                   struct inode * inode;
       //首先得到指定路径名最顶层目录 i 结点，顶层文件名和长度
117.           if (!(base = dir_namei(pathname,&namelen,&basename,base)))
118.                   return NULL;                                    //操作失败返回 NULL
       //因为下面要用到顶层目录 i 结点，所以要增加其引用计数
```

```
119.            base->i_count++;                                    /*lookup 要用到 base*/
120.        error = lookup(base,basename,namelen,&inode);           //在顶层目录下查找顶层文件的 i 结点
121.        if (error) {                                            //如果失败
122.                iput(base);                                     //释放顶层目录 i 结点，返回 NULL
123.            return NULL;
124.        }
125.            if (follow_links)                                   //如果符号连接标志置位
126.                inode = follow_link(base,inode);                //则取符号连接的目标文件 i 结点
127.        else                                                    //符号连接标志未置位
128.            iput(base);                                         //则释放顶层目录 i 结点
129.            if (inode) {                                        //如果文件 i 结点有效
130.            inode->i_atime=CURRENT_TIME;                        //修改该 i 结点时间
131.            inode->i_dirt=1;                                    //置位修改标志
132.            }
133.            return inode;                                       //返回文件 i 结点指针
134.    }
        //获取指定路径名的符号连接文件 i 结点
135.    struct inode * lnamei(const char * pathname)
136.        {
137.            return _namei(pathname, NULL, 0);       //调用_namei()，符号连接标志置 0。返回 i 结点指针
138.    }
        //取指定路径名的非符号连接文件 i 结点
139.    struct inode * namei(const char * pathname)
140.        {
141.            return _namei(pathname,NULL,1);         //调用_namei()，符号连接标志置 1。返回 i 结点指针
142.    }
        //文件打开 namei 函数
143.    int open_namei(const char * pathname, int flag, int mode,
144.            struct inode ** res_inode)
145.    {
146.        const char * basename;
147.            int namelen,error;
148.            struct inode * dir, *inode;
149.       if ((flag & O_TRUNC) && !(flag & O_ACCMODE)) //如果文件截断标志置位，同时文件访
150.                                                    //问许可模式标志是只读
151.            flag |= O_WRONLY;                       //则改写为只写标志
152.            mode &= 0777 & ~current->umask;         //使用进程的文件访问许可屏蔽码屏蔽指定
153.                                                    //模式相应位
154.            mode |= I_REGULAR;                      //置位普通文件标志
        //根据路径名获取顶层目录 i 结点和顶层文件名及长度
155.            if (!(dir = dir_namei(pathname,&namelen,&basename,NULL)))
156.            return -ENOENT;                         //失败则返回错误码
        //如果得到的顶层文件名长度为 0，例如 /user/ 等一些特例
157.        if (!namelen) {                             /*特例： /usr/ 等*/
158.            if (!(flag & (O_ACCMODE|O_CREAT|O_TRUNC))) {   //如果打开操作不是创建或截
159.                    *res_inode=dir;                 //断，则表示打开一个目录名，返回该目录 i 结
                                                        //点，返回 0
160.                return 0;
161.            }
162.                iput(dir);                          //释放该目录 i 结点，返回错误码
```

```
163.                return -EISDIR;
164.            }
165.        dir->i_count++;                                  /*lookup 将使用该目录 i 结点*/
166.        error = lookup(dir,basename,namelen,&inode);   //获取底层文件 i 结点
167.            if (error) {                                 //如果失败，则可能是创建文件操作
168.            if (!(flag & O_CREAT)) {                     //如果不是创建操作
169.                iput(dir);                               //释放该目录的 i 结点，返回错误码
170.                    return error;
171.                }
172.                if (!permission(dir,MAY_WRITE)) {//如果对于该目录，用户没有写权利
173.                iput(dir);                               //则释放该目录 i 结点
174.                return -EACCES;                          //返回错误号
175.                }
176.                if (!dir->i_op || !dir->i_op->create) {//如果该目录 i 结点的 i_op 字段无效或不存在创
177.                iput(dir);                               //建函数，则释放该目录 i 结点，返回错误码
178.                return -EACCES;
179.                }
        //调用创建函数，并返回创建的文件 i 结点指针
180.                return dir->i_op->create(dir,basename,namelen,mode,res_inode);
181.        }
182.        if (flag & O_EXCL) {                             //如果独占使用标志置位
183.                iput(dir);                               //则释放该目录 i 结点
184.                iput(inode);                             //则释放该文件 i 结点，返回错误码
185.                return -EEXIST;
186.        }
187.            if (!(inode = follow_link(dir,inode)))       //获取符号连接的目标文件 i 结点
188.                    return -ELOOP;                       //失败，返回错误码
      //若顶层 i 结点是一个目录结点并且访问模式为只读或只写，或者没有访问许可权限
189.        if ((S_ISDIR(inode->i_mode) && (flag & O_ACCMODE)) ||
190.            !permission(inode,ACC_MODE(flag))) {
191.                iput(inode);                             //则释放该 i 结点，返回错误码
192.            return -EPERM;
193.        }
194.            inode->i_atime = CURRENT_TIME;               //修改该 i 结点的访问时间
195.            if (flag & O_TRUNC)                          //如果截断标志置位
196.            minix_truncate(inode);                       //则将该 i 结点的文件截断
197.            *res_inode = inode;                          //返回顶层目录 i 结点指针，返回 0
198.            return 0;
199.        }
      //创建一个文件结点
200.        int sys_mknod(const char * filename, int mode, int dev)
201.    {
202.        const char * basename;
203.            int namelen;
204.            struct inode * dir;
205.        if (!suser())                                    //如果不是超级用户，则返回错误码
206.            return -EPERM;
207.        if (!(dir = dir_namei(filename,&namelen,&basename, NULL))) //如果找不到指定路径名的
208.                                                         //顶层目录 i 结点，则返回错误码
209.            return -ENOENT;
```

```
210.        if (!namelen) {                                  //如果顶层文件名长度为 0，表明指定的路径名
                                                             //没有指定文件名
211.                iput(dir);                               //释放该目录 i 结点，返回错误码
212.            return -ENOENT;
213.        }
214.            if (!permission(dir,MAY_WRITE)) {            //如果对于该目录，用户没有写权利
215.                iput(dir);                               //则释放该目录 i 结点，则返回错误码
216.            return -EPERM;
217.        }
218.            if (!dir->i_op || !dir->i_op->mknod) {       //如果该目录 i 结点的 i_op 字段无效或不存在文件
219.                                                         //创建函数
220.            iput(dir);                                   //释放该目录 i 结点，返回错误码
221.            return -EPERM;
222.            }
223.            return dir->i_op->mknod(dir,basename,namelen,mode,dev);   //调用文件创建函数
224.    }
        //创建目录
225.        int sys_mkdir(const char * pathname, int mode)
226.        {
227.            const char * basename;
228.            int namelen;
229.            struct inode * dir;
230.            if (!(dir = dir_namei(pathname,&namelen,&basename, NULL))) //如果找不到指定路径名
231.                                                         //的顶层目录 i 结点，则返回错误码
232.            return -ENOENT;
233.        if (!namelen) {                                  //如果顶层文件名长度为 0，表明指定的路径名
                                                             //没有指定文件名
234.                iput(dir);                               //释放该目录 i 结点，返回错误码
235.            return -ENOENT;
236.        }
237.            if (!permission(dir,MAY_WRITE)) {            //如果对于该目录，用户没有写权利
238.                iput(dir);                               //则释放该目录 i 结点，则返回错误码
239.            return -EPERM;
240.        }
241.            if (!dir->i_op || !dir->i_op->mkdir) {       //如果该目录 i 结点的 i_op 字段无效或不存在
                                                             //目录
242.                                                         //创建函数
243.            iput(dir);                                   //释放该目录 i 结点，返回错误码
244.            return -EPERM;
245.            }
246.            return dir->i_op->mkdir(dir,basename,namelen,mode);   //调用目录创建函数
247.    }
        //删除目录
248.    int sys_rmdir(const char * name)
249.        {
250.            const char * basename;
251.            int namelen;
252.            struct inode * dir;
253.        if (!(dir = dir_namei(name,&namelen,&basename, NULL)))//如果找不到指定路径名的
254.                                                         //顶层目录 i 结点，则返回错误码
```

```
255.            return -ENOENT;
256.        if (!namelen) {                          //如果顶层文件名长度为 0，表明指定的路径名
                                                     //没有指定文件名
257.                iput(dir);                       //释放该目录 i 结点，返回错误码
258.            return -ENOENT;
259.        }
260.            if (!permission(dir,MAY_WRITE)) {    //如果对于该目录，用户没有写权利
261.                iput(dir);                       //则释放该目录 i 结点，返回错误码
262.            return -EPERM;
263.        }
264.            if (!dir->i_op || !dir->i_op->rmdir) {   //如果该目录 i 结点的 i_op 字段无效或不存在目
265.                                                     //录删除函数
266.            iput(dir);                           //释放该目录 i 结点，返回错误码
267.            return -EPERM;
268.            }
269.            return dir->i_op->rmdir(dir,basename,namelen);          //调用目录删除函数
270.    }
     //删除文件名对应的目录项
271.    int sys_unlink(const char * name)
272.        {
273.            const char * basename;
274.            int namelen;
275.            struct inode * dir;
276.            if (!(dir = dir_namei(name,&namelen,&basename, NULL))) //如果找不到指定路径名的顶
277.                                                     //层目录 i 结点，则返回错误码
278.            return -ENOENT;
279.        if (!namelen) {                          //如果顶层文件名长度为 0，表明指定的路径名
                                                     //没有指定文件名
280.                iput(dir);                       //释放该目录 i 结点，返回错误码
281.            return -ENOENT;
282.        }
283.            if (!permission(dir,MAY_WRITE)) {    //如果对于该目录，用户没有写权利
284.                iput(dir);                       //则释放该目录 i 结点，返回错误码
285.            return -EPERM;
286.        }
287.            if (!dir->i_op || !dir->i_op->unlink) {   //如果该目录 i 结点的 i_op 字段无效或不存在文
                                                     //件对应目录项删除函数
288.
289.            iput(dir);                           //释放该目录 i 结点，返回错误码
290.            return -EPERM;
291.            }
292.            return dir->i_op->unlink(dir,basename,namelen);    //调用文件对应目录项删除函数
293.    }
     //为一个已存在的文件创建一个符号链接
294.    int sys_symlink(const char * oldname, const char * newname)
295.        {
296.            struct inode * dir;
297.            const char * basename;
298.            int namelen;
299.
```

```
300.            dir = dir_namei(newname,&namelen,&basename, NULL);    //获取指定新路径名的顶层
301.                                                                  //目录 i 结点
302.            if (!dir)                                             //如果该目录 i 结点不存在，返回错误码
303.            return -EACCES;
304.            if (!namelen) {                                       //如果顶层文件名长度为 0，表明指定的路径名
                                                                      //没有指定文件名
305.                iput(dir);                                        //释放该目录 i 结点，返回错误码
306.            return -EPERM;
307.        }
308.            if (!permission(dir,MAY_WRITE)) {    //如果对于该目录，用户没有写权利
309.                iput(dir);                       //则释放该目录 i 结点，则返回错误码
310.            return -EPERM;
311.        }
312.            if (!dir->i_op || !dir->i_op->symlink) {  //如果该目录 i 结点的 i_op 字段无效或不存在符号
313.                                                      //链接创建函数
314.            iput(dir);                                //释放该目录 i 结点，返回错误码
315.            return -EPERM;
316.            }
317.            return dir->i_op->symlink(dir,basename,namelen,oldname); //调用符号链接创建函数
318.    }
        //为一个已存在的文件创建一个硬链接
319.    int sys_link(const char * oldname, const char * newname)
320.        {
321.            struct inode * oldinode, * dir;
322.            const char * basename;
323.            int namelen;
324.
325.            oldinode = namei(oldname);            //获取原文件路径名对应的文件 i 结点
326.            if (!oldinode)                        //如果该 i 结点不存在，返回错误码
327.            return -ENOENT;
328.        dir = dir_namei(newname,&namelen,&basename, NULL);      //获取指定新路径名的顶层
329.                                                                //目录 i 结点
330.            if (!dir) {                                         //如果该目录 i 结点不存在
331.            iput(oldinode);                                     //释放原文件 i 结点，返回错误码
332.                return -EACCES;
333.            }
334.            if (!namelen) {                                     //如果顶层文件名长度为 0，表明指定的路径名
                                                                    //没有指定文件名
335.            iput(oldinode);                                     /释放原文件 i 结点
336.            iput(dir);                                          //释放该目录 i 结点，返回错误码
337.                return -EPERM;
338.            }
339.            if (dir->i_dev != oldinode->i_dev) {    //如果新路径名所属设备与原路径名所属设备不同
340.            iput(dir);                              //释放该目录 i 结点
341.            iput(oldinode);                         /释放原文件 i 结点，返回错误码
342.                return -EXDEV;
343.            }
344.            if (!permission(dir,MAY_WRITE)) {       //如果对于该目录，用户没有写权利
345.            iput(dir);                              //则释放该目录 i 结点
346.            iput(oldinode);                         //释放原文件 i 结点，返回错误码
```

```
347.                return -EACCES;
348.            }
349.            if (!dir->i_op || !dir->i_op->link) {        //如果该目录i结点的i_op字段无效或不存在符号
350.                                                         //链接创建函数
351.            iput(dir);                                   //则释放该目录 i 结点
352.                iput(oldinode);                          //释放原文件 i 结点，返回错误码
353.            return -EPERM;
354.        }
355.            return dir->i_op->link(oldinode, dir, basename, namelen);    //调用硬链接创建函数
356.         }
      //文件重命名函数
357.    int sys_rename(const char * oldname, const char * newname)
358.    {
359.            struct inode * old_dir, * new_dir;
360.            const char * old_base, * new_base;
361.            int old_len, new_len;
362.
363.        old_dir = dir_namei(oldname,&old_len,&old_base, NULL);    //获取指定原路径名的顶层
364.                                                                  //目录 i 结点
365.            if (!old_dir)                                        //如果该目录 i 结点不存在，返回错误码
366.            return -ENOENT;
367.            if (!permission(old_dir,MAY_WRITE)) {        //如果对于该目录，用户没有写权利
368.                iput(old_dir);                           //则释放该目录 i 结点，则返回错误码
369.            return -EACCES;
370.        }
371.            if (!old_len || (get_fs_byte(old_base) == '.' &&
372.                (old_len == 1 || (get_fs_byte(old_base+1) == '.' &&
373.                 old_len == 2)))) {
374.                iput(old_dir);
375.                return -EPERM;
376.            }
377.            new_dir = dir_namei(newname,&new_len,&new_base, NULL);//获取指定新路径名的顶
378.                                                         //层目录 i 结点
379.        if (!new_dir) {                                  //如果该目录 i 结点不存在
380.                iput(old_dir);                           //则释放原路径名的顶层目录 i 结点
381.            return -ENOENT;                              //返回错误码
382.        }
383.            if (!permission(new_dir,MAY_WRITE)) {        //如果对于该目录，用户没有写权利
384.                iput(old_dir);                           //则释放原路径名的顶层目录 i 结点
385.                iput(new_dir);                           //释放该目录 i 结点，则返回错误码
386.                return -EACCES;
387.            }
388.            if (!new_len || (get_fs_byte(new_base) == '.' &&
389.                (new_len == 1 || (get_fs_byte(new_base+1) == '.' &&
390.                 new_len == 2)))) {
391.                iput(old_dir);
392.                iput(new_dir);
393.                return -EPERM;
394.            }
395.            if (new_dir->i_dev != old_dir->i_dev) {      //如果新路径名所属设备与原路径名所属
```

```
                                                        //设备不同
396.            iput(old_dir);                          //则释放原路径名的顶层目录 i 结点
397.            iput(new_dir);                          //释放该目录 i 结点，则返回错误码
398.                return -EXDEV;
399.            }
400.            if (!old_dir->i_op || !old_dir->i_op->rename) {   //如果该目录 i 结点的 i_op 字段无效或不存
401.                                                    //在文件重命名函数
402.            iput(old_dir);                          //则释放原路径名的顶层目录 i 结点
403.                iput(new_dir);                      //释放该目录 i 结点，则返回错误码
404.            return -EPERM;
405.        }
406.            return old_dir->i_op->rename(old_dir, old_base, old_len,
407.                new_dir, new_base, new_len);        //调用文件重命名函数
408.    }
```

第 14～26 行：permission()函数用于检测一个文件的读/写/执行权限。

第 27～59 行：lookup()利用文件系统相关程序（现在该程序是 minix_lookup）寻找一个路径名的一部分。它同样可以查找父目录（伪根目录，安装点）。

第 73～110 行：dir_namei()函数返回指定目录名的 i 结点指针，以及最顶层目录名。

第 139～142 行：许多简单的命令利用 namei()函数取得指定路径名的 i 结点。open，link 等使用它们自己的函数，但对于 chmod 等命令，这个函数就足够了。

第 143～199 行为：open_namei() 给 open()使用的 namei 函数。事实上，这几乎是个完整的打开程序。

8.6 super.c 文件

该文件描述了对文件系统中超级块操作的函数，这些函数属于稳健系统底层函数，供上层的文件名和目录操作函数使用。

8.6.1 功能描述

每一种文件系统都有自己的超级块，在超级块中存放着该文件系统的相关信息，所以在内核中有一个超级块表，它记录着内核所使用的各种文件系统的超级块信息。超级块表中除了会把磁盘上超级块内容加载进来以外，还会记录另外一些信息，例如 i 结点位图在高速缓冲块的指针数组和逻辑块位图在高速缓冲块的指针数组。

本程序实现了对文件系统中超级块的操作，其中的函数是供上层文件名和目录操作使用的文件系统底层函数。函数 lock_super()对指定的超级块上锁；函数 free_super()对指定的超级块解锁；函数 wait_on_super()等待超级块解锁。函数 get_super()在超级块表中查找指定设备的超级块。

函数 put_super()释放指定设备的超级块，把该超级块对应的文件系统的 i 结点位图和逻辑块位图所占用的缓冲块都释放掉；函数 read_super()把指定设备的文件系统的超级块读入到缓冲区中，并登记到超级块数组中，同时也把文件系统的 i 结点位图和逻辑块位图读入内存超级块结构的相应数组中；函数 sys_umount()卸载指定设备的文件系统；函数 sys_mount()往一个目录名上加载一个文件系统；函数 mount_root()安装根文件系统。

8.6.2 代码注释

super.c 文件的详细代码注释如下：

```
1.      #include <linux/config.h>     //内核配置头文件。定义使用的键盘语言类型和硬盘类型可选项
2.      #include <linux/sched.h>      //调度程序头文件。定义了 task_struct 和任务 0 的数据。一些关
                                      //于描述符参数设置和获取以及 switch_to()的嵌入式汇编函数宏
3.      #include <linux/minix_fs.h>   //Minix 文件系统头文件。定义了与 Minix 文件系统相关的函
                                      //数和结构体
4.      #include <linux/kernel.h>     //内核头文件。定义了一些内核常用函数的原形
5.      #include <asm/system.h>       //系统头文件。定义了设置或修改描述符、中断门等的嵌入式汇
                                      //编宏
6.      #include <errno.h>            //错误号头文件。包含系统中各种出错号
7.      #include <sys/stat.h>         //文件状态头文件。主要说明了函数 stat()返回的数据及其结构类型
                                      //一些属性操作测试宏、函数原型
8.      int sync_dev(int dev);        //声明对指定设备进行数据同步操作的函数
9.      void wait_for_keypress(void);//声明等待按键的函数
      //set_bit 使用 setb 指令，因为 gas 不能识别 setc 指令
      //嵌入式汇编宏，返回指定比特位的值
10.     #define set_bit(bitnr,addr) ({
11.     register int __res __asm__("ax");
12.     __asm__("bt %2,%3;setb %%al":"=a" (__res):"a" (0),"r" (bitnr),"m" (*(addr)));
13.     __res; })
14.
15.        struct super_block super_block[NR_SUPER];
16.
17.        /*这已在 init/main.c 中初始化*/
18.     int ROOT_DEV = 0;
      //对指定的超级块上锁
19.     static void lock_super(struct super_block * sb)
20.        {
21.            cli();                                    //关中断
22.            while (sb->s_lock)                        //如果指定超级块已经上锁
23.            sleep_on(&(sb->s_wait));                  //则睡眠等待其解锁
24.            sb->s_lock = 1;                           //对指定超级块上锁
25.       sti();                                         //开中断
26.     }
      //对指定超级块解锁
27.        static void free_super(struct super_block * sb)
28.        {
29.            cli();                                    //关中断
30.            sb->s_lock = 0;                           //对该超级块解锁
31.            wake_up(&(sb->s_wait));                   //唤醒等待队列上的进程
32.            sti();                                    //开中断
33.        }
      //等待指定超级块解锁
34.     static void wait_on_super(struct super_block * sb)
35.     {
36.            cli();                                    //关中断
37.            while (sb->s_lock)                        //如果指定超级块上锁
38.            sleep_on(&(sb->s_wait));                  //则睡眠等待其解锁
39.            sti();                                    //开中断
40.     }
      //取指定设备的超级块
```

```
41.         struct super_block * get_super(int dev)
42.         {
43.             struct super_block * s;
44.
45.             if (!dev)                                   //如果设备不存在，则返回 NULL
46.             return NULL;
47.             s = 0+super_block;                          //指向超级块数组首项
48.             while (s < NR_SUPER+super_block)            //遍历超级块数组
49.             if (s->s_dev == dev) {                      //如果找到指定设备的超级块
50.                     wait_on_super(s);                   //等待该超级块解锁
51.                             if (s->s_dev == dev)        //经过睡眠，超级块可能有变动，再次检查超级
                                                            //块设备号
52.                             return s;                   //如果仍为指定设备的超级块，返回该超级块指针
53.                             s = 0+super_block;          //否则，指向超级块数组首项，重新检测
54.                     } else
55.                             s++;                        //如果当前搜索项不是需要的超级块，则检测下
                                                            //一个超级块
56.         return NULL;                                    //如果未找到指定设备超级块，返回 NULL
57.       }
        //释放指定设备的超级块
58.         void put_super(int dev)
59.         {
60.             struct super_block * sb;
61.             int i;
62.
63.             if (dev == ROOT_DEV) {                      //如果指定设备为根文件系统的设备，则显示错
                                                            //误并返回
64.             printk("root diskette changed: prepare for armageddon\n\r");
65.             return;
66.             }
67.             if (!(sb = get_super(dev)))                 //如果指定设备超级块不存在，则返回
68.                     return;
69.             if (sb->s_covered) {                        //如果该超级块代表的文件系统被安装在其他文
                                                            //件系统的结点上
70.             printk("Mounted disk changed - tssk, tssk\n\r"); //显示错误信息，退出
71.             return;
72.             }
73.             lock_super(sb);                             //锁定该超级块
74.             sb->s_dev = 0;                              //将设备号置为 0
75.             for(i=0;i<MINIX_I_MAP_SLOTS;i++) //释放 i 结点位图缓冲区
76.             brelse(sb->s_imap[i]);
77.             for(i=0;i<MINIX_Z_MAP_SLOTS;i++)//释放逻辑位图缓冲区
78.                     brelse(sb->s_zmap[i]);
79.         free_super(sb);                                 //对该超级块解锁。返回
80.         return;
81.           }
        //从指定设备上读取超级块
82.           static struct super_block * read_super(int dev)
83.           {
84.               struct super_block * s;
```

```
85.             struct buffer_head * bh;
86.             int i,block;
87.
88.             if (!dev)                                   //如果指定设备号无效，返回 NULL
89.             return NULL;
90.             check_disk_change(dev);                     //检查该设备是否更换过盘片
91.             if (s = get_super(dev))                     //如果在内存的超级块数组中找到所需超级块
92.             return s;                                   //返回该超级块指针
93.             for (s = 0+super_block ;; s++) {            //否则，遍历超级块数组，寻找空闲超级块
94.             if (s >= NR_SUPER+super_block)              //如果未找到，返回 NULL
95.                     return NULL;
96.                     if (!s->s_dev)                      //如果找到，跳出循环
97.                             break;
98.             }
99.         //得到的空闲超级块要作为指定设备的超级块，首先对空闲超级块进行初始化
100.            s->s_dev = dev;
101.            s->s_mounted = NULL;
102.            s->s_covered = NULL;
103.            s->s_time = 0;
104.            s->s_rd_only = 0;
105.            s->s_dirt = 0;
106.        lock_super(s);                                  //锁定该超级块
107.        if (!(bh = bread(dev,1))) {                     //从指定设备上读取超级块信息
108.                    s->s_dev=0;                         //如果读取失败，则释放空闲超级块
109.            free_super(s);                              //对空闲超级块解锁。返回 NULL
110.            return NULL;
111.            }
112.            *((struct minix_super_block *) s) =         //将从设备上读取的超级块信息复制到得
113.            *((struct minix_super_block *) bh->b_data);     //空闲超级块中
114.        brelse(bh);                                     //释放缓冲区
115.        if (s->s_magic != MINIX_SUPER_MAGIC) {  //如果读取的超级块的魔幻数不是 MINIX 文
116.                                                        //件系统的魔幻数，即设备上的文件系统不是
                                                            //MINIX 文件系统
117.            s->s_dev = 0;                               //释放得到的超级块
118.            free_super(s);                              //对该超级块解锁，返回 NULL
119.            return NULL;
120.            }
121.            for (i=0;i < MINIX_I_MAP_SLOTS;i++) //初始化得到的超级块的 i 结点位图缓冲块指针
122.            s->s_imap[i] = NULL;                        //数组
123.        for (i=0;i < MINIX_Z_MAP_SLOTS;i++)             //初始化得到的超级块的逻辑位图缓冲块指针
124.                    s->s_zmap[i] = NULL;                //数组
125.                    block=2;                            //在设备上，i 结点位图从第 2 个逻辑块开始。
126.        for (i=0 ; i < s->s_imap_blocks ; i++)          //从设备读取 i 结点位图逻辑块到缓冲区，并将缓
127.                    if (s->s_imap[i]=bread(dev,block))  //冲头指针赋给超级块的 i 结点位图缓冲块数组
128.                    block++;
129.            else                                        //如果读取失败，则跳出循环
130.                            break;
131.            for (i=0 ; i < s->s_zmap_blocks ; i++)      //从设备读取逻辑块位图的逻辑块到缓冲区，并将
132.        if (s->s_zmap[i]=bread(dev,block))              //缓冲头指针赋给超级块的逻辑块位图缓冲块数组
133.                    block++;
```

```
134.                    else                                    //如果读取失败，则跳出循环
135.                            break;
136.            if (block != 2+s->s_imap_blocks+s->s_zmap_blocks) {   //如果 block 不等于设备上逻辑
137.            //位图后的第一个逻辑块号，表明文件系统位图存在问题，超级块逻辑位图
138.            //缓冲块指针数组初始化失败
139.                    for(i=0;i<MINIX_I_MAP_SLOTS;i++) //释放已申请的 i 结点位图缓冲块
140.                    brelse(s->s_imap[i]);
141.            for(i=0;i<MINIX_Z_MAP_SLOTS;i++)        //释放已申请的逻辑块位图缓冲块
142.                            brelse(s->s_zmap[i]);
143.                    s->s_dev=0;                             //释放该超级块
144.                    free_super(s);                                  //对该超级块解锁。返回 NULL
145.                    return NULL;
146.            }
        //运行到此，表明上述操作全部成功
147.            s->s_imap[0]->b_data[0] |= 1;           //将 i 结点位图的第一个比特位置 1，防止 0 号 i
                                                        //结点被使用
148.        s->s_zmap[0]->b_data[0] |= 1;               //将逻辑块位图的第一个比特位置 1，防止 0 号
                                                        //逻辑块被
149.        free_super(s);                              //使用。对该超级块解锁，返回超级块指针
150.            return s;
151.        }
      //卸载指定设备文件名的文件系统
152.          int sys_umount(char * dev_name)
153.      {
154.            struct inode * inode;
155.            struct super_block * sb;
156.            int dev;
157.
158.            if (!(inode=namei(dev_name)))           //获取与指定设备文件名对应的 i 结点
159.            return -ENOENT;                         //失败，则返回错误码
160.            dev = inode->i_rdev;                    //得到指定设备的设备号
161.            if (!S_ISBLK(inode->i_mode)) {          //如果该设备不是块设备
162.            iput(inode);                            //释放该 i 结点，返回错误码
163.            return -ENOTBLK;
164.            }
165.            iput(inode);                            //释放指定设备文件名的 i 结点
166.            if (dev==ROOT_DEV)                      //如果指定设备是装载根文件系统的设备
167.            return -EBUSY;                          //则返回错误码
168.        if (!(sb=get_super(dev)) || !(sb->s_covered))   //如果获取该设备的超级块失败或该设备文
169.                    return -ENOENT;                 //件系统未被安装，则返回错误码
170.        if (!sb->s_covered->i_mount)                //如果超级块被安装到的 i 结点没有对安装标志
171.            printk("Mounted inode has i_mount=0\n"); //置位，则打印错误信息
172.            for (inode=inode_table+0 ; inode<inode_table+NR_INODE ; inode++) //遍历内存 i 结点
173.            if (inode->i_dev==dev && inode->i_count)  //表，如果该设备的 i 结点正在被使用
174.                    if (inode == sb->s_mounted && inode->i_count == 1)//同时，如果该结点是指
175.                                    continue;       //定设备文件系统的根结点并且只被当前进程使
                                                        //用，则继续
176.                    else                            //否则，返回错误码
177.                            return -EBUSY;
178.            sb->s_covered->i_mount=0;               //复位指定设备文件系统被安装到的 i 结点的安
```

```
                                                    //装标志
179.            iput(sb->s_covered);                //并释放该结点
180.            sb->s_covered = NULL;               //置超级块中文件系统被安装到的 i 结点为 NULL
181.            iput(sb->s_mounted);                //释放该设备文件系统的根结点
182.            sb->s_mounted = NULL;               //置超级块中文件系统根结点字段为 NULL
183.            put_super(dev);                     //释放该设备的超级块
184.            sync_dev(dev);                      //同步高速缓冲区与该设备上的数据，返回 0
185.            return 0;
186.    }
        //安装文件系统
187.    int sys_mount(char * dev_name, char * dir_name, int rw_flag)
188.        {
189.            struct inode * dev_i, * dir_i;
190.            struct super_block * sb;
191.            int dev;
192.
193.            if (!(dev_i=namei(dev_name)))       //获取指定的设备文件名的 i 结点
194.                return -ENOENT;                 //如果失败，返回错误码
195.                dev = dev_i->i_rdev;            //得到指定设备的设备号
196.                if (!S_ISBLK(dev_i->i_mode)) {  //如果该设备不是块设备
197.                iput(dev_i);                    //释放该 i 结点，返回错误码
198.                return -EPERM;
199.        }
200.            iput(dev_i);                        //释放该设备文件的 i 结点
201.            if (!(dir_i=namei(dir_name)))       //获取指定的目录文件名的 i 结点
202.            return -ENOENT;                     //如果失败，返回错误码
203.        if (dir_i->i_count != 1 || dir_i->i_ino == MINIX_ROOT_INO) { //如果该 i 结点除被本进程引
204.                //用外，还被其他进程引用或者该 i 结点号是根文件系统的结点号
205.            iput(dir_i);                        //则释放该 i 结点，返回错误码
206.            return -EBUSY;
207.            }
208.            if (!S_ISDIR(dir_i->i_mode)) {      //如果该 i 结点不是目录结点
209.                iput(dir_i);                    //则释放该 i 结点，返回错误码
210.            return -EPERM;
211.        }
212.            if (!(sb=read_super(dev))) {        //读取指定设备文件系统的超级块
213.                iput(dir_i);                    //如果失败，释放该 i 结点，返回错误码
214.            return -EBUSY;
215.        }
216.            if (sb->s_covered) {                //如果指定设备的文件系统已经被安装
217.                iput(dir_i);                    //则释放该 i 结点，返回错误码
218.            return -EBUSY;
219.        }
220.            if (dir_i->i_mount) {               //如果在该 i 结点上已经安装了文件系统
221.                iput(dir_i);                    //则释放该 i 结点，返回错误码
222.            return -EPERM;
223.        }
224.            if (!(sb->s_mounted = iget(dev,MINIX_ROOT_INO))) { //获取指定设备文件系统的根节
225.                iput(dir_i);                    //点，如果失败，则释放该 i 结点，返回错误码
226.            return -EPERM;
227.        }
228.            sb->s_covered=dir_i;                //将指定设备的文件系统安装在该 i 结点上
```

```
229.            dir_i->i_mount=1;                    //置位该 i 结点的安装标志
230.        dir_i->i_dirt=1;                         /*注意！我们没有释放该结点*/
231.        return 0;                                /*我们将在 umount 中释放*/
232.          }
        //安装根文件系统
233.          void mount_root(void)
234.        {
235.          int i,free;
236.                struct super_block * p;
237.                struct inode * mi;
238.
239.                if (32 != sizeof (struct minix_inode))        //如果MINIX 文件系统i 结点结构不是32 字节
240.                panic("bad i-node size");                     //则停机
241.                for(i=0;i<NR_FILE;i++)                        //初始化文件表数组
242.                      file_table[i].f_count=0;                //置文件结构中的引用计数为 0
243.                if (MAJOR(ROOT_DEV) == 2) {                   //如果根文件系统所在设备是软盘
244.                printk("Insert root floppy and press ENTER");//打印“插入跟文件系统盘并按回
245.                      wait_for_keypress();                    //车”，等待按键、
246.          }
247.          for(p = &super_block[0] ; p < &super_block[NR_SUPER] ; p++) {     //初始化超级块数组
248.                p->s_dev = 0;                                 //将超级块的设备号置 0
249.                p->s_lock = 0;                                //复位超级块的锁定标志
250.                p->s_wait = NULL;                             //等待队列置 NULL
251.                }
252.                if (!(p=read_super(ROOT_DEV)))                //读取根设备上超级块
253.                panic("Unable to mount root");                //如果失败，停机
254.          if (!(mi=iget(ROOT_DEV,MINIX_ROOT_INO)))  //读取根设备文件系统的根结点
255.                      panic("Unable to read root i-node");    //如果失败，停机
256.          mi->i_count += 3 ;
257.                                        /*注意，逻辑上它被引用了 4 次，而不是 1 次*/
258.                                       //下面将会引用该结点，因此将引用次数增加 3
259.          p->s_mounted = p->s_covered = mi;                   //将根文件系统的根结点和被安装到的i 结点
260.          current->pwd = mi;                                  //置为该结点。设置当前进程的当前工作目
                                                                  //录为该结点
261.                current->root = mi;                           //设置根目录为该结点
262.                free=0;
263.                i=p->s_nzones;                                //将 i 赋值为逻辑块总数
264.                while (-- i >= 0)                             //统计该设备上空闲逻辑块数，遍历逻辑块
                                                                  //位图
265.                if (!set_bit(i&8191,p->s_zmap[i>>13]->b_data))    //如果该比特位为 0
266.                      free++;                                 //则 free 加 1
267.                printk("%d/%d free blocks\n\r",free,p->s_nzones); //打印出空闲逻辑块数/总逻辑块数
268.                free=0;
269.                i=p->s_ninodes+1;                             //将 i 赋值为 i 结点总数+1，将 0 结点
                                                                  //也统计在内
270.                while (-- i >= 0)                             //统计该设备上空闲 i 结点数，遍历 i
                                                                  //结点位图
271.                if (!set_bit(i&8191,p->s_imap[i>>13]->b_data))    //如果该比特位为 0
272.                      free++;                                 //则 free 加 1
273.                printk("%d/%d free inodes\n\r",free,p->s_ninodes); //打印出空闲 i 结点数/总 i 结点数
274.        }
```

源代码第 19～26 行定义了对超级块上锁的函数。第 27～33 行定义了对超级块解锁的函数。第 34～40 行定义了等待指定超级块解锁的函数，第 41～81 行分别定义了获取和释放指定设备的超级块的函数。

8.7 file_table.c 文件

该文件定义了文件数组，主要用来记录系统中的文件，从而对文件进行管理。本文件中没有包含任何函数，只是定义了文件表数组。file_table.c 文件的详细代码注释如下：

```
#include <Linux/fs.h>                    //文件系统头文件。定义了有关文件系统的一些常数和结构
struct file file_table[NR_FILE];         //文件表数组
```

8.8 block_dev.c 文件

block_dev.c、char_dev.c、pipe.c、file_dev.c 都是为 read_write.c 服务的。这 4 个程序可以看作是系统调用与块设备、字符设备、管道和文件系统的接口驱动程序。

8.8.1 功能描述

本程序实现了对块设备文件的操作，例如将块设备的数据读出与写入等。其中的函数只被系统调用函数 read ()和 write()所使用。

函数 block_write()首先将文件指针位置所在的数据块读入缓冲区，然后将用户数据复制到这些缓冲块中，最终这些缓冲块将被写入设备。

block_read()的操作方式与 block_write()相似，首先将文件指针位置所在的数据块读入缓冲区，然后将这些缓冲块中的数据复制到用户缓冲区。

这里的用户缓冲区是用户程序在开始执行时由系统分配的，或者是在执行过程中动态申请的。用户缓冲区使用的虚拟线性地址，在调用本函数之前，系统会将虚拟线性地址映射到主内存区中相应的内存页中。

8.8.2 代码注释

block_dev.c 文件的详细代码注释如下：

```
1.      #include <errno.h>                          //错误号头文件。包含系统中各种出错号
2.      #include <linux/sched.h>                    //调度程序头文件。定义了 task_struct 和任
                                                    //务 0 的数据。一些关于描述符参数设置和
                                                    //获取以及 switch_to()的嵌入式汇编函数宏
3.      #include <linux/kernel.h>                   //内核头文件。定义了一些内核常用函数的
                                                    //原形
4.      #include <asm/segment.h>                    //段操作头文件。定义了有关段寄存器操作
                                                    //的嵌入式汇编函数
5.      #include <asm/system.h>                     //系统头文件。定义了设置或修改描述符、
                                                    //中断门等的嵌入式汇编宏
6.
7.      extern int *blk_size[];
    //从指定的用户缓冲区向指定设备文件的指定位置写入指定长度字节数据
8.          int block_write(struct inode * inode, struct file * filp, char * buf, int count)
9.      {
10.       int block = filp->f_pos >> BLOCK_SIZE_BITS;   //由偏移量换算出数据要写入的第
                                                        //一个盘块号
```

```
11.             int offset = filp->f_pos & (BLOCK_SIZE-1); //由偏移量换算出在要写入的第一个盘块
12.                                                        //中的偏移值
13.             int chars;
14.             int written = 0;
15.             int size;
16.             unsigned int dev;
17.             struct buffer_head * bh;
18.             register char * p;
19.
20.             dev = inode->i_rdev;              //获取指定设备文件所对应的设备号
        //获取该设备的盘块数目
21.             if (blk_size[MAJOR(dev)])
22.             size = blk_size[MAJOR(dev)][MINOR(dev)];
23.         else
24.                 size = 0x7fffffff;
25.             while (count>0) {                 //循环写入数据
26.                 if (block >= size)            //如果当前盘块号大于指定设备的盘块数目
27.                 return written;               //返回已写字节数
28.             chars = BLOCK_SIZE - offset;      //计算在当前的盘块中可写入的字节数
29.                 if (chars > count)            //如果此次可写入的字节数大于需要写入的字节数
30.                 chars=count;                  //则置 chars 为需要写入的字节数
31.             if (chars == BLOCK_SIZE)          //如果要写入的字节数正好等于一个盘块的大小
32.                 bh = getblk(dev,block);       //则申请 1 块高速缓冲块
33.             else                              //否则，就需要读入将被写入部分数据的数据块，并
                                                  //预读下两块数据
34.                 bh = breada(dev,block,block+1,block+2,-1);
35.                 block++;                      //将盘块号加 1，指向下一个盘块
36.             if (!bh)                          //如果缓冲块操作失败
37.                 return written?written:-EIO; //如果已写入数据，返回写入的字节数，如果
38.                                               //没有，则返回错误号
39.             p = offset + bh->b_data;          //令 p 指向数据写入缓冲块的开始位置
40.             offset = 0;                       //置偏移值为 0
41.                 filp->f_pos += chars;         //将偏移指针前移此次要写入的字节数
42.             written += chars;                 //已写入的字节数增加此次要写入的字节数
43.             count -= chars;                   //需要写入的字节数减去此次要写入的字节数
44.                 while (chars-->0)             //从用户缓冲区复制 chars 个字节到缓冲块中
45.                 *(p++) = get_fs_byte(buf++);
46.                 bh->b_dirt = 1;               //置位该缓冲块修改标志
47.                 brelse(bh);                   //释放该缓冲块
48.             }
49.         return written;                       //返回已写入的字节数
50.     }
    //从指定设备文件的指定位置读取指定长度的数据到指定的用户缓冲区
51.         int block_read(struct inode * inode, struct file * filp, char * buf, int count)
52.     {
53.         unsigned int block = filp->f_pos >> BLOCK_SIZE_BITS; //由偏移量换算出要读出数据所
54.                                                              //在的第一个盘块号
55.         unsigned int offset = filp->f_pos & (BLOCK_SIZE-1);  //由偏移量换算出要出的数据在
56.                                                              //第一个盘块中的偏移值
57.             unsigned int chars;
```

```
58.             unsigned int size;
59.             unsigned int dev;
60.             int read = 0;
61.             struct buffer_head * bh;
62.             register char * p;
        //获取指定设备文件所对应的设备号
63.             dev = inode->i_rdev;
        //获取该设备的盘块数目
64.         if (blk_size[MAJOR(dev)])
65.             size = blk_size[MAJOR(dev)][MINOR(dev)];
66.             else
67.                 size = 0x7fffffff;
68.             while (count>0) {                          //循环读出数据
69.             if (block >= size)                         //如果当前盘块号大于指定设备的盘块数目
70.                 return read;                           //返回已读字节数
71.                 chars = BLOCK_SIZE-offset;             //计算在当前的盘块中可读出的字节数
72.                 if (chars > count)                     //如果此次可读出的字节数大于需要读出的字节数
73.                 chars = count;                         //则置 chars 为需要读出的字节数
74.             if (!(bh = breada(dev,block,block+1,block+2,-1)))    //读取当前盘块到缓冲块，并预
75.                                                        //读下两块数据
76.                 return read?read:-EIO;                 //如果失败，则如果已读出数据，返回读出的字
                                                           //节数
77.                                                        //如果没有读出，则返回错误码
78.                 block++;                               //将盘块号加 1，指向下一个盘块
79.                 p = offset + bh->b_data;               //令 p 指向从缓冲块读取数据的开始位置
80.                 offset = 0;                            //置偏移值为 0
81.                 filp->f_pos += chars;                  //将偏移指针前移此次要读出的字节数
82.                 read += chars;                         //已读出的字节数增加此次要读出的字节数
83.                 count -= chars;                        //需要读出的字节数减去此次要读出的字节数
84.                 while (chars-->0)                      //从缓冲块中复制 chars 个字节到用户缓冲区
85.                 put_fs_byte(*(p++),buf++);
86.                 brelse(bh);                            //释放该缓冲块
87.             }
88.             return read;                               //返回已读出的字节数
89.         }
```

8.9 pipe.c 文件

管道通信是进程间通信的几种主要手段之一。正常情况下，对于一个管道会有两个文件句柄，分别用于读管道和写管道。这两个句柄都指向一个虚拟（因为硬盘上没有这样的问题）的文件，叫做管道文件。管道是一个定长的缓冲区，具有一个头指针和一个尾指针，数据由管道尾部读取，从管道头部写入。

本程序主要实现了读管道数据和写管道数据的功能，其中的函数只被系统调用函数 read ()和 write()所使用。函数 pipe_read()用于管道数据的读取。当读者要从管道取数据时，可能管道中为空，在这种情况下，当前进程都只能睡眠，等待对方写入数据。如果管道不为空，则读取数据并调整管道的头指针，最后将数据复制到用户的缓冲区中。如果管道被读空，则唤醒等待的写者；函数 pipe_write()用于管道数据的写入，其操作方式与读管道类似；函数 sys_pipe()用于创建管道；函数 pipe_ioctl()用于管道的输入输出控制。

pipe.c 文件的详细代码注释如下：

```
1.      #include <signal.h>          //信号头文件。定义了内核中使用的所有信号的名称和基本操作函数
2.      #include <errno.h>           //错误号头文件。包含系统中各种出错号
3.      #include <termios.h>
4.      #include <fcntl.h>           //文件控制头文件。用于文件及其描述符的操作控制常数符号的定义
5.      #include <linux/sched.h>     //调度程序头文件。定义了 task_struct 和任务 0 的数据。一些关
                                     //于描述符参数设置和获取以及 switch_to()的嵌入式汇编函数宏
6.
7.      #include <asm/segment.h>     //段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数
8.      #include <linux/kernel.h>    //内核头文件。定义了一些内核常用函数的原型管道读操作函数
9.      int pipe_read(struct inode * inode, struct file * filp, char * buf, int count)
10.     {
11.       int chars, size, read = 0;
12.
13.       if (!(filp->f_flags & O_NONBLOCK))                    //如果指定管道没有设置非阻塞标志
14.             while (!PIPE_SIZE(*inode)) {                    //如果管道为空，则执行循环操作
15.                   wake_up(& PIPE_WRITE_WAIT(*inode));       //唤醒等待该管道的写进程
      //如果没有写管道者，则返回 0
16.                   if (inode->i_count != 2)                  /*有写进程？ */
17.                         return 0;
18.                   if (current->signal & ~current->blocked)  //如果当前进程信号没有全部被阻塞
19.                         return -ERESTARTSYS;                //返回错误码
20.                   interruptible_sleep_on(& PIPE_READ_WAIT(*inode)); //否则，将当前进程列
21.                                                             //入等待读管道的队列
22.             }
23.       while (count>0 && (size = PIPE_SIZE(*inode))) {   //若欲读取的字节数大于 0 并且管道不
24.             chars = PAGE_SIZE-PIPE_TAIL(*inode); //为空，取管道尾到缓冲区末端的字节数
25.             if (chars > count)                          //如果该字节数大于还需读取的字节数
26.                   chars = count;                        //则令 chars 为还需读取的字节数
27.             if (chars > size)                           //如果 chars 大于当前管道中的数据长度
28.                   chars = size;                         //则令 chars 为当前管道中的数据长度
29.             count -= chars;                             //还需读取的字节数减掉此次可读字节数
30.             read += chars;                              //已读字节数加上此次可读字节数
31.             size = PIPE_TAIL(*inode);                   //令 size 指向管道尾
32.             PIPE_TAIL(*inode) += chars;                 //管道尾前移此次读取的字节数
33.             PIPE_TAIL(*inode) &= (PAGE_SIZE-1);     //若尾指针超过管道末端则绕回
34.             while (chars-->0)                           //将管道中的数据复制到用户缓冲区中
35.                   put_fs_byte(((char *)inode->i_size)[size++],buf++);
36.       }
37.       wake_up(& PIPE_WRITE_WAIT(*inode));   //唤醒等待该管道的写进程
38.       return read;                          //返回读取的字节数
39.     }
      //管道写操作函数
40.     int pipe_write(struct inode * inode, struct file * filp, char * buf, int count)
41.     {
42.       int chars, size, written = 0;
43.
44.       while (count>0) {                             //若仍有需要写入的数据，则执行循环操作
45.             while (!(size=(PAGE_SIZE-1)-PIPE_SIZE(*inode))) { //如果指定管道已经写满了
46.                   wake_up(& PIPE_READ_WAIT(*inode));      //唤醒等待该管道的读进程
```

```
47.                 if (inode->i_count != 2) {                    //如果没有读该管道的进程
48.                     current->signal |= (1<<(SIGPIPE-1)); //则向进程发送 SIGPIPE 信号
49.                     return written?written:-EINTR;    //返回已写的字节数，如果已写字节数
50.                                                       //为 0，返回错误码
51.                 }
52.                 if (current->signal & ~current->blocked) //否则，如果当前进程信号没有全部被阻塞
53.                     return written?written:-EINTR;    //则返回写的字节数，如果已写字节数
54.                                                       //阻塞，为 0，返回错误码
55.                 interruptible_sleep_on(&PIPE_WRITE_WAIT(*inode)); //否则，将当前进程列
56.                                                       //入等待写管道的队列
57.             }
58.             chars = PAGE_SIZE-PIPE_HEAD(*inode); //计算管道头部到缓冲区末端的字节数
59.             if (chars > count)                        //如果该字节数大于需要写入的字节数
60.                 chars = count;                        //则令 chars 为还需写入的字节数
61.             if (chars > size)                         //如果 chars 大于当前管道中空闲空间长度
62.                 chars = size;                         //则令 chars 为当前管道中空闲空间长度
63.             count -= chars;                           //还需写入的字节数减掉此次要写入的字
                                                          //节数
64.             written += chars;                         //已写字节数加上此次要写入的字节数
65.             size = PIPE_HEAD(*inode);                 //令 size 指向管道头
66.             PIPE_HEAD(*inode) += chars;               //管道头前移此次写入的字节数
67.             PIPE_HEAD(*inode) &= (PAGE_SIZE-1);       //若头指针超过管道末端则绕回
68.             while (chars-->0)                         //将用户缓冲区中 chars 个字节的数据写入
                                                          //管道
69.                 ((char *)inode->i_size)[size++]=get_fs_byte(buf++);
70.         }
71.         wake_up(& PIPE_READ_WAIT(*inode));            //唤醒等待该管道的读进程
72.         return written;                               //返回已写字节数
73.         }
74.
        //创建管道函数
75.         int sys_pipe(unsigned long * fildes)
76.         {
77.           struct inode * inode;
78.           struct file * f[2];
79.           int fd[2];
80.           int i,j;
81.
82.           j=0;
83.           for(i=0;j<2 && i<NR_FILE;i++)               //遍历系统文件表，获取两个空闲项
84.                 if (!file_table[i].f_count)           //如果该项的引用计数为 0，即该项空闲
85.                       (f[j++]=i+file_table)->f_count++; //则将该项的指针存入文件结构指针数组
86.                                                       //并置该项的引用计数为 1
87.           if (j==1)                                   //如果只有一个空闲项
88.                 f[0]->f_count=0;                      //则释放已获得的空闲项
89.           if (j<2)                                    //如果没有找到两个空闲项
90.                 return -1;                            //则返回-1
91.           j=0;
92.           for(i=0;j<2 && i<NR_OPEN;i++)               //遍历进程的文件结构指针表
93.                 if (!current->filp[i]) {              //如果该项空闲
```

```
94.                     current->filp[ fd[j]=i ] = f[j];        //则让该项存储已申请的文件结构的指针并将该
                                                                //索引号，即文件句柄，存储到文件句柄数组中
95.
96.                     j++;                                    //j 加 1
97.             }
98.         if (j==1)                                           //如果只有一个空闲文件句柄
99.             current->filp[fd[0]]=NULL;                      //则释放该句柄
100.        if (j<2) {                                          //如果没有找到两个空闲句柄
101.            f[0]->f_count=f[1]->f_count=0;                  //则释放已申请的两个文件结构项
102.            return -1;                                      //返回-1
103.        }
104.        if (!(inode=get_pipe_inode())) {                    //申请管道 i 结点，并分配缓冲区给管道
105.            current->filp[fd[0]] =                          //如果不成功，则释放两个文件句柄
106.            current->filp[fd[1]] = NULL;
107.            f[0]->f_count = f[1]->f_count = 0;              //释放文件结构项
108.            return -1;                                      //返回-1
109.        }
110.        f[0]->f_inode = f[1]->f_inode = inode;              //令两个文件结构都指向该管道 i 结点
111.        f[0]->f_pos = f[1]->f_pos = 0;                      //两个文件结构读写指针都置 0
112.        f[0]->f_mode = 1;                                   /*读*/ //该文件结构的文件模式置为读
113.        f[1]->f_mode = 2;                                   /*写*/ //该文件结构的文件模式置为写
    //将文件句柄复制到对应的用户数组当中
114.        put_fs_long(fd[0],0+fildes);
115.        put_fs_long(fd[1],1+fildes);
116.        return 0;                           //返回 0
117.    }
    //管道输入输出控制函数
118.    int pipe_ioctl(struct inode *pino, int cmd, int arg)
119.    {
120.        switch (cmd) {                                      //根据不同的命令，执行不同的操作
121.            case FIONREAD:                                  //如果是 FIONREAD 命令
122.                verify_area((void *) arg,4);                //验证参数内存空间
123.                put_fs_long(PIPE_SIZE(*pino),(unsigned long *) arg);    //将管道大小复制到
124.                                                            //参数空间
125.                return 0;                                   //返回 0
126.            default:                                        //如果是其他命令
127.                return -EINVAL;                             //返回错误码
128.        }
129.    }
```

8.10 char_dev.c 文件

在 Linux 系统中都是将设备看作文件的形式访问的。该文件主要包括字符设备文件访问函数。通过这些函数，可以方便的对设备进行操作。

1. 功能描述

本程序实现了对字符设备文件的访问功能，主要包括对串口终端设备的读写、对控制台终端的读写、对内存设备文件的读写。

文件系统的操作函数 open()、read()等都通过 rw_char()对所有字符设备文件进行操作。函数

rw_char()是字符设备读写操作的接口函数。其他字符设备通过该函数对字符设备读写函数指针表进行相应字符设备的操作；函数 rw_tty()通过调用相应的驱动程序实现了对控制台终端的读写操作；函数 rw_ttyx()通过调用驱动程序实现了对串口终端设备的读写操作；函数 rw_ram()用于对内存数据的读写，但在本版本内函数内容为空；函数 rw_mem()是对内存数据读写操作函数，在本版本内未实现其内容；函数 rw_kmem()实现了内核数据区的读写操作；函数 rw_port()用于对端口的读写；函数 rw_memory()是内存设备文件读写函数，实现了对内存映像的字节操作；函数 rw_char()是字符设备读写操作的接口函数。其他字符设备通过该函数对字符设备读写函数指针表进行相应字符设备的操作。

2. 代码注释

char_dev.c 文件的详细代码注释如下：

```
1.      #include <errno.h>                  //错误号头文件。包含系统中各种出错号
2.      #include <sys/types.h>              //类型头文件。定义了基本的数据类型
3.      #include <linux/sched.h>            //调度程序头文件。定义了 task_struct 和任务 0 的数据。一
                                            //些关于描述符参数设置和获取以及 switch_to()的嵌入式汇
                                            //编函数宏
4.      #include <linux/kernel.h>           //内核头文件。定义了一些内核常用函数的原型
5.      #include <asm/segment.h>            //段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数
6.      #include <asm/io.h>                 //IO 头文件。定义了对硬件 IO 端口访问的嵌入式汇编宏函
                                            //数：outb()、inb()以及 outb_p()和 inb_p()
    //终端读函数
7.      extern int tty_read(unsigned minor,char * buf,int count,unsigned short flags);
    //终端写函数
8.      extern int tty_write(unsigned minor,char * buf,int count);
    //定义字符设备读写函数的指针
9.      typedef (*crw_ptr)(int,unsigned,char *,int,off_t *,unsigned short);
    //串口终端设备读写函数
10.     static int rw_ttyx(int rw,unsigned minor,char * buf,int count,off_t * pos, unsigned short flags)
11.     {
    //判断读写命令，如果是读命令，则调用终端读函数，返回实际读的字节数
     //如果是写命令，则调用终端写函数，返回实际写的字数
12.         return ((rw==READ)?tty_read(minor,buf,count,flags):
13.         tty_write(minor,buf,count));
14.     }
     //控制台终端读写函数
15.     static int rw_tty(int rw,unsigned minor,char * buf,int count, off_t * pos, unsigned short flags)
16.     {
17.      if (current->tty<0)                //判断当前进程是否有对应的控制终端
18.         return -EPERM;                  //如果没有，返回错误码
19.         return rw_ttyx(rw,current->tty,buf,count,pos,flags);  //否则，调用串口终端设备读写函数
20.                                         //返回实际读写的字节数
21.     }
     //内存数据读写函数
22.     static int rw_ram(int rw,char * buf, int count, off_t *pos)
23.     {
24.         return -EIO;                    //无内容，直接返回错误码
25.     }
     //内存数据读写操作函数
26.     static int rw_mem(int rw,char * buf, int count, off_t * pos)
```

```
27.         {
28.             return -EIO;                                    //无内容，直接返回错误码
29.         }
        //内核数据区读写函数
30.         static int rw_kmem(int rw,char * buf, int count, off_t * pos)
31.         {
32.             int i = *pos;                                   /*当前的读写位置*/
33.         while ( (count-- > 0)   && (i <LOW_MEM) ) {         //当仍有数据要读写，并且 i 小于
34.                                                             // LOW_MEM，则进入循环执行下列操作
35.                 if (rw==READ)                               //如果是读命令
36.                 put_fs_byte( *(char *)i ,buf++);            //则将 i 指向的数据读入缓冲区
37.             else                                            //否则
38.                     return (-EIO);                          //返回错误码
39.                 i++;                                        //i 指向下一个数据
40.         }
41.         i -= *pos;
42.                                 /*计算读写了多少个字节的数据*/
43.         *pos += i;
44.                                 /*更新读写操作指针*/
45.         return (i);
46.                                 /*返回读取的字节数*/
47.         }
        //端口读写操作函数
48.         static int rw_port(int rw,char * buf, int count, off_t * pos)
49.         {
50.         int i=*pos;                                         //当前的读写位置
51.
52.             while (count-->0 && i<65536) {                  //当仍有数据要读写，并且 i 小于 64K 时
53.             if (rw==READ)                                   //如果是读命令
54.                 put_fs_byte(inb(i),buf++);                  //将 i 位置的数据读入缓冲区
55.             else                                            //否则
56.                     outb(get_fs_byte(buf++),i);             //将缓冲区的数据放入 i 位置
57.             i++;                                            //i 指向下一个位置
58.         }
59.             i -= *pos;                                      //计算读写了多少个字节的数据
60.             *pos += i;                                      //更新读写操作指针
61.             return i;                                       //返回读写的字节数
62.         }
        //内存读写操作函数
63.         static int rw_memory(int rw, unsigned minor, char * buf, int count,
64.             off_t * pos, unsigned short flags)
65.         {
66.         switch(minor) {                                     //根据内存设备的子设备号，分别调用不同
                                                                //的内存读写函数
67.                 case 0:
68.                     return rw_ram(rw,buf,count,pos);
69.                 case 1:
70.                     return rw_mem(rw,buf,count,pos);
71.                 case 2:
72.                     return rw_kmem(rw,buf,count,pos);
73.                 case 3:
74.                     return (rw==READ)?0:count;
```

```
75.                     case 4:
76.                         return rw_port(rw,buf,count,pos);
77.                     default:
78.                         return -EIO;
79.                 }
80.             }
        //定义系统中设备种类数
81.         #define NRDEVS ((sizeof (crw_table))/(sizeof (crw_ptr)))
        //字符设备读写函数指针表
82.         static crw_ptr crw_table[]={
83.             NULL,                                   /*空设备*/
84.             rw_memory,                              /* /dev/mem 等 */
85.             NULL,                                   /* /dev/fd 软驱 */
86.             NULL,                                   /* /dev/hd 硬盘 */
87.             rw_ttyx,                                /* /dev/ttyx 串口终端 */
88.             rw_tty,                                 /* /dev/tty 终端 */
89.             NULL,                                   /* /dev/lp 打印机 */
90.             NULL};                                  /* 无名管道 */
        //字符设备读操作函数
91.         int char_read(struct inode * inode, struct file * filp, char * buf, int count)
92.         {
93.             unsigned int major,minor;
94.             crw_ptr call_addr;
95.
96.             major = MAJOR(inode->i_rdev);           //获取主设备号
97.             minor = MINOR(inode->i_rdev);           //获取次设备号
98.             if (major >= NRDEVS)                    //如果设备号超出系统设备数
99.                 return -ENODEV;                     //返回错误码
100.            if (!(call_addr = crw_table[major]))    //获取该设备的读写函数指针
101.            return -ENODEV;                         //如果失败，返回错误码
102.        return call_addr(READ,minor,buf,count,&filp->f_pos,filp->f_flags); //调用该设备的读写操
103.                                                    //作函数，返回读取的字节数
104.    }
        //字符设备写操作函数
105.    int char_write(struct inode * inode, struct file * filp, char * buf, int count)
106.    {
107.        unsigned int major,minor;
108.        crw_ptr call_addr;
109.
110.        major = MAJOR(inode->i_rdev);               //获取主设备号
111.        minor = MINOR(inode->i_rdev);               //获取次设备号
112.        if (major >= NRDEVS)                        //如果设备号超出系统设备数
113.            return -ENODEV;                         //返回错误码
114.        if (!(call_addr=crw_table[major]))          //获取该设备的读写函数指针
115.        return -ENODEV;                             //如果失败，返回错误码
116.    return call_addr(WRITE,minor,buf,count,&filp->f_pos,filp->f_flags); //调用该设备的读写
117.                                                    //操作函数，返回写入的字节数
118.    }
```

源代码中第48～62行定义了端口读写操作的rw-port()函数。第33行代码用来判断记数值是否等于0到LOW-MEM之间的数，从而决定是否执行循环。

8.11 read_write.c 文件

该文件实现了文件操作系统调用 read()、write()、lseek()。read()和 write()将根据不同的文件类型，分别调用相应读写函数。lseek()用于设置文件读写指针。

1. 功能描述

本程序实现了对块设备、字符设备、管道及正规文件访问的上层接口，该接口通过识别文件类型分别调用相应的访问函数，如图 8-3 所示。函数 sys_lseek()根据指定的定位标志，分别重新定位文件读写指针；函数 sys_read()根据文件的 i 结点信息判断文件的类型，从而调用不同的设备或文件的读函数；函数 sys_write()根据文件的 i 结点信息判断文件的类型，从而调用不同的设备或文件的写函数。

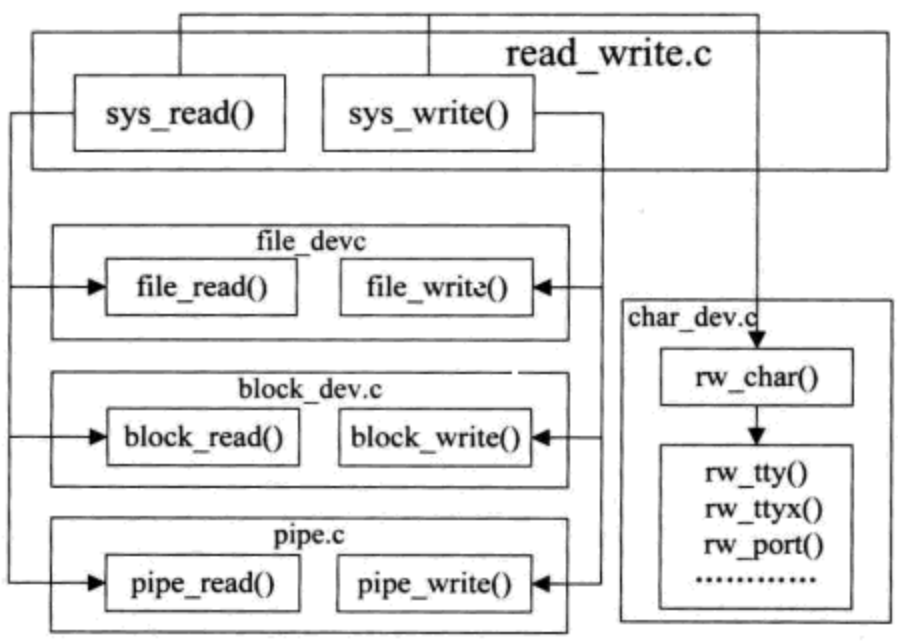


图 8-3　各种类型文件与接口函数

2. 代码注释

read_write.c 文件的详细代码注释如下：

```
1.      #include <sys/stat.h>          //文件状态头文件。主要说明了函数 stat()返回的数据及其结构类型
                                       //一些属性操作测试宏、函数原型
2.      #include <errno.h>             //错误号头文件。包含系统中各种出错号
3.      #include <sys/types.h>         //类型头文件。定义了基本的数据类型
4.      #include <linux/kernel.h>      //内核头文件。定义了一些内核常用函数的原型
5.      #include <linux/sched.h>       //调度程序头文件。定义了 task_struct 和任务 0 的数据。一些关
                                       //于描述符参数设置和获取以及 switch_to()的嵌入式汇编函数宏
6.      #include <asm/segment.h>       //段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数
    //文件读写指针重定位系统调用函数
7.      int sys_lseek(unsigned int fd,off_t offset, unsigned int origin)
8.        {
9.            struct file * file;
10.       int tmp;
11.       if (fd >= NR_OPEN || !(file=current->filp[fd]) || !(file->f_inode))
12.                return -EBADF;
13.           if (origin > 2)                          //如果指定的定位标志大于 2
14.           return -EINVAL;                          //则返回错误码
```

```
15.             if (file->f_inode->i_pipe)                  //如果文件是一个管道文件
16.                     return -ESPIPE;                     //则返回错误码
17.             if (file->f_op && file->f_op->lseek)        //如果文件操作结构存在并且重定位函数存在
18.             return file->f_op->lseek(file->f_inode,file,offset,origin); //则调用该重定位函数
19.             switch (origin) {                           //根据指定的定位标志，分别重新定位文件读写
                                                            //指针
20.             case 0:                                     //当指定的定位标志为 SEEK_SET，即要求以文
                                                            //件开始处为原点设置文件读写指针
21.
22.                     if (offset<0) return -EINVAL;       //如果偏移值小于零，则返回错误码
23.                 file->f_pos=offset;                     //否则，置文件读写指针等于 offset
24.                 break;
25.             case 1:                                     //如果指定的定位标志为 SEEK_CUR，即要求以
                                                            //文件当前读写指针作为原点重定位读写指针
26.                 if (file->f_pos+offset<0) return -EINVAL;   //如果当前指针加上偏移值小于 0
27.                                                         //则返回错误码
28.                     file->f_pos += offset;              //否则当前读写指针加上偏移值
29.                 break;
30.             case 2:                                     //如果指定的定位标志为 SEEK_END，即要求以
                                                            //文件末尾为原点重定位读写指针
31.
32.                 if ((tmp=file->f_inode->i_size+offset) < 0) //如果文件大小加上偏移值小于 0
33.                     return -EINVAL;                     //则返回错误码
34.                     file->f_pos = tmp;                  //重定位读写指针为文件长度加上偏移值
35.         }
36.           return file->f_pos;                           //返回当前文件读写指针
37.           }
        //读文件系统调用函数
38.           int sys_read(unsigned int fd,char * buf,unsigned int count)
39.         {
40.                 struct file * file;
41.                 struct inode * inode;
42.                 if (fd>=NR_OPEN || !(file=current->filp[fd]) || !(inode=file->f_inode))
43.                 return -EBADF;
44.                 if (!(file->f_mode & 1))                //如果文件不允许读
45.                     return -EBADF;                      //返回错误码
46.                 if (!count)                             //如果需要读取的字节数等于 0
47.                 return 0;                               //则返回 0
48.                 verify_area(buf,count);                 //验证存放数据的缓冲区内存限制
49.                 if (file->f_op && file->f_op->read)     //如果文件操作结构存在并且读操作函数存在
50.                 return file->f_op->read(inode,file,buf,count); //则调用该读操作函数
51.                 if (inode->i_pipe)                          //如果为管道文件
52.                     return pipe_read(inode,file,buf,count);   //调用读管道操作函数，返回读取的字节数
53.                 if (S_ISCHR(inode->i_mode))                 //如果是字符型文件
54.                 return char_read(inode,file,buf,count);     //调用读字符设备函数，返回读取的字节数
55.                 if (S_ISBLK(inode->i_mode))                 //如果是块设备文件
56.                 return block_read(inode,file,buf,count);    //调用块设备读操作函数，返回读取的字节
57.             if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) //数。如果是目录或常规文件
58.                     return minix_file_read(inode,file,buf,count);  //调用 Minix 文件读函数，返回读取
```

```
59.         printk("(Read)inode->i_mode=%06o\n\r",inode->i_mode); //字节数。否则，打印文件属性
60.         return -EINVAL;                                          //返回错误码
61.          }
        //写文件系统调用函数
62.          int sys_write(unsigned int fd,char * buf,unsigned int count)
63.        {
64.                 struct file * file;
65.                 struct inode * inode;
        //如果文件句柄大于程序最多所能打开的文件数，或者该句柄对应的文件结构不存在
        //或者文件结构对应的文件 i 结点不存在，则返回错误码
66.         if (fd>=NR_OPEN || !(file=current->filp[fd]) || !(inode=file->f_inode))
67.                 return -EBADF;
68.                 if (!(file->f_mode&2))                          //如果文件不允许写
69.                         return -EBADF;                          //返回错误码
70.                 if (!count)                                     //如果需要读取的字节数等于 0
71.                 return 0;                                       //则返回 0
72.                 if (file->f_op && file->f_op->write)            //如果文件操作结构存在并且写操作函数
                                                                    //存在
73.                 return file->f_op->write(inode,file,buf,count); //则调用该写操作函数
        //下面是默认的写函数功能
74.         if (inode->i_pipe)                                      //如果为管道文件
75.                         return pipe_write(inode,file,buf,count); //调用写管道操作函数，返回写入的字节数
76.         if (S_ISCHR(inode->i_mode))                             //如果是字符型文件
77.                 return char_write(inode,file,buf,count);        //调用写字符设备函数，返回写入的字节数
78.                 if (S_ISBLK(inode->i_mode))                     //如果是块设备文件
79.                 return block_write(inode,file,buf,count);       //调用块设备写操作函数，返回写入的字
80.         if (S_ISREG(inode->i_mode))                             //节数。如果是目录或常规文件
81.                         return minix_file_write(inode,file,buf,count); //调用 Minix 文件写函数，返回读取字
82.         printk("(Write)inode->i_mode=%06o\n\r",inode->i_mode); //节数。否则，打印文件属性
83.         return -EINVAL;                                         //返回错误码
84.          }
```

源代码中第 7 行声明如果没有重定位函数，下面的是默认的处理方式。第 38～61 行声明下面是默认的读函数功能。第 62～84 行声明下面是默认的写函数功能。

8.12 open.c 文件

该文件主要实现了许多与文件操作相关的系统调用。所以在内核中涉及到对文件进行的所有操作都需要调用该文件中定义的这些系统调用函数。

1. 功能描述

本程序主要实现了对文件操作的系统调用。主要包括文件的创建、打开及关闭，文件的访问权限、操作时间和宿主及属性的修改等。函数 sys_ustat()用于取指定设备文件系统信息，但在本版本中没有实现，函数体为空；函数 sys_utime()设置文件访问和修改时间，如果参数 times 指针不为 NULL，则取 utimbuf 结构中的时间信息来设置文件的访问和修改时间。

如果 times 指针是 NULL，则取系统当前时间来设置指定文件的访问和修改时间域。函数 sys_access()是根据屏蔽码检查对文件的访问权限的系统调用；函数 sys_chdir()是改变当前工作目录的系统调用；函数 sys_chroot()是改变根目录为指定的目录的系统调用；函数 sys_chmod()是用于修改文件的文件属性的系统调；函数 sys_chown()是修改文件的文件宿主的系统调；函数

sys_open()是用于打开指定的文件的系统调；函数 sys_creat()是用于创建文件的系统调；函数 sys_close()是关闭文件的系统调用。

2. 代码注释

open.c 文件的详细代码注释如下：

```
1.      #include <string.h>              //字符串头文件。定义了所有字符串操作函数
2.      #include <errno.h>               //错误号头文件。包含系统中各种出错号
3.      #include <fcntl.h>               //文件控制头文件。用于文件及其描述符的操作控制常数符号的
                                         //定义
4.      #include <sys/types.h>           //类型头文件。定义了基本的数据类型
5.      #include <utime.h>               //用户时间头文件。定义了文件访问和修改时间结构 utimbuf{}，
                                         //以及 utime()函数原型
6.      #include <sys/stat.h>            //文件状态头文件。主要说明了函数 stat()返回的数据及其结构类型
                                         //一些属性操作测试宏、函数原型
7.      #include <linux/sched.h>         //调度程序头文件。定义了 task_struct 和任务 0 的数据。一些关
                                         //于描述符参数设置和获取以及 switch_to()的嵌入式汇编函数宏
8.      #include <linux/kernel.h>        //内核头文件。定义了一些内核常用函数的原形
9.      #include <asm/segment.h>         //段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数
    //取指定设备的文件系统信息系统调用函数
10.         int sys_ustat(int dev, struct ustat * ubuf)
11.     {
12.         return -ENOSYS;              //无内容，直接返回错误码
13.         }
    //根据指定的访问和修改时间设置指定文件
14.         int sys_utime(char * filename, struct utimbuf * times)
15.     {
16.             struct inode * inode;
17.             long actime,modtime;
18.
19.             if (!(inode=namei(filename)))      //获取指定文件名的文件 i 结点
20.             return -ENOENT;          //如果失败，返回错误码
21.         if (times) {                 //如果访问和修改时间数据结构指针不为空
22.                 actime = get_fs_long((unsigned long *) &times->actime);//则获取用户设置的访问
23.             modtime = get_fs_long((unsigned long *) &times->modtime);  //时间，获取用户设置
24.                                                                        //的修改时间
25.             } else
26.                 actime = modtime = CURRENT_TIME;  //否则，置访问和修改时间为当前时间
27.         inode->i_atime = actime;     //修改 i 结点的访问时间字段
28.         inode->i_mtime = modtime; //修改 i 结点的修改时间字段
29.         inode->i_dirt = 1;           //置位 i 结点的修改标志
30.         iput(inode);                 //释放该 i 结点
31.         return 0;                    //返回 0
32.         }
    //检查对指定文件的访问权限
33.         int sys_access(const char * filename,int mode)
34.     {
35.         struct inode * inode;
36.             int res, i_mode;
37.
38.             mode &= 0007;                          //清除屏蔽码以外的高比特位
```

```
39.             if (!(inode=namei(filename)))          //获取指定文件名的文件 i 结点
40.             return -EACCES;                        //如果失败，则返回错误码
41.         i_mode = res = inode->i_mode & 0777;       //取文件的属性码
42.             iput(inode);                           //释放该 i 结点
43.         if (current->uid == inode->i_uid)          //如果当前进程是该文件的宿主
44.             res >>= 6;                             //则取文件宿主属性
45.             else if (current->gid == inode->i_gid) //如果当前进程是该文件同属一组
46.             res >>= 6;                             //取文件组属性
47.         if ((res & 0007 & mode) == mode)           //如果文件属性具有查询的属性位，则访问许可
48.                 return 0;                          //返回 0
49.         if ((!current->uid) &&                     //如果当前用户 id 为超级用户，并且屏蔽码执行位为
50.             (!(mode & 1) || (i_mode & 0111)))      //0 或者文件可以被任何人访问
51.                 return 0;                          //则返回 0
52.         return -EACCES;                            //否则，返回错误码
53.     }
    //改变当前工作目录为指定目录的系统调用函数
54.         int sys_chdir(const char * filename)
55.         {
56.             struct inode * inode;
57.
58.             if (!(inode = namei(filename)))        //获取指定文件名对应的 i 结点
59.             return -ENOENT;                        //如果失败，则返回错误码
60.         if (!S_ISDIR(inode->i_mode)) {             //如果指定文件的 i 结点不是目录文件 i 结点
61.                 iput(inode);                       //则释放该 i 结点
62.             return -ENOTDIR;                       //返回错误码
63.         }
64.             iput(current->pwd);                    //释放当前进程的工作目录 i 结点
65.             current->pwd = inode;                  //设置当前进程的工作目录为指定的工作目录
66.             return (0);                            //返回 0
67.     }
    //改变根目录为指定目录的系统调用函数
68.         int sys_chroot(const char * filename)
69.         {
70.             struct inode * inode;
71.
72.         if (!(inode=namei(filename)))              //获取指定文件名对应的 i 结点
73.             return -ENOENT;                        //如果失败，则返回错误码
74.             if (!S_ISDIR(inode->i_mode)) {         //如果指定文件的 i 结点不是目录文件 i 结点
75.             iput(inode);                           //则释放该 i 结点
76.             return -ENOTDIR;                       //返回错误码
77.             }
78.             iput(current->root);                   //释放当前进程的根目录 i 结点
79.             current->root = inode;                 //设置当前进程的根目录为指定的目录
80.             return (0);                            //返回 0
81.         }
    //修改指定文件的文件属性系统调用函数
82.         int sys_chmod(const char * filename,int mode)
83.     {
84.         struct inode * inode;
85.
```

```
86.             if (!(inode=namei(filename)))          //获取指定文件名对应的 i 结点
87.                 return -ENOENT;                    //如果失败，则返回错误码
88.         if ((current->euid != inode->i_uid) && !suser()) { //如果当前进程的有效用户 id 不等于指定
89.                                                    //文件的用户 id，并且当前用户不是超级用户
90.             iput(inode);                           //则释放该 i 结点
91.             return -EACCES;                        //返回错误码
92.         }
93.             inode->i_mode = (mode & 07777) | (inode->i_mode & ~07777); //重置该 i 结点的文件属
94.             inode->i_dirt = 1;                     //性，置位该 i 结点的修改标志
95.             iput(inode);                           //释放该 i 结点
96.             return 0;                              //返回 0
97.         }
    //修改指定文件的文件宿主系统调用函数
98.         int sys_chown(const char * filename,int uid,int gid)
99.     {
100.        struct inode * inode;
101.
102.            if (!(inode=namei(filename)))          //获取指定文件名对应的 i 结点
103.                return -ENOENT;                    //如果失败，则返回错误码
104.        if (!suser()) {                            //如果当前进程不是超级用户
105.            iput(inode);                           //释放该 i 结点
106.                return -EACCES;                    //返回错误码
107.            }
108.            inode->i_uid=uid;                      //设置该 i 结点的用户 id 字段为指定用户 id
109.            inode->i_gid=gid;                      //设置该 i 结点的组 id 字段为指定组 id
110.            inode->i_dirt=1;                       //置位该 i 结点的修改标志
111.            iput(inode);                           //释放该 i 结点
112.            return 0;                              //返回 0
113.    }
    //打开指定文件的系统调用函数
114.    int sys_open(const char * filename,int flag,int mode)
115.        {
116.            struct inode * inode;
117.            struct file * f;
118.            int i,fd;
119.
120.            for(fd=0 ; fd<NR_OPEN ; fd++)          //遍历进程的结构文件指针数组，搜索空闲项
121.                if (!current->filp[fd])            //如果该项空闲，则跳出循环
122.                break;
123.                if (fd>=NR_OPEN)                   //如果已没有空闲项
124.                return -EINVAL;                    //则返回错误码
125.            current->close_on_exec &= ~(1<<fd);    //设置执行时关闭文件句柄位图，复位对应比特位
126.            f=0+file_table;                        //f 指向文件结构表首项
127.            for (i=0 ; i<NR_FILE ; i++,f++)        //遍历文件结构表，搜索空闲项
128.                if (!f->f_count) break;            //如果找到空闲项，跳出循环
129.        if (i>=NR_FILE)                            //如果没有找到空闲项
130.            return -EINVAL;                        //返回错误码
131.            (current->filp[fd]=f)->f_count++;      //让获取的文件句柄对应的文件结构指针指向获
                                                       //取的文件结构，并令文件结构引用计数加 1
132.
```

```
133.        if ((i=open_namei(filename,flag,mode,&inode))<0) {   //调用函数执行打开操作
134.                current->filp[fd]=NULL;                      //如果出错，释放刚申请到的文件结构指针
135.                f->f_count=0;                                //释放申请的文件结构
136.                return i;                                    //返回错误码
137.            }
138.            f->f_op = NULL;                                  //置空文件结构的文件操作结构指针
139.            f->f_mode = "\001\002\003\000"[flag & O_ACCMODE]; //设置文件结构的文件属性
140.            f->f_flags = flag;                               //设置文件结构标志
141.            f->f_count = 1;                                  //置引用计数为 1
142.            f->f_inode = inode;                              //设置 i 结点字段
143.            f->f_pos = 0;                                    //文件读写指针初始化为 0
144.            if (inode->i_op && inode->i_op->open)//如果指定文件结点的文件操作字段不为
145.                                                             //空，并且存在打开函数
146.            if (i = inode->i_op->open(inode,f)) {            //则调用该打开函数打开文件
147.                    iput(inode);                             //如果失败，释放指定文件 i 结点
148.                f->f_count=0;                                //释放申请的文件结构
149.                current->filp[fd]=NULL;                      //置空申请的文件结构指针
150.                    return i;                                //返回错误码
151.            }
152.        return (fd);                                         //返回文件句柄
153.        }
    //创建文件系统调用函数
154.        int sys_creat(const char * pathname, int mode)
155.     {
    //调用打开文件系统调用函数创建文件，返回文件句柄
156.        return sys_open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode);
157.        }
    //关闭指定文件系统调用函数
158.     int sys_close(unsigned int fd)
159.     {
160.            struct file * filp;
161.
162.            if (fd >= NR_OPEN)                               //若文件句柄值大于程序能打开的最大文件数
163.                return -EINVAL;                              //则返回错误码
164.        current->close_on_exec &= ~(1<<fd);                  //复位进程的执行时关闭文件句柄位图的对应位
165.        if (!(filp = current->filp[fd]))                     //如果该文件句柄对应的文件结构指针为空
166.                return -EINVAL;                              //则返回错误码
167.        current->filp[fd] = NULL;                            //置空该文件句柄对应的文件结构指针
168.        if (filp->f_count == 0)                              //如果该文件结构的引用计数为 0
169.                panic("Close: file count is 0");             //打印警告信息，停机
170.        if (--filp->f_count)                                 //将该文件结构的引用计数减 1
171.            return (0);                                      //返回 0
172.            iput(filp->f_inode);                             //释放指定文件的 i 结点
173.        return (0);                                          //返回 0
174.     }
```

源代码中第 33～35 行定义了检查对指定文件的访问权限的函数，在检查过程中该用真实用户 id 还是有效用户 id，BSD 系统使用了真实用户 id，从而使该调用可以被 setid 使用。

第 82～113 行的代码需要交换真实用户 id 和有效用户 id（临时地），然后使用 suser 有效用户 id 来进行判断。

8.13 exec.c 文件

该文件中主要的函数是 do_execve()，它是系统中断调用功能号_NR_execve()调用的 C 处理函数。

1. 功能描述

本程序主要实现了对可执行文件和 shell 脚本的加载与执行。其中：

函数 sys_uselib()是加载用户库文件的系统调用；

函数 create_tables()在新用户堆栈中创建环境和参数变量指针表；

函数 count()计算命令行参数/环境参数的个数；

函数 copy_strings()复制指定个数的参数字符串到参数和环境空间；

函数 change_ldt()修改局部描述符表中的描述符基址和段限长；

do_execve()是 execve()系统中断调用函数。

2. 代码注释

exec.c 文件的详细代码注释如下：

```
1.      #include <signal.h>          //信号头文件。定义了内核中使用的所有信号的名称和基本操作
                                     //函数
2.      #include <errno.h>           //错误号头文件。包含系统中各种出错号
3.      #include <string.h>          //字符串头文件。定义了所有字符串操作函数
4.      #include <sys/stat.h>        //文件状态头文件。主要说明了函数 stat()返回的数据及其结构
                                     //类型
                                     //一些属性操作测试宏、函数原型
5.      #include <a.out.h>           //a.out 头文件。定义了二进制执行文件 a.out(Assembley out)的格式
6.      #include <linux/fs.h>        //文件系统头文件。定了文件表结构
7.      #include <linux/sched.h>     //调度程序头文件。定义了 task_struct 和任务 0 的数据。一些关
                                     //于描述符参数设置和获取以及 switch_to()的嵌入式汇编函数宏
8.      #include <linux/kernel.h>    //内核头文件。定义了一些内核常用函数的原型
9.      #include <linux/mm.h>        //内存管理头文件。定义了内存页面的大小和页面释放函数原型
10.     #include <asm/segment.h>     //段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数
11.
12.     extern int sys_exit(int exit_code);
13.     extern int sys_close(int fd);
14.
15.     #define MAX_ARG_PAGES 32
    //加载用户库文件
16.     int sys_uselib(const char * library)
17.     {
18.             struct inode * inode;
19.             unsigned long base;
20.
21.             if (get_limit(0x17) != TASK_SIZE)      //如果数据段段限长不等于用户进程空间大小
22.             return -EINVAL;                        //则返回错误码
23.       if (library) {                               //如果指定的库文件不为空
24.                     if (!(inode=namei(library)))   //获取库文件的 i 结点
25.                     return -ENOENT;                //失败，则返回错误码
26.       } else                                       //如果指定库为空
27.                     inode = NULL;                  //则置 i 结点指针为 NULL
28.                     iput(current->library);        //释放当前进程的库文件
```

```
29.                     current->library = NULL;          //将当前进程库文件指针置空
30.                     base = get_base(current->ldt[2]); //取得当前进程的数据段段基址
31.                     base += LIBRARY_OFFSET;           //取得库文件的起始地址
32.                     free_page_tables(base,LIBRARY_SIZE);     //将库文件占用内存页释放
33.                     current->library = inode;         //将当前进程的库文件 i 结点指针指向指定库文
                                                          //件 i 结点
34.             return 0;                                 //返回 0
35.         }
        //在新用户堆栈中创建环境和参数变量指针表
36.         static unsigned long * create_tables(char * p,int argc,int envc)
37.             {
38.                 unsigned long *argv,*envp;
39.                 unsigned long * sp;
40.
41.                 sp = (unsigned long *) (0xfffffffc & (unsigned long) p);   //令 sp 为 4 的整数倍
42.                 sp -= envc+1;                             //令 sp 向下移动环境参数占用的空间个数
43.                 envp = sp;                                //令环境参数指针指向 sp 的位置
44.                 sp -= argc+1;                             //sp 向下移动命令行参数指针占用的空间个数
45.                 argv = sp;                                //令命令行参数指针指向 sp 的位置
46.                 put_fs_long((unsigned long)envp,--sp); //将环境参数指针压入堆栈
47.                 put_fs_long((unsigned long)argv,--sp); //将命令行参数指针压入栈中
48.                 put_fs_long((unsigned long)argc,--sp); //将命令行参数个数压入栈中
         //将命令行参数指针放入前面相应空出来的空间
49.             while (argc-->0) {
50.                 put_fs_long((unsigned long) p,argv++);
51.                 while (get_fs_byte(p++)) /* nothing */ ;
52.             }
53.                 put_fs_long(0,argv);                      //存放命令行参数指针的空间最后放置一个空指针
        //将环境参数指针放入前面相应空出来的空间
54.                 while (envc-->0) {
55.                 put_fs_long((unsigned long) p,envp++);
56.                 while (get_fs_byte(p++)) /* nothing */ ;
57.                 }
58.                 put_fs_long(0,envp);                      //存放环境参数指针的空间最后放置一个空指针
59.                 return sp;                                //返回当前新堆栈指针
60.         }
61.
62.             static int count(char ** argv)
63.             {
64.                 int i=0;
65.                 char ** tmp;
66.
67.                 if (tmp = argv)
68.                     while (get_fs_long((unsigned long *) (tmp++))) //计算参数指针数组中参数的个数
69.                     i++;
70.
71.                 return i;                                 //返回参数个数
72.             }
        //复制指定个数的参数字符串到参数和环境空间
73.             static unsigned long copy_strings(int argc,char ** argv,unsigned long *page,
```

```
74.             unsigned long p, int from_kmem)
75.     {
76.             char *tmp, *pag;
77.             int len, offset = 0;
78.             unsigned long old_fs, new_fs;
79.
80.             if (!p)
81.                     return 0;                       /*验证偏移指针*/
82.             new_fs = get_ds();                      //取 ds 寄存器的值
83.             old_fs = get_fs();                      //取 fs 寄存器的值
84.             if (from_kmem==2)                       //如果字符串和字符串来自内核空间
85.             set_fs(new_fs);                         //则令 fs 段寄存器指向内核数据段
86.       while (argc-- > 0) {                          //循环处理各个参数，从最后一个开始复制到指
                                                        //定偏移地址
87.                     if (from_kmem == 1)             //如果字符串在用户空间，同时字符串数组在内
                                                        //核空间
88.                     set_fs(new_fs);                 //则令 fs 段寄存器指向内核数据段
89.             if (!(tmp = (char *)get_fs_long(((unsigned long *)argv)+argc))) //取 fs 段最后一个参
90.                                                     //数指针
91.                     panic("argc is wrong");         //如果失败，则出错停机
92.             if (from_kmem == 1)                     //如果字符串在用户空间同时字符串数组在内核
                                                        //空间
93.                     set_fs(old_fs);                 //则恢复 fs 段寄存。
94.             len=0;
95.                                                     /*直到串以 0 结尾*/
96.                     do {                            //计算参数字符串长度，并使 tmp 指向该参数字
                                                        //符串末端
97.                             len++;
98.             } while (get_fs_byte(tmp++));
99.             if (p < len) {                          //如果该字符串长度超过此时参数和环
100.                                                    //境空间中还剩余的空闲长度
101.                set_fs(old_fs);                     //则恢复 fs 段寄存器，返回 0
102.                return 0;
103.                }
104.                while (len) {                       //逆向复制 fs 段中当前指定的参数字符串
105.                        --p; --tmp; --len;
106.                if (--offset < 0) {                 //如果 offset-1 小于 0，
107.                        offset = p % PAGE_SIZE;     //则令其等于 p 指针在页面内的偏移值
108.                                if (from_kmem==2)   //如果字符串和字符串数组在内核空间
109.                                set_fs(old_fs);     //则恢复 fs 段寄存器原值
110.              if (!(pag = (char *) page[p/PAGE_SIZE]) && //如果当前偏移值 p 所在的串空间
111.                        !(pag = (char *) page[p/PAGE_SIZE] =   //页面指针数组项等于 0，则申请
112.                                (unsigned long *) get_free_page()))//新的内存空闲页面，将该页面
113.                                                                //指针放入指针数组，同时使 pag
114.                                                                //指向该页面
115.                                return 0;                       //如果失败，则返回 0
116.                        if (from_kmem==2)                       //如果字符串和字符串数组来自内核空间
117.                                set_fs(new_fs);                 //则令 fs 段寄存器指向内核数据段
118.
119.                }
```

```
120.                    *(pag + offset) = get_fs_byte(tmp);      //从 fs 中复制参数字符串的一个字节到
121.                                                             //pag+offset
122.            }
123.        }
124.            if (from_kmem==2)                                //如果字符串和字符串数组在内核空间
125.                set_fs(old_fs);                              //则恢复 fs 段寄存器的原值
126.        return p;                                            //返回已复制参数信息的头部偏移值
127.    }
        //修改局部描述符表中的描述符基址和段限长
128.      static unsigned long change_ldt(unsigned long text_size,unsigned long * page)
129.    {
130.      unsigned long code_limit,data_limit,code_base,data_base;
131.            int i;
132.
133.            code_limit = TASK_SIZE;                          //代码段限长赋值
134.            data_limit = TASK_SIZE;                          //数据段限长赋值
135.            code_base = get_base(current->ldt[1]);           //获取当前进程中局部描述符表代码段描
                                                                 //述符的代码基址
136.
137.      data_base = code_base;                                 //代码段基址和数据段基址相同
138.      set_base(current->ldt[1],code_base);                   //重新设置局部表中代码段描述符的基址
139.      set_limit(current->ldt[1],code_limit);               //重新设置局部表中代码段描述符的段限长
140.      set_base(current->ldt[2],data_base);                   //重新设置局部表中数据段描述符的基址
141.      set_limit(current->ldt[2],data_limit);                 //重新设置局部表中数据段描述符的段限长
142.            __asm__("pushl $0x17\n\tpop %%fs"::);            //fs 段寄存器中放入局部表数据段描述符
                                                                 //的选择符
        //将参数和环境空间已存放数据的页面放到数据段线性地址的末端
143.      data_base += data_limit - LIBRARY_SIZE;
144.      for (i=MAX_ARG_PAGES-1 ; i>=0 ; i--) {
145.                  data_base -= PAGE_SIZE;
146.                  if (page[i])                               //如果该页面存在
147.                  put_dirty_page(page[i],data_base);         //则释放该页面
148.      }
149.            return data_limit;                               //返回数据段限长
150.       }
        //execve()系统中断调用函数
151.       int do_execve(unsigned long * eip,long tmp,char * filename,
152.      char ** argv, char ** envp)
153.     {
154.            struct inode * inode;
155.            struct buffer_head * bh;
156.            struct exec ex;
157.            unsigned long page[MAX_ARG_PAGES];
158.            int i,argc,envc;
159.            int e_uid, e_gid;
160.            int retval;
161.            int sh_bang = 0;
162.            unsigned long p=PAGE_SIZE*MAX_ARG_PAGES-4; //令参数和环境字符串空间的偏
                                                           //移指针为指向该空间的最后一个长字处
163.      if ((0xffff & eip[1]) != 0x000f)                     //如果 eip[1]中的选择符是内核段选择符
```

```
164.            panic("execve called from supervisor mode"); //则打印警告信息，停机
165.            for (i=0 ; i<MAX_ARG_PAGES ; i++)        //初始化参数和环境空间的页面指
166.            page[i]=0;                               //针表
167.        if (!(inode=namei(filename)))                //取可执行文件的 i 结点号
168.                return -ENOENT;                      //如果失败，则返回错误码
169.        argc = count(argv);                          //计算参数个数
170.        envc = count(envp);                          //计算环境变量个数
171.
172.         restart_interp:
173.            if (!S_ISREG(inode->i_mode)) {           //如果要执行的文件不是常规文件
174.                                                     //必须是常规文件
175.            retval = -EACCES;                        //置错误码
176.                goto exec_error2;                    //跳转到 exec_error2
177.        }
178.        i = inode->i_mode;                           //取文件属性字段值
179.            if (current->flags & PF_PTRACED) { //如果当前进程设置了 PF_PTRACED
180.            e_uid = current->euid; //则将要执行进程的有效用户 ID 设置为当前进程的有效用户
181.            e_gid = current->egid; // ID，将要执行进程的有效组 ID 设置为当前进程的有效组 ID
182.            } else {
183.                //如果文件的设置用户 ID 标志置位，则将要执行进程的有效用户 ID 就设置为
184.              //文件的用户 ID，否则设置为当前进程有效用户 ID
185.            e_uid = (i & S_ISUID) ? inode->i_uid : current->euid;
186.                //如果文件的设置组 ID 标志置位，则将要执行进程的有效组 ID 就设置为
187.              //文件的组 ID，否则设置为当前进程有效组 ID
188.            e_gid = (i & S_ISGID) ? inode->i_gid : current->egid;
189.            }
190.            if (current->euid == inode->i_uid)   //如果文件属于运行进程的用户
191.            i >>= 6;                                 //则使文件属性字最低 6 位是文件宿主的
                                                         //访问权限标志
192.        else if (in_group_p(inode->i_gid))           //如果文件与运行进程的用户属于同组
193.                i >>= 3;                             //则使属性字最低 3 位是文件组用户的访
                                                         //问权限标志
194.        if (!(i & 1) &&                              //如果用户没有执行权并且其他用户页没
                                                         //有任何权限，并且不是超级用户
195.            !((inode->i_mode & 0111) && suser())) {
196.                retval = -EACCES;                    //则错误码赋值
197.            goto exec_error2;                        //跳转到 exec_error2
198.        }
199.        if (!(bh = bread(inode->i_dev,inode->i_data[0]))) { //读取执行文件的第一块数据到高
                                                         //速缓冲区，如果失败，则错误码赋值
200.                retval = -EACCES;
201.            goto exec_error2;                        //跳转到 exec_error2
202.        }
203.            ex = *((struct exec *) bh->b_data);      //令 ex 指向执行头部分的数据结构
204.                                                     //读执行头部分
    //如果执行文件开始的两个字节为#，并且 sh_bang 标志没有置位，则进行脚本文件的处理
205.        if ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') && (!sh_bang)) {
206.            char buf[128], *cp, *interp, *i_name, *i_arg;
207.                unsigned long old_fs;
208.
```

```
209.                    strncpy(buf, bh->b_data+2, 127);  //赋值执行程序头一行字符#！后的字符串到 buf
210.                    brelse(bh);                       //中。释放高速缓冲区
211.                    iput(inode);                      //释放该执行文件 i 结点
        //取第一行内容，并删除开始的空格、制表符
212.                    buf[127] = '\0';
213.            if (cp = strchr(buf, '\n')) {
214.                    *cp = '\0';
215.                            for (cp = buf; (*cp == ' ') || (*cp == '\t'); cp++);
216.                    }
217.                    if (!cp || *cp == '\0') {         //如果该行没有其他内容
218.                    retval = -ENOEXEC;                //出错码赋值
219.                                                      //没有找到解释名
220.                            goto exec_error1;         //跳转到 exec_error1 执行
221.            }
222.            interp = i_name = cp;                     //否则获取开头是脚本解释执行程序名称的一行内容
223.                    i_arg = 0;
224.                    for ( ; *cp && (*cp != ' ') && (*cp != '\t'); cp++) { //取第一个字符串
225.                    if (*cp == '/')                   //如果是脚本解释程序名
226.                            i_name = cp+1;            // i_name 指向该名称
227.                    }
228.                    if (*cp) {                        //如果文件名后还有字符
229.                    *cp++ = '\0';
230.                    i_arg = cp;                       //令 i_arg 指向该串
231.                    }
232.                            if (sh_bang++ == 0) {     //如果没有设置 sh_bang 标志，则设置它
233.            p = copy_strings(envc, envp, page, p, 0); //复制指定个数的环境变量串到环境空间
234.            p = copy_strings(--argc, argv+1, page, p, 0);   //复制指定个数的参数量串到参数空间
235.
236.            }
        //复制脚本程序名到参数和环境空间中
237.            p = copy_strings(1, &filename, page, p, 1);
238.                    argc++;
239.                    if (i_arg) {
240.                    p = copy_strings(1, &i_arg, page, p, 2);
241.                    argc++;
242.                    }
243.                    p = copy_strings(1, &i_name, page, p, 2);      //复制解释程序文件名到参数和环境
244.                    argc++;                                        //空间中
245.            if (!p) {                                 //如果出错
246.                    retval = -ENOMEM;                 //错误码赋值
247.                            goto exec_error1;         //跳转到 exec_error1 执行
248.            }
249.                    old_fs = get_fs();                //存放原 fs 段寄存器值
250.                    set_fs(get_ds());                 //令其指向内核数据段
251.                    if (!(inode=namei(interp))) {     //取解释程序 i 结点
252.                                                      /*得到可执行文件结点*/
253.                    set_fs(old_fs);                   //如果失败，则恢复 fs 段寄存器原值
254.                            retval = -ENOENT;         //错误码赋值
255.                    goto exec_error1;                 //跳转到 goto exec_error1
256.            }
```

```
257.                set_fs(old_fs);                    //恢复 fs 段寄存器原值
258.                goto restart_interp;               //跳转到 restart_interp
259.        }
260.        brelse(bh);                                //释放该缓冲区
261.            if (N_MAGIC(ex) != ZMAGIC || ex.a_trsize || ex.a_drsize ||
262.            ex.a_text+ex.a_data+ex.a_bss>0x3000000 ||
263.            inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N_TXTOFF(ex)) {
264.                retval = -ENOEXEC;                 //则错误码赋值
265.            goto exec_error2;                      //跳转到 exec_error2
266.        }
267.            if (N_TXTOFF(ex) != BLOCK_SIZE) {//如果执行文件执行头部分长度不等于一个
                                                     //内存块大小
268.                printk("%s: N_TXTOFF != BLOCK_SIZE. See a.out.h.", filename); //打印警告信息
269.            retval = -ENOEXEC;                     //错误码赋值
270.            goto exec_error2;                      //跳转到 exec_error2
271.            }
272.            if (!sh_bang) {                        //如果没有设置 sh_bang
273.            p = copy_strings(envc,envp,page,p,0);  //则复制指定个数的环境变量字符串到环境空
274.            p = copy_strings(argc,argv,page,p,0);  //间。复制指定个数的参数字符串到参数空间
275.                if (!p) {                          //如果 p 等于 0
276.                retval = -ENOMEM;                  //则错误码赋值
277.                goto exec_error2;                  //跳转到 exec_error2
278.                }
279.            }
280.            if (current->executable)               //如果原程序是一个执行程序
281.                iput(current->executable);         //则释放其 i 结点
282.            current->executable = inode;           //并令进程的 executable 字段指向新程序的 i 结点
283.            current->signal = 0;                   //清空进程的 signal 字段
284.        for (i=0 ; i<32 ; i++) {                   //复位所有信号处理句柄
285.            current->sigaction[i].sa_mask = 0;     //复位信号处理掩码
286.            current->sigaction[i].sa_flags = 0;    //复位信号处理标志
287.            if (current->sigaction[i].sa_handler != SIG_IGN)    //如果不是 SIG_IGN 句柄
288.            current->sigaction[i].sa_handler = NULL;            //则复位该句柄
289.            }
290.            for (i=0 ; i<NR_OPEN ; i++)
291.            if ((current->close_on_exec>>i)&1)     //根据执行时关闭文件句柄位图标志
292.                sys_close(i);                      //关闭指定的打开文件
293.            current->close_on_exec = 0;            //复位该标志
294.
295.        free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));    //释放原来进程代码段所对
296.                                                   //应的内存页表指定的内存块和页表
297.        free_page_tables(get_base(current->ldt[2]),get_limit(0x17));    //释放原来进程数据段所
298.                                                   //对应的内存页表指定的内存块和页表
299.            if (last_task_used_math == current)    //如果 last_task_used_math 指向的是当前
                                                       //进程
300.            last_task_used_math = NULL;            //则将 last_task_used_math 置空
301.            current->used_math = 0;                //复位协处理器标志
302.            p += change_ldt(ex.a_text,page);       //修改局部表中描述符基址和段限长
303.        p -= LIBRARY_SIZE + MAX_ARG_PAGES*PAGE_SIZE; //将参数和环境空间页面放置
304.                                                   //在数据段末端
```

```
305.            p = (unsigned long) create_tables((char *)p,argc,envc);  //在新用户堆栈中创建环境和参
306.                                                                      //数变量指针表，并获取该堆栈指针
307.        current->brk = ex.a_bss +                                     //令进程堆结尾字段等于 a_text+a_data+a_bss
308.                    (current->end_data = ex.a_data +                  //令进程数据段尾字段等于 a_data + a_text
309.            (current->end_code = ex.a_text));                         //令进程代码段尾字段等于 a_text
310.        current->start_stack = p;                                     //设置进程堆栈开始字段为堆栈指针所在
                                                                          //的页面
311.            current->suid = current->euid = e_uid;                    //重新设置进程的有效用户 id。
312.        current->sgid = current->egid = e_gid;                        //重新设置进程的有效用户 id。
313.        eip[0] = ex.a_entry;                                          /*eip，魔法发生了*/
314.            eip[3] = p;                                               /*栈指针*/
315.        if (current->flags & PF_PTRACED)                              //如果当前进程设置了 PF_PTRACED 标志
316.          send_sig(SIGTRAP, current, 0);                              //发送给当前进程 SIGTRAP 信号
317.            return 0;                                                 //返回 0
318.            exec_error2:
319.            iput(inode);                                              //释放该 i 结点
320.            exec_error1:
321.            for (i=0 ; i<MAX_ARG_PAGES ; i++)                         //释放所有页面
322.            free_page(page[i]);
323.        return(retval);                                               //返回错误码
324.         }
```


第 15 行：MAX_ARG_PAGES 定义了分配给新程序的参数和环境变量使用的内存页数。32 页应该足够了，这使得环境和参数空闲总和达到 128kB！

第 36～60 行：create_tables()在新用户的内存中解析了环境和参数字符串，并且根据它们创建指针表，把它们的地址放到“堆栈”中，返回新堆栈的指针值。

第 62～72 行：count()计算命令行参数/环境参数的个数。

第 73～127 行：copy_string()函数从用户内存复制参数和环境字符串到内核内存的空闲页面中。这些已具有直接放到新用户内存中的格式。后来由 TYT(Tytso)修改，增加了 from_kmem 参数，该参数指明了字符串或字符串数组是来自用户段还是内核段。我们是通过巧妙处理 fs 段寄存器来操作的。由于加载一个段寄存器代价太大，所以尽量避免调用 set_fs()，除非实在必要。

第 151～236 行：do_execve()执行一个新程序。

8.14 stat.c 文件

该文件实现了获取文件状态信息的系统调用函数 state()和 fstat()。本程序主要实现了对文件状态的操作，主要为读取文件状态信息。函数 cp_stat()复制指定文件的状态信息；函数 sys_stat()利用文件名读取信息，并将信息存放在用户的文件状态结构缓冲区；函数 sys_fstat()使用文件句柄来取信息，并将信息存放在用户的文件状态结构缓冲区。

stat.c 文件的详细代码注释如下：

```
1.      #include <errno.h>                          //错误号头文件。包含系统中各种出错号
2.      #include <sys/stat.h>                       //文件状态头文件。主要说明了函数 stat()返回的数据
                                                    //及其结构类型
                //一些属性操作测试宏、函数原型
3.      #include <linux/fs.h>                       //文件系统头文件。定了文件表结构
4.      #include <linux/sched.h>                    //调度程序头文件。定义了 task_struct 和任务 0 的数据
            //一些关于描述符参数设置和获取以及 switch_to()的嵌入式汇编函数宏
```

```
5.      #include <linux/kernel.h>                    //内核头文件。定义了一些内核常用函数的原形
6.      #include <asm/segment.h>                     //段操作头文件。定义了有关段寄存器操作的嵌入式
                                                     //汇编函数
      //赋值指定文件的状态信息
7.        static void cp_stat(struct inode * inode, struct stat * statbuf)
8.      {
9.        struct stat tmp;
10.             int i;
11.
12.       verify_area(statbuf,sizeof (struct stat)); //验证存放数据的内存空间
13.       tmp.st_dev = inode->i_dev;                 //临时复制指定文件所在的设备号
14.       tmp.st_ino = inode->i_ino;                 //临时复制指定文件的 i 结点号
15.       tmp.st_mode = inode->i_mode;               //临时复制指定文件的属性
16.       tmp.st_nlink = inode->i_nlink;             //临时复制指定文件的连接数
17.       tmp.st_uid = inode->i_uid;                 //临时复制指定文件的用户 ID
18.       tmp.st_gid = inode->i_gid;                 //临时复制指定文件的组 ID
19.       tmp.st_rdev = inode->i_rdev;               //临时复制指定文件挂载的设备号
20.       tmp.st_size = inode->i_size;               //临时复制指定文件的大小
21.       tmp.st_atime = inode->i_atime;             //临时复制指定文件的最后访问时间
22.       tmp.st_mtime = inode->i_mtime;             //临时复制指定文件最后修改时间
23.       tmp.st_ctime = inode->i_ctime;             //临时复制指定文件最后结点修改时间
24.             for (i=0 ; i<sizeof (tmp) ; i++)     //将临时信息复制到用户缓冲区中
25.             put_fs_byte(((char *) &tmp)[i],i + (char *) statbuf);
26.     }
      //获取指定文件名的文件状态信息
27.       int sys_stat(char * filename, struct stat * statbuf)
28.       {
29.             struct inode * inode;
30.
31.       if (!(inode=namei(filename)))                //获取指定文件名的 i 结点
32.             return -ENOENT;                      //失败，则返回错误码
33.       cp_stat(inode,statbuf);                    //将该 i 结点的文件状态信息复制到用户缓冲区
34.       iput(inode);                               //释放该 i 结点，返回 0
35.             return 0;
36.       }
    //获取指定符号连接文件名的目标文件的状态信息
37.     int sys_lstat(char * filename, struct stat * statbuf)
38.     {
39.             struct inode * inode;
40.
41.             if (!(inode = lnamei(filename)))     //获取指定符号连接文件的目标文件 i 结点
42.             return -ENOENT;                      //失败，则返回错误码
43.             cp_stat(inode,statbuf);              //将该 i 结点的文件状态信息复制到用户缓冲区
44.             iput(inode);                         //释放该 i 结点，返回 0
45.             return 0;
46.     }
    //获取指定文件句柄的文件状态信息
47.       int sys_fstat(unsigned int fd, struct stat * statbuf)
48.       {
49.             struct file * f;
```

```
50.             struct inode * inode;
51.             if (fd >= NR_OPEN || !(f=current->filp[fd]) || !(inode=f->f_inode)) //如果文件句柄大于一个
52.         //程序所能打开的最大文件数，或者句柄对应的文件结构指针为空，或
53.         //者该文件结构的文件 i 结点字段为空
54.                 return -EBADF;                          //则返回错误码
55.                 cp_stat(inode,statbuf);                 //将该 i 结点的文件状态信息复制到用户缓冲区
56.                 return 0;                               //返回 0
57.         }
        //读取指定符号连接文件的目标文件名
58.         int sys_readlink(const char * path, char * buf, int bufsiz)
59.           {
60.                 struct inode * inode;
61.                 if (bufsiz <= 0)                        //如果缓冲区大小小于等于 0
62.                         return -EINVAL;                 //则返回错误码
63.                 verify_area(buf,bufsiz);                //验证存放文件名的内存空间
64.                 if (!(inode = lnamei(path)))            //获取指定路径的文件 i 结点
65.                 return -ENOENT;                         //如果失败，则返回错误码
66.                 if (!inode->i_op || !inode->i_op->readlink) {//如果该 i 结点的 i_op 指针为空，或者读取符
67.                                                         //号连接的函数不存在
68.                 iput(inode);                            //则释放该 i 结点
69.                         return -EINVAL;                 //返回错误码
70.           }
71.           return inode->i_op->readlink(inode,buf,bufsiz);//调用相应读取符号连接目标文件名的函数
72.           }
```


8.15 fcntl.c 文件

该文件实现了文件控制系统调用和两个文件句柄赋值的系统调用定义。

本程序是对目录和文件进行操作的上层处理程序，实现了文件控制系统调用和两个文件句柄复制的系统调用。函数 dupfd()复制文件句柄，该函数被两个文件句柄复制系统调用所使用；函数 sys_dup2()指定了新句柄的数值来复制一个句柄；函数 sys_dup()复制文件句柄，新句柄的值是当前最小未使用的句柄；函数 sys_fcntl()是文件控制系统调用函数。

fcntl.c 文件的详细代码注释如下：

```
1.      #include <string.h>             //字符串头文件。定义了所有字符串操作函数
2.      #include <errno.h>              //错误号头文件。包含系统中各种出错号
3.      #include <linux/sched.h>        //调度程序头文件。定义了 task_struct 和任务 0 的数据。一
                                        //些关于描述符参数设置和获取以及 switch_to()的嵌入式汇
                                        //编函数宏
4.      #include <linux/kernel.h>       //内核头文件。定义了一些内核常用函数的原形
5.      #include <asm/segment.h>        //段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数
6.      #include <fcntl.h>              //文件控制头文件。用于文件及其描述符的操作控制常数符
                                        //号的定义
7.      #include <sys/stat.h>           //文件状态头文件。主要说明了函数 stat()返回的数据及其
                                        //结构类型，一些属性操作测试宏、函数原型
      //关闭文件系统调用函数
8.      extern int sys_close(int fd);
      //复制文件句柄
9.      static int dupfd(unsigned int fd, unsigned int arg)
```

```
10.         {
11.             if (fd >= NR_OPEN || !current->filp[fd])    //如果文件句柄大于一个程序最多能打开的文
12.                                                         //件数，或者该句柄对应的文件结构指针为空
13.             return -EBADF;                              //则返回错误码
14.             if (arg >= NR_OPEN)                         //如果指定的新句柄最小值大于一个程序最多能
15.                                                         //打开的文件数
16.             return -EINVAL;                             //则返回错误码
17.             while (arg < NR_OPEN)                       //在当前进程的文件结构指针表内寻找大于 arg 的空闲项
18.             if (current->filp[arg])                     //如果该项不为空
19.                 arg++;                                  //则继续检查下一项
20.                 else                                    //否则，跳出循环
21.                     break;
22.         if (arg >= NR_OPEN)                             //如果 arg 大于一个程序最多能打开的文件数
23.             return -EMFILE;                             //返回错误码
24.             current->close_on_exec &= ~(1<<arg);        //在执行时关闭标志位图中复位该句柄位
25.         (current->filp[arg] = current->filp[fd])->f_count++; //复制原句柄的文件结构指针到该句柄
26.                                                         //的文件结构指针，并将文件引用数加 1
27.             return arg;                                 //返回新的文件句柄
28.     }
    //复制文件句柄系统调用函数
29.     int sys_dup2(unsigned int oldfd, unsigned int newfd)
30.         {
31.             if (newfd == oldfd)                         //如果新旧句柄相同
32.             return newfd;                               //则返回新句柄
33.             sys_close(newfd);                           //若新句柄已经打开，则关闭它
34.             return dupfd(oldfd,newfd);                  //复制文件句柄，返回新句柄
35.         }
    //复制文件句柄系统调用函数
36.     int sys_dup(unsigned int fildes)
37.     {
38.             return dupfd(fildes,0);                     //复制文件句柄，新句柄的值是当前最小未使用的句
                                                            //柄。返回新句柄
39.         }
    //文件控制系统调用函数
40.     int sys_fcntl(unsigned int fd, unsigned int cmd, unsigned long arg)
41.     {
42.             struct file * filp;
43.
44.             if (fd >= NR_OPEN || !(filp = current->filp[fd])) //如果文件句柄大于一个程序最多能打
                                                            //开的文件数，或者该句柄对应的文件结构指针为空
45.
46.             return -EBADF;                              //则返回错误码
47.             switch (cmd) {                              //根据不同的命令参数选择处理函数
48.             case F_DUPFD:                               //复制文件句柄
49.                 return dupfd(fd,arg);
50.             case F_GETFD:                               //获取文件句柄的执行时关闭标志
51.                 return (current->close_on_exec>>fd)&1;
52.             case F_SETFD:                               //设置句柄执行时关闭标志
```

```
53.                     if (arg&1)
54.                             current->close_on_exec |= (1<<fd);
55.                     else
56.                             current->close_on_exec &= ~(1<<fd);
57.                     return 0;
58.             case F_GETFL:                           //获取文件状态标志和访问模式
59.                     return filp->f_flags;
60.             case F_SETFL:                           //设置文件状态和访问模式
61.                     filp->f_flags &= ~(O_APPEND | O_NONBLOCK);
62.                     filp->f_flags |= arg & (O_APPEND | O_NONBLOCK);
63.                     return 0;
64.             case F_GETLK: case F_SETLK:  case F_SETLKW: //未实现，返回错误码
65.                     return -ENOSYS;
66.             default:                                //不是以上任何命令，则返回错误码
67.                     return -EINVAL;
68.             }
69.     }
```

8.16 ioctl.c 文件

该文件实现了输入输出控制系统调用的定义。通过该系统调用可以控制终端设备的输入和输出操作。函数 sys_ioctl()为系统输入输出控制系统调用函数，该函数主要是调用函数 tty_ioctl()对终端的 IO 进行控制。

ioctl.c 文件的详细代码注释如下：

```
1.      #include <string.h>                            //字符串头文件。定义了所有字符串操作函数
2.      #include <errno.h>                             //错误号头文件。包含系统中各种出错号
3.      #include <sys/stat.h>                          //文件状态头文件。主要说明了函数 stat()返回的数据
                                                       //及其结构类型一些属性操作测试宏、函数原型
4.      #include <linux/sched.h>                       //调度程序头文件。定义了 task_struct 和任务 0
                                                       //的数据。一些关于描述符参数设置和获取以及
                                                       //switch_to()的嵌入式汇编函数宏
5.      extern int tty_ioctl(int dev, int cmd, int arg);        //终端输入输出控制函数
6.      extern int pipe_ioctl(struct inode *pino, int cmd, int arg);       //管道输入输出控制函数
7.
8.      typedef int (*ioctl_ptr)(int dev,int cmd,int arg);      //定义输入输出控制函数指针
9.
10.     #define NRDEVS ((sizeof (ioctl_table))/(sizeof (ioctl_ptr)))    //定义系统中设备的种数
        //ioctl 操作函数指针表
11.     static ioctl_ptr ioctl_table[]={
12.             NULL,
13.             NULL,
14.             NULL,
15.             NULL,
16.             tty_ioctl,
17.             tty_ioctl,
18.             NULL,
19.             NULL};
        //系统输入输出控制系统调用函数
```

```
20.     int sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
21. {
22.     struct file * filp;
23.     int dev,mode;
24.
25.     if (fd >= NR_OPEN || !(filp = current->filp[fd])) //如果文件句柄超过一个进程最多打开的文
26.                                                       //件数，或者该句柄对应的文件结构指针不存在
27.         return -EBADF;                                //则返回错误码
28.     if (filp->f_inode->i_pipe)                        //如果该文件是一个管道文件
29.         return (filp->f_mode&1)?pipe_ioctl(filp->f_inode,cmd,arg):-EBADF; //如果该文件
30.                     //可读，则调用管道输入输出控制函数并返回结果，否则返回错误码
31.
32.     mode=filp->f_inode->i_mode;                       //取对应文件的属性
33.     if (!S_ISCHR(mode) && !S_ISBLK(mode))             //如果该文件不是字符文件，并且不是块文件
34.         return -EINVAL;                               //则返回错误码
35.     dev = filp->f_inode->i_rdev;                      //取对应文件所挂载的设备号
36.     if (MAJOR(dev) >= NRDEVS)                         //如果设备号大于系统现有的设备数
37.         return -ENODEV;                               //则返回错误码
38.     if (!ioctl_table[MAJOR(dev)])                         //获取该设备对应的函数指针
39.         return -ENOTTY;                                   //失败，则返回错误码
40.     return ioctl_table[MAJOR(dev)](dev,cmd,arg);          //调用相应的函数，并返回结果
41. }
```


8.17 select.c 文件

本文件实现了 Linux 的选择处理功能，其中主要实现了系统调用 select 函数的定义。

该系统调用是用来让我们的程序监视多个文件句柄(file descriptor)的状态变化的。程序会停在 select 这里等待，直到被监视的文件句柄有某一个或多个发生了状态改变。

函数 add_wait()将等待地址加入到选择表之中；

函数 free_wait()用于释放选择表；

函数 get_tty()用于获取指定文件 i 结点所对应的字符设备结构；

函数 check_in()用于检查可读性文件 i 结点；

函数 check_out()用于检查可写性文件 i 结点；

函数 check_ex()用于检查意外状态文件 i 结点；

函数 do_select()完成了系统调用 select 的选择处理部分；

函数 sys_select()是选择处理系统调用函数。

select.c 文件的详细代码注释如下：

```
1.  #include <linux/fs.h>          //文件系统头文件。定义了有关文件系统的一些常数和结构
2.  #include <linux/kernel.h>      //内核头文件。定义了一些内核常用函数的原型
3.  #include <linux/tty.h>         //tty 头文件，定义了 tty_io.c 程序使用的某些结构和其他一些定义
4.  #include <linux/sched.h>       //调度程序头文件。定义了 task_struct 和任务 0 的数据。一些关
                                   //于描述符参数设置和获取以及 switch_to()的嵌入式汇编函数宏
5.  #include <asm/segment.h>       //段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数
6.  #include <asm/system.h>        //系统头文件。定义了设置或修改描述符、中断门等的嵌入式汇编宏
7.  #include <sys/stat.h>          //文件状态头文件。主要说明了函数 stat()返回的数据及其结构类型
                                   //一些属性操作测试宏、函数原型
8.  #include <sys/types.h>         //类型头文件。定义了基本的数据类型
```

```
9.        #include <string.h>          //字符串头文件。定义了所有字符串操作函数
10.         #include <const.h>         //常数符号头文件。定义了 i 结点中 i_mode 字段的各标志位
11.       #include <errno.h>           //错误号头文件。包含系统中各种出错号
12.       #include <sys/time.h>        //系统时间头文件。该头文件中主要定义了文件访问与修改时间
                                       //结构
13.         #include <signal.h>        //信号头文件。定义了内核中使用的所有信号的名称和基本操作
                                       //函数
      //定义选择表中数组元素类型
14.       typedef struct {
15.         struct task_struct * old_task;
16.         struct task_struct ** wait_address;
17.         } wait_entry;
      //定义选择表
18.       typedef struct {
19.         int nr;
20.         wait_entry entry[NR_OPEN*3];
21.         } select_table;
      //将指定的等待地址加入到指定的选择表
22.         static void add_wait(struct task_struct ** wait_address, select_table * p)
23.       {
24.         int i;
25.
26.               if (!wait_address)                    //如果指定等待地址为空
27.                     return;                         //则返回
28.               for (i = 0 ; i < p->nr ; i++)         //遍历选择表检查指定等待地址是否已经存在
29.               if (p->entry[i].wait_address == wait_address)  //如果已存在，则返回
30.                     return;
      //如果指定的等待队列在选择表中不存在，则在选择表尾申请新的一项
31.               p->entry[p->nr].wait_address = wait_address; //将该项的等待地址字段置为指定等待
                                                       //地址
32.               p->entry[p->nr].old_task = * wait_address; //将该项的原任务字段置为指定等待地址所
33.                                                    //指向的进程指针
34.               *wait_address = current;             //将指定的等待地址指向当前进程的指针
35.               p->nr++;                             //选择表项目加 1
36.       }
      //释放指定选择表
37.         static void free_wait(select_table * p)
38.         {
39.               int i;
40.               struct task_struct ** tpp;
41.
42.               for (i = 0; i < p->nr ; i++) {       //遍历选择表
43.                     tpp = p->entry[i].wait_address;  //获取选择表中该项的等待地址
44.               while (*tpp && *tpp != current) {    //如果等待地址指向的进程指针不为空，并且
                                                       //该指针不指向当前进程
45.
46.                           (*tpp)->state = 0;       //则将该进程的状态字段置为 0.
47.                           current->state = TASK_UNINTERRUPTIBLE; //将当前进程的状态字段置为
48.                                                    //不可中断状态
49.                           schedule();              //重新调度
```

```
50.             }
51.             if (!*tpp)                                  //如果等待地址指向的进程指针如果为空
52.                     printk("free_wait: NULL");          //则打印警告信息
53.             if (*tpp = p->entry[i].old_task)            //将该项的原任务指针赋给等待地址指向
54.                                                         //的进程指针
55.                     (**tpp).state = 0;                  //如果失败，则将等待地址指向的进程指针对应
                                                            //的进程状态置为 0
56.
57.         }
58.             p->nr = 0;                                  //最后，将选择表的项数置为 0
59.         }
        //获取指定文件 i 结点所对应的字符设备结构
60.         static struct tty_struct * get_tty(struct inode * inode)
61.     {
62.         int major, minor;
63.
64.             if (!S_ISCHR(inode->i_mode))                //如果指定的 i 结点不是字符设备文件 i 结点
65.                     return NULL;                        //返回 NULL
66.         if ((major = MAJOR(inode->i_rdev)) != 5 && major != 4) //如果该 i 结点挂载的主设备号不
67.                                                         //为 5 或 4，则返回 NULL
68.                     return NULL;
69.         if (major == 5)                                 //如果该 i 结点对应设备是控制终端设备
70.             minor = current->tty;                       //则取当前进程的控制终端号为次设备号
71.          else                                           //否则，取指定 i 结点对应设备的次设备号
72.             minor = MINOR(inode->i_rdev);
73.         if (minor < 0)                                  //如果次设备号小于 0
74.                     return NULL;                        //返回 NULL
75.         return TTY_TABLE(minor);                        //根据次设备号获取设备结构并返回
76.         }
        //检查指定的可读性文件 i 结点
77.          static int check_in(select_table * wait, struct inode * inode)
78.          {
79.              struct tty_struct * tty;
80.
81.         if (tty = get_tty(inode))                       //获取指定 i 结点对应的字符设备结构
82.             if (!EMPTY(tty->secondary))                 //如果该设备的辅助队列为空
83.             return 1;                                   //返回 1
84.         else                                            //否则，将设备的读等待队列地址放入指定
85.             add_wait(&tty->secondary->proc_list, wait);         //的选择表
86.             else if (inode->i_pipe)                             //如果该设备是一个管道
87.             if (!PIPE_EMPTY(*inode) || inode->i_count < 2)      //则如果该管道为空或者引用计数
88.                     return 1;                                   //小于 2，则返回 1
89.             else
90.                 add_wait(&inode->i_wait, wait);         //否则，将该管道的等待队列地址放入指定
91.                                                         //的选择表
92.         return 0;                                       //返回 0
93.         }
        //检查指定的可写性文件 i 结点
94.          static int check_out(select_table * wait, struct inode * inode)
95.          {
```

```
96.             struct tty_struct * tty;
97.
98.             if (tty = get_tty(inode))              //获取指定 i 结点对应的字符设备结构
99.                 if (!FULL(tty->write_q))           //如果该设备的写队列不满，则返回 1
100.                return 1;
101.            else                                   //否则，将设备的写等待队列地址放入指定的选择表
102.                add_wait(&tty->write_q->proc_list, wait);
103.            else if (inode->i_pipe)                //如果该设备是一个管道
104.            if (!PIPE_FULL(*inode))                //则如果该管道不为满，则返回 1
105.                    return 1;
106.                else                               //否则，将该管道的等待队列地址放入指定的选择表
107.                add_wait(&inode->i_wait, wait);
108.      return 0;                                    //返回 0
109.       }
      //检查指定的意外状态文件 i 结点
110.       static int check_ex(select_table * wait, struct inode * inode)
111.    {
112.      struct tty_struct * tty;
113.
114.            if (tty = get_tty(inode))              //获取指定 i 结点对应的字符设备结构
115.                if (!FULL(tty->write_q))           //如果该设备的写队列不满，则返回 0
116.                return 0;
117.            else                                   //否则，返回 0
118.                    return 0;
119.            else if (inode->i_pipe)                //如果该设备是一个管道
120.            if (inode->i_count < 2)                //则如果该管道引用计数小于 2
121.                return 1;                          //则返回 1
122.            else                                   //否则，将该管道的等待队列地址放入指定的选择表
123.                    add_wait(&inode->i_wait,wait);
124.      return 0;                                    //返回 0
125.    }
      //选择函数
126.       int do_select(fd_set in, fd_set out, fd_set ex,
127.            fd_set *inp, fd_set *outp, fd_set *exp)
128.    {
129.      int count;
130.            select_table wait_table;
131.            int i;
132.            fd_set mask;
133.
134.            mask = in | out | ex;                  //获取文件句柄掩码
135.      for (i = 0 ; i < NR_OPEN ; i++,mask >>= 1) { //遍历该掩码对应的文件句柄，检查是否符合
136.            if (!(mask & 1))                       //要求。如果不是该掩码所对应的文件句柄，则继续
137.                    continue;
138.                if (!current->filp[i])             //如果该文件句柄对应的文件结构指针为空
139.                return -EBADF;                     //则返回错误码
140.            if (!current->filp[i]->f_inode)        //如果该文件句柄对应的文件 i 结点不存在
141.                    return -EBADF;                 //则返回错误码
142.            if (current->filp[i]->f_inode->i_pipe)     //如果该文件句柄对应的文件为管道文件
143.                continue;                          //则继续
```

```
144.            if (S_ISCHR(current->filp[i]->f_inode->i_mode))      //如果该文件句柄对应的文件为字
145.                                                    //符设备文件
146.                continue;                           //则继续
147.            if (S_ISFIFO(current->filp[i]->f_inode->i_mode))  //如果该文件句柄对应的文件为
148.                                                    //有名管道
149.                continue;                           //则继续
150.                return -EBADF;                      //如果以上都不是，则返回错误码
151.        }
152.    repeat:
153.            wait_table.nr = 0;
154.            *inp = *outp = *exp = 0;
155.            count = 0;
156.            mask = 1;
157.            for (i = 0 ; i < NR_OPEN ; i++, mask += mask) {                  //遍历进程的文件句柄
158.            if (mask & in)                          //如果该句柄对应的是可读性文件
159.                if (check_in(&wait_table,current->filp[i]->f_inode)) {     //则调用可读性文件检
                                                        //查函数
160.                    *inp |= mask;                   //如果检查通过，在指定的可读性文件句柄集中加
                                                        //入该句柄
161.                    count++;                                    //选择表项计数加 1
162.                }
163.                if (mask & out)                                 //如果该句柄对应的是可写性文件
164.                    if (check_out(&wait_table,current->filp[i]->f_inode)) {   //则调用可写性文件
                                                                                  //检查函数
165.                    *outp |= mask;                              //如果检查通过，在指定的可读性文件句柄
                                                                    //集中加入该
166.                    count++;                                    //句柄。选择表项计数加 1
167.                    }
168.                if (mask & ex)                                  //如果该句柄对应的是意外状态文件
169.                if (check_ex(&wait_table,current->filp[i]->f_inode)) {   //则调用意外状态文件
170.                                                                //检查函数
171.                    *exp |= mask;     //如果检查通过，在指定的意外状态文件句柄集中加入该
172.                    count++;                                    //句柄。选择表项计数加 1
173.                }
174.            }
175.            if (!(current->signal & ~current->blocked) &&  //如果当前进程没有信号要处理，并且仍
176.                current->timeout && !count) {              //有运行时间，并且选择表项计数为 0
177.                current->state = TASK_INTERRUPTIBLE;   //将当前进程运行状态置为不可中断
178.                schedule();                                 //重新调度
179.                free_wait(&wait_table);                     //释放指定选择表
180.                goto repeat;                                //跳转到 repeat
181.        }
182.            free_wait(&wait_table);                         //释放指定的选择表
183.            return count;                                   //返回选择表项计数
184.    }
        //选择系统调用函数
185.    int sys_select( unsigned long *buffer )
186.        {
        //运行 select(nd, in, out, ex, tv)系统调用
187.        int i;
```

```
188.        fd_set res_in, in = 0, *inp;
189.        fd_set res_out, out = 0, *outp;
190.        fd_set res_ex, ex = 0, *exp;
191.        fd_set mask;
192.        struct timeval *tvp;
193.        unsigned long timeout;
194.
195.        mask = get_fs_long(buffer++);                    //获取文件句柄掩码
196.        if (mask >= 32)                                  //如果该掩码大于等于 32
197.                mask = ~0;                               //则该掩码置为全 1
198.        else                                             //否则，该掩码置为全 1 左移 mask 个位再取反
199.        mask = ~((~0) << mask);
200.        inp = (fd_set *) get_fs_long(buffer++);          //获取可读性文件句柄集指针
201.        outp = (fd_set *) get_fs_long(buffer++);         //获取可写性文件句柄集指针
202.        exp = (fd_set *) get_fs_long(buffer++);          //获取意外状态文件句柄集指针
203.        tvp = (struct timeval *) get_fs_long(buffer);    //获取执行时间指针
204.
205.                if (inp)                                 //如果可读性文件句柄集指针不为空
206.                        in = mask & get_fs_long(inp);    //则取可读性文件句柄集
207.                if (outp)                                //如果可写性文件句柄集指针不为空
208.                        out = mask & get_fs_long(outp);  //则取可写性文件句柄集
209.                if (exp)                                 //如果意外状态文件句柄集指针不为空
210.                        ex = mask & get_fs_long(exp);    //则取意外状态文件句柄集
211.                timeout = 0xffffffff;                    //初始化执行时间
212.                if (tvp) {                               //如果执行时间指针不为空，则取执行时间
213.                        timeout = get_fs_long((unsigned long *)&tvp->tv_usec)/(1000000/HZ);
214.                        timeout += get_fs_long((unsigned long *)&tvp->tv_sec) * HZ;
215.                        timeout += jiffies;
216.                }
217.                current->timeout = timeout;              //将当前进程的执行时间置为该执行时间
218.        cli();                                           //开中断
219.        i = do_select(in, out, ex, &res_in, &res_out, &res_ex);    //调用 do_select 函数，取选择表
220.        if (current->timeout > jiffies)                  //项数。如果当前进程执行时间大于 jiffies
221.                timeout = current->timeout - jiffies;                    //则将执行时间减 jiffies
222.        else                                             //否则，执行时间置为 0
223.                        timeout = 0;
224.        sti();                                           //开中断
225.        current->timeout = 0;                            //置当前进程的执行时间为 0
226.        if (i < 0)                                       //如果选择表项数小于 0，返回错误码
227.                return i;
228.                if (inp) {                               //如果可读性文件句柄集指针不为空
229.                        verify_area(inp, 4);             //验证该文件句柄集的内存空间
230.                        put_fs_long(res_in,inp);         //将 do_select 返回的可读性文件句柄集复制到该文件
231.                }                                        //句柄集
232.        if (outp) {                                      //如果可写性文件句柄集指针不为空
233.                verify_area(outp,4);                     //验证该文件句柄集的内存空间
234.                put_fs_long(res_out,outp);   //将 do_select 返回的可写性文件句柄集复制到该文件
235.        }                                                //句柄集
236.        if (exp) {                                       //如果意外状态文件句柄集指针不为空
237.                        verify_area(exp,4);              //验证该文件句柄集的内存空间
```

```
238.                    put_fs_long(res_ex,exp);      //将 do_select 返回的意外状态文件句柄集复制到该文
                                                      //件句柄集
239.        }
240.        if (tvp) {                                //如果执行时间指针不为空
241.                    verify_area(tvp, sizeof(*tvp));      //验证该时间结构的内存空间
        //将执行时间赋给该时间结构
242.                    put_fs_long(timeout/HZ, (unsigned long *) &tvp->tv_sec);
243.                    timeout %= HZ;
244.                    timeout *= (1000000/HZ);
245.                    put_fs_long(timeout, (unsigned long *) &tvp->tv_usec);
246.        }
247.        if (!i && (current->signal & ~current->blocked))    //如果选择表项为 0，并且当前进程有信
248.                                                  //号要处理
249.            return -EINTR;                        //返回错误码
250.            return i;                             //返回选择表项数
251.         }
```

必须在选择过程中屏蔽中断，但那并不是真的这样的损失：当不在这个任务中时，睡眠自动释放中断。第 77～125 行：这些 check_XX 函数检查一个文件。我们知道这个文件置可能是一个管道，一个字符设备或者一个有名管道（有名管道还未实现）。

8.18 Minix 的 bitmap.c 文件

本程序主要实现了对 Minix 文件系统 i 结点位图和逻辑位图的管理功能，其中包括对 i 结点位图的释放及占用和对逻辑块位图的释放及占用。

1. 功能描述

函数 minix_free_block()用于释放指定设备上的逻辑块。首先根据指定设备的超级块，判断逻辑块号的有效性，然后查找高速缓冲区，如果该逻辑块正在缓冲区中，则将其释放掉。最后对逻辑块位图进行操作，复位对应的比特位。

函数 minix_new_block()用于从指定设备上申请一个逻辑块。首先获取指定设备的超级块，搜索整个逻辑块位图寻找首个是 0 的比特位，将其置 1。然后计算出该逻辑块的盘块号，并在高速缓冲区中申请相应的缓冲块将其清零。

函数 minix_free_inode()用于释放指定的 i 结点，并复位对应的 i 结点位图比特位。

函数 minix_new_inode()用于为设备建立一个新 i 结点。主要操作过程是在内存 i 结点表中获取一个空闲 i 结点表项，并从 i 结点位图中找一个空闲 i 结点。

2. 代码注释

bitmap.c 文件的详细代码注释如下：

```
1.      #include <string.h>                  //字符串头文件。定义了所有字符串操作函数
2.      #include <linux/sched.h>             //调度程序头文件。定义了 task_struct 和任务 0 的数据。一
                                             //些关于描述符参数设置和获取以及 switch_to()的嵌入式汇
                                             //编函数宏
3.      #include <linux/minix_fs.h>          //Minix 文件系统头文件。定义了与 Minix 文件系统相关的
                                             //函数和结构体
4.      #include <linux/kernel.h>            //内核头文件。定义了一些内核常用函数的原形
    //将指定地址处的一块内存清零，嵌入汇编程序宏
5.      #define clear_block(addr)
```

```
6.              __asm__("cld\n\t"
7.                      "rep\n\t"
8.                      "stosl"
9.              ::"a" (0),"c" (BLOCK_SIZE/4),"D" ((long) (addr)):"cx","di")
        //将指定地址开始的第 nr 个比特位置位并返回原比特位的嵌入汇编程序宏
10.         #define set_bit(nr,addr) ({
11.             char res;
12.             __asm__ __volatile__("btsl %1,%2\n\tsetb %0":
13.         "=q" (res):"r" (nr),"m" (*(addr)));
14.         res;})
        //将指定地址开始的第 nr 个比特位复位并返回原比特位的嵌入汇编程序宏
15.             #define clear_bit(nr,addr) ({
16.             char res;
17.             __asm__ __volatile__("btrl %1,%2\n\tsetnb %0":
18.         "=q" (res):"r" (nr),"m" (*(addr)));
19.         res;})
        //从 addr 开始寻找第一个 0 值比特位并返回距离 addr 比特位偏移值的嵌入汇编程序宏
20.             #define find_first_zero(addr) ({
21.             int __res;
22.         __asm__("cld\n"
23.             "1:\tlodsl\n\t"
24.                     "notl %%eax\n\t"
25.                     "bsfl %%eax,%%edx\n\t"
26.                     "jne 2f\n\t"
27.                     "addl $32,%%ecx\n\t"
28.             "cmpl $8192,%%ecx\n\t"
29.             "jl 1b\n\t"
30.                     "xorl %%edx,%%edx\n"
31.                     "2:\taddl %%edx,%%ecx"
32.                     :"=c" (__res):"0" (0),"S" (addr):"ax","dx","si");
33.         __res;})
        //释放指定设备上的指定逻辑块。成功返回 1，失败返回 0
34.     int minix_free_block(int dev, int block)
35.         {
36.             struct super_block * sb;
37.             struct buffer_head * bh;
38.             unsigned int bit,zone;
39.
40.             if (!(sb = get_super(dev)))          //取指定设备的超级块，不存在则出错死机
41.                 panic("trying to free block on nonexistent device");
42.             if (block < sb->s_firstdatazone || block >= sb->s_nzones) //如果指定的逻辑块号小于该设
43.                 panic("trying to free block not in datazone");  //备第一个逻辑块号或者大于总逻辑块
44.                                                 //数，则出错死机
45.                 bh = get_hash_table(dev,block);   //从 hash 表中获取指定设备和块的缓冲区
46.                 if (bh) {                       //判定指定设备和块的缓冲区是否存在，不存在，则
                                                    //直接对逻辑
47.                                                 //块位图中对应的比特位复位
48.             if (bh->b_count > 1) {              //存在，则先判断该缓冲块被使用的进程数，如果进
                                                    //程数大于 1
49.                 brelse(bh);                     //表明除了调用程序外，还有其他程序在使用，释放
```

```
                                                    //该缓冲区并
50.                 return 0;                       //返回
51.                 }
52.                 bh->b_dirt=0;                   //将已修改标志复位
53.                 bh->b_uptodate=0;                   //将更新标志复位
54.                 if (bh->b_count)                    //判定缓冲区被使用的进程数
55.                     brelse(bh);                 //释放该缓冲块
56.             }
57.             zone = block - sb->s_firstdatazone + 1; //zone 的值为从第一个逻辑块号算起，指定的逻
58.         //辑块为第几个逻辑块。该值也即该逻辑块在逻辑位图的位置
59.             bit = zone & 8191;                  //bit 的值为该逻辑块在所属的逻辑位图缓冲块中的位置
60.             zone >>= 13;                        //zone 的值为该逻辑块所属的逻辑位图缓冲块的指
                                                    //针在
61.                                                 //逻辑位图缓冲指针数组的位置
62.             bh = sb->s_zmap[zone];              //得到该逻辑块所属的逻辑位图缓冲块
63.             if (clear_bit(bit,bh->b_data))      //复位该逻辑块所对应的逻辑位图比特位。若对应比
                                                    //特位原来即是 0，则出错死机
64.             printk("free_block (%04x:%d): bit already cleared\n",dev,block);
65.
66.             bh->b_dirt = 1;                     //置该逻辑块所属逻辑位图缓冲块已修改标志
67.             return 1;
68.         }
        //向指定设备申请一个逻辑块，返回逻辑块号
69.         int minix_new_block(int dev)
70.         {
71.             struct buffer_head * bh;
72.             struct super_block * sb;
73.             int i,j;
74.
75.             if (!(sb = get_super(dev)))         //取指定设备的超级块，不存在则出错死机
76.                 panic("trying to get new block from nonexistant device");
77.             j = 8192;
78.             for (i=0 ; i<8 ; i++)               //扫描逻辑块位图，寻找首个 0 比特位
79.                 if (bh=sb->s_zmap[i])
80.                     if ((j=find_first_zero(bh->b_data))<8192)
81.                     break;
82.         if (i>=8 || !bh || j>=8192)             //如果扫描完仍未找到 0 比特位或者逻辑块位图缓冲
                                                    //块无效，退
83.                 return 0;                       //出，返回 0
84.         if (set_bit(j,bh->b_data))              //对找到的 0 比特位置位。若该比特位已置位，则出
                                                    //错死机
85.             panic("new_block: bit already set");
86.             bh->b_dirt = 1;                     //将对应的逻辑位图缓冲块的已修改标志置位
87.         j += i*8192 + sb->s_firstdatazone-1;    //计算新逻辑块号
88.         if (j >= sb->s_nzones)                  //如果新逻辑块号大于指定设备的总逻辑块数，表明
89.                 return 0;                       //申请失败，返回 0
90.         if (!(bh=getblk(dev,j)))                //获取指定设备新逻辑块的缓冲块，若失败则死机
91.             panic("new_block: cannot get block");
92.             if (bh->b_count != 1)               //新块的使用计数应该为 1，否则死机
93.             panic("new block: count is != 1");
```

```
94.             clear_block(bh->b_data);          //将新块的缓冲块清零
95.             bh->b_uptodate = 1;               //将更新标志置位
96.             bh->b_dirt = 1;                   //将已修改标志置位
97.             brelse(bh);                       //释放该缓冲块
98.             return j;                         //返回新逻辑块号
99.         }
        //释放指定的 i 结点函数
100.        void minix_free_inode(struct inode * inode)
101.            {
102.            struct buffer_head * bh;
103.
104.            if (!inode)                       //如果 i 结点指针为 NULL，则退出
105.            return;
106.            if (!inode->i_dev) {              //如果 i 结点上的设备号为 0，则清空对应 i 结点所占内存
                                                  //并退出
107.                memset(inode,0,sizeof(*inode));
108.            return;
109.        }
110.            if (inode->i_count>1) {           //如果此 i 结点引用计数大于 1，说明还有其他程序引用
                                                  //内核异常，退出
111.                printk("free_inode: inode has count=%d\n",inode->i_count);
112.            return;
113.        }
114.            if (inode->i_nlink) {             //如果文件目录项连接不为 0，则表明该目录项仍在使用中
                                                  //不应/释放，退出
115.                printk("free_inode: inode has nlink=%d\n",inode->i_nlink);
116.            return;
117.        }
118.            if (!inode->i_sb) {               //如果 i 结点所在设备的超级块不存在，则退出
119.                printk("free_inode: inode on nonexistent device\n");
120.            return;
121.        }
122.            if (inode->i_ino < 1 || inode->i_ino > inode->i_sb->s_ninodes) {    //如果 i 结点号小于 1
                                                                                    //或大于该
123.                printk("free_inode: inode 0 or nonexistent inode\n"); //设备的 i 结点总数，则退出
124.            return;
125.            }
126.            if (!(bh=inode->i_sb->s_imap[inode->i_ino>>13])) {      //如果该 i 结点对应的结点位图
                                                                        //缓冲块
127.                printk("free_inode: nonexistent imap in superblock\n"); //存在，则退出
128.            return;
129.        }
130.            if (clear_bit(inode->i_ino&8191,bh->b_data))        //将 i 结点对应的结点位图的比特位
                                                                    //复位，若该比特位已经为 0，则出错
131.                printk("free_inode: bit already cleared.\n\r");
132.        bh->b_dirt = 1;                       //将该 i 结点位图缓冲区已修改标志置位
133.        memset(inode,0,sizeof(*inode));   //清空 i 结点结构所占内存区
134.          }
        //为指定设备建立一个新 i 结点，返回 i 结点指针
135.            struct inode * minix_new_inode(int dev)
```

```
136.    {
137.            struct inode * inode;
138.            struct buffer_head * bh;
139.            int i,j;
140.
141.            if (!(inode=get_empty_inode()))    //从内存 i 结点表中获取一个空闲 i 结点表项，失败返
                                                   //回 NULL
142.            return NULL;
143.        if (!(inode->i_sb = get_super(dev))) {    //将该 i 结点超级块指针指向指定设备超级块，即
144.                printk("new_inode: unknown device\n"); //将该 i 结点添加为指定设备的 i 结点。如果
145.            iput(inode);                           //失败，则释放该内存 i 结点，返回 NULL
146.            return NULL;
147.            }
148.            j = 8192;
149.            for (i=0 ; i<8 ; i++)                  //扫描 i 结点位图，寻找首个 0 比特位
150.                if (bh=inode->i_sb->s_imap[i])
151.                if ((j=find_first_zero(bh->b_data))<8192)
152.                    break;
153.            if (!bh || j >= 8192 || j+i*8192 > inode->i_sb->s_ninodes) { //如果全部扫描完仍未找到，或
154.                iput(inode);                       //者位缓冲块无效，则释放该内存 i 结点，返回 NULL
155.            return NULL;
156.        }
    //对该 i 结点对应的 i 结点位图比特位置位，如果已经置位，则释放该内存 i 结点，返回 NULL
157.            if (set_bit(j,bh->b_data)) {              /*不应该发生*/
158.            printk("new_inode: bit already set");
159.            iput(inode);
160.                return NULL;
161.            }
162.            bh->b_dirt = 1;                    //对该 i 结点对应的 i 结点位图缓冲区的已修改标志置位
163.            inode->i_count = 1;                //对该 i 结点应用计数置 1
164.            inode->i_nlink = 1;                //对该 i 结点文件目录项链接数置 1
165.            inode->i_dev = dev;                //设置该 i 结点所属设备
166.            inode->i_uid = current->cuid;      //设置该 i 结点所属用户 id
167.            inode->i_gid = current->egid;      //设置该 i 结点所属组 id
168.            inode->i_dirt = 1;                 //对已修改标志位置位
169.            inode->i_ino = j + i*8192;         //对应设备中的 i 结点号
170.            inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME;   //设置时间
171.            inode->i_op = &minix_inode_operations; //设置该 i 结点的操作方式为 Minix 结点操作
172.        return inode;                          //方式。返回该 i 结点指针
173.    }
```

源代码中第 11～19 行声明了 bitmap.c 包含处理 i 结点位图和块位图的代码。

8.19 Minix 的 inode.c 文件

本文件主要实现了 Minix 文件系统的文件 i 结点读取和写入操作

函数 int _bmap()指定文件数据块映射到逻辑块；函数 minix_bmap()取指定文件的指定块所对应的逻辑块号；函数 minix_create_block()创建指定文件的指定块所对应的逻辑块；函数 minix_read_inode()从设备上读取指定 i 结点的信息到内存中；函数 minix_write_inode()将指定 i 结

点信息写入缓冲区相应设备的缓冲块中，待缓冲区刷新时会写入盘中。

inode.c 文件的详细代码注释如下：

```
1.      #include <string.h>            //字符串头文件。定义了所有字符串操作函数
2.      #include <sys/stat.h>          //文件状态头文件。主要说明了函数 stat()返回的数据及其
                                       //结构类型
                                       //一些属性操作测试宏、函数原型
3.      #include <linux/sched.h>       //调度程序头文件。定义了 task_struct 和任务 0 的数据。一些关
                                       //于描述符参数设置和获取以及 switch_to()的嵌入式汇编函数宏
4.      #include <linux/minix_fs.h     //Minix 文件系统头文件。定义了与 MINIX 文件系统相关的函
                                       //数和结构体
5.      #include <linux/kernel.h>      //内核头文件。定义了一些内核常用函数的原形
6.      #include <linux/mm.h>          //内存管理头文件。定义了内存页面的大小和页面释放函数原型
7.      #include <asm/system.h>        //系统头文件。定义了设置或修改描述符、中断门等的嵌入式汇编宏
    //指定文件数据块映射到逻辑块的函数
8.      static int _bmap(struct inode * inode,int block,int create)
9.      {
10.             struct buffer_head * bh;
11.             int i;
12.
13.       if (block<0)                 //如果块号小于 0，则表明出错，停机
14.             panic("_bmap: block<0");
15.       if (block >= 7+512+512*512//如果块号超出文件系统表示范围，则表明出错，停机
16.                   panic("_bmap: block>big");
17.       if (block<7) {               //如果指定块为直接块
18.       if (create && !inode->i_data[block]) //如果创建标志置位并且指定块对应逻辑块为 0，
19.                 if (inode->i_data[block]=minix_new_block(inode->i_dev)) {   //则向设备申
20.                       //请一逻辑块
21.                 inode->i_ctime=CURRENT_TIME;   //设置 i 结点修改时间
22.                       inode->i_dirt=1;                    //置位修改标志
23.           }
24.       return inode->i_data[block];                        //返回逻辑块号
25.       }
26.       block -= 7;
27.       if (block<512) {         //如果指定块为一次间接块
28.       if (create && !inode->i_data[7]) //如果创建标志置位并且一次间接块对应逻辑块为 0
29.           if (inode->i_data[7]=minix_new_block(inode->i_dev)) { //则向设备申请一逻
30.                       inode->i_dirt=1;                    //辑块，置位修改标志
31.                 inode->i_ctime=CURRENT_TIME;              //设置 i 结点修改时间
32.           }
33.           if (!inode->i_data[7])                          //如果申请逻辑块失败，则返回 0
34.                 return 0;
35.           if (!(bh = bread(inode->i_dev,inode->i_data[7])))   //从设备上读取一次间接块，如
                                                              //果失败，返回 0
36.           return 0;
37.       i = ((unsigned short *) (bh->b_data))[block];       //取一次间接块上对应指定块的
                                                              //逻辑块号
38.           if (create && !i)        //如果创建标志置位并且指定块对应逻辑块为 0
39.           if (i=minix_new_block(inode->i_dev)) {          //则向设备申请一逻辑块
40.                 ((unsigned short *) (bh->b_data))[block]=i;   //指定块与新逻辑块建立映
```

```
41.                         bh->b_dirt=1;                          //射关系，置位修改标志
42.                 }
43.         brelse(bh);                                            //释放一次间接块
44.                 return i;                                      //返回指定块对应的逻辑块号
45.     }
46.     block -= 512;                                              //程序能够运行到这个地方，表明指定块为二次
                                                                   //间接块
47.         if (create && !inode->i_data[8])                       //如果创建标志置位并且二次间接块对应逻辑块为 0
48.         if (inode->i_data[8]=minix_new_block(inode->i_dev)) {  //则向设备申请一逻辑块
49.                 inode->i_dirt=1;                               //置位修改标志
50.                         inode->i_ctime=CURRENT_TIME;           //设置 i 结点修改时间
51.         }
52.     if (!inode->i_data[8])                                     //如果申请逻辑块失败，则返回 0
53.                 return 0;
54.     if (!(bh=bread(inode->i_dev,inode->i_data[8]))) //从设备上读取二级间接块一级块，如
55.         return 0;                                              //果失败，返回 0
56.     i = ((unsigned short *)bh->b_data)[block>>9];//取指定块所属二次间接块的二级块的
                                                                   //逻辑块号
57.         if (create && !i)                                      //如果创建标志置位并且指定块所属二级块对应
                                                                   //逻辑块为 0
58.         if (i=minix_new_block(inode->i_dev)) {//则向设备申请一逻辑块
59.                 ((unsigned short *) (bh->b_data))[block>>9]=i;      //该二级块与新逻辑块建立
60.                         bh->b_dirt=1;                          //映射关系，置位二次间接块一级块修改标志
61.         }
62.     brelse(bh);                                                //释放二次间接块一级块
63.             if (!i)                                            //如果指定块所属二级块对应逻辑块为 0，表明
                                                                   //逻辑块
64.                 return 0;                                      //申请失败，返回 0
65.         if (!(bh=bread(inode->i_dev,i)))                       //读取该二次间接块的二级块，失败返回 0
66.                         return 0;
67.         i = ((unsigned short *)bh->b_data)[block&511];                  //取指定块所对应的逻辑块号
68.         if (create && !i)                                      //如果创建标志置位并且指定块对应逻辑块为 0
69.         if (i=minix_new_block(inode->i_dev)) {                 //则向设备申请一逻辑块
70.                         ((unsigned short *) (bh->b_data))[block&511]=i;    //指定块与新逻辑块建立映射
71.                         bh->b_dirt=1;                          //关系，置位指定块所属二级块修改标志
72.                         }
73.                 brelse(bh);                                    //释放指定块所属二级块，返回指定块对应逻辑
                                                                   //块号
74.         return i;
75.       }
      //取指定文件的指定块所对应的逻辑块号
76.         int minix_bmap(struct inode * inode,int block)
77.         {
78.                 return _bmap(inode,block,0);     //调用创建标志为 0 的_bmap()函数
79.       }
      //创建指定文件的指定块所对应的逻辑块，返回该逻辑块号
80.       int minix_create_block(struct inode * inode, int block)
81.         {
82.        return _bmap(inode,block,1);                            //调用创建标志置位的_bamp()函数
83.       }
      //Minix 文件系统的从设备上读取指定 i 结点信息到内存中的函数
```

```
84.          void minix_read_inode(struct inode * inode)
85.          {
86.              struct buffer_head * bh;
87.          struct minix_inode * raw_inode;
88.          int block;
89.          block = 2 + inode->i_sb->s_imap_blocks + inode->i_sb->s_zmap_blocks +
90.                      (inode->i_ino-1)/MINIX_INODES_PER_BLOCK; //计算指定 i 结点所在的逻辑块号
91.          if (!(bh=bread(inode->i_dev,block)))                    //取该 i 结点所在的逻辑块
92.              panic("unable to read i-node block");               //失败则打印错误信息，停机
93.              raw_inode = ((struct minix_inode *) bh->b_data) +  //raw_inode 指向从内存读出的指定 i
94.              (inode->i_ino-1)%MINIX_INODES_PER_BLOCK;//结点对应的 Minix 文件系统 i 结点
95.           //将从内存中读出的 Minix 文件系统 i 结点信息赋给指定 i 结点
96.              inode->i_mode = raw_inode->i_mode;
97.              inode->i_uid = raw_inode->i_uid;
98.              inode->i_gid = raw_inode->i_gid;
99.              inode->i_nlink = raw_inode->i_nlinks;
100.             inode->i_size = raw_inode->i_size;
101.             inode->i_mtime = inode->i_atime = inode->i_ctime = raw_inode->i_time;
102.             if (S_ISCHR(inode->i_mode) || S_ISBLK(inode->i_mode))  //如果是块设备或者字符设备的 i
103.             inode->i_rdev = raw_inode->i_zone[0];                  //结点，此时 MINIX 文件系统中的
                                                                       //i_zone[0]
104.            //是设备号，将其赋给指定 i 结点的设备号
105.         else for (block = 0; block < 9; block++)                  //如果是常规文件的 i 结点
106.           inode->i_data[block] = raw_inode->i_zone[block];        //将文件所占用逻辑块号数组复制
107.             brelse(bh);                                           //释放读入的缓冲区
108.         inode->i_op = &minix_inode_operations; //指定结点操作方式为 Minix 文件系统操作方式
109.     }
     //Minix 文件系统的将指定 i 结点信息写入设备
110.         void minix_write_inode(struct inode * inode)
111.         {
112.             struct buffer_head * bh;
113.             struct minix_inode * raw_inode;
114.             int block;
115.
116.             block = 2 + inode->i_sb->s_imap_blocks + inode->i_sb->s_zmap_blocks +
117.                     (inode->i_ino-1)/MINIX_INODES_PER_BLOCK;  //计算指定 i 结点所在的逻辑
    块号。取该 i 结点所在的逻辑块，失败则打印错误信息，停机
118.        if (!(bh=bread(inode->i_dev,block)))
119.             panic("unable to read i-node block");
120.             raw_inode = ((struct minix_inode *)bh->b_data) +  //raw_inode 指向从内存读出的指定 i
121.             (inode->i_ino-1)%MINIX_INODES_PER_BLOCK;  //结点对应的 Minix 文件系统 i 结点
    //将指定 i 结点信息赋给从内存中读出的 Minix 文件系统 i 结点
122.             raw_inode->i_mode = inode->i_mode;
123.             raw_inode->i_uid = inode->i_uid;
124.             raw_inode->i_gid = inode->i_gid;
125.             raw_inode->i_nlinks = inode->i_nlink;
126.             raw_inode->i_size = inode->i_size;
127.             raw_inode->i_time = inode->i_mtime;
128.        if (S_ISCHR(inode->i_mode) || S_ISBLK(inode->i_mode)) //如果是块设备或者字符设备的 i
129.             raw_inode->i_zone[0] = inode->i_rdev;        //结点，将指定 i 结点的设备号赋给 Minix
130.                                                          //文件系统 i 结点的 i_zone[0]
131.        else for (block = 0; block < 9; block++)           //如果是常规文件的 i 结点
132.             raw_inode->i_zone[block] = inode->i_data[block]; //将文件所占用逻辑块号数组
```

```
133.        bh->b_dirt=1;                    //复制。置位该缓冲块修改标志，刷
                                             //新时可写入盘中
134.        inode->i_dirt=0;                 //复位指定 i 结点修改标志
135.        brelse(bh);                      //释放该缓冲块
136.      }
```

8.20 Minix 的 truncate.c 文件

本文件用于释放指定 i 结点在设备上占用的所有逻辑块。

1. 功能描述

本程序实现了将 Minix 文件系统的文件长度截断为 0 的功能，实现方式是将指定文件结点在设备上占用的所有逻辑块释放，包括直接块、一次间接块和二次间接块，文件 i 结点逻辑块数组如图 8-4 所示。函数 minix_free_ind()用于释放一次间接块；函数 minix_free_dind()用于释放二次间接块；minix_truncate(struct inode * inode) 将指定结点对应的文件长度截断为 0，并释放占用的设备空间；

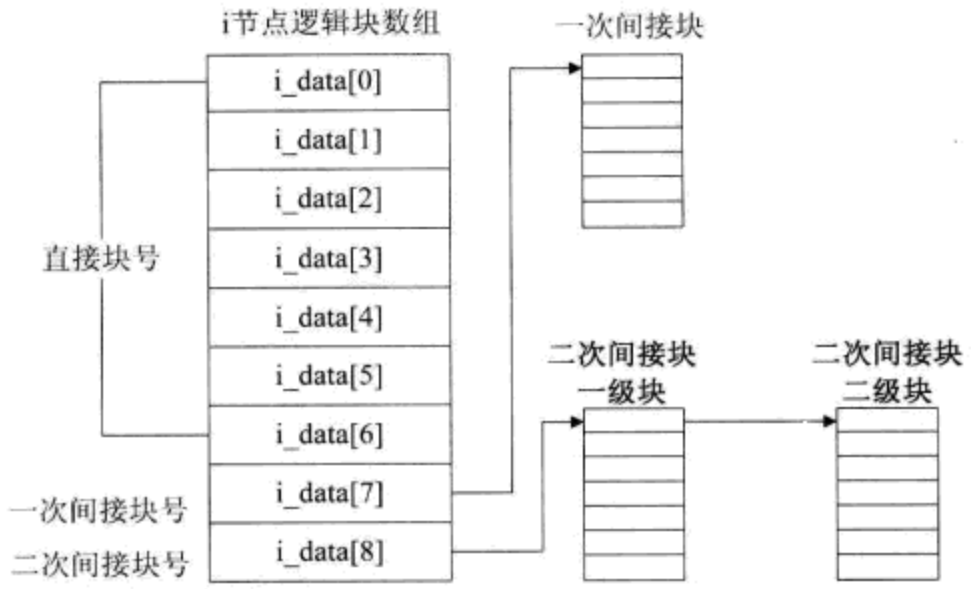


图 8-4　i 结点逻辑块数组

2. 代码注释

truncate.c 文件的详细代码注释如下：

```
1.          #include <linux/sched.h>   //调度程序头文件。定义了 task_struct 和任务 0 的数据。一些关
                                       //于描述符参数设置和获取以及 switch_to()的嵌入式汇编函数宏
2.          #include <linux/minix_fs.h> //Minix 文件系统头文件。定义了与 MINIX 文件系统相关的函
                                       //数和结构体
3.          #include <linux/tty.h>     //tty 头文件，定义了 tty_io.c 程序使用的某些结构和其他一些定义
4.          #include <errno.h>         //错误号头文件。包含系统中各种出错号
5.          #include <fcntl.h>         //文件控制头文件。用于文件及其描述符的操作控制常数符号的
                                       //定义
6.          #include <sys/stat.h>      //文件状态头文件。主要说明了函数 stat()返回的数据及其结构类型
                                       //一些属性操作测试宏、函数原型
      //释放指定的一次间接块。
7.          static int minix_free_ind(int dev,int block)
8.          {
9.              struct buffer_head * bh;
10.             unsigned short * p;
11.             int i;
```

```
12.             int block_busy;
13.             if (!block)                                    //如果指定逻辑块号为 0，返回 1
14.                     return 1;
15.             block_busy = 0;
16.             if (bh=bread(dev,block)) {                     //读取一次间接块
17.                     p = (unsigned short *) bh->b_data;     //指向一次间接块首项
18.                     for (i=0;i<512;i++,p++)                //遍历一次间接块的 512 项
19.                             if (*p)                        //如果该项存有逻辑块号
20.                             if (minix_free_block(dev,*p)) {//释放相应逻辑块
21.                                     *p = 0;                //该项置 0
22.                                     bh->b_dirt = 1;        //对一次间接块的缓冲区修改标志置位
23.                             } else                         //如果释放相应逻辑块失败
24.                                     block_busy = 1;        //置 block_busy 为 1，表明存在逻辑块未释放
25.                     brelse(bh);                            //释放一次间接块缓冲区
26.             }
27.             if (block_busy)                                //如果存在逻辑块未释放，返回 0
28.                     return 0;
29.             else                                           //一次间接块上所有逻辑块都被释放
30.                     return minix_free_block(dev,block);    //释放一次间接块
31.     }
    //释放二次间接块
32.     static int minix_free_dind(int dev,int block)
33.     {
34.             struct buffer_head * bh;
35.             unsigned short * p;
36.             int i;
37.             int block_busy;
38.
39.             if (!block)                                    //如果指定逻辑块号为 0，返回 1
40.                     return 1;
41.             block_busy = 0;
42.             if (bh=bread(dev,block)) {                     //读取二次间接块的一级块
43.                     p = (unsigned short *) bh->b_data;     //指向二次间接块一级块的首项
44.                     for (i=0;i<512;i++,p++)                //遍历一级块的 512 项
45.                             if (*p)                        //如果该项存有二级块逻辑块号
46.                                     if (minix_free_ind(dev,*p)) { //释放该二级块，如果释放成功
47.                                             *p = 0;        //该项置 0
48.                                             bh->b_dirt = 1;//置位二次间接块的一级块缓冲区修改标志
49.                                     } else                 //如果释放失败，则置 block_busy 为 1，表
                                                               //明存
50.                                             block_busy = 1;//在逻辑块未被释放
51.                     brelse(bh);                            //释放二次间接块的一级块缓冲区
52.             }
53.             if (block_busy)                                //如果存在逻辑块未释放，返回 0
54.                     return 0;
55.             else                                           //否则，释放二次间接块
56.                     return minix_free_block(dev,block);
57.     }
    //将指定结点对应的文件长度截为 0，并释放占用的设备空间
58.     void minix_truncate(struct inode * inode)
```

```
59.     {
60.         int i;
61.         int block_busy;
62.
63.         if (!(S_ISREG(inode->i_mode) || S_ISDIR(inode->i_mode) ||
64.               S_ISLNK(inode->i_mode)))        //如果不是常规文件或者是目录文件、LNK，则返回
65.             return;
66.     repeat:
67.         block_busy = 0;
68.         for (i=0;i<7;i++)                     //释放直接块
69.             if (inode->i_data[i]) {
70.                 if (minix_free_block(inode->i_dev,inode->i_data[i]))
71.                     inode->i_data[i]=0;
72.                 else
73.                     block_busy = 1;           //如果存在直接块未被释放，block_busy 置 1
74.             }
75.         if (minix_free_ind(inode->i_dev,inode->i_data[7]))        //释放一次间接块
76.             inode->i_data[7] = 0;
77.         else                                  //如果一次间接块未被释放，block_busy 置 1
78.             block_busy = 1;
79.         if (minix_free_dind(inode->i_dev,inode->i_data[8]))       //释放二次间接块
80.             inode->i_data[8] = 0;
81.         else                                  //如果二次间接块未被释放，block_busy 置 1
82.             block_busy = 1;
83.         inode->i_dirt = 1;                    //置位指定结点的修改位
84.         if (block_busy) {                     //如果存在块未被释放，则重新释放
85.             current->counter = 0;
86.             schedule();
87.             goto repeat;
88.         }
89.         inode->i_size = 0;                    //文件大小置为 0
90.         inode->i_mtime = inode->i_ctime = CURRENT_TIME;   //重置文件和结点修改时间为当
                                                              //前时间
91.     }
    //文件 i 结点释放函数
92.     void minix_release(struct inode * inode, struct file * filp)
93.     {
94.         printk("minix_release not implemented\n");   //该函数未实现，打印警告信息
95.     }
    //检查字符设备文件函数
96.     static int check_char_dev(struct inode * inode, struct file * filp)
97.     {
98.         struct tty_struct *tty;
99.         int min, dev;
100.
101.        dev = inode->i_rdev;                  //获取该字符设备的设备号
102.        if (MAJOR(dev) == 4 || MAJOR(dev) == 5) { //如果该设备为控制终端设备或者串口终端
103.            if (MAJOR(dev) == 5)              //设备，如果该设备为串口终端设备
104.                min = current->tty;           //则获取该设备的次设备号
105.            else                              //如果该设备为控制终端设备
```

```
106.                    min = MINOR(dev);                    //则获取该设备的次设备号
107.                if (min < 0)                             //如果次设备号小于 0
108.                    return -1;                           //则返回-1
109.                if ((IS_A_PTY_MASTER(min)) && (inode->i_count>1)) //如果是空设备并且正在被
110.                    return -1;                           //使用，则返回-1
111.                tty = TTY_TABLE(min);                    //获取该设备的 tty 数据结构
        //如果文件创建标志中设置了分配控制终端位，并且当前进程是会话头领，并且当前进程
        //没有控制终端，并且该设备没有任会话
112.                if (!(filp->f_flags & O_NOCTTY) &&
113.                    current->leader &&
114.                    current->tty<0 &&
115.                    tty->session==0) {
116.                    current->tty = min;                  //则设置当前进程的控制终端为该设备
117.                    tty->session= current->session;      //将当前进程的会话作为设备的会话
118.                    tty->pgrp = current->pgrp;           //设置设备所属进程组号为当前进程组号
119.                }
120.            }
121.            return 0;                                    //返回 0
122.        }
    //一个 Minix 文件打开时都会调用 Minix 文件打开函数
123.        int minix_open(struct inode * inode, struct file * filp)
124.        {
125.            if (S_ISCHR(inode->i_mode)) {                //如果指定文件是字符设备文件
126.                if (check_char_dev(inode,filp))          //则检查该字符文件设备
127.                    return -EAGAIN;                      //如果失败，返回错误码
128.            } else if (S_ISBLK(inode->i_mode))           //如果指定文件是块设备文件
129.                check_disk_change(inode->i_rdev);        //则检查是否更换过磁盘
130.            filp->f_op = &minix_file_operations;         //将指定的文件结构操作指针指向 Minix 文
                                                             //件操作
131.                                                         //结构体
132.            return 0;                                    //返回 0
133.        }
```

源代码中第 92～95 行定义了文件节点释放函数，当一个文件 i 结点被释放，这个函数被调用。注意，这个函数与函数 minix_open 不同：每次文件打开的时候 open 都被调用，但只有所有文件被关闭时 release 才被调用。

8.21 Minix 的 namei.c 文件

本文件主要实现了在 Minix 文件系统中对目录和文件的一些基本操作，例如创建、删除等操作的系统调用函数的定义。

1. 功能描述

本程序主要实现了在 Minix 文件系统中一些关于目录、文件的建立和删除等操作函数及系统调用。

函数 minix_match()将指定长度字符串与目录项中文件名比较；

函数 minix_find_entry()查找指定目录下指定文件名的目录项；

函数 minix_follow_link()获取指定目录下指定符号连接文件的目标文件 i 结点；

函数 minix_lookup()在指定目录下寻找指定文件名的 i 结点；

函数 minix_add_entry()在指定目录下添加指定文件名的目录项；
函数 minix_create()在制定目录下创建一个指定文件名的结点；
函数 minix_mknod()在指定目录下创建一个指定文件名的特殊文件或普通文件结点；
函数 minix_mkdir()在指定目录下创建指定目录名的目录结点；
函数 empty_dir()检查指定目录是否为空目录；
函数 minix_rmdir()删除指定目录下指定目录名的目录；
函数 minix_unlink()删除指定目录下指定文件名的文件；
函数 minix_symlink()创建一个符号连接文件;
函数 minix_link()为一个已经存在的文件创建一个硬连接；
函数 subdir()判定指定文件 new 是否在指定目录 old 或其子目录下；
函数 do_minix_rename()用于文件重命名；
函数 minix_rename()调用文件重命名函数的互斥锁函数；
函数 minix_readlink()读取符号连接目标文件名函数。

2. 代码注释

namei.c 文件的详细代码注释如下：

```
1.      #include <linux/sched.h>          //调度程序头文件。定义了 task_struct 和任务 0 的数据。一些关
                                          //于描述符参数设置和获取以及 switch_to()的嵌入式汇编函数宏
2.      #include <linux/minix_fs.h>       //Minix 文件系统头文件。定义了与 Minix 文件系统相关的函数
                                          //和结构体
3.      #include <linux/kernel.h>         //内核头文件。定义了一些内核常用函数的原型
4.      #include <asm/segment.h>          //段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数
5.      #include <string.h>               //字符串头文件。定义了所有字符串操作函数
6.      #include <fcntl.h>                //文件控制头文件。用于文件及其描述符的操作控制常数符号的
                                          //定义
7.      #include <errno.h>                //错误号头文件。包含系统中各种出错号
8.      #include <const.h>                //常数符号头文件。定义了 i 结点中 i_mode 字段的各标志位
9.      #include <sys/stat.h>             //文件状态头文件。主要说明了函数 stat()返回的数据及其结构类型
                                          //一些属性操作测试宏、函数原型
10.     extern int permission(struct inode * inode,int mask);
11.     extern struct inode * _namei(const char * filename, struct inode * base,
12.         int follow_links);
        //指定长度字符串与目录项中文件名比较函数
13.     static int minix_match(int len,const char * name,struct minix_dir_entry * de)
14.     {
15.         register int same __asm__("ax");
16.
17.         if (!de || !de->inode || len > MINIX_NAME_LEN) //如果目录项指针为空，或者目录项 i 结点号为
                                                           //0，或者指定长度超过 MINIX 文件名长度
18.             return 0;                                  //则返回 0
        //"" 表示 "." 像这样路径"/usr/lib//libc.a" 的情况
19.         if (!len && (de->name[0]=='.') && (de->name[1]=='\0')) //如果长度不为 0，并且目录项中文件名
20.                                                                //为“.”，返回 1
21.             return 1;
22.         if (len < MINIX_NAME_LEN && de->name[len]) //如果指定的长度小于 Minix 文件名长度，同
23.                                                    //时目录项文件名长度超过指定长度
24.             return 0;                              //返回 0
```

```
        //利用嵌入式汇编语句，在数据空间执行字符串比较操作
25.         __asm__("cld\n\t"
26.                 "fs ; repe ; cmpsb\n\t"
27.                 "setz %%al"
28.                 :"=a" (same)
29.                 :"0" (0),"S" ((long) name),"D" ((long) de->name),"c" (len)
30.                 :"cx","di","si");
31.         return same;                                    //返回比较结果
32.         }
        //查找指定目录下指定文件名的目录项
33.         static struct buffer_head * minix_find_entry(struct inode * dir,
34.         const char * name, int namelen, struct minix_dir_entry ** res_dir)
35.     {
36.         int entries;
37.         int block,i;
38.         struct buffer_head * bh;
39.         struct minix_dir_entry * de;
40.
41.         *res_dir = NULL;
42.         if (!dir)                                       //如果指定目录 i 结点不存在，则返回 NULL
43.                 return NULL;
44.         #ifdef NO_TRUNCATE                              //如果定义了 NO_TRUNCATE
45.         if (namelen > MINIX_NAME_LEN)                   //如果指定文件名长度超过 MINIX_NAME_LEN
46.                 return NULL;                            //则返回 NULL
47.         #else                                           //如果未定义 NO_TRUNCATE
48.         if (namelen > MINIX_NAME_LEN)                   //如果指定文件名长度超过 MINIX_NAME_LEN
49.                 namelen = MINIX_NAME_LEN; //则截断超过的部分
50.         #endif
51.         entries = dir->i_size / (sizeof (struct minix_dir_entry)); //获取本目录含有的目录项个数
52.         if (!(block = dir->i_data[0]))                  //如果该目录 i 结点指向的第一个逻辑块号为 0
53.                 return NULL;                            //返回 NULL
54.         if (!(bh = bread(dir->i_dev,block)))            //读取目录项数据块
55.                 return NULL;                            //如果失败，返回 NULL
56.         i = 0;
57.         de = (struct minix_dir_entry *) bh->b_data;             //de 指向目录项数据块的第一个目录项
58.         while (i < entries) {                           //遍历所有的目录项
59.                 if ((char *)de >= BLOCK_SIZE+bh->b_data) { //如果当前目录项数据块已经搜索完
60.                                                         //仍未找到指定目录项
61.                         brelse(bh);                     //释放当前目录项数据块
62.                         bh = NULL;                      //bh 指针置空
63.                         if (!(block = bmap(dir,i/MINIX_DIR_ENTRIES_PER_BLOCK)) ||  //然后读入下一目
64.                             !(bh = bread(dir->i_dev,block))) {  //录项数据块，如果该块为空，并且并未搜
65.                                                         //索完所有目录项
66.                                 i += MINIX_DIR_ENTRIES_PER_BLOCK; //则跳过该块，读入下一目录项
                                                                    //数据块
67.                                 continue;
68.                         }
69.                         de = (struct minix_dir_entry *) bh->b_data; //如果该块不为空，则使 de 指向该块的
70.                                                         //第一个目录项，继续搜索
71.                 }
```

```
72.             if (minix_match(namelen,name,de)) {          //如果找到匹配的目录项
73.                 *res_dir = de;                           //指向则返回该目录项指针
74.                 return bh;                               //返回包含该目录项的缓冲区指针
75.             }
76.             de++;                                        //否则，de 指向下一个目录项
77.             i++;                                         //i 加 1，继续比较下一个目录项
78.         }
79.         brelse(bh);                                      //如果在指定目录中未找到指定的目录项
                                                             //则释放目录项数据块
80.         return NULL;                                     //返回 NULL
81.     }
    //获取指定目录下指定符号连接文件的目标文件 i 结点
82.     struct inode * minix_follow_link(struct inode * dir, struct inode * inode)
83.     {
84.         unsigned short fs;
85.         struct buffer_head * bh;
86.
87.         if (!dir) {                                      //如果指定目录 i 结点不存在
88.             dir = current->root;                         //则将 dir 指向当前进程的根目录 i 结点
89.             dir->i_count++;                              //将该目录 i 结点引用计数加 1
90.         }
91.         if (!inode) {                                    //如果指定的符号连接文件 i 结点不存在
92.             iput(dir);                                   //则释放目录 i 结点，返回 NULL
93.             return NULL;
94.         }
95.         if (!S_ISLNK(inode->i_mode)) {                   //如果指定的文件 i 结点不是符号连接文件
                                                             //i 结点
96.             iput(dir);                                   //释放目录 i 结点
97.             return inode;                                //返回指定文件 i 结点
98.         }
99.         __asm__("mov %%fs,%0":"=r" (fs));
    //如果当前进程连接数大于 5，或者指定符号连接文件 i 结点的第一个逻辑块号为 0
    //或者读取第一个逻辑块的操作失败，
100.        if ((current->link_count > 5) || !inode->i_data[0] ||
101.            !(bh = bread(inode->i_dev, inode->i_data[0]))) {
102.            iput(dir);                                   //则释放目录 i 结点
103.            iput(inode);                                 //释放符号连接文件 i 结点，返回 NULL
104.            return NULL;
105.        }
106.        iput(inode);                                     //释放符号连接文件 i 结点
107.        __asm__("mov %0,%%fs"::"r" ((unsigned short) 0x10));
108.        current->link_count++;                           //当前进程连接计数加 1
109.        inode = _namei(bh->b_data,dir,1);                //获取符号连接的目标文件 i 结点
110.        current->link_count--;                           //当前进程连接计数减 1
111.        __asm__("mov %0,%%fs"::"r" (fs));
112.        brelse(bh);                                      //释放该缓冲区
113.        return inode;                                    //返回符号连接目标文件 i 结点
114.    }
    //在指定目录下寻找指定文件名的 i 结点
115.    int minix_lookup(struct inode * dir,const char * name, int len,
```

```
116.        struct inode ** result)
117.    {
118.        int ino;
119.        struct minix_dir_entry * de;
120.        struct buffer_head * bh;
121.
122.        *result = NULL;
123.        if (!dir)                                   //如果指定目录 i 结点不存在，返回错误码
124.                return -ENOENT;
125.        if (!S_ISDIR(dir->i_mode)) {                //如果指定目录 i 结点不是目录 i 结点
126.                iput(dir);                          //释放该 i 结点，返回错误码
127.                return -ENOENT;
128.        }
129.        if (!(bh = minix_find_entry(dir,name,len,&de))) { //获取指定文件的目录项
130.                iput(dir);                          //如果失败，则释放指定目录 i 结点
131.                return -ENOENT;                     //返回错误码
132.        }
133.        ino = de->inode;                            //获取指定文件的 i 结点号
134.        brelse(bh);                                 //释放包含指定文件目录项的缓冲区
135.        if (!(*result = iget(dir->i_dev,ino))) {    //获取指定文件的 i 结点
136.                iput(dir);                          //如果失败，则释放指定目录 i 结点
137.                return -EACCES;                     //返回错误码
138.        }
139.        iput(dir);                                  //最终，释放指定目录 i 结点，返回 0
140.        return 0;
141.    }
        //在指定目录下添加指定文件名的目录项
142.    static struct buffer_head * minix_add_entry(struct inode * dir,
143.        const char * name, int namelen, struct minix_dir_entry ** res_dir)
144.        {
145.        int block,i;
146.        struct buffer_head * bh;
147.        struct minix_dir_entry * de;
148.
149.        *res_dir = NULL;
150.        if (!dir)                                   //如果指定目录 i 结点不存在，返回错误码
151.                return NULL;
152.    #ifdef NO_TRUNCATE                              //如果定义了 NO_TRUNCATE
153.        if (namelen > MINIX_NAME_LEN)               //如果指定文件名长度超过 MINIX_NAME_LEN
154.                return NULL;                        //则返回 NULL
155.    #else                                           //如果未定义 NO_TRUNCATE
156.        if (namelen > MINIX_NAME_LEN)               //则如果指定文件名长度超过 MINIX_NAME_LEN
157.                namelen = MINIX_NAME_LEN;           //则截断超过的部分
158.    #endif
159.        if (!namelen)                               //如果指定文件名长度为 0，返回 NULL
160.                return NULL;
161.        if (!(block = dir->i_data[0]))              //如果该目录 i 结点指向的第一个逻辑块号为 0
162.                return NULL;                        //返回 NULL
163.        if (!(bh = bread(dir->i_dev,block)))        //读取目录项数据块
164.                return NULL;                        //失败返回 NULL
```

```
165.        i = 0;
166.        de = (struct minix_dir_entry *) bh->b_data;    //de 指向目录项数据块的第一个目录项
167.        while (1) {                                    //循环查找未使用的目录项
168.            if ((char *)de >= BLOCK_SIZE+bh->b_data) {  //如果当前目录项数据块已经搜索完
169.                brelse(bh);                            //释放该目录项数据块
170.                bh = NULL;                             //bh 置为空
171.                block = minix_create_block(dir,i/MINIX_DIR_ENTRIES_PER_BLOCK); //重新申请
172.                                                       //一块逻辑块 block
173.                if (!block)                            //如果失败，返回 NULL
174.                    return NULL;
175.                if (!(bh = bread(dir->i_dev,block))) {    //读取刚申请的逻辑块
176.                    i += MINIX_DIR_ENTRIES_PER_BLOCK; //如果失败，则跳过该块，继续
177.                    continue;
178.                }
179.                de = (struct minix_dir_entry *) bh->b_data; //获取该目录项数据块的第一个目录项
180.            }
181.            if (i*sizeof(struct minix_dir_entry) >= dir->i_size) { //如果当前所判定的目录项序号乘以目
182.            //录项结构大小已经超过了指定目录的指出的目录项总
183.            //大小，则表明第 i 个目录项未使用，因此知道空闲目录项
184.                de->inode=0;                                  //该目录项 i 结点号为 0
185.                dir->i_size = (i+1)*sizeof(struct minix_dir_entry);  //更新指定目录长度值
186.                dir->i_dirt = 1;                              //置位指定目录 i 结点修改标志
187.                dir->i_ctime = CURRENT_TIME;                  //更新指定目录 i 修改时间
188.            }
189.            if (!de->inode) {                                 //若该目录项 i 结点为空
190.                dir->i_mtime = CURRENT_TIME;                  //更新目录的修改时间
191.                for (i=0; i < MINIX_NAME_LEN ; i++)  //将文件名从用户数据区复制到该目录项
192.                    de->name[i]=(i<namelen)?get_fs_byte(name+i):0; //的文件名字段中
193.                bh->b_dirt = 1;                               //置位相应缓冲块的修改标志
194.                *res_dir = de;                                //返回该目录项指针
195.                return bh;                                    //返回该缓冲块指针
196.            }
197.            de++;                                         //指向下一个目录项
198.            i++;                                          //目录项计数加 1
199.        }
200.        brelse(bh);                                       //释放该缓冲块
201.        return NULL;                                      //返回 NULL
202.    }
        //在指定目录下创建一个指定文件名的结点
203.    int minix_create(struct inode * dir,const char * name, int len, int mode,
204.        struct inode ** result)
205.        {
206.        struct inode * inode;
207.        struct buffer_head * bh;
208.        struct minix_dir_entry * de;
209.
210.        *result = NULL;
211.        if (!dir)                                         //如果指定目录 i 结点不存在，返回错误码
212.            return -ENOENT;
213.        inode = minix_new_inode(dir->i_dev);              //在指定目录所在设备申请一个 i 结点
```

```
214.        if (!inode) {                                   //如果申请该 i 结点失败
215.            iput(dir);                                  //则释放指定目录的 i 结点，返回错误码
216.            return -ENOSPC;
217.        }
218.        inode->i_mode = mode;                           //根据指定的属性模式设置该 i 结点的属性模式
219.        inode->i_dirt = 1;                              //置位该 i 结点的修改标志
220.        bh = minix_add_entry(dir,name,len,&de);         //在指定目录下添加一个新的目录项
221.        if (!bh) {                                      //如果失败
222.            inode->i_nlink--;                           //则将该目录 i 结点的链接计数减 1
223.            iput(inode);                                //释放该目录 i 结点
224.            iput(dir);                                  //释放新申请的 i 结点，返回错误码
225.            return -ENOSPC;
226.        }
227.        de->inode = inode->i_ino;                       //令该目录项的 i 结点号字段等于新申请的 i 结点号
228.        bh->b_dirt = 1;                                 //置位包含该目录项的缓冲区修改标志
229.        brelse(bh);                                     //释放该缓冲区
230.        iput(dir);                                      //释放指定目录的 i 结点
231.        *result = inode;                                //返回新申请的 i 结点
232.        return 0;                                       //返回 0
233.    }
        //在指定目录下创建一个指定文件名的特殊文件或普通文件结点
234.    int minix_mknod(struct inode * dir, const char * name, int len, int mode, int rdev)
235.    {
236.        struct inode * inode;
237.        struct buffer_head * bh;
238.        struct minix_dir_entry * de;
239.
240.        if (!dir)                                       //如果指定目录 i 结点不存在，返回错误码
241.            return -ENOENT;
242.            bh = minix_find_entry(dir,name,len,&de); //获取指定文件名的目录项
243.            if (bh) {                                    //如果指定文件名的目录项已经存在
244.            brelse(bh);                                 //释放包含该目录项的缓冲区
245.            iput(dir);                                  //释放指定目录 i 结点，返回错误码
246.            return -EEXIST;
247.        }
248.        inode = minix_new_inode(dir->i_dev);            //在指定目录所在设备申请一个 i 结点
249.        if (!inode) {                                   //如果申请该 i 结点失败
250.            iput(dir);                                  //则释放指定目录的 i 结点，返回错误码
251.            return -ENOSPC;
252.        }
253.        inode->i_uid = current->euid;                   //置该 i 结点的用户 id
254.        inode->i_mode = mode;                           //置该 i 结点的属性模式
255.        if (S_ISBLK(mode) || S_ISCHR(mode))             //如果要创建的是块设备文件或者是字符设备文件
256.            inode->i_rdev = rdev;                       //则置该 i 结点的设备号
257.        inode->i_mtime = inode->i_atime = CURRENT_TIME; //置该 i 结点的修改时间为当前时间
258.        inode->i_dirt = 1;                              //置位该 i 结点的修改标志
259.        bh = minix_add_entry(dir,name,len,&de);         //在指定目录下添加一个新的目录项
260.        if (!bh) {                                      //如果失败
261.            inode->i_nlink--;                           //则将该目录 i 结点的链接计数减 1
262.            iput(inode);                                //释放新申请的 i 结点
```

```
263.            iput(dir);                                  //释放该目录 i 结点，返回错误码
264.            return -ENOSPC;
265.        }
266.        de->inode = inode->i_ino;                       //令该目录项的 i 结点号字段等于新申请的 i 结点号
267.        bh->b_dirt = 1;                                 //置位包含该目录项的缓冲区修改标志
268.        brelse(bh);                                     //释放该缓冲区
269.        iput(dir);                                      //释放指定目录的 i 结点
270.        iput(inode);                                    //释放新申请的 i 结点
271.        return 0;                                       //返回 0
272.    }
        //在指定目录下创建指定目录名的目录结点
273.    int minix_mkdir(struct inode * dir, const char * name, int len, int mode)
274.    {
275.        struct inode * inode;
276.        struct buffer_head * bh, *dir_block;
277.        struct minix_dir_entry * de;
278.
279.        bh = minix_find_entry(dir,name,len,&de);        //获取指定目录名的目录项
280.        if (bh) {                                       //如果指定目录名的目录项已经存在
281.            brelse(bh);                                 //释放包含该目录项的缓冲区
282.            iput(dir);                                  //释放指定目录 i 结点，返回错误码
283.            return -EEXIST;
284.        }
285.        inode = minix_new_inode(dir->i_dev);            //在指定目录所在设备申请一个 i 结点
286.        if (!inode) {                                   //如果申请该 i 结点失败
287.            iput(dir);                                  //则释放指定目录的 i 结点，返回错误码
288.            return -ENOSPC;
289.        }
290.        inode->i_size = 32;                             //置该新 i 结点对应的文件长度为 32
291.        inode->i_dirt = 1;                              //置该 i 结点已修改标志
292.        inode->i_mtime = inode->i_atime = CURRENT_TIME;  //置该 i 结点修改时间为当前时间
293.        if (!(inode->i_data[0] = minix_new_block(inode->i_dev))) { //为该 i 结点申请一个新的逻辑块
294.                                                        //并将其赋给第一个直接块
295.            iput(dir);                                  //如果申请失败，则释放指定目录 i 结点
296.            inode->i_nlink--;                           //将该 i 结点连接计数减 1
297.            iput(inode);                                //释放该 i 结点，返回错误码
298.            return -ENOSPC;
299.        }
300.        inode->i_dirt = 1;                              //置位该 i 结点的修改标志
301.        if (!(dir_block = bread(inode->i_dev,inode->i_data[0]))) { //读取刚刚申请的逻辑块
302.            iput(dir);                                  //如果失败，则释放指定目录 i 结点
303.            inode->i_nlink--;                           //将该 i 结点连接计数减 1
304.            iput(inode);                                //释放该 i 结点，返回错误码
305.            return -ERROR;
306.        }
307.        de = (struct minix_dir_entry *) dir_block->b_data; //令 de 指向目录项数据块的第一个目录项
308.        de->inode=inode->i_ino;                         //置该目录项的 i 结点号字段为刚申请的 i 结点号
309.        strcpy(de->name,".");                           //置该目录项的名字字段为“.”
310.        de++;                                           //令 de 指向目录项数据块中的第 2 个目录项
311.        de->inode = dir->i_ino;                         //置该目录项 i 结点号字段为指定目录的 i 结点号
```

```
312.        strcpy(de->name,"..");                          //置该目录项的名字字段为“..”
313.        inode->i_nlink = 2;                             //置新申请的 i 结点连接计数为 2
314.        dir_block->b_dirt = 1;                          //置该目录项数据块缓冲区的修改标志
315.        brelse(dir_block);                              //释放该缓冲区
316.        inode->i_mode = I_DIRECTORY | (mode & 0777 & ~current->umask);//初始化设置新申请的 i
317.                                                        //结点模式字段
318.        inode->i_dirt = 1;                              //置位该 i 结点的修改标志
319.        bh = minix_add_entry(dir,name,len,&de);         //在指定目录下添加一个新的目录项
320.        if (!bh) {                                      //如果失败
321.              iput(dir);                                //释放该目录 i 结点
322.              inode->i_nlink=0;                         //则将该目录 i 结点的链接计数减 1
323.              iput(inode);                              //释放新申请的 i 结点，返回错误码
324.              return -ENOSPC;
325.        }
326.        de->inode = inode->i_ino;                       //令该目录项的 i 结点号字段等于新申请的 i 结点号
327.        bh->b_dirt = 1;                                 //置位包含该目录项的缓冲区修改标志
328.        dir->i_nlink++;                                 //将指定目录 i 结点的连接加 1
329.        dir->i_dirt = 1;                                //置位指定目录 i 结点的修改标志
330.        iput(dir);                                      //释放指定目录的 i 结点
331.        iput(inode);                                    //释放新申请的 i 结点
332.        brelse(bh);                                     //释放该缓冲区，返回 0
333.        return 0;
334.      }
      //检查指定目录是否为空目录
335.      static int empty_dir(struct inode * inode)
336.  {
337.        int nr,block;
338.        int len;
339.        struct buffer_head * bh;
340.        struct minix_dir_entry * de;
341.
342.        len = inode->i_size / sizeof (struct niinix_dir_entry); //获取指定目录中目录项的个数
343.        if (len<2 || !inode->i_data[0] ||                   //如果目录项个数小于 2，或者该目录 i 结点的第
344.              !(bh=bread(inode->i_dev,inode->i_data[0]))) { //一直接块中无逻辑块号，或者直接块指向
345.                                                          //的逻辑块读取失败
346.              printk("warning - bad directory on dev %04x\n",inode->i_dev); //则显示警告信息，返回 0
347.              return 0;
348.        }
349.        de = (struct minix_dir_entry *) bh->b_data;     //令 de 指向目录项数据块
350.        if (de[0].inode != inode->i_ino || !de[1].inode ||  //如果第一个目录项的 i 结点号字段不等于该
351.              strcmp(".",de[0].name) || strcmp("..",de[1].name)) {  //目录 i 结点号，或者第二个目录项 i
352.                                                          //结点号字段为 0，或者第一个目录项名字字段
                                                            //不为“.”，或者第二个目录项名字字段不为“..”
353.
354.              printk("warning - bad directory on dev %04x\n",inode->i_dev); //则打印警告信息，返回 0
355.              return 0;
356.        }
357.        nr = 2;                                         //nr 为目录项序号
358.        de += 2;                                        //的指向第三个目录项
359.        while (nr<len) {                                //遍历所有目录项，检测是否存在目录项的 i 结
```

```
                                                    //点号字段不为 0
360.            if ((void *) de >= (void *) (bh->b_data+BLOCK_SIZE)) { //如果该逻辑块中的目录项已检
361.                                                //测完
362.                brelse(bh);                     //则释放该逻辑块的缓冲区
363.                block=bmap(inode,nr/MINIX_DIR_ENTRIES_PER_BLOCK); //获取下一文件块
364.                if (!block) {                   //如果该文件块没有被使用
365.                    nr += MINIX_DIR_ENTRIES_PER_BLOCK; //则跳过该文件块，继续
366.                    continue;
367.                }
368.                if (!(bh=bread(inode->i_dev,block)))    //读取该文件块指向的逻辑块
369.                    return 0;                   //如果失败，返回 0
370.                de = (struct minix_dir_entry *) bh->b_data; //de 指向该逻辑块缓冲区
371.            }
372.            if (de->inode) {                    //如果该目录项 i 结点号字段不为 0
373.                brelse(bh);                     //则释放该目录项所在缓冲区，返回 0
374.                return 0;
375.            }
376.            de++;                               //否则，de 指向下一个目录项
377.            nr++;                               //目录项序号加 1
378.        }
          //执行到此，表明所有目录项都未被使用，即指定目录为空
379.        brelse(bh);                             //则释放缓冲区，返回 1
380.        return 1;
381.        }
        //删除指定目录下指定目录名的目录
382.        int minix_rmdir(struct inode * dir, const char * name, int len)
383.    {
384.        int retval;
385.        struct inode * inode;
386.        struct buffer_head * bh;
387.        struct minix_dir_entry * de;
388.
389.        inode = NULL;
390.        bh = minix_find_entry(dir,name,len,&de);    //获取指定目录下指定目录名的目录项
391.        retval = -ENOENT;                           //错误码赋值
392.        if (!bh)                                    //如果包含目录项的缓冲区不存在
393.            goto end_rmdir;                         //跳转到 end_rmdir
394.        retval = -EPERM;                            //错误码赋值
395.        if (!(inode = iget(dir->i_dev, de->inode)))     //获取该目录项对应的目录 i 结点
396.            goto end_rmdir;                         //如果失败，则跳转到 end_rmdir
397.        if ((dir->i_mode & S_ISVTX) && current->euid && //如果指定目录设置了受限制删除标志并且
398.            inode->i_uid != current->euid)          //进程的有效用户 id 不等于要被删除目录 i 节
399.                                                    //点的用户 id
400.            goto end_rmdir;                         //则跳转到 end_rmdir
401.        if (inode->i_dev != dir->i_dev)             //如果要被删除的目录 i 结点的设备号不等于指
                                                        //定目录的设备号
402.
403.            goto end_rmdir;                         //则跳转到 end_rmdir
404.        if (inode == dir)
405.                                /*我们可以不删除“.”，但“../dir”是可以删除的*/
```

```
406.            goto end_rmdir;                          //则跳转到 end_rmdir
407.        if (!S_ISDIR(inode->i_mode)) {               //如果要被删除的目录 i 结点不是一个目录
408.            retval = -ENOTDIR;                       //则错误码赋值
409.            goto end_rmdir;                          //跳转到 end_rmdir
410.        }
411.        if (!empty_dir(inode)) {                     //如果要被删除的目录不为空
412.            retval = -ENOTEMPTY;                     //则错误码赋值
413.            goto end_rmdir;                          //跳转到 end_rmdir
414.        }
415.        if (inode->i_count > 1) {                    //如果要被删除的目录 i 结点还有其他进程引用
416.            retval = -EBUSY;                         //则错误码赋值
417.            goto end_rmdir;                          //则跳转到 end_rmdir
418.        }
419.        if (inode->i_nlink != 2)                     //如果要被删除的目录 i 结点链接计数不等于 2
420.            printk("empty directory has nlink!=2 (%d)\n",inode->i_nlink);  //打印警告信息
421.        de->inode = 0;                               //要被删除的目录的目录项 i 结点号字段赋 0
422.        bh->b_dirt = 1;                              //置位包含该目录项的缓冲区修改标志
423.        inode->i_nlink=0;                            //置要被删除的目录 i 结点连接计数为 0
424.        inode->i_dirt=1;                             //置位该目录 i 结点的修改标志
425.        dir->i_nlink--;                              //指定的目录 i 结点连接计数减 1
426.        dir->i_ctime = dir->i_mtime = CURRENT_TIME; //置该 i 结点的修改时间为当前时间
427.        dir->i_dirt=1;                               //置位该 i 结点的修改标志
428.        retval = 0;                                  //置错误码为 0
429.    end_rmdir:
430.        iput(dir);                                   //释放指定目录 i 结点
431.        iput(inode);                                 //释放要被删除的目录 i 结点
432.        brelse(bh);                                  //释放包含要被删除的目录的目录项的缓冲区
433.        return retval;                               //返回错误码
434.        }
    //删除指定目录下指定文件名的文件
435.        int minix_unlink(struct inode * dir, const char * name, int len)
436.    {
437.        int retval;
438.        struct inode * inode;
439.        struct buffer_head * bh;
440.        struct minix_dir_entry * de;
441.
442.        retval = -ENOENT;
443.        inode = NULL;
444.        bh = minix_find_entry(dir,name,len,&de);     //获取指定目录下指定文件名的目录项
445.        if (!bh)                                     //如果包含目录项的缓冲区不存在
446.            goto end_unlink;                         //跳转到 end_unlink
447.        if (!(inode = iget(dir->i_dev, de->inode)))  //获取该目录项对应的文件 i 结点
448.            goto end_unlink;                         //如果失败，则跳转到 end_unlink
449.        retval = -EPERM;                             //错误码赋值
450.        if ((dir->i_mode & S_ISVTX) && !suser() &&   //如果指定目录设置了受限制删除标志并且
451.            current->euid != inode->i_uid &&         //进程的有效用户 id 不等于要被删除文件 i 节
452.            current->euid != dir->i_uid)             //点的用户 id
453.            goto end_unlink;                         //则跳转到 end_unlink
454.        if (S_ISDIR(inode->i_mode))                  //如果要被删除的文件是一个目录
```

```
455.            goto end_unlink;                        //则跳转到 end_unlink
456.        if (!inode->i_nlink) {                      //如果要被删除的文件 i 结点连接计数为 0
457.            printk("Deleting nonexistent file (%04x:%d), %d\n", //则打印警告信息
458.                inode->i_dev,inode->i_ino,inode->i_nlink);
459.            inode->i_nlink=1;                       //置该文件 i 结点的连接计数为 1
460.        }
461.        de->inode = 0;                              //为要被删除的文件的目录项 i 结点号字段赋 0
462.        bh->b_dirt = 1;                             //置位包含该目录项的缓冲区修改标志
463.        inode->i_nlink--;                           //将要被删除的文件 i 结点连接计数减 1
464.        inode->i_dirt = 1;                          //置位该文件 i 结点的修改标志
465.        inode->i_ctime = CURRENT_TIME;              //置该 i 结点的修改时间为当前时间
466.        retval = 0;                                 //置错误码为 0
467.    end_unlink:
468.        brelse(bh);                                 //释放包含要被删除的文件的目录项的缓冲区
469.        iput(inode);                                //释放要被删除的文件 i 结点
470.        iput(dir);                                  //释放指定目录 i 结点
471.        return retval;                              //返回错误码
472.    }
    //创建一个符号连接文件
473.    int minix_symlink(struct inode * dir, const char * name, int len, const char * symname)
474.    {
475.        struct minix_dir_entry * de;
476.        struct inode * inode = NULL;
477.        struct buffer_head * bh = NULL, * name_block = NULL;
478.        int i;
479.        char c;
480.
481.        if (!(inode = minix_new_inode(dir->i_dev))) { //在指定目录所在设备上申请一个 i 结点
482.            iput(dir);                              //如果失败，则释放指定目录 i 结点
483.            return -ENOSPC;                         //返回错误码
484.        }
485.        inode->i_mode = S_IFLNK | 0777;             //置新申请的 i 结点属性模式
486.        inode->i_dirt = 1;                          //置位该 i 结点的修改标志
487.        if (!(inode->i_data[0] = minix_new_block(inode->i_dev))) { //为该 i 结点的第一个直接块申请
488.                                                    //一个逻辑块
489.            iput(dir);                              //如果失败，释放指定目录 i 结点
490.            inode->i_nlink--;                       //该 i 结点的连接计数减 1
491.            iput(inode);                            //释放该 i 结点
492.            return -ENOSPC;                         //返回错误码
493.        }
494.        inode->i_dirt = 1;                          //置位该 i 结点的修改标志
495.        if (!(name_block = bread(inode->i_dev,inode->i_data[0]))) {//读取刚申请的逻辑块到缓冲区中
496.            iput(dir);                              //如果失败，则释放指定目录的 i 结点
497.            inode->i_nlink--;                       //该 i 结点的连接计数减 1
498.            iput(inode);                            //释放该 i 结点
499.            return -ERROR;                          //返回错误码
500.        }
501.        i = 0;
502.        while (i < 1023 && (c=get_fs_byte(symname++))) //将指定的符号连接目标文件名复制到该
503.            name_block->b_data[i++] = c;            //缓冲区中
```

```
504.        name_block->b_data[i] = 0;                  //将缓冲区中的符号连接目标文件名后一字符置 0
505.        name_block->b_dirt = 1;                     //置位该缓冲区的修改标志
506.        brelse(name_block);                         //释放该缓冲区
507.        inode->i_size = i;                          //置该 i 结点的大小为 i
508.        inode->i_dirt = 1;                          //置位该 i 结点的修改标志
509.        bh = minix_find_entry(dir,name,len,&de);    //在指定目录下查找指定文件名的目录项
510.        if (bh) {                                   //如果目录项已存在
511.                inode->i_nlink--;                   //则将新申请的 i 结点连接计数减 1
512.                iput(inode);                        //释放该 i 结点
513.                brelse(bh);                         //释放包含该目录项的缓冲区
514.                iput(dir);                          //释放指定目录的 i 结点
515.                return -EEXIST;                     //返回错误码
516.        }
517.        bh = minix_add_entry(dir,name,len,&de);     //在指定目录下创建指定文件名的目录项
518.        if (!bh) {                                  //如果创建失败
519.                inode->i_nlink--;                   //则将新申请的 i 结点连接计数减 1
520.                iput(inode);                        //释放该 i 结点
521.                iput(dir);                          //释放指定目录的 i 结点
522.                return -ENOSPC;                     //返回错误码
523.        }
524.        de->inode = inode->i_ino;                   //置该目录项的 i 结点号字段为新申请的 i 结点号
525.        bh->b_dirt = 1;                             //置位包含该目录项的缓冲区修改标志
526.        brelse(bh);                                 //释放该缓冲区
527.        iput(dir);                                  //释放指定目录的 i 结点
528.        iput(inode);                                //释放新申请的 i 结点
529.        return 0;                                   //返回 0
530.    }
        //为一个已经存在的文件创建一个硬连接
531.    int minix_link(struct inode * oldinode, struct inode * dir, const char * name, int len)
532.    {
533.        struct minix_dir_entry * de;
534.        struct buffer_head * bh;
535.
536.        if (S_ISDIR(oldinode->i_mode)) {            //如果指定的原文件是一个目录文件
537.                iput(oldinode);                     //则释放该文件 i 结点
538.                iput(dir);                          //释放指定的目录 i 结点
539.                return -EPERM;                      //返回错误码
540.        }
541.        bh = minix_find_entry(dir,name,len,&de);    //在指定目录下查找指定文件名的目录项
542.        if (bh) {                                   //如果目录项已存在
543.                brelse(bh);                         //释放包含该目录项的缓冲区
544.                iput(dir);                          //释放指定目录 i 结点
545.                iput(oldinode);                     //释放原文件 i 结点
546.                return -EEXIST;                     //返回错误码
547.        }
548.        bh = minix_add_entry(dir,name,len,&de);     //在指定目录下创建指定文件名的目录项
549.        if (!bh) {                                  //如果创建失败
550.                iput(dir);                          //释放指定目录的 i 结点
551.                iput(oldinode);                     //释放原文件 i 结点
552.                return -ENOSPC;                     //返回错误码
```

```
553.        }
554.        de->inode = oldinode->i_ino;                    //置该目录项的 i 结点号字段为原文件 i 结点号
555.        bh->b_dirt = 1;                                 //置位包含该目录项的缓冲区修改标志
556.        brelse(bh);                                     //释放该缓冲区
557.        iput(dir);                                      //释放指定目录的 i 结点
558.        oldinode->i_nlink++;                            //原文件 i 结点连接计数加 1
559.        oldinode->i_ctime = CURRENT_TIME;               //置该 i 结点修改时间为当前时间
560.        oldinode->i_dirt = 1;                           //置位该 i 结点的修改标志
561.        iput(oldinode);                                 //释放该 i 结点
562.        return 0;                                       //返回 0
563.    }
        //判定指定文件 new 是否在指定目录 old 或其子目录下
564.      static int subdir(struct inode * new, struct inode * old)
565.      {
566.      unsigned short fs;
567.      int ino;
568.      int result;
569.
570.      __asm__("mov %%fs,%0":"=r" (fs));
571.      __asm__("mov %0,%%fs"::"r" ((unsigned short) 0x10));
572.      new->i_count++;                                   //new 文件的应用计数加 1
573.      result = 0;
574.      for (;;) {                                        //循环判断 new 文件是否在 old 目录或其子目录下
575.              if (new == old) {                         //如果 new 的 i 结点和 old 的 i 结点相同
576.                      result = 1;                       //则返回结果置 1
577.                      break;                            //跳出循环
578.              }
579.              if (new->i_dev != old->i_dev)             //如果 new 的 i 结点和 old 的 i 结点不在同一设备上
580.                      break;                            //则跳出循环
581.              ino = new->i_ino;                         //获取 new 的 i 结点号
582.              if (minix_lookup(new,"..",2,&new))        //寻找 new 的父目录 i 结点
583.                      break;                            //如果查找失败，则跳出循环
584.              if (new->i_ino == ino)                    //如果赋目录 i 结点号等于原来 new 的 i 结点号
585.                      break;                            //跳出循环
586.      }
587.      iput(new);                                        //释放 new 所指向的 i 结点
588.      __asm__("mov %0,%%fs"::"r" (fs));
589.      return result;                                    //返回判定结果
590.  }
      //定义查找父目录 i 结点号的函数宏
591.  #define PARENT_INO(buffer) \
592.      (((struct minix_dir_entry *) (buffer))[1].inode)
      //定义查找父目录名的函数宏
593.  #define PARENT_NAME(buffer) \
594.  (((struct minix_dir_entry *) (buffer))[1].name)
      //文件重命名函数
595.      static int do_minix_rename(struct inode * old_dir, const char * old_name, int old_len,
596.      struct inode * new_dir, const char * new_name, int new_len)
597.  {
598.      struct inode * old_inode, * new_inode;
```

```
599.        struct buffer_head * old_bh, * new_bh, * dir_bh;
600.        struct minix_dir_entry * old_de, * new_de;
601.        int retval;
602.        goto start_up;                                  //跳转到 start_up
603.       try_again:
604.        brelse(old_bh);                                 //释放包含指定原文件目录项的缓冲区
605.        brelse(new_bh);                                 //释放包含指定新文件目录项的缓冲区
606.        brelse(dir_bh);                                 //释放指定原文件的第一直接块的缓冲区
607.        iput(old_inode);                                //释放指定原文件 i 结点
608.        iput(new_inode);                                //释放指定新文件 i 结点
609.        current->counter = 0;                           //当前进程的引用计数置 0
610.        schedule();                                     //再次调度
611.    start_up:
612.        old_inode = new_inode = NULL;
613.        old_bh = new_bh = dir_bh = NULL;
614.        old_bh = minix_find_entry(old_dir,old_name,old_len,&old_de); //获取指定原文件的目录项
615.        retval = -ENOENT;                               //错误码赋值
616.        if (!old_bh)                                    //如果包含指定原文件目录项的缓冲区不存在
617.            goto end_rename;                            //跳转到 end_rename
618.        old_inode = iget(old_dir->i_dev, old_de->inode); //获取指定原文件的 i 结点
619.        if (!old_inode)                                 //如果指定原文件 i 结点不存在
620.            goto end_rename;                            //跳转到 end_rename
621.        new_bh = minix_find_entry(new_dir,new_name,new_len,&new_de); //获取指定新文件的目录项
622.
623.        if (new_bh) {                                   //如果包含指定新文件目录项的缓冲区存在
624.            new_inode = iget(new_dir->i_dev, new_de->inode); //获取指定新文件的 i 结点
625.            if (!new_inode) {                           //如果指定新文件 i 结点不存在
626.                brelse(new_bh);                         //释放包含指定新文件目录项的缓冲区
627.                new_bh = NULL;                          //指向包含指定新文件目录项的缓冲区指针置 NULL
628.            }
629.        }
630.        if (new_inode == old_inode) {                   //如果指定原文件 i 结点和指定新文件 i 结点相等
631.            retval = 0;                                 //错误码置 0
632.            goto end_rename;                            //跳转到 end_rename
633.        }
634.        if (S_ISDIR(old_inode->i_mode)) {               //如果指定原文件是一个目录文件
635.            retval = -EEXIST;                           //错误码赋值
636.            if (new_bh)                                 //如果包含新文件目录项的缓冲区存在
637.                goto end_rename;                        //则跳转到 end_rename
638.            retval = -EACCES;                           //错误码赋值
639.            if (!permission(old_inode, MAY_WRITE)) //如果指定原文件不允许写操作
640.                goto end_rename;                        //则跳转到 end_rename
641.            retval = -EINVAL;                           //错误码赋值
642.            if (subdir(new_dir, old_inode))             //如果指定新目录是指定原文件目录的子目录
643.                goto end_rename;                        //则跳转到 end_rename
644.            retval = -EIO;                              //错误码赋值
645.            if (!old_inode->i_data[0])                  //如果指定原文件目录的第一直接块不存在
646.                goto end_rename;                        //则跳转到 end_rename
647.            if (!(dir_bh = bread(old_inode->i_dev, old_inode->i_data[0]))) //读取原文件目录的第一直
648.                                                        //接块对应的逻辑块
```

```
649.                goto end_rename;                    //如果失败，则跳转到 end_rename
650.            if (PARENT_INO(dir_bh->b_data) != old_dir->i_ino) //如果指定原文件目录的父目录号不
651.                                                    //等于指定原目录的目录号
652.                goto end_rename;                    //则跳转到 end_rename
653.        }
654.        if (!new_bh)                                //如果包含指定新文件目录项的缓冲区不存在
655.            new_bh = minix_add_entry(new_dir,new_name,new_len,&new_de); //则在指定新目录下
656.                                                    //创建一个指定新文件名的目录项
657.        retval = -ENOSPC;                           //错误码赋值
658.        if (!new_bh)                                //如果包含指定新文件目录项的缓冲区不存在
659.            goto end_rename;                        //则跳转到 end_rename
        //在开始改名操作之前进行完整性检查——避免竞争
660.        if (new_inode && (new_de->inode != new_inode->i_ino)) //如果指定新文件的 i 结点存在，但
661.    //指定新文件目录项的结点号字段不等于指定新文件的 i 结点的 i 结点号字段
662.            goto try_again;                         //则跳转到 try_again
663.        if (new_de->inode && !new_inode)            //如果指定新文件目录项 i 结点号存在，但指定新文
664.                                                    //件 i 结点不存在
665.            goto try_again;                         //则跳转到 try_again
666.        if (old_de->inode != old_inode->i_ino)      //如果指定原文件目录项的结点号字段不等于指定原
667.                                                    //文件的 i 结点的 i 结点号字段
668.            goto try_again;                         //则跳转到 try_again
        //开始重命名操作
669.        old_de->inode = 0;                          //指定原文件目录项的 i 结点号置 0
670.        new_de->inode = old_inode->i_ino;           //指定新文件目录项的 i 结点号置为指定原文件的 i 结
                                                        //点号
671.        if (new_inode)                              //如果指定新文件的 i 结点存在
672.            new_inode->i_nlink--;                   //该 i 结点的连接计数减 1
673.        old_bh->b_dirt = 1;                         //置位包含原文件目录项的缓冲区修改标志
674.        new_bh->b_dirt = 1;                         //置位包含新文件目录项的缓冲区修改标志
675.        if (dir_bh) {                               //如果原文件第一直接块对应的逻辑块的缓冲块存在
676.            PARENT_INO(dir_bh->b_data) = new_dir->i_ino;  //则将原文件父目录目录项的 i 结点号
677.                                                    //置为指定新目录的 i 结点号
678.            dir_bh->b_dirt = 1;                     //置位该缓冲区的修改标志
679.            old_dir->i_nlink--;                     //指定原目录的连接计数减 1
680.            new_dir->i_nlink++;                     //指定新目录的连接计数加 1
681.            old_dir->i_dirt = 1;                    //置位指定原目录的 i 结点修改标志
682.            new_dir->i_dirt = 1;                    //置位指定新目录的 i 结点修改标志
683.        }
684.        retval = 0;                                 //错误码赋值
685.    end_rename:
686.        brelse(dir_bh);                             //释放指定原文件的第一直接块的缓冲区
687.        brelse(old_bh);                             //释放包含指定原文件目录项的缓冲区
688.        brelse(new_bh);                             //释放包含指定新文件目录项的缓冲区
689.        iput(old_inode);                            //释放指定原文件 i 结点
690.        iput(new_inode);                            //释放指定新文件 i 结点
691.        iput(old_dir);                              //释放指定原目录 i 结点
692.        iput(new_dir);                              //释放指定新目录 i 结点
693.        return retval;                              //返回错误码
694.    }
        /调用文件重命名函数的互斥锁函数
```

```
695.    int minix_rename(struct inode * old_dir, const char * old_name, int old_len,
696.        struct inode * new_dir, const char * new_name, int new_len)
697.        {
698.        static struct task_struct * wait = NULL;
699.        static int lock = 0;
700.        int result;
701.        while (lock)                                    //如果文件重命名函数正在被调用
702.                sleep_on(&wait); //则将当前进程加入到等待队列
703.        lock = 1;                                       //上锁
704.        result = do_minix_rename(old_dir, old_name, old_len,
705.                new_dir, new_name, new_len);    //调用文件重命名函数
706.        lock = 0;                                       //解锁
707.        wake_up(&wait);                                 //唤醒在等待队列上的进程
708.        return result;                                  //返回结果
709.    }
        //读取符号连接目标文件名函数
710.        int minix_readlink(struct inode * inode, char * buffer, int buflen)
711.        {
712.        struct buffer_head * bh;
713.        int i;
714.        char c;
715.        if (!S_ISLNK(inode->i_mode)) {          //如果指定文件不是符号连接文件
716.                iput(inode);                            //释放该文件 i 结点
717.                return -EINVAL;                         //返回错误代码
718.        }
719.        if (buflen > 1023)                              //如果缓冲区的长度大于 1023
720.                buflen = 1023;                          //则重置该长度为 1023
721.        if (inode->i_data[0])                           //如果指定文件的第一个直接块对应的逻辑块号不为 0
722.                bh = bread(inode->i_dev, inode->i_data[0]); //则读取该逻辑块
723.        else                                            //否则，bh 置为 NULL
724.                bh = NULL;
725.        iput(inode);                                    //释放指定文件 i 结点
726.        if (!bh)                                        //如果读取逻辑块的缓冲区不存在
727.                return 0;                               //返回 0
728.        i = 0;
729.        while (i<buflen && (c = bh->b_data[i])) {  //将符号连接目标文件名复制到指定缓冲区中
730.                i++;
731.                put_fs_byte(c,buffer++);
732.        }
733.        brelse(bh);                                     //释放读取逻辑块的缓冲区
734.        return i;                                       //返回符号连接目标文件名长度
735.    }
```

源代码中第 17 行提示如果你想让文件名长度大于 MINIX_NAME_LEN 的字符被截断，就注释掉这一行。否则这将是不被允许的。

第 44 行为 NO_TRUNCATE 的宏定义。

第 13～32 行定义了比较函数 minix-match()，这里不能用 strncmp 函数，因为该函数不在我们的数据空间内，因此将必须使用 minix-match()。该函数同样做一些完整的比较测试。

第 33～81 行定义了 minix-find-entry()函数，该函数在指定的目录中寻找与想要的名字匹配的

目录项。返回含有找到目录项的高速缓冲区，以及目录想本身。该函数并不读取目录项的 i 结点，如果想要该 i 结点将必须自己实现。

第 142～202 行：minix_add_entry()用和 minix_find_entry()相同的方式，向一个指定的目录中添加目录项，如果失败则返回 NULL。注意，de 的 i 结点部分被设置为 0——意味着在调用这个函数和向目录项中添加信息之间不能睡眠，因为当睡眠时其他进程可能会使用它。

第 595～694 行定义的 do-minix-rename()函数利用重复检测避免条件竞争：至少能使条件竞争最小化。它试图分配所有的块，然后完整性检测，如果完整性检测失败，它会试图使自己重新再来。

第 695～709 行定义的重命名函数的互斥锁函数主要是为了文件重命名时进行互斥。rename 也会锁住其他的 rename，因为它们能够改变一个目录的父目录，并且我们不想要任何的竞争。其他的竞争已经被 do_rename()检查了，如果有问题的话，它会重新开始。注意在两个不同的文件系统之间没有竞争：竞争只存在于同一设备当中：多个 rename 可以同时发生，只有它们是在不同的分区内。

8.22 Minix 的 file_dev.c 文件

该文件实现了对 Minix 文件系统正规文件的访问功能的定义。

本程序主要实现了对正规文件数据的读出和写入。其中的函数主要供系统调用函数 read()和 write()调用。函数 minix_file_read()从指定文件中指定的位置读取指定长度的数据到指定的用户缓冲区中；函数 minix_file_write()从指定的用户缓冲区写入指定长度的数据到指定文件的指定位置。file_dev.c 文件的详细代码注释如下：

```
1.      #include <errno.h>                  //错误号头文件。包含系统中各种出错号
2.      #include <fcntl.h>                  //文件控制头文件。用于文件及其描述符的操作控制常数符号的
                                            //定义
3.      #include <linux/sched.h>            //调度程序头文件。定义了 task_struct 和任务 0 的数据。一些关
                                            //于描述符参数设置和获取以及 switch_to()的嵌入式汇编函数宏
4.      #include <linux/minix_fs.h>         //Minix 文件系统头文件。定义了与 Minix 文件系统相
                                            //关的函数和结构体
5.      #include <linux/kernel.h>           //内核头文件。定义了一些内核常用函数的原型
6.      #include <asm/segment.h>            //段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数
7.      #define MIN(a,b) (((a)<(b))?(a):(b))    //取最小值
8.      #define MAX(a,b) (((a)>(b))?(a):(b))    //取最大值
        //从指定文件中指定的位置读取指定长度的数据到指定的用户缓冲区中
9.      int minix_file_read(struct inode * inode, struct file * filp, char * buf, int count)
10.     {
11.             int read,left,chars,nr;
12.             struct buffer_head * bh;
13.             if (filp->f_pos > inode->i_size)    //如果当前读写位置超过指定文件的大小
14.             left = 0;                           //则实际需要读取的字节数置 0
15.       else
16.                 left = inode->i_size - filp->f_pos; //否则，实际需要读取的字节数置为可读取的字节数
17.             if (left > count)                   //如果实际需要读取的字节数大于指定需要读取的字
                                                    //节数
18.             left = count;                       //则实际需要读取的字节数置为指定需要读取的字节数
19.       read = 0;                                 //已读字节数初始化为 0
20.             while (left > 0) {                  //如果仍有数据要读，就继续循环操作
21.             if (nr = bmap(inode,(filp->f_pos)>>BLOCK_SIZE_BITS)) {    //取当前读写位置在设
```

```
22.                                                    //备上对应的逻辑块号
23.                     if (!(bh=bread(inode->i_dev,nr))) //如果该逻辑块号不为 0，则读取该逻辑块
24.                                                    //到缓冲区
25.                     return read?read:-EIO; //如果读取失败，则返回已读字节数，如果已
26.                                                    //读字节数为 0，返回错误码
27.             } else                                 //如果获取的逻辑块号为 0
28.                     bh = NULL;                     //缓冲区指针置为 NULL
29.                     nr = filp->f_pos & (BLOCK_SIZE-1);  //计算文件读写指针在缓冲块中的偏移值。
30.                     chars = MIN( BLOCK_SIZE-nr , left ); //该块中可读字节数与实际还需读取的字节数
31.                                                    //比较，较小值为本次需要读取的字节数
32.                     filp->f_pos += chars;          //读写文件指针前移本次要读取的字节数
33.                     left -= chars;                 //实际需要读取的字节数减去本次要读取的字节数
34.                     read += chars;                 //已读取的字节数加上本次要读取的字节数
35.                     if (bh) {                      //如果从设备上读取逻辑块到缓冲区成功
36.                     char * p = nr + bh->b_data;    //令 p 指向缓冲区开始读取数据的位置
37.                     while (chars-->0)              //将本次要读取的数据复制到用户缓冲区中
38.                             put_fs_byte(*(p++),buf++);
39.                             brelse(bh);            //释放该缓冲区
40.                     } else {                       //否则，向用户缓冲区复制 chars 个 0
41.                             while (chars-->0)
42.                                     put_fs_byte(0,buf++);
43.                             }
44.                     }
45.                     inode->i_atime = CURRENT_TIME;  //修改指定 i 结点的访问时间为当前时间
46.             return read;                           //返回已读取的字节数
47.     }
        //从指定的用户缓冲区写入指定长度的数据到指定文件的指定位置
48.             int minix_file_write(struct inode * inode, struct file * filp, char * buf, int count)
49.     {
50.             off_t pos;
51.                     int written,block,c;
52.                     struct buffer_head * bh;
53.                     char * p;
54.                     if (filp->f_flags & O_APPEND)  //如果是要在文件后写入数据
55.                     pos = inode->i_size;           //则将文件读写指针指向文件尾部
56.             else                                   //否则，就在当前文件读写指针指向的位置开始写入
57.                             pos = filp->f_pos;
58.                     written = 0;                   //初始化已写入字节数
59.             while (written<count) {                //只要已写入数据大小小于需要写入的数据大小，则
                                                       //继续循环
60.                     if (!(block = minix_create_block(inode,pos/BLOCK_SIZE))) { //取当前读写位置在设
61.                                                    //备上对应的逻辑块号
62.                             if (!written)          //如果失败，而已写入字节数为 0
63.                                     written = -ENOSPC;  //则已写入字节数置为错误码
64.                                     break;         //跳出循环
65.                             }
66.                             if (!(bh=bread(inode->i_dev,block))) { //否则，读取该逻辑块到缓冲区
67.                             if (!written)          //如果读取失败，而已写入字节数为 0
68.                                     written = -EIO;  //则已写入字节数置为错误码
69.                                     break;         //跳出循环
70.                             }
```

```
71.                 c = pos % BLOCK_SIZE;    //计算文件读写指针在缓冲块中的偏移值
72.                 p = c + bh->b_data;      //令 p 指向缓冲区开始写入数据的位置
73.                 bh->b_dirt = 1;          //置位该缓冲区的修改标志
74.                 c = BLOCK_SIZE-c;        //计算该缓冲区可写入的字节数
75.             if (c > count-written)       //如果该缓冲区可写入的字节数大于还需要写入的字
                                             //节数
76.                 c = count-written;       //则本次要写入的字节数为还需要写入的字节数
77.                 pos += c;                //读写文件指针前移本次要写入的字节数
78.             if (pos > inode->i_size) {   //如果读写文件指针超过文件大小
79.                 inode->i_size = pos;     //则重置文件大小
80.                     inode->i_dirt = 1;   //置位指定 i 结点的修改标志
81.                 }
82.             written += c;                //已写入的字节数加本次要读取的字节数
83.             while (c-->0)                //将本次要写入的数据复制到该缓冲区
84.                 *(p++) = get_fs_byte(buf++);
85.                 brelse(bh);              //释放该缓冲区
86.         }
87.         inode->i_mtime = CURRENT_TIME;   //置文件修改时间为当前时间
88.             if (!(filp->f_flags & O_APPEND)) {//如果此次写入不是从文件尾开始
89.             filp->f_pos = pos;           //则把文件读写指针指向当前读写位置
90.             inode->i_ctime = CURRENT_TIME;  //置该 i 结点修改时间为当前时间
91.                 inode->i_dirt = 1;       //置位指定 i 结点的修改标志
92.         }
93.         return written;                  //返回已写入的字节数
94.          }
```

源代码本节 48～94 行定义的写文件操作函数 minix-file-write()中，当多个进程同时进行写操作时，append 操作可能不能工作。

8.23 Minix 的 minix_op.c 文件

该文件主要定义了 Minix 文件系统中的结点操作结构体。该文件是 Minix 文件系统的基本构成单位。本程序主要包含了 Minix 文件系统的结点操作结构体和文件操作结构体。在对 Minix 文件系统进行操作时，实际上就是针对这两种结构体进行操作。

minix_op.c 文件的代码注释如下：

```
1.      #include <linux/fs.h>          //文件系统头文件。定义了有关文件系统的一些常数和结构
2.      #include <linux/minix_fs.h>    //Minix 文件系统头文件。定义了与 Minix 文件系统相关的函数
                                       //和结构体
      //定义 Minix 结点操作结构体
3.      struct inode_operations minix_inode_operations = {
4.        minix_create,
5.        minix_lookup,
6.        minix_link,
7.        minix_unlink,
8.        minix_symlink,
9.        minix_mkdir,
10.       minix_rmdir,
11.       minix_mknod,
12.       minix_rename,
```

```
13.         minix_readlink,
14.         minix_open,
15.         minix_release,
16.         minix_follow_link
17.     };
```

源代码中第 3～17 行定义了 Minix 节点操作结构体的数据结构方式。

第9章 内存管理

涉及 Linux 内核的内存管理的文件共有五个，分别是包含在 include 目录下的 linux/mm.h 和 asm/memory.h 头文件以及包含在 mm 目录下的 Makefile、memory.c 和 swap.c 文件。其中 memory.c 是内存页面管理的核心文件，用于内存的初始化操作、页目录和页表的管理和内核其他部分对内存的申请处理过程，swap.c 文件主要是完成内存缺页时的换入换出，本章主要学习 memory.c 文件和 swap.c 文件。表 9-1 所示为 Linux/mm 目录。

表 9-1 Linux/mm 目录

名　　称	大小（byte）	修 改 时 间
Makefile	1237	1992-03-06 08:56:44
memory.c	15303	1992-03-06 21:27:40
swap.c	5887	1992-03-07 0:43:18

9.1 总体功能描述

Linux 内存管理主要是通过分页机制来完成的，分页机制是应用程序通过软硬件协作来访问内存的一种方法，其基本原理是将整个线性空间划分成若干大小相等的页面，通常，在一个体系结构中是以 4KB 为一页的内存页面。程序申请使用内存时，就以页为单位进行内存分配。

9.1.1 内存分页机制管理

在 80x86 体系结构中，内存的分页机制管理是通过页目录表和页表所组成的二级页表来完成的，其中页目录表和页表的结构是一样的。页目录项用来寻址一个页表，而页表项用来寻址一页物理页。因此，当指定了一个页目录项和一个页表项，就可以唯一地确定所对应的物理页。在 Linux 0.95 内核中，所有进程都使用一个页目录表，而每个进程都有自己的页表。

当使用 80x86 时，需要区分三个不同的地址：

- 逻辑地址：逻辑地址又称虚拟地址，这种地址在分段结构中常被用到，一个逻辑地址是由段基址和偏移量组成；
- 线性地址：线性地址是经过分段之后转化而来的用于分页之前的地址；
- 物理地址：物理地址是实际用于内存芯片的单元寻址，与处理器和 CPU 连接的地址总线相对应。

从 80386 开始，80x86 处理器都支持分页，它通过 CR0 寄存器的 PG 标志位启用。当 PG=1 时，启用分页机制，处理器通过分页机制实现线性地址到物理地址的转换；当 PG=0 时，禁用分页机制，线性地址直接可用作物理地址。为了使用分页机制，一个 32 位的线性地址被分成了三个部分，分别用来指定一个页目录项、一个页表项和对应物理内存页上的偏移地址，从而能间接地寻址到线性地址指定的物理内存位置。线性地址的位 31～22 共 10 个比特用来确定页目录中的目录项，页目录表占用一页内存，因此最多可以寻址 1024 个页表。位 21～12 也是 10 个比特，用来寻址页目录项指定的页表中的页表项，它也同样占用一页内存，因此一个页表可以寻址最多 1024 个物理内存页面。最后的 12 个比特正好用作页表项指定的一页物理内存中的偏移地址。

这样在 80386 中，一个页目录表所寻址的所有页表共可以寻址 1024 *1024 *4096 = 232 个存储单元，这也就是为什么是 32 位地址存储空间是 4GB。

在内存管理的函数中，大量使用了从线性地址到实际物理地址的变换计算。对于给定一个进程的线性地址，通过图 9-1 中所示的地址变换关系，我们可以很容易地找到该线性地址对应的实际物理内存页的地址。反之，如果需要从一个已知被使用的物理内存页地址，寻找对应的线性地址，则需要对整个页目录表和所有页表进行搜索。若该物理内存页被共享，我们就可能会找到多个对应的线性地址来。

每个页表项对应的物理内存页在 4GB 的地址范围内是随机的，是由页表项中页框地址内容确定的，是由内存管理程序利用设置页表项确定的。每个表项由页框地址、访问标志位、脏（已改写）标志位和存在标志位等构成。其结构可如图 9-2 所示。

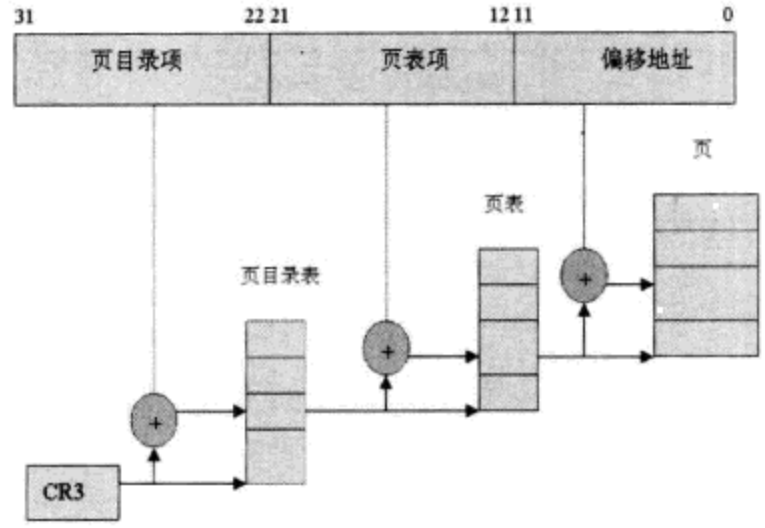


图 9-1　线性地址的变换

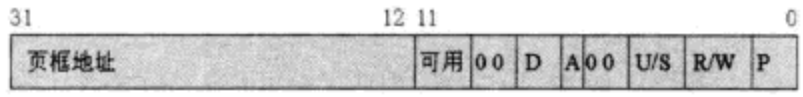


图 9-2　页目录项和页表项结构

页目录项和页表项有相同的结构，每项都包含下列字段：

- 页框地址高 20 位字段：指定了一页内存的物理起始地址。由于每个页框有 4KB 的容量，因此其物理地址必须是 4096 的倍数，所以其低 12 比特总是 0。因此表项的低 12 比特可作他用。如果在一个页目录表中，表项的页框地址是一个页表的起始地址；如果在一个页表项中，表项的页框就含有一页数据。
- 存在位（P）标志：确定了一个页表项是否可以用于地址转换过程。P=1 表示该项可用。当目录表项或第二级表项的 P=0 时，则该表项是无效的，不能用于地址转换过程。此时该表项的所有其他比特位都可供程序使用；处理器不对这些位进行测试。当 CPU 试图使用一个页表项进行地址转换时，如果此时任意一级页表项的 P=0，则处理器就会发出页异常信号。此时缺页中断异常处理程序就可以把所请求的页加入到物理内存中，并且导致异常的指令会被重新执行。
- 已访问位（Accessed – A）标志：在对一页内存进行读或写操作之前，CPU 将设置相关的目录和二级页表项的已访问位。当选中的页被交换出去时，这一标志由操作系统使用，分页单元不重置这个标志。
- 已修改位（Dirty – D）标志：在向一个二级页表项所涵盖的地址进行写操作之前，处理器将设置该二级页表项的已修改位，而页目录项中的已修改位是不用的。
- 读/写位（Read/Write – R/W）标志：含有页或页表的存取权限。
- 用户/超级用户位（User/Supervisor – U/S）标志：含有访问页或页表所需的特权级。

9.1.2 Linux 中内存的使用

对于 Linux 0.95 内核，为了有效地利用内存，系统开始把内存划分成若干个区域：内核区域、高速缓冲、虚拟盘、主内存区。其中 Linux 内核占开始的 16MB 物理内存，然后是供软盘或硬盘使用的块设备使用的高速缓冲区，它的最高内存地址是 4MB。

剩余的内存部分就是由内存管理程序进行分配和管理的主内存区，如果需要使用内存区的空间时，就要向内存管理程序申请，内存管理程序以一页为单位进行分配。系统中如果存在 RAM 虚拟盘，则从主内存区的前段扣除虚拟盘需要的空间。

9.2 memory.c 文件

memory.c 文件是 Linux 内存管理中最核心的文件，其中定义了 Linux 内核中内存管理的主要方法，完成了对内存的动态分配和回收以及页面异常的处理。

9.2.1 功能描述

实现内存管理的函数主要有以下几个：free_page()用来回收内存页，参数是物理地址。该函数会找到物理地址所对应的 mem_map 数组的下标，然后对下标对应项目值进行递减改变。

copy_page_tables()来把父进程的页表复制给子进程。该函数中利用 from 和 to 的线性地址计算出源目录项和目标目录项的位置，这样就可以得到原页表项。然后为 to 申请一个新的物理内存页，将该物理页地址设置到 to 的页表项，将页表项地址设置到 to 的目录项中。最后将源页表项复制到新申请的物理页并将页表项置为只读。如果 from 不在内核内存区，还会将源页表项置为只读。

get_empty_page()调用 get_free_page()和 put_page()先取得一个物理页再把取得的物理页和线性地址建立起对应关系。

free_page_tables()函数利用参数计算出要释放几个页、页表在页目录中的位置后，将页表中的 1k 个物理页调用 free_page()进行释放，最后清除所对应的页目录项。

put_page()函数通过参数所给的线性地址算出对应的页目录项和页表项，然后将物理页地址存入页表项，页表项地址存入目录项，以完成从线性地址到物理页地址的映射。

do_wp_page()是处理页面异常的函数 do_page_fault()调用的页写保护处理函数。它首先判断地址是否在进程的代码区域，若是则终止程序；然后执行写时复制页面的操作。

do_no_page()是处理页面异常的函数 do_page_fault()调用的由于缺页引起的页异常函数。首先判断指定的线性地址在一个进程空间中相对于进程基址的偏移长度值。如果它大于代码加数据长度，或者进程刚开始创建，则立刻申请一页物理内存，并映射到进程线性地址中，然后返回；接着尝试进行页面共享操作，若成功，则立刻返回；否则申请一页内存并从设备中读入一页信息。

do_page_fault()处理页面异常的函数，主要功能的实现是通过调用 do_wp_page()和 do_no_page()函数来实现。

9.2.2 代码分析

memory.c 文件的详细代码注释如下：

```
1.  #include <signal.h>
2.  #include <asm/system.h>
3.  #include <linux/sched.h>
```

```
4.    #include <linux/head.h>
5.    #include <linux/kernel.h>
       //判断给定地址是否位于当前进程的代码段中
6.    #define CODE_SPACE(addr) ((((addr)+4095)&~4095) < \
7.    current->start_code + current->end_code)
8.    unsigned long HIGH_MEMORY = 0;    // 放实际物理内存最高端地址
      //复制一页内存
9.        #define copy_page(from,to) \
10. __asm__("cld ; rep ; movsl"::"S" (from),"D" (to),"c" (1024):"cx","di","si")
11. 47   #define CHECK_LAST_NR   16
12. static unsigned long last_pages[CHECK_LAST_NR] = { 0, };
      //内存映射字节图，每个页面对应的字节用于标志页面当前被占用次数
13.   unsigned char mem_map [ PAGING_PAGES ] = {0,};
14.   void free_page(unsigned long addr)
15.   {//释放物理地址 addr 开始的一页内存，用于函数 free_page_tables()
16.   if (addr < LOW_MEM) return;     //如果物理地址 addr 小于低端地址，则返回
17.   if (addr < HIGH_MEMORY) {       //如果物理地址 addr 小于高端地址
18.   addr -= LOW_MEM;                //物理地址减去低端地址
19.   addr >>= 12;                    //然后再除以 4KB，得页面号
20.   if (mem_map[addr]--)            //如果对应内存映射字节不等于零，则减 1，返回
21.   return;
22.   mem_map[addr]=0;                //否则置对应页面映射字节为零
23.   }
24.   printk("trying to free free page: memory probably corrupted");
25.   }
26. int free_page_tables(unsigned long from,unsigned long size)
27. {//该函数用来释放页表连续的内存块，该函数被 exit()需要
      //与 copy_page_tables()函数类似，该函数仅处理 4MB 内存块
28.   unsigned long page;
29.   unsigned long page_dir;
30.   unsigned long *pg_table;
31.   unsigned long * dir, nr;
32.   if (from & 0x3fffff)            // 要释放内存块的地址需以 4MB 为边界
33.   panic("free_page_tables called with wrong alignment");
34.   if (!from)                      //出错，试图释放内核和缓冲所占空间
35.   panic("Trying to free up swapper memory space");
36.   size =(size + 0x3fffff) >>22;   //计算所占页目录项数（4MB 的进位整数倍），也即所占页表数
37.   dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */ //计算起始目录项
38.   for ( ; size-->0 ; dir++) {     // size 现在是需要被释放内存的目录项数
39.   if (!(page_dir = *dir))
40.   continue;
41.   *dir = 0;
42.   if (!(page_dir & 1)) {          //如果该目录项无效，打印出错误信息，并继续
43.   printk("free_page_tables: bad page directory.");
44.   continue;
45.   }
46.   pg_table = (unsigned long *) (0xfffff000 & page_dir);// 取目录项中页表地址
47.   for (nr=0 ; nr<1024 ; nr++,pg_table++) {//每个页表有 1024 个页表项
48.   if (!(page = *pg_table))
49.   continue;
```

```
50.   *pg_table = 0; //页表项先复位
51.   if (1 & page)
52.   free_page(0xfffff000 & page);          // 若该页表项有效（P 位=1），则释放对应内存页
53.   else
54.   swap_free(page >> 1);                  //不在物理内存
55.   }
56.   free_page(0xfffff000 & page_dir);      // 释放该页表所占内存页面。但由于页表在物理地
                                             // 址 1M 以内，所以这句什么都不做
57.   }
58.   invalidate();                          //刷新页变换高速缓冲
59.   for (page = 0; page < CHECK_LAST_NR ; page++) // 指向页表中下一项
60.   last_pages[page] = 0;                  //清零
61.   return 0;
62.   }
63.   int copy_page_tables(unsigned long from,unsigned long to,long size)
64.   {
65.   unsigned long * from_page_table;
66.   unsigned long * to_page_table;
67.   unsigned long this_page;
68.   unsigned long * from_dir, * to_dir;
69.   unsigned long new_page;
70.   unsigned long nr;
      // 源地址和目的地址都需要是在 4MB 的内存边界地址上。否则出错，死机
71.   if ((from&0x3fffff) || (to&0x3fffff))
72.   panic("copy_page_tables called with wrong alignment");
      // 取得源地址和目的地址的目录项指针（from_dir 和 to_dir）
73.   from_dir = (unsigned long *) ((from>>20) & 0xffc); /* _pg_dir = 0 */
74.   to_dir = (unsigned long *) ((to>>20) & 0xffc);
      //计算要复制的内存块占用的目录项数
75.   size = ((unsigned) (size+0x3fffff)) >> 22;
      //对每个占用的页表依次进行复制操作
76.     for( ; size-->0 ; from_dir++,to_dir++) {
77.   if (*to_dir)// 如果目的目录项指定的页表已经存在，则出错，死机
78.   printk("copy_page_tables: already exist, "
79.   "probable memory corruption\n");
      //如果此源目录项未被使用，则不用复制对应页表，跳过
80.   if (!*from_dir)
81.   continue;
82.   if (!(1 & *from_dir)) {
83.   printk("copy_page_tables: page table swapped out, "
84.   "probable memory corruption");
85.   *from_dir = 0;
86.   continue;
87.   }
       //取当前源目录项中页表的地址
88.   from_page_table = (unsigned long *) (0xfffff000 & *from_dir);
      //为目的页表取一页空闲内存，如果返回是 0 则说明没有申请到空闲内存页面。返回值=-1，退出
89.   if (!(to_page_table = (unsigned long *) get_free_page()))
90.         return -1;
91.   //设置目的目录项信息。7 是标志信息，表示（Usr, R/W, Present）
```

```
92.  *to_dir = ((unsigned long) to_page_table) | 7;
     //针对当前处理的页表，设置需复制的页面数。如果是在内核空间，则仅需复制头 160 页（640KB）
     // 否则需要复制一个页表中的所有 1024 页面
93.  nr = (from==0)?0xA0:1024;
     //对于当前页表，开始复制指定数目 nr 个内存页面
94.  for ( ; nr-- > 0 ; from_page_table++,to_page_table++) {
95.  this_page = *from_page_table;          // 取源页表项内容
96.  if (!this_page)
97.  continue;
98.  if (!(1 & this_page)) {
99.  if (!(new_page = get_free_page()))
100. return -1;
101. read_swap_page(this_page>>1, (char *) new_page);
102. *to_page_table = this_page;           // 将该页表项复制到目的页表中
103. *from_page_table = new_page | (PAGE_DIRTY | 7);
104. continue;
105. }
     //复位页表项中 R/W 标志（置 0）
106. this_page &= ~2;
107. *to_page_table = this_page;           // 将该页表项复制到目的页表中
108. if (this_page > LOW_MEM) {
109. *from_page_table = this_page;        // 令源页表项只读
110. this_page -= LOW_MEM;
111. this_page >>= 12;
112. mem_map[this_page]++;
113. }
114. }
115. }
116. invalidate();                         // 刷新页变换高速缓冲
117. return 0;
118. }
119. static unsigned long put_page(unsigned long page,unsigned long address)
120. {
121. unsigned long tmp, *page_table;
     /* 注意!这里使用了页目录基址 _pg_dir=0 的条件*/
     //如果申请的页面位置低于 LOW_MEM 或超出系统实际含有内存高端 HIGH_MEMORY，则发出
     //警告
122. if (page < LOW_MEM || page >= HIGH_MEMORY) {
123. printk("put_page: trying to put page %p at %p\n",page,address);
124. return 0;
125. }
      // 如果申请的页面在内存页面映射字节图中没有置位，则显示警告信息
126. if (mem_map[(page-LOW_MEM)>>12] != 1) {
127. printk("mem_map disagrees with %p at %p\n",page,address);
128. return 0;
129. }
      // 计算指定地址在页目录表中对应的目录项指针
130. page_table = (unsigned long *) ((address>>20) & 0xffc);
     // 如果该目录项有效，即在内存中有指定的页表，则从中取得指定页表的地址
131. if ((*page_table)&1)
```

```
page_table = (unsigned long *) (0xfffff000 & *page_table);
else {
    //否则，申请空闲页面给页表使用，并在对应目录项中置相应标志 7（User, U/S, R/W）
if (!(tmp=get_free_page()))
return 0;
*page_table = tmp | 7;
page_table = (unsigned long *) tmp;//将页表的地址保存在 page_table 中
}
    // 在页表中设置指定地址的物理内存页面的页表项内容。每个页表共可有 1024 项（0x3ff）
page_table[(address>>12) & 0x3ff] = page | 7;
/* 不需要刷新页变换高速缓冲*/
return page;  //返回物理地址
}
unsigned long put_dirty_page(unsigned long page, unsigned long address)
{
unsigned long tmp, *page_table;
    /* 注意!这里使用了页目录基址_pg_dir=0 的条件 */
    /*如果申请的页面位置低于 LOW_MEM(1Mb)或超出系统实际含有内存高端 HIGH_MEMORY，
    则发出警告。*/
if (page < LOW_MEM || page >= HIGH_MEMORY)
printk("put_dirty_page: trying to put page %p at %p\n",page,address);
    // 如果申请的页面在内存页面映射字节图中没有置位，则显示警告信息
if (mem_map[(page-LOW_MEM)>>12] != 1)
printk("mem_map disagrees with %p at %p\n",page,address);
    // 计算指定地址在页目录表中对应的目录项指针
page_table = (unsigned long *) ((address>>20) & 0xffc);
    // 如果该目录项有效（P=1）（也即指定的页表在内存中），则从中取得指定页表的地址
//到 page_table
if ((*page_table)&1)
page_table = (unsigned long *) (0xfffff000 & *page_table);
    // 否则，申请空闲页面给页表使用，并在对应目录项中置相应标志 7（User, U/S, R/W）
    //然后将该页表的地址保存在 page_table 中
else {
if (!(tmp=get_free_page()))
return 0;
*page_table = tmp|7;
page_table = (unsigned long *) tmp;
}
    // 在页表中设置指定地址的物理内存页面的页表项内容。每个页表共可有 1024 项（0x3ff）
page_table[(address>>12) & 0x3ff] = page | (PAGE_DIRTY | 7);
    /* 不需要刷新页变换高速缓冲 */
    // 返回页面地址
return page;
}
//取消写保护页面函数。用于页异常中断过程中写保护异常的处理（写时复制）
void un_wp_page(unsigned long * table_entry)
{
unsigned long old_page;
unsigned long new_page = 0;
unsigned long dirty;
```

```
168.  repeat:
169.  old_page = *table_entry;
170.  dirty = old_page & PAGE_DIRTY;
171.  if (!(old_page & 1)) {
172.  if (new_page)
173.  free_page(new_page);
174.  return;
175.  }
176.  old_page &= 0xfffff000; // 取指定页表项内物理页面地址
       //如果 old_page >= HIGH_MEMORY，且新的一页申请成功，释放新页，打印出错误信息退出系统
177.  if (old_page >= HIGH_MEMORY) {
178.  if (new_page)
179.  free_page(new_page);
180.  printk("bad page address\n\r");
181.  do_exit(SIGSEGV);
182.  }
183.  if (old_page >= LOW_MEM && mem_map[MAP_NR(old_page)]==1) {
184.  *table_entry |= 2;
185.  invalidate();
186.  if (new_page)
187.  free_page(new_page);
188.  return;
189.  }
190.  if (!new_page) {
       //否则，在主内存区内申请一页空闲页面
191.  if (!(new_page=get_free_page()))
192.  oom();  //内存不够处理。
193.  goto repeat;
194.  }
       //将原页面内容复制到新页面，将指定页表项内容更新为新页面的地址并置可读写等标志
       //(U/S, R/W, P)，并释放原页面内存
195.  copy_page(old_page,new_page);
196.  *table_entry = new_page | dirty | 7;
197.  free_page(old_page);
198.  invalidate();
199.  }
200.  void do_wp_page(unsigned long error_code,unsigned long address)
201.  {
202.  if (address < TASK_SIZE)
203.  printk("\n\rBAD! KERNEL MEMORY WP-ERR!\n\r");
204.  if (address - current->start_code >= TASK_SIZE) {
205.  printk("Bad things happen: page error in do_wp_page\n\r");
206.  do_exit(SIGSEGV);
207.  }
208.  un_wp_page((unsigned long *)
209.  (((address>>10) & 0xffc) + (0xfffff000 &
210.  *((unsigned long *) ((address>>20) &0xffc)))));
211.  }
       //写页面验证。若页面不可写，则复制页面
212.  void write_verify(unsigned long address)
```

```
213.  {
214.  unsigned long page;
       //判断指定地址所对应页目录项的页表是否存在（P），若不存在（P=0）则返回
215.  if (!( (page = *((unsigned long *) ((address>>20) & 0xffc)) )&1))
216.  return;
       // 取页表的地址，加上指定地址的页面在页表中的页表项偏移值，得对应物理页面的页表项指针
217.  page &= 0xfffff000;
218.  page += ((address>>10) & 0xffc);
       // 如果该页面不可写（标志 R/W 没有置位），则执行共享检验和复制页面操作（写时复制）
219.  if ((3 & *(unsigned long *) page) == 1)
220.  un_wp_page((unsigned long *) page);
221.  return;
222.  }
223.  void get_empty_page(unsigned long address)
224.  {
225.  unsigned long tmp;
       // 若不能取得一空闲页面，或者不能将页面放置到指定地址处，则显示内存不够的信息
226.  if (!(tmp=get_free_page()) || !put_page(tmp,address)) {
227.   free_page(tmp);
228.  oom();
229.  }
230.  }
231.  static int try_to_share(unsigned long address, struct task_struct * p)
232.  {
233.  unsigned long from;
234.  unsigned long to;
235.  unsigned long from_page;
236.  unsigned long to_page;
237.  unsigned long phys_addr;
       // 求指定内存地址的页目录项
238.  from_page = to_page = ((address>>20) & 0xffc);
       // 计算进程 p 的代码起始地址所对应的页目录项
239.  from_page += ((p->start_code>>20) & 0xffc);
       // 计算当前进程中代码起始地址所对应的页目录项
240.  to_page += ((current->start_code>>20) & 0xffc);
       /* 在 from 处是否存在页目录？ */
       //  对 p 进程页面进行操作。取页目录项内容。如果该目录项无效（P=0），则返回。否则取该
       //  目录项对应页表地址保存在 from 中
241.  from = *(unsigned long *) from_page;
242.  if (!(from & 1))
243.  return 0;
244.  from &= 0xfffff000;
       // 计算地址对应的页表项指针值，并取出该页表项内容 放在 phys_addr
245.  from_page = from + ((address>>10) & 0xffc);
246.  phys_addr = *(unsigned long *) from_page;
247.   /* 页面干净并且存在吗？ */
       // 0x41 对应页表项中的 Dirty 和 Present 标志。如果页面不干净或无效则返回
248.  if ((phys_addr & 0x41) != 0x01)
249.  return 0;
       // 取页面的地址 phys_addr。如果该页面地址大于高端地址或小于内存低端（1M）也返回退出
```

```
250.  phys_addr &= 0xfffff000;
251.  if (phys_addr >= HIGH_MEMORY || phys_addr < LOW_MEM)
252.  return 0;
       //对当前进程页面进行操作，取页目录项内容并保存到 to。如果该目录项无效（P=0），
       //则取空闲页面，并更新 to_page 所指目录项
253.  to = *(unsigned long *) to_page;
254.  if (!(to & 1)) {
255.  if (to = get_free_page())
256.  *(unsigned long *) to_page = to | 7;
257.  else
258.  oom();
259.  }
       // 取对应页表地址保存 to，页表项地址保存在 to_page。如果对应的页面已经存在，则出错，死机
260.  to &= 0xfffff000;
261.  to_page = to + ((address>>10) & 0xffc);
262.  if (1 & *(unsigned long *) to_page)
263.  panic("try_to_share: to_page already exists");
       // 对 p 进程中页面置写保护标志（置 R/W=0 只读）。并且当前进程中的对应页表项指向它
264.  *(unsigned long *) from_page &= ~2;
265.  *(unsigned long *) to_page = *(unsigned long *) from_page;
266.  invalidate();//刷新页变换高速缓冲
       // 计算所操作页面的页面号，并将对应页面映射数组项中的引用递增 1
267.  phys_addr -= LOW_MEM;
268.  phys_addr >>= 12;
269.  mem_map[phys_addr]++;
270.  return 1;
271.  }
272.  static int share_page(struct inode * inode, unsigned long address)
273.  { // 共享页面。在缺页处理时看看能否共享页面。 返回 1 - 成功，0 - 失败
274.  struct task_struct ** p;
       // 如果只能单独执行（inode->i_count=1）或 inode 结点是不可执行的，则返回
275.  if (inode->i_count < 2 || !inode)
276.  return 0;
       //搜索任务数组中所有任务。寻找与当前进程可共享页面的进程，并尝试对指定地址的页面进行
       //共享
277.  for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
278.  if (!*p) // 如果该任务项空闲，则继续寻找
279.  continue;
280.  if (current == *p) // 如果就是当前任务，也继续寻找
281.  continue;
282.  if (address < LIBRARY_OFFSET) {
       // 如果执行进程的内存结点结构不等，也继续
283.  if (inode != (*p)->executable)
284.  continue;
285.  } else {
286.  if (inode != (*p)->library)
287.  continue;
288.  }
289.  if (try_to_share(address,*p)) // 尝试共享页面
290.  return 1;
```

```
291.  }
292.  return 0;
293.  }
       //页异常中断处理调用的函数，参数 error_code 是由 CPU 自动产生，address 是页面线性地址
       //由 do_page_fault 函数调用
294.  void do_no_page(unsigned long error_code,
295.  unsigned long address, struct task_struct *tsk)
296.  {
297.  static unsigned int last_checked = 0;
298.  int nr[4];
299.  unsigned long tmp;
300.  unsigned long page;
301.  int block,i;
302.  struct inode * inode;
303.
304.  for (i = 0; i < CHECK_LAST_NR; i++)
305.  if ((address & 0xfffff000) == last_pages[i]) {
306.  current->counter = 0;
307.  schedule();
308.  }
309.  last_checked++;
310.  if (last_checked >= CHECK_LAST_NR)
311.  last_checked = 0;
312.  last_pages[last_checked] = address & 0xfffff000;
313.  if (address < TASK_SIZE)
314.  printk("\n\rBAD!! KERNEL PAGE MISSING\n\r");
315.  if (address - tsk->start_code >= TASK_SIZE) {
316.  printk("Bad things happen: nonexistent page error in do_no_page\n\r");
317.  do_exit(SIGSEGV);
318.  }
319.  page = *(unsigned long *) ((address >> 20) & 0xffc);
320.  /* check the page directory: make a page dir entry if no such exists */
321.  if (page & 1) {//如果该页被占用，
       //取页表地址，加上指定地址的页面在页表中的页表项偏移值，得到对应物理页面的页表项指针
322.  page &= 0xfffff000;
323.  page += (address >> 10) & 0xffc;
324.  tmp = *(unsigned long *) page;
325.  if (tmp && !(1 & tmp)) {
326.  swap_in((unsigned long *) page);
327.  return;
328.  }
329.  } else {
330.  if (page)
331.  printk("do_no_page: bad page directory\n");
       // 如果把物理页面映射到指定线性地址的操作成功，就返回。否则就释放内存页，显示内存不够
332.  if (!(page = get_free_page()))
333.  oom();
334.  page |= 7;
335.  *(unsigned long *) ((address >> 20) & 0xffc) = page;
336.  }
```

```
337.  address &= 0xfffff000;                    // 取页面地址
      //首先算出指定线性地址在进程空间中相对于进程基址的偏移长度值
338.  tmp = address - tsk->start_code;
339.  if (tmp >= LIBRARY_OFFSET ) {
340.  inode = tsk->library;
341.  block = 1 + (tmp-LIBRARY_OFFSET) / BLOCK_SIZE;
342.  } else if (tmp < tsk->end_data) {
343.  inode = tsk->executable;
      // 首先计算缺页所在的数据块项
344.  block = 1 + tmp / BLOCK_SIZE;
345.  } else {
346.  inode = NULL;
347.  block = 0;
348.  }
349.  if (!inode) {
350.  get_empty_page(address);
351.  return;
352.  }
353.  if (tsk == current)
354.  if (share_page(inode,tmp))              // 如果尝试共享页面成功，则退出
355.  return;
       //取空闲页面，如果内存不够，则显示内存不够，终止进程
356.  if (!(page = get_free_page()))
357.  oom();
358.   // BLOCK_SIZE = 1024 字节，因此一页内存需要 4 个数据块。 根据 i 结点信息
      //取数据块在设备上的对应的逻辑块号
359.  for (i=0 ; i<4 ; block++,i++)
360.  nr[i] = bmap(inode,block);.
       // 读设备上一个页面的数据（4 个逻辑块）到指定物理地址 page 处
361.  bread_page(page,inode->i_dev,nr);
       //下面部分是对物理页面超出的部分进行清零处理
362.  i = tmp + 4096 - tsk->end_data;
363.  if (i>4095)
364.  i = 0;
365.  tmp = page + 4096;
366.  while (i-- > 0) {
367.  tmp--;
368.  *(char *)tmp = 0;
369.  }
370.  if (put_page(page,address))             // 如果把物理页面映射到指定线性地址的操作成功，返回
371.  return;
372.  free_page(page);                        // 否则就释放内存页，显示内存不够
373.  oom();
374.  }
       //物理内存初始化
       // start_mem：可用作分页处理的物理内存起始位置（已去除 RAMDISK 所占内存空间等）
       //end_mem：实际物理内存最大地址
375.  void mem_init(long start_mem, long end_mem)
376.  {
377.  int i;
```

```
swap_device = 0;
swap_file = NULL;
HIGH_MEMORY = end_mem;                  // 设置内存最高端
for (i=0 ; i<PAGING_PAGES ; i++)        // 将页面映射数组全置成 USED
mem_map[i] = USED;
i = MAP_NR(start_mem);                  // 计算可使用起始内存的页面号
end_mem -= start_mem;                   // 计算可分页处理的内存块大小
end_mem >>= 12;                         // 计算出可用于分页处理的页面数
while (end_mem-->0)                     // 将可用页面对应的页面映射数组清零
mem_map[i++]=0;
}
void show_mem(void)
{
int i,j,k,free=0,total=0;
int shared=0;
unsigned long * pg_tbl;
printk("Mem-info:\n\r");
for(i=0 ; i<PAGING_PAGES ; i++) {       //扫描内存页面映射数组
if (mem_map[i] == USED)                 //如果页面已经被占用，则继续
continue;
total++;
if (!mem_map[i])                        //如果没有被占用，记录空闲页面数
free++;
else
shared += mem_map[i]-1;                 //记录共享页面数
}
  //分别打印出记录信息
printk("%d free pages of %d\n\r",free,total);
printk("%d pages shared\n\r",shared);
k = 0;
  //扫描所有页目录项（除 0，1，2，3 项）
for(i=4 ; i<1024 ;) {
  //如果页目录项有效，则统计对应页表中有效页面数，并显示
if (1&pg_dir[i]) {
if (pg_dir[i]>HIGH_MEMORY) {
printk("page directory[%d]: %08X\n\r",
i,pg_dir[i]);
i++;
continue;
}
if (pg_dir[i]>LOW_MEM)
free++,k++;
pg_tbl=(unsigned long *) (0xfffff000 & pg_dir[i]);
for(j=0 ; j<1024 ; j++)
if ((pg_tbl[j]&1) && pg_tbl[j]>LOW_MEM)
if (pg_tbl[j]>HIGH_MEMORY)
printk("page_dir[%d][%d]: %08X\n\r",
i,j, pg_tbl[j]);
else
k++,free++;
```

```
425.  }
426.  i++;
427.  if (!(i&15) && k) {
428.  k++,free++;
429.  printk("Process %d: %d pages\n\r",(i>>4)-1,k);
430.  k = 0;
431.  }
432.  }
433.  printk("Memory found: %d (%d)\n\r",free-shared,total);
434.  }
435.  //页面异常处理函数
436.  void do_page_fault (unsigned long *esp, unsigned long error_code)
437.  {
438.  unsigned long address;
439.
       //取得引起页面异常的地址
440.  __asm__ ("movl %%cr2,%0":"=r" (address));
441.  if (!(error_code & 1)) {
       //由于缺页引起的页异常中断，调用 do_no_page（）函数
442.  do_no_page(error_code, address, current);
443.  return;
444.  } else {//由页写保护引起的异常，调用写保护处理函数
445.  do_wp_page(error_code, address);
446.  return;
447.  }
```

第 63 行～118 行复制指定线性地址和长度内存对应的页目录项和页表，从而被复制的页目录和页表对应的原物理内存区被共享使用。复制指定地址和长度的内存对应的页目录项和页表项，需申请页面来存放新页表，原内存区被共享；此后两个进程将共享内存区，直到有一个进程执行写操作时，才分配新的内存页。

第 119～142 行主要在内存中指定的地址处放入一页，它返回一页的物理地址，如果内存不够（在访问页表或页面时）返回零。

第 143 行～162 行置页面为脏。

第 200～207 行写共享页面处理函数，参数 error_code 是由 CPU 自动产生，address 是页面线性地址，写共享页面时，需复制页面。

第 208～210 行处理取消页面保护。参数指定页面在页表中的页表项指针,((address>>10) &0xffc)：计算指定地址的页面在页表中的偏移地址；(0xfffff000 &((address>>20) &0xffc))：取目录项中页表的地址值((address>>20) &0xffc)，计算页面所在页表的目录项指针，两者相加即得指定地址对应页面的页表项指针。

第 223～230 行取得一页空闲内存并映射到指定线性地址处，与 get_free_page()不同。get_free_page()仅是申请取得了主内存区的一页物理内存，而该函数不仅是获取到一页物理内存页面，还进一步调用 put_page()，将物理页面映射到指定的线性地址处。

第 231 行函数尝试对进程指定地址处的页面进行共享操作。同时还验证指定的地址处是否已经申请了页面，若是则出错，死机。返回 1-成功，0-失败。

9.3 swap.c 文件

虚拟内存系统能够把整个应用程序代码和数据，即完整的进程从磁盘上移出或移入磁盘。这种技术被称为交换（swapping），因为它是把一个进程同另一个进程进行了对调。也正是基于此，磁盘上为虚拟内存所保留的区域常常被称为交换区或交换空间。

1. 功能描述

swap.c 文件主要包括用于对交换设备进行操作的函数 rw_swap_page()和 get_swap_page() 、用来释放交换设备某页的函数 swap_free() 以及换入换出函数 swap_in()、try_to_swap_out() 和 swap_out()。get_free_page()用做内存页的申请，该函数在 mem_map 中找到一个空项并由该项下标计算出实际的物理页地址，然后对该页清 0，最后返回该物理页的地址，在早期版本的内核中该函数放在了 memory.c 文件中。sys_swapon()用来向系统中增加交换文件或交换设备。

2. 代码注释

swap.c 文件的详细代码注释如下：

```
1.  #include <string.h>          //字符串头文件。定义了所有字符串操作函数
2.  #include <errno.h>           //错误号头文件。包含系统中各种出错号
3.  #include <linux/mm.h>        //内存管理头文件。含有页面大小定义和一些页面释放函数原型的定义
4.  #include <sys/stat.h>        //文件状态头文件。一些属性操作测试宏、函数原型
5.  #include <linux/sched.h>     //调度程序头文件
6.  #include <linux/head.h>
7.  #include <linux/kernel.h>    //内核头文件。定义了一些内核常用函数的原型
8.  #define SWAP_BITS (4096<<3)
9.  #define bitop(name,op)
10. static inline int name(char * addr,unsigned int nr)
11. {
12. int __res;
13. __asm__ __volatile__("bt" op " %1,%2; adcl $0,%0"
14. :"=g" (__res)
15. :"r" (nr),"m" (*(addr)),"0" (0));
16. return __res;
17. }
18. bitop(bit,"")
19. bitop(setbit,"s")
20. bitop(clrbit,"r")
21. static char * swap_bitmap = NULL;
22. unsigned int swap_device = 0;
23. struct inode * swap_file = NULL;
24. void rw_swap_page(int rw, unsigned int nr, char * buf)
25. {
26. unsigned int zones[4];
27. int i;
28. if (swap_device) {                          //当是交换设备的情况下
29. ll_rw_page(rw,swap_device,nr,buf);         //调用底层的读写函数
30. return;
31. }
    //否则就是文件交换
32. if (swap_file) {
```

```
33.     nr <<= 2;
34.     for (i = 0; i < 4; i++)
35.     //如果 i 对应的区域交换文件的 bmap 为空，显示错误信息
36.     if (!(zones[i] = bmap(swap_file,nr++))) {
37.     printk("rw_swap_page: bad swap file\n");
38.     return;
39.     }
40.     ll_rw_swap_file(rw,swap_file->i_dev, zones,4,buf);   //调用底层的读写函数
41.     return;
42.     }
        //显示没有交换文件或交换设备
43.     printk("ll_swap_page: no swap file or device\n");
44.     }
        //线性空间的 64MB ~ 4GB 被交换到 swap 设备上去
45.     #define FIRST_VM_PAGE (TASK_SIZE>>12)
46.     #define LAST_VM_PAGE (1024*1024)
47.     #define VM_PAGES (LAST_VM_PAGE - FIRST_VM_PAGE)
        //进行内存的交换，从交换设备中获得空闲的页面
48.     static int get_swap_page(void)
49.     {
50.     int nr;
51.     if (!swap_bitmap)                          // 如果 swap_bitmap 的位都为零，直接返回
52.     return 0;
53.     for (nr = 1; nr < SWAP_BITS ; nr++)
54.     if (clrbit(swap_bitmap,nr))                // 如果 swap_bitmap 的 nr 位为零，返回 nr
55.     return nr;
56.     return 0;
57.     }
        //释放交换设备上的一个页
58.     void swap_free(int swap_nr)
59.     {
        //如果 swap_nr 为空，直接返回
60.     if (!swap_nr)
61.     return;
        //如果 swap_bitmap 没有为零的位，且 swap_nr < SWAP_BITS
62.     if (swap_bitmap && swap_nr < SWAP_BITS)
63.     if (!setbit(swap_bitmap,swap_nr))          //设置 swap_bitmap，成功并返回
64.     return;
65.     90  printk("swap_free: swap-space bitmap bad\n");    //否则显示错误信息
66.     return;
67.     }
        //将交换设备中的内存页面交换至物理内存中，参数 table_ptr 为页表项
68.     void swap_in(unsigned long *table_ptr)
69.     {
70.     int swap_nr;
71.     unsigned long page;
        //如果 swap_bitmap 为空，试图将交换设备中不存在的交换到内存中
72.     if (!swap_bitmap) {
73.     printk("Trying to swap in without swap bit-map");
74.     return;
```

```
75.     }
        //判断是否是当前页，试图将当前页交换到内存中
76.     if (1 & *table_ptr) {
77.     printk("trying to swap in present page\n\r");
78.     return;
79.     }
        //查看下一个页表项，如果不存在，则显示出没有可以交换的页
80.     swap_nr = *table_ptr >> 1;
81.     if (!swap_nr) {
82.     printk("No swap page in swap_in\n\r");
83.     return;
84.       }
         //从主内存区取一空闲页面
85.       if (!(page = get_free_page()))
86.       oom();//显示内存不够信息
87.       read_swap_page(swap_nr, (char *) page);
88.       if (setbit(swap_bitmap,swap_nr))
89.       printk("swapping in multiply from same page\n\r");
         //在对应页表项中置 User, U/S, R/W,D 等相应标志
90.       *table_ptr = page | (PAGE_DIRTY | 7);
91.       }
         //将内存中的数据交换至交换设备中
92.       int try_to_swap_out(unsigned long * table_ptr)
93.       {
94.       unsigned long page;
95.       unsigned long swap_nr;
96.       page = *table_ptr;
        //如果内存中缺少该页面，不能从内存写回到设备，则失败返回
97.       If (!(PAGE_PRESENT & page))
98.       return 0;
        //如果给定的地址是不合法的，则失败返回
99.       if (page - LOW_MEM > PAGING_MEMORY)
100.      return 0;
101.      if (PAGE_DIRTY & page) {   //如果该页面被使用过，取页面地址并置页面空闲，并成功返回
102.      page &= 0xfffff000;
103.      if (mem_map[MAP_NR(page)] != 1)
104.      return 0;
105.      if (!(swap_nr = get_swap_page()))
106.      return 0;
107.      *table_ptr = swap_nr<<1;
108.      invalidate();
109.      write_swap_page(swap_nr, (char *) page);
110.      free_page(page);                  //调用 free_page 函数释放页面
111.      return 1;                         //返回非零值表示成功
112.      }
113.      page &= 0xfffff000;               //取页面地址
114.      *table_ptr = 0;
115.       invalidate();                    //刷新高速缓冲
116.      free_page(page);                  //调用 free_page 函数释放页面，并成功返回
117.      return 1;
```

```
118.      }
          //通过查找页表，找到一个可以换出的页面
119.      int swap_out(void)
120.      {
121.      static int dir_entry = 1024;
122.      static int page_entry = -1;
123.      Int counter = VM_PAGES;
124.      int pg_table = 0;
125.      repeat:
126.      while (counter > 0) {
127.      counter -= 1024;
128.      dir_entry++;
129.        if (dir_entry >= 1024)        //页目录占一个内存页
130.      dir_entry = FIRST_VM_PAGE>>10;
131.      if (pg_table = pg_dir[dir_entry])
132.      break;
133.      }
134.      169    if (counter <= 0) {      //如果不大于零，交换空间不够，显示错误信息并返回
135.      printk("Out of swaρ-memory\n");
136.      return 0;
137.      }
138.        if (!(pg_table & 1)) {         //如果页表在页目录中不存在，显示错误信息并返回
139.      printk("bad page-table at pg_dir[%d]: %08x\n\r",dir_entry,
140.      pg_table);
141.      return 0;
142.      }
143.      pg_table &= 0xfffff000;        //取页表地址
144.      while (counter > 0) {
145.      counter--;
146.      page_entry++;
147.      if (page_entry >= 1024) {
148.      page_entry = -1;
149.      goto repeat;
150.      }
          //如果调用 try_to_swap_out（）函数成功，成功返回
151.      if (try_to_swap_out(page_entry + (unsigned long *) pg_table))
152.      return 1;
153.      }
          //否则显示交换空间不够信息
154.      printk("Out of swap-memory\n\r");
155.      return 0;
156.      }
         //得到一空闲物理页面，标志为已用，如果没有这样的页面，返回零
157.      unsigned long get_free_page(void)
158.      {
159.      unsigned long result;
160.      repeat:
161.        __asm__("std ; repne ; scasb\n\t" //方向位置位，将 al(0)与对应每个页面的（di）内容比较
162.        "jne 1f\n\t"                      //如果没有等于 0 的字节，则跳转结束（返回 0）
163.        "movb $1,1(%%edi)\n\t"            //将对应页面的内存映像位置 1
```

```
164.      "sall $12,%%ecx\n\t"                  //页面数*4K = 相对页面起始地址
165.      "addl %2,%%ecx\n\t"                   //再加上低端内存地址，即获得页面实际物理起始地址
166.       "movl %%ecx,%%edx\n\t"               //将页面实际起始地址放到 edx 寄存器
167.       "movl $1024,%%ecx\n\t"               //寄存器 ecx 置计数值 1024
168.       "leal 4092(%%edx),%%edi\n\t "        //将 4092+edx 的位置放到 edi（该页面的末端）
169.       "rep ; stosl\n\t"                    //将 edi 所指内存清零（反方向，也即将该页面清零）
170.       "movl %%edx,%%eax\n"                 //将页面起始地址放到 eax（返回值）
171.      "1:\tcld"
172.      :"=a" (result)
173.      :"0" (0),"i" (LOW_MEM),"c" (PAGING_PAGES),
174.      "D" (mem_map+PAGING_PAGES-1)
175.      :"di","cx","dx");
176.       if (result >= HIGH_MEMORY)    //若页面地址大于 HIGH_MEMORY，继续循环
177.      goto repeat;
         //若页面地址小于 LOW_MEM 或不在内存区，显示错误信息并返回零
178.      if ((result && result < LOW_MEM) || (result & 0xfff)) {
179.      printk("weird result: %08x\n",result);
180.      result = 0;
181.      }
182.      if (!result && swap_out())
183.      goto repeat;
184.       return result;                       // 返回空闲页面地址
185.      }
         // sys_swapon（）函数用来向系统中增加交换设备或交换文件
186.      int sys_swapon(const char * specialfile)
187.      {
188.       struct inode * swap_inode;
189.       int i,j;
190.        if (!suser())                       //如果不是超级用户操作不允许
191.       return -EPERM;
         //根据给定的 specialfile，寻找 swap_inode，如果找不到，返回没有这个目录或文件信息
192.       if (!(swap_inode    = namei(specialfile)))
193.       return -ENOENT;
194.       if (swap_file || swap_device || swap_bitmap) {
195.        iput(swap_inode);                   //将 swap_inode 结点放回到设备中
196.        return -EBUSY;                      //否则返回设备或资源忙
197.       }
198.        if (S_ISBLK(swap_inode->i_mode)) {          //结点是块设备
199.       swap_device = swap_inode->i_rdev;
200.       iput(swap_inode);
201.        } else if (S_ISREG(swap_inode->i_mode))    //结点是普通文件
202.       swap_file = swap_inode;
203.       else {
204.       iput(swap_inode);
205.        return -EINVAL;                             //否则返回参数不可用
206.       }
         //从内存申请一空闲页面
207.       swap_bitmap = (char *) get_free_page();
208.        if (!swap_bitmap) {                         //如果申请不成功，显示内存不够
209.       iput(swap_file);
```

```
210.        swap_device = 0;
211.        swap_file = NULL;
212.        printk("Unable to start swapping: out of memory :-)\n");
213.        return -ENOMEM;
214.        }
215.        read_swap_page(0,swap_bitmap);
216.        if (strncmp("SWAP-SPACE",swap_bitmap+4086,10)) {
217.        printk("Unable to find swap-space signature\n\r");
218.        free_page((long) swap_bitmap);
219.        iput(swap_file);
220.        swap_device = 0;
221.        swap_file = NULL;
222.        swap_bitmap = NULL;
223.        return -EINVAL;
224.        }
225.        memset(swap_bitmap+4086,0,10);
226.        j = 0;
227.        for (i = 1 ; i < SWAP_BITS ; i++)
228.        if (bit(swap_bitmap,i))
229.        j++;
230.        if (!j) {
231.        printk("Empty swap-file\n");
232.        free_page((long) swap_bitmap);
233.        iput(swap_file);
234.        swap_device = 0;
235.        swap_file = NULL;
236.        swap_bitmap = NULL;
237.        return -EINVAL;
238.        }
239.        printk("Adding Swap: %d pages (%d bytes) swap-space\n\r",j,j*4096);
240.        return 0;
241.        }
```

第 10 章　Linux 内核编程环境

10.1　建造工具介绍

本章所涉及的 Linux 内核代码主要是生成 Linux 内核磁盘镜像文件的建造工具代码（见表 10-1）。

表 10-1　Linux/tools 目录

名　　称	大　小（byte）	修　改　时　间
build.c	4522	1992-03-06 10:19:24

10.1.1　总体功能

Linux 内核源代码的建造工具主要用于将所有内核编译代码连接、合并成一个内核映像文件 image。Linux/tools 目录下的 build.c 程序就是建造工具程序，该程序被编译并执行后，会将 boot 目录下的 bootsect.s、setup.s 所编译生成的两个执行模块以及其他所有程序编译连接而成的 system 模块组合成一个内核映像文件 image。build.c 源程序编译而成的是一个独立的执行程序，它本身并没有被编译链接到内核代码中。

10.1.2　重要数据结构

与建造工具相关的数据结构主要是 Minix 可执行文件 a.out 的头部数据结构。

a.out 的头部数据结构详细信息参见 minix 2.0 源代码。

```
struct exec {
  unsigned char a_magic[2];          //执行文件魔数
  unsigned char a_flags;             //标志
  unsigned char a_cpu;               //cpu 标识
  unsigned char a_hdrlen;            //头部长度
  unsigned char a_unused;            //保留字段，未使用
  unsigned char a_version;           //版本号
  long     a_text;                   //代码段长度
  long     a_data;                   //数据段长度
  long     a_bss;                    //堆长度
  long     a_entry;                  //执行入口点地址
  long     a_total;                  //分配的内存总量
  long     a_syms;                   //符号表大小
  ......
};
```

在该数据结构中，标志字段 a_flags 有如下定义：

```
A_UZP       0x01          //未映射的 0 页数
A_PAL       0x02          //以页边界调整的可执行文件
A_NSYM      0x01          //新型符号表
A_EXEC      0x10          //可执行文件
A_SEP       0x20          //代码和数据是分开的
```

在该数据结构中，CPU 标识号 a_cpu 有如下几种定义：

```
A_NONE      0x00        //未知
A_I8086     0x04        //Intel i8086/8088
A_M68K      0x0B        //Motorola m68000
A_NS16K     0x0C        //国家半导体公司 16032
A_I80386    0x10        //Intel i80386
A_SPARC     0x17        //Sun 公司 SPARC
```

10.1.3 build.c 文件

build.c 文件用于将所有内核编译代码连接并组合成一个可运行的内核映像文件 image。该程序将先后对可选的根文件系统设备、bootsect 文件、setup 文件和 system 文件进行处理。

1. 功能描述

对可选的根文件系统设备进行处理，检查设备参数是否存在，如果存在，则读取该设备文件的状态信息结构并取出设备号。如果不存在，则使用默认值。

对 bootsect 文件进行处理，判断其 minix 执行头部信息的有效性，检查之后的 512 字节引导代码数据是否有可引导标志，并将可选的根文件系统设备号写入到 508、509 位移处，最后将该 512 字节的代码写到标准输出并重定向到映像文件。

对 setup 文件处理的方式与 bootsect 相似。如果该文件长度小于 4 个扇区，则将不足的部分用 0 填充，并写到标准输出中。对 system 文件进行处理，判断执行入口点，将数据写到标准输出中。如果其代码长度超过 128KB，则显示错误信息。

2. 代码注释

build.c 文件的详细代码注释如下：

```
1.      #include <stdio.h>                          /*主要使用其中的 fprintf() */
2.      #include <string.h>                         //用来做字符串相关操作
3.      #include <stdlib.h>                         /*含有 exit()*/
4.      #include <sys/types.h>                      /*unistd.h 使用这个头文件*/
5.      #include <sys/stat.h>                       //文件状态信息结构头文件
6.      #include <linux/fs.h>                       //文件系统头文件
7.      #include <unistd.h>                         /*包含 read()/write()*/
8.      #include <fcntl.h>                          //文件操作模式符号常数头文件
9.      #define MINIX_HEADER 32                     //设定 minix 二进制模块头部长度为 32 个字节
10.     #define GCC_HEADER 1024                     //设定 GCC 头部长度为 1024 个字节
11.     #define SYS_SIZE 0x4000                     //设定 system 文件最长节数
12.     #define DEFAULT_MAJOR_ROOT 0                //设定默认根设备主设备号为 0
13.     #define DEFAULT_MINOR_ROOT 0                //设定默认根设备次设备号为 0
14.     #define SETUP_SECTS 4                       //设定 setup 最大扇区数为 4 个扇区
15.     #define STRINGIFY(x) #x                     //出错时在显示语句中表示扇区数
16.      //该函数用于显示错误信息，并终止程序
17.     void die(char * str)
18.     {
19.         fprintf(stderr,"%s\n",str);             //显示错误信息
20.         exit(1);                                //终止程序
21.     }
       //该函数用于显示程序使用方法，并退出
22.     void usage(void)
```

```
23.     {
24.      die("Usage: build bootsect setup system [rootdev] [> image]"); //调用函数 die()显示信息
25.     }
        //该程序的主函数
26.     int main(int argc, char ** argv)
27.     {
28.      int i,c,id;
29.      char buf[1024];
30.      char major_root, minor_root;
31.      struct stat sb;
32.
33.      if ((argc < 4) || (argc > 5))                    //如果程序命令行参数小于 4 或者大于 5
34.        usage();                                       //则显示程序使用方法并退出
35.      if (argc > 4) {                                  //如果程序命令行参数为 5 时
36.        if (strcmp(argv[4], "FLOPPY")) {               //如果根设备是软盘
37.              if (stat(argv[4], &sb)) {                //则取其状态信息
38.                    perror(argv[4]);                   //如果出错
39.                    die("Couldn't stat root device."); //则显示错误信息并退出
40.              }
41.              major_root = MAJOR(sb.st_rdev);          //取该设备主设备号
42.              minor_root = MINOR(sb.st_rdev);          //取该设备次设备号
43.        } else {                                       //如果该设备不是软盘
44.              major_root = 0;                          //使主设备号为 0
45.              minor_root = 0;                          //使次设备号为 0
46.        }
47.        } else {                                       //如果只有四个参数
48.              major_root = DEFAULT_MAJOR_ROOT;         //则使主设备号为默认的主设备号
49.              minor_root = DEFAULT_MINOR_ROOT;         //则使次设备号为默认的次设备号
50.        }
51.      fprintf(stderr, "Root device is (%d, %d)\n", major_root, minor_root); //在标准错误终端上显
52.                                                       //示根设备的主设备和次设备号
53.     if ((major_root != 2) && (major_root != 3) &&     //如果主设备号不等于 2、3、0
54.          (major_root != 0)) {
55.      fprintf(stderr, "Illegal root device (major = %d)\n",  //则显示非法的根设备号
56.        major_root);
57.      die("Bad root device --- major #");              //同时，显示错误信息并退出
58.     }
59.     for (i=0;i<sizeof buf; i++) buf[i]=0;             //将 buf 缓冲区初始化为 0
60.     if ((id=open(argv[1],O_RDONLY,0))<0)              //以只读方式打开文件 bootsect
61.      die("Unable to open 'boot'");                    //如果出错，则显示错误信息并退出
62.     if (read(id,buf,MINIX_HEADER) != MINIX_HEADER)    //读取该文件 minix 执行头部信息
63.      die("Unable to read header of 'boot'");          //如果出错，则显示错误信息并退出
64.     if (((long *) buf)[0]!=0x04100301)                //判断头部中魔数字段 a_magic、可执行字段
65.                                                       //a_flag、CPU 标识字段 a_cpu 是否为 0x0301、
66.                                                       //0x10、0x04
67.      die("Non-Minix header of 'boot'");               //如果不相符，则显示出错信息并退出
68.     if (((long *) buf)[1]!=MINIX_HEADER)              //判断头部长度字段 a_hdrlen 是否正确
69.      die("Non-Minix header of 'boot'");               //如果不正确，则显示错误信息并退出
70.     if (((long *) buf)[3]!=0)                         //判断数据段长字段 a_data 内容是否为 0
71.      die("Illegal data segment in 'boot'");           //如果不为 0，则显示错误信息并退出
```

```
72.     if (((long *) buf)[4]!=0)                       //判断堆字段 a_bss 内容是否为 0
73.       die("Illegal bss in 'boot'");                 //如果不为 0，则显示错误信息并退出
74.     if (((long *) buf)[5] != 0)                     //判断执行点字段 a_entry 内容是否为 0
75.       die("Non-Minix header of 'boot'");            //如果不为 0，则显示错误信息并退出
76.     if (((long *) buf)[7] != 0)                     //判断符号表长字段 a_sym 内容是否为 0
77.       die("Illegal symbol table in 'boot'");        //如果不为 0，则显示错误信息并退出
78.     i=read(id,buf,sizeof buf);                      //读取实际代码数据
79.     fprintf(stderr,"Boot sector %d bytes.\n",i);    //显示读取的实际代码数据的字节数
80.     if (i != 512)                                   //如果读取实际代码数据的字节数不为 512
81.       die("Boot block must be exactly 512 bytes");  //则显示错误信息并退出
82.     if ((*(unsigned short *)(buf+510)) != 0xAA55)   //判断 boot 块 0x510 处是否有可引
83.                                                     //导标志 0XAA55
84.       die("Boot block hasn't got boot flag (0xAA55)"); //如果没有，则显示错误信息并退出
85.     buf[508] = (char) minor_root;                   //在引导块的 508 偏移处存放根设备的次设备号
86.     buf[509] = (char) major_root;                   //在引导块的 509 偏移处存放根设备的主设备号
87.     i=write(1,buf,512);                             //将该 boot 块 512 字节的数据写到标准输出
88.     if (i!=512)                                     //如果写出的字节数不是 512 个
89.       die("Write call failed");                     //则显示错误信息并退出
90.     close (id);                                     //关闭 bootsect 模块文件
91.     if ((id=open(argv[2],O_RDONLY,0))<0)            //以只读方式打开 setup 模块
92.       die("Unable to open 'setup'");                //如果出错，则显示错误信息并退出
93.     if (read(id,buf,MINIX_HEADER) != MINIX_HEADER) //读取该文件中的 minix 执行头部信息
94.       die("Unable to read header of 'setup'");      //如果出错，则显示错误信息并退出
95.     if (((long *) buf)[0]!=0x04100301)              //判断头部中魔数字段 a_magic、可执行字段
96.                                 //a_flag、CPU 标识字段 a_cpu 是否为 0x0301、//0x10、0x04
97.       die("Non-Minix header of 'setup'");           //如果出错，则显示错误信息并退出
98.     if (((long *) buf)[1]!=MINIX_HEADER)            //判断头部长度字段 a_hdrlen 是否正确
99.       die("Non-Minix header of 'setup'");           //如果不正确，则显示错误信息并退出
100.    if (((long *) buf)[3]!=0)                       //判断数据段长字段 a_data 内容是否为 0
101.      die("Illegal data segment in 'setup'");       //如果不是，则显示错误信息并退出
102.    if (((long *) buf)[4]!=0)                       //判断堆字段 a_bss 内容是否为 0
103.      die("Illegal bss in 'setup'");                //如果不是，则显示错误信息并退出
104.    if (((long *) buf)[5] != 0)                     //判断执行点字段 a_entry 内容是否为 0
105.      die("Non-Minix header of 'setup'");           //如果不是，则显示错误信息并退出
106.    if (((long *) buf)[7] != 0)                     //判断符号表长字段 a_sym 内容是否为 0
107.      die("Illegal symbol table in 'setup'");       //如果不是，则显示错误信息并退出
                                                        //读取随后的执行代码数据，并写出到标准输出
108.    for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c )
109.      if (write(1,buf,c)!=c)
110.            die("Write call failed");
111.    close (id);                                     //关闭 setup 模块文件
112.    if (i > SETUP_SECTS*512)                        //如果 setup 模块长度大于 4 个扇区
113.      die("Setup exceeds " STRINGIFY(SETUP_SECTS)   //则显示出错信息并退出
114.            " sectors - rewrite build/boot/setup");
115.    fprintf(stderr,"Setup is %d bytes.\n",i);       //显示 setup 模块文件的长度
116.    for (c=0 ; c<sizeof(buf) ; c++)                 //将缓冲区 buf 清零
117.      buf[c] = '\0';
      //如果 setup 长度小于 4*512 个字节，则用\0 将 setup 补充到 4*512 个字节
118.    while (i<SETUP_SECTS*512) {
119.      c = SETUP_SECTS*512-i;
```

```
120.        if (c > sizeof(buf))
121.                c = sizeof(buf);
122.        if (write(1,buf,c) != c)
123.                die("Write call failed");
124.        i += c;
125.    }
126.        if ((id=open(argv[3],O_RDONLY,0))<0)        //以只读方式打开 system 模块文件
127.        die("Unable to open 'system'");             //如果失败，显示错误信息并退出
128.    if (read(id,buf,GCC_HEADER) != GCC_HEADER)      //读取 GCC 格式头部数据结构信息
129.        die("Unable to read header of 'system'");   //如果出错，则显示错误信息并退出
130.    if (((long *) buf)[5] != 0)                     //判断执行代码入口点字段 a_entry 是否为 0
131.        die("Non-GCC header of 'system'");          //如果不为 0，则显示错误信息并退出
132.    for (i=0 ; (c=read(id,buf,sizeof buf))>0 ; i+=c ) //读取随后的执行代码数据
133.        if (write(1,buf,c)!=c)                      //写到标准输出
134.                die("Write call failed");           //如果出错，显示错误信息并退出
135.    close(id);                                      //关闭 system 模块文件
136.    fprintf(stderr,"System is %d bytes.\n",i);      //显示 system 文件的字节数
137.    if (i > SYS_SIZE*16)                            //如果该文件字节数大于 SYS_SIZE*16 字节
138.                die("System is too big");           //则显示错误信息并退出
139.    return(0);
140.    }
```


源代码中第 9～15 行描述了这个程序将三个不同的文件创建成的磁盘镜像文件：- bootsect: 该文件的 8086 机器代码最大 510 个字节，加载其他程序；- setup: 该文件的 8086 机器代码最大 4 个扇区，设置系统参数；- system: 实际系统的 80386 代码。该程序检查所有的文件是否为正确的类型，并且将结果写到 stdout，删除头部并扩充到正确的长度。它也会写一些系统数据到 stderr。

10.2 在 Redhat 9 下修改和编译 Linux 0.95 内核

最初 Linux 将 GNU 的系列开发工具 gcc、gld、emacs、bash 等移植到 Minix-386 上，并在这个平台上进行交叉编译，开发出 Linux0.01、0.03、0.11、0.95 等版本的内核。但该开发平台的建立非常烦琐，因此本节将介绍如何修改 Linux 0.95 版内核源代码，以使其能在目前常用的 RedHat 9 操作系统标准的编译环境下进行编译，生成可运行的启动映像文件 bootimage。

10.2.1 源代码中注释语句的修改

as86 编译程序不能识别 C 语言的注释语句，因此需要使用“!”注释掉 boot/bootsect.s 文件中的 C 注释语句。

10.2.2 源代码中 makefile 文件的修改

在 Linux 0.95 内核代码文件中，需要对每个子目录中的 makefile 文件修改：

① 现在 gas 和 gld 已经改名为 as 和 ld 了，所以需要将 makefile 文件中的 gas 修改为 as，gld 修改为 ld。

② 去掉 gcc 的编译标志选项：-fcombine-regs、-mstring-insns 以及所有子目录中 Makefile 中的这两个选项。

③ as（原 gas）已经不用-c 选项，因此需要去掉其-c 编译选项。

④ 在 gcc 的编译标志选项中，增加-m386 选项。

10.2.3 源代码中 align 值的修改

boot 目录下有三个汇编程序，在这些汇编程序中 align 后面带的数值原本是指对齐内存位置的幂次值，而现在是对齐的整数地址值。因此要将原来的语句 align 3 修改成 align。

10.2.4 源代码中嵌入宏汇编程序的修改

目前 as 自动化程度越来越高，不再需要人工指定一个变量需使用的 CPU 寄存器，所以内核代码中的_asm_（"ax"）要全部去掉。另外还需要去掉嵌入的汇编代码中所有对寄存器内容无效的声明。

10.2.5 源代码中汇编程序 C 变量的修改

在开发 Linux 0.95 时所用的汇编器，在引用 C 程序中的变量时需要在变量名前加一下划线字符“_”，而目前的 gcc 编译器可以直接识别使用这些汇编中引用的 c 变量，因此需要将汇编程序中所有 c 变量之前的下画线去掉。例如：

.globl _idt,_gdt,_pg_dir,_tmp_area 需要修改成.globl idt,gdt,pg_dir,tmp_area

lss_stack_start, %esp 需要修改成 lss stack_start, %esp.

10.2.6 在保护模式下调试显示函数

进入了保护模式后，ROM BIOS 中的 int 0x10 这些中断无法使用。使用下面的数据显示函数 check_data32()可以在保护模式运行中了解内核的内部数据结构和状态。该函数在进入保护模式后，在屏幕上可以打印出需要的东西。

```
Inline void check_data32(int value, int pos)
{
_asm_ _volatile_(
“shl          $4, %%ebx\n\t”
“addl         $0xb8000, %%ebx\n\t”
“movl         $0xf0000000, %%eax\n\t”
“movb      $28, %%cl\n”
“1:\n\t”
“movl      $0, %%edx\n\t”
“andl       %%eax, %%edx\n\t”
“shr        %%cl, %%edx\n\t”
“add       $0x30, %%dx\n\t”
“cmp        $0x3a, %%dx\n\t”
“jb2f\n\t”
“add       $0x07, %%dx\n”
“2:\n\t”
“add       $0x0c00, %%dx\n\t”
“movw    %%dx, (%%ebx)\n\t”
“sub       $0x04, %%cl\n\t”
“shr       $0x04, %%eax\n\t”
“add       $0x02, %%ebx\n\t”
“cmpl      $0x0, %%eax\n\t”
“jnzlb\n”
::”m”(value), “b”(pos));
}
```

10.2.7 编译 Linux 0.95 内核

将文件修改完成后，在 Linux0.95 内核源代码的/usr/src/linux 目录下输入 make 命令即可进行编译，并生成引导启动映像文件 image。在编译内核时通常很顺利就会完成，如果出现编译器 gcc 不能识别参数-mstring-ins 的问题，可以直接删除所有 Makefile 中的这个参数再重新编译内核。如果出现找不到 gar 命令的问题，可以把/usr/local/bin 下的 ar 直接链接或复制/改名成 gar。

用编译好的启动映像文件软盘启动时，屏幕上应该会显示以下信息：

```
Booting from Floppy……..
Loading system …..
Insert root floppy and press ENTER
```

如果再显示出“Loading system”后就没反应了，这说明内核不能识别计算机中的硬盘控制器子系统，读者可以尝试使用 Bochs 虚拟机软件进行试验。

10.3 在 Bochs 上运行 Linux 0.95 系统

Bochs 是一个能仿真 Intel x86 计算机的程序。在执行过程中，Bochs 仿真了所有的指令，并且含有外设所有的设备模块。在其中执行的软件会感觉是在一个真实的机器上运行。使用 Bochs 可以知道被执行程序在仿真硬件环境中的具体状态和精确时序，而并非实际硬件系统执行的结构。

10.3.1 创建磁盘映像文件

磁盘映像文件是以文件形式存在的软盘或硬盘的完整映像。映像文件中信息格式与对应磁盘的信息格式相同。创建的空磁盘映像文件容量与对应磁盘相同，但内容为 0。

在为 Linux 0.95 创建硬盘映像文件时，由于其本身的限制，每个硬盘分区最大为 64MB，并且最多只支持 4 个分区，所以该系统可使用的硬盘映像文件最大为 256MB。刚创建的硬盘映像文件内容全为 0，只是容量要求一致。因此可以计算出要求容量的映像文件的扇区数，然后在 Linux 系统下使用 dd 命令来产生相应的映像文件。

假设我们想要创建的硬盘映像文件的柱面数是 400、磁头数是 15、每磁道扇区数为 64，其扇区总数为 384000，则应执行的命令为：

```
dd if=/dev/diskimage of=hdc.img bs=512 count=384000
```

10.3.2 访问磁盘映像文件

Linux 系统能够访问多种文件系统，同样可以利用 loop 设备访问存储在文件中的文件系统。为了访问一个硬盘映像文件中某个分区的信息，我们需要了解这个硬盘映像文件的分区信息，以确定需要访问的分区在文件中的偏移位置。假设硬盘映像文件名为 diskimage-0.95.img，以此为例说明访问第一个分区中文件系统的方法。

首先将 diskimage-0.95.img 与 loopl 相关联，并利用 fdisk 命令查看信息：

```
losetup /dev/loopl diskimage-0.95.img
fdisk /dev/loopl
Command （m for help）：x
Expert command （m for help）：p
```

执行上面的命令后，将显示分区信息，记下所要访问的分区起始扇区号。然后使用 losetup

的-d 选项把 diskimage-0.95.img 文件与 loopl 关联解除，重新把其关联到 diskimage-0.95.img 第一个分区的起始位置（假设第一分区的起始偏移位置为 1024 字节），这需要使用 losetup 的-o 选项：

```
losetup –d /dev/loopl
losetup –o 1024 /dev/loopl hdc-0.11.img
mount –t minix /dev/loopl/mnt
cd /mnt
ls
```

对分区中文件系统信息访问结束后要卸载和解除关联。

```
umount /dev/loopl
losetup –d /dev/loopl
```

10.3.3 创建文件系统

对于已经创建的硬盘映像文件，必须进行分区和创建文件系统才能使用。假设你已经安装了 Bochs 下的模拟系统（可以到 Bochs 网站下载），那么可以将硬盘映像文件挂接到该模拟系统中，然后使用模拟系统中的相关命令对其进行处理。我们将在上文制作的硬盘映像文件中创建一个分区，并且建立 Minix 文件系统，执行步骤如下：

在模拟系统的目录下创建子目录 Linux-0.95，将 diskimage-0.95.img 移动到该目录下。进入模拟系统的 Bochs 配置文件，在其中加入硬盘映像文件的配置参数行。

启动 Bochs，利用 fdisk 命令在 diskimage-0.95.img 中建立一个分区，记住该分区中数据块数大小，在创建文件系统时会使用到这个值。当分区建立好后，重新启动系统，让模拟系统识别这个新分区。再次进入模拟系统后，使用 mkfs 命令在建立的第一个分区上创建 Minix 文件系统：

```
mkfs /dev/hdb1 64000
21333 inodes
64000 blocks
Firstdatazone=680
Zonesize=1024
Maxsize=268966912
```

完成了第一分区创建文件系统的工作后，可以把这个分区建立成一个根文件系统。

10.3.4 在硬盘映像文件上建立根文件系统

之前已经在硬盘映像文件的第一分区上创建了 Minix 文件系统，接下来就可以加载硬盘上的文件系统了。把新的文件系统加载到/mnt 目录上：

```
mount /dev/hd 1 /mnt
```

加载了硬盘分区的文件系统之后，我们就可以把存有根文件系统的软盘上的内容复制到硬盘上去了：

```
cd /mnt
for I in bin dev etc usr tmp
>do
>cp +recursive+verbose /$i $i
```

在硬盘映像文件上建立好文件系统后，就可以将它作为根文件系统进行启动。

10.3.5 使用硬盘映像上的根文件系统运行 Linux 0.95

在硬盘映像文件上建立好根文件系统，就可以让 Linux 0.95 以它为根文件系统进行启动。这样要修改已经编译好的启动映像文件的第 509、510 字的内容才能实现。首先复制启动映像文件和 Bochs 的配置文件，将复制的 Bochs 配置文件中的'floppya:上的文件名修改成复制的启动映像文件名并存盘。用二进制编辑器修改复制的启动映像文件的第 509、510 字节为 01、03，表示根文件系统设备在硬盘映像文件的第 1 个分区上。如果把文件系统安装在了别的分区上，那么需要修改前 1 个字节以对应到所指定的分区上。

运行复制的 Bochs 配置文件，Bochs 系统就会快速地运行 Linux 0.95 并显示出图形来。

明天是什么日子？（**搜索**）、糟糕，菜被你偷光了（**SNS**）、2元团购原价120元的优惠券（**团购**）、@嫦娥：广寒宫好玩吗？（**微博**）.....**各种网站火爆全球，心动了，但却感觉网站建设无从下手？**

HOT！

现在有了新方法！从《21天网站建设实录》开始，来创建属于你自己的商业网站吧！

SNS、互动论坛

书名：《21天网站建设实录》
定价：59元
书号：ISBN 978-7-113-12662

■26小时高品质视频语音讲解+4类大型完整商务网站源代码+300例网站建设通用图片模板
■大容量DVD完整提供书中源码、教学录像、商务网站源码与网站开发技术文档；
■电子商务网站开发全程实录，直指实践能力与项目工程思想，逐层剖析关键技术；
■一天一个任务驱动，大型商业案例开发贯穿全书，浓缩一线网站开发工程师多年经验与感悟。

从零开始学系列

图书推荐

从零开始学系列书特点

本系列书由名家精心编写，知识体系循序渐进，由浅入深；技术全面，内容充实；示例讲解，理解深刻；案例精讲深入剖析；语言生动，知识体系完善；操作步骤详细，代码注释丰富；光盘运行界面布局合理，方便读者对光盘的运用；附送长达十几小时的多媒体教学视频和教学PPT。帮助读者快速掌握语言开发和软件使用诀窍，全面提高编程实战能力。

系列书适用读者群体

本系列书适合于未学过编程语言的零基础读者，也适合于有一定编程语言基础，想进一步提高其编程水平的读者，还可以作为各大中专院校计算机编程语言学习的教材或参考用书。

ÓÉ Foxit PDF Editor ±à¼-
°æÈ¨ËùÓÐ (c) by Foxit Software Company, 2004
½öÓÃÓÚÆÀ¹À¡£

- android与iphone及ipad开发书籍 ----------持续不断更新中.....
- c、c++、c#语言pdf书籍及vip视频教程 c、c++、c#、vc等-------持续不断更新中.....
- delphi《书籍》及《视频》教程 ----------持续不断更新中.....
- E网情深VIP系列视频教程 黑客破解菜鸟修练班，VB编程学习班，仿站学习培训，免杀培训，个人系统攻防系列教程，服务器搭建学习班，PHOTOSHOP平面设计班，基础制作论坛（论坛网站搭建），网赚系列教程，网站建设教程，网站漏洞基础，远程控制教程，软件破解班，脚本漏洞提权班
- IT9网络学院VIP系列视频教程 免杀培训班，VMware虚拟机，零基础学习C语言，网游外挂开发精品系列语音教程（外挂教程学习必备研修31课全），VB语言教程30课全，Delphi编程到精通，远程控制软件，加密解密班，网络安全与黑客攻防培训，从入门到精通完整系统化学习C++编程，从入门到精通零基础学习汇编，wordpress教程(个人博客系统49课全)，外行人做易语言盗号和钓鱼程序语音教程 网址：WLSAM168.400GB.COM
- Java书籍 ----------持续不断更新中.....
- photoshop、CorelDRAW、AutocAD等图像处理书籍及vip视频教程 -------持续不断更新中.....
- powerbuilder书籍大全
- Visual Basic语言vip视频教程及pdf书籍 ----------持续不断更新中.....
- windows、linux系统开发、系统封装等pdf书籍及VIP视频教程 ----------持续不断更新中.....
- 《3DS Max》pdf书籍
- 《汇编语言》、《反汇编》及《调试》pdf书籍及vip视频教程 ----------持续不断更新中.....
- 《电子书、电子书、还是电子书》pdf专题库 编程开发，家居美食，儿童益智，人物传记，增强记忆，快速阅读
- 信息系统项目管理师、网络工程师、系统分析师等软考类书籍
- 华中红客系列vip视频教程 脚本攻防培训班，源码免杀培训班，Css语言培训班，C语言，Dreamweaver网页设计，html网页设计培训班，PC安全班， php脚本语言培训班，VMWare虚拟机专题，webshell提权培训班，防站教程，零基础免杀培训班，刷钻速成班，脱壳破解班，外挂编写班，网络赚钱培训班，网站入侵培训班
- 外挂、驱动、逆向及封包视频教程 郁金香、独立团、夜猫论坛、天都吧、看流星论坛、一切从零开始等等
- 安全中国系列vip视频教程 易语言软件编程培训班，ASP.net网站开发项目实战培训班
- 我的收藏
- 按键精灵及TC脚本开发软件视频教程 ----------持续不断更新中.....

当前位置：/《电子书、电子书、还是电子书》pdf专题库

文件名

PDF电子书专题库，内容详尽，每天不断更新！！

- 办公类软件使用指南
- 医学
- 历史人物传记
- 哲学宗教
- 外语资料（除英语外） （除英语外）
- 官场类小说
- 建筑工程类
- 情感生活类小说
- 政治军事
- 教育学习科普大全
- 文学理论
- 智力开发、增强记忆、快速阅读技巧大全
- 社会生活
- 科学技术
- 程序编程类
- 经济管理
- 网络安全及管理
- 网赚系列
- 美食小吃烹饪煲汤大全
- 课外读物

本网盘内容太多，持续不断更新，发布各类视频教程、pdf书籍，包括破解、加解密、外挂辅助制作，易语言培训教程、编程语言、网页制作等等，教程及书籍仅用于学习,如用于商业或非法用途的后果自负！

网址：WLSAM168.400GB.COM

VIP培训教程，易语言黑月VIP视频教程，天都网易语言系列培训教程(100集全)，集中营易语言学习视频(80集)

- 棉猴系列vip视频教程 gh0st远控控制源码讲解教程，套接字编程，DLL程序编写，键盘监听驱动程序编写，驱动基础教程，AsyncSelect模型QQ程序教程，C++语言入门基础，NB5.5源码分析教程
- 游戏开发pdf书籍 ----------持续不断更新中.....
- 炒股投资pdf书籍及视频教程 短线高手系列，短线天王系列，操盘论道系列，翻倍黑马，看盘快速入门，庄家手法大曝光等等。 网址：WLSAM168.400GB.COM
- 热门小说集中营 傲世九重天，网游之三国时代，武动乾坤
- 甲壳虫VIP教程全集 asp教程，Delphi培训班，FLASH培训班，Java培训班，linux培训班，PHP培训班，源码免杀班，甲壳虫C++，脚本功防班，免杀班初、中、高级班，破解班，源码免杀班，脱壳班，易语言培训班，无特征码免杀，网站架构培训班，外挂高级班，外挂初级班第1、2部
- 破解、免杀、入侵、脱壳、攻防及漏洞分析系列VIP视频教程（80多部） 天草、黑客动画吧等等--------持续不断更新中....
- 网站建设相关的pdf书籍及各种vip视频教程 ----------持续不断更新中.....
- 网赚、淘宝系列vip视频教程 网赚30天新人魔鬼训练，屠龙网赚团队vip课程，站长大学网赚视频（50课全），图腾团队日赚1000元竞价行销教程，屠龙团队淘宝宝贝卖疯系列，站群网赚系列，淘宝开店视频，红星挂机日赚10元，百万流量系列，漂流瓶圣手全自动挂机引，贴吧邮件定向营销疯狂成交量月入万元
- 英语学习资料百科大全 不断更新。。。
- 饭客论坛系列VIP视频教程 脚本入侵班，黑客之免杀教程，易语言教程，无线网络攻防教程，入侵教程，delphi系列教程，黑客基础入门
- 黑客书籍 有关黑客、安全、加解密技术等等----------持续不断更新中.....
- 黑手安全网VIP系列视频教程 DIV+CSS网页布局，Dreamweaver教程，flsah动画教程，photoshop教程，跟我一起学C++课程，抓鸡
- 黑鹰、黑基、黑防、黑盾vip系列视频教程 破解提高班66课全，SQL注入，ASP注入教程，完完全全学会抓肉鸡，脱壳破解教程50课全，提权班，C语言特训班26讲全，黑客脚本特训班，黑客工具特训班，dedecms仿站教程，VC编写远控30课全，网页美工特训班，木马免杀特训班，驱动开发技术VIP培训班，外挂破解等等。

- [电脑世界的通关密语：电脑编程基础].(杉浦贤).滕永红.扫描版.pdf
- [程序语言的奥妙：算法解读（四色全彩）].(杉浦贤).李克秋.扫描版.pdf
- [差错：软件错误的致命影响].(帕伯斯).邝宇恒等.扫描版.pdf
- [算法之道（第2版）].邹恒明.扫描版.pdf
- [O'Reilly：深入学习MongoDB].(霍多罗夫).巨成等.扫描版.pdf
- [深入浅出WPF].刘铁猛.扫描版.pdf
- [Go语言·云动力（云计算时代的新型编程语言）].樊虹剑.扫描版.pdf
- [精通.NET互操作：P/ Invoke、C++ Interop和COM Interop].黄际洲等.扫描版.pdf
- [编程的奥秘：.NET软件技术学习与实践].金旭亮.扫描版.pdf
- [O'Reilly：学习OpenCV（中文版）].(布拉德斯基等).于仕琪等.扫描版.pdf
- [Go语言编程].许式伟等.扫描版.pdf 网址：WLSAM168.400GB.COM
- [MySQL技术内幕：SQL编程].姜承尧.扫描版.pdf
- [Tomcat权威指南（第2版）].(布里泰恩等).吴豪等.扫描版.pdf
- [Ext江湖].大漠穷秋.扫描版.pdf
- [IT名人堂·Oracle DBA突击：帮你赢得一份DBA职位].张晓明.扫描版.pdf

Total: 77 1 2 3 4 5 6 ›

HTTP://WLSAM168.400GB.COM

读者意见反馈表

亲爱的读者：

感谢您对中国铁道出版社的支持，您的建议是我们不断改进工作的信息来源，您的需求是我们不断开拓创新的基础。为了更好地服务读者，出版更多的精品图书，希望您能在百忙之中抽出时间填写这份意见反馈表发给我们。随书纸制表格请在填好后剪下寄到：北京市西城区右安门西街8号中国铁道出版社综合编辑部 苏茜 收（邮编：100054）。或者采用传真（010-63549458）方式发送。此外，读者也可以直接通过电子邮件把意见反馈给我们，E-mail地址是：suqian@tqbooks.net。我们将选出意见中肯的热心读者，赠送本社的其他图书作为奖励。同时，我们将充分考虑您的意见和建议，并尽可能地给您满意的答复。谢谢！

- - - -