

高等院校计算机实验与实践系列示范教材

计算机操作系统 实践教程

黄廷辉 王宇英 编著

清华大学出版社
北京

读者意见反馈

亲爱的读者：

感谢您一直以来对清华版计算机教材的支持和爱护。为了今后为您提供更优秀的教材，请您抽出宝贵的时间来填写下面的意见反馈表，以便我们更好地对本教材做进一步改进。同时如果您在使用本教材的过程中遇到了什么问题，或者有什么好的建议，也请您来信告诉我们。

地址：北京市海淀区双清路学研大厦 A 座 602 计算机与信息分社营销室 收

邮编：100084 电子邮件：jsjjc@tup.tsinghua.edu.cn

电话：010-62770175-4608/4409 邮购电话：010-62786544

教材名称：计算机操作系统实践教程

ISBN：978-7-302-14800-5

个人资料

姓名：_____ 年龄：_____ 所在院校/专业：_____

文化程度：_____ 通信地址：_____

联系电话：_____ 电子信箱：_____

您使用本书是作为：□指定教材 □选用教材 □辅导教材 □自学教材

您对本书封面设计的满意度：

□很满意 □满意 □一般 □不满意 改进建议 _____

您对本书印刷质量的满意度：

□很满意 □满意 □一般 □不满意 改进建议 _____

您对本书的总体满意度：

从语言质量角度看 □很满意 □满意 □一般 □不满意

从科技含量角度看 □很满意 □满意 □一般 □不满意

本书最令您满意的是：

□指导明确 □内容充实 □讲解详尽 □实例丰富

您认为本书在哪些地方应进行修改？（可附页）

电子教案支持

敬爱的教师：

为了配合本课程的教学需要，本教材配有配套的电子教案（素材），有需求的教师可以与我们联系，我们将向使用本教材进行教学的教师免费赠送电子教案（素材），希望有助于教学活动的开展。相关信息请拨打电话 010-62776969 或发送电子邮件至 jsjjc@tup.tsinghua.edu.cn 咨询，也可以到清华大学出版社主页 (<http://www.tup.com.cn> 或 <http://www.tup.tsinghua.edu.cn>) 上查询。

编审委员会

主 编 张基温

委 员 (以汉字拼音顺序排列)

陈家琪 丁 岭 李仁发

苗夺谦 孙佳文 索 梅

殷晓峰 张基温 赵一鸣

出版说明

当前，重视实验与实践教育是各国高等教育界的发展潮流，我国与国外教学工作的差距也主要表现在实践教学环节上。面对新的形式和新的挑战，完善实验与实践教育体系成为一种必然。为了培养具有高质量、高素质、高实践能力和高创新能力的人才，全国很多高等院校在实验与实践教学方面进行了大力改革，在实验与实践教学内容、教学方法、教学体系、实验室建设等方面积累了大量的宝贵经验，起到了教学示范作用。

实验与实践性教学与理论教学是相辅相成的，具有同等重要的地位。它是在开放教育的基础上，为配合理论教学、培养学生分析问题和解决问题的能力以及加强训练学生专业实践能力而设置的教学环节；对于完成教学计划、落实教学大纲，确保教学质量，培养学生分析问题、解决问题的能力和实际操作技能更具有特别重要的意义。同时，实践教学也是培养应用型人才的重要途径，实践教学质量的好坏，实际上也决定了应用型人才培养质量的高低。因此，加强实践教学环节，提高实践教学质量，对培养高质量的应用型人才至关重要。

近年来，教育部把实验与实践教学作为对高等院校教学工作评估的关键性指标。2005年1月，在教育部下发的《关于进一步加强高等学校本科教学工作的若干意见》中明确指出：“高等学校要强化实践育人的意识，区别不同学科对实践教学的要求，合理制定实践教学方案，完善实践教学体系。要切实加强实验、实习、社会实践、毕业设计（论文）等实践教学环节，保障各环节的时间和效果，不得降低要求。”，“要不断改革实践教学内容，改进实践教学方法，通过政策引导，吸引高水平教师从事实践环节教学工作。要加强产学研合作教育，充分利用国内外资源，不断拓展校际之间、校企之间、高校与科研院所之间的合作，加强各种形式的实践教学基地和实验室建设。”

为了配合开展实践教学及适应教学改革的需要，我们在全国各高等院校精心挖掘和遴选了一批在计算机实验与实践教学方面具有潜心研究并取得

了富有特色、值得推广的教学成果的作者，把他们多年积累的教学经验编写成教材，为开展实践教学的学校起一个抛砖引玉的示范作用。

为了保证出版质量，本套教材中的每本书都经过编委会委员的精心筛选和严格评审，坚持宁缺毋滥的原则，力争把每本书都做成精品。同时，为了能够让更多、更好的实践教学成果应用于社会和各高等院校，我们热切期望在这方面有经验和成果的教师能够加入到本套丛书的编写队伍中，为实践教学的发展和取得成效做出贡献；也衷心地期望广大读者对本套教材提出宝贵意见，以便我们更好地为读者服务。

清华大学出版社

联系人：索梅 suom@tup.tsinghua.edu.cn

前言

PREFACE

操作系统课程的内容涉及理论、算法、技术、实现和应用，知识体系繁杂，概念和原理抽象，是一门实践性较强的课程。目前国内操作系统课程设置多偏重于理论学习，对课程实践重视不够。学生普遍反映，不实际动手参与操作系统内核开发，只能掌握一些抽象的概念，不能深刻理解操作系统的根本，更不能解决实际问题。

教学实践也证明，学生想要真正掌握操作系统的实际工作原理，并根据需求灵活运用操作系统的最好方法是参与到操作系统内核设计的实践工作中，即通过阅读现有操作系统的代码，理解操作系统内核工作基本原理，在此基础上修改和完成指定操作系统功能的部分代码。通过与实践相结合，学生就能比较容易地掌握操作系统的抽象概念，也会发现操作系统复杂的结构是可以逐步分析和理解的。现在，操作系统课程实践的重要性正在被越来越多的国内大学所重视，但大部分院校采用的都是 Linux 操作系统设计平台。由于 Linux 操作系统是为实际应用开发的，主要的操作系统功能已很完善，结构庞大且复杂，没有专门为学生练习而设计的项目，所以学生在实践时会遇到很多困难，实践效果也不理想。本书正是在这样的背景下编著的。

针对目前操作系统实践性教学中存在的不足，本书向读者提供的是一个专门为操作系统课程教学而设计的操作系统——GeekOS。国外知名大学采用的教学操作系统还有很多，如 OSP/OS161、NACHOS、MINIX、TOYOS 等。相比而言，GeekOS 的设计尽可能简单，让学生易于阅读、设计和添加代码，但它又涵盖了操作系统课程的核心内容，能满足操作系统课程教学的需求。GeekOS 具有的最大优势是精心为学生设计了 7 个由易到难的设计项目，每个项目都对应操作系统的一大管理功能，学生每实现一个项目就相当于完成了操作系统的一个功能，如果完成全部 7 个项目，就实现了一个具备基本功能的实用操作系统。它让学生一步一步循序渐进地实现一个简单的类 UNIX 操作系统，有很好的教学和实际应用价值。

本书内容共分为两个部分：第一部分主要介绍 GeekOS 的设计原理和项目设计环境，由第 1 章~第 7 章组成；第二部分是对 GeekOS 各项目设计的指导性说明，由第 8 章~第 13 章组成。第 1 章是 GeekOS 教学操作系统的概述，介绍了 GeekOS 内核的特点和项目实现时的一些好的编程经验。第 2

章是课程设计环境介绍，包括 Linux 操作系统的安装、各种工具软件的使用、bochs PC 模拟器的使用及如何开始 GeekOS 项目。第 3 章是介绍 make 工具和 makefile 文件规则项目管理文件的使用，它是了解 GeekOS 系统程序组织结构的重要部分。第 4 章详细介绍了 PC 启动原理，实模式到保护模式的转换，GeekOS 启动代码的详细分析。第 5 章介绍了 GeekOS 进程管理的设计原理，包括进程的状态、进程创建、用户态进程和核态进程的不同之处以及进程调度策略等。第 6 章介绍了 Intel 处理器结构的分页存储管理的基本原理和 GeekOS 项目 4 有关的函数代码分析。第 7 章介绍了 GeekOS 文件系统的设计原理，包括虚拟文件系统、PFAT 文件系统和 GOSFS 文件系统的设计与实现。第 8 章～第 13 章的内容分别是对 GeekOS 要求学生完成的项目设计内容和技术原理的介绍，包括熟悉 GeekOS 项目编译运行环境、核态进程的实现、用户态进程的实现、进程调度策略和算法实现、分页存储管理的实现和文件系统的实现等。

在本书的编写过程中，得到了桂林电子科技大学广大教师的支持和帮助，同时清华大学出版社的同志对本书内容的组织提出了宝贵建议，在此表示特别感谢！

本书成文后，王宇英同志做了进一步处理和修改。全书由黄廷辉同志统一策划与设计，并做了最后的校对、完善和统稿。

本书的主要读者是计算机及相关专业的高年级本科生和硕士生，也可供计算机、信息和嵌入式等相关专业的教学、科研和工程技术人员参考。它是一本操作系统理论教材的配套教材，作为对操作系统课程实践的一次尝试，也受作者水平与时间仓促的限制，如书中出现错误与不足，敬请广大读者不吝赐教。

桂林电子科技大学计算机与控制学院

黄廷辉

2007 年 2 月

目录

CONTENTS

第 1 章 GeekOS 教学操作系统概论	1
1.1 引言	1
1.2 GeekOS 教学操作系统	2
1.2.1 GeekOS 概述	2
1.2.2 GeekOS 的存储器管理	3
1.2.3 GeekOS 支持的设备	4
1.2.4 GeekOS 的中断和线程	6
1.2.5 GeekOS 系统引导和初始化	9
1.2.6 GeekOS 系统源代码结构和设计项目	11
第 2 章 课程设计开发环境	15
2.1 Cygwin 介绍	15
2.1.1 Cygwin 简述	15
2.1.2 Cygwin 安装与设置	15
2.1.3 Cygwin 使用	18
2.2 安装 Linux	18
2.2.1 安装虚拟机	19
2.2.2 在虚拟机上安装 Linux	21
2.2.3 安装 VMware Tools 和实现文件共享	26
2.3 工具软件	28
2.3.1 GNU gcc 编译器	28
2.3.2 NASM 汇编器	29
2.3.3 GNU gdb 调试器	31
2.4 Bochs PC 模拟器	34
2.4.1 Bochs 安装和使用	34
2.4.2 在 Bochs 中运行 GeekOS	36
第 3 章 make 工具和 makefile 规则	38
3.1 makefile 文件	38

高等院校计算机实验与实践系列示范教材

3.1.1 makefile 文件内容	38
3.1.2 makefile 规则	39
3.1.3 makefile 文件示例	39
3.1.4 make 工作原理	41
3.1.5 makefile 宏	42
3.1.6 make 隐含规则	43
3.1.7 clean 命令的应用	44
3.2 GeekOS 的 makefile 文件	45
第 4 章 PC 启动原理及 GeekOS 启动程序	51
4.1 PC 启动原理	51
4.1.1 计算机系统启动	51
4.1.2 引导程序	52
4.1.3 内核程序导入	53
4.2 保护模式	54
4.2.1 保护模式	54
4.2.2 实模式和保护模式	55
4.2.3 进入保护模式	58
4.3 GeekOS 启动程序分析	59
4.3.1 fd_boot.asm 代码分析	59
4.3.2 setup.asm 代码分析	64
4.3.3 lowlevel.asm 代码分析	69
第 5 章 GeekOS 进程管理	77
5.1 GeekOS 进程状态及转换	77
5.2 GeekOS 内核进程	78
5.2.1 内核进程控制块	78
5.2.2 GeekOS 系统中最早的内核进程	79
5.2.3 内核进程对象	81
5.3 进程调度	82
5.3.1 内核进程切换	82
5.3.2 用户进程切换	82
5.3.3 GeekOS 进程调度策略	83
5.4 内核进程主要操作函数	84
5.4.1 Init_Thread 函数	84
5.4.2 Create_Thread 函数	85
5.4.3 Destroy_Thread 函数	85
5.4.4 Reap_Thread 函数	86

5.4.5 Detach_Thread 函数.....	86
5.4.6 Start_Kernel_Thread 函数.....	87
5.4.7 Setup_Kernel_Thread 函数.....	87
5.4.8 Make_Runnable 函数.....	88
5.4.9 Make_Runnable_Atomic 函数.....	88
5.4.10 Get_Current 函数	88
5.4.11 Get_Next_Runnable 函数.....	89
5.4.12 Schedule 函数.....	89
5.4.13 Join 函数.....	89
5.4.14 Lookup_Thread 函数.....	90
5.4.15 Wait 函数.....	90
5.4.16 Wake_Up 函数.....	91
5.4.17 Wake_Up_One 函数.....	91
5.4.18 Dump_All_Thread_List 函数.....	92
第 6 章 GeekOS 分页存储管理.....	93
6.1 存储器分页管理机制.....	93
6.2 线性地址到物理地址的转换.....	94
6.2.1 映射表结构	94
6.2.2 表项格式	94
6.2.3 线性地址到物理地址的转换.....	95
6.2.4 不存在的页表	96
6.2.5 页的共享	97
6.3 页级保护和虚拟存储器支持.....	97
6.3.1 页级保护	97
6.3.2 虚拟存储器技术	98
6.4 页故障	99
6.5 GeekOS 分页系统数据结构	100
6.5.1 页目录表和页表项数据结构	100
6.5.2 物理页数据结构和页状态	100
6.6 GeekOS 分页系统主要操作函数	101
6.6.1 Alloc_Page 函数	101
6.6.2 Alloc_Pageable_Page 函数	102
6.6.3 Find_Page_To_Page_Out 函数	103
6.6.4 Free_Page 函数	104
6.6.5 Page_Fault_Handler 函数	105
6.6.6 Print_Fault_Info 函数	105

第 7 章 GeekOS 文件系统	107
7.1 GeekOS 文件系统框架	107
7.2 虚拟文件系统层	108
7.3 高速缓冲区	108
7.4 PFAT 文件系统	110
7.5 PFAT 文件系统操作函数	111
7.5.1 Copy_Stat 函数	111
7.5.2 PFAT_FStat 函数	112
7.5.3 PFAT_Read 函数	112
7.5.4 PFAT_Write 函数	113
7.5.5 PFAT_Seek 函数	114
7.5.6 PFAT_Read_Entry 函数	114
7.5.7 PFAT_Lookup 函数	114
7.5.8 Get_PFAT_File 函数	115
7.5.9 PFAT_Open 函数	116
7.5.10 PFAT_Open_Directory 函数	117
7.5.11 PFAT_Mount 函数	118
7.5.12 Init_PFAT 函数	120
7.5.13 Register_Filesystem 函数	120
7.6 虚拟文件系统函数	121
7.6.1 Unpack_Path 函数	121
7.6.2 Lookup_Filesystem 函数	122
7.6.3 Lookup_Mount_Point 函数	122
7.6.4 Format 函数	123
7.6.5 Mount 函数	123
7.6.6 Open 函数	125
7.6.7 Do_Open 函数	125
7.6.8 Close 函数	126
7.6.9 Read 函数	126
7.6.10 Write 函数	126
7.6.11 Seek 函数	127
7.6.12 Create_Directory 函数	127
7.6.13 Delete 函数	127
第 8 章 GeekOS 设计项目 0	129
8.1 项目设计目的	129
8.2 项目设计要求	129

8.3 GeekOS 键盘处理函数	129
8.4 项目设计提示.....	134
第 9 章 GeekOS 设计项目 1	135
9.1 项目设计目的.....	135
9.2 项目设计要求.....	135
9.3 ELF 文件格式	135
9.3.1 可执行文件	135
9.3.2 ELF (可执行连接格式)	136
9.3.3 ELF Header.....	136
9.3.4 程序头部 (Program Header)	137
9.3.5 节区头部表格 (section header table)	138
9.4 用户可执行程序装入.....	138
9.5 项目设计提示.....	139
第 10 章 GeekOS 设计项目 2	144
10.1 项目设计目的.....	144
10.2 项目设计要求.....	144
10.3 项目设计提示.....	145
10.3.1 GeekOS 的用户态进程	145
10.3.2 用户态进程空间.....	147
10.3.3 用户堆栈空间初始化.....	149
10.3.4 用户态进程创建.....	149
第 11 章 GeekOS 设计项目 3.....	152
11.1 项目设计目的.....	152
11.2 项目设计要求.....	152
11.3 项目设计提示.....	152
11.3.1 GeekOS 进程调度处理过程	152
11.3.2 四级反馈队列调度策略实现	154
11.3.3 进程调度策略评价	156
11.3.4 GeekOS 系统中的进程同步	156
第 12 章 GeekOS 设计项目 4	161
12.1 项目设计目的.....	161
12.2 项目设计要求.....	161
12.3 项目设计提示.....	162
12.3.1 为内核程序空间建立页表	162

12.3.2 为用户进程建立页表.....	163
12.3.3 请求分页技术实现.....	164
12.3.4 进程终止处理.....	166
12.3.5 系统完善处理.....	166
第 13 章 GeekOS 设计项目 5	167
13.1 项目设计目的.....	167
13.2 项目设计要求.....	167
13.3 项目设计提示.....	168
13.3.1 GOSFS 磁盘格式	168
13.3.2 文件与目录	169
13.3.3 GOSFS 文件系统数据结构和操作	170
参考文献.....	175

第 1 章 GeekOS 教学操作系统概论

CHAPTER

1.1 引言

操作系统是管理系统软、硬件资源，控制程序运行，改善人机界面，提供各种服务，合理组织计算机工作流程和为用户有效使用计算机提供良好运行环境的系统软件，它为用户使用计算机提供一个方便、灵活、安全、可靠的工作环境，也是其他应用软件赖以存在的基础。操作系统是计算机系统的重要组成部分，操作系统课程是计算机教育的必修课程，作为计算机专业的核心课程，不但高校计算机相关专业的学生必须学习操作系统，从事计算机行业的从业人员也需要深入了解它。

计算机操作系统课程是理论性和实践性都较强的课程，具有概念多、抽象、涉及面广的特点。在设置操作系统课程教学要求时，教师就要考虑对学生作出什么样的要求。是纯理论的对书本习题和概念作出解答就可以了呢，还是要求学生能动手参与实践。实践也有不同程度的要求，是单纯的实现操作系统的某些算法呢，还是实际编写或修改操作系统功能模块。由于实践环境的限制，许多高校目前都偏重对理论知识的要求，注重基本理论知识的掌握和一些典型算法的实践（一般选择 UNIX 或 Linux 作为实验环境，要求学生用 C 语言编程实现简单的进程创建、进程调度等算法），所以学生基本没有机会去了解、实践操作系统的内部结构和实现技术。实践证明，要真正学好操作系统原理和设计技术，最好的方法就是让学生参与到操作系统的开发工作中。因此，越来越多的高校在开设操作系统理论课程的同时，会要求学生对现有操作系统进行功能改进或再开发，以增加学生对操作系统核心技术的实践，真正做到理论与实践相结合。

那么，能用作学生操作系统课程实践的平台有哪些呢？大家一般很容易想到使用现有的商业操作系统和开放源代码的操作系统，也有很多这样的操作系统可供学生选择，比较流行的有 Linux、Minix 等。虽然也有一些学校确实采用这些操作系统作为实践平台，但采用这些

操作系统存在的缺点也是不容忽视的：这些操作系统一般都结构庞大，过于复杂，学生在短时间内很难理解，而且这些操作系统几乎已经实现了所有的功能（进程管理、存储器管理、文件系统等），不需要学生自行设计或实现一些子系统，因此从教学实践的角度讲，价值不高。最好的方法不是选择一个完整的、实用的、庞大的商业操作系统，而是选择一个既具备基本操作系统核心功能，与实际使用的操作系统比较接近，但又易于理解、规模较小的操作系统作为教学平台，在这个教学平台上，学生可以修改和扩充基本系统以实现更多功能，这种操作系统称为教学操作系统。

当我们决定选用教学操作系统作为我们的操作系统课程实践平台后，剩下的工作就是在现有的多种教学操作系统中选择一种。教学操作系统有两大类，一类是针对 RISC 结构 MIPS 处理器的，另外一类是针对 CISC 结构的 Intel IA-32（或 X86）通用处理器的。这样分类是因为：处理器是操作系统运行的硬件环境中最重要的部分。

针对 RISC 结构 MIPS 处理器的教学操作系统有 Nachos（Not Another Completely Heuristic Operating System）和 OS/161。其中 Nachos 是建立在软件模拟的虚拟机之上的教学操作系统，采用 MIPS R2/3000 的指令集，能模拟主存、中断、网络以及磁盘系统等所必须的硬件系统，美国加州大学伯克利分校多次采用该操作系统作为课程设计平台。OS/161 是运行在与操作系统无关的 System/161 模拟器上的，操作系统代码是 MIPS 对应的机器代码。但无论是 Nachos 还是 OS/161，若学生使用 Windows 或 Linux 开发环境的话，都需要使用交叉编译器才能把代码编译成 MIPS 相应的机器代码。

Minix 和 GeekOS 是针对 CISC 结构的 Intel IA-32（或 X86）通用处理器的。其中，Minix 是 Andrew S. Tanenbaum（AST）于 1987 年开发的，目前主要有 1.5 版和 2.0 版两个版本在使用。Minix 系统是免费的，可以从许多 FTP 上下载，但 Minix 是一个包括了虚拟内存管理、文件系统、设备驱动程序、网络和用户态程序等的比较完整的操作系统，它由两万多行代码组成，对于教学有点过于庞大和复杂，而且由于它已经实现了操作系统的全部基本功能，没有留下合适的练习让学生自己完成。

大家知道，最通用的处理器是 CISC 结构的 Intel IA-32（或 X86）通用处理器，所以选用针对该结构的教学操作系统是比较合适的，我们选用 GeekOS 作为操作系统课程设计平台主要原因还有：它是一个用 C 语言开发的操作系统，学生可以在 Linux 或 UNIX 环境下对其进行功能扩充，也可以在 Windows 下使用 Cygwin 工具进行开发，且其针对进程、文件系统、存储管理等操作系统核心内容分别设计了 7 个难度逐渐增加的项目供教师选择。我们将在后面的章节中详细介绍 GeekOS 教学操作系统。

1.2 GeekOS 教学操作系统

1.2.1 GeekOS 概述

GeekOS 是一个基于 X86 架构的 PC 上运行的微操作系统内核，由美国马里兰大学的教师开发，主要用于操作系统课程设计，目的是使学生能够实际动手参与到一个操作系统的开发工作中。出于教学目的，这个系统内核设计简单，却又兼备实用性，它可以

运行在真正的 X86 PC 硬件平台。作为一个课程设计平台，GeekOS 由一个基本的操作系统内核作为基础，提供了操作系统与硬件之间的所有必备接口，实现了系统引导、实模式到保护模式的转换、中断调用及异常处理、基于段式的内存管理、FIFO 进程调度算法以及内核进程、基本的输入输出（键盘作为输入设备，显示器作为输出设备），以及一个用于存放用户程序的只读文件系统 PFAT。

1.2.2 GeekOS 的存储器管理

GeekOS 内核有两种存储器分配方式，分页分配方式和堆分配方式。

1. 分页分配方式

系统中所有存储器都分成大小相等的块，称作页。在 X86 系统中，页的大小是 4KB。若在 GeekOS 中增加了支持虚拟存储器的功能，页也可以是虚拟存储空间的存储单元。在不支持虚存的系统中，页也可以看作是一个固定大小的存储块，页的分配和回收用函数 Alloc_Page() 和 Free_Page()，这两个函数的定义在头文件<geekos/mem.h>中。在 GeekOS 中每一页都是一个 Page 结构：

```
struct Page {  
    unsigned flags;           /* 页状态 */  
    DEFINE_LINK(Page_List, Page); /* Page_List 页链表指针 */  
    int clock;  
    ulong_t vaddr;           /* 页映射到的用户空间虚拟地址 */  
    pte_t *entry;             /* 指向页表中本页的页表项 */  
};
```

其中 DEFINE_LINK 是在 list.h 文件中的宏定义，定义了指向链表节点的指针。GeekOS 使用宏定义链表结构及操作，具体代码请参考 list.h 文件。系统全局页链表 struct Page_List g_pageList 记录内存所有页的 Page 结构，其中 flags 标记为 PAGE_AVAIL 的页为空闲页，s_freeList 记录系统所有的空闲页。

2. 堆分配方式

堆分配提供不同大小存储块的分配，使用函数 Malloc() 和 Free() 进行存储块的分配和回收。

3. 系统初始化内存布局

系统初始化时由 Init_Mem 函数将系统内存划分为内核空间、可用空间等若干部分，如图 1-1 所示：其中内存空洞是系统设计时留作其他功能使用的，属保留区域。内核堆是一块用于动态分配和回收的内存，系统使用 bget、brel、bpool 三个函数管理这块空间，堆分配方式中的 Malloc 函数与 Free 函数就是通过调用这些函数实现动态分配和回收。可用空闲内存用于分配其他的系统数据。

未使用	0
可用空闲内存	0x1000(4KB, 一页内存)
内核	0x10000
内存空洞	0x0A0000
内核堆 (1MB)	0x100000
可用空间	0x200000
	内存末尾

图 1-1 系统初始化内存布局

1.2.3 GeekOS 支持的设备

1. 文本显示器

文本显示器支持显示文本信息，GeekOS 中的显示驱动仅能支持 VT100 和 ANSI 的一个子集，且不含方向移动和字符特性设置等功能。实现文本显示的函数在</include/geekos/screen.h>头文件中定义。

用户在编程时经常会用到的一个函数是 print()，它是标准 C 语言函数 Printf()的子集，功能是将文本信息输出到显示器。其他还有一些输出函数，如 Put_Char()和 Put_Buf()，使用这两个函数分别可以输出单个字符和字符串。

2. 键盘

键盘设备驱动程序提供了一系列高级接口以使用键盘。用户需要注意的是键盘事件的逻辑关系：用户按键引发键盘中断，键盘中断读取用户按键并将键码放到键盘缓冲区 s_queue 中，而用户进程则将缓冲区的键码读出来作进一步处理。

若用户进程需要从键盘输入信息，可调用 Wait_For_Key()函数，该函数首先检查键盘缓冲区是否有按键，如果有，就读取一个键码，如果此时键盘缓冲区中没有按键，就将进程放入键盘事件等待队列 s_waitQueue，由于用户的按键操作触发了键盘中断，键盘中断处理函数 Keyboard_Interrupt_Handler 就会读取用户按键，将低级键扫描码转换为含 ASCII 字符的高级代码，并刷新键盘缓冲区，最后唤醒等待按键的进程继续运行。若用户按下 Shift、Control、Alt 等键时，也能同样处理。键盘处理程序的代码在头文件</include/geekos/keyboard.h>中，详见第 8 章项目设计 0。

3. 系统时钟

GeekOS 中用户一般不直接使用任何时钟服务，系统时钟主要用于时钟中断，一般用于保证所有的线程都有机会占用 CPU，即线程运行一段时间后会发生时钟中断，调度程序就选择另外的线程运行。

4. 块设备：软盘和 IDE 硬盘

块设备是指按固定大小的块（扇区）存取信息的存储设备，块设备一般作为文件系统的基本存储设备，文件系统会在物理块存储的基础上创建文件、目录等以方便操作。不同块设备的扇区大小不完全一样，但 GeekOS 系统中假设所有块设备的扇区大小都一样——512 个字节，并用宏 SECTOR_SIZE 常量进行了定义。

GeekOS 支持两种块设备：软盘和 IDE 硬盘，系统用名字 fd0 表示第一个软驱，ide0 表示第一个 IDE 硬盘分区，ide1 表示第二个 IDE 硬盘分区。块设备的分区信息用内核的 BLOCK_DEVICE 数据结构表示，用户要使用某个设备的时候只要调用函数 Open_Block_Device()，函数参数就是用户要使用的设备名。打开设备后，用户就可以分别调用 Block_Read() 和 Block_Write() 来读、写设备指定扇区的信息。GeekOS 系统中块设备操作处理过程如图 1-2 所示。

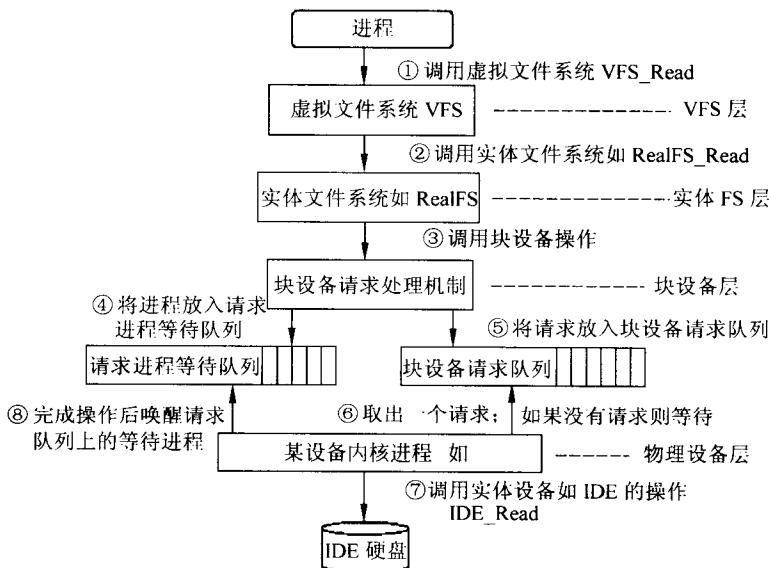


图 1-2 GeekOS 系统块设备操作处理流程

在 GeekOS 系统中，每个块设备都用一个 Block_Device 结构记录：

```

struct Block_Device {
    char name[BLOCKDEV_MAX_NAME_LEN]; // 块设备名称，如 “ide0”
    struct Block_Device_Ops *ops; // 指向块设备操作函数指针列表的指针
    int unit; // 单位磁盘块的大小，如一个硬盘磁盘块为4KB
    bool inUse; // 设备是否在使用
    void *driverData; // 实体块设备信息
    struct Thread_Queue *waitQueue; // 等待访问块设备的进程的队列
    struct Block_Request_List *requestQueue; // 块设备访问请求队列
    DEFINE_LINK(Block_Device_List, Block_Device); // 块设备链表指针
}

```

```
};
```

其中 Block_Device_Ops 结构定义：

```
struct Block_Device_Ops {  
    int (*Open)(struct Block_Device *dev);           // 打开块设备  
    int (*Close)(struct Block_Device *dev);          // 关闭块设备  
    int (*Get_Num_Blocks)(struct Block_Device *dev); // 块号  
};
```

在 GeekOS 设计项目中，系统主要将块设备用作 GeekOS 文件系统的物理载体。在系统初始化时分别将检测到的软盘、硬盘作为块设备注册到系统块设备列表，在注册块设备时，系统将其 Block_Device 结构初始化，并添加到 s_deviceList 设备链表。这个链表由各个设备的 Block_Device 结构指针组成。系统提供了一系列块设备操作函数：Open_Block_Device、Block_Read、Block_Write、Close_Block_Device 等。标准的块设备使用过程顺序是：首先调用 Open_Block_Device 函数打开 s_deviceList 设备列表中指定的设备，然后调用 Block_Read 或 Block_Write 函数执行读写操作，完成操作后调用 Close_Block_Device 关闭块设备。实现块设备存取的函数在头文件 </include/geekos/blockdev.h> 中进行定义。

1.2.4 GeekOS 的中断和线程

1. 中断

中断是用来向 CPU 通知重要事件的，中断的重要特性是：引发 CPU 控制权的转移。中断发生后，线程会暂停执行，而转去执行相应的中断处理程序，中断处理程序实际上是一个 C 语言程序。中断处理结束后，控制权返回给发生中断的线程，大多数情况下，线程会像没有发生中断一样继续执行。但由于中断可能引发线程交换，所以也可能导致共享的内核数据结构被修改等。

下面是一些不同类型的中断及中断处理方法。

- **Exception (异常)**: 若当前运行的线程执行了非法操作，则发生异常。异常是一种同步中断，因为异常的发生是可预知的。这类中断的例子有执行非法指令、被 0 整除等。因为这种异常无法恢复，所以通常都只能采取撤销父线程的方式处理。
- **Faults (故障)**: 故障和异常一样属于同步中断。和异常不同的是，故障通常是可恢复的，内核通过执行一系列操作，去除引起故障的条件，然后使发生故障的线程继续执行即可。缺页中断就是故障的一种，缺页中断是指 CPU 要访问的数据或代码还没有调入内存时发生的中断，当内核找到要访问的内容并调入内存空间时，发生中断的线程就可以继续执行了。
- **Hardware interrupts (硬件中断)**: 外设用硬件中断将某些事件通知 CPU。硬件中断是不可预知的，所以是异步中断，也就是说，异步中断随时会发生。但有时系统可能无法立即处理异步中断，在这种情况下，系统会暂时屏蔽中断直到系统能

处理中断时再处理中断。时钟中断就是一种硬件中断。

- **Software interrupt**（软件中断）：用户态进程用于发出信号表示它需要系统内核的干涉。GeekOS 中仅使用一种软件中断——系统调用。线程用系统调用向内核发出服务请求，如线程要打开文件、执行输入、输出操作、创建新线程等。

2. 中断处理函数

GeekOS 中断调用的实现主要涉及到两个数据结构：中断描述符表 `s_IDT` 与中断处理函数表 `g_interruptTable`。`s_IDT` 在内核中静态分配，每个描述符指向 `lowlevel.asm` 文件中定义的中断向量宏定义程序段，这些程序段定义了 256 个中断向量的调用。其中，0~17 号中断为 Intel CPU 的 CPU 中断向量，18~225 号为用户自定义的中断向量。GeekOS 将大多数中断处理函数初始化为函数 `Dummy Interrupt Handler`，该函数在中断时显示当时保存的计算机各寄存器状态。12~13 号 CPU 中断处理函数为 `GPF Handler`，系统调用使用 `Syscall Handler` 中断处理函数，键盘、计时器等硬件中断也有各自的外部中断处理函数。

3. 线程

在支持线程的系统中，允许多个任务分时共享 CPU。GeekOS 中每个线程都是一个 `Kernel_Thread` 对象，在 `</include/geekos/kthread.h>` 中定义。系统有一个调度程序 (`scheduler`) 用于选择线程运行。任意时刻系统只能有一个线程在运行，这个运行线程就称作当前线程 (`current thread`)，在全局变量 `g_currentThread` 中有一个指针指向当前正在运行进程的 `Kernel_Thread` 对象。如果线程已经准备好可以运行了，但不是正在运行的，就放入准备运行队列 (`run queue`)。线程若在等待某件事情发生则放入等待队列 (`wait queue`)。无论是等待队列还是准备运行队列的线程都用数据结构 `Thread Queue` 记录，`Thread Queue` 是 `Kernel_Thread` 对象中的一个链表结构。

系统中一些线程完全运行于核态，这类线程称为系统线程。系统线程一般用于完成服务请求，如 `reaper` 线程用于释放已经执行结束的线程所占用的资源；软盘和 IDE 接口的磁盘各使用一个系统线程用于 I/O 请求、I/O 操作并将操作结果传输给请求线程。

进程不同于系统线程，进程大多情况下在用户态执行，我们对进程应该很熟悉了，当我们在 Linux 或 Windows 下运行一个程序时，系统都会为程序创建相应的进程。每个进程都占用各自的存储空间、文件、信号量等资源。GeekOS 中的进程其实就是一个简单的线程，每个进程有一个特殊的数据结构 `User_Context`，由于 GeekOS 中的进程是能运行于用户态的普通线程，所以也称为用户线程。由于进程在用户态运行，当发生中断时，系统会切换到核态进行中断处理，当中断处理结束后，系统再返回到用户态继续运行进程。

4. 线程同步

GeekOS 提供了高级线程同步机制：互斥信号量（`mutexes`）和条件变量（`condition variables`），它们的定义在头文件 `</include/geekos/synch.h>` 中。需要注意的是互斥信号量

和条件变量只能用于线程同步，异步中断处理不能访问互斥信号量或条件变量。

互斥信号量用于保护临界区的互斥访问，即同一时刻仅允许一个线程访问临界区，如果一个线程要访问一个互斥信号量，而此时已经有另外一个线程已经在使用该互斥信号量，那该线程只能被阻塞，直到该互斥信号量可以被访问。下面是一个例子。

```
#include <geekos/synch.h>
struct Mutex lock;
struct Node_List nodeList;
void Add_Node(struct Node *node) {
    Mutex_Lock(&lock);
    Add_To_Back_of_Node_List(&nodeList, node);
    Mutex_Unlock(&lock);
}
```

每个条件变量代表一个线程可以等待的条件，每个条件变量与一个互斥信号量相关，当检测或修改与条件变量有关的程序段时，必须保持互斥访问。下面是一个条件变量访问的例子。

```
#include <geekos/synch.h>
struct Mutex lock;
struct Condition nodeAvail;
struct Node_List nodeList;
void Add_Node(struct Node *node) {
    Mutex_Lock(&lock);
    Add_To_Back_of_Node_List(&nodeList, node);
    Cond_Broadcast(&nodeAvail);
    Mutex_Unlock(&lock);
}

struct Node *Wait_For_Node(void) {
    struct Node *node;
    Mutex_Lock(&lock);
    While (Is_Node_List_Empty(&nodeList)) {
        /*等待另外一个线程调用Add_Node()*/
        Cond_Wait(&nodeAvail, &lock);
    }
    node = Remove_From_Front_of_Node_List(&nodeList);
    Mutex_Unlock(&lock);
    Return node;
}
```

5. 线程和中断的交互

理解中断和线程的交互对扩充内核功能是极为重要的。GeekOS 内核是允许抢占的，

这就意味着线程切换可能随时发生。在某个抢占点，由调度程序选择哪个线程运行，一般情况下调度程序会选择最高优先级的运行态线程运行。但线程切换还是会经常发生，如为防止某个线程长时间占用 CPU，系统提供时钟中断。时钟中断会引发异步线程切换；其他硬件中断，如软盘访问中断，也会引发异步线程切换。因为线程切换或中断处理会引起一些共享数据结构的修改，可能导致内核崩溃或其他不可预知的后果。好在系统可以通过屏蔽中断使抢占暂时不可用。调用函数 `Disable_Interrupts()` 就可以屏蔽中断了（函数原型定义在 `</include/geekos/int.h>` 中），一旦调用这个函数，处理器将忽略所有外部硬件中断，其他线程和中断处理程序都不能运行，就可以保证当前线程对 CPU 的控制权，若要允许抢占，只要调用函数 `Enable_Interrupts()` 开中断就可以了。系统在某些情况下也必须屏蔽中断，如执行原子操作（构成操作的一系列指令需要当作一个整体执行，执行过程不能中断）；修改调度程序使用的数据结构时也必须屏蔽中断，如将当前线程放入等待队列时等。下面有一个需要屏蔽中断的例程。

```
/*线程等待*/
Disable_Interrupt();
While (!eventHasOccurred) {
    Wait(&waitQueue);
}
Enable_Interrupt();
```

在这个例子中，线程在等待一个异步事件的发生，在该事件发生前，线程都将自我阻塞在等待队列。当等待的事件发生后，该事件的中断处理程序将 `eventHasOccurred` 标志改为 `true`，并将该进程从阻塞队列移到运行队列。用户可以假设一下，如果在上面的代码中，没有屏蔽中断会怎么样。首先，事件的中断处理可能在检测 `eventHasOccurred` 标志和将线程放入等待队列的中间发生，那就意味着无论等待的事件是否已经发生，线程都将永远等待。第二种可能性是，在线程放入等待队列的过程中，可能会发生线程切换，中断处理程序会将当前运行的进程放入运行队列，而等待队列处于修改（不一致）状态。此时，若运行的代码需要访问等待队列，将引起系统崩溃。幸运的是，在 GeekOS 中，所有需要屏蔽中断才能运行的函数中都有断点，如果没有屏蔽中断就调用这些函数，系统会立即发出提示。所以，在函数代码中，我们经常可以看到这样的语句：`KASSERT(!Interrupts_Enabled())`。这个语句就表示若要继续执行下面的代码，请先屏蔽中断。刚开始写内核代码的时候，用户可能不容易知道运行哪些代码段是需要屏蔽中断的，慢慢的用户就能熟悉了。

1.2.5 GeekOS 系统引导和初始化

GeekOS 系统的开发是基于真正的硬件环境的，因此完全可以在一台 x86 PC 上安装并运行，但还是推荐大家在 PC 模拟器 Bochs 上运行，因为软件模拟硬件的可靠性比真实硬件高得多，不会因为硬件故障而导致系统崩溃，也便于内核程序的调试，不必要经

常关闭和重新启动计算机系统。Bochs 是用 C++ 开发的可移植的 IA-32(x86)PC 模拟器，能够在大部分常见操作系统平台上运行，它包括了对 Intel x86 CPU、通用 I/O 设备和定制 BIOS 等的模拟，当前是由 Bochs 项目组进行维护。在 Bochs 模拟器上也能够运行大部分的操作系统，Linux、Windows、DOS 等。Bochs 可以从网站 <http://bochs.sourceforge.net> 下载，详细的 Bochs 内容将在第 2 章介绍。用户首先启动 Bochs 模拟器，主机操作系统加载 Bochs 程序到主机的内存中。BIOS 加载完成后，在 Bochs 中的操作就会像在真的计算机上操作一样。

1. GeekOS 系统引导

GeekOS 内核编译后，在 build 目录下会生成一个软盘镜像文件 fd.img。在 Bochs 开始运行系统后，会自动检测启动设备。因为软盘首扇区最后一个字在编译时是写入 55AA 数据，而 Bochs 被配置为从软盘启动，这样 Bochs 得以成功地检测到 GeekOS 的启动软盘。之后 Bochs 就会像一台真正的计算机一样，首先导入软盘的首扇区数据到从内存地址 0x7c00 开始的一块内存区，之后跳转到这个地址，开始执行这段首扇区内的程序代码。首扇区内的代码是由位于 /src/geekos 目录中的 fd_boot.asm 编译生成的引导程序。这段汇编程序完成搜索并装载软盘中的 GeekOS 内核二进制文件的功能。在装载完毕后，装载程序执行段间跳转，转入程序 Setup(/src/geekos 目录中的 setup.asm 编译生成)继续执行。

Setup 程序完成装载临时 GDT、IDT 描述符，打开 A20 地址线，初始化 PIC 中断控制器，最后由实模式跳入保护模式。完成了实模式向保护模式的转换之后，Setup 跳转到内核 ENTRY_POINT 入口点。至此，GeekOS 的引导过程结束，内核初始化过程开始。

需要说明的一点是 Setup 除了跳转到保护模式之外，还作了一些内核初始化的准备工作：通过调用 BIOS 中断重置软盘驱动，检测计算机可用内存；之后 Setup 在堆栈创建一个 Boot_Info 结构，并将检测到的内存大小存放到这个数据结构中保存。

2. GeekOS 系统初始化

GeekOS 的内核入口点 ENTRY_POINT 指向的是内核 Main 函数的函数入口，在编译时完成对 ENTRY_POINT 的初始化。Main 函数在 src/src/geekos/main.c 中实现。Main 函数通过调用内核各模块的初始化函数来完成系统内核的初始化，这些函数绝大部分都由原始的 GeekOS 内核提供，不需要自己编制，这大大减轻了 GeekOS 的项目开发难度。

下面简要介绍这些内核各模块的初始化函数，不同项目的初始化函数基本相同。

- Init_BSS() 函数：初始化内核的 BSS 段。
- Init_Screen() 函数：初始化文本显示器，这之后内核可以使用 Print 函数输出文本字符信息。
- Init_Mem(bootInfo) 函数：这里接收的参数 bootInfo 就是之前在 Setup 程序中初始化的 Boot_Info 结构，这个结构保存了 Setup 在实模式检测到的内存大小数据。Init_Mem 函数根据这个值来初始化系统内存，首先将 Setup 中使用的临时 GDT 复制到内核，之后再重新划分内存使用空间，包括划分内核程序所占内存，内核堆内存，系统空闲内存，跳过内存空洞等；系统接着建立了一个系统空闲内存链表。

- `Init_CRC32()`函数：初始化一个 CRC 校验表，这个校验表用于内存分页管理系统的实现，在项目 4 中要用到。
 - `Init_TSS()`函数：初始化 TSS 段描述符及 TSS 段，TSS 段的空间是内核静态分配的，并全部初始化为零值。
 - `Init_Interrupts()`函数：初始化中断向量表以及中断、异常处理的默认函数，之后允许中断。从这里开始内核就可以响应中断了。
 - `Init_VM(bootInfo)`函数：初始化内存分页功能，这个函数用于分页内存管理系统的实现，在项目 4 中要用到。
 - `Init_Scheduler()`函数：初始化内核主进程 `Main`，创建用于管理进程的内核 `Idle` 进程和 `Repair` 进程，初始化全局进程列表。
 - `Init_Traps()`函数：初始化 12 号 CPU 中断异常处理函数，13 号 CPU 中断异常处理函数以及系统调用中断处理函数。GeekOS 系统调用使用 0x90 号中断向量。
 - `Init_Timer()`函数：初始化计时器，包括制定计时器中断周期时间片，初始化时钟中断处理函数；之前在初始化 PIC 外部中断控制器时屏蔽了所有的外部中断，这里最后将时钟中断重新开启。需要注意的是时钟中断处理函数，这个函数负责系统进程的调度。
 - `Init_Keyboard()`函数：初始化键盘输入，包括初始化键盘缓冲区队列，初始化键盘中断处理函数，最后开启键盘中断。
 - `Init_DMA()`函数：初始化 DMA 控制器。
 - `Init_Floppy()`函数：初始化软盘驱动。GeekOS 默认提供一个软驱与两个 IDE 接口硬盘。这里函数添加了用于处理软驱中断的中断处理函数，以及内核进程 `Floppy_Request_Thread` 专门用于处理访问软驱的请求。
 - `Init_IDE()`函数：初始化块设备驱动，这之后系统识别出挂载的硬盘，并添加处理硬盘访问请求的内核进程 `IDE_Request_Thread`。出于系统运行效率考虑，GeekOS 没有为 IDE 中断编制专门的中断处理函数，而是采用直接读取 IDE 控制器状态字的形式判断 IDE 工作的完成状态。
 - `Init_PFAT()`函数：初始化只读文件系统 PFAT。
 - `Init_GOSFS()`函数：初始化文件系统 GOSFS，在项目 5 中要用到。
 - `Mount_Root_Filesystem()`函数：挂载默认的文件系统 PFAT 系统。在后来的项目设计 5 中，用户会看到 GeekOS 最终挂载了 PFAT 与 GOSFS 两个文件系统。
- 至此系统初始化完毕，可以开始运行用户程序了。

1.2.6 GeekOS 系统源代码结构和设计项目

GeekOS 操作系统源文件可以从 <http://geekos.sourceforge.net> 下载，最新版本是 2005 年 4 月的 GeekOS-0.3.0。从下载压缩包解压出来后的 GeekOS 目录结构如图 1-3 所示。

在 `doc` 目录下文件 `hacking.pdf` 和 `index.htm`，是 GeekOS 系统的参考文档。`Scripts` 目录下有 `startProject` 和 `removeEmptyConflicts` 两个脚本文件。GeekOS 系统的源文件在

src 目录下，分 7 个项目：project0、project1、project2、project3、project4、project5 和 project6。每个项目的文件结构都类似，以 project0 为例，结构如图 1-4 所示。

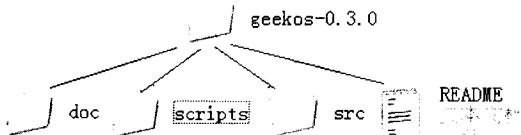


图 1-3 GeekOS 系统主目录

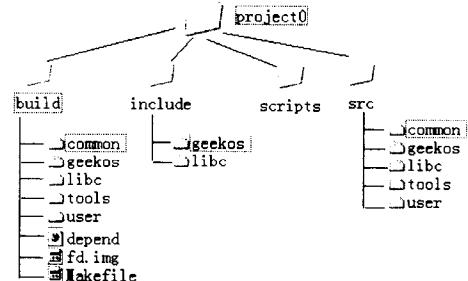


图 1-4 项目文件结构图

在 build 文件夹中，包含系统编译后的可执行文件的文件、软盘镜像 fd.img (project1 等项目中还包含有磁盘镜像 diskc.img)、makefile 项目管理文件。在 Include 文件夹中有 geekos 和 libc 两个子目录，在 geekos 子目录中有 kthread.h、keyboard.h 等头文件，在 libc 中包含有 GeekOS 支持的 C 语言标准函数 string.h 头文件。在 scripts 文件夹中是项目编译时要用到的一些脚本文件。src 文件夹中存放系统内核源代码，用户修改 GeekOS 系统时要修改的源代码如 main.c 等都位于这个目录中。在 User 子目录中一般是用来存放用户的测试文件，在 tools 子目录中的代码是用来建立 pfat 测试文件系统的。

在提供的 GeekOS 内核系统的基础上，为学生设计了 7 个由易到难的设计项目用于 GeekOS 的改进。这些项目分别涵盖了操作系统内核的各个基本模块：系统启动、进程管理、存储管理、文件系统、访问控制以及进程间网络通信。7 个项目都规定了改进的目标，并提供了一些设计指导性的意见，但没有提供源代码，所以学生首先必须熟悉 GeekOS 的基本工作原理，才能开展各个项目的设计与实现。

- 项目 0：主要是让学生熟悉 GeekOS 的编译、运行过程，了解计算机系统的启动原理。项目 0 要求实现一个内核进程，功能是实现从键盘接收一个按键，并在屏幕上显示。
- 项目 1：主要让学生熟悉可执行链接文件（ELF 文件）的结构，并学会加载和运行可执行文件。项目要求学生熟悉 ELF 文件格式，并编写代码对 ELF 文件进行分析，并将分析结果传送给加载器。
- 项目 2：要求学生实现对用户态进程的支持。在项目 2 实现之前，GeekOS 一直使用内核进程。对用户态进程执行的支持包括用户态进程结构的初始化、用户进程空间的初始化、用户进程切换和用户程序导入等。该项目中，存储分配依然使用分段分配方式。实现项目 2 后，用户就可以使用 GeekOS 提供的命令行解释器 Shell 运行一些命令来执行 PFAT 文件系统内的用户测试程序。
- 项目 3：要求学生改进 GeekOS 的调度程序，实现基于 4 级反馈队列的调度算法（初始 GeekOS 系统仅提供了 FIFO 进程调度算法），并实现信号量，支持进程间通信。

- 项目 4：要求学生实现分页虚存管理，以替代在项目 1 和项目 2 中采用的分段存储管理。实现分页虚存管理后。系统在内存不够的情况下就可以将部分页调到硬盘，以释放内存实现虚拟存储技术。
- 项目 5：要求实现 GOSFS 文件系统。由于 GeekOS 使用了虚拟文件系统，可以加载不同的文件系统，而系统默认加载的是 PFAT 只读文件系统。在这个项目中，需要实现一个多极目录的、可读写的文件系统。
- 项目 6：要求为文件系统增加访问控制列表，并使用匿名半双工管道实现进程间通信。

在某种程度上，GeekOS 就是一个简单的 C 程序，有功能函数、线程、内存分配等。但与在装有 Linux 或 Windows 操作系统中运行的 C 语言程序不同，一般 C 语言程序是运行在用户态的，而 GeekOS 是运行在核态的（也称系统态）。在核态运行的程序可以完全控制计算机的 CPU、内存和外部设备，所以编写运行于核态的程序时有一些需要特别注意的问题，修改 GeekOS 内核代码时要特别注意 GeekOS 内核运行环境的一些限制。

1. 有限库函数

因为操作系统是计算机中最底层的软件，内核使用的所有功能函数必须能在内核执行。这与用户程序不同，用户程序可与一系列包含常用函数的标准库连接。

GeekOS 中唯一能使用的标准 C 语言库函数是字符串函数的一个子集（`strcpy()`、`memcpy()` 函数）和 `snprintf()` 函数，这些函数的原型定义在 `<GeekOS/string.h>` 头文件中。除标准 C 语言库函数外，GeekOS 内核还有一些与 C 语言库函数类似的函数，如 `Print()` 函数（函数原型定义在 `<GeekOS/screen.h>` 中），是标准 C 函数 `printf()` 函数的功能子集；`Malloc()` 和 `Free()` 函数则相当于标准 C 的 `malloc()` 和 `free()` 函数（函数原型定义在 `<GeekOS/malloc.h>` 中）。

2. 有限栈空间

GeekOS 内核中的每个线程可以使用大小为 4KB 的栈。如果某个线程堆栈溢出将导致系统内核崩溃。因此，在编程时应谨慎使用栈空间：

- 不要用栈分配大的数据结构，使用栈时尽量使用堆栈分配函数 `Malloc()` 和 `Free()`；
- 程序中不要使用递归，避免函数深层次的调用。

3. 存储器保护限制

一旦系统内核开始运行，没有任何存储器保护，内核的每次存储器访问都是对物理内存单元的访问，即使是引用空指针系统也无法捕获（trapped）。因此，系统在核态运行比在用户态运行时指针引用更容易出问题。要调试运行于核态的程序是比较难的，所以用户必须仔细检查编写的代码，确保没有内存访问错误。

如果用户为 GeekOS 内核增加虚拟内存管理功能，那内核将获得较好的内存访问保护，但用户在编写内核代码时仍然要仔细检查，避免一些不易调试的错误发生。

4. 异步中断

当有重要事件发生的时候，许多硬件设备都用中断来通知 CPU：如定时器定时到、I/O 请求完成等。中断发生时，控制权异步传送给中断处理程序，当中断处理结束时，控制返回到中断点继续执行。值得注意的是，中断处理可能会引起线程交换，这就意味着在控制返回到被中断的线程前，系统可能会执行其他线程。

如果用户遵守上述的限制，在内核模式下编程会相对容易。不仅如此，为使用户更顺利地深入内核编程，还建议用户养成下列编程习惯。

- 使用断点。

在头文件<GeekOS/kassert.h>中，有一个宏定义 KASSERT()，参数是一个布尔表达式。如果表达式的值是 false 时，就打印出一条信息，并暂停内核运行。举例如下：

```
void My_Function(struct Thread_Queue *queue)
{
    KASSERT(!Interrupts_Enabled());      /* 必须禁止中断 */
    KASSERT(queue != 0);                /* 队列不能为空 */
    ...
}
```

用户应尽可能多地用断点检查函数执行的前提条件、后继条件和数据结构的特性等。使用断点有两个好处：第一，若程序执行出错，断点可以在内核崩溃前，精确快速的帮助程序员定位程序中的出错代码；第二，断点还可以帮助程序员检测代码正确性。

- 使用 Print 语句。

在<GeekOS/screen.h>头文件定义了 Print()函数，它支持标准 C 语言函数 printf()的大部分功能。Print 是最常用也是最有效的用于调试内核代码的语句，因为调试时，用户采用的策略都是先作假设，然后再收集证据来支持或反驳假设。

- 尽可能提前、经常测试。

与用户程序相比，内核程序需要更好的可扩展性，所以内核代码常常分成一个个小的模块开发和测试。在系统开发过程中，每一个模块应尽早地、单独地进行测试，提高系统可靠性，当内核开发达到一个稳态时，最好采用版本控制，建议用户使用 CVS 软件来存储内核代码。

GeekOS 项目设计的第一步是要搭建系统开发环境，在开始编写程序代码前，还需要熟悉一些开发工具的安装和使用，掌握一些调试技巧，这样做可以让以后的工作得到事半功倍的效果。GeekOS 系统开发调试环境有多种选择：在 Windows 下使用 Cygwin 模拟 Linux 的开发环境；在 PC 上直接安装 Linux 进行开发调试或者在虚拟机上安装 Linux 进行开发调试等。下面介绍的工具软件中有一些并不是不可或缺的，介绍它们是为读者提供多一些可供选择的方法。

2.1 Cygwin 介绍

2.1.1 Cygwin 简述

Cygwin 是一个在 Windows 平台上运行的 UNIX 模拟环境，是 Cygnus Solutions 公司开发的自由软件。Cygnus 首先把 gcc、gdb、gas 等开发工具进行改进，使它们能生成并解释 Win32 目标文件，然后写了一个共享库（cygwin.dll），把 Win32 api 中没有的 UNIX 风格的调用（如 fork、spawn、signals、select、sockets 等）封装在里面，这样，只要把这些工具的源代码和这个共享库连接到一起，就可以使用 UNIX 主机上的交叉编译器来生成可以在 Windows 平台上运行的工具集。以这些移植到 Windows 平台上的开发工具为基础，Cygnus 又逐步把其他工具软件移植到 Windows 上。所以，Cygwin 对于学习 UNIX/Linux 操作环境，或者从 UNIX 到 Windows 的应用程序移植，或者进行某些特殊的开发工作，尤其是使用 GNU 工具集在 Windows 上进行嵌入式系统开发非常有用。

2.1.2 Cygwin 安装与设置

1. Cygwin 安装

安装 Cygwin 首先要从 <http://cygwin.com> 下载一个安装程序 setup.exe。

运行 setup.exe，有三种安装方式可以选择：从网络设备上安装、只把软件安装包下载到本机和从本机的目录上安装。一般从网络安装会比较慢，所以建议选择第二个选项，即先把需要的安装包下载到本地磁盘，下载完后再从本地目录安装。下面介绍的就是从本机的目录上的安装，如果从网络上安装的操作过程会有一些不同。下载安装包的时候有很多网站可以选择，推荐的网站是：<ftp://ic.laogu.com/down/cygwin.rar>。

安装包下载完成后运行 setup.exe，选择从本机的目录上安装。如图 2-1 所示，单击“下一步”按钮，会出现选择安装路径的界面，如图 2-2 所示。一般我们会将 Cygwin 安装在 C 盘或者 D 盘，另外有 2 个选项 Install for 和 Default Text File Type，在此推荐选择 All users 选项和 UNIX/binary。再下一步就是选择用户存放下载的软件包的目录，建议不要和安装目录在一起。将来如果安装目录出了问题，可以从这个目录重新安装而恢复。选定后，再单击“下一步”按钮，如图 2-3 所示。从图 2-3 我们可以看出，安装包有 Prev（老版本）、Curr（当前版本）、exp（最新版本测试版本）供选择，一般建议大家选择 Curr（当前版本）。在图 2-3 中，可以看到 category 有 all 和 default 两种方式，系统默认的安装（default 方式）不包括 gcc 等软件包，如果硬盘空间足够大，建议大家用完全安装的方式安装 Cygwin。方法是：单击 All 旁边顺时针旋转的箭头，将 Default 变成 Install，然后单击 View 按钮，看到 View 后面显示 full 即可。完全安装大概需要 3.5GB 的空间，安装过程大概需要 1 个小时。选项设置后，单击“下一步”按钮安装就可以了。

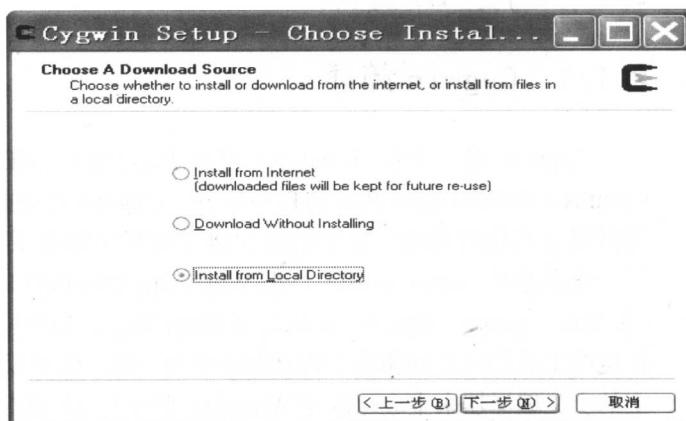


图 2-1 cygwin 安装方式选择

2. Cygwin 设置

Cygwin 安装完成后，要进行简单的设置。安装目录下有一个 cygwin.bat 文件，里面已设置好最重要的环境变量，用户可以编辑该文件。其中，CYGWIN 变量用来针对 Cygwin 运行时系统进行多种全局设置的，开始时，可以不设置 CYGWIN 或者在执行 bash 前用类似下面的格式在 dos 框下把它设为 tty：

```
C:\> set CYGWIN=tty notitle glob
```

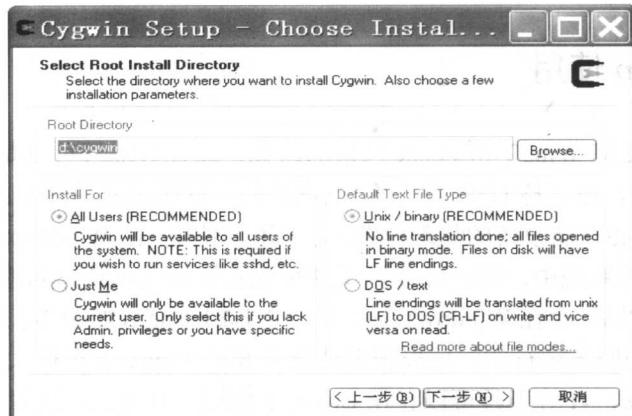


图 2-2 安装路径选择



图 2-3 选择安装包

PATH 变量作为搜索可知性文件的路径列表，当一个 Cygwin 进程启动时，该变量被从 Windows 格式转换成 UNIX 格式。如果想随时能够使用 Cygwin 工具集，PATH 应该包含 x:\cygwin\bin，其中 x:\cygwin 是 Cygwin 安装目录。HOME 变量用来指定主目录，推荐在执行 bash 前定义该变量。当 Cygwin 进程启动时，该变量也被从 Windows 格式转换成 UNIX 格式，如 HOME 值为 C:\ (dos 命令 set HOME 就可以看到它的值，set HOME=XXX 可以进行设置)，在 bash 中用 echo \$HOME 查看，其值为/cygdrive/c。TERM 变量指定终端形态，默认设置为 Cygwin。

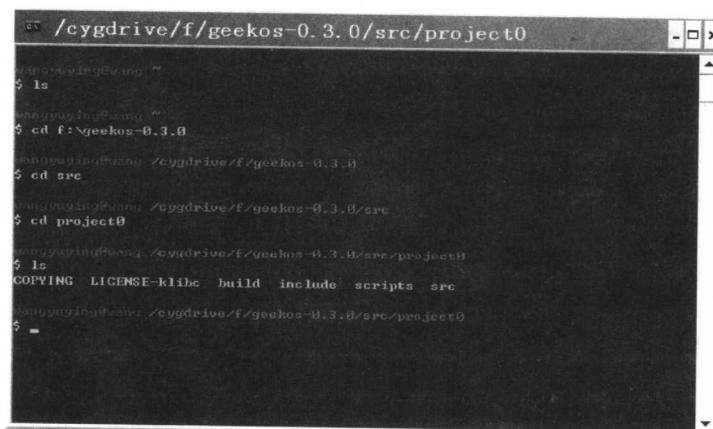
Cygwin 程序默认可以分配的内存不超过 384 MB (program+data)，多数情况下不需要修改这个限制。然而，如果需要更多实际或虚拟内存，应该修改注册表的 HKEY_LOCAL_MACHINE 或 HKEY_CURRENT_USER。添加一个 DWORD 键 heap_chunk_in_mb 并把它的值设为需要的内存，单位是十进制 MB。也可以用 Cygwin 中的 regtool 完成该设置。

```
regtool -i set /HKLM/Software/Cygnus\Solutions\Cygwin/heap_chunk_in_mb 1024
regtool -v list /HKLM/Software/Cygnus\Solutions\Cygwin
```

2.1.3 Cygwin 使用

Cygwin 同时支持 Win32 和 Posix 风格的路径，路径分隔符可以是正斜杠 (/)，也可以是反斜杠 (\)，还支持 UNC 路径名（UNC 是一种确定文件位置的方法，使用这种方法用户可以不关心存储设备的物理位置，方便用户使用。在 Windows 操作系统、Novell Netware 和其他操作系统中，都已经使用了这种规范以取代本地命名系统）。

启动 Cygwin 以后，会在 Windows 下得到一个 Bash Shell，如图 2-4 所示，由于 Cygwin 是以 Windows 下的服务运行的，所以很多情况下和在 Linux 下有很大的不同，但 Linux 的命令，gcc、make、Vi 等工具可以和在 Linux 下使用一样方便，所以在 Cygwin 下面开发 GeekOS 项目需要大家学习的不多，有关 Cygwin 的更详细信息大家可以参考 <http://cygwin.com/cygwin-ug-net/highlights.html>。



The screenshot shows a terminal window titled 'C:\cygdrive\f\geekos-0.3.0\src\project0'. The user has run several commands:

```
$ ls
$ cd f:/geekos-0.3.0
$ cd src
$ cd project0
$ ls
COPYING LICENSE klibc build include scripts src
```

图 2-4 cygwin 运行界面

2.2 安装 Linux

除了使用 Cygwin 模拟 UNIX 环境外，大家也可以使用 Linux 操作系统环境。Linux 是一套免费使用和自由传播的类 UNIX 操作系统，这个系统是由全世界各地成千上万的程序员设计和实现的。它以高效性和灵活性著称，并且能够在 PC 计算机上实现全部的 UNIX 特性，具有多任务、多用户的能力。Linux 之所以受到广大计算机爱好者的喜爱，主要原因有两个：一是它属于自由软件，用户不用支付任何费用就可以获得它和它的源代码，并且可以根据自己的需要对它进行必要的修改；另一个原因是，它具有 UNIX 的全部功能。如果使用 Linux 作为 GeekOS 开发调试环境，推荐使用 Red Hat 7.0 以上的 Linux 版本，在这个版本下进行 GeekOS 开发和调试几乎不需要人为进行任何配置。其余新版本的 Linux 所携带的工具理论上是与旧版本兼容的，也可以使用。

Linux 操作系统的安装有两种方式，一种是在硬盘上直接分一个足够大的分区专门

安装 Linux 操作系统，采用这种方式可以安装一个完整的 Linux 操作系统，使用 Linux 所有功能。另一种方式是在 Windows 环境下先安装一个 PC 虚拟机，然后在虚拟机上安装 Linux 操作系统。为了便于大家在 Linux 和 Windows 交叉环境中编译和调试 GeekOS 系统，建议大家采用第二种方法。

虚拟机（virtual machine）的概念比较广泛，通常人们接触到的虚拟机概念有 VMware 那样的硬件模拟软件，也有 JVM 这样的介于硬件和编译程序之间的软件。简单而言，虚拟机是一个抽象的计算机，和实际的计算机一样，具有一个指令集并使用不同的存储区域。它负责执行指令，还要管理数据、内存和寄存器。通过虚拟机软件，用户可以安装操作系统、安装应用程序、访问网络资源等。对于用户而言，它只是运行在物理计算机上的一个应用程序，但是对于在虚拟机中运行的应用程序而言，它就是真正的计算机，是支持多操作系统并行运行在单个物理服务器上的一种系统。因此，在虚拟机中进行软件调试，系统可能一样会崩溃，但是，崩溃的只是虚拟机上的操作系统，而不是物理计算机上的操作系统，使用虚拟机的“Undo”（恢复）功能，就马上可以恢复。

目前 PC 虚拟机有 Bochs、Virtual PC 和 VMware 三种开源软件。建议大家采用后两种虚拟机软件来安装 Linux 操作系统，因为它们可以很方便地实现与 Windows 系统进行文件共享，这对于以后在进行 GeekOS 项目开发和调试系统时要频繁在 Linux 和 Windows 环境之间切换是很有益的。但由于目前的 GeekOS 操作系统只能在 Bochs 虚拟机上调试运行，所以还必须安装第二个 PC 虚拟机——Bochs。如果大家有兴趣，也可以把 GeekOS 系统移植到 Virtual PC 和 VMware 虚拟机上去调试运行。

2.2.1 安装虚拟机

1. VMware 的运行原理

由于 GeekOS 项目开发时需要使用虚拟机的功能不多，用户是安装 Virtual PC，还是 VMware 都可以。而从使用习惯上和系统运行的性能来讲，多数用户倾向于 VMware，在此只介绍 VMware。VMware 是一个具有创新意义的应用程序，通过 VMware 独特的虚拟功能，用户可以在同一个窗口运行多个全功能的虚拟机操作系统，而且 VMware 中的 GuestOS 直接在 X86 保护模式下运行，使所有的虚拟机操作系统就像运行在单独的计算机上一样。因此，VMware 在性能上有十分出色的表现。熟悉 Linux 的用户可能会想到 Linux 下的模拟器——Wine，但它们有本质上的区别，Wine（Wine Is Not an Emulator）是一个在 Xwindows 和 Linux 之上的，提供了 Windows 3.x 和 Windows 9x API 函数接口，它是一个 Windows 兼容层，这个层提供了一个用来从 Windows 进出到 UNIX 的开发工具包（Winelib），也提供了一个程序加载器，简单的说，Wine 是一个 Linux 下 Windows 应用程序模拟器，而不能独立的运行一个全功能的操作系统。目前 Wine 仍在发展阶段，仅能执行少部分的 Windows 软件，大部分的软件仍然无法正常执行。如图 2-5 所示，这是 VMware 的运行原理图。

2. 下载及安装

目前，VMware 的最新版本是 VMware-workstation-5.5.3-34685，用户可以到 VMware

的站点：<http://www.vmware.com/> 去了解更详细的内容，确定 VMware 是否支持用户的计算机硬件。下载的时候注意要选择 VMware 的版本，国内下载地址为：[http://www.vmware.cn/soft>List_12.html](http://www.vmware.cn/soft/List_12.html)（Windows 版本）。VMware 是一个商业软件，如果要想运行 VMware，还必须到它的公司站点上申请一个可以免费试用 30 天的 License（许可证）。同时也可下载 VMware workstation 5.5.3 34685 汉化补丁对软件进行汉化。

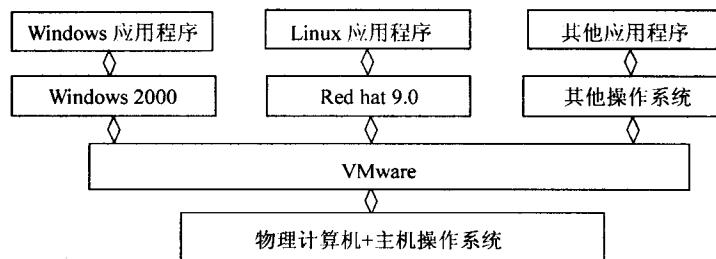


图 2-5 VMware 运行原理

下载 VMWare 软件包解压后根据提示就可以正确安装 VMWare 到用户的计算机硬盘中，相信安装过 Windows 软件的用户都能正确完成这一步。安装完成后桌面会出现 VMware-workstation 图标。

3. 建立虚拟机

VMware 安装完成后，利用它就可以建立虚拟机，每新建一个虚拟机，都会要求建立一个配置文件，这个配置文件相当于电脑的“硬件配置”表，用户可以在配置文件中决定虚拟机的硬盘如何配置、内存多大、准备运行哪种操作系统、是否有网络等。配置 Linux 虚拟机的步骤如下。

(1) 用户双击桌面的 VMware-workstation 图标，会出现如图 2-6 所示的 VMware-workstation 主界面。

(2) 选择 File 菜单下的 New Virtual Machine，出现新虚拟机向导后单击“下一步”按钮，选择 Typical 典型安装。

(3) 再单击“下一步”按钮，在选择操作系统界面的 Guest Operation System 中选择 Linux，然后单击 Version 对应的下拉菜单选择具体的 Linux 版本，如选择 Red Hat Linux。

(4) 单击“下一步”按钮进入安装目录选择界面。该界面上面的文本框是系统的名字，保持默认值即可，下面的文本框需要选择虚拟机操作系统的安装位置。

(5) 根据需要选择好后，单击“下一步”按钮，出现设置虚拟机内存大小的界面。Linux 9.0 对内存的要求是：文本模式至少需要 64MB；图形化模式至少需要 96MB，推荐使用 128MB 以上。

(6) 单击“下一步”按钮进入网络连接方式选择界面。VMware 有四种网络设置方式，一般来说选择 Bridged 方式。Bridged 方式使虚拟机就像网络内一台独立的计算机一样，最为方便好用（四种连网方式的区别大家可参考 VMware 的有关资料）。

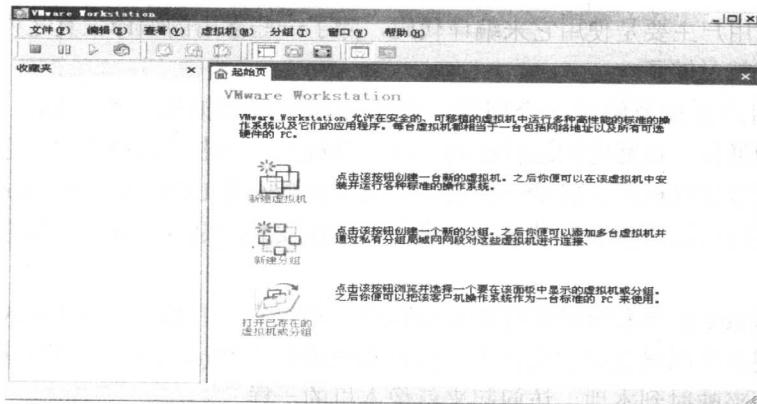


图 2-6 VMware-workstation 主界面

(7) 单击“下一步”按钮进入虚拟磁盘的设置界面。这里有三种方式 (Create a new virtual disk、Use an existing virtual disk、Use a physical disk) 可供选择，建议初学者选择 Create a new Virtual disk，其含义是新建一个虚拟磁盘，该虚拟磁盘只是主机上的一个独立文件。

(8) 单击“下一步”按钮，进入指定虚拟机磁盘的容量界面，这应该根据用户计算机的物理硬盘分区的大小来设置，当然最好不要小于 2GB 的空间。

(9) 单击“下一步”按钮，进入文件存放路径选择界面。在此界面可单击 Browse 按钮进行设置。如果使用默认值，单击“完成”按钮，系统将返回 VMWARE 主界面，如图 2-7 所示的界面。至此，完成了一个 Linux 操作系统虚拟机的建立。

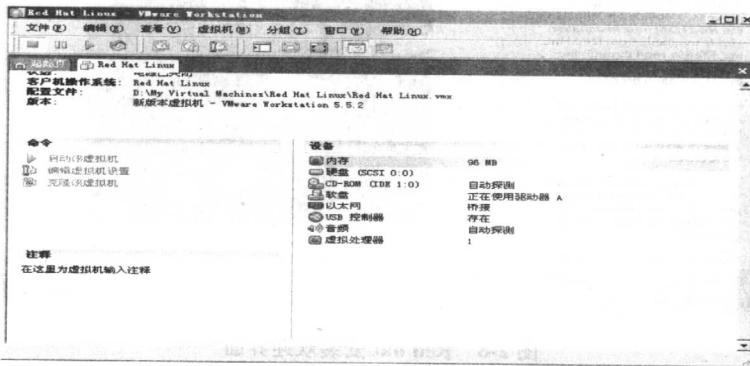


图 2-7 VMware-workstation 主界面

2.2.2 在虚拟机上安装 Linux

建立 Linux 操作系统虚拟机后，下一步就是在虚拟机上安装 Linux 操作系统。而在虚拟机上安装 Red Hat 9.0 比在真正的计算机上安装费时要多一些，毕竟，虚拟机中间隔了一层，速度受到了一定的影响。不过，用户完全可以选择只安装必要的组件，因为完

成课程设计时用户主要是使用它来编译代码，除了一些常用的编译工具之外，只需要跟 Windows 通信就足够了。

现在，用户又需要解决一个问题，就是如何实现虚拟机上的 Linux 和主机上的 Windows 之间通信。虽然它们运行在同一台计算机上，但虚拟机和宿主机之间的通信通常情况下跟两台计算机的情况是一样的。如果虚拟机安装 DOS 或者 Windows 的话会好一些，两者之间可以直接设置共享文件夹，但现在安装的是 Linux，不能直接实现文件共享。

幸好在 Linux 操作系统中带有 Samba 组件，通过它，虚拟机上的 Linux 和主机上的 Windows 之间就可以设置共享文件夹，访问起来就好像 Windows 之间的互相访问一样，若把网络文件夹映射到本地，访问起来就像本机的一样。

现在用户应该知道，安装 Linux 时有两个组件是不可或缺的，一个是我们要编译 GeekOS 项目时的编译器 GCC 和 NASM，另一个是 Samba 相关组件。下面就可以在虚拟机上安装 Linux 操作系统了。

(1) 选中 Linux 虚拟机，插入 Red hat 9.0 安装光盘，单击 VMWARE 工具栏中的 Power ON 按钮，启动 Linux 虚拟机进入如图 2-8 所示的安装欢迎界面。

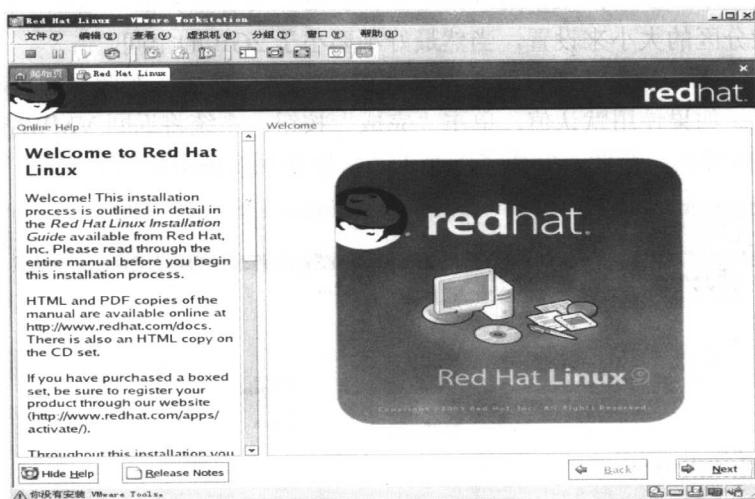


图 2-8 Red hat 安装欢迎界面

(2) 单击 Next 按钮，出现如图 2-9 所示的界面。

在选择安装类型时，一定记得选择“定制”选项，以便于安装最小 Linux 系统。

(3) 单击“下一步”按钮，出现如图 2-10 所示的“选择软件包组”界面。在这里，有几项是需要确定选中的。

选择“Windows 文件服务器”复选框，如图 2-10 所示。

单击“细节”链接，可以看到，这里的“Windows 文件服务器”指的就是 Samba。如图 2-11 所示。



图 2-9 安装类型选择界面

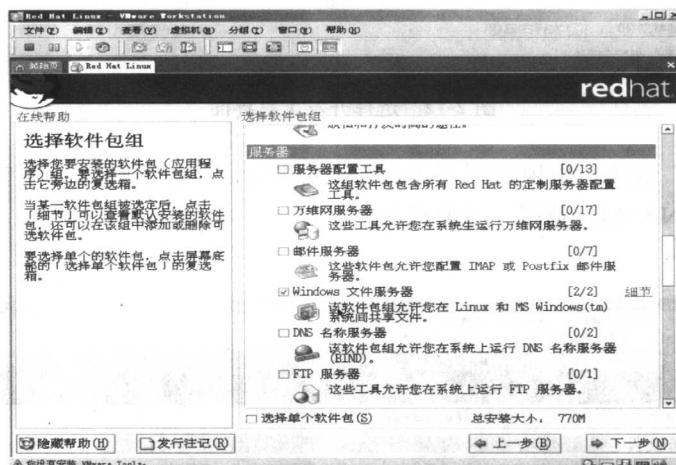


图 2-10 选择 Windows 文件服务器界面

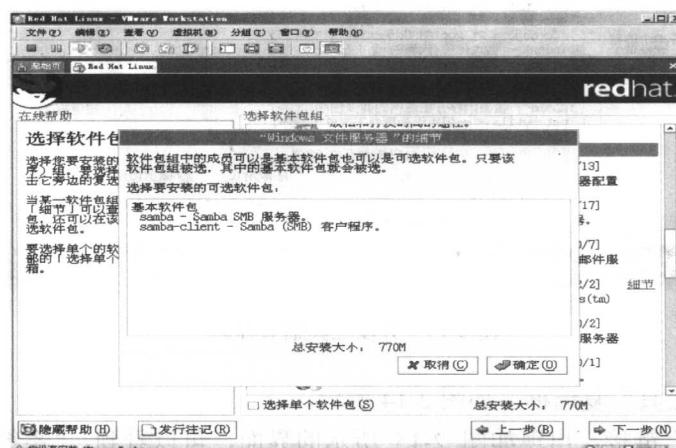


图 2-11 Windows 文件服务器细节界面

选择“开发工具”复选框，如图 2-12 所示。

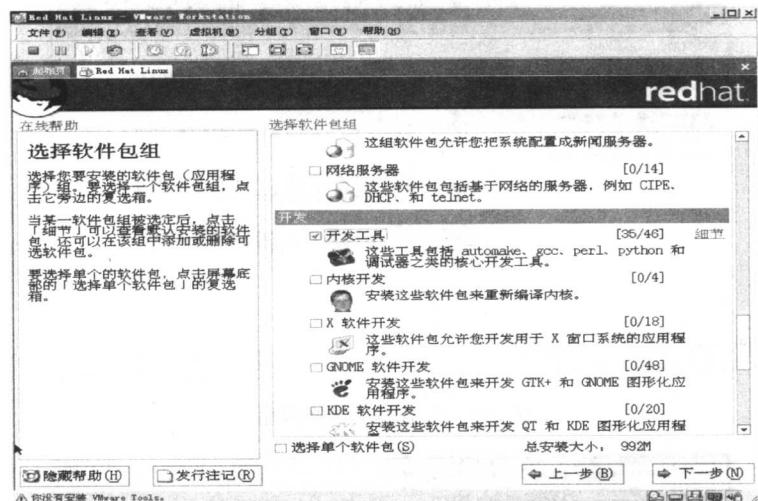


图 2-12 选择开发工具界面

单击“细节”链接，出现如图 2-13 所示的界面，可以看到，开发工具中不但包含 GCC，而且还有 NASM，记得一定要选中它。

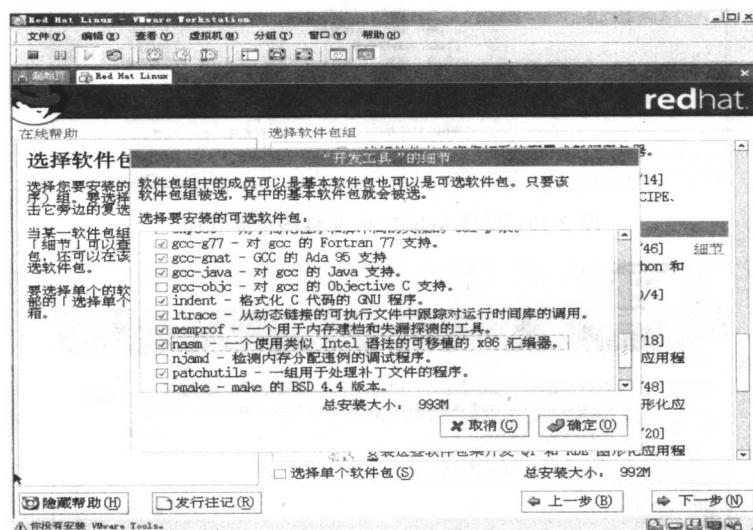


图 2-13 开发工具细节界面

选择“系统工具”复选框，如图 2-14 所示。

单击“细节”链接，出现如图 2-15 所示的界面，可以看到，“系统工具”中包含 semba-client。

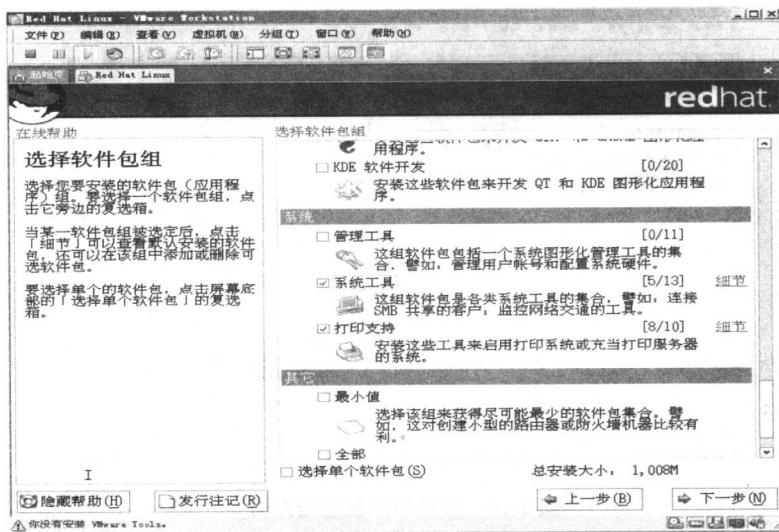


图 2-14 选择系统工具界面

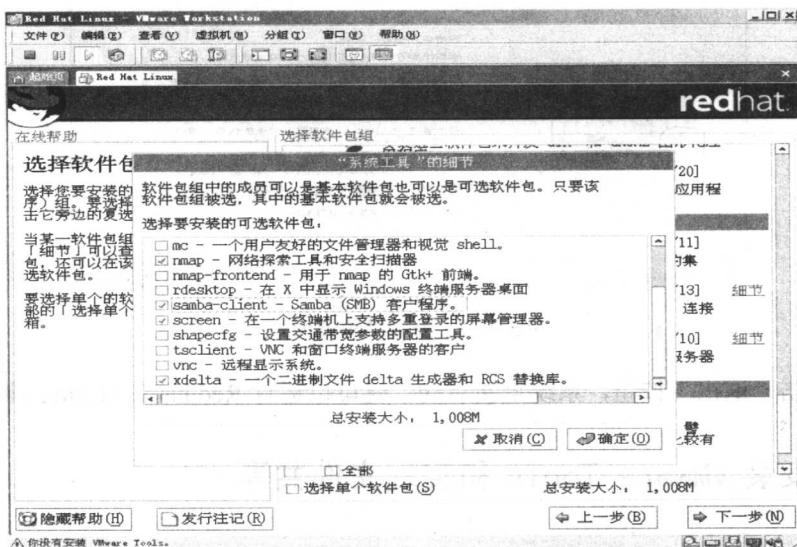


图 2-15 选择系统工具细节界面

对于其他各个选项，尽可能不选，就 GeekOS 项目开发来说，这些选项已经足够了。

(4) 在图 2-10 中单击“下一步”按钮，出现如图 2-16 所示的界面。Linux 系统软件安装开始，整个过程所需的时间随计算机的配置不同而不同，大概需要 30 分钟左右。

(5) 安装进程结束后，单击“下一步”按钮，出现如图 2-17 所示的界面。

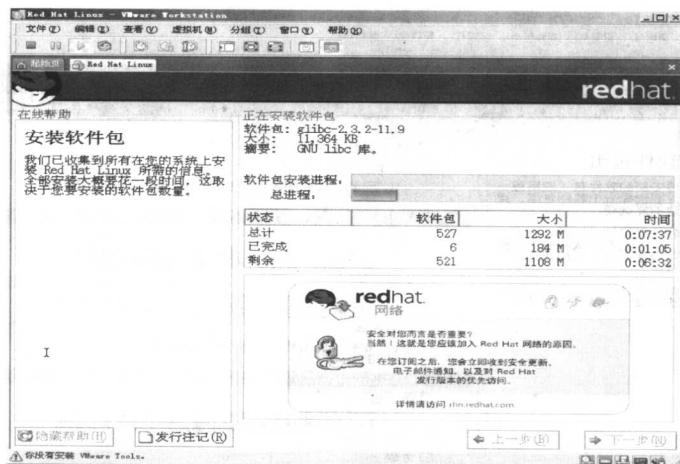


图 2-16 软件包安装进程界面

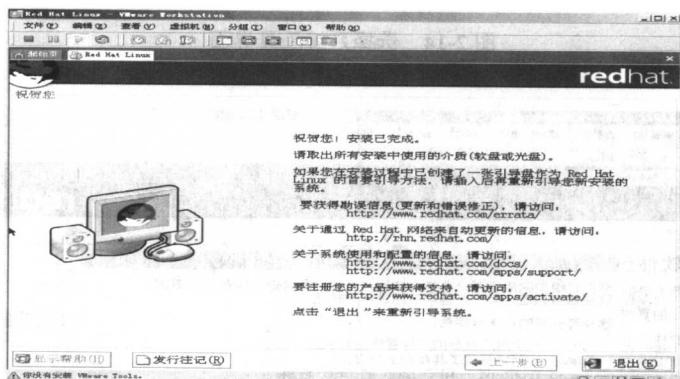


图 2-17 安装结束界面

(6) 单击“退出”按钮，系统重新启动，就可以运行 Red hat 9.0 Linux 操作系统了。

2.2.3 安装 VMware Tools 和实现文件共享

安装完 Linux 之后，还需要在此虚拟机安装 VMware Tools。这相当于给 Linux 安装各种驱动程序。安装步骤具体如下。

(1) 首先要在虚拟机上安装好 Linux 操作系统，以 root 身份进入 Linux，按 Ctrl+Alt 组合键（它是鼠标在虚拟机操作系统和主机操作系统之间切换的组合功能键），进入主机操作系统，单击 VM 菜单下的 VMware Tools Install 子菜单。进入 Linux 的桌面，如图 2-18 所示，单击 VMware 上虚拟机选项——安装 vmware 工具。

(2) 这时在 Linux 的桌面上会出现 VMware Tools 光驱的图标，单击图标（或者直接进 /mnt/cdrom 目录）里面有 VMware Tools 的两个文件，一个是 rpm 类型文件，另一个是 tar.gz 类型文件。这时系统并没有真正的安装上了 VMware Tools 软件包。



图 2-18 选择安装 VMware Tools 界面

(3) 双击直接安装 rpm 文件，把后缀为 tar.gz 的文件复制到/temp（或者用户目录）目录。

(4) 打开命令终端，用 cd 命令转移到存放 tar.gz 的文件目录/temp 下，执行如下命令解压软件包：tar zxf vmware-linux-tools.tar.gz。默认解压到 vmware-linux-tools 目录下（与文件名同名）。

(5) 进入到 vmware-linux-tools（进入刚解压的文件目录）目录下，执行安装 VMware tools 命令：./vmware-install.pl。在屏幕的提示下，一直按回车直至执行完成。至此，VMware Tools 安装完成，虚拟机的 Linux 和主机的 Windows 可以实现文件共享了。

(6) 这时在/mnt 目录下会出现一个 hgfs 的子目录，它就是用户要实现的共享目录。在 vmware workstation 菜单的虚拟机中单击设置，在出现的界面中单击“选项”选项卡，出现如图 2-19 所示界面。单击共享文件夹后，在出现的界面中单击“添加”按钮，就可以任意添加想要共享的文件夹。每增加一个共享文件夹，在 Linux 系统的/mnt/hgfs 目录下就会出现一个共享的文件夹的子目录。

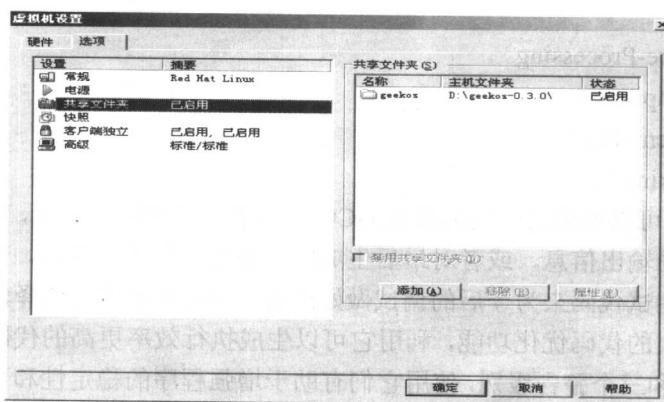


图 2-19 添加共享文件夹界面

2.3 工具软件

编译 GeekOS 需要用到很多工具。如果在 Linux/X86 或 FreeBSD/X86 系统，或在 Windows 系统运行 Cygwin，那大多数需要的工具已具备了。下面是编译 GeekOS 需要的工具软件列表：

- gcc <<http://gcc.gnu.org/>> 2.95.2 版本及以上。
- GNUbinutils <<http://www.gnu.org/software/binutils/>>。
- GNU Make <<http://www.gnu.org/software/make/>>。
- Perl <<http://www.perl.org/>> 5.0 版本及以上。
- egrep <<http://www.gnu.org/software/grep/grep.html>>。
- AWK <<http://www.gnu.org/software/gawk/>>。
- diff3 <<http://www.gnu.org/software/difftools/>>。
- NASM<<http://nasm.sourceforge.net/> (在 Linux 操作系统安装时系统唯一需要安装的软件组件)>。

2.3.1 GNU gcc 编译器

在 Linux 环境下开发应用程序时，大多数情况下使用的都是 C 语言，因此几乎每一位 Linux 程序员面临的首要问题就是如何灵活运用 C 编译器。目前 Linux 下最常用的 C 语言编译器是 GCC (GNU Compiler Collection)，它是 GNU 项目中符合 ANSI C 标准的编译系统，能够编译用 C、C++ 和 Object C 等语言编写的程序。GCC 不仅功能非常强大，结构也异常灵活。最值得称道的一点就是它可以通过不同的前端模块来支持各种语言，如 Java、Fortran、Pascal、Modula-3 和 Ada 等。

开放、自由和灵活是 Linux 的魅力所在，而这一点在 GCC 上的体现就是程序员通过它能够更好地控制整个编译过程。在使用 GCC 编译程序时，编译过程可以被细分为四个阶段：

- 预处理 (Pre-Processing)。
- 编译 (Compiling)。
- 汇编 (Assembling)。
- 链接 (Linking)。

Linux 程序员可以根据自己的需要让 GCC 在编译的任何阶段结束，以便检查或使用编译器在该阶段的输出信息，或者对最后生成的二进制文件进行控制，以便通过加入不同数量和种类的调试代码来为今后的调试做好准备。和其他常用的编译器一样，GCC 也提供了灵活而强大的代码优化功能，利用它可以生成执行效率更高的代码。GCC 提供了 30 多条警告信息和三个警告级别，使用它们有助于增强程序的稳定性和可移植性。此外，GCC 还对标准的 C 和 C++ 语言进行了大量的扩展，提高程序的执行效率，有助于编译器进行代码优化，能够减轻编程的工作量。

1. GCC 起步

在学习使用 GCC 之前，下面的这个例子能够帮助用户迅速理解 GCC 的工作原理，并将其立即运用到实际的项目开发中去。

首先用熟悉的编辑器输入清单 1 所示的代码。

清单 1：

```
/*hello.c*/
#include "stdio.h"
int main(void)
{
    printf ("Hello world, Linux programming!\n");
    return 0;
}
```

然后在 shell 中执行下面的命令编译和运行这段程序就可以得到结果。

```
# gcc hello.c -o hello      '编译链接得到执行程序
# ./hello                      '运行程序
输出结果: Hello world, Linux programming!
```

2. GCC 常用选项

GCC 作为 Linux 下 C/C++重要的编译环境，功能强大，编译选项繁多。为了方便大家日后编译方便，在此将常用的选项及说明罗列出来如下：

- o FILE 指定执行文件名为 FILE，否则默认为 a.out。
- c 通知 GCC 取消链接步骤，即编译源码并在最后生成目标文件。
- Dmacro 定义指定的宏，使它能够通过源码中的#define 进行检验。
- E 不经过编译预处理程序的输出而输送至标准输出。
- g 获得有关调试程序的详细信息，它不能与-o 选项联合使用。
- Idirectory 在包含文件搜索路径的起点处添加指定目录。
- llibrary 提示链接程序在创建最终可执行文件时包含指定的库。
- O 将优化状态打开，该选项不能与-g 选项联合使用。
- S 要求编译程序生成来自源代码的汇编程序输出。
- v 启动所有报警提示。
- Werror 在发生警报时取消编译操作，即把报警当作是错误。
- w 禁止所有的报警提示。

GCC 是在 Linux 下开发程序时必须掌握的工具之一，要详细掌握 GCC 工具的所有功能，请用户阅读 GCC 的技术文档。

2.3.2 NASM 汇编器

1. 什么是 NASM

NASM 是一个为可移植性与模块化而设计的一个 80x86 的汇编器，支持相当多的目

标文件格式。包括 Linux 和 NetBSD/FreeBSD 的 a.out、ELF、COFF、微软 16 位和 32 位的 OBJ、还可以输出纯二进制代码文件。NASM 的语法设计和 Intel 语法相似，比较简洁易懂，支持 Pentium、P6、MMX、SSE 和 SSE2 指令集，NASM 当初被设计出来的想法从本质上讲，是因为当时没有一个好的免费的 X86 体系的汇编器可以使用，为了使开源软件项目活动顺利开展，必须为用户提供一个好用的、免费的汇编器。

- a86 不错，但不是免费的，而且不可能得到 32 位代码编写的功能，除非你付费，它只使用在 DOS 上。
- gas 是免费的，而且在 DOS 下和 UNIX 下都可以使用，但是它是作为 gcc 的一个后台而设计的，并不是很好。gcc 一直就提供给它绝对正确的代码，所以它的错误检测功能相当弱，还有就是对于任何一个想真正利用它写点程序的用户来讲，它的语法非常麻烦，并且无法在里面书写正确的 16 位代码。
- as86 是专门为 Minix 和 Linux 设计的，但看上去并没有很多文档可以参考。
- MASM 不是很好，相当贵，并且只能运行在 DOS 下。
- TASM 功能强一些，它的语法本质上就是 MASM 的。它也是相当贵的，并且只能运行在 DOS 下。

2. NASM 安装

1) 在 DOS 和 Windows 下安装 NASM

如果拿到了 NASM 的 DOS 安装包 nasmXXX.zip（这里.XXX 表示该安装包的 NASM 版本号），把它解压到它自己的目录下（比如：c:\nasm）该包中会包含有 4 个可执行文件：NASM 可执行文件 nasm.exe，nasmw.exe，还有 NDISASM 可执行文件 ndisasm.exe，ndisasmw.exe。文件名以 w 结尾的是 Win32 可执行格式，是运行在 Windows 系统的 Intel 处理器上的。另外的是 16 位的 DOS 可执行文件。NASM 运行时需要的唯一文件就是它自己的可执行文件，所以可以复制 nasm.exe 和 nasmw.exe 的其中一个到工作路径下，或者可以编写一个 autoexec.bat 把 nasm 的路径加到 PATH 环境变量中去。甚至文件名也可以重新命名（如果安装了 Win32 版本，可能希望把文件名改成 nasm.exe）。

2) 在 Linux 下安装 NASM

如果下载的是 Linux 下的 NASM 源码包 nasm-x.xx.tar.gz（这里 x.xx 表示该源码包中的 nasm 的版本号），把它解压到一个目录，比如/usr/local/src。包被解压后会创建自己的子目录 nasm-x.xx，NASM 是一个自动配置的安装包：一旦解压了它，改变到它的目录下，输入./configuer，该脚本就会找到最好的 C 编译器来构造 NASM，并据此建立 Makefiles。一旦 NASM 被自动配置好后，就可以输入 make 命令来构造 nasm 和 ndisasm 二进制文件，然后输入 make install 把它们安装到/usr/local/bin，并把 man 页（帮助文档）安装到/usr/local/man/man1 下的 nasm.1 和 ndisasm.1，或者可以给配置脚本一个--prefix 选项来指定安装目录，或者也可以自己来安装。

在安装好的 Linux 系统中，一般都已带有 NASM 组件，但系统默认是不安装的。可以使用安装组件的形式来安装 NASM 汇编器。

3) NASM 的简单应用

要汇编一个文件，可以以下面的格式执行一个命令：

```
#nasm -f <format> <filename> [-o <output>]
```

例如：

- 把文件 myfile.asm 汇编成 ELF 格式的文件 myfile.o。

```
#nasm -f elf myfile.asm
```

- 把文件 myfile.asm 汇编成纯二进制格式的文件 myfile.com。

```
#nasm -f bin myfile.asm -o myfile.com
```

- 想要以十六进制代码的形式产生列表文件输出，并让代码显示在源代码的左侧，使用-l 选项并给出列表文件名，比如：

```
#nasm -f coff myfile.asm -l myfile.lst
```

- 想要获取更多的关于 NASM 的使用信息，请输入：

```
#nasm -h
```

它同时还会输出可以使用的输出文件格式。

如果使用 Linux 并且不清楚用户的系统是 a.out 还是 ELF，请输入：

```
#file nasm (在nasm二进制文件的安装目录下使用)
```

- 如果系统输出如下信息

nasm: ELF 32-bit LSB executable i386 (386 and up) Version 1，那么系统就是 ELF 格式的，然后就应该在产生 Linux 目标文件时使用选项-f elf。

- 如果系统输出如下信息

nasm: Linux/i386 demand-paged executable (QMAGIC)，或者与此相似的，系统是 a.out 格式的，那就应该使用-f aout。

就像其他的 UNIX 编译器与汇编器，NASM 在碰到错误以前是不输出任何信息的，所以除了出错信息看不到任何其他信息。这里只是对 NASM 的基本命令的简单介绍，有关 NASM 汇编器的详细技术请参考 <http://nasm.sourceforge.net/> 网站技术资料。

2.3.3 GNU gdb 调试器

1. gdb 介绍

在了解 gdb 可以做什么，怎么做之前，让我们先来看看为什么要用 gdb。一般用户使用 gdb（或其他调试工具）是为了发现程序错误，更经常地是在已知程序有错的情况下定位 bug 的位置。既然这样，就需要跟踪程序的执行情况，查看程序执行是否正确，当然这就需要有个让用户与执行程序交互的环境，调试工具提供一个能让程序在用户的掌控下执行，并能够查看一些在程序执行过程中的“内幕信息”的环境。

为了查看程序运行过程中的状态，希望程序能在适当的位置或者在一定的条件下能

够暂停运行。为此，调试工具应该提供了断点、查看变量/表达式、显示程序栈等功能。看了某个点的“内幕”后，还期望更多，所以要能控制程序运行才行，这就要求断点、继续运行、单步（多步）运行、进入函数运行等功能。在某些情况下，还需要通过修改当前的执行环境（变量等）来达到期望的执行顺序。也就是说，光看到信息是不够的，还需要能够修改信息才行。

理解了这些问题后，就明白 `gdb` 的各个功能的用意，自然也就明白该如何使用调试工具了。当然，要让 `gdb` 有效的发挥作用，还需要一定的经验与技巧，而这主要靠用户在实践中才能学到，学习资料只能提供最基本的使用帮助。

2. `gdb` 能做什么

`gdb` 可以用来调试 C、C++、Modula-2 编写的程序。一般来说，`gdb` 能执行的工作大致可以分为四类：

- 1) 启动程序，按指定的方式执行程序。
- 2) 在指定条件下使程序暂停。
- 3) 当程序被停住时，可以检查此时程序中的变化。
- 4) 改变程序中的变量或执行顺序来试验。

3. `gdb` 使用概述

首先要了解的是 `gdb` 的 `help` 命令，因为用户可能记不住各个命令的语法和用途，但只要能正确使用 `help` 命令，就不需要任何其他的 `gdb` 资料。启动 `gdb` 后，输入 `help`，系统将把 `gdb` 命令按种类显示到屏幕上。

`help` 命令只是列出 `gdb` 的命令种类，如果要看某类中的命令，可以使用 `help <class>` 命令，如：`help breakpoints`，查看设置断点的所有命令。也可以直接 `help <command>` 来查看某个命令的具体信息。

为了使 `gdb` 正常工作，必须让程序在编译时包含调试信息。调试信息包含程序里的每个变量的类型和在可执行文件里的地址映射以及源代码的行号。`gdb` 利用这些信息使源代码和机器码相关联，在编译时用 `-g` 选项打开调试选项。

4. 在 `gdb` 中运行程序

当以 `gdb <program>` 方式启动 `gdb` 后，可以使用 `r` 或是 `run` 命令运行程序。在程序运行之前，可能需要设置下面四项内容。

(1) 程序运行参数。

`set args` 指定运行时参数（如：`set args 10 20 30 40 50`）。

`show args` 命令可以查看设置好的运行参数。

(2) 运行环境。

`path <dir>` 设定程序的运行路径。

`show paths` 查看程序的运行路径。

`set environment varname [=value]` 设置环境变量，如：`set env USER=hchen`。

`show environment [varname]` 查看环境变量。

(3) 工作目录。

`cd <dir>` 相当于 shell 的 `cd` 命令。

`pwd` 显示当前的所在目录。

(4) 程序的输入输出。

`info terminal` 显示程序用到的终端模式。

使用重定向控制程序输出, 如: `run > outfile`。

`tty` 命令可以指定输入输出的终端设备, 如: `tty /dev/ttys0`。

5. 调试已运行的程序

- 可以有两种方法调试已运行程序。

(1) 用 `ps` 查看正在运行的程序的进程 ID, 然后用 `gdb <program> PID` 格式挂接正在运行的程序。

(2) 先用 `gdb <program>` 关联上程序, 并运行 `gdb`, 在 `gdb` 中用 `attach` 命令来挂接程序正在运行的进程。`detach` 可用来取消挂接的进程。

- 暂停/恢复程序运行。

可以使用 `info program` 来查看程序的当前的执行状态。

在 `gdb` 中, 可以有以下几种暂停方式: 断点 (BreakPoint)、观察点 (WatchPoint)、捕捉点 (CatchPoint)、信号 (Signals)、线程停止 (Thread Stops)。如果要恢复程序运行, 可以使用 `c` 或是 `continue` 命令。

- 查看变量/表达式的值。

可以使用 `print expr` (或 `p expr`) 来查看程序变量/表达式的值。

- 显示程序栈。

可以使用 `backtrace` (或 `bt`) 来显示程序栈。

- 单步跟踪。

`next [n]` 执行下一条 (或 `n` 条) 语句, 不进入子程序。

`step [n]` 执行下一条 (或 `n` 条) 语句, 进入子程序, 可用 `finish` 从子程序返回。

6. gdb 常用命令

- `backtrace` 显示程序中的当前位置和表示如何到达当前位置的栈跟踪;
- `breakpoint` 在程序中设置一个断点;
- `cd` 改变当前工作目录;
- `clear` 删除刚才停止处的断点;
- `commands` 命中断点时, 列出将要执行的命令;
- `continue` 从断点开始继续执行;
- `delete` 删除一个断点或监测点, 也可与其他命令一起使用;
- `display` 程序停止时显示变量和表达式;
- `down` 下移栈帧, 使得另一个函数成为当前函数;
- `frame` 选择下一条 `continue` 命令的帧;

- **info** 显示与该程序有关的各种信息;
- **jump** 在源程序中的另一点开始运行;
- **kill** 异常终止在 gdb 控制下运行的程序;
- **list** 列出相应于正在执行的程序的原文件内容;
- **next** 执行下一个源程序行, 从而执行其整体中的一个函数;
- **print** 显示变量或表达式的值;
- **pwd** 显示当前工作目录;
- **pype** 显示一个数据结构 (如一个结构或 C++类) 的内容;
- **quit** 退出 gdb;
- **reverse-search** 在源文件中反向搜索正规表达式;
- **run** 执行该程序;
- **search** 在源文件中搜索正规表达式;
- **set variable** 给变量赋值;
- **signal** 将一个信号发送到正在运行的进程;
- **step** 执行下一个源程序行, 必要时进入下一个函数;
- **undisplay display** 命令的反命令, 不要显示表达式;
- **until** 结束当前循环;
- **up** 上移栈帧, 使另一函数成为当前函数;
- **watch** 在程序中设置一个监测点 (即数据断点);
- **whatis** 显示变量或函数类型。

这里只是简述了 gdb 的基本使用, 具体的一些操作就要请读者自己去实践了。

2.4 Bochs PC 模拟器

GeekOS 运行于 Linux 下的 Bochs PC 模拟器, Bochs 是用 C++ 开发的可移植的 IA-32 (x86) PC 模拟器, 它几乎可以运行在所有流行的平台上。包括对 Intel x86 CPU、通用 I/O 设备和可定制的 BIOS 的模拟。目前, Bochs 可以模拟 386, 486, Pentium Pro 或者 AMD64 CPU, 包括可选的 MMX, SSE, SSE2 和 3DNow 指令。Bochs 的模拟环境中可以运行大部分的操作系统, 包括 Linux、Windows 95、DOS、Windows NT 4、FreeBSD 和 MINIX 等。Bochs 由 Kevin Lawton 创建, 此项目当前仍由他维护。

2.4.1 Bochs 安装和使用

Bochs 有 Linux 和 Windows 等不同环境的软件安装包, 用户根据操作系统平台下载相应的版本。如果用户选择在 Cygwin 中开发调试则选择 Windows 版本下载, 在 Windows 系统下安装 Bochs 非常简单, 直接运行安装软件, 按提示操作就可完成。在 Linux 系统中需先解压软件包, 然后再配置编译生成系统文件。推荐使用的 Bochs 版本是 2.0 以上, 版本 2.1.1 可以很好地运行 GeekOS。

为了模拟一台计算机执行一个操作系统软件,Bochs 需要几个文件来代替 PC 硬件的不同部分:

- bochs—模拟器程序本身。
- BIOS-bochs-lastest—模拟 Bochs 硬件的 BIOS。
- VGABIOS-lgpl-lastest—模拟 Bochs 显示系统的 BIOS。
- bochsrc.txt—描述模拟器硬件配置的配置文件。
- disk image (.img)—包含了一个模拟器能引导的操作系统。

使用 Bochs 可以更清楚地了解 PC 的运行原理,Bochs 使用 bochsrc.txt 配置文件来模拟系统环境,所以使用 Bochs 的关键是配置 bochsrc.txt 文件。

仔细阅读 Bochs 安装目录下的 bochsrc-sample.txt 文件,注意看前面不带#的行(带#的行是不执行的行,起注释作用)。将 bochsrc-sample.txt 文件另存为 bochsrc.txt,然后修改以下几项配置,假设 Bochs 的安装目录是 c:/bochs。

```
# 配置模拟器的BIOS和显示系统的BIOS文件
vgaromimage: file= c:/bochs/VGABIOS-lgpl-latest
romimage: file= c:/bochs/BIOS-bochs-latest , address=0xf0000
#配置模拟器的内存大小
megs: 8
#配置模拟器从软盘引导系统
boot: a
#这是对模拟器硬盘的描述,其中disk.img是硬盘映像文件
ata0:enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata0-master:type=disk,mode=flat,path=disk.img,cylinders=40,heads=8,
spt=64
#软盘A的描述,其中的fd.img为软盘映像文件,
floppya: 1_44=fd.img, status=inserted
#floppya: 1_44=fd_aug.img, status=inserted
#这是配置模拟器系统硬件
keyboard_serial_delay: 200
floppy_command_delay: 500
vga_update_interval: 300000
ips: 1000000
mouse: enabled=0
private_colormap: enabled=0
i440fxsupport: enabled=0
#newharddrivesupport: enabled=1

# Uncomment this to write all bochs debugging messages to
# bochs.out. This produces a lot of output, but can be very
# useful for debugging the kernel.
#debug: action=report
#配置模拟器的日志文件
log: ./bochs.out
```

在开发和调试 GeekOS 系统时, 用户需要修改上面斜体的 disk.img 和 fd.img 的路径, 如调试项目 1, 假设 GeekOS 源代码目录为 f:/geekos-0.3.0, 则上面的 path = diskc.img 应修改为 path=f:/geekos-0.3.0/src/project1/build/diskc.img, 1_44=fd.img 应修改为 1_44=f:/geekos-0.3.0/src/project1/build/fd.img, 配置完以后, 就可以运行 bochs 模拟器来调试系统了。

2.4.2 在 Bochs 中运行 GeekOS

1. 开始一个 GeekOS 项目

开始一个 GeekOS 项目开发要用到一个用 perl 写的脚本 startProject, 主要功能是从 GeekOS 解压的文件夹复制相关 project 到工作目录, 命令格式是:

```
./ startProject <project name> <master directory> [<previous project>]
```

假设项目的工作目录是 /home/fred/work, 首先将 f:/geekos-0.3.0/scripts 目录下的 startProject 脚本复制到工作目录, 如要开始项目 0, 需要运行命令 (假设 GeekOS 源文件解压到文件夹 f:/geekos-0.3.0) :

```
$ cd /home/fred/work  
$ ./startProject project0 f:/geekos-0.3.0/src
```

系统就会创建一个有 GeekOS 源代码的目录 (/home/fred/work/project0), 为继续开发做好准备。当然用户也可以不使用脚本命令, 直接创建工作目录, 复制文件就可以。一般来说, 推荐用户使用 CVS<<http://www.cvshome.org/>>或其他版本控制系统来存储项目文件, 从而实现对项目版本的控制。

GeekOS 的每个项目源代码中都包含占位符, 指示需要用户添加代码的地方。这些占位符用 TODO 宏实现, 如 project0 中用:

```
TODO ("Start a kernel thread to echo pressed keys");
```

指示该处需要添加按键处理的代码。占位符一般有注释, 提示怎样添加代码实现新的功能。

2. 编译一个项目

当使用 startProject 脚本创建工作目录, 并已安装了编译 Geekos 所需要的所有软件后, 编译项目就是一件容易的事。假设要编译的是 project0, 运行命令即可完成, 如果系统无错误, 编译会在相应项目的 build 目录下生成 fd.img 和 diskc.img:

```
$ cd /home/fred/work/project0/build  
$ make depend  
$ make
```

3. 在 Bochs 中运行 GeekOS

编辑完 bochsrc.txt 文件后,就可以开始运行 Bochs 了。进入 Bochs 安装目录, 运行 Bochs 命令即可。假设 Bochs 安装目录是 c:/bochs, 运行下面命令:

```
$ cd c:/bochs  
$ bochs
```

运行后, 屏幕上会有一些提示。运行 GeekOS 选择 Begin simulation, 如果 GeekOS 编译成功, 并且 Bochs 的配置也没问题, 将会看到一个模拟 VGA 的文本窗口, Geekos 就能运行程序输出相应信息。

如果 GeekOS 没有正确运行, 查阅在 build 目录中的 bochs.out 文件(运行后产生的)。如果是错误使得 Bochs 不能运行, 或者 GeekOS 系统崩溃, bochs.out 文件中都将会包含相应的诊断信息。

4. 开始一个新项目

GeekOS 的每个项目都含有一个基本 GeekOS 系统内核, 在用户继续新的项目时, 为方便将实现的代码从前面的项目移到新的项目, 系统可以使用脚本命令 startProject, 用法是:

```
$ cd /home/fred/work  
$ ./startProject project1 f:/geekos-0.3.0/src project0
```

使用上述命令的时候, 有时会出现以下提示:

```
Warning:Conflicts detected in merge of file src/geekos/main.c
```

看到这个提示信息时, 我们可以在新项目的目录中看到记录冲突的文件, 里面有这样的代码:

```
<<<<< Your version of src/geekos/main.c  
=====  
static void Mount_Filesystems(void);  
static void Spawn_Init_Process(void);  
>>>>> Master version of src/geekos/main.c from project1
```

这种情况下, 源文件 src/geekos/main.c 中将增加两个函数原型。通常情况下, 简单地删除冲突标记 (“<<<<<” , “=====”, 和 “>>>>>”) 就可以解决冲突。这样就可以获得前面项目的代码和新项目需要的代码。有时也需要在前面项目代码中将后面的项目不再需要的代码标记出来。当所有冲突标记都删除后, 就可以编译项目了。代码编译通过后, 就可以添加项目需要的新特性了。

如果后一个项目要使用前一个项目已经实现的功能代码, 完全可以不使用系统提供的脚本命令, 而直接把代码复制到项目中相应的地方就可以使用。

第3章 make 工具和 makefile 规则

CHAPTER
3

无论是在 Linux 还是在 UNIX 环境中, make 都是一个非常重要的系统开发工具。不管是自己进行项目开发还是安装应用软件, 都经常要用到 make 或 make install。利用 make 工具, 可以将大型的开发项目分解成为多个更易于管理的模块, 对于一个包括几百个源文件的应用程序, 使用 make 和 makefile 工具就可以简洁明快地理顺各个源文件之间纷繁复杂的相互关系。而且如此多的源文件, 如果每次都要输入 gcc 命令编译每一个 C 程序的话, 就会增加程序员很大的工作量, 并容易出错。而 make 能够按照规则自动完成编译工作, 并且可以只对程序员在上次编译后修改过的文件进行编译, 减少重复编译的工作量。因此, 在 Linux 和 UNIX 环境中有效地利用 make 和 makefile 工具可以大大提高项目开发的效率。在掌握 make 命令和 makefile 文件之后, 用户就可以高效地编写 Linux 下的应用软件了。但在许多讲述 Linux 应用的书籍上都没有详细介绍这个功能强大的项目管理工具, 在这里向大家简单介绍 make 工具的使用及其描述文件 makefile 的书写规则。

3.1 makefile 文件

3.1.1 makefile 文件内容

makefile 文件包含五方面的内容, 即具体规则、隐含规则、宏定义、指令和注释。下面分别对它们进行说明。

具体规则: 用于阐述什么时间或怎样重新生成称为规则目标的一个或多个文件。它列举了目标所依靠的文件, 这些文件称为该目标的依赖。具体规则可能同时提供了创建或更新该目标的命令。

隐含规则: 用于阐述什么时间或怎样重新生成同一个文件名的一系列文件。它描述的目标是由和它名字相同的文件进行创建或更新的, 同时提供了创建或更新该目标的命令。

宏定义: 为一个宏赋一个固定的字符串值, 从而在以后的文件中能够用该宏代替这个字符串。注意, 在 makefile 文件中定义宏占单独

一行。宏定义的语法形式如下：

```
(1) 宏名 = 字符串值
(2) 宏名 ?= 字符串值
(3) 宏名 := 字符串值
(4) define 宏名
    字符串值
    endef
```

指令：make 根据 makefile 文件执行一定任务的命令。包括以下几个方面。

- (1) 读其他 makefile 文件（如果包括）。
- (2) 根据变量的值判定是否使用或忽略 makefile 文件的部分内容。
- (3) 定义多行变量，即定义变量值可以包含多行字符的变量。

以“#”开始的行是注释行，注释行在处理时将被 make 命令忽略，如果一个注释行在行尾是“\"，则表示下一行继续为注释行。

3.1.2 makefile 规则

在讲述 Makefile 文件编写之前，还是先来粗略地看一看 makefile 的书写规则：

```
targets ... : dependency ...
    command
    ...
    ...
```

- targets 是一个目标文件或一组目标文件，可以是中间目标文件，也可以是执行文件，还可以是一个标签（Label），对于标签这种特性，在下面会有解释。
- dependency 就是用来说明要生成对应 targets 所需要的文件或是目标，即由哪些源文件来生成 targets。
- command 就是 make 需要执行的命令（任意的 Shell 命令），每一个命令必须以 Tab 键开始，不能用空格符号替代。它是用来说明如何生成 targets。

这是一个文件的依赖关系说明，也就是说，target 这一个或多个的目标文件依赖于 dependency 中的文件，其生成规则定义在 command 中。具体一点就是说，每一次执行 make 命令时，如果 dependency 中有一个以上的文件比 target 文件新的话，command 所定义的命令就会被系统执行，否则表示目标文件原来已经生成，并且是最新的，command 命令不再执行。

简单地说，makefile 的功能就是这么简单。当然，这只是 makefile 的主线和核心，要写好一个 makefile，还需要掌握其他内容。下面将通过实际例子来介绍 makefile 文件的部分主要功能。

3.1.3 makefile 文件示例

如果一个工程有 3 个头文件和 8 个 C 语言源文件，这个工程最终生成的执行文件为

edit。则按照前面所述的规则，根据程序的文件依赖关系 makefile 文件应该编写成下面的这些内容。

```

edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
cc -o edit main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
main.o : main.c defs.h
cc -c main.c
kbd.o : kbd.c defs.h command.h
cc -c kbd.c
command.o : command.c defs.h command.h
cc -c command.c
display.o : display.c defs.h buffer.h
cc -c display.c
insert.o : insert.c defs.h buffer.h
cc -c insert.c
search.o : search.c defs.h buffer.h
cc -c search.c
files.o : files.c defs.h buffer.h command.h
cc -c files.c
utils.o : utils.c defs.h
cc -c utils.c
clean :
rm edit main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o

```

在书写一个规则时，若一行过长，可以把该行用“\”分为两行或多行。一行的尾部是“\”，表示下一行是本行的继续行。因为 make 未对 makefile 文件行长度进行限制，用“\”分割长行仅仅是为了书写和阅读 makefile 文件的需要。具体规则中可以使用的文件通配符有“*”、“?”、“~”。它们的含义同 Bourne shell 通配符一样。

可以把上面的内容保存在文件名为“Makefile”或“makefile”的文件中，然后在该目录下直接输入命令“make”，就可以生成执行文件 edit。如果要删除执行文件和所有的中间目标文件，只要简单地执行“make clean”命令就可以了。

在这个 makefile 中，目标文件（target）包含执行文件 edit 和中间目标文件 (*.o)，依赖文件（dependencys）就是冒号后面的那些 .c 文件和 .h 文件。每一个 .o 文件都有一组依赖文件，而这些 .o 文件又是执行文件 edit 的依赖文件。依赖关系实质上说明了目标文件是由哪些文件编译生成的，也就是说，这些依赖文件的任意一个内容改变，目标文件必须重新生成。

在定义好依赖关系后，后续的那一行定义了如何生成目标文件的操作系统命令，一定要以一个 Tab 键作为开头。make 并不管命令是怎么工作的，只管执行所定义的命令。

make 会比较 targets 文件和 dependency 文件的修改日期，如果 dependency 文件的修改日期比 targets 文件的修改日期更新，或者 target 还不存在时，make 就会执行后续定义的命令。

clean 不是一个文件，它只不过是一个动作名字，类似于 C 语言中的标签（Label）一样。其冒号当前行后面不能有任何字符，这样 make 就不会自动去找文件的依赖性，也就认为 clean 该目标已经是最新的，不会自动执行其后所定义的命令。如果要执行其后的命令，就要在执行 make 命令后显示的指出这个标签（Label）的名字。这样的方法非常有用，可以在一个 makefile 中定义不用的编译命令或者是和编译无关的命令，比如程序的打包，程序的备份。

3.1.4 make 工作原理

在默认的方式下，只要输入 make 命令就可以工作。具体的处理过程如下：

- (1) make 会在当前目录下找文件名为“Makefile”或“makefile”的文件。
- (2) 如果找到，它会找文件中的第一个目标文件（target），在上面的例子中，它会找到 edit 这个文件，并把这个文件作为最终的目标文件。
- (3) 如果 edit 文件不存在，或是 edit 所依赖的后面的.o 文件的修改时间要比 edit 这个文件新，那么，就会执行后面所定义的命令来生成 edit 这个文件。
- (4) 如果 edit 所依赖的.o 文件也不存在，那么 make 会在当前文件中找目标为.o 文件的依赖性，如果找到则再根据那一个规则生成.o 文件（这有点像一个堆栈的过程）。
- (5) 如果指定的 C 文件和 H 文件是存在的，make 会生成.o 文件，然后再用.o 文件生成 make 的最终任务，也就是链接生成执行文件 edit。

这就是整个 make 的工作原理。make 会一层又一层地去找文件的依赖关系，直到最终编译生成第一个目标文件。在找寻的过程中，如果最后被依赖的文件找不到，那么 make 就会直接退出，并报错。而如果是所定义的命令输入错误，结果是编译不成功退出，make 不会报错。make 只管文件的依赖性，即如果根据依赖关系在系统寻找文件之后，冒号后面的依赖文件还是找不到，那么 make 命令就会停止工作，打印出错误信息。

通过上述分析，大家可以知道，像 clean 这个命令，没有被第一个目标文件直接或间接关联，那么它后面所定义的命令将不会被自动执行，不过，也可以显式要求 make 执行。即命令——make clean，以此来清除所有的目标文件，以便整个工程可以重新编译。

这样在编程中，如果某个工程已经被编译过，而之后修改了其中某一个源文件，比如 file.c，那么根据文件依赖性，目标 file.o 会被重新编译（也就是在这个依赖关系后面所定义的命令）。当 file.o 文件被更新，由于 file.o 文件的修改时间比 edit 更新，所以 edit 也会被重新链接（详见 edit 目标文件后定义的命令）。而如果改变了 command.h，那么，kdb.o、command.o 和 files.o 都会被重新编译，并且 edit 也会被重新链接。

3.1.5 makefile 宏

1. 宏定义

在上面的例子中，先看看 edit 的规则：

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
cc -o edit main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
```

可以看到[.o]文件的字符串被重复了两次，如果工程需要加入一个新的 [.o] 文件，那么需要在两个地方加(应该是三个地方，还有一个地方在 clean 中)。当然，上面的 makefile 并不复杂，只要在两个地方加就可以了。但如果是一个复杂 makefile，那么就有可能会忘掉一个需要加入的地方，而导致编译失败。所以，为了使 makefile 文件减少输入，易于维护，在 makefile 文件中可以使用宏。makefile 的宏也就是一个合法的标识符，即由大写字母 A~Z 和小写字母 a~z、数字 (0~9) 和下划线组成。它的定义类似于 C 语言中的宏定义。定义的基本语法为：name = value。其中 name 为定义的宏名，value 为宏替代的字符串，若字符串太长可用 “\” 换行在下一行继续定义。

比如，声明一个宏，叫 objects，它代表所有.o 文件。在 makefile 一开始就应该这样定义：

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

于是，就可以很方便地在 makefile 中以\$(objects)或\${objects}的方式来使用这个宏了。如果宏名只有一个字符，可以省略括号。改进后的 makefile 文件就变成下面的内容：

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
edit : $(objects)
cc -o edit $(objects)
main.o : main.c defs.h
cc -c main.c
kbd.o : kbd.c defs.h command.h
.....
.....
clean :
rm edit $(objects)
```

这时如果有新的.o 文件加入，只需简单地修改一下 objects 宏就可以了。

2. 特殊的宏

为了简化使用规则，make 提供了几个特殊的宏。这些宏使用得比较广泛，特别在隐

含规则的定义中。这些特殊的宏也可以没有值，这取决于当调用宏时 make 处于什么状态。一些宏仅在后缀规则调用中有效，有如下这些宏。

- **\$%:** 当规则的目标文件是一个静态库时，代表静态库的一个成员名。例如，规则的目标是 foo.a(bar.o)，那么，“\$%”的值就为 bar.o，“\$@”的值为 foo.a。如果目标不是静态库文件，其值为空。
- **\$@:** 表示规则的目标文件名。如果目标是一个文档文件（Linux 中，一般称.a 文件为文档文件，也称为静态库文件），那么它代表这个文档的文件名。在多目标模式规则中，它代表的是导致规则命令被执行的目标文件名。
- **\$?:** 该宏的值是代表比目标文件更新的所有依赖名。宏“\$?”能用于目标和后缀规则，然而“\$?”代替了在一个目标规则中许多可能的名字。
- **\$<:** 规则的第一个依赖文件名。如果是一个目标文件使用隐含规则来重建，则它代表由隐含规则加入的第一个依赖文件。
- **\$*:** 在模式规则和静态模式规则中，代表基名。基名是目标模式中“%”所代表的部分（当文件名中存在目录时，基名也包含目录（斜杠之前）部分）。例如，文件 dir/a.foo.b，基名的值为 dir/a.foo。基名对于构造相关文件名非常有用。

【例 1】 编译所有.c 文件生成相应.o 文件的规则如下：

```
% .c : %.o
      gcc -c $(CGFLAGS) $(CPPFLAGS) $< -o $@
```

说明：该规则定义了一条编译所有.c 文件生成对应.o 文件的规则。命令使用“\$@”和“\$<”替换任何情况使用该规则的目标文件和源文件。

3.1.6 make 隐含规则

GNU 的 make 功能很强大，它可以自动推导文件以及文件依赖关系后面的命令，文件就没必要去在每一个.o 文件后都写上类似的命令，因为 make 会自动识别，并自己推导命令。

只要 make 看到一个.o 文件，它就会自动地把.c 文件加在依赖关系中，如果 make 找到一个 whatever.o，那么 whatever.c 就会是 whatever.o 的依赖文件。并且 cc -c whatever.c 也会被推导出来，因此，makefile 就不用写得这么复杂。新的 makefile 又可以改写成如下内容：

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
      cc -o edit $(objects)

main.o : defs.h
```

```

kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
    rm edit $(objects)

```

用户可以注意到，这个 makefile 中并没有写下如何生成所有.o 文件目标的规则和命令。因为 make 的“隐含规则”功能会自动为用户去推导这些目标的依赖目标和生成命令。这种方法也就是 make 的“隐含规则”。上面文件内容中，.PHONY 表示 clean 是个伪目标文件。

既然 make 可以自动推导命令，那么许多.o 和.h 的依赖就有点烦琐，能不能把那么多重复的.h 简单组织起来。makefile 提供自动推导命令和文件的功能就可以满足这个要求。重新编写的 makefile 文件内容如下：

```

objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
    cc -o edit $(objects)

$(objects) : defs.h
    kbd.o command.o files.o : command.h
    display.o insert.o search.o files.o : buffer.h

.PHONY : clean
clean :
    rm edit $(objects)

```

采用这种方法写的 makefile 会变得很简单，但对于初学者来说，文件依赖关系就显得有点凌乱了。初学者最好是不采取这种方法：一是文件的依赖关系看不清楚；二是如果文件很多，加入多个新的.o 文件，那就更理不清楚了。

3.1.7 clean 命令的应用

每个 makefile 中都应该写一个清空中间目标文件 (.o 和生成的其他中间文件) 的规



则，这不仅便于重新编译，也很利于保持文件系统的整洁。一般的方法是：

```
clean:
    rm edit $(objects)
```

前面已经说明了伪目标文件 clean 的用途。然而，如果恰好有一个文件名为 clean 的文件存在，make 就会把它作为有效文件处理。但是，像前面一样，因为 clean 没有相关文件，make 就会认为这个文件是最新的，所以不会执行后面的命令。在这种情况下，就需要使用特殊的 make 目标.PHONY，.PHONY 目标的相关文件的含义与通常一样，但是 make 将不检查是否存在这些文件而直接执行与之相关的命令。即将上面的 make 命令写成如下形式：

```
.PHONY : clean
clean :
    rm edit $(objects)
```

当然，clean 的规则不要放在文件的开头，不然，这就会变成 make 的默认最终目标，没有哪个项目会这样设计。所以 clean 一般都是放在文件的最后。

3.2 GeekOS 的 makefile 文件

前面介绍了 make 工具的功能，下面是结合 makefile 的书写原则，具体分析 GeekOS 操作系统 project2 的项目管理 makefile 文件编写内容。明确每一个项目中 makefile 文件命令的作用是很重要的，从这里用户可以明白 GeekOS 项目是如何组织、编译和生成执行镜像文件的。程序清单中以“#”开头的当前行内容只起注释作用，make 命令执行时不运行。

程序清单 3-1 (\geekos-0.3.0\src\project2\build\makefile)

```
# 编译GeekOS软件需要的工具软件及版本:
# - GNU Make (http://www.gnu.org/software/make)
# - gcc 2.95.2 或更新的编译器版本
# - nasm 汇编工具(http://nasm.sourceforge.net)
# - Perl5, AWK
# Cygwin (http://cygwin.com) 在Windows下编译GeekOS才需要
# - 这个makefile 是可以正常工作的项目管理文件
# 使用-j 选项可以在多台处理器上并行编译系统

PROJECT_ROOT := ..                                ;#指定项目操作的根路径
VPATH := $(PROJECT_ROOT)/src                      ;#指定项目操作的工作目录
# 如果在Windows下编译软件用到的一些变量定义
SYSTEM_NAME := $(shell uname -s)
ifeq ($(findstring CYGWIN,$(SYSTEM_NAME)),CYGWIN)
    SYM_PFX          := _
    EXTRA_C_OPTS     := -DNEED_UNDERSCORE -DGNU_WIN32
```

```
EXTRA_NASM_OPTS      := -DNEED_UNDERSCORE
NON_ELF_SYSTEM       := yes
EXTRA_CC_USER_OPTS   := -Dmain=geekos_main
endif
#
# 系统配置 -
# 以下操作是指定GeekOS如何被编译,哪些源文件被编译,
# 哪些用户程序被编译等等。通常不同项目的编译仅仅需要修改这一部分。
#
# 列出了生成的默认目标文件。
# 目标文件包含了系统引导和运行GeekOS所需的全部代码。
ALL_TARGETS := fd.img diskc.img      # 最终生成的目标文件
# 定义包含实现用户地址空间的文件常量定义,分段系统调用的userseg.c,
# 而分页系统调用的是uservm.c
USER_IMP_C := userseg.c
#包括的核心C语言源文件常量定义
KERNEL_C_SRCS := idt.c int.c trap.c irq.c io.c \
    keyboard.c screen.c timer.c \
    mem.c crc32.c \
    gdt.c tss.c segment.c \
    bget.c malloc.c \
    synch.c kthread.c \
    user.c $(USER_IMP_C) argblock.c syscall.c dma.c floppy.c \
    elf.c blockdev.c ide.c \
    vfs.c pfat.c bitset.c \
    main.c
# 核心源文件.c编译得到相应的.o目标对象文件常量定义
KERNEL_C_OBJS := $(KERNEL_C_SRCS:%.c=geekos/%.o)
#包括的核心汇编语言程序文件常量定义
KERNEL_ASM_SRCS := lowlevel.asm
# 核心源文件.asm编译得到相应的.o目标对象文件常量定义
KERNEL_ASM_OBJS := \
    $(KERNEL_ASM_SRCS:%.asm=geekos/%.o)
# 所有的内核对象文件常量定义
KERNEL_OBJS := $(KERNEL_C_OBJS) \
    $(KERNEL_ASM_OBJS)
# 通用的库文件常量定义,文件存放在common目录下。
# 这个库将被链接到内核程序和用户程序中。
# 它提供字符串操作函数和类printf()格式化输出函数。
COMMON_C_SRCS := fmtout.c string.c memmove.c
# 通用库文件.c编译得到相应的.o目标对象文件的常量定义
COMMON_C_OBJS := $(COMMON_C_SRCS:%.c=common/%.o)
# 用户的库文件常量定义,文件存放在libc目录下。
LIBC_C_SRCS := \
```

```

compat.c process.c\
conio.c
# 用户库文件中的.c文件编译得到相应的.o目标对象文件常量定义
LIBC_C_OBJS := $(LIBC_C_SRCS:%.c=libc/%.o)
# 生成的libc 源文件常量定义
GENERATED_LIBC_SRCS := libc(errno.c
# 用户程序源文件常量定义,存放在user目录下
USER_C_SRCS := \
    null.c long.c \
    shell.c b.c c.c
# 用户程序.c编译生成对应的.exe文件的常量定义
USER_PROGS := $(USER_C_SRCS:%.c=user/%.exe)
# 内核地址常量定义
KERNEL_BASE_ADDR := 0x00010000
# 内核入口点函数定义
KERNEL_ENTRY = $(SYM_PFX)Main
# 用户程序的地址定义
USER_BASE_ADDR := 0x1000
# 用户程序的入口点定义
USER_ENTRY = $(SYM_PFX)_Entry
# -----
# 编译系统的工具 -
# 这一部分定义了编译GeekOS要用到的工具程序
# -----
# 如果使用交叉编译器编译系统下一行就不能注释掉
#TARGET_CC_PREFIX := i386-elf-
#目标C编译器,gcc 2.95.2 或者更新版本。
TARGET_CC := $(TARGET_CC_PREFIX)gcc
# 主机C编译器,这个用来编译在本地主机执行的C语言代码,不是在其他
# 目标平台执行在X86/ELF系统,如Linux 和 FreeBSD,它也能被同样的
# 目标编译器生成。
HOST_CC := gcc
# 目标链接器,GNU ld是目前唯一一个能正常工作的。
TARGET_LD := $(TARGET_CC_PREFIX)ld
# 目标档案管理器
TARGET_AR := $(TARGET_CC_PREFIX)ar
# 目标 ranlib
TARGET_RANLIB := $(TARGET_CC_PREFIX)ranlib
# 目标nm
TARGET_NM := $(TARGET_CC_PREFIX)nm
# 目标objcopy
TARGET_OBJCOPY := $(TARGET_CC_PREFIX)objcopy
# 汇编语言程序编译器
NASM := nasm
# 编译PFAT文件系统映像的工具。
BUILDFAF := tools/builtFat.exe

```

```

# Perl5或更新的版本
PERL := perl
# 填充一个文件使它为某一单位大小的倍数（如扇区大小）
PAD := $(PERL) $(PROJECT_ROOT)/scripts/pad
# 创建一个用0来填充其所有内容的文件
ZEROFILE := $(PERL) $(PROJECT_ROOT)/scripts/zerofile
# 计算一个文件需要占用多少个扇区
NUMSECS := $(PERL) $(PROJECT_ROOT)/scripts/numsecs
# 生成一个包含错误字符串处理的C文件
GENERRS := $(PERL) $(PROJECT_ROOT)/scripts/generrs
# -----
# 定义 -
# 传送给工具软件的选项。
# -----
# 所有C语言源程序使用的标志定义
GENERAL_OPTS := -O -Wall $(EXTRA_C_OPTS)
CC_GENERAL_OPTS := $(GENERAL_OPTS) -Werror
# 核心C语言源程序使用的标志定义
CC_KERNEL_OPTS := -g -DGEEKOS -I$(PROJECT_ROOT)/include
# 核心汇编语言源程序使用的标志定义
NASM_KERNEL_OPTS := -I$(PROJECT_ROOT)/src/geekos/ -f elf \
$(EXTRA_NASM_OPTS)
# 通用库和用户库C语言源程序使用的标志定义
CC_USER_OPTS := -I$(PROJECT_ROOT)/include -I$(PROJECT_ROOT)/include/libc \
$(EXTRA_CC_USER_OPTS)
# 传送给objcopy程序的标志
OBJCOPY_FLAGS := -R .dynamic -R .note -R .comment
# -----
# 规则 -
# 用来描述系统如何编译源程序。
# -----
# 编译核心C语言.c文件得到相应的.o文件
geekos/%.o : geekos/%.c
$(TARGET_CC) -c $(CC_GENERAL_OPTS) $(CC_KERNEL_OPTS) $< -o geekos/$*.o
# 编译核心汇编语言.asm文件得到相应的.o文件
geekos/%.o : geekos/%.asm
$(NASM) $(NASM_KERNEL_OPTS) $< -o geekos/$*.o
geekos/%.S : geekos/%.S
$(TARGET_CC) -c $(CC_GENERAL_OPTS) $(CC_KERNEL_OPTS) $< -o geekos/$*.o
# 编译通用库中C语言.c文件得到相应的.o文件
common/%.o : common/%.c
$(TARGET_CC) -c $(CC_GENERAL_OPTS) $(CC_USER_OPTS) $< -o common/$*.o
# 编译用户库中C语言.c文件得到相应的.o文件
libc/%.o : libc/%.c
$(TARGET_CC) -c $(CC_GENERAL_OPTS) $(CC_USER_OPTS) $< -o libc/$*.o
# 编译用户C语言程序.c文件得到相应的.exe文件

```

```

user/%.exe : user/%.c libc/libc.a libc/entry.o
    $(TARGET_CC) -c $(CC_GENERAL_OPTS) $(CC_USER_OPTS) $< -o user/$*.o
    $(TARGET_LD) -o $@ -Ttext $(USER_BASE_ADDR) -e $(USER_ENTRY) \
        libc/entry.o user/$*.o libc/libc.a
ifeq ($(NON_ELF_SYSTEM),yes)
    $(TARGET_OBJCOPY) -O elf32-i386 $@ $@
endif
# -----
# 目标文件 -
#     编译生成指定文件。
# -----
# 默认的目标文件- 在配置区已把它定义成ALL_TARGETS常量
all : $(ALL_TARGETS)
# 标准的软盘镜像文件- 仅仅用来引导系统内核程序,
# 由geekos/fd_boot.bin geekos/setup.bin geekos/kernel.bin三个文件连接而成
fd.img : geekos/fd_boot.bin geekos/setup.bin geekos/kernel.bin
    cat geekos/fd_boot.bin geekos/setup.bin geekos/kernel.bin > $@
# 扩展的软盘镜像文件- 包含核心程序和存放在PFAT文件系统中的用户执行程序
fd_aug.img : geekos/fd_boot.bin geekos/setup.bin geekos/kernel.bin \
    $(USER_PROGS) $(BUILDFAT)
    $(ZEROFILE) $@ 2880
    $(BUILDFAT) -b geekos/fd_boot.bin $@ geekos/setup.bin \
        geekos/kernel.bin $(USER_PROGS) $(BUILDFAT)
# 生成第一个硬盘设备镜像文件(10 MB)。
# 它包含一个用来存放用户的执行程序的PFAT文件系统。
diskc.img : $(USER_PROGS) $(BUILDFAT)
    $(ZEROFILE) $@ 20480
    $(BUILDFAT) $@ $(USER_PROGS)
# 生成编译PFAT文件系统镜像文件的工具
$(BUILDFAT) : $(PROJECT_ROOT)/src/tools/buildFat.c \
    $(PROJECT_ROOT)/include/geekos/pfat.h $(HOST_CC) \
    $(CC_GENERAL_OPTS) -I$(PROJECT_ROOT)/include \
    $(PROJECT_ROOT)/src/tools/buildFat.c -o $@
# 生成软盘引导扇区fd_boot.bin程序(由BIOS最先装入内存)。
# 得到setup.bin和kernel.bin文件所占用的扇区数
geekos/fd_boot.bin : geekos/setup.bin geekos/kernel.bin \
    $(PROJECT_ROOT)/src/geekos/fd_boot.asm $(NASM) -f bin \
    -I$(PROJECT_ROOT)/src/geekos/ \
    -DNUM_SETUP_SECTORS=`$(NUMSECS)` geekos/setup.bin` \
    -DNUM_KERN_SECTORS=`$(NUMSECS)` geekos/kernel.bin` \
    $(PROJECT_ROOT)/src/geekos/fd_boot.asm \
    -o $@
# 生成Setup.bin程序(由引导程序fd_boot.bin装入内存)。
geekos/setup.bin : geekos/kernel.exe $(PROJECT_ROOT)/src/geekos/setup.asm
    $(NASM) -f bin \
    -I$(PROJECT_ROOT)/src/geekos/ \

```

```

-DENTRY_POINT=0x`egrep 'Main$$' geekos/kernel.sysms |awk '{print $$1}'` \
$(PROJECT_ROOT)/src/geekos/setup.asm \
-o $@ \
$(PAD) $@ 512
# 生成可装入的核心镜像文件kernel.bin。
geekos/kernel.bin : geekos/kernel.exe
$(TARGET_OBJCOPY) $(OBJCOPY_FLAGS) -S -O binary \
geekos/kernel.exe geekos/kernel.bin $(PAD) $@ 512
# 生成核心可执行程序和符号映射。
geekos/kernel.exe : $(KERNEL_OBJS) $(COMMON_C_OBJS)
$(TARGET_LD) -o geekos/kernel.exe -Ttext $(KERNEL_BASE_ADDR) \
-e $(KERNEL_ENTRY) $(KERNEL_OBJS) $(COMMON_C_OBJS) \
$(TARGET_NM) geekos/kernel.exe > geekos/kernel.sysms
# 用户级程序使用的C库文件
libc/libc.a : $(LIBC_C_OBJS) libc/errno.o $(COMMON_C_OBJS)
$(TARGET_AR) ruv $@ $(LIBC_C_OBJS) libc/errno.o $(COMMON_C_OBJS)
$(TARGET_RANLIB) $@
# 生成包含每一个错误代码的字符串表的源文件。
# This is derived automatically from the comments in <geekos/errno.h>.
libc/errno.c : $(PROJECT_ROOT)/include/geekos/errno.h \
$(PROJECT_ROOT)/scripts/generrs \
$(GENERRS) $(PROJECT_ROOT)/include/geekos/errno.h > $@
# 清除在编译过程中新生成的文件
clean :
for d in geekos common libc user tools; do \
(cd $$d && rm -f *); \
done
# 编译生成头文件的依赖关系，当依赖的头文件被修改时，
# 相对应的源文件应该重新被编译。
depend : $(GENERATED_LIBC_SRCS)
$(TARGET_CC) -M $(CC_GENERAL_OPTS) $(CC_KERNEL_OPTS) \
$(KERNEL_C_SRCS:%.c=$(PROJECT_ROOT)/src/geekos/%.c) \
| $(PERL) -n -e 's,^(\$),geekos/$$1,;print' \
> depend.mak
$(TARGET_CC) -M $(CC_GENERAL_OPTS) $(CC_USER_OPTS) \
$(LIBC_C_SRCS:%.c=$(PROJECT_ROOT)/src/libc/%.c) libc/errno.c \
| $(PERL) -n -e 's,^(\$),libc/$$1,;print' \
>> depend.mak
$(TARGET_CC) -M $(CC_GENERAL_OPTS) $(CC_USER_OPTS) \
$(COMMON_C_SRCS:%.c=$(PROJECT_ROOT)/src/common/%.c) \
| $(PERL) -n -e 's,^(\$),common/$$1,;print' \
>> depend.mak
# 默认情况下，没有头文件依赖关系。
depend.mak :
touch $@
include depend.mak

```

第 4 章 PC 启动原理及 GeekOS 启动程序

CHAPTER
4

要真正掌握 GeekOS 系统设计原理，用户第一步就是要弄清楚 GeekOS 操作系统的启动过程。而要掌握 GeekOS 启动原理，用户首先必须了解 80x86 PC 的启动原理。这一章主要就是介绍个人计算机系统启动原理过程。当然，这里介绍的不是一台计算机安装多个操作系统的引导原理，而是从编写操作系统的角度出发，简单介绍计算机怎样从加电开始，从无到有，如何将操作系统核心程序运行起来。

4.1 PC 启动原理

4.1.1 计算机系统启动

计算机系统启动过程如图 4-1 所示。当给计算机加电时，电源产生 POWER GOOD 低电位信号，该信号通过 8284A 时钟产生驱动器输出有效的 RESET 信号，使 CPU 进入复位状态。由于 Intel 80x86 处理器的特性，CS(Code Segment)寄存器中全部都置 1。而 IP(Instruction Pointer)寄存器中全部都置 0，也就是说， $CS=0xFFFF$ ，而 $IP=0x0000$ 。此时，CPU 就依据 CS 及 IP 的值，到 $0xFFFF0$ 去执行一条 JMP 指令。系统开始执行 ROM-BIOS 区的第一条 JMP START 指令。随后 BIOS 启动一个程序，进行主机自检。主机自检的主要工作是确保系统的每一个部分都得到了电源支持，内存储器、主板上的其他芯片、键盘、鼠标、磁盘控制器及一些 I/O 端口正常可用，此后，自检程序将控制权还给 BIOS。接下来 BIOS 读取 BIOS 中的相关设置，得到引导驱动器的顺序，然后依次检查，直到找到可以用来引导的驱动器（或者说可以用来引导的磁盘，包括软盘、硬盘、光盘等），然后读入这个驱动器上磁盘的引导扇区程序到内存。而 BIOS 又是怎么确定哪一个磁盘可以用来引导的呢？

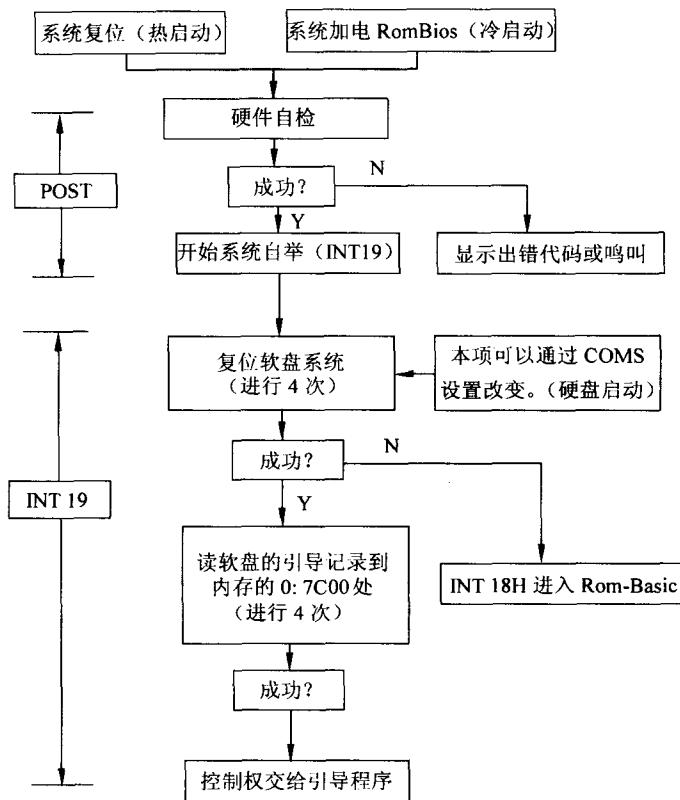


图 4-1 计算机系统启动流程

4.1.2 引导程序

计算机系统的 BIOS 将磁盘的第一个扇区（磁盘最开始的 512 字节）载入内存，放在 0x0000:0x7c00 处，如果这个扇区的最后两个字节是 55 AA，那么这就是一个引导扇区，这个磁盘也就是一张可引导盘。通常这个大小为 512B 的程序就称为引导程序（boot）。如果最后两个字节不是 55 AA，那么 BIOS 就检查下一个磁盘驱动器。

通过上面的概述可以总结出引导程序具有如下三个特点：

- (1) 它的大小是 512B，不能多一个字节也不能少一个字节，因为 BIOS 只读 512B 到内存中去；
- (2) 它的结尾两字节必须是 55 AA，这是引导扇区的标志；
- (3) 它总是放在磁盘的第一个扇区上（0 磁头，0 磁道，1 扇区），因为 BIOS 只读这一个扇区。

因此，用户在编写引导程序的时候，必须注意上面的三个原则。只有符合上面三个原则的程序才可以作为引导程序，至少目前 BIOS 是这样规定的。即使它可能是用户随意写的一段并没有什么实际意义的引导程序测试代码也必须这样。

因为 BIOS 只读一个磁盘扇区——512 个字节的数据到内存中，而现在操作系统核

心程序都比较庞大，512个字节的空间显然是不够存放。因此必须通过执行引导扇区里的程序，将存储在磁盘上的操作系统的内核程序读进内存，然后再跳转到操作系统的内核程序，去执行操作系统功能。

4.1.3 内核程序导入

从上面的描述可以知道，引导程序需要将存在于磁盘上的操作系统的内核程序读入内存，下面将介绍不通过操作系统（因为现在操作系统内核还没有运行）如何去读取磁盘上的内容。一般说来这两种方法可以实现：一种是直接读写磁盘的I/O端口；一种是通过BIOS中断实现。前一种方法是最低层的方法（后一种方法也是在它的基础上实现的），具有极高的灵活性，可以将磁盘上的内容读到内存中的任意地方，但编程复杂。第二种方法是在前一种方法更高层次的实现，牺牲了一点灵活性，比如，它不能把磁盘上的内容读到0x0000:0x0000~0x0000:0x03FF处。为什么不能读到此处呢？这里将首先描述CPU在加电后的中断处理机制。

1. BIOS 的中断处理

作为计算机专业的学生都应该知道中断机制。如果对中断一点都不了解，那么建议用户先阅读教材《计算机组成原理》（高等教育出版社，唐朔飞），书中对中断有非常详尽的描述。当然其他《计算机组成原理》的教材和一般的汇编教材也有介绍，因此这里只介绍BIOS对中断的处理。

如图4-2所示可以清楚的看到，当中断信号产生时，中断信号通过“中断地址形成部件”产生一个中断向量地址，此向量地址其实是指向一个实际内存地址的指针，而这个实际内存地址中往往安排一条跳转指令(jmp)跳转到实际处理此中断的中断服务程序中去执行。这一块专门用于处理中断跳转指令的内存被称为中断向量表。在内存中，这块中断向量表被存放在什么地方呢？而实际的中断处理程序又存放在什么地方呢？

2. 系统的内存安排(1MB)

要回答上面的两个问题，用户需要看看系统中内存是怎么安排的。在CPU被加电的时候，最初的1MB的内存，是由BIOS为用户安排好的，每一字节都有特殊的用处。

如图4-3所示，现在可以很容易地回答上面提出的两个问题。因为0x00000~0x003FF是用来存储中断向量表的空间，所以不能将磁盘中的操作系统程序读到此处，否则会覆盖中断向量表，从而导致无法再通过BIOS中断来读取磁盘内容。一些用户也许会说：是先调用中断再读取磁盘内容的，不会有影响。但事实上BIOS在读的过程中会需要多次调用其他中断处理程序来辅助完成。

3. 利用BIOS 13号中断读取磁盘扇区

有了前面的描述作为基础，下面可以正式描述怎样通过BIOS中断读取磁盘扇区了。要读取磁盘扇区，需要使用BIOS的13号中断，13号中断会将几个寄存器的值作为其参数，因此，在调用13号中断的过程中需要首先设置寄存器。要设置的寄存器及内容如下。

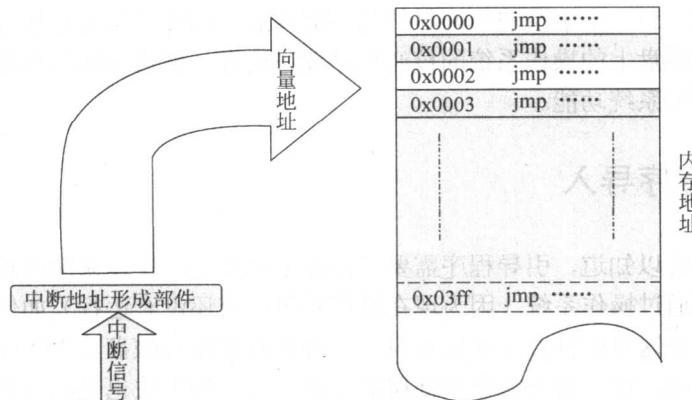


图 4-2 中断处理过程图

0x00000~0x003FF:	中断向量表
0x00400~0x004FF:	BIOS 数据区
0x00500~0x07BFF:	自由内存区
0x07C00~0x07DFF:	引导程序加载区
0x07E00~0x9FFFF:	自由内存区
0xA0000~0xBFFFF:	显示内存区
0xC0000~0xFFFFF:	BIOS 中断处理程序区

图 4-3 实模式下的内存使用区间布局

- AH 寄存器：存放功能号，为 2 的时候，表示使用读磁盘功能。
- DL 寄存器：存驱动器号，表示欲读哪一个驱动器。
- CH 寄存器：存磁头号，表示欲读哪一个磁头。
- CL 寄存器：存扇区号，表示欲读的起始扇区。
- AL 寄存器：存计数值，表示欲读入的扇区数量。

在设置了这几个寄存器后，就可以使用 int 13 这条指令调用 BIOS 13 号中断读取指定的磁盘扇区读到 ES:BX 处。因此，在调用它之前，实际上还需要设置 ES 与 BX 寄存器，以指定数据在内存中存放的位置。

4.2 保护模式

4.2.1 保护模式

自从 1969 年推出第一个微处理器以来，Intel 处理器就在不断地更新换代，从 8086、

8088、80286，到 80386、80486、奔腾、奔腾 II、奔腾 4 等，其体系结构也在不断变化。从 80386 开始，增加了一些新的功能，弥补了 8086 的缺陷。这其中包括内存保护、多任务及使用 640KB 以上的内存等，但仍然保持和 8086 家族的兼容性。也就是说 80386 仍然具备了 8086 和 80286 的所有功能，但是在功能上有了很大的增强。早期的处理器是工作在实模式之下的，80286 以后引入了保护模式，而在 80386 以后保护模式又进行了很大的改进。在 80386 中，保护模式为程序员提供了更好的保护，提供了更多的内存。事实上，保护模式的目的不是为了保护程序，而是要保护程序以外的所有程序（包括操作系统）。

简单地讲，保护模式是处理器的一种最自然的模式。在这种模式下，处理器的所有指令及体系结构的所有特色都是可用的，并且能够达到最高的性能。

4.2.2 实模式和保护模式

从表面上看，保护模式和实模式并没有太大的区别，二者都使用了内存段、中断和设备驱动来处理硬件，但二者有很多不同之处。在实模式中内存被划分成段，每个段的大小为 64KB，而这样的段地址可以用 16 位来表示。内存段是通过和段寄存器相关联的内部机制来处理的，这些段寄存器（CS、DS、SS 和 ES）的内容形成了物理地址的一部分。具体来说，最终的物理地址是由 16 位的段地址和 16 位的段内偏移地址组成的。用公式表示为：

$$\text{物理地址} = \text{左移 4 位的段地址} + \text{偏移地址}$$

在保护模式下，段是通过一系列被称之为“描述符表”的表项所定义的。段寄存器存储的是指向这些表的指针。用于定义内存段的表有两种：全局描述符表（GDT）和局部描述符表（LDT）。GDT 是一个段描述符数组，其中包含所有应用程序都可以使用的基本描述符。在实模式中，段长是固定的（64KB），而在保护模式中，段长是可变的，其最大可达 4GB。LDT 也是段描述符的一个数组。与 GDT 不同，其中存放的是局部的、不需要全局共享的段描述符。每一个操作系统都必须定义一个 GDT，而每一个正在运行的任务都会有一个自己的 LDT。每一个描述符的长度是 8 个字节，格式如图 4-4 所示。当段寄存器被加载的时候，段基地址就会从相应的表项获得。描述符的内容会被存储在一个程序员不可见的影像寄存器（shadow register）之中，以便下一次同一个段可以使用该信息而不用每次都到表中提取。物理地址由 16 位或者 32 位的偏移加上影像寄存器中的基址组成。实模式和保护模式的不同可以从图 4-5 和图 4-6 中很清楚地看出来。

此外，还有一个中断描述符表（IDT）。这些中断描述符会告诉处理器到哪里可以找到中断处理程序。和实模式一样，每一个中断都有一个入口，但是这些入口的格式却完全不同。

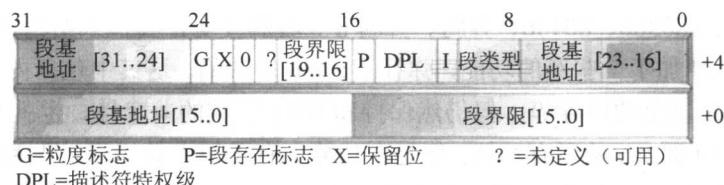


图 4-4 段描述符的格式

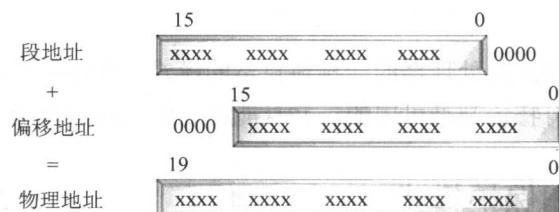


图 4-5 实模式的寻址

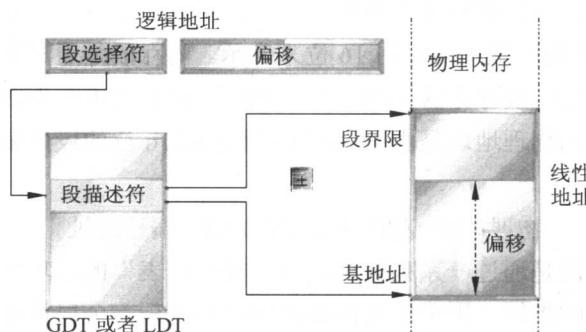


图 4-6 保护模式下的寻址

现在可以正式开始描述在保护模式下段模式是怎样访问内存的。这里之所以要强调“段模式”是因为在保护模式下还有另外一种内存访问模式——页模式，它负责将线性地址按照某种映射转换为物理地址。“页模式”也是基于段模式的，在不使用它的情况下，线性地址会被直接放到地址线上作为物理地址使用。“段模式”是不可避免的，所谓“纯页模式”只是将整个线性地址当作一个整段，没有什么方法可以真正绕过“段模式”，因为这是由 CPU 内存访问机制所规定的。下面只描述段模式，页模式将在第 6 章介绍。

从上面已经知道程序使用的逻辑地址到线性地址的映射是通过“描述符”来完成的，而“描述符”又是放在描述符表中的，那么，一个描述符表中有许多描述符，到底选用哪一个描述符呢？这是由一个索引来决定，这个索引将指出是表中的第几个描述符，这个索引有一个专门的术语来描述，通常称它为“段选择子”。“段选择子”由 2 个字节共 16 位组成，如图 4-7 所示。下面，就来看看它提供了哪些信息。



图 4-7 段选择子格式

其中，

RPL：标识特权级，00, 01, 10, 11，对应0、1、2、3共4个系统特权级。

TI：为0时表明这是一个用于全局描述符表的选择子，为1时表明用于局部描述符表。

索引值：用来指示表中第几个描述符。索引值共有13位，因此，每张描述符表共可有8KB个表项，而一个表项如前所述，占8个字节，因此一张描述符表最大可达64KB。

不知道大家是否注意到这样一个事实，如果将“段选择子”的最后3位置0，这整个段选择子其实就是一个描述符在描述符表中的偏移量（每个描述符的单位长度都是8个字节）。从这里可以发现Intel的工程师设计得非常精巧，如此的安排，结果是极大加快了选取一个描述符的速度。因为将一个段选择子最后3位清零后与描述符表的基址相加，就立即可以得到一个描述符的物理地址，通过这个地址就可以直接得到一个描述符。那么这个描述符表的基址又是放在哪儿的呢？

所谓描述符表的基址也就是此描述符表在内存中的起始地址，即表中第一个描述符所在的内存地址，系统中用两个特殊的寄存器来存放，一个用于存放全局描述符表的基址，称之为“全局描述符表寄存器（GDTR）”，另一个用来存放局部描述符表的基址，称之为“局部描述符表寄存器（LDTR）”，它们的结构如图4-8所示。

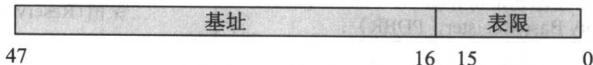


图 4-8 全局描述符表寄存器内容

其中表限即表的大小限制，它的使用与前面所描述的段限是类似的，因此，这里就不再描述了。在保护模式下，以前实模式下的段寄存器还是有用的，不过它不再用来存放段的基址，而是用来存放“段选择子”，它的名字也变成了“段选择子寄存器”，在访问内存的时候，需要给出的是“段选择子”，而不是段基址了。比如，用户现在想使用全局描述符表中的第二个表项，即其中的第二个“段描述符”，这个“段选择子”就需按如下的方式构成。

RPL：00，因为现在是在写操作系统，工作在0特权级。

TI：0，使用全局描述符。

索引值：1，使用第二个全局描述符，第一个全局描述符编号为0，第二个为1。

因此，系统的“段选择子”为：0000 0000 0000 1000，也即0x0008。对于0x0008:0x0000这样一个逻辑地址，在保护模式下就应看成是使用全局描述符段中第二个描述符所描述的段，并且偏移量为0的内存地址。这个逻辑地址的线性地址是怎样形成的呢？

如图4-9所示，用户可以清楚地看出一个逻辑地址是怎样转换为一个32位的线性地址的。

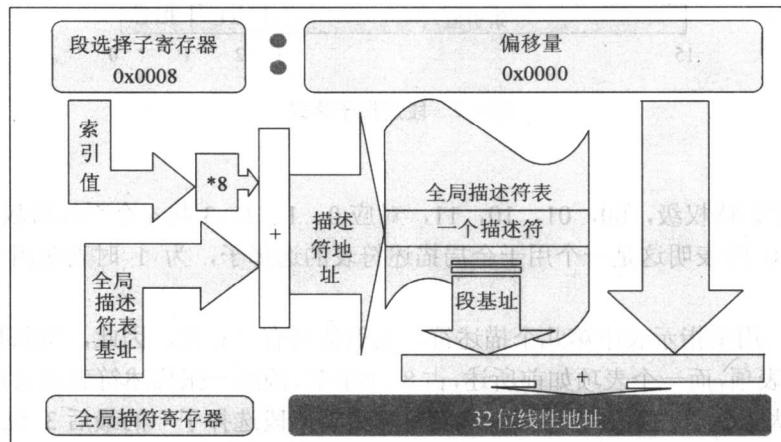


图 4-9 保护模式下的地址转换原理图

4.2.3 进入保护模式

80386 有 4 个 32 位控制寄存器，名字分别为 CR0、CR1、CR2 和 CR3。这些寄存器仅能够由系统程序通过 MOV 指令访问。格式如图 4-10 所示。

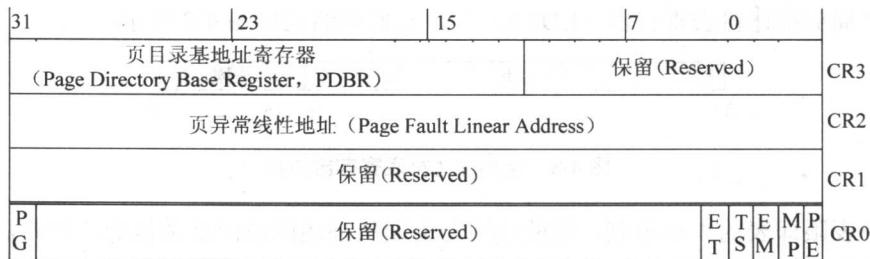


图 4-10 控制寄存器结构

控制寄存器 CR0 包含系统整体的控制标志，它控制或指示出整个系统的运行状态或条件。其中，

- PE——保护模式开启位（Protection Enable，比特位 0）。如果设置了该比特位，就会使处理器开始在保护模式下运行。
- MP——协处理器存在标志（Math Present，比特位 1）。用于控制 WAIT 指令的功能，以配合协处理器的运行。
- EM——仿真控制（Emulation，比特位 2）。指示是否需要仿真协处理器的功能。
- TS——任务切换（Task Switch，比特位 3）。每当任务切换时处理器就会设置该比特位，并且在解释协处理器指令之前测试该位。
- ET——扩展类型（Extention Type，比特位 4）。该位指出了系统中所含有的协处理器类型（是 80287 还是 80387）。

- PG——分页操作（Paging，比特位 31）。该位指示出是否使用页表将线性地址变成物理地址。参见第 6 章对分页内存管理的描述。

CR1 是保留在未来处理器中使用的，在 80386 中没有定义。CR2 用于 PG 设置为分页系统时处理页异常操作，CPU 会将引起错误的线性地址保存在该寄存器中。CR3 同样也是在 PG 设置为分页系统时起作用。该寄存器为 CPU 指定当前运行的任务所使用的页目录表。

在此，值得关注的是 CR0 寄存器的 PE 位控制，它负责实模式和保护模式之间的切换。当 PE=1 时，说明处理器运行于保护模式之下，其采用的段机制和前面所述的相应内容对应。如果 PE=0，那么处理器就工作在实模式之下。

系统要切换到保护模式，实际就是把 PE 位设置为 1。当然为了把系统切换到保护模式，还要做一些其他的事情。启动程序必须要对系统的段寄存器和控制寄存器进行初始化。把 PE 位设置为 1 以后，还要执行跳转指令。过程简述如下。

- (1) 创建 GDT 表。
- (2) 用 lgdt 命令加载 gdtr。
- (3) 启用 A20 地址线。
- (4) 通过置 CR0 的 PE 位为 1。
- (5) 执行跳转，进入保护模式。

4.3 GeekOS 启动程序分析

有了上面的 Intel PC 机启动原理的介绍，用户就可以对 GeekOS 的启动过程进行具体分析了。在 GeekOS 操作系统中，系统启动所涉及的程序主要有三个：fd_boot.bin（由 fd_boot.asm 编译得到）、setup.bin（由 setup.asm 编译得到）和 kernel.bin（系统编译得到的内核程序）。首先由 BIOS 从规定的系统启动盘读入 bootstrap.bin 到内存地址 07C00 处，然后跳转到此处开始执行。bootstrap.bin 程序：首先将自身复制到内存的 09000 处，以便让操作系统内核程序装入到内存低端部分；然后将磁盘中的 setup.bin 程序读入到内存 09020 处，将操作系统内核 kernel.bin 装入到内存 01000 处，之后跳转到 setup 程序执行。Setup.bin 程序（setup.asm）：首先检测内存大小，关闭中断，之后创建临时的 GDT 并进行实模式到保护模式的转换，最后跳入内核程序入口_Main。Main 函数（src/geekos/main.c）：这个函数是 GeekOS 的内核初始化函数，它负责将系统各部分设备和数据结构初始化，完成系统启动工作。之后系统就可以开始执行核心级线程或用户级线程。

系统还包含另外一个主要的程序 lowlevel.asm，它主要是提供底层中断和线程切换处理代码。由于它是用汇编语言编写，所以把它放在这里和其他两个汇编语言程序一起介绍。

4.3.1 fd_boot.asm 代码分析

1. 功能描述

fd_boot.bin 代码是磁盘引导块程序，驻留在磁盘的第一个扇区中（引导扇区，0 磁

道（柱面），0 磁头，第 1 个扇区）。在 PC 加电 ROM BIOS 自检后，引导扇区由 BIOS 执行 int19H 中断，加载到内存 0x7C00 处，然后引导扇区程序将自己移动到内存 0x90000 处。该程序的主要作用是首先将 setup 模块（由 setup.asm 编译得到）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，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（是否为 1.44MB 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. 代码分析

```
; GeekOS操作系统的引导程序 (src/geekos/fd_boot.asm)
%include "defs.asm"           ;文件defs.asm中定义了程序用到的常量定义。
;定义一个宏Pad_From_Symbol使用数据0来填充一段区间。
%macro Pad_From_Symbol 2
    times (%1 - ($ - %2)) db 0
%endmacro
;-----
;实际代码部分
;-----

[BITS 16]                      ;十六位编程模式
[ORG 0x0]                       ;偏移地址为0
BeginText:                      ;引导程序的起始地址
;复制引导扇区程序到0x90000处
    mov ax, BOOTSEG            ;BOOTSEG值为0x07c0
    mov ds, ax                  ;将ds段寄存器置为0x07C0
    xor si, si                 ;源地址 ds:si = 0x07C0:0x0000
    mov ax, INITSEG            ;INITSEG值为0x9000
    mov es, ax                  ;将es段寄存器置为0x9000
    xor di, di                 ;目的地址 es:di = 0x9000:0x0000
    cld                         ;清除方向标志位
    mov cx, 256                ;移动256个字
    rep movsw                  ;重复复制,指导cx为0
    jmp INITSEG:after_move    ;间接跳转。这里INITSEG 指出跳转到的段地址
after_move:
;现在开始在INITSEG段执行代码
;程序需要的数据段定义在INITSEG段
;它们和代码在同一个段中
    mov ds, ax                  ;ax仍然包含INITSEG的值
;设置堆栈指针为原来装入代码的位置BOOTSEG。
    mov ax, 0
```

```

mov ss, ax ;堆栈段地址为0
mov sp, (BOOTSEG << 4) + 512 - 2 ;设置堆栈指针
load_setup: ;装入setup.bin代码
    mov ax, word [setupStart] ;setupStart为setup.bin占用的第一个扇区号
    mov word [sec_count], ax ;sec_count变量存放第一个扇区号
    add ax, [setupSize] ;ax得到最大扇区号
    mov word [max_sector], ax ;max_sector存放最大扇区号
.again:
    mov ax, [sec_count]
    push ax ;为ReadSector调用设置第一个参数(扇区号)
    push word SETUPSEG ;为ReadSector调用设置第二个参数(起始地址)
    sub ax, [setupStart] ;ax为总的扇区数
    shl ax, 9 ;ax为总的读取的字节数(扇区数×512)
    push ax ;为ReadSector调用设置第三个参数(总字节数)
    call ReadSector ;调用ReadSector从软盘中读入一个扇区
    add sp, 6 ;在堆栈中清除前面设置的三个参数
    ;为读下一个扇区准备
    inc word [sec_count] ;读取的扇区号加1
    ;判断读取是否成功
    mov bx, word [max_sector] ;是否为要读取的最大扇区号
    cmp word [sec_count], bx ;没有读完继续读下一个扇区
    jl .again

load_kernel:
    ;从磁盘上装入核心影像文件的所有扇区到内存的KERNSEG处
    ;每个64KB的段总共有128个段,装入扇区时需计算它装入哪一个段
    ;算出kernel.bin的起始扇区号和最大扇区号
    mov ax, word [kernelStart] ;起始扇区号
    mov word [sec_count], ax ;sec_count存放第一个扇区号
    add ax, word [kernelSize] ;ax得到最大扇区号
    mov word [max_sector], ax ;max_sector存放最大扇区号
.again:
    mov ax, [sec_count] ;
    push ax ;为ReadSector调用设置第一个参数(扇区号)
    sub ax, [kernelStart] ;ax为总的扇区数
    mov cx, ax ;总的扇区数保存在cx
    shr ax, 7 ;计算扇区属于哪一个段
    shl ax, 12 ;计算段的基地址
    add ax, KERNSEG ;计算KERNSEG的相对地址
    push ax ;为ReadSector调用设置第二个参数(起始地址)
    and cx, 0x7f ;得到扇区号
    shl cx, 9 ;读取的字节总数
    push cx ;为ReadSector调用设置第三个参数(总字节数)
    call ReadSector ;调用ReadSector从软盘中读入一个扇区
    add sp, 6 ;清除前面设置的三个参数

```

```

; 为读下一个扇区准备
inc word [sec_count]
; 判断是否装入所有扇区, 没有就继续读
mov bx, word [max_sector]
cmp word [sec_count], bx
jle .again
; 一旦装入完setup.bin和kernel.bin文件到内存,
; 系统马上跳转到setup.bin程序起始处开始执行。
jmp SETUPSEG:0

; ReadSector每次只读一个扇区, 如果要一次读多个扇区,
; 这段代码必须要重写。

; 参数:
;   - 扇区号      [bp+8]
;   - 目标段地址    [bp+6]
;   - 目标偏移地址  [bp+4]

ReadSector:
    push    bp          ; 设置堆栈区间
    mov     bp, sp
    pusha
    ; 保存所有寄存器内容到堆栈

    %if 0
        ; 出错参数设置, 以便打印信息
        mov dx, [bp+8]
        call PrintHex
        call PrintNL
        mov dx, [bp+6]
        call PrintHex
        call PrintNL
        mov dx, [bp+4]
        call PrintHex
        call PrintNL
    %endif

    ; 计算扇区号 = log_sec % SECTORS_PER_TRACK
    ; 计算磁头号 = (log_sec / SECTORS_PER_TRACK) % HEADS
    mov ax, [bp+8]          ; 从堆栈得到逻辑扇区号
    xor dx, dx              ; 使dx的值为0
    mov bx, SECTORS_PER_TRACK ; 除数
    div bx                  ; 执行除法
    mov [sec], dx            ; 扇区号为余数
    and ax, 1
    mov [head], ax           ; 保存磁头号
    ; 计算磁道号 = log_sec / (SECTORS_PER_TRACK*HEADS)
    mov ax, [bp+8]          ; 从堆栈得到逻辑扇区号
    xor dx, dx              ; 使dx的值为0
    mov bx, SECTORS_PER_TRACK*2 ; 除数

```

```
div bx          ; 执行除法
mov [track], ax ; 商为磁道号
%if 0
; 出错参数设置,以便打印信息
    mov dx, [sec]
    call PrintHex
    call PrintNL
    mov dx, [head]
    call PrintHex
    call PrintNL
    mov dx, [track]
    call PrintHex
    call PrintNL
%endif
; 现在,真正开始从磁盘读一个扇区,
; 如果不成功,可以重复读3次。
    mov [num_retries], byte 0
.again:
    mov ax, [bp+6]          ;
    mov es, ax              ; 缓冲区段地址
    mov ax, (0x02 << 8) | 1 ; 中断调用功能号ah为02h
                           ; 读取的扇区数al为1
    mov bx, [track]         ; 取磁道号
    mov ch, bl              ; 赋值给ch
    mov bx, [sec]            ; 取扇区号
    mov cl, bl              ; 赋值给cl
    inc cl                  ; 原来计算以0开始,但扇区编号是从1开始
    mov bx, [head]           ; 取磁头号
    mov dh, bl              ; 赋值给dh
    xor dl, dl              ; 当前磁盘驱动器为0
    mov bx, [bp+4]           ; 赋值偏移地址给bx
                           ; (es:bx 缓冲区地址)
; 调用BIOS中断读取磁盘扇区服务
int 0x13
; 如果执行的标志位没有设置,又没有错误将结束读取
jnc .done
; 出错代码存放在ah中
    mov dx, ax
    call PrintHex           ; 打印调试信息
    inc byte [num_retries]
    cmp byte [num_retries], 3 ; 如果出错,继续读,直到三次为止
    jne .again
; 如果程序执行到这里,系统失败,进入死循环
    mov dx, 0xdead
```

```

        call      PrintHex
.here:   jmp .here
.done:    ; 读取成功
        popa      ; 恢复所有寄存器内容到寄存器
        pop bp    ; 退出堆栈区间
        ret

; 包含文件util.asm,它里面包含PrintHex和PrintNL函数定义
%include "util.asm"

-----  

; 变量定义  

-----  

; 这些变量在ReadSector调用中使用
head: dw 0
track: dw 0
sec: dw 0
num_retries: db 0
; 这些变量被使用在从磁盘读扇区操作中
sec_count: dw 0
max_sector: dw 0
; 使得PFAT Boot Record数据位正好存放在BIOS标定的地方
Pad_From_Symbol PFAT_BOOT_RECORD_OFFSET, BeginText
; PFAT boot record 占用了28个字节
; 主要描述装入setup程序和kernel程序用到的变量的初值
        dw 10 dup(?)
;; pfat boot record的一部分
setupStart:
        dw 1           ; setup.bin默认所占用的起始扇区号
;;pfat boot record的一部分
setupSize:
        dw NUM_SETUP_SECTORS ; setup.bin文件所占用的扇区数
;;pfat boot record的一部分
kernelStart:
        dw 1+NUM_SETUP_SECTORS ; kernel.bin默认所占用的起始扇区号
;;pfat boot record的一部分
kernelSize:
        dw NUM_KERN_SECTORS ; kernel.bin文件所占用的扇区数
; 使得扇区成为 BIOS 认为合法的引导扇区
Pad_From_Symbol BIOS_SIGNATURE_OFFSET, BeginText
Signature dw 0xAA55 ; BIOS规定系统盘这两个字节的内容必须为0xAA55

```

4.3.2 setup.asm 代码分析

1. 功能描述

setup.asm 程序的作用主要是利用 ROM BIOS 中断读取机器系统数据，并将这些数

据保存到相应变量中。这些参数将被内核中相关程序使用。然后 setup.asm 程序为系统进入保护模式做准备：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（也称机器状态字），从而进入 32 位保护模式运行；跳转到位于内核程序入口 Main 函数继续运行。为了能让内核程序在 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。

2. 代码分析

```
; GeekOS操作系统setup代码(/src/geekos/setup.asm)
%include "defs.asm"           ;文件defs.asm定义了要用到的常量定义
[BITS 16]                     ;十六位编程模式
[ORG 0x0]                      ;偏移地址为0
start_setup:
;重新定义数据段指向SETUPSEG,以便程序可以访问
;在setup.asm文件中定义的数据
mov ax, SETUPSEG
mov ds, ax
;下面是取扩展内存的大小值(KB)
;是调用中断0x15,功能号ah = 0x88
;返回: ax = 从0x100000(1M)处开始的扩展内存大小(KB)
;用这种方法只能检测到64MB内存,但对于GeekOS已经足够。
mov ah, 0x88
int 0x15
add ax, 1024      ; 1024 KB == 1 MB
mov [mem_size_kbytes], ax
;关闭软驱马达,否则软驱的灯会一直亮着
call Kill_Motor
;从这里开始系统将为进入保护模式做准备工作,
;系统此时不允许中断。
cli                  ;屏蔽中断
;设置中断描述表IDT和全局描述表GDT寄存器的初值
lidt    [IDT_Pointer]
lgdt    [GDT_Pointer]
;初始化中断控制器的值,
;使A20地址线有效,系统可以访问4GB的地址空间
call    Init_PIC
call    Enable_A20
;使CR0寄存器的PE位置1,切换到保护模式
mov ax, 0x01
lmsw    ax
;跳转到32位保护模式的代码区执行
jmp dword KERNEL_CS:(SETUPSEG << 4) + setup_32
```

```
[BITS 32] ; 32位编程模式
setup_32:
; 设置数据段寄存器
mov ax, KERNEL_DS
mov ds, ax
mov es, ax
mov fs, ax
mov gs, ax
mov ss, ax
; 为初始化核心级线程建立堆栈指针
mov esp, KERN_STACK + 4096
; 在堆栈构建 Boot_Info数据结构
; 进栈顺序与使用顺序应该相反
xor eax, eax
mov ax, [(SETUPSEG<<4)+mem_size_kbytes]
push eax ; 存储器容量大小=常规内存1MB+扩展内存
push dword 8 ; boot_Info数据结构大小
; 把Boot_Info数据结构的指针作为参数给内核入口函数
push esp
; 当系统出错的返回地址进栈
push dword (SETUPSEG<<4)+.returnAddr
; 跳转到内核程序入口执行Main函数
jmp KERNEL_CS:ENTRY_POINT
.returnAddr:
; 系统不应该返回到这里,否则进入死循环
.here: jmp .here
[BITS 16] ; 16位编程模式
; 关闭软驱马达
; 这段代码是从Linux系统复制过来的
Kill_Motor:
mov dx, 0x3f2
xor al, al
out dx, al
ret
Init_PIC:
; 初始化主PIC和从PIC的值,重新对中断进行编程
mov al, ICW1
out 0x20, al ; 主PIC的ICW1
call Delay
out 0xA0, al ; 从PIC的ICW1
call Delay
mov al, ICW2_MASTER
out 0x21, al ; 主PIC的ICW2
call Delay
```

```
mov al, ICW2_SLAVE
out 0xA1, al      ; 从PIC的ICW2
call Delay
mov al, ICW3_MASTER
out 0x21, al      ; 主PIC的ICW3
call Delay
mov al, ICW3_SLAVE
out 0xA1, al      ; 从PIC的ICW3
call Delay
mov al, ICW4
out 0x21, al      ; 主PIC的ICW4
call Delay
out 0xA1, al      ; 从PIC的ICW4
call Delay
mov al, 0xff      ; 屏蔽从PIC所有中断
out 0xA1, al
call Delay
mov al, 0xfb      ; 除了2号中断
out 0x21, al      ; 屏蔽主PIC所有中断
call Delay
ret
Delay:           ; 延迟一段时间, 等待I/O操作完成
jmp .done
.done: ret
; 下面是启用A20地址线, 以至于系统能访问1MB以上的内存空间
Enable_A20:
mov al, 0xD1
out 0x64, al
call Delay
mov al, 0xDF
out 0x60, al
call Delay
ret
; -----
; 设置用到的数据
; -----
mem_size_kbytes: dw 0      ; 内存空间大小变量
; -----
; 全局描述表GDT为核心建立了一个32位的地址空间,
; 下面的GDT只是为运行32位代码建立一个初始环境,
; 内核以后将建立和管理它自己的GDT。
; -----
; GDT initialization stuff
NUM_GDT_ENTRIES equ 3      ; 在初始GDT中只有3个描述符
```

```

GDT_ENTRY_SZ equ 8          ; 每一个描述符的大小为8个字节
align 8, db 0              ;按8字节对齐访问内存

GDT:
; 描述符表项0是空的,没有用到
dw 0
dw 0
dw 0
dw 0
; 描述符表项1是核心代码段
dw 0xFFFF    ; 字节0和1是存放段界限的低16位
dw 0x0000    ; 字节2和3是存放段的基地址低16位
db 0x00      ; 字节4是存放段的基地址的高8位
db 0x9A      ; 段的属性为已在内存、0级、非系统代码段、可执行/只读
db 0xCF      ; 段粒度为页,指令使用32位地址,段界限的高4位
db 0x00      ; 字节7存放段的基地址的最高8位
; 描述符表项1是核心数据段和堆栈段
; 解释: Intel 堆栈操作是从高地址向低地址扩展
; 这样系统可以使用一个段来保存数据和堆栈
; 低地址区间存放数据,高地址区间存放堆栈
dw 0xFFFF    ; 字节0和1是存放段界限的低16位
dw 0x0000    ; 字节2和3是存放段的基地址低16位
db 0x00      ; 字节4是存放段的基地址的高8位
db 0x92      ; 段的属性为已在内存、0级、非系统代码段、可写/可读
db 0xCF      ; 段粒度为页,指令使用32位地址,段界限的高4位
db 0x00      ; 字节7存放段的基地址的最高8位
GDT_Pointer:
dw NUM_GDT_ENTRIES*GDT_ENTRY_SZ ; GDT空间大小
dd (SETUPSEG<<4) + GDT       ; GDT区间的基地址
IDT_Pointer:
dw 0           ; IDT的大小
dd 00          ; IDT的基地址

```

3. A20 地址线问题

1981 年 8 月,IBM 公司最初推出的个人计算机 IBM PC 使用的 CPU 是 Intel 8088。在这种微机中地址线只有 20 根 (A0~A19)。在当时内存 RAM 只有几百 KB 或不到 1MB 时,20 根地址线已足够用来寻址这些内存。其所能寻址的最高地址是 0xffff:0xffff,也即 0x10ffef。对于超出 0x100000 (1MB) 的寻址地址将默认地环绕到 0x0ffef。当 IBM 公司于 1985 年引入 AT 机时,使用的是 Intel 80286 CPU,具有 24 根地址线,最高可寻址 16MB,并且有一个与 8088 完全兼容的实模式运行方式。然而,在寻址值超过 1MB 时它却不能像 8088 那样实现地址寻址的环绕。但是当时已经有一些程序是利用这种地址环绕机制进行工作的。为了实现完全的兼容性,IBM 公司发明了使用一个开关来开启或禁止 0x100000 地址比特位。由于在当时的 8042 键盘控制器上恰好有空闲的端口引脚

(输出端口 P2，引脚 P21)，于是便使用了该引脚来作为与门控制这个地址比特位。该信号即被称为 A20。如果它为零，则比特 20 及以上地址位都被清除。从而实现了兼容性。

4.3.3 lowlevel.asm 代码分析

1. 功能描述

Lowlevel.asm 文件主要是提供底层中断和线程处理的代码。它是一个 32 位代码程序，编译时被链接到内核中。在文件中定义了在 IDT 中要使用的底层中断处理程序入口指针。同时也包含了中断处理代码和线程上下文切换代码。

2. 代码分析

```
; GeekOS操作系统Lowlevel.asm代码(src/geekos/Lowlevel.asm)
%include "defs.asm"
%include "symbol.asm"
[BITS 32]
; -----
; 宏定义
; -----
; 这是在int.h文件定义的中断状态数据结构的大小
INTERRUPT_STATE_SIZE equ 64
; 定义一个宏Save_Registers在调用一个处理函数之前，保存所有寄存器的内容。
; 这必须保持下面两个结构最新的内容：
%macro Save_Registers 0
    push    eax
    push    ebx
    push    ecx
    push    edx
    push    esi
    push    edi
    push    ebp
    push    ds
    push    es
    push    fs
    push    gs
%endmacro
; 定义了一个宏Restore_Registers，在调用另一个处理函数之后，
; 恢复所有寄存器的内容和清除堆栈数据（也就是仅仅在我们从中断处理程序
; 返回之前通过一条iret汇编指令实现）。
%macro Restore_Registers 0
    pop    gs
    pop    fs
```

```
pop es
pop ds
pop ebp
pop edi
pop esi
pop edx
pop ecx
pop ebx
pop eax
add esp, 8 ; 跳过中断号和出错处理代码
%endmacro
; 定义一个宏Activate_User_Context, 在返回到执行一个线程之前, 激活一个
; 新的用户上下文(如果需要)。
; 它应该在恢复寄存器内容之前被调用(因为中断的上下文要用到)。
%macro Activate_User_Context 0
    ; 如果新的线程有一个上下文, 不是当前正在使用的, 就激活它
    push    esp           ; Interrupt_State pointer
    push    dword [g_currentThread] ; Kernel_Thread pointer
    call    Switch_To_User_Context ; 系统从核心态切换到用户进程执行
    add    esp, 8          ; 清除前面刚压入堆栈的两个参数
%endmacro
; 定义一个宏Process_Signal, 实现重新整理堆栈来激活一个信号处理程序。
%macro Process_Signal 1
    ; 检查一个信号是否被挂起, 如果这样, 我们需要重新整理堆栈以至于
    ; 当线程重新开始时, 它将先进入信号处理程序执行, 然后再返回到它原
    ; 来的中断现场。
    push    esp
    push    dword [g_currentThread]
    call    Check_Pending_Signal
    add esp, 8
    cmp eax, dword 0
    je    *1
    ; 有一个信号被挂起, 这样系统必须安排调用它的信号处理函数。
    ;push    esp
    ;call    Print_IS
    ;add    esp, 4
    push    esp
    push    dword [g_currentThread]
    call    Setup_Frame
    add esp, 8
    ;push    esp
    ;call    Print_IS
    ;add    esp, 4
%endmacro
```

```
; 这是在通用寄存器和段寄存器被保存以后从堆栈的顶部到中断号之间的字节数。  
REG_SKIP equ (11*4)  
; 下面定义了中断入口程序段代码模板, 它有一个明显的处理器产生的出错  
; 处理代码, 参数为中断号  
%macro Int_With_Err 1  
align 8  
    push    dword %1      ; 压入中断号为参数  
    jmp Handle_Interrupt ; 跳到通用的中断处理程序  
%endmacro  
; 下面定义了中断入口程序段代码模板, 处理器没有产生一个明显的出错处理  
; 代码, 我们将压入一个伪装的出错代码到堆栈, 以至于堆栈的格式和所有的  
; 中断处理相同。  
%macro Int_No_Err 1  
align 8  
    push    dword 0      ; 伪装出错处理代码  
    push    dword %1      ; 压入中断号为参数  
    jmp Handle_Interrupt ; 跳到通用的中断处理程序  
%endmacro  
-----  
; EXPORT 定义一个符号作为全局变量, IMPORT 访问一个在其他模块定义的符号  
;  
; g_interruptTable是在文件idt.c中定义的符号, 它是C中断处理程序的地址表  
IMPORT g_interruptTable  
; g_currentThread是一个指向当前执行线程上下文的全局变量指针  
IMPORT g_currentThread  
; g_needReschedule是一个在中断返回时用来决定是否需要选择一个新线程的变量  
IMPORT g_needReschedule  
; g_preemptionDisabled用来决定系统执行方式是否为抢占式  
IMPORT g_preemptionDisabled  
; Get_Next_Runnable是选择下一个运行线程的函数  
IMPORT Get_Next_Runnable  
; Make_Runnable定义了一个放一个线程到运行队列的函数  
IMPORT Make_Runnable  
; Switch_To_User_Context是一个激活一个新用户线程上下文的函数  
IMPORT Switch_To_User_Context  
; Check_Pending_Signal是一个用来检查当前线程是否有一个挂起信号的函数  
IMPORT Check_Pending_Signal  
; Setup_Frame是一个用来设置一个信号处理程序堆栈帧的函数  
IMPORT Setup_Frame  
; 定义了中断处理程序入口地址表的开始和结束地址符号  
EXPORT g_handlerSizeNoErr  
EXPORT g_handlerSizeErr  
; 简单装入IDTR, GDTR, 和LDTR的函数符号定义  
EXPORT Load_IDTR
```

```
EXPORT Load_GDTR
EXPORT Load_LDTR
; 定义了中断入口地址表的开始和结束地址符号
EXPORT g_entryPointTableStart
EXPORT g_entryPointTableEnd
; 定义了线程上下文切换函数的符号定义
EXPORT Switch_To_Thread
; 定义了返回当前状态寄存器值的符号
EXPORT Get_Current_EFLAGS
; 定义了支持虚拟存储器用到的符号
EXPORT Enable_Paging
EXPORT Set_PDBR
EXPORT Get_PDBR
EXPORT Flush_TLB

; -----
; 功能代码定义
; -----
[SECTION .text]
; 装入IDTR的代码,以6字节的地址指针为参数
align 8
Load_IDTR:
    mov eax, [esp+4]
    lidt    [eax]
    ret

; 装入GDTR的代码,以6字节的地址指针为参数,假定中断是被屏蔽的
align 8
Load_GDTR:
    mov eax, [esp+4]
    lgdt   [eax]
    ; Reload segment registers
    mov ax, KERNEL_DS
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
    jmp KERNEL_CS:.here
.here:
    ret

; 装入LDT的代码,以选择子为参数。
align 8
Load_LDTR:
    mov eax, [esp+4]
```

```

    lldt      ax
    ret
; 开始使用分页系统处理,
; 把页目录表指针装入cr3寄存器,设置cr2寄存器,使分页有效
align 8
Enable_Paging:
    mov eax, [esp+4]
    mov cr3, eax
    mov eax, cr3
    mov cr3, eax
    mov ebx, cr0
    or ebx, 0x80000000
    mov cr0, ebx
    ret
; 通过把传送的页目录表指针装入cr3改变PDBR
align 8
Set_PDBR:
    mov eax, [esp+4]
    mov cr3, eax
    ret
; 获得当前执行进程的PDBR,通过修改cr3中PDBR内容,就可实现进程切换
align 8
Get_PDBR:
    mov eax, cr3
    ret
; 刷新TLB内容,仅仅需要重新装入cr3的内容就可以了
align 8
Flush_TLB:
    mov eax, cr3
    mov cr3, eax
    ret
; 通用的中断处理程序代码,主要是保存寄存器的内容,调用C语言处理函数,
; 最后可能选择一个新的线程来运行,恢复相应线程的寄存器内容,从中断返回
align 8
Handle_Interrupt:
    ; 保存通用的寄存器和段寄存器的内容
    Save_Registers
    ; 确保现在使用的是核心数据段
    mov ax, KERNEL_DS
    mov ds, ax
    mov es, ax
    ; 从处理函数表得到C语言处理函数的地址
    mov eax, g_interruptTable    ; 得到处理表的地址
    mov esi, [esp+REG_SKIP]      ; 得到中断号

```

```

    mov ebx, [eax+esi*4]          ; 得到相应中断处理函数的地址
    ; 调用处理函数,参数是一个指向Interrupt_State结构的指针。
    push    esp
    call    ebx
    add esp, 4                   ; 清除刚压入的一个参数
    ; 如果抢占方式被屏蔽,当前线程继续运行
    cmp [g_preemptionDisabled], dword 0
    jne .restore
    ; 确定系统是否需要选择一个新的线程运行
    cmp [g_needReschedule], dword 0
    je  .restore
    ; 把当前线程放入运行队列中等待下次被运行
    push    dword [g_currentThread]
    call    Make_Runnable
    add esp, 4                   ; 清除刚压入的一个参数
    ; 保存堆栈指针到当前线程的上下文,清除numTicks域的值
    mov eax, [g_currentThread]
    mov [eax+0], esp
    mov [eax+4], dword 0         ; 使numTicks域的值为0
    ; 调用Get_Next_Runnable让一个新线程来运行,切换到它相应的堆栈区间
    call    Get_Next_Runnable
    mov [g_currentThread], eax
    mov esp, [eax+0]             ; esp指向新的堆栈区
    ; 清除need reschedule标志
    mov [g_needReschedule], dword 0
.restore:
    Process_Signal .finish
.finish:
    ; 如果需要,激活用户的上下文
    Activate_User_Context
    ; 恢复寄存器的内容
    Restore_Registers
    ; 从中断返回
    iret
;
; -----
; Switch_To_Thread() 函数是保存当前执行线程的上下文,激活被传递参数对应
; 线程的上下文。
; 参数: 指向要激活的核心线程结构的指针。
; 注意: 调用此函数时必须屏蔽中断,保持核心线程结构的定义不被修改。
;
.align 16
Switch_To_Thread:
    ; 修改堆栈指针允许以后通过一条iret指令返回

```

```
push    eax          ; 保存eax寄存器内容到堆栈
mov eax, [esp+4]      ; 获取当前线程的返回地址
mov [esp-4], eax      ; 移动返回地址到当前堆栈指针下8个字节处
add esp, 8            ; 移动堆栈指针到顶部
pushfd               ; 压入返回地址时刻的eflags到堆栈
mov eax, [esp-4]      ; 恢复刚保存的eax寄存器内容
push    dword KERNEL_CS ; 压入cs代码段选择子到堆栈
sub esp, 4            ; 安排堆栈指针指向保存返回地址的位置
; 压入伪装的出错代码和中断号到堆栈
push    dword 0
push    dword 0
; 保存寄存器的内容
Save_Registers
; 保存堆栈指针到线程的上下文结构中
mov eax, [g_currentThread]
mov [eax+0], esp
; 在线程的上下文中清除numTicks域的值，因为当前线程被停止运行了,
; 下一次重新计数。
mov [eax+4], dword 0
; 装入指向新的线程上下文的指针到eax寄存器,
; 在堆栈区跳过Interrupt_State struct得到相应的参数。
mov eax, [esp+INTERRUPT_STATE_SIZE]
; 切换到新的线程的堆栈,使得新的线程成为当前线程。
mov [g_currentThread], eax
mov esp, [eax+0]
Process_Signal .complete
.complete:
; 如果需要,则激活用户的上下文。
Activate_User_Context
; 恢复通用寄存器和段寄存器的内容,清除中断号和出错代码。
Restore_Registers
; 系统将返回最后执行的线程继续执行
iret
; 返回当前标志寄存器的内容
align 16
Get_Current_EFLAGS:
pushfd               ; 压入eflags到堆栈
pop eax              ; 从堆栈中弹出eflags内容到eax寄存器
ret
;
; 为所有中断生成指定中断的入口点
;
align 8
g_entryPointTableStart:
```

```
; 下面是处理器产生异常的处理程序的入口点, 这些是在Intel 486 manual中定义的。
Int_No_Err 0
align 8
Before_No_Err:
Int_No_Err 1
align 8
After_No_Err:
Int_No_Err 2      ; FIXME:未在486手册中说明
Int_No_Err 3
Int_No_Err 4
Int_No_Err 5
Int_No_Err 6
Int_No_Err 7
align 8
Before_Err:
Int_With_Err 8
align 8
After_Err:
Int_No_Err 9      ; FIXME:未在486手册中说明
Int_With_Err 10
Int_With_Err 11
Int_With_Err 12
Int_With_Err 13
Int_With_Err 14
Int_No_Err 15    ; FIXME:未在486手册中说明
Int_No_Err 16
Int_With_Err 17
; 下面的中断 (18~255) 没有出错代码
; 系统通过一个nasm的 %rep 结构生成它们全部
%assign intNum 18
%rep (256 - 18)
Int_No_Err intNum
%assign intNum intNum+1
%endrep
align 8
gEntryPointTableEnd:
[SECTION .data]
; 下面是定义了有关错误处理中断程序入口点大小的符号
; 分为有错误代码和没有错误代码两种情况
align 4
g_handlerSizeNoErr: dd (After_No_Err - Before_No_Err)
align 4
g_handlerSizeErr: dd (After_Err - Before_Err)
```

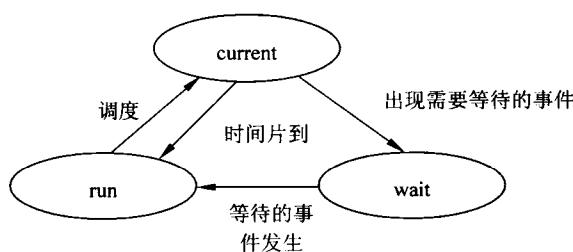
进程是在计算机系统引入多道程序设计技术后，为描述和控制系统内部各程序运行而引入的一个概念。后来，为了适应计算机新的体系结构，提高程序运行的并行度和系统执行效率，计算机系统在进程概念基础之上又引入了线程概念，线程是进程中一个运行单位，共享进程所拥有的资源。GeekOS 系统中的线程实际上就是进程，GeekOS 提供的内核只支持核心级进程，但通过系统开发和扩充可以支持核心级进程和用户态进程两种。

5.1 GeekOS 进程状态及转换

在操作系统中引入进程概念，虽然可以实现多个任务共享 CPU，但一个进程在整个生命周期中有时能占用 CPU 处于运行；有时虽然可以运行，但由于分配不到 CPU 而等待运行；有时虽然 CPU 空闲但进程却因等待某个事件的发生而无法执行。在 GeekOS 系统中，为了描述进程这些状态，进程执行过程可以分为以下 3 种不同的状态。

- 当前运行（current）状态：进程占用 CPU，正在运行。
- 准备运行（run）状态：进程具备运行条件，等待调度程序调度，以便执行。
- 等待（wait）状态：进程正在等待某个事件发生。

进程在执行过程中任一时刻仅处于 3 种不同状态之一，图 5-1 展示了进程之间的状态转换关系。



3 种状态发生转换的原因如下。

- **current→run:** 运行时间片到或 run 队列出现优先级更高的进程。
- **current→wait:** 运行进程需要等待某种资源或等待某个事件发生, 如需要键盘输入。
- **wait→run:** 资源得到满足, 或者等待的事件发生。
- **run→current:** CPU 空闲时, 进程被调度程序选中运行。

GeekOS 系统中为不同状态的进程准备了不同的进程队列 (Thread Queue)。如果一个进程正处于准备运行状态, 就会在队列 s_runQueue 中出现。如果一个进程处于等待状态, 就会在 s_reaperWaitQueue 队列中出现。如果一个进程准备被销毁, 就会在 s_graveyardQueue 队列中出现。由于占用 CPU 运行的进程最多只能有一个, 所以没有队列, 由指针 g_currentThread 指向该进程。

5.2 GeekOS 内核进程

5.2.1 内核进程控制块

系统中每个内核进程有且仅有一个进程控制块, 进程控制块是用于记录进程状态及有关信息的数据结构。GeekOS 操作系统中用数据结构 Kernel_Thread 作为内核进程控制块, 对系统中的进程信息、执行情况、控制信息等加以维护, 这个结构在 include/kthread.h 中定义, 具体结构如下。

```
struct Kernel_Thread {
    ulong_t esp;                                // 进程的内核堆栈esp指针
    volatile ulong_t numTicks;                   // 计时器
    int priority;                               // 进程优先级
    DEFINE_LINK(Thread_Queue, Kernel_Thread);
    // 指针指向进程队列下一进程
    void* stackPage;                            // 内核堆栈页指针
    struct User_Context* userContext;           // 用户进程上下文
    struct Kernel_Thread* owner;                // 父进程指针
    int refCount;                               // 引用计数
    bool alive;                                 // 是否活跃
    struct Thread_Queue joinQueue;              // 加入队列
    int exitCode;                               // 返回代码
    int pid;                                    // 进程ID
    DEFINE_LINK(All_Thread_List, Kernel_Thread); // 全局进程链表指针
#define MAX_TLOCAL_KEYS 128
    const void* tlocalData[MAX_TLOCAL_KEYS];     // 本地信息
    int currentReadyQueue;                      // 进程当前所在的运行队列的索引编号
    bool blocked;                              // 是否被阻塞
};
```

5.2.2 GeekOS 系统中最早的内核进程

GeekOS 系统最早创建的内核进程有 Idle、Reaper 和 Main 3 个进程。用户可以看到，在系统初始化时，Main 函数调用了一系列系统初始化函数（部分代码如下）。

```
...
Init_BSS();
Init_Screen();
Init_Mem(bootInfo);
Init_CRC32();
Init_TSS();
Init_Interrupts();
Init_Scheduler();
Init_Traps();
Init_Timer();
Init_Keyboard();
Init_DMA();
Init_Floppy();
Init_IDE();
Init_PFAT();
...
...
```

其中，除了初始化外设、中断、计时器等，还调用了 Init_Scheduler 函数(`\src\geekos\kthread.c`)，代码如下。

```
void Init_Scheduler(void)
{
    struct Kernel_Thread* mainThread = (struct Kernel_Thread *)
KERN_THREAD_OBJ; //定义指向进程控制块的指针
/* KERN_THREAD_OBJ代表(1024 * 1024)在defs.h中定义*/
    Init_Thread(mainThread, (void *) KERN_STACK, PRIORITY_NORMAL, true);
//初始化main进程，指定进程优先级为PRIORITY_NORMAL
/*在kthread.h中定义了5个进程优先级PRIORITY_IDLE, PRIORITY_USER,
PRIORITY_LOW, PRIORITY_NORMAL和PRIORITY_HIGH, 分别代表0, 1, 2, 5, 10,
Init_Thread函数分析见本章*/
    g_currentThread = mainThread;
    // g_currentThread是系统中的指向当前执行进程的指针
    Add_To_Back_Of_All_Thread_List(&s_allThreadList, mainThread);
    //将进程连入系统的进程队列
    Start_Kernel_Thread(Idle, 0, PRIORITY_IDLE, true);
    /*创建Idle进程*/
    Start_Kernel_Thread(Reaper, 0, PRIORITY_NORMAL, true);
    /*创建Reaper进程*/
}
```

该函数的功能是初始化一个核态进程 mainThread，并将该进程作为当前运行进程，函数最后还调用 Start_Kernel_Thread 函数创建了两个系统进程 Idle 和 Reaper。所以，Idle、Reaper 和 Main 3 个进程是系统中最早存在的进程。

类似 Windows 系统的系统闲置进程，内核进程 Idle 实际上什么事情也不做，Idle 在创建后就一直存在于系统中，它存在的惟一目的是保证准备运行进程队列中有可调度的进程。Idle 代码(\src\geekos\kthread.c)如下。

```
static void Idle(ulong_t arg)
{
    while (true)
        Yield();
}
```

Idle 进程始终放在准备运行队列的队尾，当系统中没有可运行的进程时，CPU 就运行 Idle 进程，一旦系统中有其他准备运行的进程进入，Idle 进程立即放弃 CPU，系统会将其重新放入准备运行队列的队尾。

Idle 进程中，Yield 函数 (\src\geekos\kthread.c) 的功能是在准备运行队列有可运行的进程时，调用 Schdule 函数从队列中选择一个进程，并调用函数 Switch_To_Thread（其定义在 lowlevel.asm 文件中）使新选择的进程占用 CPU 运行。

```
void Yield(void)
{
    Disable_Interrupts();
    Make_Runnable(g_currentThread);
    Schedule();
    Enable_Interrupts();
}
```

内核进程 Reaper 负责消亡进程的善后工作，如释放消亡进程占用的资源，内存、堆栈等。系统中将消亡后需要处理的进程记录在 s_graveyardQueue 队列中，Reaper 进程首先检测 s_graveyardQueue 队列是否为空，如果队列为空，表示暂时没有消亡进程，则阻塞 Reaper 进程，Reaper 进程开始等待信号量 s_reaperWaitQueue。若系统有新的消亡进程进入 s_graveyardQueue 队列，而此时 Reaper 进程在等待，就唤醒 Reaper 进程，进行进程消亡的处理工作。Reaper 进程代码 (\src\geekos\kthread.c) 如下。

```
static void Reaper(ulong_t arg)
{
    struct Kernel_Thread *kthread;
    Disable_Interrupts(); //关闭中断
    while (true) {
        /* 查看系统中是否有需要善后的进程 */
        if ((kthread = s_graveyardQueue.head) == 0) {
```

```

/* 若暂时没有需要善后的进程，Reaper进程等待 */
Wait(&s_reaperWaitQueue);

}

else {
    /* 若有进程在等待善后，则记录下需要善后进程队列的队首，并把系统的
     s_graveyardQueue 队列清空*/
    Clear_Thread_Queue(&s_graveyardQueue);
    Enable_Interrupts();
    Yield();    /* 调度其他进程运行 */
    while (kthread != 0) { //进行所有消亡进程的善后工作
        struct Kernel_Thread* next = Get_Next_In_Thread_Queue(kthread);

#if 0
        Print("Reaper: disposing of thread @ %x, stack @ %x\n",
              kthread, kthread->stackPage);
#endif

        Destroy_Thread(kthread); //撤销kthread进程并回收资源
        kthread = next;
    }
    Disable_Interrupts();
}
}
}

```

系统初始化时，除了创建上述两个系统进程外，Main 函数本身也作为内核进程运行，只不过它本身的初始化比较特殊，而且随着 Main 函数执行完毕会调用 exit 函数，使 Main 函数退出，而其他系统进程不会调用 exit 退出，它们的生命周期将伴随系统的运行直到用户关机。

5.2.3 内核进程对象

内核进程对象是 GeekOS 系统进程存在的一个体现，GeekOS 系统中所有进程都有一个内核进程对象。内核进程对象由两部分组成：Kernel_Thread 结构体以及一个内核堆栈。这两个结构用 Kernel_Thread 中的 esp 指针与 stackPage 指针联系起来。内核态堆栈的一个重要功能是保存进程切换时被切换进程的上下文。因此，一个 GeekOS 的进程至少需要具备内核进程对象与进程程序体两个部分。

由上节可知，创建一个 GeekOS 内核进程需要调用 Start_Kernel_Thread 函数，而 Start_Kernel_Thread 函数内部调用 Create_Thread 函数，该函数主要是创建内核进程对象，并调用 Alloc_Page 函数为进程对象、进程内核堆栈各分配一页内存（若内存分配失败，则返回 0，同时释放内核进程控制块空间）。

一般进程通过调用函数 Create_Thread 创建，而 Main 比较特殊，Main 进程的任务是初始化计算机并生成其他进程，是系统中最早执行的代码段，在调用任何创建函数前就

已经开始执行了，无法通过 `Start_Kernel_Thread` 函数初始化。因此其内核进程对象在 `Init_Mem` 函数执行时被分配空间，之后在 `Init_Scheduler` 函数执行时进行初始化。`Main` 进程只运行一次，仅出现在全局进程链表中，且不参与进程调度。

5.3 进程调度

5.3.1 内核进程切换

`GeekOS` 在几种情况下会进行进程切换：一是时间片用完时，二是执行内核进程 `Idle` 时，三是进程退出调用 `Exit` 函数时，四是进程进入等待状态调用 `Wait` 函数时。

第一种情况下，当时间片用完时引发中断，进程切换在中断处理函数 `Handle_Interrupt`（在 `lowlevel.asm` 中定义）中完成。`Handle_Interrupt` 首先将待切换的进程运行信息保存，然后从中断向量表取得中断处理程序的地址，中断处理结束后，重新调度一个新的进程运行。

后 3 种情况下，函数内部都有调用 `Schedule()` 函数，`Schedule` 函数定义在 `\src\geekos\kthread.c` 文件中，具体代码如下。

```
void Schedule(void)
{
    struct Kernel_Thread* runnable;
    KASSERT(!Interrupts_Enabled()); /*确认中断已屏蔽*/
    KASSERT(!g_premptionDisabled); /*确认系统允许抢占*/
    runnable = Get_Next_Runnable(); /*从准备运行的进程队列取一个进程*/
    Switch_To_Thread(runnable); /*切换到新进程运行*/
}
```

`Switch_To_Thread` 函数（在 `lowlevel.asm` 中定义）接受从运行队列取下的指向 `Kernel_Thread` 结构的指针作为参数，首先将中断状态结构 `Interrupt_State` 压入当前的内核堆栈，将当前内核堆栈指针压入堆栈；由于发生进程切换一般来说表明本进程时间片已用完，因此会清除当前进程的时间片计数，使得下次重新调度此进程时可以从头开始计时（注：对于退出进程而言，重置时间片不起任何作用，因为这个内核堆栈空间稍后就会释放；对于进入等待队列的进程，相当于重新赋予进程运行时间）；之后装入调度选中进程的 `Kernel_Thread` 结构指针（即传递给 `Switch_To_Thread` 函数的参数）并切换到其内核堆栈；最后将目标进程的中断状态结构从内核堆栈弹出，并执行一个 `iret` 指令中断返回，跳入目标进程。

5.3.2 用户进程切换

用户进程切换比内核进程要复杂，因为内核进程的进程体在内核的代码段范围内，内核进程的切换使用段内返回就可以了，但用户进程拥有自己的地址空间，因此执行的

是中断返回。用户进程切换同样使用 lowlevel.asm 中的 Switch_To_Thread 函数，只是在进行用户进程切换的过程中多了对 Activate_User_Context 函数的调用。这个函数定义如下。

```
%macro Activate_User_Context 0
    push    esp          ; 中断状态指针
    push    dword [g_currentThread] ; 当前运行进程指针
    call    Switch_To_User_Context
    add     esp, 8        ;
%endmacro
```

这个函数将内核堆栈 esp 指针和当前运行进程的 Kernel_Thread 结构指针作为参数压入内核堆栈，之后调用 Switch_To_User_Context 函数，在函数返回后，将两个参数清除。

Switch_To_User_Context 函数在\src\geekos\user.c 文件中实现，它负责检测当前进程是否为用户进程，如果是用户进程，就切换到用户进程空间。这个函数由开发者自己实现。

5.3.3 GeekOS 进程调度策略

GeekOS 的初始系统提供的进程调度是时间片轮转调度，所有准备运行进程（其实是进程的控制块 Kernel Thread 结构）都放在一个 FIFO（First In First Out）队列里面，新建进程放在该队列尾部。发生进程调度时，系统在准备运行队列中查找优先级最高的进程投入运行。在 GeekOS 中，Get_Next_Runnable 函数就是进程调度算法实现的地方，由\src\geekos\kthread.c 文件中的 Find_Best 函数在准备运行进程的队列(s_runQueue 指针指向该队列队首) 中查找，找到优先级最高的进程作为下一个占用 CPU 的进程。

```
// Find_Best函数代码
static __inline__ struct Kernel_Thread* Find_Best(struct Thread_Queue*
queue)
{
    struct Kernel_Thread *kthread = queue->head, *best = 0;
    while (kthread != 0) {
        if (best == 0 || kthread->priority > best->priority)
            best = kthread;
        kthread = Get_Next_In_Thread_Queue(kthread);
    }
    return best;
}
```

用户可以修改 GeekOS 内核，扩充和修改进程调度策略。项目 3 就提供了改进调度策略的一些建议，目的是实现一个四队列的多级反馈队列调度算法。多级反馈调度的主要思想是将准备运行进程分成多个队列（初始系统仅有一个队列），给予队列不同的优先

级和时间片，一般较高优先级的队列分配较短的时间片。进程调度的时候一般先从高一级的准备运行队列中选择进程运行，同一队列的进程按照先来先服务的原则排队，只有在高一优先级的队列为空时，进程调度才从低一级的进程队列中选取进程运行。新创建的进程首先被放入最高优先级的准备运行队列，若进程被选中运行相应时间片后还没有运行结束，进程就被放入优先级低一级的队列中等待。值得注意的是：由于多级反馈队列调度总是优先调度高优先级队列里面的进程，所以可能导致低优先级的进程发生进程饥饿现象，即长时间不能得到系统调度运行。

5.4 内核进程主要操作函数

内核进程的主要操作函数和数据结构都在文件\src\geekos\kthread.c 中定义。在操作函数中出现的主要数据结构如下。

```
//系统所有内核进程链表
static struct All_Thread_List s_allThreadList;
//准备运行进程队列
static struct Thread_Queue s_runQueue;
//当前运行进程指针
struct Kernel_Thread* g_currentThread;
//运行结束需要处理的进程队列
static struct Thread_Queue s_graveyardQueue;
//进程队列，用于退出进程和系统reaper进程之间的通信
static struct Thread_Queue s_reaperWaitQueue;
```

5.4.1 Init_Thread 函数

函数功能：初始化进程信息。

函数参数：进程对象指针，进程堆栈指针，进程优先级，进程是否为子进程。

函数返回值：无。

```
static void Init_Thread(struct Kernel_Thread* kthread, void* stackPage,
    int priority, bool detached)
{
    static int nextFreePid = 1;
    struct Kernel_Thread* owner = detached ? (struct Kernel_Thread*)0 : g_currentThread;
    /*若detached为1，表示该进程是独立创建的进程，否则进程是当前运行进程请求创建的子进程*/
    memset(kthread, '\0', sizeof(*kthread));
    //初始化进程对象内存空间为0
    kthread->stackPage = stackPage;
    //初始化进程堆栈地址
    kthread->esp = ((ulong_t) kthread->stackPage) + PAGE_SIZE;
```

```

kthread->numTicks = 0;
//设置进程已运行的时间为0
kthread->priority = priority;
//设置进程优先级
kthread->userContext = 0;
//初始化进程为内核进程
kthread->owner = owner;
//初始化进程的父进程指针
kthread->refCount = detached ? 1 : 2;
//设置引用该进程的进程数,若是独立进程,值为1;若是子进程,值为2
kthread->alive = true;
Clear_Thread_Queue(&kthread->joinQueue);
kthread->pid = nextFreePid++;
/*设置进程id,并将nextFreePid加1,用于下次创建新进程,nextFreePid是系统全局整型变量,初值为1*/
}
}

```

5.4.2 Create_Thread 函数

函数功能：创建一个新的进程对象。

函数参数：进程优先级，进程是否为子进程（调用 Init_Thread 函数时使用）。

返回值：若内存不够，返回空指针，否则返回进程对象指针。

```

static struct Kernel_Thread* Create_Thread(int priority, bool detached)
{
    struct Kernel_Thread* kthread;
    void* stackPage = 0;
    kthread = Alloc_Page(); //为进程对象分配一页内存
    if (kthread != 0)
        stackPage = Alloc_Page(); //为进程堆栈分配一页内存
    if (kthread == 0)
        return 0;
    if (stackPage == 0) {
        Free_Page(kthread);
        return 0; //若进程对象或堆栈内存分配失败，则返回0
    }
    Init_Thread(kthread, stackPage, priority, detached);
    //初始化进程堆栈指针和进程信息
    Add_To_Back_Of_All_Thread_List(&s_allThreadList, kthread);
    //将进程链入系统所有进程的链表,以便统一管理
    return kthread; //返回进程对象指针
}

```

5.4.3 Destroy_Thread 函数

函数功能：撤销给定进程，并回收进程所占用的资源。

函数参数：待撤销进程的指针。

返回值：无。

```
static void Destroy_Thread(struct Kernel_Thread* kthread)
{
    Disable_Interrupts();
    Free_Page(kthread->stackPage); //回收分配给进程的堆栈空间
    Free_Page(kthread);
    Remove_From_All_Thread_List(&s_allThreadList, kthread);
    //将进程从系统进程链表移出
    Enable_Interrupts();
}
```

5.4.4 Reap_Thread 函数

函数功能：将给定进程放入进程回收队列，等待撤销。

函数参数：待回收进程的指针。

返回值：无。

```
static void Reap_Thread(struct Kernel_Thread* kthread)
{
    KASSERT(!Interrupts_Embled());
    Enqueue_Thread(&s_graveyardQueue, kthread);
    /*将参数指向的进程放入待回收的进程队列，唤醒正在等待进程回收信号量的reaper进程*/
    Wake_Up(&s_reaperWaitQueue);
}
```

5.4.5 Detach_Thread 函数

函数功能：在参数进程的引用断开后，调用该函数。

函数参数：进程指针。

返回值：无。

```
static void Detach_Thread(struct Kernel_Thread* kthread) {
    KASSERT(!Interrupts_Embled());
    KASSERT(kthread->refCount > 0);
    --kthread->refCount;
    if (kthread->refCount == 0) {
        Reap_Thread(kthread); //进程引用为0，表示进程可以回收
    }
}
```

5.4.6 Start_Kernel_Thread 函数

函数功能：生成内核进程，以给定 startFunc 函数体作为进程体。

函数参数：作为进程体的函数指针，传递给进程体函数的参数，进程优先级，进程是否为子进程的标志。

返回值：刚生成进程的指针。

```
struct Kernel_Thread* Start_Kernel_Thread(Thread_Start_Func startFunc,
                                         ulong_t arg, int priority,
                                         bool detached)
{
    struct Kernel_Thread* kthread = Create_Thread(priority, detached);
    if (kthread != 0) {
        Setup_Kernel_Thread(kthread, startFunc, arg);
        Make_Runnable_Atomic(kthread);
    }
    return kthread;
}
```

5.4.7 Setup_Kernel_Thread 函数

函数功能：根据参数将有关参数压栈，为进程准备运行条件。

函数参数：进程指针，进程体函数入口指针，参数。

返回值：无。

```
static void Setup_Kernel_Thread(struct Kernel_Thread* kthread,
                                Thread_Start_Func startFunc,
                                ulong_t arg)
{
    /* 将参数和 Shutdown_Thread 返回值压入进程体函数，以便进程体函数返回时可以退出 */
    Push(kthread, arg);
    Push(kthread, (ulong_t) &Shutdown_Thread);
    /* 将进程函数体入口地址压栈 */
    Push(kthread, (ulong_t) startFunc);
    /* 为使进程有机会被调度，必须使进程看起来像被中断运行一样，所以必须压入“eflags,
     * cs, eip”序列到堆栈 */
    Push(kthread, OUL); /* EFLAGS */
    Push(kthread, KERNEL_CS);
    Push(kthread, (ulong_t) &Launch_Thread);
    /* 压入伪造的错误代码和中断号 */
    Push(kthread, 0);
    Push(kthread, 0);
```

```

/*为通用寄存器压入初值*/
Push_General_Registers(kthread);
Push(kthread, KERNEL_DS); /* ds */
Push(kthread, KERNEL_DS); /* es */
Push(kthread, 0); /* fs */
Push(kthread, 0); /* gs */
}

```

5.4.8 Make_Runnable 函数

函数功能：将参数给定的进程链入准备运行队列。

函数参数：进程指针。

返回值：无。

```

void Make_Runnable(struct Kernel_Thread* kthread)
{
    KASSERT(!Interrupts_Enabled());
    Enqueue_Thread(&s_runQueue, kthread);
}

```

5.4.9 Make_Runnable_Atomic 函数

函数功能：调用函数 Make_Runnable，将参数指向的进程链入准备运行队列。

函数参数：进程指针。

返回值：无。

```

void Make_Runnable_Atomic(struct Kernel_Thread* kthread)
{
    Disable_Interrupts();
    Make_Runnable(kthread);
    Enable_Interrupts();
}

```

5.4.10 Get_Current 函数

函数功能：获取系统当前运行的进程指针。

函数参数：无。

返回值：当前运行进程的指针。

```

struct Kernel_Thread* Get_Current(void)
{
    return g_currentThread;
// g_currentThread是全局变量，存储当前运行进程的指针
}

```

5.4.11 Get_Next_Runnable 函数

函数功能：获取下一个准备运行队列。

函数参数：无。

返回值：进程指针。

```
struct Kernel_Thread* Get_Next_Runnable(void)
{
    struct Kernel_Thread* best = 0;
    best = Find_Best(&s_runQueue);
    //根据进程调度算法获取最佳准备运行进程
    KASSERT(best != 0);
    Remove_Thread(&s_runQueue, best);
    //将进程从准备运行队列移出
    return best;
}
```

5.4.12 Schedule 函数

函数功能：进程调度函数。

函数参数：无。

返回值：无。

```
void Schedule(void)
{
    struct Kernel_Thread* runnable;
    KASSERT(!Interrupts_Enabled());
    /*保证进程抢占是允许的*/
    KASSERT(!g_preemptionDisabled);
    /*从准备运行队列获取进程*/
    runnable = Get_Next_Runnable();
    Switch_To_Thread(runnable); //切换到新进程运行
}
```

5.4.13 Join 函数

函数功能：等待给定进程死亡。

函数参数：进程指针。

返回值：退出代码。

```
int Join(struct Kernel_Thread* kthread)
```

```

{
    int exitCode;
    KASSERT(Interrupts_Enabled());
    KASSERT(kthread->owner == g_currentThread);
    Disable_Interrupts();
    while (kthread->alive) {
        Wait(&kthread->joinQueue); //若进程活动，则等待
    }
    exitCode = kthread->exitCode; //获得进程退出代码
    Detach_Thread(kthread); //去除该进程的引用
    Enable_Interrupts();
    return exitCode;
}

```

5.4.14 Lookup_Thread 函数

函数功能：根据进程 id 查找进程，只有进程的 owner 才能使用该函数。

函数参数：进程 id。

返回值：进程指针。

```

struct Kernel_Thread* Lookup_Thread(int pid)
{
    struct Kernel_Thread *result = 0;
    bool iflag = Begin_Int_Atomic();
    result = Get_Front_Of_All_Thread_List(&s_allThreadList);
    //获取进程链表的头指针
    while (result != 0) {
        if (result->pid == pid) {
            if (g_currentThread != result->owner)
                result = 0;
            break;
        }
        result = Get_Next_In_All_Thread_List(result);
    }
    End_Int_Atomic(iflag);
    return result;
}

```

5.4.15 Wait 函数

函数功能：将当前运行进程放入指定的等待队列，并重新调度新进程运行。

函数参数：等待队列指针。



返回值：无。

```
void Wait(struct Thread_Queue* waitQueue)
{
    struct Kernel_Thread* current = g_currentThread;
    KASSERT(!Interrupts_Enabled());
    Enqueue_Thread(waitQueue, current); //将当前进程链入等待队列
    Schedule(); //调度新进程运行
}
```

5.4.16 Wake_Up 函数

函数功能：唤醒所有在指定等待队列等待的进程。

函数参数：等待队列指针。

返回值：无。

```
void Wake_Up(struct Thread_Queue* waitQueue)
{
    struct Kernel_Thread *kthread = waitQueue->head, *next;
    KASSERT(!Interrupts_Enabled());
    while (kthread != 0) {
        next = Get_Next_In_Thread_Queue(kthread);
        Make_Runnable(kthread);
        kthread = next;
    } //遍历等待队列，将队列中的进程放入准备运行队列
    Clear_Thread_Queue(waitQueue);
}
```

5.4.17 Wake_Up_One 函数

函数功能：唤醒一个进程。

函数参数：等待队列。

返回值：无。

```
void Wake_Up_One(struct Thread_Queue* waitQueue)
{
    struct Kernel_Thread* best;
    KASSERT(!Interrupts_Enabled());
    best = Find_Best(waitQueue); //按调度算法查找优先级最高的进程
    if (best != 0) {
        Remove_Thread(waitQueue, best);
        Make_Runnable(best); //将进程从等待队列移出，并放入准备运行队列
    }
}
```

5.4.18 Dump_All_Thread_List 函数

函数功能：打印出系统中所有进程。

函数参数：无。

返回值：无。

```
void Dump_All_Thread_List(void)
{
    struct Kernel_Thread *kthread;
    int count = 0;
    bool iflag = Begin_Int_Atomic();
    kthread = Get_Front_Of_All_Thread_List(&s_allThreadList);
    Print("[");
    while (kthread != 0) {
        ++count;
        Print("<%lx,%lx,%lx>",
              (ulong_t) Get_Prev_In_All_Thread_List(kthread),
              (ulong_t) kthread,
              (ulong_t) Get_Next_In_All_Thread_List(kthread));
        KASSERT(kthread != Get_Next_In_All_Thread_List(kthread));
        kthread = Get_Next_In_All_Thread_List(kthread);
    }
    Print("]\n");
    Print("%d threads are running\n", count);
    End_Int_Atomic(iflag);
}
```



为了实现 GeekOS 内核对内存的分页管理设计技术，用户需要了解 Intel X86 处理器内存分页管理的工作原理，了解其寻址的机制。分页管理的目的是将某一线性地址映射到实际物理内存的某一地址，最终实现虚拟存储技术。在分析本章的内存管理程序时，要明确区分清楚给定的地址是指线性地址还是实际物理内存的地址。

Intel 80386 开始支持存储器分页管理机制。分页机制是存储器管理机制的一部分。段管理机制实现虚拟地址（由段和偏移构成的逻辑地址）到线性地址的转换，而分页管理机制是实现线性地址到物理地址的转换。如果不启用分页管理机制，那么线性地址就是物理地址。下面介绍 80386 的存储器分页管理机制。

6.1 存储器分页管理机制

在第 4 章已经介绍过，控制寄存器 CR0 中的最高位 PG 位用来控制分页管理机制是否启用。如果 PG=1，系统启用分页机制，把线性地址转换为物理地址。如果 PG=0，分页机制无效，线性地址就直接作为物理地址。必须注意，只有在保护方式下分页机制才可能生效。即只有在保证使 PE 位为 1 的前提下，才能够使 PG 位为 1，否则将引起通用保护故障。

分页机制把线性地址空间和物理地址空间分别划分为大小相同的块，这样的块称之为页。通过在线性地址空间的页与物理地址空间的页之间建立的映射，分页机制实现线性地址到物理地址的转换。线性地址空间的页与物理地址空间的页之间的映射可根据需要而改变。线性地址空间的任何一页，可以映射为物理地址空间中的任何一页。

采用分页管理机制实现线性地址到物理地址转换映射的主要目的是便于实现虚拟存储器。不像段的大小是可变的，页的大小是相等并固定的。系统根据程序的逻辑划分段，根据实现虚拟存储器的方便来划分页。

在 80386 中，页的大小一般为 4KB，每一页的边界地址必须是

4K 的倍数，这在编写程序时一定要注意。因此，4G 大小的地址空间被划分为 1M 个页，页的开始地址具有 XXXXX000H 的形式。为此，把页地址的左边高 20 位 XXXXXH 称为页码。线性地址空间页的页码也就是虚拟线性地址的高 20 位。可见，页码左移 12 位就是页的开始地址，所以页码规定了页的位置。

由于页的大小固定为 4K，且页的边界是 4K 的倍数，所以在把 32 位线性地址转换成 32 位物理地址的过程中，右边低 12 位地址保持不变。也就是说，线性地址的低 12 位就是物理地址的低 12 位。假设分页机制采用的转换映射把线性地址空间的 XXXXXH 页映射到物理地址空间的 YYYYH 页，那么线性地址 XXXXXxxxH 被转换为 YYYYYxxxH。因此，线性地址到物理地址的转换要解决的是线性地址空间的页到物理地址空间的页的映射，也就是线性地址高 20 位到物理地址高 20 位的转换。

6.2 线性地址到物理地址的转换

6.2.1 映射表结构

线性地址空间的页到物理地址空间的页之间的映射用表来描述。由于 4GB 的地址空间划分为 1M 个页，因此，如果用一张表来描述这种映射，那么该映射表就要有 1M 个表项，若每个表项占用 4 个字节，那么该映射表就需要占用 4MB。为避免映射表占用如此巨大的存储器资源，所以 80386 把页映射表分为两级。

页映射表的第一级称为页目录表，存储在一个 4KB 的物理页中。页目录表共有 1K 个表项，其中，每个表项为 4 个字节，包含对应第二级表所在物理地址空间的地址。页映射表的第二级称为页表，每张页表也安排在一个 4KB 的物理页中。每张页表都有 1K 个表项，每个表项为 4 字节长，包含对应物理地址空间页的页码。由于页目录表和页表均由 1K 个表项组成，所以使用 10 个二进制位的索引就能指定表项，即用 10 位的索引值乘以 4 加上页表基址就得到了表项的物理地址。

图 6-1 显示了由页目录表和页表构成的页映射表结构。从图中可见，控制寄存器 CR3 指定页目录表；页目录表可以指定 1K 个页表，这些页表可以分散存放在任意的物理页中，而不需要连续存放；每张页表可以指定 1K 个物理地址空间的页，这些物理地址空间的页可以任意地分散在物理地址空间中。需要注意的是，存储页目录表和页表的地址是对齐在 4KB 边界上的。

6.2.2 表项格式

页目录表和页表中的表项都采用如图 6-2 所示的格式。从图中可见，最高 20 位（位 12~位 31）包含物理地址空间页的页码，也就是物理地址的高 20 位。低 12 位包含页的属性。如图 6-2 所示的属性中内容为 0 的位是 Intel 公司为 80486 等处理器所保留的位，在为 80386 编程使用到它们时必须设置为 0。在位 9~位 11 的 AVL 字段供软件使用。表项的最低位是存在属性位，记作 P。P 位表示该表项是否有效：P=0 表项无效；P=1 表

项有效，此时表项中的其余各位均可供软件使用。80386 不解释 P=0 的表项中的任何其他的位。在通过页目录表和页表进行的线性地址到物理地址的转换过程中，无论在页目录表还是在页表中遇到无效表项，都会引起页故障。其他属性位的作用将在下面介绍。

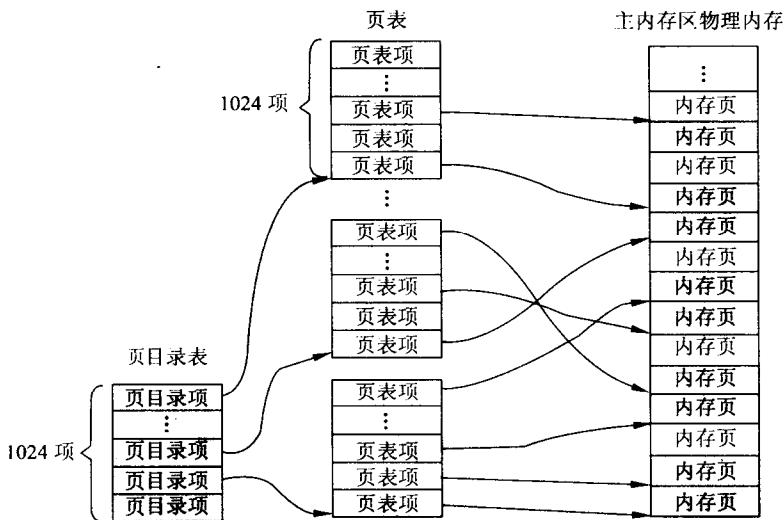


图 6-1 二级页表结构

31	12	11	0
页框地址 位 31..12	可用	0 0 D A 0 0 U / S R / W P	

图 6-2 页目录项和页表项格式

6.2.3 线性地址到物理地址的转换

分页管理机制通过上述页目录表和页表实现 32 位线性地址到 32 位物理地址的转换。控制寄存器 CR3 的高 20 位作为页目录表所在物理页的页码。首先把线性地址的最高 10 位（即位 22~位 31）作为页目录表的索引，对应表项所包含的页码指定页表；然后，再把线性地址的中间 10 位（即位 12~位 21）作为所指定的页目录表中的页表项的索引，对应表项所包含的页码指定物理地址空间中的一页；最后，把所指定的物理页的页码作为高 20 位，把线性地址的低 12 位不加改变地作为 32 位物理地址的低 12 位。如图 6-3 所示，形象地表示出了一个给定的线性地址是如何映射到物理内存页上的。

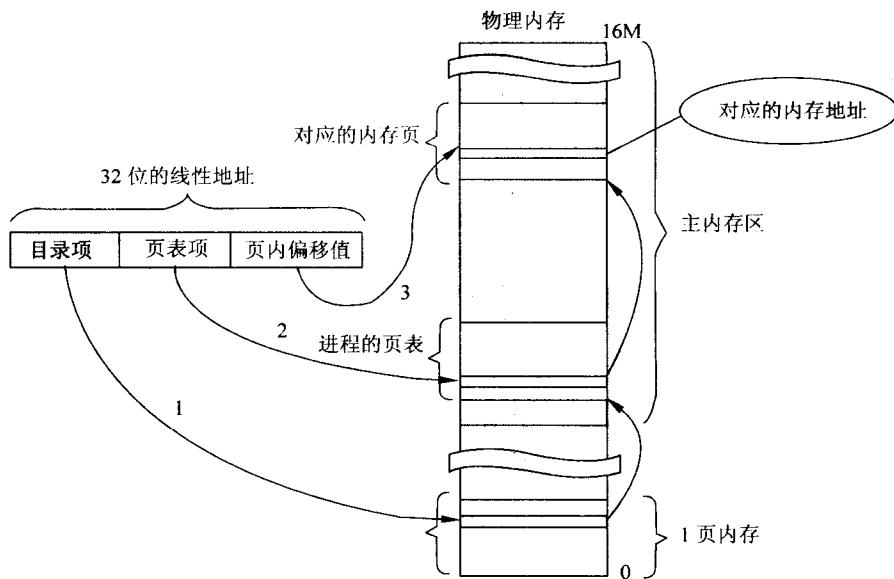


图 6-3 线性地址到物理地址的转换过程

为了避免在每次存储器访问时都要访问内存中的页表，以便提高访问内存的速度，80386 处理器的硬件把最近使用的线性——物理地址转换的映射关系存储在处理器内部的页转换高速缓存中。在访问存储器页表之前总是先查找高速缓存，仅当必须的转换不在高速缓存中时，才访问存储器中的二级页表。页转换高速缓存也称为页转换查找缓存，记为 TLB。

在分页机制转换高速缓存中的数据与页表中数据的相关性，不是由 80386 处理器进行维护的，而必须由操作系统软件保存，也就是说，处理器不知道软件什么时候会修改页表，在一个合理的系统中，页表只能由操作系统修改，操作系统可以直接地在软件修改页表后通过刷新高速缓存来保证相关性。高速缓存的刷新通过装入处理器控制寄存器 CR3 完成。

6.2.4 不存在的页表

采用上述页映射表结构，存储全部 1K 张页表需要 4MB，此外还需要 4KB 用于存储页目录表。这样的两级页映射表似乎反而比单一的整张页映射表多占用 4KB。其实不然，事实上不需要在内存中存储完整的两级页映射表。两级页映射表结构中对于线性地址空间中不存在的或未使用的部分不必分配页表。除必须给页目录表分配物理页外，仅当在需要时才给页表分配物理页，于是页映射表的大小就对应于实际使用的线性地址空间大小。因为任何一个实际运行的程序使用的线性地址空间都远小于 4GB，所以用于分配给页表的物理页也远小于 4MB。

页目录表项中的存在位 P 表明对应页表是否有效。如果 P=1，表明对应页表有效，

可利用它进行地址转换；如果 $P=0$ ，表明对应页表无效。如果试图通过无效的页表进行线性地址到物理地址的转换，那么将引起页故障。因此，页目录表项中的属性位 P 使得操作系统只需给覆盖实际使用的线性地址范围的页表分配物理页。

页目录表项中的属性位 P 可用于把页表存储在虚拟存储器中。当发生由于所需页表无效而引起的页故障时，页故障处理程序再申请物理页，从磁盘上把对应的页表读入，并把对应页目录表项中的 P 位置 1。换言之，可以当需要时才为所需要的页表分配物理页。这样页表占用的物理页数量可降到最小。

6.2.5 页的共享

由上述页映射表结构可见，分页机制没有全局页和局部页的规定。每一个任务可使用自己的页映射表独立地实现线性地址到物理地址的转换。但是，如果让每一个任务所用的页映射表具有部分相同的映射，也可以实现部分页的共享。

常用的实现页共享的方法是线性地址空间的共享，也就是不同任务的部分相同的线性地址空间的映射信息相同，具体表现为部分页表相同或页表内的部分表项的页码相同。例如，如果任务 A 和任务 B 分别使用的页目录表 A 和页目录表 B 内的第 0 项中的页码相同，也就是页表 0 相同，那么任务 A 和任务 B 的 00000000H 至 003FFFFH 线性地址空间就映射到相同的物理页。再如，任务 A 和任务 B 使用的页表 0 不同，但这两张页表内第 0 至第 0FFH 项的页码对应相同，那么任务 A 和任务 B 的 00000000H 至 000FFFFH 线性地址空间就映射到相同的物理页。

需要注意的是，共享的页表最好由两个页目录中同样的目录项所指定。这一点很重要，因为它保证了在两个任务中同样的线性地址范围将映射到该全局区域。

6.3 页级保护和虚拟存储器支持

6.3.1 页级保护

80386 不仅提供段级保护，也提供页级保护。分页机制只区分两种特权级。特权级 0、1 和 2 统称为系统特权级，特权级 3 称为用户特权级。在如图 6-2 所示的页目录表和页表的表项中的保护属性位 R/W 和 U/S 就是用于对页进行保护。

表项的位 1 是读写属性位，记作 R/W。R/W 位指示该表项所指定的页是否可读、写或执行。若 $R/W=1$ ，对表项所指定的页可进行读、写或执行；若 $R/W=0$ ，对表项所指定的页可读或执行，但不能对该指定的页写入。但是，R/W 位对页的写保护只在处理器处于用户特权级时发挥作用，当处理器处于系统特权级时，R/W 位被忽略，即系统拥有读、写和执行的全部权限。

表项的位 2 是用户/系统属性位，记作 U/S。U/S 位指示该表项所指定的页是否是用户级页。若 $U/S=1$ ，表项所指定的页是用户级页，可由任何特权级下执行的程序访问；

如果 $U/S=0$, 表项所指定的页是系统级页, 只能由系统特权级下执行的程序访问。如表 6-1 所示, 列出了上述属性位 R/W 和 U/S 所确定的页级保护下, 用户级程序和系统级程序分别具有的对用户级页和系统级页进行操作的权限。

由表 6-1 可见, 用户级页可以规定为只允许读/执行或规定为读/写/执行。系统级页对于系统级程序总是可读/写/执行, 而对用户级程序总是不可访问的。和分段机制一样, 外层用户级执行的程序只能访问用户级的页, 而内层系统级执行的程序, 既可访问系统级页, 也可访问用户级页。与分段机制不同的是, 在内层系统级执行的程序, 对任何页都有读/写/执行访问权, 即使规定为只允许读/执行的用户页, 内层系统级程序也对该页有写访问权。

表 6-1 页操作权限表

	U/S	R/W	用户级访问权限	系统级访问权限
页级 保护	0	0	无	读/写/执行
属性	0	1	无	读/写/执行
	1	0	读/执行	读/写/执行
	1	1	读/写/执行	读/写/执行

页目录表项中的保护属性位 R/W 和 U/S 对由该表项指定页表所指定的全部 1K 个页起到保护作用。所以, 对页访问时引用的保护属性位 R/W 和 U/S 的值是组合计算页目录表项和页表项中的保护属性位的值所得。

正如在 80386 地址转换机制中分页机制在分段机制之后起作用一样, 由分页机制支持的页级保护也在由分段机制支持的段级保护之后起作用。先测试有关的段级保护, 如果启用分页机制, 那么在检查通过后, 再测试页级保护。如果段的类型为读/写, 而页规定为只允许读/执行, 那么不允许写; 如果段的类型为只读/执行, 那么不论页保护如何, 也不允许写。

页级保护的检查是在线性地址转换为物理地址的过程中进行的, 如果违反页保护属性的规定, 对页进行访问(读/写/执行), 那么将引起页故障。

6.3.2 虚拟存储器技术

页表项中的 P 位是支持采用分页机制虚拟存储器的关键。P=1, 表示表项指定的页存在于物理存储器中, 并且表项的高 20 位是物理页的页码; P=0, 表示该线性地址空间中的页所对应的物理地址空中的页不在物理存储器中。如果程序访问不存在的页, 会引起页故障, 这样操作系统可把该不存在的页从磁盘上读入, 把所在物理页的页码填入对应表项并把表项中的 P 位置为 1, 然后让引起页故障的程序恢复运行。

此外, 表项中的访问位 A 和写标志位 D 也用于支持虚拟存储器的实现。

表项的位 5 是访问属性位, 记作 A。在为了访问某存储单元而进行线性地址到物理地址的转换过程中, 处理器总是把页目录表内的对应表项和其所指定页表内的对应表项

中的 A 位置 1，除非页表或页不存在，或者访问违反保护属性规定。所以，A=1 表示已访问过对应的物理页。处理器永不清除 A 位。但操作系统可以通过周期性地检测及清除 A 位，来确定哪些页在最近一段时间未被访问过。当存储器资源紧缺时，这些最近未被访问的页很可能就被选择出来，将它们从内存换出到磁盘上去。

表项的位 6 是写标志位，记作 D。在为了访问某存储单元而进行线性地址到物理地址的转换过程中，如果是写访问并且可以写访问，处理器就把页表内对应表项中的 D 位置 1，但并不把页目录表内对应表项中的 D 置 1。当某页从磁盘上读入内存时，页表中对应表项的 D 位被清 0。所以，D=1 表示已写过对应的物理页。当某页需要从内存换出到磁盘上时，如果该页的 D 位为 1，那么必须进行写操作（把内存中的页写入磁盘时，处理器并不清除对应页表项的 D 位，这必须由操作系统修改）。但是，如果要写到磁盘上的页的 D 位为 0（因为内存中的页与磁盘中的页具有完全相同的内容），就不需要实际的磁盘写操作，而只要简单地放弃内存中该页即可。

6.4 页故障

启用分页机制后，线性地址不再直接等于物理地址，线性地址要经过分页机制转换才成为物理地址。在转换过程中，如果出现下列情况之一就会引起页故障。

- (1) 涉及的页目录表内的表项或页表内的表项中的 P=0，即涉及页不在内存。
- (2) 发现试图违反页保护属性的规定而对页进行访问。

报告页故障的中断向量号是 14 (0EH)。在进入故障处理程序时，保存的指令指针 CS 及 EIP 指向发生故障的指令。一旦引起页故障的原因被排除后，即可从页故障处理程序通过一条 IRET 指令，直接地重新执行产生故障的指令。

当页故障发生时，处理器把引起页故障的线性地址装入 CR2。页故障处理程序可以利用该线性地址确定对应的页目录项和页表项。

页故障还在堆栈中提供一个出错码，出错码的格式如表 6-2 所示。其中，U 位表示引起故障程序的特权级，U=1 表示用户特权级（特权级 3），U=0 表示系统特权级（特权级 0、1 或 2）；W 位表示访问类型，W=0 表示读/执行，W=1 表示写；P 位表示异常类型，P=0 表示页不存在故障，P=1 表示保护故障。页故障的响应处理模式同其他故障一样。

表 6-2 出错码格式表

出错码 的格式	BIT15~BIT3	BIT2	BIT1	BIT0
	未使用	U	W	P

至此，已经介绍了 Intel 80x86 体系结构分页管理基本原理。GeekOS 操作系统的分页技术的具体实现将在第 12 章详细介绍，下面的内容是 GeekOS 系统有关分页实现的数据结构和一些函数功能的分析。

6.5 GeekOS 分页系统数据结构

6.5.1 页目录表和页表项数据结构

根据 Intel 80x86 体系结构设计原理，在文件 paging.h 文件定义了页目录项 pde_t 和页表项 pte_t 的数据结构，具体如下。

```
/* 页目录项数据类型,每一个表项用来指定一个页表的基地址和访问权限 */
typedef struct {
    uint_t present:1;                                //标识页表是否在内存中存在
    uint_t flags:4;                                  //标识页表的状态,即访问权限
    uint_t accesed:1;                               //标识页表是否被访问过
    uint_t reserved:1;                             //在此作为保留位
    uint_t largePages:1;                            //标识页表的大小
    uint_t globalPage:1;                           //标识页表是否为全局页表
    uint_t kernelInfo:3;                          //标识页表是否为内核页表
    uint_t pageTableBaseAddr:20;                  //标识页表在内存中的地址
} pde_t;

/* 页表项数据类型,每一个表项用来指定存储空间的一个页的物理地址和访问权限*/
typedef struct {
    uint_t present:1;                                //标识页是否在内存中存在
    uint_t flags:4;                                  //标识页的状态,即访问权限
    uint_t accesed:1;                               //标识页是否被访问过
    uint_t dirty:1;                                 //标识页是否被修改过
    uint_t pteAttribute:1;                         //标识页的大小
    uint_t globalPage:1;                           //标识页是否为全局页
    uint_t kernelInfo:3;                          //标识页是否为内核页
    uint_t pageBaseAddr:20;                      //标识页在内存中的物理页号
} pte_t;

/* pde_t和 pte_t中flags域的取值数据定义 */
#define VM_WRITE 1      /* 此页可以修改 */
#define VM_USER 2      /* 此页可以被用户程序访问 */
#define VM_NOCACHE 8   /* 此页没有缓冲区 */
#define VM_READ 0       /* 此页可以被读(ignored for x86) */
#define VM_EXEC 0       /* 此页可以被执行 (ignored for x86) */
```

6.5.2 物理页数据结构和页状态

GeekOS 分页系统将内存划分为以页为单位的一个大数组，其中每一页为 4KB，每一页都有一个结构 Page 加以控制，具体定义如下（在 mem.h 文件中）。

```

/* 物理内存的每一页都由以下一个数据结构来描述 */
struct Page {
    unsigned flags;           /* 标识页的状态信息 */
    DEFINE_LINK(Page_List, Page); /* Page_List页链表的链接指针 */
    int clock;                /* 页最近一次的访问时间 */
    ulong_t vaddr;            /* 页映射到的用户空间虚拟地址 */
    pte_t *entry;              /* 指向本页的页表项的入口 */
};

/* 物理内存页的状态数据定义 */
#define PAGE_AVAIL     0x0000 /* 页在空闲链表中 */
#define PAGE_KERN      0x0001 /* 内核代码和数据占用的页 */
#define PAGE_HW        0x0002 /* 被硬件设备使用的页 */
#define PAGE_ALLOCATED 0x0004 /* 页已经被分配 */
#define PAGE_UNUSED    0x0008 /* 页没有使用 */
#define PAGE_HEAP      0x0010 /* 内核堆的页 */
#define PAGE_PAGEABLE  0x0020 /* 被换出到page file的页 */
#define PAGE_LOCKED    0x0040 /* 不能被释放的页 */

```

其中页数据结构的 `DEFINE_LINK` 是位于 `list.h` 中的宏定义，定义了指向链表节点的指针。GeekOS 使用宏定义定义了链表结构及操作，具体代码如下（在文件 `list.h` 中）。

```
#define DEFINE_LINK(listTypeName, nodeTypeName) \
    struct nodeTypeName * prev##listTypeName, * next##listTypeName
```

系统全局页链表 `g_pageList` 从头到尾包含了内存所有物理页的 `Page` 结构，其中 `flags` 标记为 `PAGE_AVAIL` 的页为空闲页，`s_freeList` 则维护系统所有的空闲页。

6.6 GeekOS 分页系统主要操作函数

GeekOS 分页系统的主要操作函数的定义和提示是在目录 `/src/geekos` 中的文件 `mem.c`、`paging.c` 和 `uservm.c`，下面将简单介绍这些已经实现函数的主要功能。

6.6.1 Alloc_Page 函数

函数功能：为内核进程分配一个物理页。

函数参数：无。

函数返回值：若分配成功，返回分配页的地址，否则返回 0。

```
void * Alloc_Page(void)
{
    struct Page* page;
    void *result = 0;
```

```

bool iflag = Begin_Int_Atomic();
/* 首先在空闲页链表中判断是否有空闲的物理页 */
if (!Is_Page_List_Empty(&s_freeList)) { /* 有空闲页处理 */
    /* 把空闲页链表指定的第一页分配给进程 */
    page = Get_Front_Of_Page_List(&s_freeList);
    KASSERT((page->flags & PAGE_ALLOCATED) == 0);
    Remove_From_Front_Of_Page_List(&s_freeList);
    /* 在总页表中标识刚分配页的状态为已分配 */
    page->flags |= PAGE_ALLOCATED;
    g_freePageCount--;
    result = (void*) Get_Page_Address(page); /* 返回刚分配页的地址 */
}
End_Int_Atomic(iflag);
return result;
}

```

6.6.2 Alloc_Pageable_Page 函数

函数功能：为用户进程地址空间分配一个物理页。

函数参数：entry 是指向刚分配页对应的用户页表项，vaddr 虚拟地址所对应的页将映射在用户地址空间中。

函数返回值：若分配成功，返回分配页的地址，否则返回 0。

```

void * Alloc_Pageable_Page(pte_t *entry, ulong_t vaddr)
{
    bool iflag;
    void* paddr = 0;
    struct Page* page = 0;
    iflag = Begin_Int_Atomic(); //屏蔽中断
    KASSERT(!Interrupts_Enabled());
    KASSERT(Is_Page_Multiple(vaddr));
    paddr = Alloc_Page(); //分配一个空闲页
    if (paddr != 0) { //系统有空闲页的处理
        page = Get_Page((ulong_t) paddr);
        KASSERT((page->flags & PAGE_PAGEABLE) == 0);
    }
    else { //系统无空闲页的处理
        int pagefileIndex;
        /* 从另一个进程选取一个页换出到page file保存 */
        Debug("About to hunt for a page to page out\n");
        page = Find_Page_To_Page_Out(); //选取被换出页
        KASSERT(page->flags & PAGE_PAGEABLE);
        paddr = (void*) Get_Page_Address(page);
    }
}

```

```

Debug("Selected page at addr %p (age = %d)\n", paddr, page->clock);
    /* 在page file中为换出页找到空闲磁盘空间 */
pagefileIndex = Find_Space_On_Paging_File();
if (pagefileIndex < 0)           //在paging file无空闲空间函数提前结束
    goto done;
Debug("Free disk page at index %d\n", pagefileIndex);
/* 标识此页不能再被其他用户进程换出 */
page->flags &= ~(PAGE_PAGEABLE);
/* 在写此页到磁盘时要锁定此页,免得可能会被系统释放*/
    page->flags |= PAGE_LOCKED;
/* 写此页到磁盘,系统应该响应中断,因为I/O操作可能被阻塞 */
Debug("Writing physical frame %p to paging file at %d\n", paddr,
pagefileIndex);
Enable_Interrupts();
Write_To_Paging_File(paddr, page->vaddr, pagefileIndex);
Disable_Interrupts();
if (page->flags & PAGE_ALLOCATED)
{
    /* 换出的页仍然在使用,在页表中修改相应页的信息 */
    page->entry->present = 0;
    page->entry->kernelInfo = KINFO_PAGE_ON_DISK;
    page->entry->pageBaseAddr = pagefileIndex; // 记录在page file的地址
}
else
{
    /* 换出的页是不再有用,释放此页所占用的磁盘空间 */
    Free_Space_On_Paging_File(pagefileIndex);
    /* 但此页仍标识为已分配状态 */
    page->flags |= PAGE_ALLOCATED;
}
/* 此页为非PAGE_LOCKED状态 */
page->flags &= ~(PAGE_LOCKED);
/* 由于修改了页表内容,需要刷新TLB中的内容 */
Flush_TLB();
}
/* 填写此页对应结构的描述信息 */
page->flags |= PAGE_PAGEABLE;
page->entry = entry;
page->entry->kernelInfo = 0;
page->vaddr = vaddr;
KASSERT(page->flags & PAGE_ALLOCATED);
done:                                //无空闲磁盘空间处理
    End_Int_Atomic(iflag);        //使得系统相应中断
    return paddr;
}

```

6.6.3 Find_Page_To_Page_Out 函数

函数功能：从内存物理页中选取一个换出页。

函数参数：无。

函数返回值：若选取成功，返回选取页，否则返回 0。

```
static struct Page *Find_Page_To_Page_Out()
{
    int i;
    struct Page *curr, *best;
    best = NULL;
    for (i=0; i < s_numPages; i++) {           //扫描页总链表g_pageList
        if ((g_pageList[i].flags & PAGE_PAGEABLE) &&
            (g_pageList[i].flags & PAGE_ALLOCATED)) {
            if (!best) best = &g_pageList[i];
            curr = &g_pageList[i];
            if ((curr->clock < best->clock) && (curr->flags & PAGE_PAGEABLE))
            {
                best = curr;
            }
        }
    }
    return best;           //best指向要换出的页
}
```

6.6.4 Free_Page 函数

函数功能：从内存物理页中释放一个页。

函数参数：释放页的基地址。

函数返回值：无。

```
void Free_Page(void* pageAddr)
{
    ulong_t addr = (ulong_t) pageAddr;
    struct Page* page;
    bool iflag;
    iflag = Begin_Int_Atomic();           //屏蔽中断
    KASSERT(Is_Page_Multiple(addr));    //保证addr为页大小的倍数
    /* 得到释放页对应的page数据结构 */
    page = Get_Page(addr);
    KASSERT((page->flags & PAGE_ALLOCATED) != 0);
    /* 清除页的已分配状态信息 */
    page->flags &= ~(PAGE_ALLOCATED);
    /* 当此页正被其他进程使用时，不能释放它，只能标识本进程不需要 */
    if (page->flags & PAGE_LOCKED)
        return;
    /* 清除页的PAGE_PAGEABLE状态信息 */
    page->flags &= ~(PAGE_PAGEABLE);
```

```

/* 把此页加到空闲链表freelist中 */
Add_To_Back_Of_Page_List(&s_freeList, page);
g_freePageCount++;
End_Int_Atomic(iflag);      //使系统能响应中断
}

```

6.6.5 Page_Fault_Handler 函数

函数功能：缺页中断处理程序。

函数参数：缺页中断产生时的 state 数据结构。

函数返回值：无。

```

/* 用户应该调用Install_Interrupt_Handler() 函数注册此函数为14号中断 */
void Page_Fault_Handler(struct Interrupt_State* state)
{
    ulong_t address;
    faultcode_t faultCode;
    KASSERT(!Interrupts_Enabled());
    /* 得到引起缺页中断的地址 */
    address = Get_Page_Fault_Address();
    Debug("Page fault @%lx\n", address);
    /* 得到缺页错误代码,即哪一类中断*/
    faultCode = *((faultcode_t *) &(state->errorCode));
    /* 其余的缺页处理,现在仅仅是打印出错的情况信息,可以添加代码*/
    Print ("Unexpected Page Fault received\n");
    Print_Fault_Info(address, faultCode);
    Dump_Interrupt_State(state);
    /* 用户地址非法访问系统仅仅杀死此进程 */
    if (!faultCode.userModeFault) KASSERT(0);
    /* 系统现在实现的也是仅仅杀死引起缺页的进程 */
    Exit(-1);
}

```

6.6.6 Print_Fault_Info 函数

函数功能：显示引起缺页中断的信息。

函数参数：引起缺页中断产生的地址，缺页中断的分类代码。

函数返回值：无。

```

static void Print_Fault_Info(uint_t address, faultcode_t faultCode)
{
    extern uint_t g_freePageCount;
    Print("Pid %d, Page Fault received, at address %x (%d pages free)\n",

```

```
    g_currentThread->pid, address, g_freePageCount);
if (faultCode.protectionViolation)
    Print (" Protection Violation, "); //非法访问受保护的内存单元
else
    Print (" Non-present page, ");
if (faultCode.writeFault)
    Print ("Write Fault, "); //非法修改只读内存单元
else
    Print ("Read Fault, ");
if (faultCode.userModeFault)
    Print ("in User Mode\n"); //非法访问内核代码
else
    Print ("in Supervisor Mode\n");
}
```

7.1 GeekOS 文件系统框架

文件系统是操作系统中负责存取和管理文件信息的模块，现代操作系统都配备了文件系统以适应系统管理和用户使用软件资源的需要，文件系统可以为用户提供方便有效的文件使用和操作方法。

GeekOS 初始内核只提供一个只读文件系统 PFAT，但 GeekOS 有一个虚拟文件系统层（Virtual File System），在虚拟文件系统层上，允许 GeekOS 实现多个文件系统共存，所以用户可以编程扩充文件系统类型，如在设计项目 5 中，要求用户编程实现一个文件系统 GOSFS。

GeekOS 的文件系统的层次如图 7-1 所示，用户进程通过调用 C 语言库函数进行文件打开、读、写等操作，库函数通过相应的系统调用将文件操作请求传递到虚拟文件系统层，虚拟文件系统层将文件操作分配给相应的具体文件系统函数来执行。

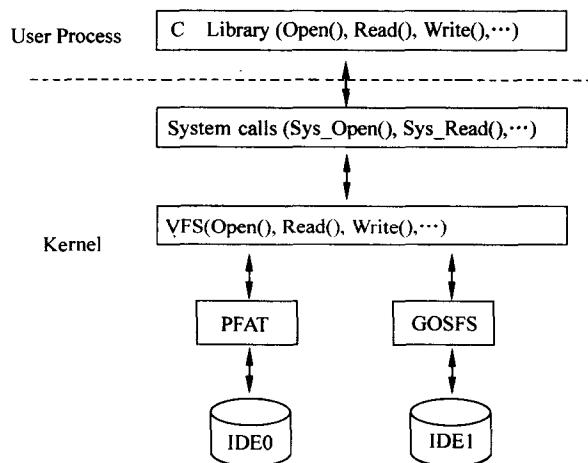


图 7-1 GeekOS 文件系统框架

7.2 虚拟文件系统层

GeekOS 的虚拟文件系统（Virtual File System）是源自 Linux VFS 的一种抽象机制，它将各种文件系统的基本操作抽象出来，并组织在一起形成一个系统调用与实际文件系统间的中间层，具体见图 7-1。在虚拟文件系统层的上层是文件系统的系统调用，下层是具体的文件系统，它负责将系统调用要求的文件操作（如打开、读、写等）映射到相应的文件系统接口。VFS 的实现使一个操作系统使用多种文件系统成为可能。

下面我们以用户进程打开并读 GOSFS 文件系统的文件为例，说明读文件的工作过程，通过这个过程可以更好地理解 VFS 的工作原理（如图 7-2 所示）。

(1) 首先用户进程调用 C 语言库函数 Read(), 引发一个软件中断通知内核有系统调用发生。（Read 函数有 3 个参数，需要打开的文件描述符、从文件中读出的数据保存使用的缓冲区、从文件中读出数据的字节数）。

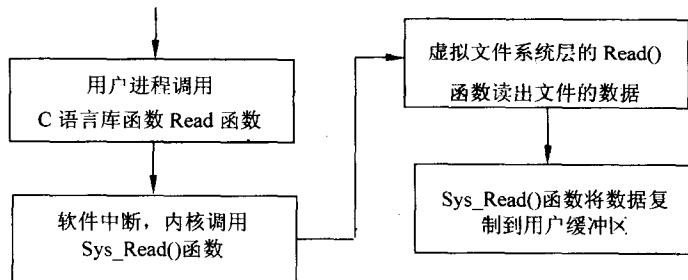


图 7-2 GOSFS 读文件过程

(2) 内核调用 Sys_Read() 函数，Sys_Read() 函数首先在该进程的打开文件列表中查找该文件对象，同时为该进程分配临时的内核缓冲区，用于存放从文件中读出的数据，然后调用内核 VFS 函数 Read(); (GOSFS 中，定义了每个用户进程有一个打开文件表 open file table，用于保存文件当前可以读写的文件列表，任意用户进程最多可以打开 10 个文件，在<geekos/user.h>中用常量 USER_MAX_FILES 宏定义)。

(3) VFS 函数 Read() 首先在文件的虚拟函数表中找到 Read 函数的地址，因为文件是 GOSFS 文件系统的一部分，因此这个函数指向的是 GOSFS_Read()。

(4) GOSFS_Read() 从文件的当前位置开始读取需要的数据，并将数据复制到 Sys_Read() 函数分配的缓冲区。

(5) 控制返回到 Sys_Read()，它调用 Copy_To_User() 函数将数据从内核缓冲区复制到用户缓冲区，并释放分配得到的内核缓冲区。

(6) 最后 Sys_Read() 返回，系统调用结束，用户进程继续运行。

7.3 高速缓冲区

在文件系统实现过程中，需要频繁地从文件系统读取数据到内存，有时也需要将数

据写回文件系统，为加快磁盘操作的运行效率，GeekOS 为文件系统提供了高速缓冲区。高速缓冲区位于文件系统与块设备服务模块之间，如图 7-3 所示。

高速缓冲区（代码在 `bufcache.h`, `bufcache.c` 中实现）是专门用于保存磁盘块数据的内存区，从这个角度上看高速缓冲区是一个虚拟磁盘。文件系统在进行磁盘操作时，首先检查所需磁盘块是否已经在高速缓冲区中，若已在高速缓冲区，就直接进行内存上的块操作，否则就由高速缓冲区的管理机制向块设备提出磁盘访问请求，读入所需磁盘块。在操作完毕后，高速缓冲区可以依据情况将内存中的缓冲磁盘块立即写回磁盘以保证内存块与磁盘块的一致，也可以考虑稍后系统空闲时写入磁盘以保证系统运行效率。GeekOS 的缓冲块默认大小是 4KB，正好与一个物理页面大小相等。根据需要，开发人员可以调整缓冲块的大小，使之与磁盘块成比例，这样便于进行磁盘块操作。

高速缓冲区结构定义为 `FS_Buffer` 结构，缓冲区的管理实行动态的分配回收制。当文件系统需要一块新缓冲区时，就构造一个 `FS_Buffer` 结构，并分配一页内存，之后将这页物理内存的起始地址赋给 `FS_Buffer` 的 `data` 字段，并初始化其他字段，然后根据磁盘块号将磁盘中的数据导入物理内存，最后将 `FS_Buffer` 结构指针插入 `FS_Buffer_List` 链表的头部，并将这个指针返回给文件系统。

```
// FS_Buffer 结构
struct FS_Buffer {
    ulong_t fsBlockNum; // 磁盘的块号
    void *data; // 内存中的磁盘块
    uint_t flags; // 块状态
    DEFINE_LINK(FS_Buffer_List, FS_Buffer); // 指针指向缓冲队列下一项
};
```

缓冲区 `FS_Buffer_List` 链表初始状态是没有任何缓冲块的，随着文件系统操作的开始，缓冲区会变得越来越大，但由于系统空闲内存有限，缓冲区不可能无限地增大，所以 GeekOS 的高速缓冲区有默认的数量限制（在 `bufcache.c` 中，用常量 `FS_BUFFER_CACHE_MAX_BLOCKS` 表示），开发人员可以根据系统运行的具体环境进行调整。一旦缓冲区块数已经达到最大，就要再利用系统中已经分配的缓冲块。

系统不会立即释放已经用完的缓冲区块占有的内存，而是将其状态变为空闲，以便后面的文件操作使用。缓冲区队列管理机制的维护采用 LRU 算法，即每次在申请缓冲区块时，从链表头开始检测每个缓冲块，若不是目标块且未使用，就成为一个候选块，算法执行直到找到目标块为止或整个链表都搜索结束都未找到目标块为止。由于总是从表头到表尾的顺序搜索，最后得到的候选块就是最久未使用的缓冲块。为重新使用这个内存块，系统首先要将得到的这个缓冲块的数据写回磁盘，重新初始化该块，并插入链表头，这样缓冲区块就可以重新使用了。若无法找到合适的候选块，即缓冲区无法提供空闲块，此时文件系统的操作请求就会失败，因此设置合理的缓冲区大小是很重要的。

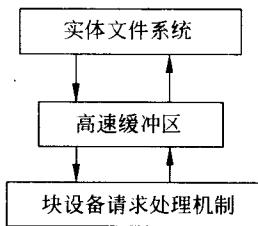


图 7-3 高速缓冲区层次图

7.4 PFAT 文件系统

PFAT (Pseudo File Allocation Table) 文件系统是 GeekOS 初始内核提供的一个只读文件系统，主要用来存放系统项目测试要执行的用户程序。PFAT 是一个单级目录结构的文件系统（实现代码在源文件 pfat.h 和 pfat.c 中）。PFAT 默认加载在硬盘 ide0 上，根目录名为 C。在系统编译时由工具 buildFAT 将所有的用户测试文件都写入这个文件系统的根目录下。由于系统根目录可容纳的目录项有限，因此可容纳的用户测试程序个数会受到限制。

PFAT 文件系统用到的一些重要的数据结构如下。

1. 启动扇区

```
typedef struct {
    int magic;                      /* 标记文件系统的类型 */
    int fileAllocationOffset;        /* 文件分配表的存储位置 */
    int fileAllocationLength;        /* 文件分配表的长度 */
    int rootDirectoryOffset;         /* 根目录的存储位置(在扇区的偏移量) */
    int rootDirectoryCount;          /* 目录中的项目数 */
    short setupStart;                /* 二次加载的第一个扇区 */
    short setupSize;                 /* 二次加载的字节数 */
    short kernelStart;               /* 内核运行的第一个扇区 */
    short kernelSize;
} bootSector;
```

2. 目录结构

```
typedef struct {
    char fileName[8+4]; //文件名
    /* 文件的属性位 */
    char readOnly:1;
    char hidden:1;
    char systemFile:1;
    char volumeLabel:1;
    char directory:1;
    short time;
    short date;
    int firstBlock;      //文件的第一块
    int fileSize;        //文件大小
} directoryEntry;
```

3. PFAT 结构

用于描述加载的 PFAT 文件系统。

```

struct PFAT_InstanceId {
    bootSector fsinfo;
    int *fat;
    directoryEntry *rootDir;
    directoryEntry rootDirEntry;
    struct Mutex lock;
    struct PFAT_File_List fileList;
};

}

```

4. PFAT 文件结构

用于描述系统中的打开文件信息。

```

struct PFAT_File {
    directoryEntry *entry;           /* 文件入口 */
    ulong_t numBlocks;              /* 文件使用的磁盘块 */
    char *fileDataCache;            /* 保存文件数据的缓冲区 */
    struct Bit_Set *validBlockSet;
    /* 用于描述数据块是否有效 */
    struct Mutex lock;
    /* 用于并发存取时候的互斥信号量 */
    DEFINE_LINK(PFAT_File_List, PFAT_File);
};

}

```

7.5 PFAT 文件系统操作函数

7.5.1 Copy_Stat 函数

函数功能：从目录项中复制文件源数据。

函数参数：虚拟文件系统层文件指针，目录指针。

函数返回值：无。

```

static void Copy_Stat(struct VFS_File_Stat *stat, directoryEntry *entry)
{
    stat->size = entry->fileSize;
    stat->isDirectory = entry->directory;
    stat->isSetuid = 0;
    memset(&stat->acls, '\0', sizeof(stat->acls));
    stat->acls[0].uid = 0;
    stat->acls[0].permission = O_READ;
    if (!entry->readOnly)
        stat->acls[0].permission |= O_WRITE;
}

```

7.5.2 PFAT_FStat 函数

函数功能：PFAT 系统的 FStat 函数，调用 Copy_Stat 函数，复制文件源数据。

函数参数：文件指针，虚拟文件系统层文件指针。

函数返回值：复制成功，则返回 0。

```
static int PFAT_FStat(struct File *file, struct VFS_File_Stat *stat)
{
    struct PFAT_File *pfatFile = (struct PFAT_File*) file->fsData;
    Copy_Stat(stat, pfatFile->entry);
    return 0;
}
```

7.5.3 PFAT_Read 函数

函数功能：从 PFAT 文件系统读取文件数据到文件缓冲区。

函数参数：文件指针，缓冲区地址，需要读取文件的字节数。

函数返回值：若读取成功，则返回实际读取的文件字节数。

```
static int PFAT_Read(struct File *file, void *buf, ulong_t numBytes)
{
    struct PFAT_File *pfatFile = (struct PFAT_File*) file->fsData;
    struct PFAT_Instance *instance = (struct PFAT_Instance*)
        file->mountPoint->fsData;
    ulong_t start = file->filePos;
    ulong_t end = file->filePos + numBytes;
    ulong_t startBlock, endBlock, curBlock;
    ulong_t i;
    /*若读的长度大于INT_MAX，则返回无效 */
    if (numBytes > INT_MAX)
        return EINVAL;
    /* 确保需要读取文件的长度是有效的，若要读取的数据位置超出文件范围，则无效*/
    if (start >= file->endPos || end > file->endPos || end < start) {
        Debug("Invalid read position: filePos=%lu, numBytes=%lu,
        endPos=%lu\n",
            file->filePos, numBytes, file->endPos);
        return EINVAL;
    }
    /*检测需要读取的数据是否都在文件缓冲区中*/
    startBlock = (start % SECTOR_SIZE) / SECTOR_SIZE;
    endBlock = Round_Up_To_Block(end) / SECTOR_SIZE;
    /*遍历PFAT文件系统，查找需要读取的文件块 */
}
```

```

curBlock = pfatFile->entry->firstBlock;
for (i = 0; i < endBlock; ++i) {
    /* 检测数据块是否可用 */
    if (curBlock == FAT_ENTRY_FREE || curBlock == FAT_ENTRY_EOF) {
        Print("Unexpected end of file in FAT at file block %lu\n", i);
        return EIO; /* probable filesystem corruption */
    }
    /* 检测读取块号是否有效 */
    if (i >= startBlock) {
        int rc = 0;
        /* 对文件进行互斥锁，确保每次仅有一个进程对文件进行读取 */
        Mutex_Lock(&pfatFile->lock);
        if (!Is_Bit_Set(pfatFile->validBlockSet, i)) {
            /* 读一个磁盘块到文件数据缓冲区 */
            Debug("Reading file block %lu (device block %lu)\n", i, curBlock);
            rc = Block_Read(file->mountPoint->dev, curBlock, pfatFile->
                fileDataCache + i*SECTOR_SIZE);
            if (rc == 0)
                /* 置该块的已读标记 */
                Set_Bit(pfatFile->validBlockSet, i);
        }
        Mutex_Unlock(&pfatFile->lock);
        if (rc != 0)
            return rc;
    }
    /* 继续读下一块数据 */
    ulong_t nextBlock = instance->fat[curBlock];
    curBlock = nextBlock;
}
/* 将缓冲区数据复制到调用读函数的进程缓冲区 */
memcpy(buf, pfatFile->fileDataCache + start, numBytes);
Debug("Read satisfied!\n");
return numBytes;
}

```

7.5.4 PFAT_Write 函数

函数功能：将文件缓冲区的数据写入 PFAT 文件系统。

函数参数：文件指针，缓冲区地址，写入文件的字节数。

函数返回值：错误标志 EACCESS。

```

static int PFAT_Write(struct File *file, void *buf, ulong_t numBytes)
{

```

```

/* 因为PFAT文件系统是只读的，所以不允许写入，直接返回标志 */
return EACCESS;
}

```

7.5.5 PFAT_Seek 函数

函数功能：文件定位函数，若参数位置值大于文件的最大长度，则失败。

函数参数：文件指针，位置值。

函数返回值：设置成功则返回 0。

```

static int PFAT_Seek(struct File *file, ulong_t pos)
{
    if (pos >= file->endPos)
        return EINVAL;
    file->filePos = pos;
    return 0;
}

```

7.5.6 PFAT_Read_Entry 函数

函数功能：从 PFAT 文件系统读出文件目录结构。

函数参数：文件指针，虚拟文件系统层的目录结构指针。

函数返回值：读取成功则返回 0。

```

static int PFAT_Read_Entry(struct File *dir, struct VFS_Dir_Entry *entry)
{
    directoryEntry *pfatDirEntry;
    struct PFAT_Instance *instance = (struct PFAT_Instance*) dir->mountPoint->
        fsData;

    if (dir->filePos >= dir->endPos) /* 最后一个目录项 */
        return VFS_NO_MORE_DIR_ENTRIES;
    pfatDirEntry = &instance->rootDir[dir->filePos++];
    strncpy(entry->name, pfatDirEntry->fileName, sizeof(pfatDirEntry->
        fileName));
    entry->name[sizeof(pfatDirEntry->fileName)] = '\0';
    Copy_Stat(&entry->stats, pfatDirEntry);
    return 0;
}

```

7.5.7 PFAT_Lookup 函数

函数功能：在 PFAT 文件系统中查找给定名称的目录项。

函数参数：PFAT 文件系统实例指针，路径名。

函数返回值：若查找成功则返回路径指针，否则返回 0。

```

static directoryEntry *PFAT_Lookup(struct PFAT_Instance *instance, const
char *path)
{
    directoryEntry *rootDir = instance->rootDir;
    bootSector *fsinfo = &instance->fsinfo;
    int i;
    KASSERT(*path == '/');
    /*若要查找的目录是根目录*/
    if (strcmp(path, "/") == 0)
        return &instance->rootDirEntry;
    /*跳过目录中的第一个 '/' */
    ++path;
    /*PFAT是单级目录结构，所以不需要处理层次目录*/
    for (i = 0; i < fsinfo->rootDirectoryCount; ++i) {
        directoryEntry *entry = &rootDir[i];
        if (strcmp(entry->fileName, path) == 0) {
            /*若找到给定名字的目录项，则返回目录指针*/
            Debug("Found matching dir entry for %s\n", path);
            return entry;
        }
    }
    /*若没有找到给定名的目录，则返回0*/
    return 0;
}

```

7.5.8 Get_PFAT_File 函数

函数功能：获取给定名称的 PFAT 文件对象。

函数参数：PFAT 文件系统指针，目录指针。

函数返回值：PFAT 文件对象指针。

```

static struct PFAT_File *Get_PFAT_File(struct PFAT_Instance *instance,
directoryEntry *entry)
{
    ulong_t numBlocks;
    struct PFAT_File *pfatFile = 0;
    char *fileDataCache = 0;
    struct Bit_Set *validBlockSet = 0;

    KASSERT(entry != 0);

```

```

KASSERT(instance != 0);
Mutex_Lock(&instance->lock);
/* 检测给定名称的文件是否已经打开，若文件已经打开，则直接得到已经存在的文件对象*/
for (pfatFile = Get_Front_Of_PFAT_File_List(&instance->fileList);
    pfatFile != 0;
    pfatFile = Get_Next_In_PFAT_File_List(pfatFile)) {
    if (pfatFile->entry == entry)
        break;
}
if (pfatFile == 0) {
    /* 确定文件数据缓冲块的大小*/
    numBlocks = Round_Up_To_Block(entry->fileSize) / SECTOR_SIZE;
    /*分配文件块，PFAT文件对象和文件数据缓冲区空间等*/
    if ((pfatFile = (struct PFAT_File *) Malloc(sizeof(*pfatFile))) == 0 ||
        (fileDataCache = Malloc(numBlocks * SECTOR_SIZE)) == 0 ||
        (validBlockSet = Create_Bit_Set(numBlocks)) == 0) {
        goto memfail;
    } //分配内存失败，则出错处理
    pfatFile->entry = entry;
    pfatFile->numBlocks = numBlocks;
    pfatFile->fileDataCache = fileDataCache;
    pfatFile->validBlockSet = validBlockSet;
    Mutex_Init(&pfatFile->lock);
    Add_To_Back_Of_PFAT_File_List(&instance->fileList, pfatFile);
    KASSERT(pfatFile->nextPFAT_File_List == 0);
}
goto done;
memfail: //出错处理，用于释放已经分配的内存空间
if (pfatFile != 0)
    Free(pfatFile);
if (fileDataCache != 0)
    Free(fileDataCache);
if (validBlockSet != 0)
    Free(validBlockSet);
done:
Mutex_Unlock(&instance->lock);
return pfatFile;
}

```

7.5.9 PFAT_Open 函数

函数功能：打开 PFAT 文件系统的文件。

函数参数：文件系统加载点指针，路径名，打开方式，文件指针。

```

static int PFAT_Open(struct Mount_Point *mountPoint, const char *path, int
mode, struct File **pFile)
{
    int rc = 0;
    struct PFAT_Instance     *instance     = (struct PFAT_Instance*)
mountPoint->fsData;
    directoryEntry *entry;
    struct PFAT_File *pfatFile = 0;
    struct File *file = 0;
    /*若打开文件系统的方式是写或创建，则拒绝操作 */
    if ((mode & (O_WRITE | O_CREATE)) != 0)
        return EACCESS;
    /* 查找目录项*/
    entry = PFAT_Lookup(instance, path);
    if (entry == 0)
        return ENOTFOUND;
    /* 若要打开的不是文件，则拒绝操作 */
    if (entry->directory)
        return EACCESS;
    /*得到文件的 PFAT_File指针 */
    pfatFile = Get_PFAT_File(instance, entry);
    if (pfatFile == 0)
        goto done;
    /*创建文件完成 */
    file = Allocate_File(&s_pfatFileOps, 0, entry->fileSize, pfatFile, 0, 0);
    if (file == 0) {
        rc = ENOMEM;
        goto done;
    }
    *pFile = file;
done:   return rc;
}

```

7.5.10 PFAT_Open_Directory 函数

函数功能：打开 PFAT 文件系统的目录结构。

函数参数：文件系统加载点指针、路径名、文件指针。

```

static int PFAT_Open_Directory(struct Mount_Point *mountPoint, const char
*path, struct File **pDir)
{
    struct PFAT_Instance *instance = (struct PFAT_Instance*) mountPoint->
fsData;
    struct File *dir;

```

```

if (strcmp(path, "/") != 0)
    return ENOTFOUND; //若目录名无效
dir = (struct File*) Malloc(sizeof(*dir));
if (dir == 0)
    return ENOMEM; //分配存储空间，若分配失败，则返回
dir->ops = &s_pfatDirOps;
dir->filePos = 0; /*读取下一个dir目录 */
dir->endPos = instance->fsinfo.rootDirectoryCount;
dir->fsData = 0;
*pDir = dir;
return 0;
}

```

7.5.11 PFAT_Mount 函数

函数功能：加载 PFAT 文件系统。

函数参数：加载点指针。

```

static int PFAT_Mount(struct Mount_Point *mountPoint)
{
    struct PFAT_Instance *instance = 0;
    bootSector *fsinfo;
    void *bootSect = 0;
    int rootDirSize;
    int rc;
    int i;
    /* 分配文件系统实例需要的内存空间 */
    instance = (struct PFAT_Instance*) Malloc(sizeof(*instance));
    if (instance == 0)
        goto memfail;
    memset(instance, '\0', sizeof(*instance));
    fsinfo = &instance->fsinfo;
    Debug("Created instance object\n");
    /*为文件系统的启动扇区分配缓冲区 */
    bootSect = Malloc(SECTOR_SIZE);
    if (bootSect == 0)
        goto memfail;
    /* 读文件系统的启动扇区数据 */
    if ((rc = Block_Read(mountPoint->dev, 0, bootSect)) < 0)
        goto fail;
    Debug("Read boot sector\n");
    memcpy(&instance->fsinfo, ((char*)bootSect) + PFAT_BOOT_RECORD_OFFSET,
           sizeof(bootSector));
    Debug("Copied boot record\n");
}

```

```
/* 检测魔数，若不是PFAT文件系统，则出错处理*/
if (fsinfo->magic != PFAT_MAGIC) {
    Print("Bad magic number (%x) for PFAT filesystem\n", fsinfo->magic);
    goto invalidfs;
}
Debug("Magic number is good!\n");
/* 检测文件系统参数是否合法*/
if (fsinfo->fileAllocationOffset <= 0 ||
    fsinfo->fileAllocationLength <= 0 ||
    fsinfo->rootDirectoryCount < 0 ||
    fsinfo->rootDirectoryOffset <= 0) {
    Print("Invalid parameters for PFAT filesystem\n");
    goto invalidfs;
}
Debug("PFAT filesystem parameters appear to be good!\n");
instance->fat = (int*) Malloc(fsinfo->fileAllocationLength *SECTOR_SIZE);
if (instance->fat == 0)
    goto memfail;
/* 读入FAT数据 */
for (i = 0; i < fsinfo->fileAllocationLength; ++i) {
    int blockNum = fsinfo->fileAllocationOffset + i;
    char *p = ((char*)instance->fat) + (i * SECTOR_SIZE);
    if ((rc = Block_Read(mountPoint->dev, blockNum, p)) < 0)
        goto fail;
}
Debug("Read FAT successfully!\n");
/* 分配根目录空间 */
rootDirSize = Round_Up_To_Block(sizeof(directoryEntry) * fsinfo->
rootDirectoryCount);
instance->rootDir = (directoryEntry*) Malloc(rootDirSize);
/* 读根目录空间 */
Debug("Root directory size = %d\n", rootDirSize);
for (i = 0; i < rootDirSize; i += SECTOR_SIZE) {
    int blockNum = fsinfo->rootDirectoryOffset + i;
    if ((rc = Block_Read(mountPoint->dev, blockNum, instance->rootDir +
(i*SECTOR_SIZE))) < 0)
        goto fail;
}
Debug("Read root directory successfully!\n");
memset(&instance->rootDirEntry, '\0', sizeof(directoryEntry));
instance->rootDirEntry.readOnly = 1;
instance->rootDirEntry.directory = 1;
instance->rootDirEntry.fileSize =
    instance->fsinfo.rootDirectoryCount * sizeof(directoryEntry);
```

```

/* 初始化文件系统锁和PFAT_File列表 */
Mutex_Init(&instance->lock);
Clear_PFAT_File_List(&instance->fileList);
PFAT_Register_Paging_File(mountPoint, instance);
mountPoint->ops = &s_pfatMountPointOps;
mountPoint->fsData = instance;
return 0;

memfail:
rc = ENOMEM; goto fail;

invalidfs:
rc = EINVALIDDFS; goto fail;

fail:
if (instance != 0) {
    if (instance->fat != 0)
        Free(instance->fat);
    if (instance->rootDir != 0)
        Free(instance->rootDir);
    Free(instance);
}
if (bootSect != 0)
    Free(bootSect);
return rc;
}

```

7.5.12 Init_PFAT 函数

函数功能：对 PFAT 文件系统初始化的函数。

函数参数：无。

函数返回值：无。

```

void Init_PFAT(void)
{
    Register_Filesystem("pfat", &s_pfatFilesystemOps);
}

```

7.5.13 Register_Filesystem 函数

函数功能：注册文件系统。

函数参数：文件系统名，文件系统指针。

函数返回值：注册成功则返回 true，否则返回 false。

```

bool Register_Filesystem(const char *fsName, struct Filesystem_Ops *fsOps)
{
}

```

```

struct Filesystem *fs;
KASSERT(fsName != 0);
KASSERT(fsOps != 0);
KASSERT(fsOps->Mount != 0);
Debug("Registering %s filesystem type\n", fsName);
fs = (struct Filesystem*) Malloc(sizeof(*fs));
//为文件系统结构分配空间
if (fs == 0)
    return false;//分配空间失败
fs->ops = fsOps; //为文件系统结构体成员赋值
strncpy(fs->fsName, fsName, VFS_MAX_FS_NAME_LEN);
fs->fsName[VFS_MAX_FS_NAME_LEN] = '\0';
/* 将文件系统结构体链入链表*/
Mutex_Lock(&s_vfsLock);
Add_To_Back_Of_Filesystem_List(&s_filesystemList, fs);
Mutex_Unlock(&s_vfsLock);
return true;
}

```

7.6 虚拟文件系统函数

7.6.1 Unpack_Path 函数

函数功能：将给定路径拆分为文件系统加载的路径（前缀）和文件名（后缀）。

函数参数：完整的路径名，保存前缀的缓冲区地址，保存后缀的缓冲区地址。

函数返回值：布尔值，路径有效返回 true，否则返回 false。

```

static bool Unpack_Path(const char *path, char *prefix, const char
**pSuffix)
{
    char *slash;
    size_t pfxLen;
    Debug("path=%s\n", path);
    if (*path != '/') //路径必须以'/'开头
        return false;
    ++path;
    slash = strchr(path, '/');
    if (slash == 0) { //若给定目录是根目录
        pfxLen = strlen(path);
        if (pfxLen == 0 || pfxLen > MAX_PREFIX_LEN)
            return false;
        strcpy(prefix, path);
        *pSuffix = "/";
    }
}

```

```

    }
else {
    pfxLen = slash - path;
    if (pfxLen == 0 || pfxLen > MAX_PREFIX_LEN)
        return false;
    memcpy(prefix, path, pfxLen);
    prefix[pfxLen] = '\0';
    *pSuffix = slash;
}
Debug("prefix=%s, suffix=%s\n", prefix, *pSuffix);
KASSERT(**pSuffix == '/');
return true;
}

```

7.6.2 Lookup_Filesystem 函数

函数功能：查找给定类型的文件系统。

函数参数：文件系统类型。

函数返回值：文件系统指针*。

```

static struct Filesystem *Lookup_Filesystem(const char *fstype)
{
    struct Filesystem *fs;
    Mutex_Lock(&s_vfsLock); //加互斥操作锁
    fs = Get_Front_Of_Filesystem_List(&s_filesystemList);
    //获取文件系统链表的头指针
    while (fs != 0) {
        if (strcmp(fs->fsName, fstype) == 0)
            break;
        fs = Get_Next_In_Filesystem_List(fs);
    }
    Mutex_Unlock(&s_vfsLock);
    return fs;
}

```

7.6.3 Lookup_Mount_Point 函数

函数功能：查找给定前缀的文件系统加载点。

函数参数：路径前缀。

函数返回值：加载点指针。

```

static struct Mount_Point *Lookup_Mount_Point(const char *prefix)
{

```

```
struct Mount_Point *mountPoint;
Mutex_Lock(&s_vfsLock);
/* 在加载的文件系统中查找与给定前缀吻合的文件系统 */
mountPoint = Get_Front_Of_Mount_Point_List(&s_mountPointList);
while (mountPoint != 0) {
    Debug("Lookup mount point: %s,%s\n", prefix, mountPoint->pathPrefix);
    if (strcmp(prefix, mountPoint->pathPrefix) == 0)
        break;
    mountPoint = Get_Next_In_Mount_Point_List(mountPoint);
}
Mutex_Unlock(&s_vfsLock);
return mountPoint;
}
```

7.6.4 Format 函数

函数功能：按照给定的文件系统类型格式化设备。

函数参数：待格式化的设备名，文件系统的类型名。

函数返回值：整数值。

```
int Format(const char *devname, const char *fstype)
{
    struct Filesystem *fs;
    struct Block_Device *dev = 0;
    int rc;

    fs = Lookup_Fileystem(fstype); //查找给定类型的文件系统
    if (fs == 0)
        return ENOFILESYS;
    Debug("Found %s filesystem type\n", fstype);
    if (fs->ops->Format == 0)
        return EUNSUPPORTED;
    if ((rc = Open_Block_Device(devname, &dev)) < 0)
        return rc; //打开给定名的设备
    Debug("Opened device %s\n", dev->name);
    rc = fs->ops->Format(dev); //调用文件系统的格式化函数
    Close_Block_Device(dev); //关闭设备
    return rc;
}
```

7.6.5 Mount 函数

函数功能：将文件系统加载到给定块设备。

函数参数：块设备名，加载路径，文件系统类型。

函数返回值：整数，若加载成功则返回 0。

```
int Mount(const char *devname, const char *pathPrefix, const char *fstype)
{
    struct Filesystem *fs;
    struct Block_Device *dev = 0;
    struct Mount_Point *mountPoint = 0;
    int rc;
    while (*pathPrefix == '/')
        ++pathPrefix; //跳过加载路径前面的'/
    if (strlen(pathPrefix) > MAX_PREFIX_LEN)
        return ENAMETOOLONG; //若加载路径超过最长长度，则提示出错
    fs = Lookup_FileSystem(fstype);
    if (fs == 0)      return ENOFILESYS;
    KASSERT(fs->ops->Mount != 0); /* 所有文件系统都必须加载才能使用*/
    if ((rc = Open_Block_Device(devname, &dev)) < 0)
        return rc;
    mountPoint = (struct Mount_Point*) Malloc(sizeof(*mountPoint));
    //新建一个加载点数据结构
    if (mountPoint == 0)
        goto memfail;
    memset(mountPoint, '\0', sizeof(*mountPoint));
    mountPoint->dev = dev;
    mountPoint->pathPrefix = strdup(pathPrefix);
    if (mountPoint->pathPrefix == 0)
        goto memfail;
    Debug("Mounting %s on %s using %s fs\n", devname, pathPrefix, fstype);
    if ((rc = fs->ops->Mount(mountPoint)) < 0) //调用文件系统的加载函数
        goto fail;
    Debug("Mount succeeded!\n");
    Mutex_Lock(&s_vfsLock);
    Add_To_Back_Of_Mount_Point_List(&s_mountPointList, mountPoint);
    //将文件系统链入系统维护文件系统的链表
    Mutex_Unlock(&s_vfsLock);
    return 0;
memfail:
    rc = ENOMEM;
fail:
    if (mountPoint != 0) {
        if (mountPoint->pathPrefix != 0)
            Free(mountPoint->pathPrefix);
        Free(mountPoint);
    }
}
```

```

    if (dev != 0)
        Close_Block_Device(dev);
    return rc;
}

```

7.6.6 Open 函数

函数功能：打开文件。

函数参数：文件路径，打开模式，文件对象指针。

函数返回值：表示打开成功与否的布尔值。

```

int Open(const char *path, int mode, struct File **pFile)
{
    int rc = Do_Open(path, mode, pFile, &Do_Open_File);
    /*if (rc != 0) { Print("File open failed with code %d\n", rc); }*/
    return rc;
}

```

7.6.7 Do_Open 函数

函数功能：执行打开文件和打开文件目录操作。

函数参数：文件路径，打开方式，文件对象指针。

函数返回值：表示打开成功与否的布尔值。

```

static int Do_Open(
    const char *path, int mode, struct File **pFile,
    int (*openFunc)(struct Mount_Point *mountPoint, const char *path, int
mode, struct File **pFile))
{
    char prefix[MAX_PREFIX_LEN + 1];
    const char *suffix;
    struct Mount_Point *mountPoint;
    int rc;
    if (!Unpack_Path(path, prefix, &suffix))
        return ENOTFOUND; //拆分路径名
    mountPoint = Lookup_Mount_Point(prefix); //查找文件系统加载点数据结构
    if (mountPoint == 0)
        return ENOTFOUND;
    rc = openFunc(mountPoint, suffix, mode, pFile);
    if (rc == 0) { //文件打开成功
        (*pFile)->mode = mode;
        (*pFile)->mountPoint = mountPoint;
    }
    return rc;
}

```

7.6.8 Close 函数

函数功能：关闭一个文件或目录。

函数参数：文件指针。

函数返回值：关闭成功返回 0，否则返回错误代码。

```
int Close(struct File *file)
{
    int rc;
    KASSERT(file->ops->Close != 0);
    rc = file->ops->Close(file);
    if (rc == 0)
        Free(file); //释放文件数据结构占用的存储空间
    return rc;
}
```

7.6.9 Read 函数

函数功能：从文件当前位置读指定字节的数据。

函数参数：文件对象指针，存放数据的缓冲区，读取数据字节数。

函数返回值：读取数据的字节数。

```
int Read(struct File *file, void *buf, ulong_t len)
{
    if (file->ops->Read == 0)
        return EUNSUPPORTED;
    else
        return file->ops->Read(file, buf, len);
}
```

7.6.10 Write 函数

函数功能：在文件的当前位置写入指定字节的数据。

函数参数：文件对象指针，待写入数据存放的缓冲区，待写入数据的字节数。

函数返回值：写入数据的字节数。

```
int Write(struct File *file, void *buf, ulong_t len)
{
    if (file->ops->Write == 0)
        return EUNSUPPORTED;
    else
        return file->ops->Write(file, buf, len);
}
```



7.6.11 Seek 函数

函数功能：改变读写文件数据的指针。

函数参数：文件对象指针，新的文件读写指针。

函数返回值：改变位置成功则返回 0。

```
int Seek(struct File *file, ulong_t len)
{
    if (file->ops->Seek == 0)
        return EUNSUPPORTED;
    else
        return file->ops->Seek(file, len);
}
```

7.6.12 Create_Directory 函数

函数功能：创建一个文件目录。

函数参数：目录名。

函数返回值：创建成功则返回 0。

```
int Create_Directory(const char *path)
{
    char prefix[MAX_PREFIX_LEN + 1];
    const char *suffix;
    struct Mount_Point *mountPoint;
    if (!Unpack_Path(path, prefix, &suffix)) //将路径名拆分为前缀和后缀
        return ENOTFOUND;
    mountPoint = Lookup_Mount_Point(prefix); //查找加载点
    if (mountPoint == 0)
        return ENOTFOUND;
    if (mountPoint->ops->Create_Directory == 0)
        return EUNSUPPORTED;
    else
        return mountPoint->ops->Create_Directory(mountPoint, suffix);
}
```

7.6.13 Delete 函数

函数功能：删除文件或目录。

函数参数：路径名。

函数返回值：删除成功则返回 0。

```
int Delete(const char *path)
{
    char prefix[MAX_PREFIX_LEN + 1];
    const char *suffix;
    struct Mount_Point *mountPoint;
    if (!Unpack_Path(path, prefix, &suffix))
        return ENOTFOUND;
    mountPoint = Lookup_Mount_Point(prefix);
    if (mountPoint == 0)
        return ENOTFOUND;
    if (mountPoint->ops->Delete == 0)
        return EUNSUPPORTED;
    else
        return mountPoint->ops->Delete(mountPoint, suffix);
}
```

第 8 章 GeekOS 设计项目 0

CHAPTER
08

8.1 项目设计目的

熟悉 GeekOS 的项目编译、调试和运行环境，掌握 GeekOS 运行工作过程。

8.2 项目设计要求

- (1) 搭建 GeekOS 的编译和调试平台，掌握 GeekOS 的内核进程工作原理。
- (2) 熟悉键盘操作函数，编程实现一个内核进程。该进程的功能是：接收键盘输入的字符并显示到屏幕上，当输入 Ctrl+D 时，结束进程的运行。

8.3 GeekOS 键盘处理函数

GeekOS 的键盘处理函数是定义在 `keyboard.h` 与 `keyboard.c` 两个文件中。在 `keyboard.c` 中定义了 F1~F12、Shift、Alt 等功能键常量，还定义了一个用于存放键盘扫描码的缓冲区。部分代码如下。

```
// 定义功能键常量
#define LEFT_CTRL 0x04
#define RIGHT_CTRL 0x08
// 定义缓冲区，用于存储输入的按键
#define QUEUE_SIZE 256
#define QUEUE_MASK 0xff
#define NEXT(index) (((index) + 1) & QUEUE_MASK)
static Keycode s_queue[QUEUE_SIZE];
static int s_queueHead, s_queueTail;
// 等待键盘输入的进程队列
static struct Thread_Queue s_waitQueue;
```

```

//没有按下Shift键的情况下，键盘扫描码与键值的对应关系
static const Keycode s_scanTableNoShift[] = {
    KEY_UNKNOWN, ASCII_ESC, '1', '2', /* 0x00 - 0x03 */
    '3', '4', '5', '6', /* 0x04 - 0x07 */
    '7', '8', '9', '0', /* 0x08 - 0x0B */
    '-', '=', ASCII_BS, '\t', /* 0x0C - 0x0F */
    'q', 'w', 'e', 'r', /* 0x10 - 0x13 */
    't', 'y', 'u', 'i', /* 0x14 - 0x17 */
    'o', 'p', '[', ']', /* 0x18 - 0x1B */
    '\r', KEY_LCTRL, 'a', 's', /* 0x1C - 0x1F */
    'd', 'f', 'g', 'h', /* 0x20 - 0x23 */
    'j', 'k', 'l', ';', /* 0x24 - 0x27 */
    '\'', `'', KEY_LSHIFT, '\\', /* 0x28 - 0x2B */
    'z', 'x', 'c', 'v', /* 0x2C - 0x2F */
    'b', 'n', 'm', ',', /* 0x30 - 0x33 */
    '.', '/', KEY_RSHIFT, KEY_PRINTSCRN, /* 0x34 - 0x37 */
    KEY_LALT, ' ', KEY_CAPSLOCK, KEY_F1, /* 0x38 - 0x3B */
    KEY_F2, KEY_F3, KEY_F4, KEY_F5, /* 0x3C - 0x3F */
    KEY_F6, KEY_F7, KEY_F8, KEY_F9, /* 0x40 - 0x43 */
    KEY_F10, KEY_NUMLOCK, KEY_SCROLL, KEY_KPHOME, /* 0x44 - 0x47 */
    KEY_KPUP, KEY_KPPGUP, KEY_KPMINUS, KEY_KPLEFT, /* 0x48 - 0x4B */
    KEY_KPCENTER, KEY_KPRIGHT, KEY_KPPLUS, KEY_KPEND, /* 0x4C - 0x4F */
    KEY_KPDOWN, KEY_KPPGDN, KEY_KPINSERT, KEY_KPDEL, /* 0x50 - 0x53 */
    KEY_SYSREQ, KEY_UNKNOWN, KEY_UNKNOWN, KEY_UNKNOWN, /* 0x54 - 0x57 */
};

#define SCAN_TABLE_SIZE (sizeof(s_scanTableNoShift) / sizeof(Keycode))

//按下Shift键的情况下的码表，键盘扫描码与键值的对应关系
static const Keycode s_scanTableWithShift[] = {
    KEY_UNKNOWN, ASCII_ESC, '!', '@', /* 0x00 - 0x03 */
    '#', '$', '%', '^', /* 0x04 - 0x07 */
    '&', '*', '(', ')', /* 0x08 - 0x0B */
    '_', '+', ASCII_BS, '\t', /* 0x0C - 0x0F */
    'Q', 'W', 'E', 'R', /* 0x10 - 0x13 */
    'T', 'Y', 'U', 'I', /* 0x14 - 0x17 */
    'O', 'P', '{', '}', /* 0x18 - 0x1B */
    '\r', KEY_LCTRL, 'A', 'S', /* 0x1C - 0x1F */
    'D', 'F', 'G', 'H', /* 0x20 - 0x23 */
    'J', 'K', 'L', ':', /* 0x24 - 0x27 */
    '`', `~, KEY_LSHIFT, '|', /* 0x28 - 0x2B */
    'Z', 'X', 'C', 'V', /* 0x2C - 0x2F */
    'B', 'N', 'M', '<', /* 0x30 - 0x33 */
    '>', '?', KEY_RSHIFT, KEY_PRINTSCRN, /* 0x34 - 0x37 */
    KEY_LALT, ' ', KEY_CAPSLOCK, KEY_F1, /* 0x38 - 0x3B */
    KEY_F2, KEY_F3, KEY_F4, KEY_F5, /* 0x3C - 0x3F */
};

```

```

KEY_F6, KEY_F7, KEY_F8, KEY_F9,      /* 0x40 - 0x43 */
KEY_F10, KEY_NUMLOCK, KEY_SCROLLLOCK, KEY_KPHOME,          /* 0x44 - 0x47 */
KEY_KPUP, KEY_KPPGUP, KEY_KPMINUS, KEY_KPLEFT,          /* 0x48 - 0x4B */
KEY_KPCENTER, KEY_KPRIGHT, KEY_KPPLUS, KEY_KPEND,        /* 0x4C - 0x4F */
KEY_KPDOWN, KEY_KPPGDN, KEY_KPINSERT, KEY_KPDEL,         /* 0x50 - 0x53 */
KEY_SYSREQ, KEY_UNKNOWN, KEY_UNKNOWN, KEY_UNKNOWN,        /* 0x54 - 0x57 */
};

}

```

键盘处理初始化是在 Main 函数中调用 Init_Keyboard 函数进行的，Init_Keyboard 主要功能是设置初始状态下存放键盘扫描码的缓冲区，并为键盘中断设置处理函数。函数代码如下。

```

void Init_Keyboard(void)
{
    ushort_t irqMask;
    Print("Initializing keyboard...\n");
    s_shiftState = 0;           // 设置系统初始状态下, Shift 键是不按下状态
    s_queueHead = s_queueTail = 0; // 存放键盘扫描码的缓冲区置空
    Install_IRQ(KB_IRQ, Keyboard_Interrupt_Handler); // 为键盘中断设置处
                                                    // 理函数
    irqMask = Get_IRQ_Mask();
    irqMask &= ~ (1 << KB_IRQ);
    Set_IRQ_Mask(irqMask);      // 设置键盘中断掩码
}

```

从 Init_Keyboard 函数代码可以看到，任何的按键操作都会引发键盘中断处理。键盘中断处理过程是：首先从相应 I/O 端口读取键盘扫描码，根据是否按下 Shift 键，分别在键值表中寻找扫描码对应的按键值，经过处理后将键值放入键盘缓冲区，最后通知系统重新调度进程。下面是键盘中断处理程序清单。

```

#define KEY_SPECIAL_FLAG 0x0100
#define KEY_KEYPAD_FLAG 0x0200
#define KEY_SHIFT_FLAG 0x1000
#define KEY_ALT_FLAG 0x2000
#define KEY_CTRL_FLAG 0x4000
#define KEY_RELEASE_FLAG 0x8000
static void Keyboard_Interrupt_Handler(struct Interrupt_State* state)
{
    uchar_t status, scanCode;
    unsigned flag = 0;
    bool release = false, shift;
    Keycode keycode;
    Begin_IRQ(state);           // 开始中断处理
    status = In_Bit(KB_CMD);    // 从 KB_CMD (0X64) 端口读一个字节的数据
}

```

```
IO_Delay();                                //在与低速外设通信时的短暂延时
if ((status & KB_OUTPUT_FULL) != 0) {
    scanCode = In_BytE(KB_DATA); //从KB_DATA(0X60)端口读一个字节数据
    IO_Delay();
    if (scanCode & KB_KEY_RELEASE) {
        // KB_KEY_RELEASE 代表0x80，若松开按键，键盘扫描码的高位为1
        release = true;
        scanCode &= ~(KB_KEY_RELEASE);
    }
    if (scanCode >= SCAN_TABLE_SIZE) { //若键扫描码超出上面定义的码表
        Print("Unknown scan code: %x\n", scanCode);
        goto done;
    }
    shift = ((s_shiftState & SHIFT_MASK) != 0); //判断是否按下Shift键
    keycode = shift ? s_scanTableWithShift[scanCode] : s_scanTableNoShift[scanCode];
    switch (keycode) { //若按下的是Shift、Ctrl、Alt键，则置标志flag为相应常量
        case KEY_LSHIFT:
            flag = LEFT_SHIFT;
            break;
        case KEY_RSHIFT:
            flag = RIGHT_SHIFT;
            break;
        case KEY_LCTRL:
            flag = LEFT_CTRL;
            break;
        case KEY_RCTRL:
            flag = RIGHT_CTRL;
            break;
        case KEY_LALT:
            flag = LEFT_ALT;
            break;
        case KEY_RALT:
            flag = RIGHT_ALT;
            break;
        default:
            goto noflagchange;
    }
    if (release)
        s_shiftState &= ~(flag);
    else
        s_shiftState |= flag;
    // 若按下Shift、Ctrl和Alt键，键值不需要放入缓冲区，直接置相应标记即可
    goto done;
}
```

```

noflagchange: //
    if (shift)
        keycode |= KEY_SHIFT_FLAG;
    if ((s_shiftState & CTRL_MASK) != 0)
        keycode |= KEY_CTRL_FLAG;
    if ((s_shiftState & ALT_MASK) != 0)
        keycode |= KEY_ALT_FLAG;
    if (release)
        keycode |= KEY_RELEASE_FLAG;
    Enqueue_Keycode(keycode); //将键值放入缓冲区
    Wake_Up(&s_waitQueue); //唤醒等待按键操作的进程
    g_needReschedule = true; //通知系统可以重新调度进程运行
}
done:
    End_IRQ(state); //结束中断处理
}

```

进程若需要获得键盘输入只要调用函数 `Wait_For_Key` 即可，进程调用该函数后，会阻塞进入按键操作的等待队列，直到按键操作结束，进程才被唤醒。

```

Keycode Wait_For_Key(void)
{
    bool gotKey, iflag;
    Keycode keycode = KEY_UNKNOWN;
    iflag = Begin_Int_Atomic();
    do {
        gotKey = !Is_Queue_Empty(); //判断键盘缓冲区是否为空
        if (gotKey)
            keycode = Dequeue_Keycode(); //从键盘缓冲区获得一个键值
        else
            Wait(&s_waitQueue); //若键盘缓冲区无键值，则等待
    }
    while (!gotKey);
    End_Int_Atomic(iflag);
    return keycode; //返回键值
}

```

函数 `Wait_For_Key()` 返回的是一个 16 位无符号整数 `keycode`，其低 10 位表示按下或放开的键码，而其他位作标志位使用。

- **KEY_SPECIAL_FLAG**: 该标志位用于表示按键是否有 ASCII 码表示，如按下的 是功能键或方向键等，该标志位就置 1，否则键码的低 8 位就表示按键的 ASCII 码。
- **KEY_KEYPAD_FLAG**: 置位就表示按下或释放的是数字键盘的按键。
- **KEY_SHIFT_FLAG**: 置位表示当前有一个 Shift 键是按下的，如此时按下字母键，就表示要输入的是大写的字母。
- **KEY_CTRL_FLAG** 和 **KEY_ALT_FLAG**: 则分别表示是否按下 Ctrl 和 Alt 键。

- KEY_RELEASE_FLAG: 用于标记释放按键。

文件/src/geekos/keyboard.c 中有另外一个用于获取键值的函数 Read_Key() 函数, 具体代码如下。

```
bool Read_Key(KeyCode* keycode)
{
    bool result, iflag;
    iflag = Begin_Int_Atomic();
    result = !Is_Queue_Empty(); //若键盘缓冲区为空, 表示读取键值失败
    if (result) {
        *keycode = Dequeue_Keycode(); //从键盘缓冲区读取第一个按键值
    }
    End_Int_Atomic(iflag);
    return result;
}
```

8.4 项目设计提示

从第 5 章的介绍中, 我们对内核进程数据结构、内核进程对象和内核进程操作函数等内容有了一定的认识。用户要重点关注的是 Start_Kernel_Thread 函数。

```
struct Kernel_Thread* Start_Kernel_Thread(Thread_Start_Func startFunc,
                                           ulong_t arg, int priority,
                                           bool detached)
{
    struct Kernel_Thread* kthread = Create_Thread(priority, detached);
    if (kthread != 0) {
        Setup_Kernel_Thread(kthread, startFunc, arg);
        Make_Runnable_Atomic(kthread);
    }
    return kthread;
}
```

函数的参数 startFunc 是一个 Thread_Start_Func 类型的函数指针, 其定义在 kthread.h 中:

```
typedef void (*Thread_Start_Func)(ulong_t arg);
```

该函数指针指向一个无返回值, 参数为 ulong_t 类型的函数。从函数代码可知, 该函数的功能是以参数 startFunc 指向的代码为进程体生成一个内核进程。因此项目 0 的实现主要分两个步骤完成 (主要工作都在项目 0 的/src/geekos/main.c 中完成):

(1) 编写一个函数, 函数功能是: 接收键盘输入的按键, 并将键值显示到显示器的函数, 当输入 Ctrl+D 就退出。

(2) 在 Main 函数体内调用 Start_Kernel_Thread 函数, 将步骤 (1) 编写的函数地址传递给参数 startFunc, 建立一个内核级进程。

9.1 项目设计目的

熟悉 ELF 文件格式，了解 GeekOS 系统如何将 ELF 格式的用户可执行程序装入到内存，建立内核进程并运行的实现技术。

9.2 项目设计要求

1. 修改/geekos/elf.c 文件：在函数 Parse_ELF_Executable() 中添加代码，分析 ELF 格式的可执行文件（包括分析得出 ELF 文件头、程序头，获取可执行文件长度，代码段、数据段等信息），并填充 Exe_Format 数据结构中的域值。
2. 掌握 GeekOS 在核心态运行用户程序的原理，为项目 2 的实现做准备。

9.3 ELF 文件格式

9.3.1 可执行文件

对于可执行文件有三个重要的概念：编译（compile）、连接（link，也可称为链接、联接）、加载（load，也可以称为装载）。源程序文件首先被编译成目标文件，多个目标文件被连接成一个可执行文件，最后可执行文件被加载到内存运行。相对于其他文件类型，可执行文件可能是一个操作系统中最重要的文件类型，因为它们是完成操作的真正执行者。可执行文件的大小、运行速度、资源占用情况以及可扩展性、可移植性等与文件格式的定义和文件加载过程紧密相关。

可执行文件有一些基本的要素是必需的——代码和数据，因为文件可能引用外部文件定义的符号（变量和函数），因此重定位信息和符号信息也是需要的。其他有一些辅助信息是可选的，如调试信息、硬

件信息等。系统一般使用区间（称为段 Segment）保存可执行文件的上述信息或节（Section）。除此之外，可执行文件通常都有一个文件头部用于描述文件的总体结构。

9.3.2 ELF（可执行连接格式）

可执行连接格式是 UNIX 系统实验室(USL)作为应用程序二进制接口(Application Binary Interface(ABI)而开发和发布的。工具接口标准委员会(TIS)选择了正在发展中的 ELF(Executable and Linking Format)标准作为工作在 32 位 INTEL 体系上不同操作系统之间可移植的二进制文件格式。

为了方便和高效，ELF 文件内容有两个平行视图，一个是从装入运行角度，另外一个从连接角度，如表 9-1 所示。

表 9-1 ELF 目标文件格式

连接程序视图	执行程序视图
ELF 头部	ELF 头部
程序头部表（可选）	程序头部表
节区 1	段 1
...	
节区 <i>n</i>	段 2
...	
...	...
节区头部表	节区头部表（可选）

文件开始处是一个 ELF 头部（ELF Header），用来描述整个文件的总体结构。

程序头部表（Program Header Table），用于指出如何创建进程映像，含有每个 Program Header 的入口指针，用来构造进程映像的目标文件必须具有程序头部表，可重定位文件不需要这个表。

节区（section）包含了具体程序段的数据和代码信息。

节区头部表（Section Head Table）包含了描述文件节区的信息，每个节区在表中都有一项，给出节区的名称、大小等信息。用于连接的目标文件必须包含节区头部表，其他目标文件可以有也可以没有。

实际上，文件中不一定包含全部这些内容，而且它们的位置也未必如表 9-1 所示安排，但是 ELF 头的位置是固定的，位于文件的开始处，其余各部分的位置、大小等信息由 ELF 头中的各项值来决定。

9.3.3 ELF Header

ELF 文件头的结构定义在头文件 `elf.h` 中，它的定义如下。

```
#define EI_NIDENT      16
typedef struct {
```

```

    unsigned char          e_ident[EI_NIDENT]; //目标文件标识
    Elf32_Half            e_type;           //目标文件类型
    Elf32_Half            e_machine;        //机器类型
    Elf32_Word            e_version;        //目标文件版本
    Elf32_Addr            e_entry;          //程序入口虚地址
    Elf32_Off             e_phoff;          //program header table的偏移量
    Elf32_Off             e_shoff;          //section header table的偏移量
    Elf32_Word            e_flags;          //处理器相关的标志信息
    Elf32_Half            e_ehsize;         //ELF头部的大小
    Elf32_Half            e_phentsize;       //program header table的表项大小
    Elf32_Half            e_phnum;          // program header table的表项数
    Elf32_Half            e_shentsize;       //section header table的表项大小
    Elf32_Half            e_shnum;          //section header table的表项数
    Elf32_Half            e_shstrndx;        //节区头部表与节区名称字符串表表项
                                         // 的索引
} Elf32_Ehdr;

```

结构的第一个成员是 16 字节的 `e_ident`, 用于确定该 ELF 文件是 32 位的还是 64 位或者其他位数的, 从而确定 ELF 文件头的结构, 我们假设打开 ELF 文件时返回的文件描述符是 `fd`, 我们可以用以下两条语句读出这 16 字节的内容:

```

lseek(fd,0,SEEK_SET);
read(fd,buf,16)

```

读出的内容放在 `buf` 数组中, 其中前 4 个字节是魔数, 若 `buf[0]~buf[3]` 中是 0x7f、E、L、F, 则表明这是一个 ELF 格式的二进制文件, `buf[4]` 用于表示 ELF 文件的位数, 若 `buf[4]` 为 1 则是 32 位的, 若是 2 则是 64 位的。`buf[5]` 用于给出字节序特性, 若值为 1 对应 LSB, 值为 2 对应 MSB。对于 Intel x86 `buf[5]=1`, `buf[6]` 给出 ELF 文件头的版本信息, 当前值为 `EV_CURRENT`, 对于 `buf[6]=EV_CURRENT` 的 ELF 文件头, `e_ident` 的后面 9 个字节全部为 0, 暂时没有使用。

`e_type` 为 1, 表明是重定位文件, 可以从连接视图进行解读, `e_type` 为 2, 表明是可执行文件, 可以从装载角度进行解读, 若 `e_type` 为 3, 表明是共享库动态库文件, 也可以从装载角度进行解读, 若 `e_type` 为 4, 表明是 core dump 文件, 从哪个视图进行解读依赖于具体的实现。

9.3.4 程序头部 (Program Header)

程序头部仅对于可执行文件和共享目标文件有意义, 可执行文件或者共享目标文件的程序头部是一个结构数组, 每个结构描述了一个段或者系统准备程序执行所必需的其他信息。目标文件的“段”包含一个或者多个“节区”, 也就是“段内容(Segment Contents)”。

在 ELF 头部的 `e_phentsize` 和 `e_phnum` 成员中给出了程序头部的大小和数量。程序头部的数据结构定义如下。

```

typedef struct {
    Elf32_Word p_type;           //段类型
    Elf32_Off p_offset;          //从文件头到该段第一个字节的偏移
    Elf32_Addr p_vaddr;          //段的第一个字节将被放到内存中的虚拟地址
    Elf32_Addr p_paddr;          //仅用于与物理地址相关的系统中
    Elf32_Word p_filesz;         //段在文件映像中所占的字节数
    Elf32_Word p_memsz;          //段在内存映像中占用的字节数
    Elf32_Word p_flags;          //与段相关的标志
    Elf32_Word p_align;          //段在文件中和内存中如何对齐
} Elf32_Phdr;

```

9.3.5 节区头部表格 (section header table)

在 ELF 头中用字段 e_shoff 给出了从文件头到节区头部表格的偏移字节数; e_shnum 给出表中节区的数目; e_shentsize 给出每个项目的字节数。从这些信息中可以确切地定位节区的具体位置。节区头数据结构定义如下。

```

typedef struct {
    Elf32_Word sh_name;          //节区名称
    Elf32_Word sh_type;          //节区类型
    Elf32_Word sh_flags;          //节区属性标志
    Elf32_Addr sh_addr;          //节区在进程映像中的起始位置
    Elf32_Off sh_offset;          //节区的第一个字节与文件头之间的偏移
    Elf32_Word sh_size;          //节区长度
    Elf32_Word sh_link;          //节区头部表的索引
    Elf32_Word sh_info;          //节区附加信息
    Elf32_Word sh_addralign;      //节区地址对齐约束标志
    Elf32_Word sh_entsize;        //节区中数据项的大小
} Elf32_Shdr;

```

9.4 用户可执行程序装入

GeekOS 用户程序是用户态进程的原型，由外壳程序与主程序组成，外壳程序在 Entry.c 中定义，代码如下。

```

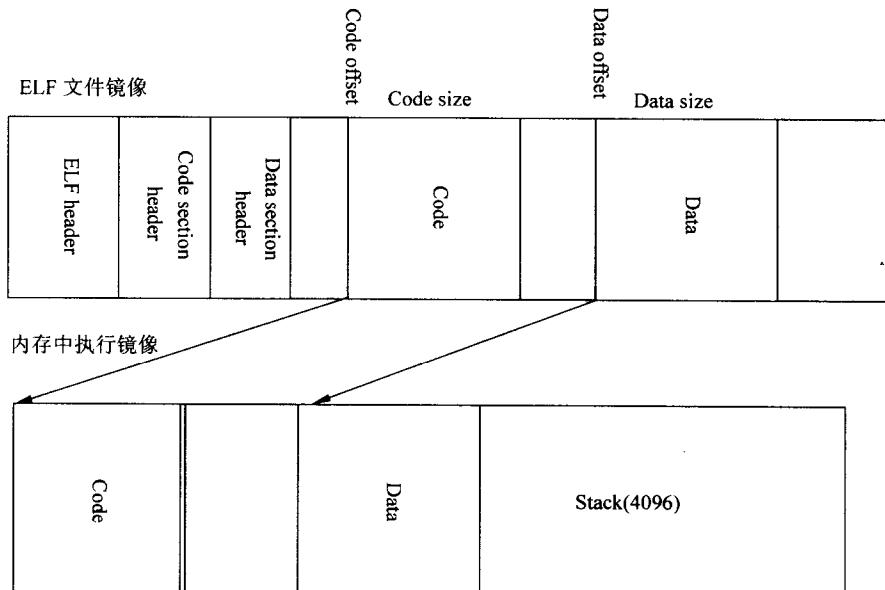
int main(int argc, char **argv); // 主函数的声明
void Exit(int exitCode);
void _Entry(void)                // 入口函数：这个函数负责调用用户程序的main函数
{
    struct Argument_Block *argBlock;
    // 参数块指针保存在ESI寄存器；在User_Context结构初始化时完成
    // 参数块的初始化，之后在创建用户进程对象的时候将指针存入ESI；
    __asm__ __volatile__ ("movl %%esi, %0" : "=r" (argBlock));
}

```

```
// 调用Main函数，并将其返回值作为退出代码传递给Exit库函数
Exit(main(argBlock->argc, argBlock->argv));
}
```

因此，用户程序的入口并不是 Main（Main 函数实际是开发者定义的程序主函数的名称，这个函数是 C 语言的约定俗称），而是 _Entry。

GeekOS 中的用户程序全部在系统编译阶段完成编译和连接，形成可执行文件。用户可执行文件保存在 PFAT 文件系统。本项目要完成的就是在 GeekOS 启动后，从 PFAT 文件系统将可执行文件装入内存，建立进程并运行得到相应的输出。在磁盘中的 ELF 文件的映像和在内存中的执行程序镜像的对应关系如图 9-1 所示。



注：为与 GeekOS 文件保持一致，本图保留英文。

图 9-1 ELF 文件和内存中的可执行文件镜像

9.5 项目设计提示

在 /src/geekos/main.c 文件中，用户可以看到函数调用 Spawner_Init_Process，该函数的功能是调用 Start_Kernel_Thread()，它以 Spawner 函数为进程体建立一个核心级进程，并使之准备就绪。函数 Spawner 主要功能是从 PFAT 文件系统读入一个用户执行程序，为之建立相应的进程影像，然后调用 Spawn_Program 函数建立相应进程。代码如下（/src/geekos/lprog.c）。

```
void Spawner( unsigned long arg )
{
    const char *program = "/c/a.exe"; // 指定要读入的用户程序的位置
```

```
char *exeFileData = 0;
ulong_t exeFileLength;
struct Exe_Format exeFormat;

if (lprogdebug)
{
    Print("Reading %s...\n", program);
}
/*调用Read_Fully函数将参数program指定文件的内容读入内存缓冲区*/
if (Read_Fully(program, (void**) &exeFileData, &exeFileLength) != 0)
{
    Print("Read_Fully failed to read %s from disk\n", program);
    goto fail;
}
/*提示读ELF文件成功*/
if (lprogdebug)
{
    Print("Read_Fully OK\n");
}
/*调用Parse_ELF_Executable函数分析ELF文件*/
if (Parse_ELF_Executable(exeFileData, exeFileLength, &exeFormat) != 0)
{
    Print("Parse_ELF_Executable failed\n");
    goto fail;
}
/*提示分析ELF文件成功，得到相应结果*/
if (lprogdebug)
{
    Print("Parse_ELF_Executable OK\n");
}
/*调用Spawn_Program函数将可执行程序的程序段和数据段等装入内存*/
if (Spawn_Program(exeFileData, &exeFormat) != 0)
{
    Print("Spawn_Program failed\n");
    goto fail;
}
/*用户程序相应进程已经创建完成，原来占用的缓冲区可以释放了 */
Free(exeFileData);
exeFileData = 0;
/* 如果系统能执行到这里，说明用户程序已经运行完成，系统可以结束了 */
Print("Hi ! This is the third (and last) string\n");
Print("If you see this you're happy\n");
Exit(0);
/* 如果系统能执行到这里，说明用户程序执行失败，释放资源，系统结束 */

```

```

fail:
    Disable_Interrupts();
    Free(virtSpace);
    Enable_Interrupts();
}

```

函数 `Spawn_Program` 在 `/src/geekos/lprog.c` 文件中已经实现，函数代码如下。参数的数据结构 `Exe_Format` 和 `Exe_Segment` 的定义在 `elf.h` 文件中。

```

struct Exe_Format {
    struct Exe_Segment segmentList[EXE_MAX_SEGMENTS];
    int numSegments;           /* 定义了ELF文件中段的个数 */
    ulong_t entryAddr;        /* 代码入口地址 */
};

struct Exe_Segment {
    ulong_t offsetInFile;     /* 段在可执行文件中的偏移值 */
    ulong_t lengthInFile;     /* 段在可执行文件中的长度 */
    ulong_t startAddress;     /* 段在内存中的起始地址 */
    ulong_t sizeInMemory;     /* 段在内存中的大小 */
    int protFlags;            /* 保护标志 */
};

static int Spawn_Program(char *exeFileData, struct Exe_Format *exeFormat)
{
    struct Segment_Descriptor* desc;
    unsigned long virtSize;
    unsigned short codeSelector, dataSelector;
    int i;
    ulong_t maxva = 0;
    /* 计算执行程序的虚地址空间大小 */
    for (i = 0; i < exeFormat->numSegments; ++i) {
        struct Exe_Segment *segment = &exeFormat->segmentList[i];
        ulong_t topva = segment->startAddress + segment->sizeInMemory;
        if (topva > maxva)
            maxva = topva;
    }
    /* 给进程分配内存空间=虚地址空间+堆栈区间 */
    virtSize = Round_Up_To_Page(maxva) + 4096;
    virtSpace = Malloc(virtSize);
    memset((char *) virtSpace, '\0', virtSize);
    /* 把缓冲区中的数据复制到刚分配的内存区间 */
    for (i = 0; i < exeFormat->numSegments; ++i) {
        struct Exe_Segment *segment = &exeFormat->segmentList[i];
    }
}

```

```

    memcpy(virtSpace + segment->startAddress,
           exeFileData + segment->offsetInFile,
           segment->lengthInFile);
}
/* 为代码段和数据段分配和初始化段描述符和选择子 */
// Kernel code segment.
desc = Allocate_Segment_Descriptor();
Init_Code_Segment_Descriptor(
    desc,
    (unsigned long)virtSpace, // 基地址
    (virtSize/PAGE_SIZE)+10, // 页数
    0                      // 特权等级
);
codeSelector = Selector( 0, true, Get_Descriptor_Index( desc ) );
// Kernel data segment.
desc = Allocate_Segment_Descriptor();
Init_Data_Segment_Descriptor(
    desc,
    (unsigned long)virtSpace, // 基地址
    (virtSize/PAGE_SIZE)+10, // 页数
    0                      // 特权等级
);
dataSelector = Selector( 0, true, Get_Descriptor_Index( desc ) );

Install_Interrupt_Handler( 0x90, &Printrap_Handler );
if (lprogdebug)      //显示程序处理的结果信息
{
    Print("Spawn_Program(): all structures are set up\n");
    Print(" virtSpace    = %x\n", (unsigned int) virtSpace);
    Print(" virtSize     = %x\n", (unsigned int) virtSize);
    Print(" codeSelector = %x\n", codeSelector);
    Print(" dataSelector = %x\n", dataSelector);

    Print("Now calling Trampoline()... \n");
}
Trampoline(codeSelector, dataSelector, exeFormat->entryAddr);
return 0;
}

```

从上述代码可以看到，`Spawn_Program` 函数完成的工作如下。

- (1) 根据 `Exe_Format` 中的 `Exe_Segment` 结构提供的用户程序段信息，及用户进程堆栈大小计算用户进程所需的最大内存空间，即要分配给用户进程的内存空间。
- (2) 为用户程序分配内存空间，并全部初始化为零，否则系统后面运行可能出错。
- (3) 根据段信息将用户程序中的各段内容复制到分配的用户内存空间。



(4) 根据 `Exe_Segment` 提供的用户段信息初始化代码段、数据段以及堆栈段的段描述符和段选择子。

由于系统已经实现了内核的所有函数，用户在完成本项目设计时要完成的任务很简单，只需要完成函数 `Parse_ELF_Executable` 即可。函数 `Parse_ELF_Executable` 声明为：

```
int Parse_ELF_Executable(char *exeFileData, ulong_t exeFileLength,
    struct Exe_Format *exeFormat)
```

参数：`exeFileData`——已装入内存的可执行文件所占用空间的起始地址；

`exeFileLength`——可执行文件长度；

`exeFormat`——保存分析得到的ELF文件信息的结构体指针。

根据 ELF 文件格式，用户可以从 `exeFileData` 指向的内容中得到 ELF 文件头，继续分析可以得到程序头，程序代码段等信息。

在项目 2 中，是要求用户在用户级上运行用户程序，很多函数是要求用户自己实现的，实现的原理跟内核级的实现原理类似，但不完全相同。所以用户在项目 1 中不仅仅仅是完成函数 `Parse_ELF_Executable` 的代码，更重要的是要了解系统如何装入用户程序并运行的原理，为项目 2 的实现做准备。

在项目 1 的`./src/user` 目录下有一个 `a.c` 文件，编译 GeekOS 后（即 `make` 后）可以得到可执行程序 `a.exe`，并写进 PFAT 文件系统，路径为`/c/a.exe`。项目将此作为待装入的可执行文件，启动 Bochs 看到以下输出就表示项目 1 设计完成了。

```
Hi ! This is the first string
Hi ! This is the second string
Hi ! This is the third (and last) string
If you see this you're happy
```

第 10 章 GeekOS 设计项目 2

CHAPTER 10

10.1 项目设计目的

扩充 GeekOS 操作系统内核，使得系统能够支持用户级进程的动态创建和执行。

10.2 项目设计要求

开始本项目前需要阅读/src/GeekOS 目录中的以下程序。

(1) Entry.c: 用户程序外壳，用户程序的入口地址就在这里。此文件在编译时与用户程序一起编译，具体请阅读 makefile 文件。

(2) Lowlevel.asm: 其中 Handle_Interrupt 是中断处理的总调度程序，该函数根据传递的中断向量查找并调用相关的中断处理程序，并实现调度进程的选择。Switch_To_Thread 函数用于实现进程的切换，此函数先对进程类型进行判断，如果是用户态进程，就切换到用户态进程上下文。

(3) Kthread.c: 内核进程有关函数以及进程调度算法都在此实现。

(4) Userseg.c: 其中要关注的函数有

- Destroy_User_Context()函数功能是释放 User_Context 空间，Detach_User_Context()调用该函数。
- Load_User_Program()函数功能是对用户态进程的 User_Context 结构初始化，并对用户态进程的初始化，Spawn()函数中调用该函数。

本项目要求用户对以下几个文件进行修改。

(1) src/GeekOS/user.c 文件中的函数 Spawn(), 其功能是生成一个新的用户级进程。

(2) src/GeekOS/user.c 文件中的函数 Switch_To_User_Context(), 调度程序在执行一个新的进程前调用该函数以切换用户地址空间。

(3) src/GeekOS/elf.c 文件中的函数 Parse_ELF_Executable(). 该

函数的实现要求和项目 1 相同。

(4) src/GeekOS/userseg.c 文件中主要是实现一些为实现对 src/GeekOS/user.c 中高层操作支持的函数。

- Destroy_User_Context() 函数的功能是释放用户态进程占用的内存资源。
- Load_User_Program() 函数的功能是通过加载可执行文件镜像创建新进程的 User_Context 结构。
- Copy_From_User() 和 Copy_To_User() 函数的功能是在用户地址空间和内核地址空间之间复制数据，在分段存储器管理模式下，只要段有效，调用 memcpy 函数就可以实现这两个函数的功能。
- Switch_To_Address_Space() 函数的功能是通过将进程的 LDT 装入到 LDT 寄存器来激活用户的地址空间。

(5) src/GeekOS/kthread.c 文件中的 Start_User_Thread 函数和 Setup_User_Thread 函数。

- Setup_User_Thread() 函数的功能是为进程初始化内核堆栈，堆栈中是为进程首次进入用户态运行时设置处理器状态要使用的数据。
- Start_User_Thread() 是一个高层操作，该函数使用 User_Context 对象开始一个新进程。

(6) src/GeekOS/kthread.c 文件中主要是实现用户程序要求内核进行服务的一些系统调用函数定义。要求用户实现的有 Sys_Exit() 函数、Sys_PrintString() 函数、Sys_GetKey()、Sys_SetAttr()、Sys_GetCursor()、Sys_PutCursor()、Sys_Spawn() 函数、Sys_Wait() 函数和 Sys_GetPID() 函数。这些函数在文件中有详细的注释，按照提示用户可以很好实现它们的功能。

最后，需要在 main.c 文件中改写生成第一个用户态进程的函数调用：Spawn_Init_Process(void)。需要注意的是：作为与用户沟通的界面，GeekOS 提供了一个简单的 Shell，保存在 PFAT 文件系统内，所以 GeekOS 系统启动后，应启动 shell 程序 /c/shell.exe 运行，所以需要将 /c/shell.exe 作为可执行文件传递给 Spawn 函数的 program 参数，创建第一个用户态进程，然后由它来创建其他进程。

添加代码运行成功后，GeekOS 就可以挂载 shell，并能运行测试文件 c.exe 和 b.exe。

10.3 项目设计提示

10.3.1 GeekOS 的用户态进程

GeekOS 的初始系统不支持用户态进程，但提供了用户态进程上下文接口和实现用户态进程需要用到的数据结构。所以，用户态进程及相关函数（包括用户程序的导入，用户态进程对象的初始化以及上下文切换）都由开发者实现。GeekOS 的进程结构如图 10-1 所示。

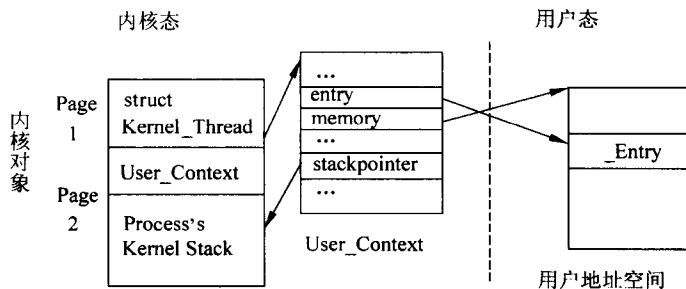


图 10-1 用户态进程结构

在 GeekOS 中为了区分用户态进程和内核进程，在 `Kernel_Thread` 结构体中设置了一个字段 `userContext`，指向用户态进程上下文。对于内核进程来说，这个指针为空，而用户态进程都拥有自己的用户上下文（`User_Context`）。因此，在 GeekOS 中要判断一个进程是内核进程还是用户态进程，只要通过 `userContext` 字段是否为空来判断就可以了。

`User_Context` 结构定义如下（在 `include/geekos/user.h` 中定义）。

```
struct User_Context {
#define NUM_USER_LDT_ENTRIES 3
    struct Segment_Descriptor ldt[NUM_USER_LDT_ENTRIES];      // 用户LDT
    struct Segment_Descriptor* ldtDescriptor;                   // LDT描述符
    char* memory;                                              // 指向用户空间
    ulong_t size;                                               // 用户空间的大小
    ushort_t ldtSelector;                                       // ldt选择子
    ushort_t csSelector;                                       // cs选择子
    ushort_t ssSelector;                                       // ss选择子
    ushort_t dsSelector;                                       // ds选择子
    pde_t *pageDir;                                            // 页表指针
    ulong_t entryAddr;                                         // 用户程序入口地址
    ulong_t argBlockAddr;                                       // 参数块地址
    ulong_t stackPointerAddr;                                   // 用户态进程的堆栈指针
    int refCount;                                              // 引用数
    struct File *fileList[USER_MAX_FILES]; // 打开文件列表
    int fileCount;                                             // 打开文件计数
};
```

`User_Context` 结构的新建和注销可以调用函数实现，新建 `User_Context` 结构的函数是 `Create_User_Context`，函数参数是 `User_Context` 结构的大小，代码如下。

```
static struct User_Context* Create_User_Context(ulong_t size)
{
    struct User_Context * pUserContext = 0;
    KASSERT(size > 0); // 如果给定的size不合法则报错
    pUserContext = Malloc(size);
```

```

if (pUserContext != 0)
    return pUserContext;
else
    return 0;
}

```

注销 User_Context 对象的函数是 Destroy_User_Context，参数是待注销的 User_Context 对象指针。

```

void Destroy_User_Context(struct User_Context* userContext)
{
    Free(userContext->ldtDescriptor); //释放占用的LDT
    userContext->ldtDescriptor = 0;
    Free(userContext->memory); // 释放内存空间
    userContext->memory = 0;
    Free(userContext); // 释放User_Context本身占用的内存
    userContext = 0;
}

```

一般来讲，内核进程会持续运行直到系统关闭，而用户态进程则在运行结束之后通过调用 Exit 函数（在 kthread.c 中定义）退出。Exit 函数与内核 Reaper 进程共同配合完成用户态进程资源的释放，Exit 函数代码如下。但释放用户态进程的机制需要开发人员自己完成。

```

void Exit(int exitCode)
{
    struct Kernel_Thread* current = g_currentThread;
    if (Interrupts_Enabled())
        Disable_Interrupts(); /*关中断*/
    current->exitCode = exitCode;
    current->alive = false;
    /*设置退出码，并将当前运行进程设置为消亡状态*/
    Tlocal_Exit(g_currentThread); /*清空进程局部存储空间*/
    Wake_Up(&current->joinQueue); /*若进程有父进程，则通知*/
    Detach_Thread(g_currentThread);
    Schedule(); /*重新调度选择进程运行*/
}

```

10.3.2 用户态进程空间

每个用户态进程都要占用一段物理上连续的内存空间，存储用户态进程的数据和代码。所以，为了实现存取访问控制，每个用户态进程都拥有属于自己的内存段空间，如代码段、数据段、堆栈段等，每个段有一个段描述符（segment descriptor），并且每个进

程有一个段描述符表（Local Descriptor Table），用于保存该进程的所有段描述符。操作系统中还设置一个全局描述符表（GDT，Global Descriptor Table），用于记录了系统中所有进程的 LDT 描述符。GDT、LDT 和 User_Context 的关系如图 10-2 所示。

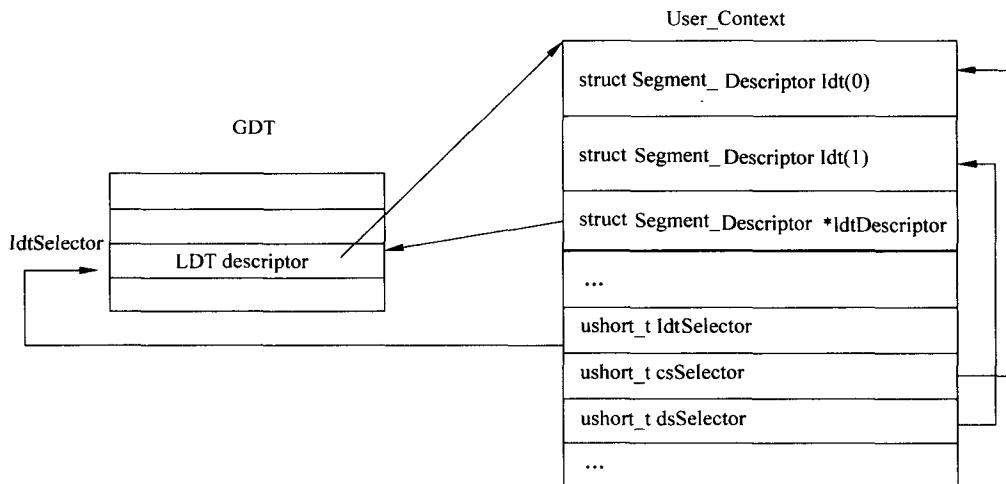


图 10-2 GDT、LDT 和 User_Context 的关系

为用户态进程创建 LDT 的步骤是：

- (1) 调用函数 `Allocate_Segment_Descriptor()`新建一个 LDT 描述符；
- (2) 调用函数 `Selector()`新建一个 LDT 选择子；
- (3) 调用函数 `Init_Code_Segment_Descriptor()`新建一个文本段描述符；
- (4) 调用函数 `Init_Data_Segment_Descriptor()`新建一个数据段；
- (5) 调用函数 `Selector()`新建一个数据段选择子；
- (6) 调用函数 `Selector()`新建一个文本（可执行代码）段选择子。

从以上描述可知，用户态进程除了需要数据段和代码段用于装入可执行文件和数据外，还需要另外两个数据结构的存储空间，一个是进程的堆栈，另一个是参数块(argument block) 数据结构。进程的堆栈段占用进程空间的最顶端，堆栈的大小为在 `userseg.c` 中定义了的常量 `DEFAULT_USER_STACK_SIZE`，而参数块数据结构在 `argblock.h` 中定义，主要用于存储传递给 Main 函数的 `argc` 和 `argv` 参数。

```

struct Argument_Block {
    int argc;
    char **argv;
};
  
```

参数块的实际数据大小由参数 `argc` 和 `argv` 决定，在文件 `argblock.c` 中定义了函数 `Get_Argument_Block_Size` 从传递给 `Spawn` 函数的 `command` 中得到参数的个数和存储参数需要的字节数，函数 `Format_Argument_Block` 从 `Get_Argument_Block_Size` 得到参数个

数和参数大小后，在分配给用户态进程的内存区构造参数块数据结构。

10.3.3 用户堆栈空间初始化

在用户态进程首次被调度前，系统必须初始化用户态进程的堆栈，使之看上去像进程刚被中断运行一样，因此需要使用 Push 函数将以下数据压入堆栈。

- 数据选择子
- 堆栈指针（注意要与数据段保持一致）
- Eflags（IF 位必须置位）
- 文本（可执行代码）选择子
- 程序计数器（代码入口地址）
- 错误代码（0）
- 中断号（0）
- 通用寄存器（必须包含参数块地址）
- DS 寄存器
- ES 寄存器
- FS 寄存器
- GS 寄存器

10.3.4 用户态进程创建

由第 5 章描述可以知道，GeekOS 系统初始内核不支持用户态进程，但在内核进程控制块 Kernel_Thread 中有一个字段 User_Context 用于标志进程是内核进程还是用户态进程，若创建的进程为核态进程，该字段就赋值为 0。User_Context 结构详见 10.3.1 节的介绍。

从 User_Context 结构可知，用户的 LDT 及各段描述符选择子、代码段入口都保存在这个结构中。因此在装入用户程序时，首先需要对 User_Context 结构进行初始化，然后根据这个结构内的信息进行用户态进程的初始化。

用户态进程的创建过程可以描述如下。

(1) Spawn 函数导入用户程序并初始化：调用 Load_User_Program 进行 User_Context 的初始化及用户态进程空间的分配、用户程序各段的装入；

(2) Spawn 函数调用 Start_User_Thread 函数，初始化一个用户态进程，包括初始化进程的 Kernel_Thread 结构以及调用 Setup_User_Thread 初始化用户态进程的内核堆栈；Setup_User_Thread 内部需要调用 Attach_User_Context 加载用户上下文；

(3) 最后 Spawn 函数退出，这时用户态进程已被添加到系统运行进程链表，可以被调度了。

Spawn 函数代码需要开发人员添加，Spawn 函数原型如下：

```
int Spawn(const char *program, const char *command, struct Kernel_Thread
**pThread)
```

参数说明：Program 对应的是要读入内存缓冲区的可执行文件，Command 是用户执行程序执行时的命令行字符串，pThread 是存放指向刚创建进程的指针。

Spawn 函数主要完成的主要功能是

- (1) 调用 **Read_Fully** 函数将名为 program 的可执行文件全部读入内存缓冲区。
- (2) 调用 **Parse_ELF_Executable** 函数，分析 ELF 格式文件。Parse_ELF_Executable 函数功能在项目 1 中已经实现。
- (3) 调用 **Load_User_Program** 将可执行程序的程序段和数据段等装入内存，初始化 **User_Context** 数据结构。
- (4) 调用 **Start_User_Thread** 函数创建一个进程并使该进程进入准备运行队列。

Read_Fully 函数在 /src/geekos/vfs.c 文件中定义，主要功能是从 PFAT 文件系统中把某一 ELF 执行文件全部读入内存缓冲区中，函数原型如下：

```
int Read_Fully(const char *path, void **pBuffer, ulong_t *pLen);
/* 参数说明:
   path——指向要读入执行文件的名称，包括路径;
   pBuffer——指向存放文件内容的缓冲区;
   pLen——返回文件内容的长度数据。
*/
```

Parse_ELF_Executable 函数功能代码在项目 1 中已经实现，可以直接复制到项目 2 中。

Load_User_Program 函数在 /src/geekos/userseg.c 文件中实现，代码也需要开发人员自己完成，函数原型如下：

```
int Load_User_Program(char *exeFileData, ulong_t exeFileLength,
                      struct Exe_Format *exeFormat, const char *command,
                      struct User_Context **pUserContext)
/* 参数说明:
   exeFileData——保存在内存缓冲中的用户程序可执行文件;
   exeFileLength——可执行文件的长度;
   exeFormat——调用Parse_ELF_Executable函数得到的可执行文件格式信息;
   command——用户输入的命令行，包括可执行文件的名称及其他参数;
   pUserContext——指向User_Context的指针，是本函数完成用户上下文初始化的对象
*/
```

主要实现功能如下。

- (1) 根据 **Parse_ELF_Executable** 函数的执行结果 **Exe_Format** 中的 **Exe_Segment** 结构提供的用户程序段信息，用户命令参数及用户态进程堆栈大小计算用户态进程所需的最大内存空间，即要分配给用户态进程的内存空间。

- (2) 为用户程序分配内存空间，并初始化。
 - (3) 根据 `Exe_Segment` 提供的用户段信息初始化代码段、数据段以及堆栈段的段描述符和段选择子。
 - (4) 根据段信息将用户程序中的各段内容复制到分配的用户内存空间。
 - (5) 根据 `Exe_Format` 结构初始化 `User_Context` 结构中的用户态进程代码段入口 `Entry` 字段，并根据 `command` 参数初始化用户内存空间中的参数块。
 - (6) 初始化 `User_Context` 结构的用户打开文件列表，并添加标准输入输出文件。
 - (7) 将初始化完毕的 `User_Context` 指针赋予 `*pUserContext`，返回 0 表示成功。
- `Load_User_Program` 成功完成后，即得到了用户程序在系统中的用户态进程体，以及用户上下文 `User_Context` 结构，这之后就可以继续调用 `Start_User_Thread` 函数创建一个新进程并使该进程进入准备运行队列等待运行。

11.1 项目设计目的

研究进程调度算法，掌握用信号量实现进程间同步的方法。为 GeekOS 扩充进程调度算法——基于时间片轮转的进程多级反馈调度算法，并能用信号量实现进程协作。

11.2 项目设计要求

- (1) 实现 src/geekos/syscall.c 文件中的 Sys_SetSchedulingPolicy 系统调用，它的功能是设置系统采用的何种进程调度策略。
- (2) 实现 src/geekos/syscall.c 文件中的 Sys_GetTimeOfDay 系统调用，它的功能是获取全局变量 g_numTicks 的值。
- (3) 实现函数 Change_Scheduling_Policy()，具体实现不同调度算法的转换。
- (4) 实现 syscall.c 中信号量有关的四个系统调用：sys_createSemaphore()、sys_P()、sys_V() 和 sys_destroySemaphore()。

11.3 项目设计提示

11.3.1 GeekOS 进程调度处理过程

在第 5 章讨论 Init_Timer 函数的时候讲到过进程调度是在时钟中断处理函数 Timer_Interrupt_Handler(/src/geekos/timer.c)内实现的，其代码如下。

```
static void Timer_Interrupt_Handler(struct Interrupt_
State* state)
{
```

```

int i;
struct Kernel_Thread* current = g_currentThread;
/* 当前运行进程指针 */
Begin_IRQ(state);
++g_numTicks;
++current->numTicks;
/* 修改系统已运行的总时间和当前运行进程运行的时间*/
for (i=0; i < timeEventCount; i++) {
    if (pendingTimerEvents[i].ticks == 0) {
        if (timerDebug) Print("timer: event %d expired (%d ticks)\n",
            pendingTimerEvents[i].id, pendingTimerEvents[i].origTicks);
        (pendingTimerEvents[i].callBack)(pendingTimerEvents[i].id);
    } else {
        pendingTimerEvents[i].ticks--;
    }
} /* 修改定时器事件 */
if (current->numTicks >= g_Quantum) {
    g_needReschedule = true;
    /* 若当前运行进程时间片用完，则修改变量g_needReschedule的值，表示需要重新调度新进程投入运行 */
    if (current->currentReadyQueue < (MAX_QUEUE_LEVEL - 1)) {
        current->currentReadyQueue++;
    } /* 将当前运行进程放入下一优先级队列 */
}
End_IRQ(state);
}

```

用户可以看到，第 5 章介绍进程的 `Kernel_Thread` 结构中有一个 `numTicks` 变量，`numTicks` 在进程对象初始化时被初始化为零，之后每次时钟中断处理时，进程的该变量都会加 1，然后系统会检查进程执行的时间是否超过了系统规定的时间片 `g_Quantum`，如果超过了这个值，就说明当前进程时间片已用完，系统应调度新的进程运行，于是将变量 `g_needReschedule` 置为 `true`，用以标志系统需要重新调度新进程运行。在时钟中断处理函数返回到 `Handle_Interrupt` 后检查 `g_needReschedule` 变量，如果变量值为 `true`，就调用 `Make_Runnable` 函数（在 `kthread.c` 中实现），将当前运行进程放入准备运行进程队列 `s_runQueue`；之后再调用 `Get_Next_Runnable` 函数（在 `kthread.c` 中实现）找到优先级最高的进程作为当前运行进程，并将其从准备运行进程队列中取出。最后返回 `Handle_Interrupt` 将 `g_needReschedule` 改回为 `false`，并切换到新进程运行。进程调度处理过程如图 11-1 所示。

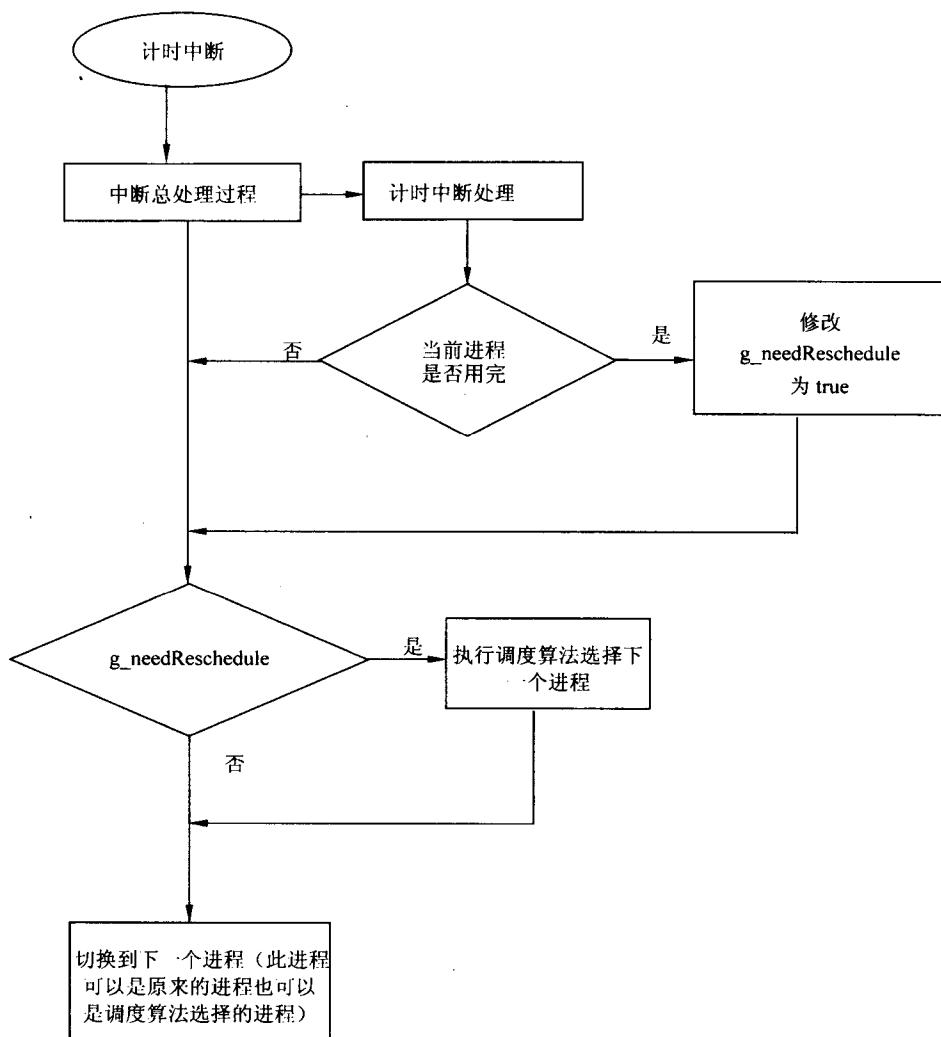


图 11-1 进程调度处理过程

11.3.2 四级反馈队列调度策略实现

本项目要求设计一个四级反馈队列调度算法，即用四个准备运行队列代替 GeekOS 原始系统中的一个队列，实现思想是：给四个准备运行队列不同的优先级，优先级别标记为数字 0~3，如果数字为 0 表示最高优先级，数字为 3 表示最低优先级。在四级反馈队列调度中，进程从创建到消亡的生命过程又如何在各队列存在呢？

首先，新创建的进程被放入优先级最高的准备运行队列，即优先级为 0 的准备运行队列。若优先级为 0 的进程被调度后，在给定时间片内无法完成，那么进程就被移到下一优先级队列的尾部（即优先级为 1 的准备运行队列），以此类推，直到进程被放到优先

级为 3 的队列为止。因此，要求运行时间较长的进程最终会被放入到优先级为 3 的准备运行队列。若进程被阻塞，每经过一个时间片，进程优先级都将增加 1，所以当进程阻塞达连续的 3 个时间片后，又将升到最高优先级。进程调度总是优先调度优先级高的进程运行，当优先级为 0 的进程队列为空时，进程调度从优先级为 1 的队列选择进程调度。大家应该还记得系统中的空闲进程 `Idle`，在四级反馈队列调度中，该进程应始终放在优先级为 3 的进程队列尾部，以便系统中没有其他可调度进程时就运行它。

为实现四级队列，设计中首先要修改 `s_runQueue`（在 `src/geekos/kthread.c`）的定义，从原来的一个结构体改为一个 4 元素的结构体数组，每一个结构体元素用于存放一个优先级队列的队首指针。在进程调度时，首先在最高优先级（优先级 0）的队列里面找，如果有进程存在，则调度它运行。如果没有，那就在次优先级的队列里面找，以此类推，直到找到一个进程投入运行为止。GeekOS 系统将 `Idle`（系统空闲）进程始终放在优先级为 3 的进程队列末尾，且不允许移动到其他队列，以保证进程调度时一定能找到进程投入运行。

当实现四级反馈队列调度策略后，GeekOS 系统就拥有 2 种进程调度策略，即单队列时间片轮转调度策略和四级反馈队列调度策略。那么，系统究竟采用何种调度策略呢？调度策略的确定是通过系统调用 `Sys_SetSchedulingPolicy` 实现的，函数原型为：

```
static int Sys_SetSchedulingPolicy(struct Interrupt_State* state);
```

参数 `state` 定义为如下 `struct Interrupt_State` 类型：

```
struct Interrupt_State {
    uint_t gs;
    uint_t fs;
    uint_t es;
    uint_t ds;
    uint_t ebp;
    uint_t edi;
    uint_t esi;
    uint_t edx;
    uint_t ecx;
    uint_t ebx;
    uint_t eax;
    uint_t intNum;
    uint_t errorCode;
    uint_t eip;
    uint_t cs;
    uint_t eflags;
};
```

系统使用 `ebx` 成员 `ecx` 成员记录调度策略和时间片长度，其中，`state->ebx` 成员用于指定调度策略，它的取值为 0 或 1，值为 0 代表系统采用时间片轮转调度策略，值为 1

则代表系统采用四级反馈队列调度策略，若取其他值则出错。State->ecx 成员用于记录相应调度策略下的时间片长度。时间片长度默认值是 4，用宏常量 DEFAULT_MAX_TICKS（在 timer.c 中定义）表示。若要实现可变时间片，将常量 MAX_TICKS 改为一个可以由系统调用来设置的全局变量就可以了。

11.3.3 进程调度策略评价

进程调度策略的优劣可以用进程的运行时间（一个进程从创建到结束完成所需要花费的时间）来衡量。进程的运行时间可以通过系统调用 Sys_GetTimeOfDay 得到。Sys_GetTimeOfDay 系统调用的功能是得到系统全局变量 g_numTicks 的值，这个变量已在内核实现，所以用户要做的是在进程开始和结束的时候分别调用 Get_TimeOfDay 读出 g_numTicks 的值，这样就能计算出进程运行所花费的时间，该时间中包含了进程交换、其他进程交替运行的时间等，所以计算得到的时间是进程的周转时间。

要对扩充后 GeekOS 系统的这两种调度算法进行评价，可以在两个算法中分别用不同的时间片长度运行应用程序 workload.exe 来测算。为尽可能减少尝试次数，可以输入下面的内容。

```
/c/workload.exe rr 1  
/c/workload.exe rr 100  
/c/workload.exe mlf 1  
/c/workload.exe mlf 100
```

用户要对运行结果进行分析，以进一步理解时间片长度与调度算法对进程运行所起的作用。

11.3.4 GeekOS 系统中的进程同步

1. 进程互斥

在 GeekOS 操作系统中，定义了 Mutex 数据结构以及 Mutex_Lock（加锁）和 Mutex_Unlock（解锁）操作用于临界区的互斥。

```
enum { MUTEX_UNLOCKED, MUTEX_LOCKED };  
struct Mutex {  
    int state;  
    struct Kernel_Thread* owner;  
    struct Thread_Queue waitQueue;  
};
```

在进程进入临界区前先执行加锁操作，退出临界区后执行解锁操作，如在 Lookup_Filesystem 函数中，在将文件系统链入链表时需要先对文件系统链表加锁，以实现互斥访问。需要注意的是：GeekOS 的 mutex 和条件变量 APIs，与系统屏蔽中断的作

用不完全相同。互斥不支持抢占，而且只有内核进程才能使用 mutex 和条件变量。

2. 利用 mutex 和条件变量实现临界区加锁、解锁的主要函数

1) Mutex_Wait 函数

函数功能：若临界区被锁，则进程等待。

```
static void Mutex_Wait(struct Mutex *mutex)
{
    KASSERT(mutex->state == MUXEX_LOCKED);
    KASSERT(g_preemptionDisabled);
    Disable_Interrupts();
    g_preemptionDisabled = false;
    Wait(&mutex->waitQueue);
    g_preemptionDisabled = true;
    Enable_Interrupts();
}
```

2) Mutex_Lock_Imp 函数

函数功能：为给定临界区加锁。

```
static __inline__ void Mutex_Lock_Imp(struct Mutex* mutex)
{
    KASSERT(g_preemptionDisabled);
    KASSERT(!IS_HELD(mutex));
    /*若临界区已被锁，则等待，直到临界区可用 */
    while (mutex->state == MUXEX_LOCKED) {
        Mutex_Wait(mutex);
    }
    mutex->state = MUXEX_LOCKED;
    mutex->owner = g_currentThread;
}
```

3) Mutex_Unlock_Imp 函数

函数功能：解锁给定临界区。

```
static __inline__ void Mutex_Unlock_Imp(struct Mutex* mutex)
{
    KASSERT(g_preemptionDisabled);
    KASSERT(IS_HELD(mutex));
    mutex->state = MUXEX_UNLOCKED;
    mutex->owner = 0;
    /*若有进程在等待使用该临界区，则唤醒等待队列的第一个进程*/
    if (!Is_Thread_Queue_Empty(&mutex->waitQueue)) {
        Disable_Interrupts();
```

```
    Wake_Up_One(&mutex->waitQueue);
    Enable_Interrupts();
}
}
```

4) Mutex_Init 函数

函数功能： 初始化互斥信号量。

```
void Mutex_Init(struct Mutex* mutex)
{
    mutex->state = MUTEX_UNLOCKED; //标记未锁
    mutex->owner = 0;
    Clear_Thread_Queue(&mutex->waitQueue); //初始化等待信号量队列空
}
```

5) Mutex_Lock 函数

函数功能： 调用 Mutex_Lock_Imp 函数给指定信号量加锁。

```
void Mutex_Lock(struct Mutex* mutex)
{
    KASSERT(InterruptsEnabled());
    g_preemptionDisabled = true;
    Mutex_Lock_Imp(mutex);
    g_preemptionDisabled = false;
}
```

6) Mutex_Unlock 函数

函数功能： 调用 Mutex_Unlock_Imp 函数给指定信号量解锁。

```
void Mutex_Unlock(struct Mutex* mutex)
{
    KASSERT(InterruptsEnabled());

    g_preemptionDisabled = true;
    Mutex_Unlock_Imp(mutex);
    g_preemptionDisabled = false;
}
```

7) Cond_Init 函数

函数功能： 初始化条件变量。

```
void Cond_Init(struct Condition* cond)
{
    Clear_Thread_Queue(&cond->waitQueue);
}
```

8) Cond_Wait 函数

函数功能：等待指定条件变量，使用 mutex 实现互斥。

```
void Cond_Wait(struct Condition* cond, struct Mutex* mutex)
{
    KASSERT(Interrupts_Enabled());
    KASSERT(IS_HELD(mutex));
    g_preemptionDisabled = true;
    Mutex_Unlock_Imp(mutex);
    Cond_Broadcast(); // 调用函数唤醒所有在等待条件变量的进程
    Disable_Interrupts();
    g_preemptionDisabled = false;
    Wait(&cond->waitQueue);
    g_preemptionDisabled = true;
    Enable_Interrupts();
    Mutex_Lock_Imp(mutex);
    g_preemptionDisabled = false;
}
```

9) Cond_Signal 函数

函数功能：唤醒正在等待条件变量的进程。

```
void Cond_Signal(struct Condition* cond)
{
    KASSERT(Interrupts_Enabled());
    Disable_Interrupts(); /* 阻止调用 */
    Wake_Up_One(&cond->waitQueue);
    Enable_Interrupts(); /* 唤醒调用 */
}
```

10) Cond_Broadcast 函数

函数功能：唤醒所有在等待指定条件变量的进程。

```
void Cond_Broadcast(struct Condition* cond)
{
    KASSERT(Interrupts_Enabled());
    Disable_Interrupts(); /* 阻止调用 */
    Wake_Up(&cond->waitQueue);
    Enable_Interrupts(); /* 唤醒调用 */
}
```

3. 信号量和 PV 操作

除了进程间可以实现互斥操作外，用户可以扩充 GeekOS 系统以支持信号量和 PV 操作。信号量和 PV 操作是实现进程同步最常用的同步机制，信号量（semaphore）是一

种特殊变量，GeekOS 系统中，每个信号量有一个信号量标志符 SID，SID 是一个 0~N-1 之间的整数，N 是系统能处理的信号量数目，用户在进行设计时系统至少要能处理 20 个信号量。信号量除 SID 外，还有名称（一般名称长度不能超过 25 个字符）、信号量初值、允许使用该信号量的进程个数、允许使用该信号量的进程列表、等待该信号量的进程队列等属性。GeekOS 系统中，已经创建的信号量都存放在一个链表中，创建信号量是使用 syscall.c 中的函数 Sys_CreateSemaphore 实现，函数原型为：

```
static int Sys_CreateSemaphore(struct Interrupt_State* state)
```

参数：state->ebx —— 存放信号量名的地址；

state->ecx —— 信号量名的长度；

state->edx —— 信号量初值。

在创建信号量时，函数首先用信号量名查找信号量链表，若该名字的信号量已经存在，则返回信号量的 SID；否则查看系统中已经创建的信号量个数是否已经达到系统能处理的信号量个数，若已达到容量上限，则返回一个负数，表示创建失败；若可以创建新信号量，就为信号量分配空间，并为信号量的属性 SID、名字、初值等赋值，最后函数返回 SID。

除赋初值外，信号量的值只能由 PV 操作改变。P 操作用于获得信号量，当执行 P 操作时，若信号量值小于等于 0，则进程阻塞，若信号量值大于 0，则信号量值减 1，进程继续运行。V 操作用于释放信号量，执行 V 操作时，将信号量的值加 1，若有进程在等待该信号量，则唤醒等待队列队首进程。在 GeekOS 中，文件/src/geekos/syscall.c 中定义了 Sys_P 和 Sys_V 两个系统调用，当执行这两个系统调用操作信号量时，内核首先检测进程是否有权限操作该信号量，若拥有操作权限，则调用相应函数获得/释放信号量，若没有权限则返回负数。

在/src/geekos/syscall.c 中，系统调用 Sys_DestroySemaphore 用于删除已用过的信号量，它首先查看有多少进程在使用该信号量，只有最后一个使用该信号量的进程才能使用 Destroy_Semaphore 删除该信号量。所以我们要确保进程退出时，即使没有调用 Destroy_Semaphore 函数，也要释放它所使用的全部信号量。

12.1 项目设计目的

了解虚拟存储器管理设计原理，掌握请求分页虚拟存储管理的具体实现技术。

12.2 项目设计要求

为了实现虚拟内存和页面调度，用户需要增加代码完成以下函数。

1. 在<src/geekos/paging.c>文件中编写代码完成以下函数。
 - `Init_VM()`(defined in)函数将建立一个初始的内存页目录和页表，并且安装一个页面出错处理函数。
 - `Init_Paging()`函数（定义在 `src/geekos/paging.c`）初始化操作页面调度文件所需的所有数据结构。就如前面说到的，`Get_Paging_Device()`函数指定分页调度文件定位在哪一个设备和占用磁盘块的地址范围。
 - `Find_Space_On_Paging_File()`函数应该在分页调度文件里面找到一个空闲的足够大的页空间。它将返回这个大块的索引，或者当没有合适的空间就返回“-1”。
 - `Free_Space_On_Paging_File()`函数将释放由 `Find_Space_On_Paging_File()`函数在分页调度文件里所分配的磁盘块。
 - `Write_To_Paging_File()`函数将把存储在内存的一页数据写出到分页调度文件里。
 - `Read_From_Paging_File()`函数将读取分页调度文件里的一页数据到内存空间。
2. 在<src/geekos/uservm.c>文件中编写代码完成以下函数，具体实现可以参考<src/geekos/userseg.c>文件中对应的函数代码。
 - `Destroy_User_Context()`释放进程所占用的所有内存和其他资源。
 - `Load_User_Program()`装载可执行文件到内存里，创建一个就绪

的用户地址空间，功能类似于分段系统的实现。

- `Copy_From_User()`从一个用户缓冲区复制数据到一个内核缓冲区。
- `Copy_To_User()`从一个内核缓冲区复制数据到一个用户缓冲区。
- `Switch_To_Address_Space()`利用它装载相应页目录和 LDT 来切换到一个用户地址空间。

12.3 项目设计提示

在一个系统中如果采用段页式存储管理方式，将有三种类型的地址：逻辑地址、线性地址和物理地址。逻辑地址是在程序编址时用到，通过段式系统逻辑地址被映射到线性地址：相关段的基地址加上逻辑地址得到线性地址。通过操作系统使用设置的页表，一个线性地址被映射到物理地址，物理地址是由处理器使用来确定读写物理内存单元的。分页系统和分段系统类似，因为它也允许每一个进程运行在它自己独立的存储空间中。然而，分页系统允许控制更细粒度的单元，即一页而不是一个固定偏移区间的段。

在 GeekOS 系统中，分页系统将使用一个页目录表和页表，即两级页表机制。在 Intel x86 系统中，每一个地址是一个 32 位长的数据。要转换一个线性地址到物理地址，处理器将访问当前进程的页目录表（这类似于为一个进程找到一个局部描述表）。处理器然后使用地址数据的最高 10 位（31 位~22 位）的值索引到页目录表（10 个二进制位允许定义 1024 个页目录表项），相应的表项内容是存放了相应页表的地址。处理器接着用地址的高 10 位（21 位~12 位）索引到页表（10 个二进制位同样可定义 1024 个页表项）。最后，相应页表项的内容是地址单元所在的逻辑页存放的物理页的基地址。线性存储地址的低 12 位（11 位~0 位）被用来索引到物理地址（12 个二进制位可以定义 4096 个字节物理单元）。

在系统中每一个进程页目录和页表是他们自己地址空间中的页。每一个页目录表项和每一个页表项是 4 个字节的长度，每一个表包含 1024 个表项，即大小都为 4096 个字节。每一个页目录项包含指向一个页表的指针，而每一个页表项包含指向一个物理页的指针。

12.3.1 为内核程序空间建立页表

本项目的第一步将修改用户的项目使用页表和段，而不仅仅是分段来实现内存保护。以下这些工作是由文件/`src/geekos/paging.c` 中的函数 `Init_VM` 来完成，这需要用户填充代码来完成。

内核开始运行时并没有采用分页管理，只采用分段，硬件直接映射逻辑地址到物理地址。增加分页管理后，在它们之间引入了另外一级映射：一个逻辑地址通过分段映射到一个线性地址，然后再由分页系统把线性地址映射到物理地址。因此，分页系统的第一步是引入一个页表给所有线性地址建立映射，而每一个线性地址将由页表来映射到物理地址。

为了建立页表，用户将需要先分配一个页目录表（通过 `Alloc_Page` 函数分配），然

后再为整个存储区间的内容分配页表。用户将需要填充页目录表项和页表项相应的内容。页目录表项和页表项数据结构在 `paging.h` 文件中已经定义（在第 6 章已经介绍 `pte_t` 和 `pde_t` 结构体）。对于内核的存储空间对应的表项应该标识为非 `VM_USER`，以至于只有核心程序访问它们才是合法的。最后，如果系统是第一次使用分页系统，用户将需要调用函数 `Enable_Paging(pdbr)` 使分页系统有效。这个函数在 `/src/geekos/lowlevel.asm` 文件中定义，它以用户设置的页目录表的基地址作为参数来运行。

函数 `Init_VM` 最后一步是加入一个缺页中断处理程序，在 `/src/geekos/paging.c` 文件中系统默认定义它的名字为 `Page_Fault_Handler`。用户应该通过调用 `Install_Interrupt_Handler` 函数（在文件 `idt.c` 中定义）来安装这个中断处理程序，并且需要注册这个中断处理程序的中断号为 14。完成 `Init_VM` 函数语句后，用户应该在 `main.c` 文件中增加一个 `Init_VM` 函数调用语句，为分页系统的使用作准备，但它应该放在 `Init Interrupts` 函数调用语句之后。

12.3.2 为用户进程建立页表

本项目第二步将是让用户进程拥有它们自己的线性地址空间。

系统为所有内核级线程只定义一个页目录表，而为每一个用户级进程分别建立一个页目录表。前面所设置的页目录表和页表是被内核所用，用户应该改变标志域为不包括 `VM_USER`。

对于用户级进程的页目录表将包含映射用户逻辑存储空间到物理存储空间的入口，也将包含到内核存储空间的入口。这样的设计不是用户进程可以直接访问内核存储空间（系统将设置存储空间的访问标志来阻止这样做），而是当用户进程需要访问内核地址空间时，将引起一个中断产生。当中断产生时它将简单使用正在运行的用户进程的页表，即不需要改变页表就可以执行内核程序访问到自己的存储单元。

一个用户进程的存储格式如图 12-1 所示，图中左边的地址是线性地址（通过分页系统它被映射到物理地址）。用户进程仍然引用逻辑地址，它被分段系统映射到图中的线性地址。为此，一个用户进程仍然将有和前面项目同样的地址。



图 12-1 用户进程的线性地址空间格局

下一步是修改用户进程在用户的存储区域使用分页。为了帮助用户做到这些，有一个新的文件调用 uservm.c 来替换先前项目使用的文件 userseg.c，它们中定义的函数名称相同，要完成的功能也是一样的，不同的是 userseg.c 中是实现分段系统，而 uservm.c 是实现分页系统。项目开始时，可以先把 userseg.c 文件中实现的函数复制到 uservm.c 文件中，然后再修改它们实现分页管理。在项目 4 的 build 目录下的 makefile 文件中有一行用来指定是使用 userseg.c，还是使用 uservm.c。用户能通过修改 USER_IMP_C := uservm.c 为 USER_IMP_C := userseg.c 在它们之间切换，反之亦然。

为用户进程建立分页系统是在 usrevm.c 文件中的 Load_User_Program 函数分两步实现。首先需要为进程分配一个页目录表，复制线性存储空间的低 2GB 的所有核心页目录表项（在 Init_VM 函数中设置的）到用户进程的页目录表。

接下来需要为用户进程的代码、数据和堆栈区间分配页表项。三个区间中的每一个将由不同数目的页组成，这些页是由函数 Alloc_Pageable_Page 来分配的。这个函数不同于 Alloc_Page 函数，Alloc_Pageable_Page 函数分配的页将返回一个特殊标志 PAGE_PAGEABLE，它被设置在页数据结构表项的标志域（在 mem.h 文件中定义）。除了页目录表和页表对应的页，进程所占用的其他所有页都应该使用这个函数来分配。

前面项目实现的分段系统和分页系统不同的是：对于分段系统，一个连续的内存空间被分配给整个用户进程存储空间，而在分页系统分配给进程的内存空间可以是不连续的页。

建议用户为用户进程建立分页系统用如下操作：先计算进程的数据段和代码段的大小，不需要包括进程的堆栈段，但大小必须是 PAGE_SIZE 的倍数，然后以页为单位，分配存储空间给进程；接着一页一页地复制进程映像到新分配的页来装入程序到用户的地址空间，每复制一页在页表中建立一个相应表项；最后在进程虚拟地址空间尾部分配两页的存储空间（也就是在页表的最后安排两个表项），一个是为参数块设置的，另一个是为堆栈设置的。设置表项时确保页目录表和页表项的标志位设置为用户可以访问模式（包含 VM_USER 标志）。

最后，用户需要修改一些当前分段系统使用的一些数据以至系统能正常使用分页。用户模式进程的线性地址的基地址应该是 0x8000 0000（常量 USER_VM_START 在 paging.h 文件中已定义），界限地址应该为 0xFFFF FFFF（常量 USER_VM_END 在 paging.h 文件中已定义）。用户也将需要增加代码来实现切换 PDBR 寄存器的内容。为了实现这个功能，在装入 LDT 之后，函数 Switch_To_Address_Space 应该加入一个 SET_PDBR（已经在 lowlevel.asm 文件中定义）调用作为上下文切换的一部分。用户将使用 usercontext 数据结构的 pageDir 域来存储进程页目录表的地址。

到此为止，用户可以通过运行 bochs 测试分页存储系统。如果能装入并运行 shell 用户程序，就证明已经正确完成了分页系统。

12.3.3 请求分页技术实现

分页系统的一个优点就是可以简单地分配任意一个新页直接加到一个进程的存储空间。例如，它允许进程的堆栈空间动态增长超出初始化分配的空间。为了实现堆栈动态增长，必须修改缺页中断处理程序。当进程尽力访问一个非法地址时，这个处理程序

被调用，当前它仅仅是终止进程执行，打印出引起缺页的地址。缺页处理程序通过读取 CR2 寄存器的内容决定引起错误的地址，它也打印数据结构 `Interrupt_State` 中 `errorcode` 的值和数据结构 `faultcode_t` 的 `fault` 的值，它们在 `/src/include/paging.h` 文件中定义。

用户也需要修改系统提供的缺页处理程序来决定基于错误地址系统应该做什么相应处理。如果错误的地址是在当前堆栈限制的页内，用户应该分配一个新的页，映射它到合适的地址，然后从出错处理程序正常返回。程序现在将能使用用户刚分配给它的存储空间。为了测试新的缺页处理程序，可以运行项目 4 提供的用户程序 `rec.c` 来进行测试。

当分页系统被使用来提高存储管理效率时，活动的进程仍然要受到物理存储空间大小的限制。为了弥补这个缺点，用户将需要实现虚拟存储作为 GeekOS 的一部分，虚拟存储技术可以把内存里的页暂时存储到磁盘上，以至于释放物理存储空间供其他进程使用。为了做到这一点，操作系统将需要在磁盘设备上创建一个 `page file` 文件暂时保存从内存中替换出去的页，和实现一个类 LRU 算法在内存中选取一个替换页把它写到磁盘的 `page file` 文件中。类 LRU 算法可以使用页表项的访问位来记录相应的页被访问的频繁度。为了实现这个算法，需要使用页表项数据结构的 `clock` 域（在 `mem.h` 文中定义）。用户应该在每一次缺页处理时把所有被访问的页的 `clock` 值修改。

`page file` 文件是由连续的 512 个磁盘块组成，每一块大小为 512B。通过调用 `vfs.h` 文件中的函数 `Get_Paging_Device` 将返回一个 `Paging_Device` 数据结构，它包含了 `page file` 的第一个磁盘块号和 `page file` 文件中的磁盘块数。每一个页由 8 个连续的磁盘块组成。要读写 `page file`，使用 `/src/include/blockdev.h` 文件提供的 `Block_Read` 和 `Block_Write` 函数。这些函数每一次读/写 512 个字节。如何管理 `page file` 完全由用户自己确定。一个好的思想是在 `paging.c` 文件中写一个 `Init_Pagefile` 函数，在 `main.c` 文件中调用它来执行。

在 `/src/geekos/mem.c` 文件中，已经定义了一个函数 `Alloc_Pageable_Page` 实现交换一页到磁盘的操作，具体执行步骤如下。

- 调用 `mem.c` 文件中已经实现的 `Find_Page_To_Page_Out` 函数来确定要替换的页（这个函数依赖于页数据结构中的 `clock` 域）。
- 调用 `paging.c` 文件中已经实现的 `Find_Space_On_Paging_File` 函数在 `page file` 中找到空闲的存储空间。
- 调用 `paging.c` 文件中已经实现的 `Write_To_Paging_File` 函数把被替换的页写到 `page file` 文件中。
- 修改页表的相应表项，清除页存在的标志，标识为此页在内存中不存在。
- 修改页表项的页基地址为包含这一页的第一个磁盘块号。
- 修改页表项的 `kernelInfo` 位标识为 `KINFO_PAGE_ON_DISK` 状态（标识这一页是在磁盘上存在，而不是没有有效）。
- 调用 `lowlevel.asm` 文件中已经实现的 `Flush_TLB` 来刷新 TLB。

最后，这一页被保存在磁盘，进程什么时候再需要时，再把它调进内存。由于这一页是在 `page file` 中存在，它所对应的页表项标识此页是不存在的，当访问它时系统将引起一个缺页中断。用户设计的缺页处理程序应该能够认识到此页是在 `page file` 中存在，并把它从磁盘读进内存。而当用户从磁盘调入一页进内存，需要释放这一页占用的磁盘空间。表 12-1 总结了缺页处理程序应做的工作。

表 12-1 缺页处理表

缺页情况	标识	相应处理
堆栈生长到新页	超出原来分配一页的限制	分配一个新页进程继续
此页保存在磁盘上	数据标识这一页在 page file 中存在	从 page file 读入需要的页继续
因为无效地址缺页	非法地址访问	终止用户进程

12.3.4 进程终止处理

作为进程终止处理的一部分，用户将需要释放进程原来占用的内存空间。这包括释放进程的页，页表和页目录表对应的内存空间。另外，用户也需要释放进程终止时使用后备存储空间。为此用户需要修改/src/geekos/uservm.c 文件中的 Destroy_User_Context 函数来完成这些工作。

12.3.5 系统完善处理

在 GeekOS 里所实现的是一个相当简单的虚拟内存管理，可以通过很多种方法来扩展和改进。

1. 改进 Find_Page_To_Page_Out()

当系统需要一内存页但却没有可利用的页，内核可以使用 Find_Page_To_Page_Out()函数（in src/geekos/mem.c）找到一可以换出的一个页面。在禁止中断的情况下（意味着没有其他线程和中段处理运行），为了找到要替换的一个合适页面，这个函数遍历整个页数据结构队列。

如何使这个函数运行的效率更高呢？

2. 增加一个用户堆

当前，GeekOS 内核只分别创建一个代码段、数据段和用户进程栈。但它没有创建一个用户堆，这就意味着用户进程没有动态分配内存的权限（使用类似于 malloc()函数的功能）。

为了在 GeekOS 里增加支持使用用户堆，需要做几件事情。

- 内核需要通告 C 函数库里的函数 Entry()(src/libc/entry.c)堆空间的开始地址和最大的堆区间，同时希望允许用户进程能申请扩展堆，在任何情况下 Entry()需要初始堆区间。
- C 函数库需要实现分配跟释放内存的函数，复制 src/geekos/bget.c 文件到 C 函数库里面，分别使用函数 bpool()、bget()和 brel()初始化堆、分配内存、释放内存。为了增加新代码文件到 C 函数库里面，必须把它们放到 src/libc 目录下，并且增加它们到 build/Makefile 文件中定义的 LIBC_C_SRCS 这个宏中。
- 页出错处理程序需要增加在堆空间负责内存存取分页。

13.1 项目设计目的

了解文件系统的设计原理。掌握操作系统文件系统的具体实现技术。

13.2 项目设计要求

为实现 GOSFS 文件系统，在/src/geekos/gosfs.c 中必须实现以下列出的函数。

GOSFS_Fstat()函数：为给定的文件得到元数据。

GOSFS_Read()函数：从给定文件的当前位置读数据。

GOSFS_Write()函数：从给定文件的当前位置写数据。

GOSFS_Seek()函数：在给定文件中定位。

GOSFS_Close()函数：关闭给定文件。

GOSFS_Fstat_Directory()函数：为一个打开的目录得到元数据。

GOSFS_Close_Directory()函数：关闭给定目录。

GOSFS_Read_Entry()函数：从打开的目录表读一个目录项。

GOSFS_Open()函数：为给定的路径名打开一个文件。

GOSFS_Create_Directory()函数：为给定的路径创建一个目录。

GOSFS_Open_Directory()函数：为给定的路径打开一个目录。

GOSFS_Delete()函数：为给定的路径名删除一个文件。

GOSFS_stat()函数：为给定的路径得到元数据（大小、权限等信息）。

GOSFS_Sync()函数：对磁盘上的文件系统数据实现同步操作。

GOSFS_Format()函数：格式化 GOSFS 文件系统操作。

GOSFS_Mount()函数：挂载文件系统操作。

13.3 项目设计提示

13.3.1 GOSFS 磁盘格式

GOSFS 文件系统是项目 5 要求用户完成的文件系统，它支持多级文件目录结构和长文件名，提供文件与目录的创建、删除等基本操作。在 Bochs 模拟器中，该文件系统驻留在二级存储设备 Ide1 硬盘上，IDE 硬盘镜像名为 diskd.img，默认大小为 10MB。GOSFS 按磁盘块为单位对磁盘进行划分，每个磁盘块大小为 4KB（IDE 硬盘扇区大小为 512 字节，所以一个磁盘块有 8 个扇区），磁盘布局如图 13-1 所示。

其中，第 0 块称为超级块（SUPERBLOCK），用于保存 GOSFS 文件系统的重要数据（第 1 块往后都用于保存文件和目录）。在超级块最起始的标记为 magic（魔数）的 4 字节必须存放 0xDEADBEEF、GOSF 或类似数据的，用于表示这是 GOSFS 文件系统。在加载 GOSFS 文件系统时若没有发现这个标记将返回错误。超级块的根目录指针标记为 Root Dir Pointer 的 4 字节用于保存存放根目录的磁盘块编号，标记为 size 的 4 字节用于记录磁盘的大小（磁盘大小按磁盘块数计算，每 4KB 为一块，那么 32MB 的磁盘就记录为 $32\text{MB} / 4\text{KB} = 8192$ ）。标记为 Free Blocks Bitmap 的 1024 字节用于空闲磁盘块的管理，在位图中每一位就代表一个 4KB 大小的磁盘块，1024 个字节的位图可以管理 $1024 \times 8 \times 4\text{KB} = 32\text{ MB}$ 的磁盘（实现位图管理的相关函数在 bitset.c 中，见本章后面的介绍）。

```
struct GOSFS_Superblock{
    struct GOSFS_Instance gfsInstance;
    struct Condition cond;
    struct Mutex lock;
    ulong_t flags;
    uchar_t dirty;
};
```

超级块数据结构在 gosfs.h 中定义如下：

```
struct GOSFS_Instance{
    ulong_t magic;           // 文件系统魔数
    ulong_t numLogicBlocks; // 逻辑块的数量
    uint_t numInodes;        // 信息结点数量
    ulong_t firstDataBlock; // 第一个数据块的块号
    struct Block_Device * dev; // 所属的块设备指针
    uchar_t inodeBitmapVector[GOSFS_NUM_INODE_BITMAP_BYTES];
    // 信息结点位图
    uchar_t blockBitmapVector[GOSFS_NUM_BLOCK_BITMAP_BYTES];
    // 数据块位图
};
```

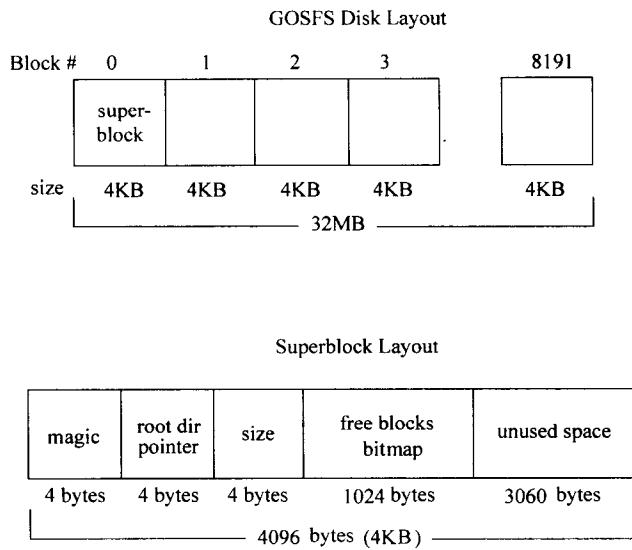


图 13-1 GOSFS 磁盘布局

超级块的信息在文件系统格式化时填写, GOSFS 文件系统中由函数 `GOSFS_Format()` 进行格式化, 函数首先获取磁盘容量, 调用函数 `Get_Num_Blocks()` 转换成磁盘块数, 然后计算出维护空闲磁盘块所用的位图向量的大小, 并将位图相应位都置为空, 然后为文件系统创建一个有效的空目录, 即根目录, 并使 `Root Dir` 指针指向该目录, 再接着将相关数据填入超级块, 然后在位图中将根目录使用的磁盘块标记为正在使用, 最后为标记为 `magic` (魔数) 的 4 个字节赋值。这样就可以加载和使用 GOSFS 格式的磁盘了。

13.3.2 文件与目录

GOSFS 中的文件可以分为 4 种: 标准输入文件、标准输出文件、文本文件和目录, 其中标准输入与标准输出文件分别映射到键盘与显示器, 而文本文件就是一般可以由用户创建、编辑和删除的普通文件。目前 GOSFS 不支持用户可执行文件的存储, 所有可执行文件都存放在 PFAT 文件系统中。

GOSFS 将目录作为特殊的文件进行管理, 在系统中每个目录或文件都对应一个 `GOSFS_Dir_Entry` 结构体, 结构体中保存了目录或文件的相关信息, 如名称、大小等。

```
struct GOSFS_Dir_Entry {
    ulong_t size;
    // 文件大小
    ulong_t flags;
    // 标记文件正在使用或该文件为目录等
    char filename[GOSFS_FILENAME_MAX+1];
    // 文件名
    ulong_t blockList[GOSFS_NUM_DIR_ENTRY];
```

```
// 文件数据块表或目录文件表
struct VFS_ACL_Entry acl[VFS_MAX_ACL_ENTRIES];
// ACL表项，保留未使用
};
```

常量 GOSFS_FILENAME_MAX 等于 127，表示 GOSFS 文件系统支持的最长文件名为 127 个字符，常量 GOSFS_NUM_DIR_ENTRY 等于 10。若节点表示一般文件，blockList 数组（其布局如图 13-2 所示）中记录文件数据块的指针，GOSFS 采用直接块与间接块组合的方式存储数据块号，目前 GOSFS 能支持二级间接块，blockList[0] 到 blockList[7] 前八个数组单元保存文件的前八个直接块指针，blockList[8] 保存一个一级间接块指针，blockList[9] 保存一个二级间接块指针。一级间接块指针指向的块保存了 1024 个直接块号，而二级间接块指向的块保存了 1024 个一级间接块，这样共计每个文件可以拥有最多 1049608 个数据块，而以每个数据块 4KB 计算，理论上可以给一个文件分配的磁盘空间大约 4GB。

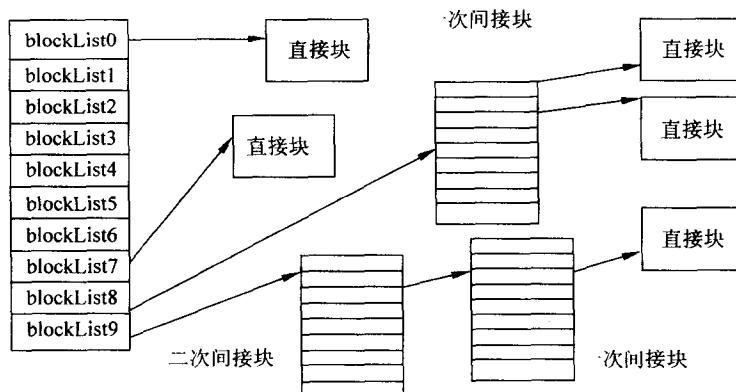


图 13-2 blockList 数组布局

13.3.3 GOSFS 文件系统数据结构和操作

1. 重要数据结构

```
// 文件系统结构定义
struct Filesystem {
    struct Filesystem_Ops *ops; // 文件系统的格式化与加载操作指针
    char fsName[VFS_MAX_FS_NAME_LEN + 1]; // 文件系统名称
    DEFINE_LINK(Filesystem_List, Filesystem); // 文件系统列表指针
};

// 加载点
struct Mount_Point {
    struct Mount_Point_Ops *ops; /* 指向 Mount_Point 支持的操作 */
```

```

char *pathPrefix;           /* 文件系统加载的路径 */
struct Block_Device *dev;   /* 文件系统加载的块设备 */
void *fsData;
DEFINE_LINK(Mount_Point_List, Mount_Point);
};

// 文件结构, 在vfs.h中定义
struct File {
    struct File_Ops *ops;          // 支持的文件操作指针
    ulong_t filePos;               // 文件当前位置
    ulong_t endPos;                // 文件末尾位置
    void *fsData;                  // 实体文件节点指针
    DEFINE_LINK(VNode_List, File); // 虚拟文件节点链表指针
    int mode;                      // 文件模式(只读、可写等等)
    struct Mount_Point *mountPoint; // 文件属于的文件系统加载点
};

```

2. 重要操作函数

1) Filesystem 有关的操作函数

与文件系统有关的主要操作有格式化和加载, GOSFS 中对应的操作由 `GOSFS_Format()` 和 `GOSFS_Mount()` (在 `gosfs.c` 中定义)。

`GOSFS_Format` 操作用于文件系统的格式化, 以至于它能被安装到 GeekOS。在编译 GeekOS 操作系统时, 会自动生成 10 兆大小的磁盘镜像, 并初始化为全零。`GOSFS_Format` 将这种原始磁盘格式化为 GOSFS 格式, 格式化前, `GOSFS_Format` 通过超级块中的魔数检查磁盘是否已格式化为 GOSFS 格式, 若磁盘已是 GOSFS 格式则不执行操作。

`GOSFS_Mount` 操作负责将一个 GOSFS 格式的磁盘加载到操作系统, 即在所有文件系统的命名空间里设置一个包含特定路径前缀的块设备。该操作首先初始化内存中的 `GOSFS_Superblock`, 之后初始化 `Mount_Point`, 最后创建一个空目录, 即 GOSFS 根目录 `d`。

2) Mount_Point 有关的操作函数

`Mount_Point` 数据结构代表被安装文件系统的一个对象。在 GeekOS 中, 文件系统被挂载到一个路径前缀上, 它说明挂载的文件系统将成为整个文件系统名字空间的一部分, 比如, 包含用户执行文件的 PFAT 文件系统通常是在挂载在/c 的路径前缀上。每一个挂载的文件系统利用一个块设备作为它的基本存储空间。

`Mount_Point` 对象支持以下几个操作:

- `Open()` 操作在挂载文件系统里打开一个文件。
- `Create_Directory()` 操作在挂载文件系统里面创建一个目录。
- `Open_Directory()` 操作在挂载文件系统里面打开一个目录, 以致于能够通过 `Read_Entry()` 操作读取它的入口。
- `Stat()` 操作在挂载文件系统里面为一个已经命名的文件恢复文件元数据, 比如大小与文件权限。
- `Sync()` 操作刷新所有存储在内存的, 还没有写入磁盘的文件数据和文件系统的元

数据。这是一个必不可少的操作，因为文件系统通常在内存缓冲区里暂时保存数据，过了一段时间以后才把修改过的数据写入磁盘。在完成 Sync() 操作后，保证所有在文件系统的数据被写入磁盘。

文件系统给所有 Mount Point 操作时设置了路径前缀，但在这些操作调用前被删除。所以如果一个 GOSFS 文件系统是在挂载到路径前缀/d 上的，文件/d/stuff/foo.txt 被传递给高层次的函数 Open()，而传递给 GOSFS_Open() 函数的路径将是/stuff/foo.txt。

3) File 有关的主要操作函数

文件数据类型代表一个进程所打开的文件或者目录。文件对象是由 Open() 和 Open_Directory() Mount Point 操作所创建。

通常文件对象保持一个文件的当前位置，所有读和写的执行是相对于文件的当前位置来进行。

- GOSFS_Open 打开已存在文件或新建文件。

```
static int GOSFS_Open(struct Mount_Point *mountPoint,
                      const char *path, int mode, struct File **pFile);
```

参数 mountPoint 是超级块指针，path 是打开文件的路径，mode 是打开方式，pFile 是指向文件结点的二级指针。若操作需要创建文件则调用 GOSFS_Create_File，否则调用 GOSFS_Open_File。

- GOSFS_Create_Directory 用于创建目录。
- GOSFS_Open_Directory 用于打开目录。
- GOSFS_Stat 用于获得文件结点信息。
- GOSFS_Sync 用于高速缓冲区块的同步；调用该函数后，会将缓冲区中所有标记为 dirty 的块立即写回磁盘，以保证缓冲区与磁盘之间数据一致性。
- GOSFS_Seek 用于改变文件位置指针 filePos。
- GOSFS_Close 用于关闭文件或目录的工作。
- GOSFS_Read 用于读文件。
- GOSFS_Write 用于写文件。

4) GOSFS_Dir_Entry

GOSFS_Dir_Entry 数据结构存储在磁盘里，代表一个独立的目录项。它包括几个域。

- size 域里，存储着文件的大小，或者存储涉及到项的目录。
- flag 域保存几个布尔变量标志。GOSFS_DIRENTY_USED 标志位表示目录项已经被使用。如果没有设置，就意味着项是可用的。GOSFS_DIRENTY_ISDIRECTORY 标志位表明是目录项指向子目录。GOSFS_DIRENTY_SETUID 标志表示目录项指向一个 setuid 的可执行文件。如果文件是可以执行的，那么创建新的进程将会有一个同样的用户 ID 作为文件的拥有者（不需要对这个位做任何事情直到下一个新的项目）。
- filename 域存储文件名，包括一个空的终端名称字符空间。
- blocklist 域存储直接的、间接的、双重间接的块号，表示给文件或者目录所分派

的存储空间。

- 最后，acl 域存储文件或者目录的访问控制列表项的列表。第一个项指定了文件所有者的入口。

5) 高速缓冲存储器

在实现文件系统的过程中，系统将会频繁地从文件系统读取数据到内存，某些时候也需要把内存中的数据写回到文件系统。大多数的操作系统内核利用一个高速缓冲存储器提高对文件的读写操作。这个高速缓冲存储器包含与具体文件系统通信的多个内存缓冲块。为了从文件系统读取一个数据块，内核必须先从高速缓冲区请求一个块。为了写一个块到文件系统中，要先修改在缓冲区中的数据，并且此缓冲区块被标记为脏块。在后来的某个时候，缓冲区的脏块将会被操作系统写回到磁盘文件系统。

在这个项目里，系统已经为大家提供了一个已实现的高速缓冲区，可以通过包含头部文件`<geekos/bufcache.h>`来使用。一个 `Buffer_Cache` 数据结构表示一个特定文件系统对象的高速缓冲区。通过传递一个块设备给函数 `Create_FS_Buffer_Cache()` 可以创建一个新的高速缓冲区。一个 `FS_Buffer` 数据结构表示一个包含某个文件系统块数据的缓冲区。通过调用 `Get_FS_Buffer()` 函数为一个特殊块申请缓冲区。装载数据的实际内存是通过数据区 `FS Buffer` 指向的，如果在一个缓冲区里修改数据，必须调用 `Modify_FS_Buffer()` 函数，让高速缓冲区知道缓冲区现在的数据是“脏”的。当完成存取数据或者修改缓冲区的数据后，就利用 `Release_FS_Buffer` 函数释放缓冲区。

缓冲区的存取是互斥进行的。在同一时刻只有一个线程能使用缓冲区。如果一个线程正在使用一个缓冲区，而另一个线程请求使用同一个缓冲区（调用 `Get_FS_Buffer()` 函数），那么第二个线程被阻塞，直到第一线程释放缓冲区（调用 `Release_FS_Buffer()`）。注意不要在同一进程连续两次调用 `Get_FS_Buffer()` 函数，因为这样会导致死锁。

GeekOS 高速缓冲区对于“脏”的缓冲区，采用延迟写入到磁盘的方式，如果想立即把缓冲区的内容写入到磁盘，就要调用函数 `Sync_FS_Buffer()`。当内容中包含文件系统等元数据，比如目录或者磁盘分配位，采取把修改的文件系统缓冲区内容马上写回磁盘是一种好的思想，可以避免由操作系统崩溃或者计算机突然关机等情况所导致的文件系统出现严重错误的可能性。通过调用 `Sync_FS_Buffer_Cache()` 函数能够清除高速缓冲区中所有脏的数据，实现 `Sync()` 函数对于挂载文件系统是相当有利的。

`Destroy_FS_Buffer_Cache()` 函数释放一个高速缓冲区，先要清除所有“脏”磁盘缓冲区。这对于实现 `GOSFS_Format()` 函数比较有用。

6) 启动

实现一个文件系统相当复杂。这一节为用户提供几点如何实现此项目的建议。

首先，实现函数 `GOSFS Format()` 时，用户可能需要创建一个临时的 `FS_Buffer_Cache` 对象用来执行 I/O 操作。

第二，实现函数 `GOSFS_Mount()` 时，这个函数用来得到一个 `Mount_Point` 对象，它包含一个指向文件系统映像所在块设备的指针。用户可能需要创建一个辅助的数据结构，它存储在挂载点里。使用这个数据结构存储指向高速缓冲区的对象和其他任何执行 `Mount_Point` 所需的信息，比如打开一个文件。用户可能需要把指向辅助数据结构的指

针存入到 Mount_Point 对象的 FsData 域中。

通过运行下面的 GeekOS 指令可以测试格式化和挂载 GOSFS 文件系统的操作：

```
$ format ide1 gosfs
$ mount ide1 /d gosfs
```

一旦格式化并且挂载了 GOSFS 文件系统，就能开始运行 Mount_Point 函数。正常开始的地方是 GOSFS_Open() 的实现。当在模式标志位里设置 O_CREATE 位时，就启动函数 GOSFS_Open()，这时候就会创建一个新文件。当使用 GOSFS_Mount() 时，可能要在创建的文件对象里存储一个临时数据结构，可以使用 fsData 区来实现这个目的。用户可以使用 touch.exe 程序来测试文件的创建。用户需要实现函数 GOSFS_Create_Directory()，同样可以使用程序 mkdir.exe 来进行测试目录的创建。

一旦创建了文件和目录，就可以开始实现文件操作了。一般是从 GOSFS_Close() 函数开始。最后，用户需要实现更复杂的函数，比如 GOSFS_Read()、GOSFS_Read_Entry() 和 GOSFS_Write() 函数。

7) 结论

这一小节是讨论在实现这个项目时所必须考虑的一些问题。

- 同步

多个进程能够同时访问挂载点、文件、目录。因此，需要为文件系统的数据结构设置一个锁机制，以确保它们能够被多个线程和多个进程安全访问。在这个项目中，用户可以使用一个简单的方法，比如使用互斥锁来保护整个文件系统实体。

- 文件共享

在<geekos/vfs.h>文件里定义文件和目录对象的数据结构，它们可以被进程打开访问。如果两个进程同时打开一个文件，两个进程有不同的文件对象。这是必须的，因为两个进程可能拥有不同的文件位置。如果一个进程执行读操作或者定位操作，那么它不能影响到其他的进程。然而，两个文件对象指向同一个文件，其中的一个进程改变了文件，那么另一个进程必须是可见的。这就意味着，必须安排两个 VFS 文件对象有一个指向真实文件，共享数据结构的指针。这个策略如图 13-3 所示。

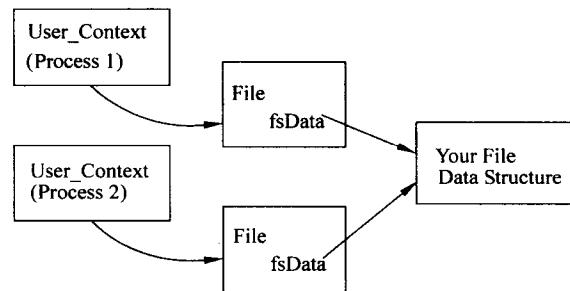


图 13-3 文件共享方式

因为支持多级内部文件数据结构，可以保存一个使用这个文件的进程计数值，当没有使用者时，就可以清零。