高等学校计算机实践教学系列教材

Linux操作系统实验教程

费翔林　主编
李　敏　叶保留　编著

高等教育出版社

高等学校计算机实践教学系列教材

Linux操作系统实验教程

本书特色

- 操作系统是理论性与实践性并重的课程，本书力求做到课程教学与实验教学彼此呼应、原理讲授与实验环节紧密结合，掌握基本理论与提高编程能力相互并重。

- 本书注重总体规划的科学性与合理性、实验环节的实用性与可操作性；实验设计紧扣基本原理与核心技术，实验内容涵盖用户空间编程与内核设计分析；实验安排循序渐进、层层渗透，实验形式丰富多样、富有启发性。

- 书中每个(组)实验包含实验目的、背景知识、实验内容、解决方案和程序框架等内容。通过实验把操作系统基本原理与Linux实现机制联系起来，以此激发学生的实验兴趣，将所学知识融会贯通和综合运用，进一步提高学生分析问题和解决问题的能力。

- 本书将同时提供配套的电子教案及所有已通过验证的实验题参考源码，为任课教师提供相应的技术支持。

ISBN 978-7-04-026294-0

9 787040 262940 >

定价 35.00 元

高等学校计算机实践教学系列教材

Linux 操作系统实验教程

费翔林　主编
李　敏　叶保留　编著

高等教育出版社

内容提要

学习操作系统的最好途径是理论和实践相结合，本书是操作系统实验课程教材，以 Linux 2.6 内核版本为平台，精心设计系列实验题目，每个(组)实验题目包括：实验目的、背景知识和实验内容，在每个具体的实验内容中又包括实验说明、解决方案和程序框架，为操作系统实践教学提供指导。

本书内容丰富、覆盖面广，由浅入深、循序渐进，可与高等教育出版社出版的《操作系统教程(第 4 版)》教材配套使用，也可作为操作系统课程的实验教材单独使用，既可以作为高等学校计算机相关专业实验课用书，也可作为 Linux 应用和内核编程参考资料。

图书在版编目（CIP）数据

Linux 操作系统实验教程/费翔林主编. —北京：高等教育出版社，2009.4
ISBN 978-7-04-026294-0

Ⅰ. L… Ⅱ. 费… Ⅲ. Linux 操作系统-教材
Ⅳ. TP316.89

中国版本图书馆 CIP 数据核字（2009）第 045739 号

策划编辑	倪文慧	责任编辑	郭福生	封面设计	于文燕	责任绘图	尹 莉
版式设计	王 莹	责任校对	王 超	责任印制	韩 刚		

出版发行	高等教育出版社	购书热线	010-58581118
社 址	北京市西城区德外大街 4 号	免费咨询	800-810-0598
邮政编码	100120	网 址	http://www.hep.edu.cn
总 机	010-58581000		http://www.hep.com.cn
		网上订购	http://www.landraco.com
经 销	蓝色畅想图书发行有限公司		http://www.landraco.com.cn
印 刷	北京中科印刷有限公司	畅想教育	http://www.widedu.com
开 本	787×1092 1/16	版 次	2009 年 4 月第 1 版
印 张	30	印 次	2009 年 4 月第 1 次印刷
字 数	680 000	定 价	35.00 元

物料号 26294-00

前　言

操作系统是计算机系统的核心和灵魂，是计算机系统必不可少的组成部分，也是计算机专业教学的重要内容。该课程概念众多、内容抽象、灵活性与综合性强，不但需要讲授操作系统的概念和原理，还需要加强操作系统实验，上机进行编程实践，这样才能让学生更好地掌握操作系统的精髓，真正做到深刻理解和融会贯通。人们已充分认识实践环节在学好操作系统过程中的重要性。实践是操作系统课程教学活动的重要环节，但实验教学难度大，由于实用操作系统的应用环境约束性强，实验环节的可操作性差，而且模块众多、盘根错节、技术细节剖析困难，动手及实践能力很难培养。编写本实验教程是为了在学习操作系统基本原理的同时，强调理论联系实际，为操作系统实验教学提供一定的指导和帮助。

操作系统实验教学的基本目的和要求：

(1) 学生应该通过实验加深理解和更好地掌握操作系统的基本概念、原理、技术和方法，巩固所学知识，激发实验兴趣，掌握实验要领，培养对操作系统课程所学知识融会贯通和综合运用的能力。

(2) 学生应该通过实验提高自己剖析和设计操作系统的能力，加强分析问题、解决问题能力的培养，加强创新意识与探索精神、科学作风与综合素质的培养。

(3) 学生应该通过实验深入了解和熟练掌握一种操作系统的组成、特点、源码构成、内部结构、替换模块、扩充功能，以及编译、调试、运行的方法，达到拓宽编程思路、把握操作系统整体的目的。

(4) 学生应该通过实验养成良好的理论联系实际、自己动手操作的习惯，以获得项目管理和团队协作的实际训练和具体经验。

编写操作系统实验教程，首先要选择实验环境和平台，其次要精选上机实验内容，这对提高实验课程的教学质量十分重要，Linux 是开放源码操作系统，也是当今最流行和广为使用的主流操作系统之一。本书选用 Linux 内核版本 v2.6 作为操作系统实验和实习环境，通过在实用操作系统环境下的实际锻炼，不但让学生对操作系统的基本原理有更深入的理解，学生的动手实践能力更上一个台阶，而且能进一步掌握 Linux 操作系统的系统结构、设计和实现的基本思路和原理，也能更好地适应社会信息化对计算机专业人才的需求。

作者根据多年来操作系统研究和教学的经验，参考国内外出版的操作系统实验教材，精心设计系列实验供实验教学选择使用，本书中的实验题目和示例都已经在 v2.6.22 内核版本上通过调试，并能正确运行。

本实验教程的内容分 3 个部分。第一部分(第 1 章～第 11 章)为 Linux API 使用与操作系统算法实验。通过编写用户空间代码，从系统外部洞察操作系统的数据结构、内部状态和工作过

程，使学生熟悉 API 的使用，深入理解操作系统原理，初步掌握系统的组成模块和接口的使用方法，在真正了解如何设计与修改一个操作系统之前，必须学习如何使用一个操作系统；这一部分实验的内容为：

第 1 章 Linux 的安装和编译
第 2 章 进程与线程
第 3 章 传统的进程间通信
第 4 章 System V 的进程间通信
第 5 章 Shell 程序设计
第 6 章 页面替换算法
第 7 章 文件系统设计与实现
第 8 章 时钟与定时器
第 9 章 网络通信编程
第 10 章 事件驱动编程
第 11 章 综合实验：一个小型远程访问 FTP 服务系统

第二部分(第 12 章～第 19 章)为 Linux 内核修改实验。深入理解内核的捷径就是对它进行修改。通过改变系统数据结构、增加内核模块、添加内核函数、替换原有算法，不同程度地涉及内核态编程，从系统内部探索其内核结构、实现机制和典型算法，使学生达到初步具有分析、修改和更新操作系统的能力。这一部分实验的内容为：

第 12 章 内核模块
第 13 章 中断与系统调用
第 14 章 同步机制
第 15 章 进程调度
第 16 章 存储管理
第 17 章 虚拟文件系统
第 18 章 proc 文件系统
第 19 章 设备驱动程序

第三部分为附录。给出 Linux 常用命令、函数以及相关编辑工具介绍，便于学生进行实验编程及调试。

本书内容丰富、覆盖面广，既有用户空间编程，也有核心空间编程，实验题涉及操作系统的基本概念、原理、技术和方法；实验安排由浅入深、循序渐进，实验内容与课程教学彼此呼应、掌握基本原理与提高编程能力相并重；实验教学应当采用“精讲多练、上机为主”的原则，目的是让学生切身体验操作系统的设计原理与实现技术，锻炼和提高学生的实际应用能力和水平。实验教程拟安排 72 学时，24 学时授课、48 学时上机，做不完的实验题由学生在课后完成；每个(组)实验完成后，必须撰写和提交实验报告。附录 E 中给出了操作系统实验报告的可选内容。每个(组)实验应包含以下内容：实验目的、背景知识、实验内容，通过实验把操作系统的

概念、原理和 Linux 设计、实现的细节联系起来。在钻研细节之前去快速浏览理论教材的相关内容，将有助于设计实验解决方案。当然在操作系统实验课程的教学中，不必局限于本教程的实验题，鼓励学生自选题目和完成更多实验内容，进一步发挥学生的主观能动性和创新能力。

本书撰写过程中得到南京大学骆斌教授、张德富教授的指导与帮助，在此表示衷心感谢。特别感谢北京大学陈向群教授、北京师范大学党德鹏教授百忙中抽时间仔细审阅了全书，并提出了许多极为宝贵的专业评论和意见；徐天音、王钦辉编写部分实例程序，并参与实例程序的调试和正确性验证；丛邵鹏认真阅读了本书初稿，并提出了许多改进建议。对他们为此书所作出的贡献，一并表示衷心感谢。本书初稿已经在南京大学计算机科学与技术系、南京大学金陵学院试用，对两个单位领导的大力支持表示由衷谢意。除参考文献所列之外，本教材在编写过程中还参阅大量网络资源，谨此向各位作者致以深深的谢意。

由于作者水平有限，错误与疏漏之处在所难免，恳请读者批评指正。请使用此书的任课老师与作者联系，以获得相关的实验技术支持，E-mail 为：yebl@nju.edu.cn 和 minli@software.nju.edu.cn。

编著者

于南京大学金陵浦苑

2009 年 1 月

目　录

第 1 章　Linux 的安装和编译

1.1　实 验 目 的

- 了解 Linux 发展历史、功能和特点。
- 学习和动手安装 Linux 操作系统。
- 学习和动手编译 Linux 内核。
- 掌握用 C 语言开发应用程序的全过程。

1.2　背 景 知 识

1.2.1　Linux 简史

Linux 是由芬兰科学家 Linus Torvalds 于 20 世纪 90 年代初编写完成的一个类 UNIX 操作系统内核，从一开始就决定自由扩散源代码，并在 Internet 上发布。Linux 的出现引起众多编程高手的兴趣，纷纷对这个系统进行改进、扩充和完善，许多人作出了关键性贡献，并从最初由一个人编写的操作系统原型发展成在 Internet 上由无数志同道合的程序高手们共同开发而成的功能完备的现代操作系统。

1984 年，比尔·盖茨(Bill Gates)的哈佛大学同学 Richard M. Stallman 发起成立自由软件基金会(Free Software Foundation，FSF)和开源项目计划 GNU，并提出著名的开源协议标准——通用公共许可(General Public License，GPL)。GNU 是“GNU is Not UNIX”的缩写，既与 UNIX 有关又具有不同于 UNIX 的特点，旨在开发出一套完整的、免费的、公开源代码的类 UNIX 操作系统及其应用软件。自由软件的出现意义深远，科技是人类社会发展的阶梯，而科技知识的探索和积累是组成这个阶梯的一个个台阶，人类社会的发展是以知识积累为依托的，不断地在前人获得的知识的基础上发展和创新才得以一步步地提高。软件产业也是如此，如果能把已有的成果加以利用，避免每次都重复开发，将大大提高目前软件的生产率，借鉴别人的开发经验，互相利用，共同提高。带有源程序和设计思想的自由软件对人们学习和进一步开发软件起到了极大的促进作用。

20 世纪 80 年代末，GNU 计划的很多工作已经完成，包括编译器和文本编辑器等，如 GCC/GCC++编译器、Emacs 编辑器和 Shell 等。但是基于 Mach 的操作系统 Hurd(Hird of UNIX-Replacing Daemons)却迟迟没有推出。目前人们熟悉的软件，如 FreeBSD、FORTRAN 77、

GNU C、Open BSD、BSD email、BIND、Perl、Apache、TCP/IP、IP accounting、HTTP server、Lynx、Samba、ApplixWare、StarOffice 等都是自由软件经典之作。1991 年 10 月，Linus Torvalds 在网上发布 Linux 0.01 版源代码，1993 年 Linux 1.0 内核诞生，代码量达 17 万行，已经有相当大的名气，同时加入 GNU 并遵循 GPL 协议。Linus 把 Linux 内核贡献给 GNU 和自由软件，最终为 GNU 工程划上一个完美句号。至此，Linux 的开发也进入良性循环，GNU 组织全力支持 Linux 的发展，引入许多 GNU 工具。Linux 内核、C 库、编译器、基本工具及 GNU 工具集等组成了人们常说的 Linux 操作系统。今天所说的 Linux，事实上只是简称，其正式名称是 GNU/Linux(Linux-based version of the GNU system)。

1994 年，Linux 的第 1 个商业发行版 Slackware 问世；1995 年用户量超过几十万，介绍 Linux 操作系统的杂志《Linux Journal》开始发行；1996 年，美国国家标准与技术局(NIST)的计算机系统实验室确认 Caldera 公司的发行版 OpenLinux 符合 POSIX 标准；同年 Linux 2.0 内核发布，已有约 40 万行代码，步入实用化阶段，被移植到 AMD、Alpha、PowerPC、MIPS、Motorola 68000、SPARC、S/390、VAX 及 HP PA-RISC 等许多处理器上，全球已有几百万人使用；1998 年 Linux 操作系统是获得迅猛发展的一年，Red Hat Linux 5.0 获得 InfoWorld 的操作系统奖项，从嵌入式系统、桌面环境直至服务器上均已得到广泛使用，它在服务器端可与商用 UNIX 相媲美，在客户端则向 Windows 发起有力挑战；1999 年起许多计算机硬件及软件公司，如 IBM、Intel、Netscape、SGI、Sun、HP、Novell、Oracle、Informix、Sybase 等，都开始大力支持 Linux，各种软件纷纷移植到 Linux 平台上，运行在 Linux 下的应用软件越来越多，大大地推动了 Linux 的商品化进程，使它逐步成为一个功能强大、稳定高效的成熟操作系统；Linux 中文版已开发完成并得以应用，为发展我国自主研发和具有知识产权的操作系统提供了良好条件。1999 年推出 Linux 2.4 内核，2000 年我国联想公司推出幸福 Linux，中科院推出红旗 Linux；2001 年初第 1 届 Linux World 大会召开，象征着 Linux 时代的到来；2002 年内核开发者宣布 Linux 系统支持 64 位计算机平台；2003 年 Linux 2.6 内核发布，2004 年 SGI 宣布 Linux 成功地支持 256 个 CPU 的并行机，同年报告披露世界 500 强超级计算机系统中，使用 Linux 操作系统的已占 280 席，抢占了许多 UNIX 市场。从 Linux 发展史可看出，Internet 孕育了 Linux，没有 Internet 就不可能有 Linux 今天的成功。从某种意义上来说，Linux 是 UNIX 和 Internet 结合的产物。自由软件 Linux 是一个充满生机，已有巨大用户群和广泛应用领域的主流操作系统，也是唯一能与 UNIX 及 Windows 抗衡的操作系统。

Linux 源自 UNIX，但它不是 UNIX；它借鉴了 UNIX 的设计思想，却并没有直接使用 UNIX 源代码。Linux 具备现代 UNIX 所具有的几乎一切功能和特征，且与 POSIX 标准兼容。其主要技术特点如下：一个功能强大、稳定可靠、便于移植的多用户、多任务、多平台且具有开放性的 32 位/64 位通用操作系统；具有虚拟主存、共享库、支持各种体系结构和一系列 UNIX 开发工具及应用程序；全面支持 TCP/IP 协议和网络功能，支持多种文件系统；提供多种友善的用户界面，以及开放源代码，有利于发展各种特色操作系统。

1.2.2 Linux 内核的功能和结构

Linux 操作系统分 3 层：内核层、Shell 层与实用程序层。Shell 属系统程序部分，但运行在

用户态；实用程序层包含编译程序、编辑程序等，均在用户空间运行。内核是 Linux 的主要组成部分，运行在内核态，其基本功能包括以下几个方面。

- 资源抽象：用软件抽象硬件资源，简化对硬件的操作，屏蔽底层物理细节。用这种方法来管理各种硬件设备，如提供设备驱动程序、创建虚拟设备、创建虚拟处理器等。抽象也可针对没有特殊硬件的资源进行。
- 资源分配：把抽象出来的各种资源分配给多个应用程序使用，并负责资源的回收。
- 资源共享：根据不同资源类型和特性，提供不同机制确保应用程序获得所需资源，允许应用程序共享资源并提供资源共享的同步、互斥和通信机制。

具体来说，Linux 内核完成进程调度和管理、主存和虚拟存储管理、虚拟文件系统(Virtual File System，VFS)和文件管理、设备驱动和管理、网络接口和通信等工作，从而实现资源抽象、资源分配和资源共享等功能。

Linux 与 UNIX 一样采用单内核结构，运行时是一个大二进制映像，模块间的交互通过直接调用其他模块中的函数来实现。其优点是系统运行开销小、效率高，缺点是内核功能的适应性、灵活性和可伸缩性差。为此，Linux 引入“模块”机制，用来动态装入和删除一个文件系统或设备驱动程序，有效地弥补单内核结构的不足。Linux 符合 IEEE 的 POSIX 标准，为应用程序提供规范的应用编程接口(函数)。应用程序通常在用户态下执行，不能直接访问内核数据或直接对硬件进行操作，但可以通过标准函数库的库函数去调用内核提供的内核函数来请求内核服务，并切换到内核态执行。当内核完成相应请求后，再将应用程序返回到用户态。图 1-1 描述了 Linux 内核结构及其与应用程序和硬件之间的关系。

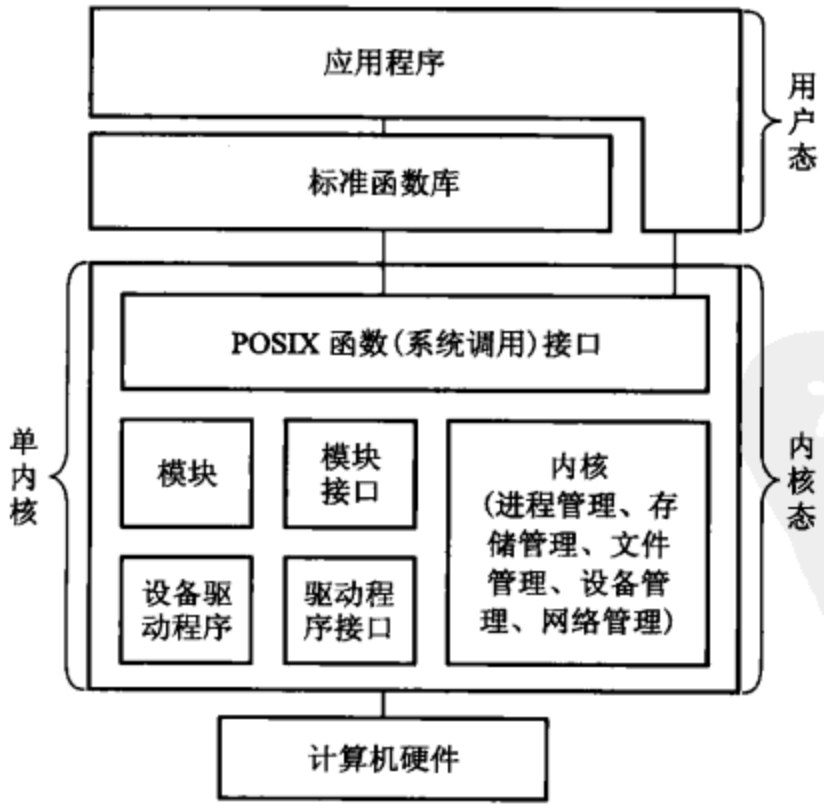


图 1-1 Linux 内核结构

Linux 的内核模块化结构、可动态装卸内核模块、可选择抢占式内核、支持多线程和内核线程等特性是其内核设计上最引以为豪及最为成功之处。Linux 的设计充分体现“自由”一词的精髓，现有的功能与特性集是 Linux 公开开发模型自由发展的结果。在 Linux 发展过程中形成一种值得称赞的务实态度，任何修改都要针对现实中确实存在的问题，经过周密完善的设计，并有正确、简洁的实现。

1.2.3 Linux 内核的版本和发行版本

Linux 有两种版本：内核版本和发行版本。

内核版本又称核心版本，由 Linus Torvalds 作为总体协调人的 Linux 开发小组及分布在各个国家的近百位开发人员参与，不断开发和推出新的内核。Linux 内核版本有两种：稳定版和开发版。一个 Linux 内核版本号形如：major.minor.patchlevel。其中，major 为主版本号，minor 为次版本号，patchlevel 表示对当前版本的修订次数。根据约定，次版本号为奇数时，表示该版本加入新内容，但不一定稳定，为开发版本；次版本号为偶数时，表示这是一个可以使用的稳定版本。例如，2.6.23 表示基于稳定版 2.6 第 23 次修订。目前最新的内核版本是 2.6 版。

发行版本是各个公司推出的版本，它们与内核版本是各自独立发展的。发行版本内附有内核源码以及很多针对不同硬件设备的内核映像。发行版本是一些基于 Linux 内核的软件包，常见的 Linux 发行版本有以下几种。

- Red Hat Linux(http://www.redhat.com)，由 Red Hat Software 公司发布，是当前流行的 Linux 版本。
- TurboLinux(http://www.turbolinux.com.cn)，提供从安装到使用的完整中文环境。
- Slackware(http://www.cdrom.com)，最早出现的 Linux 发行版本，适用于服务器。
- OpenLinux(http://www.caldera.com)，是由 Caldera 公司开发的发行版本。
- Debian(http://www.debian.org)，也称 GNU/Linux，由一群志愿者进行维护和升级。
- SuSELinux(http://www.suse.com)，用作服务器，在欧洲流行。
- Mandrake Linux(http://www.mandriva.com/)，基于 Red Hat Linux 开发的一个成功的发行版本。

国内开发成功的中文 Linux 发行版本有中科院的红旗 Linux、联想集团的幸福 Linux、中软公司的 COSIX Linux 等。

1.2.4 Linux 内核的目录结构

Linux 内核是特殊的程序，与应用程序相比较，仅仅是其程序意图不同，具有特殊权限，运行于特定空间，但也需要经过编译、链接之后才能运行。一旦基本系统安装完毕，具有系统管理员权限的用户即可编译内核。Linux 内核的源代码将依赖于体系结构的代码和独立于体系结构的代码分离开来，其重要的源代码子目录如下。

• /arch 子目录：存放与特定 CPU 和体系结构相关的源代码，而相关的.h 文件则放在 include/asm 之下；它又分子目录 alpha、i386、m68k、mips、PowerPC 和 sparc，每个子目录对应一种 CPU。在这个目录下，针对不同体系结构所移植的版本都含有 boot、kernel、lib 和 mm 子目录。

• /kernel 子目录：存放大多数内核函数，最主要的文件为 sched.c、time.c、sys.c 和 itimer.c。包含调度操作、等待队列操作、时钟和定时器操作、用户和组标识操作；进程创建及终止操作 fork.c 和 exit.c；signal.c 包含信号操作；softirq.c 包含软中断操作；此外，还包含 resource.c、dma.c 及 printk.c。

• /mm 子目录：存放含有独立于体系结构的主存管理文件，包括实现虚拟存储管理的源代码，如 swap.c、swapfile.c、page_io.c、swap_state.c、vmscan.c、slab.c、page_alloc.c、kmalloc.c、vmalloc.c、memory.c、mmap.c、mremap.c、mlock.c 和 mprotect.c 等。

• /fs 子目录：存放 VFS 和系统支持的各种文件系统源代码，每个子目录对应一个特定文件系统，如 Ext2、Ext3、FAT、NTFS、USB 及 PROC 文件系统等。VFS 的共用源代码有 devices.c、block_dev.c、super.c、inode.c、namei.c、buffer.c、open.c、read_write.c、readdir.c、select.c、pipe.c、fifo.c、ioctl.c 和 fcntl.c 等。

• /include 子目录：存放重要的内核.h 头文件，为各种 CPU 专设一个子目录；与平台无关的通用.h 头文件放在 include/asm 子目录下；还有通用子目录 include/linux 和 include/net 等。

• /drivers 子目录：存放所有设备驱动程序源代码(含有半数以上内核代码)，可分成 7 类，即块设备驱动程序/block、字符设备驱动程序/char、CD-ROM 驱动程序/cdrom、PCI 伪设备驱动程序/pci、SCSI 设备驱动程序/scsi、网络驱动程序/net 和声卡驱动程序/sound 等。

• /ipc 子目录：存放处理进程间通信所需的源代码。System V IPC 的 ipc_perm 结构在 include/linux/ipc.h 中描述，ipc_msg.c、ipc_sem.c、ipc_shm.c 和 ipc_pipe.c 分别实现消息队列、信号量、共享主存和管道。

• /net 子目录：存放网络子系统，如各种网卡和网络规程驱动程序。

• /security：存放安全子系统。

• /sound：存放音频子系统。

• /init 子目录：存放内核引导和初始化代码，许多重要文件，如 main.c、version.c 就位于该目录下。该文件还包含许多内核代码，如 cpu_idle()代码。

• /lib 子目录：存放内核需要的通用工具性内核函数，如对出错信息的处理，它能够在引导时解压内核并装入主存。

• /scripts 子目录：存放编译内核所用脚本和用于系统配置的命令文件。

• /documentation 子目录：存放内核源代码文档。

Linux 源代码可从 Internet 上内核官方网站 http://www.kernel.org 下载获得，展开 tar.gz 压缩包后，便可进行阅读；完整安装的 Linux 系统，其/usr/src/linux 目录下的文件即为内核源代码。

1.3 实验内容

1.3.1 实验1 安装Ubuntu 8.0.4

1.3.1.1 实验说明

学习和动手安装Linux发行版本Ubuntu 8.0.4，掌握操作系统的系统配置，了解建立操作系统应用环境的过程。

1.3.1.2 解决方案

1. Ubuntu的历史与演变

Ubuntu由Mark Shuttleworth创建和开发，首个版本于2004年10月20日发布，并以Debian为开发蓝本。该软件以每隔6个月发布一次新版本为目标，使人们得以更频繁地获取新软件。Ubuntu的开发目的是，使个人计算机变得简单易用并提供服务器版本。它的每个新版本均会包含最新版本的GNOME桌面环境，并且会在GNOME发布新版本后一个月内发行。与以往基于Debian的Linux发行版(如MEPIS、Xandros、Linspire、Progeny与Libranet等)相比，Ubuntu更接近Debian的开发理念，因为其主要使用自由软件，而其他发行版则会附带很多非开源软件。

Ubuntu软件套件主要基于Debian的不稳定分支，不论是软件套件格式(deb)还是软件管理与安装系统(Debian Apt/Synaptic)，Ubuntu会将所有对软件套件的修改即时地向Debian回馈，而不是在发布新版时才宣布这些修改。事实上，很多Ubuntu开发者均为Debian主要软件套件的维护者，但Debian与Ubuntu软件套件并不一定相互兼容，也就是说，将Debian软件包安装在Ubuntu上可能会出现兼容性问题，反之亦然。

Ubuntu的运作主要依靠Canonical公司的支持。在2005年7月8日，Mark Shuttleworth与Canonical公司宣布成立Ubuntu基金会，并对其提供1000万美元启动资金，成立基金会的目的是确保将来Ubuntu得以持续开发与获得支持。

Ubuntu最新支持版本为在2008年所推出的Ubuntu 8.0.4。用户可通过其邮寄服务来获取免费的安装光盘；用户也可以直接从Ubuntu网站下载光盘镜像并刻录安装。

在Ubuntu的基础上出现多种衍生版本，得到官方支持的有以下几种。

- Kubuntu：使用和Ubuntu一样的软件库，但不采用GNOME，而采用另一个用得非常普遍的KDE作为其默认桌面环境。
- Edubuntu：Ubuntu的教育发行版，使教育工作者可以在短于一小时的时间内设计计算机教室，或建立网上学习环境。
- Ubuntu Server Edition：从Ubuntu 5.10版(Breezy Badger)起与桌面版同步发行。提供服务器的应用程序，如电子邮件服务、LAMP网页服务、DNS设置工具、文件服务与数据库管理。

与原来桌面版本相比，服务器版的光盘镜像体积较小，并且其对硬件规格要求更低。若要运行服务器版，最少只需要有 500 MB 磁盘空间与 64 MB 主存便可，然而它并没有提供任何桌面环境，用户在默认环境里只可使用字符界面。

受到官方认可的版本还有以下几种。

- Gobuntu：完全以 GNU 自由软件打造的发行版。
- Xubuntu：属于轻量级发行版，使用 Xfce4 作为其默认桌面环境，与 Ubuntu 采用一样的软件库。
- Ubuntu Studio：用于多媒体编辑、创作的版本。

除此之外，还有许多其他衍生版，下面简单介绍一下。

- nUbuntu：专注于安全工具的版本。
- Ubuntu Lite：为旧计算机而设计的版本。
- zUbuntu：为 IBM zSeries 主机移植的版本。
- Ebuntu：基于 Enlightenment 0.17 桌面环境并附有视窗管理器的 Ubuntu 修改版本。
- Fluxbuntu：基于 Fluxbox 桌面环境的修改版本。
- Gnoppix：基于 Ubuntu Live CD 而研制的以 GNOME 为默认桌面环境的 Live CD 版本。
- Dubuntu：中国国内 Ubuntu 爱好者改进的 Ubuntu 版本，能够更好地支持中文，并且添加更多软件的 Live CD 版本。

2. Ubuntu 的硬件需求

一直以来，Ubuntu 均支持主流 i386、AMD 64 与 PowerPC 平台，因此大多数用户皆可在其个人计算机上安装相应的 Ubuntu 版本。2006 年 6 月，Ubuntu 新增加对 Ultra SPARC 与 Ultra SPARC T1 平台的支持，用户可下载相应版本进行安装。Ubuntu 甚至还支持在索尼的家用游戏机 Play Station 3 上安装。

自最初发行起，Ubuntu 提供一张安装光盘与一张用于预览的 Live CD。在 Ubuntu 6.06 LTS 发布时，将原来只用做预览的 Live CD 更改为不仅可用来预览，而且还可使用图形界面进行安装的光盘(即 Desktop CD)，而原来只提供字符安装界面的安装光盘则保留并改名为 Alternate install CD。

在硬件要求上，Ubuntu 桌面版本需要 256 MB 主存，并需要 4 GB 磁盘空间用来安装。Ubuntu 对硬件的支持列表可以从 https://wiki.ubuntu.com/HardwareSupport 查询。在文件系统方面，Ubuntu 默认采用 Ext3 文件系统，但也可以选择其他文件系统。而在访问 Windows 系统分区方面，可以自由读取和写入 FAT32 格式分区；但是对 NTFS 格式分区则只能读取，不能写入，不过也可以通过自行安装 ntfs-3G 软件套件实现读写 NTFS 分区的功能。在 Ubuntu 里，可以经由 Samba 软件来与其他操作系统进行文件信息交换，功能类似 Windows 平台上的“网上邻居”。

3. Ubuntu 的安装步骤

下载的光盘镜像刻录完成后，将计算机设为光盘启动并放进 Ubuntu 光盘即可开始，启动画面如图 1-2 所示，选择“Install Ubuntu”。待 Live CD 启动后，单击 Install 图标，开始安装

Ubuntu，如图 1-3 所示。

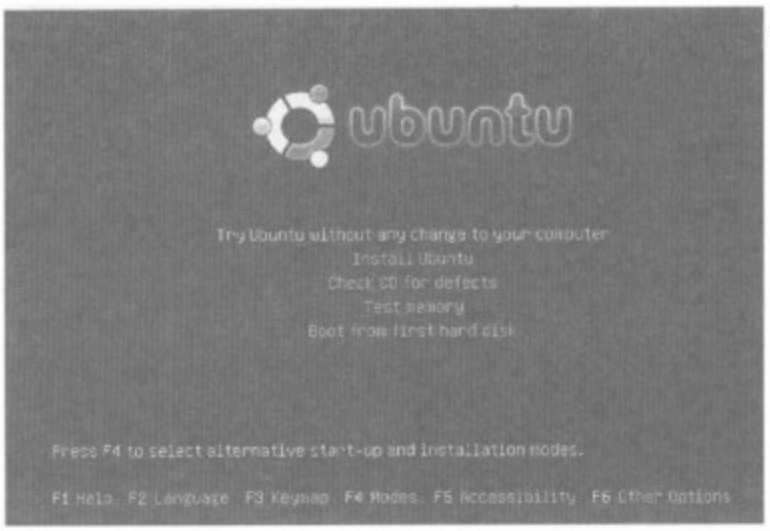


图 1-2 启动画面

图 1-3 Live CD 安装界面

Ubuntu 在图形界面下安装十分方便，只需要配置少量信息就可以完成安装任务。在安装界面出现后，通过 6 个步骤完成对安装参数的配置。

(1) 选择系统语言

首次安装 Ubuntu 系统建议选择中文(简体)，安装结束后若需改变语言配置，可使用语言配置工具。

(2) 选择时区，设置时间

区域选中国，并设置相应城市(如南京)，时区选 CST。

(3) 选择所使用的键盘

一般选择默认键盘布局(U.S. English)即可，特殊情况另行选择。安装结束后若需改变键盘类型，可使用键盘配置工具。

(4) 进行分区

多数计算机上已经安装 Windows 操作系统，若打算创建一个引导系统来兼容引导 Linux，需要使用双引导，以便计算机启动时，可选择启动其中一个操作系统。每个操作系统都从自己的磁盘分区中引导，并使用各自的磁盘或磁盘分区。

在设置分区时，Ubuntu 可以为用户自动创建分区，也可由用户手动创建分区。如果手动创建分区，建议至少创建两个分区，一个是/swap 交换分区，一个是/根分区，但一般把/boot、/home、/usr 及/var 挂载到单独的分区。由于一些老主板不支持大磁盘，建议将/boot 分离出来，并存放在磁盘前 1024 柱面上，以便引导 Linux。把/var 分离出来的原因是，该目录下存放日志和常常变动的临时文件，容易产生碎片，而/usr 就相当于 Windows 下面的 Program files+Windows 目录。创建 swap 分区时需要将分区格式设置为/swap，/根分区的文件格式可以使用 Ext3。/swap 分区的大小，需要根据计算机的主存大小决定，如果计算机主存为 n MB，则/swap 分区大小应该在 n～$2n$ MB 之间，通常/swap 分区大小为主存的两倍，即 $2n$ MB。图 1-4 所示为设置分区的操作

界面。

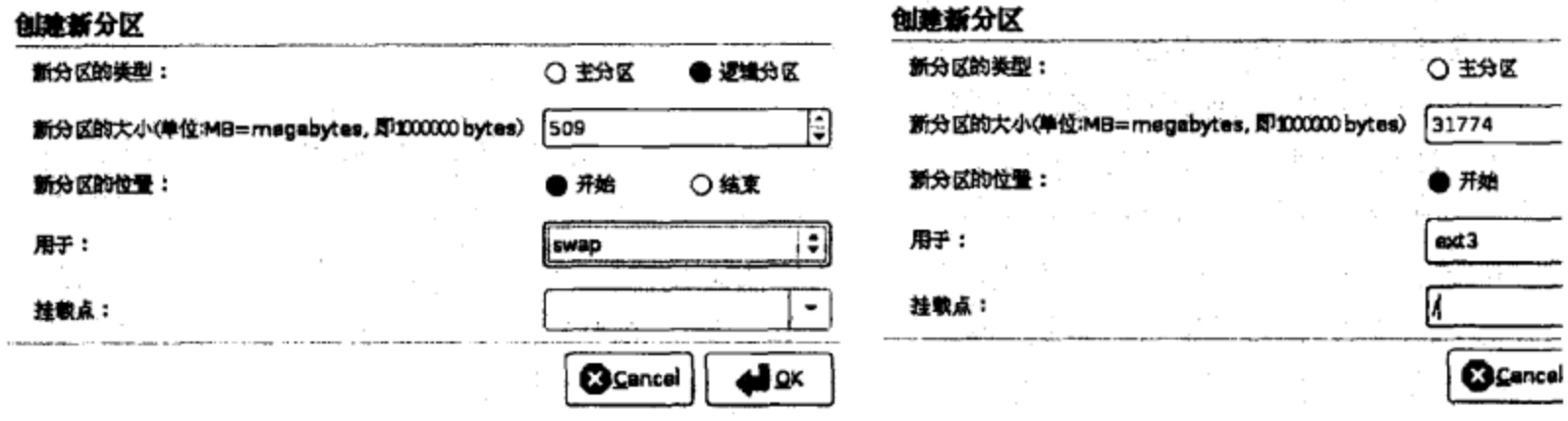


图 1-4 设置分区

(5) 设置账号

输入用户名、登录名和密码等。

(6) 确认配置

系统会把输入的配置参数输出在窗口中，让用户确认并允许再次修改。

在设置和配置参数后，Ubuntu 开始安装系统，等待安装结束后，用户重新启动一次便可以使用系统。首次进入系统后，由于很多软件都没有安装，需要对系统进行更新和安装一些必需的软件，由于篇幅关系这里不详细叙述，具体可参考 Ubuntu 中文官方网站 http://www.ubuntu.org.cn。

4. GNU 网络对象模型环境

GNU 网络对象模型环境 GNOME(The GNU Network Object Model Environment)是 GNU 计划的重要组成部分，该计划在 1997 年 8 月由 Miguel de Icaza 和 Federico Mena 发起，目标是研发 KDE(K Desktop Environment)的替代品。GNOME 计划提供两大部件：一是 GNOME 桌面环境，对最终用户来说它是一个十分有吸引力的桌面；二是 GNOME 开发平台，能使开发的应用程序与桌面其他部分集成为可扩展框架。它的主要软件工具包括以下几种。

- Abiword：文字处理器。
- Epiphany：网页浏览器。从 GNOME 2.4 起 Epiphany 取代 Galeon 成为默认浏览器。
- Evolution：联系人、日程和 E-mail 管理。
- Gaim：即时通信软件。
- Gedit：文本编辑器。
- The Gimp：高级图像编辑器。
- Gnumeric：电子表格软件。
- Ekiga：IP 电话或电话软件。
- Inkscape：矢量绘图软件。

- Nautilus：文件管理器。
- Rhythmbox：类似 Apple iTunes 的音乐管理软件。
- Totem：媒体播放器。

下面简单介绍 GNOME 桌面环境。

- 窗口：屏幕上可同时显示多个窗口，每个窗口中可运行不同应用程序。窗口管理器为窗口提供框架和按钮，利用它们来执行移动、关闭和改变窗口大小等操作。
- 菜单：用户可通过菜单访问所有 GNOME 桌面功能，可以使用“应用程序”菜单访问大部分标准功能、命令和配置选项，通过“主菜单”及“菜单栏”小程序访问“应用程序”菜单，向面板中添加“主菜单”和“菜单栏”小程序。“菜单栏”小程序包含一个“操作”菜单，“操作”菜单中包含用于执行各种功能的命令，“操作”菜单中的菜单项位于“主菜单”的顶层。
- 工作区：可将桌面分为几个独立工作区，工作区是指可供用户工作的离散区域。可以指定 GNOME 桌面上的工作区数量，可以切换到不同工作区，但是每次只能显示一个工作区。
- 文件管理器：文件管理器可用来访问文件和应用程序，可以在文件管理器内显示文件内容，或者从文件管理器中打开应用程序文件。可使用文件管理器管理文件和文件夹。
- 桌面：桌面是用户界面的活动组件，桌面位于桌面上所有其他组件的后面。将对象放在桌面上可以快速访问文件和目录，或启动常用的应用程序，也可以在桌面上单击从打开一个菜单。
- 首选项：GNOME 桌面包含专用的首选项工具，每个工具控制 GNOME 桌面行为的一个特定部分。要启动首选项工具，可从“主菜单”中选择“首选项”，从子菜单中选择要配置的项目。

5. KDE 桌面系统

Linux 系统主要采用两种桌面系统环境：KDE 和 GNOME。GNOME 图形界面非常精致，但功能不够丰富，由于 KDE 功能完善、稳定性高而受到越来越多用户的喜爱。下面简单介绍 KDE。

(1) 窗口管理器

KDE 采用 KWM(K Window Manager)作为窗口管理器。事实上，KDE 可以支持几乎所有的窗口管理器。KWM 决定了 KDE 桌面的外观和风格，其桌面环境由以下部分组成。

- 控制面板：屏幕底部是控制面板，也称为 K 面板。可以从这里启动应用程序，配置灵活，功能很强，控制面板由下述部分组成：隐藏按钮、主程序按钮、系统工具按钮(包括桌面中正在运行的程序菜单列表按钮、用户主目录按钮、KDE 控制中心按钮、退出 KDE 环境按钮、锁屏幕按钮)、虚拟桌面切换按钮、常用程序启动区、后台程序显示区、系统时间显示区等。
- KDE 桌面：屏幕中间部分是 KDE 桌面，放置常用的应用程序图标，可以在上面单击来运行或拖动它们。
- 任务栏：任务栏位于桌面上方，其中列出当前正在运行的程序。当前被激活的窗口在任务栏中的相应按钮就处于凹陷的高亮度状态。

(2) 使用桌面和菜单

KDE 可以支持多个桌面，称为虚拟桌面，在默认情况下，KDE 提供 4 个桌面。可以对它们进行不同设置。例如，为每个桌面设置不同背景、颜色和字体，放置不同应用程序图标等。桌面间切换可以通过菜单、按钮、控制键完成。

KDE 桌面系统提供各种菜单，使用十分方便。要执行某个菜单命令，可单击该菜单名，按指示执行。常用菜单有：桌面系统菜单、窗口控制菜单、窗口功能菜单(常见的菜单项有“文件”菜单、“编辑”菜单、“查看”菜单、“选项”菜单、“帮助”菜单等)。

(3) 使用文件管理器

KFM 是 KDE 提供的文件管理器，通过它来管理文件和目录，主要功能有显示文件和目录、移动和复制文件、修改文件属性、创建目录、删除文件和目录等。还提供 Kfind 工具，可用来查找指定文件。

(4) 使用文本编辑器

Kedit 是 KDE 环境下的标准文本编辑器，具有建立文件、打开文件、编辑文件、保存文件和打印文件等多项功能。

1.3.2 实验 2 编译 Linux 内核

1.3.2.1 实验说明

不管使用何种版本的 Linux，它们都有共同部分——Linux 内核，可通过修改 Linux 内核来增加对用户所需特性的支持。本书第二部分的实验大都涉及内核改动，因此，本实验首先来学习编译 Linux 内核，将在 Ubuntu 8.0.4 下动手编译并安装一个新的 Linux 2.6 内核，掌握操作系统的系统配置和常用的编译选项。

1.3.2.2 解决方案

1. 安装编译内核所需的软件包

编译内核之前，需要安装所需软件包，Ubuntu 提供方便的软件包管理工具供用户使用，可以通过一个简单命令完成所需软件包的安装：

```
#apt-get update
#apt-get install kernel-package libncurses5-dev fakeroot wget bzip2
```

需要注意，以上命令要求拥有管理员权限，普通用户可通过 su 命令切换到 root 用户。

2. 下载源代码

Linux 内核源代码可从官方网站 http://www.kernel.org 上下载获得，内核源代码以压缩包形式提供，有两种压缩格式：bzip2 和 gzip(GNU zip)。Linux 源代码包的文件名形式分别为 linux-x.y.z.tar.bz2 和 linux-x.y.z.tar.gz，其中 x.y.z 为源代码的版本号。例如，2.6.23 版本的内核源代码包为：linux-2.6.23.tar.bz2。在 Linux 下可以通过 wget 命令下载源代码：

```
$ cd /tmp
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-x.y.z.tar.bz2
```

bzip2 是默认和首选形式，它比 gzip 压缩效率更高，一般可下载 bz2 压缩包，接着再下载对应的 sign 文件，后者被用来验证内核压缩文档的 openPGP 签名。GPG 签名只是保证镜像网站提供的压缩包与 kernel.org 所提供的是相同的，如果用户在 kernel.org 下载，则不需要验证签名。

3. 解压缩

解压缩之前，需要考虑将压缩包解压到何处，即要在哪个目录进行 Linux 内核源代码编译。内核源代码树的 README 中有这样一段话:

Do NOT use the /usr/src/linux area! This area has a (usually incomplete) set of kernel headers that are used by the library header files.They should match the library, and not get messed up by whatever the kernel-du-jour happens to be.

意思是不要将内核源代码解压到/usr/src/linux 目录中，因为该目录中的头文件是供一些库文件使用的。用户可以在/usr/src 中新建一个目录，用内核版本命名，比如/usr/src/linux-2.6.23，方便内核管理。

两种压缩包格式有不同的解压方法。

解压 bzip2 格式压缩包用命令如下：

```
# tar -xjvf linux-2.6.23.tar.bz2 -C /usr/src
```

解压 gzip 格式压缩包用命令如下：

```
# tar -xzvf linux-2.6.23.tar.gz -C /usr/src
```

4. 给内核打补丁

许多开发者给 Linux 内核添加新功能，如果需要使用这些功能，用户需要下载相应的补丁，并给内核源代码打补丁。在已经下载需要的补丁(patch.bz2)到/usr/src 后，通过运行下面的命令给内核源代码直接打上补丁：

```
#bzip2 -dc /usr/src/patch.bz2 | patch -p1 --dry-run
#bzip2 -dc /usr/src/patch.bz2 | patch -p1
```

第 1 条命令用于测试，对内核没有任何影响。如果没有显示错误信息，可以运行第 2 条命令给内核打补丁；如果第 1 条命令有误，说明不能使用该补丁。

内核中的测试版本也以补丁形式提供，例如用户需要的一个功能仅存在于 2.6.19-rc4 中，正式、完整的内核版本仍没有发布，而 patch-2.6.19-rc4.biz2 已经发布，此时可以把这个补丁加到 2.6.18 内核源代码中。

下面是如何给 2.6.18 打上 2.6.19-rc4 补丁的命令：

```
#cd /usr/src
#wget http://www.kernel.org/pub/linux/kernel/v2.6/testing/patch-2.6.19-rc4.bz2
#cd /usr/src/linux
#bzip2 -dc /usr/src/patch-2.6.19-rc4.bz2 | patch -p1 --dry-run
```

```
#bzip2 -dc /usr/src/patch-2.6.19-rc4.bz2 | patch -p1
```

5. **配置内核**

在编译 Linux 内核之前，可以根据需要选择配置选项，但必须告诉编译程序 Linux 内核需要哪些功能，还必须指明是将这些功能模块编到内核中去还是将其配置成动态可加载模块。Linux 提供多种内核配置命令。

(1) make config

make config 是最基础的配置命令，以文本提示方式配置编译选项，工作时打开字符模式对话框，在终端上提出问题，并要求用户回答所有问题。对每个问题有 3 种可能选择：Yes、No 和 Module。Module 告知内核配置在运行时使用动态可装载模块，而不是静态地将功能连接到内核中。

采用 make config 对内核进行配置，有几个重要内核配置选项。

- Loadable module support(对模块的支持)：有 3 项，Enable loadable module support 一般必选，Set version information on all module symbols 可以不选，Kernel module loader 让内核在启动时有自己加载必需模块的能力，建议选择。
- Processor type and features：选择支持的 CPU 类型。
- General setup：设置通用属性，这部分内容很多，一般使用默认设置。下面是经常使用的一些选项：Networking support(网络支持)、PCI support(PCI 卡支持)、PCI access mode(PCI 访问模式)、Support for hot-pluggable devices(热插拔设备支持)、Power management support(电源管理支持)、Parallel port support(并口支持)、Block devices(块设备)、Networking options(网络配置选项)、SCSI support(SCSI 设备支持)、Character devices(字符设备支持)、File systems(文件系统)、Network file system(网络文件系统)、USB support(USB 接口支持)等。

更多的配置选项及含义可参见有关资料，用户若不熟悉配置选项，也可以按默认配置编译内核。

```
#make defconfig        /*按默认选项对内核进行配置*/
#make allnoconfig      /*除必需的选项外，其他选项一律不选*/
```

(2) make menuconfig

make menuconfig 是主流配置命令，它需要 ncurses 库支持，在 Ubuntu 中默认是不支持的，必须先安装它：

```
# apt-get install libncurses5-dev
```

工作时打开文本图形对话框，其功能和 make config 命令基本相同，但可以只配置所需部分，使用比 make config 命令方便得多。

(3) make xconfig

make xconfig 是主流配置命令，它基于 X11，使用 qt 库，在 Ubuntu 中应先安装 qt 库：

```
# apt-get install libqt3-headers libqt3-mt-dev
```

另外，如果系统没有安装 g++，make xconfig 命令也会出错，这时需要先安装 g++，通过

下面命令完成：

```
#sudo apt-get install build-essential
```

工作时打开 GUI 对话框，使用比 make menuconfig 命令更方便。

6. 编译内核

配置完毕后，通过 make 命令编译内核：

```
#make
#make modules
```

make modules 对内核模块进行编译，在编译过程中，会出现许多编译信息，如果用户不想看这些信息，可用重定向忽略编译信息。

```
#make >/dev/null
```

在编译内核时，可以通过 make -j<n>来加速内核的编译，n 一般等于 CPU 数量的两倍。对于单 CPU，通常 n=2。该命令可以为编译过程分配两个任务，提高 CPU 利用率。

```
#make -j2 >dev/null
```

还可以使用 ccache 来提高编译速度，Debian/Ubuntu 系统中默认并没有安装它，首先安装它：

```
$ sudo apt-get install ccache
```

然后，更改内核根目录的 makefile，将 CC 和 HOSTCC 变量定义前添加 ccache：

```
CC = $(CROSS_COMPILE)gcc
HOSTCC  = gcc
```

更改为：

```
CC   = ccache $(CROSS_COMPILE)gcc
HOSTCC = ccache gcc
```

7. 安装内核

在编译成功后，需要将新内核模块和内核安装到系统中：

```
# make modules_install          /*安装内核模块*/
# make install                  /*安装内核*/
```

make modules_install 会将相应内核模块安装到/lib/目录下，make install 则将内核映像复制到/boot 目录下。用户还需要为内核创建一个 initrd.img 文件，该文件用来存储挂载根文件系统所需的模块。

```
# cd /boot
# mkinitrd -o initrd.img-2.6.23 2.6.23
```

最后一步是对用户的 grub 配置文件进行修改，以便在启动时能够选择新内核。

```
# vi /boot/grub/menu.lst
title       Debian GNU/Linux, kernel 2.6.23 Default
root        (hd0,0)
kernel      /boot/vmlinuz root=/dev/hdb1 ro
initrd      /boot/initrd.img-2.6.23
```

```
savedefault
boot
```

1.3.3 实验 3 Linux 下 C 语言程序开发过程

1.3.3.1 实验说明

学会和掌握用 C 语言开发一个应用程序的全过程，包括编辑、编译、调试过程及初步掌握 makefile 的使用方法。

1.3.3.2 解决方案

1. 编译

在 POSIX 兼容系统中，C 语言编译器的名称为 c89。历史上，C 语言编译器被简称为 cc。许多年来，不同厂商销售的类 UNIX 系统中所带的 C 语言编译器均包含不同功能和选项，但它们一般都称为 cc。在 Linux 系统中，通常使用 GNU C 编译器，或简称 gcc。下面通过 HelloWorld 程序开始，简单介绍 gcc 的使用方法。

(1) 编辑源代码

```
#include <stdio.h>
int main(int argc, char *argv[ ])
{
    printf("Hello, world\n");
    return 0;
}
```

上面是文件 hello.c 的源代码，需要使用一个编辑器来输入这个程序。在典型的 Linux 系统上有许多编辑器可以使用，比较流行的编辑器是 vi 和 emacs，关于这两个编辑器的使用方法，请参见本书附录。

(2) 编译源程序

```
$ gcc -o hello hello.c
$./hello
Hello,world
```

安装系统后若调用 gcc 编译时出现编译错误，则可能是因为没有安装 gcc。安装 gcc 的命令如下：

```
#sudo apt-get install build-essential
```

调用 gcc 将 C 语言源代码转换成可执行文件 hello。如果不加-o 选项，编译器则会把编译后的可执行文件命名为 a.out。在执行 hello 文件时，在 hello 前添加./，这是让 Shell 在当前目录下去寻找需要运行的可执行文件；如果不添加./，Shell 会在 PATH 环境变量设置的目录中去寻找可执行文件，这些目录中通常不会包含当前目录，也就无法找到 hello 文件。

如果在编译过程中，需要多个源文件，可将多个源文件分别编译成目标文件，再将其进行链接。例如，需要将 program.c 和 hello.c 编译成 program，需要如下命令：

```
$gcc -c program.c
$gcc -c hello.c
$gcc -o program program.o hello.o
```

-c 标志指示编译器将源代码编译成为.o 的目标文件。

(3) 头文件

在用 C 语言进行程序设计时，通常需要使用头文件来提供对常量的定义和对库函数调用的声明。对 C 语言来说，这些头文件几乎总是在/usr/include 目录及其子目录下。那些依赖特定 Linux 版本的头文件通常可在目录/usr/include/sys 和/usr/include/linux 中找到，其他编程系统也有各自的 include 文件，并将其存储在能被相应编译器自动搜索到的目录里。例如，X 视窗系统的/usr/include/X11 目录和 GNU C++的/usr/include/g++目录。

在调用 C 语言编译器时，可以使用–I 选项来包含保存在子目录或非标准位置中的 include 文件。例如：

```
$gcc -I/usr/openwin/include fred.c
```

它指示编译器不仅在标准位置，也在/usr/openwin/include 目录中查找程序 fred.c 中包含的头文件。

(4) 库文件

库是一组预先编译好的函数集合，这些函数都是按照可重用原则编写的，它们通常由一组相互关联的函数组成并执行某项常见任务，标准库文件一般存储在/lib 和/usr/lib 目录中。C 语言编译器(或更确切地说是链接程序)需要知道要搜索哪些库文件，在默认情况下，它只搜索标准 C 语言库。库文件的名字总是以 lib 开头，其后指明是什么库(例如，c 代表 C 语言库，m 代表数学库)。文件名最后以“.”开始，然后给出库文件的类型：.a 代表传统静态函数库，.so 代表共享函数库。

函数库通常以静态库和共享库两种格式存在，可以用 ls /usr/lib 命令查看，可以通过给出完整路径名或用-l 选项来指示编译器要搜索的库文件。例如：

```
$gcc -o fred fred.c /usr/lib/libm.a
```

这条命令指示编译器编译文件 fred.c，将编译产生的程序文件命名为 fred，并且搜索标准 C 语言函数库外，还搜索数学库。下面的命令也能产生类似结果：

```
$gcc -o fred fred.c -lm
```

-lm(字母 l 和 m 之间没有空格)是一种简写方式，它代表的是标准库目录中名为 libm.a 的函数库。-lm 标志的另一个好处是：如果有共享库，编译器会自动选择共享库。

虽然库文件和头文件一样，通常都保存在标准位置，但也可以通过-L 标志为编译增加库的搜索路径。例如：

```
$gcc -o x11fred -L/usr/openwin/lib x11fred.c -lx11
```

这条命令用/usr/openwin/lib 目录中的 libX11 库版本来编译和连接程序 x11fred。

用户也可以创建自己的库文件，在创建库文件前，将需要的目标文件编译好，再将这些目标文件合并成单独的库文件。例如，用于拥有 foo1.c 和 foo2.c 两个源文件，希望将其打包成为一个库文件：

```
$gcc -c foo1.c
$gcc -c foo2.c
$ar crv libfoo.a foo1.o foo2.o
```

通过执行该命令后，库文件就创建好了。

gcc 的选项很多，表 1-1 列出 gcc 常用的一些选项。

表 1-1 gcc 选项

选　　项	说　　明
-c	只进行预处理、编译和汇编，不作连接，生成.o 文件。常用于不包含 main 的子程序文件
-S	只进行预处理和编译，生成.s 文件，是汇编源代码
-E	只进行预处理，预处理的结果送到标准输出，而生成.i 文件。可以通过重定向保存
-C	预处理时不删除注释信息，不指定将输出到 a.out
-M	预处理输出适合 make 的规则，用来描述目标文件的依赖关系
-o file	指定输出目标文件名
-ansi	支持符合 ANSI 标准的 C 程序，关闭不兼容特性
-include file	功能等同于代码中的#include
-Dmacro[=defval]	定义一个宏，相当于代码中的#define macro [defval]
-Umacro	取消宏定义，相当于代码中的#undef macro
-undef	取消对任何非标准宏定义
-Idir	头文件的搜索路径中添加目录 dir
-Ldir	库文件的搜索路径中添加目录 dir
-lname	连接 libname.so 库来编译程序
-O[0-3]	编译器优化，数字越大，优化级别越高，0 表示不优化
-g	编译器编译时加入标准 debug 信息
-pg	编译器加入信息 profile，供分析程序 gprof 使用
-I library	连接 library.a 库文件
-share	使用动态库
-static	禁止使用动态库
-shared	生成共享目标文件
-V version	选择使用的 GNU gcc 版本

2. make 命令和 makefile 文件

(1) 为什么要有 makefile

在编写小程序时，许多人都会在编辑完源程序文件后简单地重新编译所有文件，以重建目

标程序。但是对大型程序来说，使用这种方式会带来明显问题。修改一个文件就需要重新编译所有文件，如果在程序中创建多个头文件，并在不同源文件中包含它们，会带来潜在的、更严重的问题。比如说，有 3 个头文件：a.h、b.h 和 c.h，3 个 C 语言源文件：main.c、2.c 和 3.c，具体情况如下：

```
/***** main.c *****/
#include "a.h"
…
/***** 2.c *****/
#include "a.h"
#include "b.h"
…
/***** 3.c *****/
#include "b.h"
#include "c.h"
```

如果程序员只修改文件 c.h，则源文件 main.c 和 2.c 无需重新编译，3.c 需要重新编译。但如果修改的是文件 b.h，而程序员忘记重新编译源文件 2.c，则最终程序就可能无法正常工作。make 工具可以解决这些问题，它会在必要时重新编译所有受改动影响的源文件。

在使用 make 工具时，用户需要提供一个文件，用来说明源文件之间的依赖关系和构建规则，这个文件称为 makefile。make 命令会读取 makefile 文件的内容，它先确定要创建的目标文件，然后比较该目标所依赖的源文件的日期和时间，以决定应该采用哪条规则来构造目标。

(2) 依赖关系

依赖关系定义最终应用程序中的每个文件与源文件之间的关系。在上例中，可以把依赖关系定义为最终应用程序依赖于文件 main.o、2.o 和 3.o。同样，main.o 依赖于 main.c 和 a.h，2.o 依赖于 2.c、a.h 和 b.h，3.o 依赖于 3.c、b.h 和 c.h。因此，main.o 受文件 main.c 和 a.h 修改的影响，如果这两个文件之一有所改变，就需要重新编译 main.c，以重建 main.o。在 makefile 文件中，这些规则的写法是：先写目标名称，然后紧跟着一个冒号，接着是空格或制表符，最后是用空格或制表符隔开的文件列表。上例可以表示为：

```
myapp: main.o 2.o 3.o
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
```

这组依赖关系形成层次结构，显示源文件之间的关系。如果 b.h 发生改变，就需要重新编译 2.o 和 3.o，而由于 2.o 和 3.o 发生改变，目标 myapp 也需要重新创建。

如果想一次制作多个文件，就可以利用名义上的目标 all。假设应用程序由二进制文件 myapp 和使用手册 myapp.1 组成，可以用下面这行语句进行定义：

```
all: myapp myapp.1
```

如果在使用 make 命令时未指定目标 all，则 make 命令将只创建它在文件 makefile 中找到的第 1 个目标。

(3) 创建规则

makefile 的第 2 部分是创建规则，在上例的 makefile 中，只是指定文件之间的依赖关系，但是 makefile 还需要知道如何创建文件，例如 2.c 修改后，2.o 用什么命令来创建。makefile 有许多默认规则，用户也可以指定创建规则，大多数规则都包含一个简单命令。在上例中，可以创建一个 makefile，命名为 Makefile1：

```
myapp: main.o 2.o 3.o
    gcc -o myapp main.o 2.o 3.o
main.o: main.c a.h
    gcc -c main.c
2.o: 2.c a.h b.h
    gcc -c 2.c
3.o: 3.c b.h c.h
    gcc -c 3.c
```

make 命令会假设 myapp 是要创建的目标文件，然后它会检查依赖性，并根据规则创建目标文件。

```
$make -f Makefile1
gcc -c main.c
gcc -c 2.c
gcc -c 3.c
gcc -o myapp main.o 2.o 3.o
```

如果已修改 b.h，make 命令能够自动重新创建 2.o、3.o 和 myapp：

```
$touch b.h
$make -f Makefile1
gcc -c 2.c
gcc -c 3.c
gcc -o myapp main.o 2.o 3.o
```

如果删除 2.o，make 命令也会自动创建 2.o 并自动重新创建 myapp：

```
$rm 2.o
$make -f Makefile1
gcc -c 2.c
gcc -o myapp main.o 2.o 3.o
```

(4) 宏

在 makefile 文件中还可以通过 MACRONAME=value 来定义宏，引用宏的方法是使用$(MACRONAME)或${MACRONAME}。如果想把一个宏的值设置为空，可以令等号后面无内容。makefile 文件中的宏常用于设置编译器选项。在软件开发过程中，一般不对编译结果进行优化，而是将调试信息包含进去。但是对于软件发行版，通常要进行优化，并且不包含调试信

息。宏也常被用于设置编译器名称，在 Makefile1 中，假设编译器名称为 gcc，在其他环境中，编译器的名称可能为 cc 或者 c89。通过宏可以方便解决该问题。宏通常在 makefile 内部定义，但也可以在调用 make 命令时在命令行上给出宏定义，例如命令 make CC=c89。下面是一个使用宏的 makefile：

```
#Makefile2
all: myapp
CC = gcc
INCLUDE = .
CFLAGS = -g-Wall-ansi
myapp: main.o 2.o 3.o
    $(CC) -o myapp main.o 2.o 3.o
main.o: main.c a.h
    $(CC) -I$(INCLUDE) $(CFLAGS) -c main.c
2.o: 2.c a.h b.h
    $(CC) -I$(INCLUDE) $(CFLAGS) -c 2.c
3.o: 3.c b.h c.h
    $(CC) -I$(INCLUDE) $(CFLAGS) -c 3.c
```

如果通过新的 makefile，将会看到如下输出：

```
$make -f Makefile2
gcc -I. -g-Wall-ansi-c main.c
gcc -I. -g-Wall-ansi-c 2.c
gcc -o myapp main.o 2.o 3.o
```

(5) 内置规则

make 命令本身带有大量内置规则，可以极大地简化 makefile 文件的内容。对于一个源文件 foo.c，在没有 makefile 时，make 命令也知道如何编译 main.c：

```
$make foo
cc foo.c -o foo
```

make 命令知道如何调用编译器，它默认使用的编译器名称为 cc。可以通过宏改变编译器名称：

```
$make CC=gcc CFLAGS="-Wall-g" foo
gcc -Wall-g foo.c -o foo
```

可以通过 make -p 命令打印 make 命令的内置规则。比较重要的是：

```
OUTPUT_OPTION = -o $@
COMPILE.c = $(CC) -$(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
$.o: $.c
    $(COMPILE.c) $(OUTPUT_OPTION) $<
```

有了这些内置规则，在大多数情况下，用户只需在 makefile 中指定依赖关系即可。例如：

```
main.o: main.c a.h
```

```
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
```

更让用户省心的是，gcc 能够自动帮助用户创建这些依赖关系，用户通过 gcc -MM 命令就可以导出项目中的依赖关系清单：

```
$gcc -MM main.c 2.c 3.c
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
```

用户只需要通过 gcc 导出依赖关系，并直接插入 makefile 文件中，就可以方便地创建 makefile 文件。

(6) 多目标

通常需要制作的目标文件不止一个。例如，可以添加 clean 目标用于删除不需要的临时文件，通过 install 目标用于将编译后的应用程序安装到指定目录下。

```
$make
$make install
$make clean
```

用户通常通过这 3 个命令就可以完成编译、安装、清除临时文件的工作。例如：

```
all: myapp
CC = gcc
INSTDIR = /usr/local/bin
INCLUDE = .
CFLAGS = -g-Wall-ansi
myapp: main.o 2.o 3.o
	$(CC) -o myapp main.o 2.o 3.o
main.o: main.c a.h
	$(CC) -I$(INCLUDE) $(CFLAGS) -c main.c
2.o: 2.c a.h b.h
	$(CC) -I$(INCLUDE) $(CFLAGS) -c 2.c
3.o: 3.c b.h c.h
	$(CC) -I$(INCLUDE) $(CFLAGS) -c 3.c
clean:
	-rm main.o 2.o 3.o
install: myapp
	@if [ -d $(INSTDIR) ] ; \
		then \
			cp myapp $INSTDIR); \
			chmod a+x $(INSTDIR)/myapp; \
			chmod og -w $(INSTDIR)/myapp; \
			echo "Installed in $(INSTDIR)"; \
```

```
    else
        echo "Sorry, $(INSTDIR) does not exist";\
    fi
```

在这个 makefile 中，目标 all 仍然只定义 myapp 这一个目标。因此在执行 make 命令时如果未指定目标，它的默认行为就是创建目标 myapp。目标 clean 用 rm 命令来删除临时文件，rm 命令以减号开头，表示让 make 命令忽略 rm 命令的执行结果，这意味着即使 rm 要删除的文件不存在，也不会返回错误。clean 目标并没有定义任何依赖，因此每次执行 make clean 都会通过 rm 命令删除临时文件。对于目标 install，它定义依赖 myapp，因此在目标 install 被执行之前，myapp 会先被创建。用于制作 install 目标的规则由几个 Shell 脚本命令组成。由于 make 命令是通过调用 Shell 执行用户指定的规则，所以必须在每一行添加一个反斜杠\，让这些 Shell 脚本在逻辑上处于同一行，并作为一个整体传递给 Shell。这个命令以@开头，表示 make 在执行这些规则之前不会在标准输出上显示命令本身。

(7) Linux 内核中的 makefile 文件

Linux 内核也采用 makefile 文件对内核进行编译，Linux 内核中 makefile 文件包括 5 部分：

```
makefile                    /*顶层 makefile 文件*/
.config                     /* 内核配置文件*/
arch/$(ARCH)/makefile       /*机器体系 makefile 文件*/
scripts/makefile.*          /*所有内核 makefile 共用定义和规则*/
kbuild makefiles            /*其他 makefile 文件*/
```

通过内核配置操作产生.config 文件，顶层 makefile 文件读取该文件的配置，它负责产生两个主要程序：内核映像(vmlinux image)和模块。具体做法是，根据内核配置，通过递归编译内核代码树子目录建立这两个文件。顶层 makefile 文件是文本名为 arch/$(ARCH)/makefile 的机器体系 makefile 文件，它能为顶层 makefile 文件提供与计算机相关的信息。

每个子目录有一个 makefile 文件，其任务是根据上级目录 makefile 文件命令启动编译。这些 makefile 文件使用.config 文件配置数据构建各种文件列表，并使用这些文件列表编译内嵌或模块目标文件。scripts/makefile.*包含了所有的定义和规则，与 makefile 文件一起编译出内核程序。

普通用户只需要使用基于文本的命令行工具 make config(也可以使用基于 ncurses 库的图形界面工具 make menuconfig 或基于 X11 的图形界面工具 make xconfig)命令编译内核，通常不必读或编辑内核 makefile 文件或其他源文件。普通开发者维护设备驱动程序、文件系统和网络协议代码，他们维护相关子系统的 makefile 文件。体系结构开发者关注一个整体的体系架构，如 sparc 或 ia64，他们既需要掌握关于体系结构的 makefile 文件，也要熟悉内核 makefile 文件。内核开发者关注内核编译系统本身，他们需要清楚内核 makefile 文件的所有方面。

3. 调试

在商业 UNIX 系统中有许多可用的调试器，常见的有 adb、sdb 和 dbx。较复杂的调试器允许用户在源代码级别查看程序的详细状态信息。sdb、dbx 和 GNU 的调试器 gdb 都可以做到这

一点。目前有一些针对 gdb 的“前端”程序，它们提供非常友好的用户界面，xxgdb、tgdb 和 ddd 都是这样的程序。一些 IDE 也提供调试功能或一个用于 gdb 的前端。为了能够调试程序，需要在编译它时加上一个或多个特殊编译器选项。这些选项的作用是让编译器在程序中添加额外调试信息，这些信息包括各种符号的源代码行号，调试器将利用这些信息向用户显示程序已经执行到源代码的什么位置。-g 标志是对程序进行调试性编译时常用的一个选项，必须在编译每个需要调试的源文件都加上这个选项。调试信息的加入将使可执行程序的长度成倍增加。尽管可执行程序的容量可能增加，程序运行时所需要的主存数量还是和原来一样。程序调试结束后，最好将调试信息删除，可以通过 strip 命令直接将调试信息从可执行文件中删除。

在命令行上输入“gdb”并按回车键就可以运行 gdb，如果一切正常，gdb 将被启动并且显示命令提示符：

```
(gdb)
```

当启动 gdb 后，可以在命令行上指定很多的选项。也可以用下面的方式来运行 gdb：

```
gdb <fname>
```

采用这种方式运行 gdb 时，gdb 将直接加载名为 fname 的可执行文件。

gdb 支持很多命令，使开发者能实现不同的功能。这些命令从简单的文件加载到允许检查所调用的堆栈内容的复杂命令，表 1-2 列出在用 gdb 调试程序时常常会用到的一些命令。

表 1-2 gdb 基本命令

命 令	描 述
file FILE	装入欲调试的可执行文件 FILE
kill	终止正在调试的程序
list	显示产生执行文件的源代码
next	执行一行源代码但不进入函数内部
step	执行一行源代码而且进入函数内部
run	执行当前被调试的程序
quit	终止 gdb
watch	监视一个变量的值而不管它何时被改变
disable	禁用断点
enable	能用断点
break	在代码里设置断点，这将使程序执行到这里时被挂起
break num	指定行设置断点，使程序执行到这里时被挂起
clear	为指定行或函数清除断点
catch	设置事件捕捉点
info var	显示所有全局和静态变量名
info prog	显示被调试工具的执行状态
info local	显示函数中的局部变量信息
info func	显示所有函数名
info files	显示被调试文件信息
info break	显示当前断点清单

续表

命 令	描 述
display expr	程序停止点上显示表达式值
print expr	打印表达式值
print expr	显示表达式值
continue	继续执行被调试程序
bt	显示调用栈帧，可用来显示函数调用顺序
make	不退出 gdb 就可以使用 make 工具
shell	不退出 gdb 就执行 shell 命令
help	显示命令帮助信息
show	显示 gdb 的所有命令

gdb 支持很多与 Shell 程序一样的命令编辑特征，如可以像在 bash 里那样，按 Tab 键让 gdb 补齐一个唯一的命令。如果不唯一，gdb 会列出所有匹配的命令，也能用光标键上下翻动历史命令。

下面列出将被调试的程序，这个程序被称为 greeting，它显示一个简单问候句，再将该字符串反序列出(例如：hello 的反序为 olleh)。

```
/*****greeting.c*****/
#include   <stdio.h>
int main(int argc, char *argv[ ])
{
   char my_string[ ] = "hello there";
   my_print (my_string);
   my_print2 (my_string);
   return 0;
}

void my_print (char *string)
{
   printf ("The string is %s\n", string);
}

void my_print2 (char *string)
{
   char *string2;
   int size, i;
   size = strlen (string);
```

```
    string2 = (char *) malloc (size + 1);

    for (i = 0; i < size; i++)
      string2[size –i] = string[i];

    string2[size+1] = '\0';
    printf ("The string printed backward is %s\n", string2);
  }
```

该程序执行时，显示如下结果：

```
The string is hello there
The string printed backward is
```

输出的第 1 行是正确的，但第 2 行打印出的信息并不是人们所期望的，所设想的输出应该是：

```
The string printed backward is ereht olleh
```

由于某些原因，my_print2 函数没有正常工作，可以用 gdb 对程序进行调试。先输入如下命令：

```
gdb greeting
```

这时可以用 gdb 的 run 命令来运行 greeting，当它在 gdb 里被运行后结果为：

```
(gdb) run
Starting program: /root/greeting
The string is hello there
The string printed backward is
Program exited with code 041
```

这个输出和在 gdb 外面运行的结果一样。为了找出症结所在，可以在 my_print2 函数的 for 语句后设一个断点，具体的做法是在 gdb 提示符下输入 list 命令 3 次，列出源代码：

```
(gdb) list
(gdb) list
(gdb) list
```

也可以在输入一次 list 命令后直接在 gdb 提示符下按回车键重复上一条命令。

```
1          size = strlen (string);
2          string2 = (char *) malloc (size + 1);
3          for (i = 0; i < size; i++)
4            string2[size – i] = string[i];
5          string2[size+1] = `\0';
6          printf ("The string printed backward is %s\n", string2);
7        }
```

根据列出的源程序，需要到 4 行设置断点。在 gdb 命令行提示符下输入如下命令设置断点：

```
(gdb) break 4
```

现在再输入“run”命令，将产生如下输出：

```
Starting program:/root/greeting
The string is hello there
Breakpoint 1, my_print2 (string = 0xbfffdc4 "hello there") at greeting.c :4
4   string2[size-i]=string[i]
```

通过设置一个观察 string2[size – i] 变量的值的观察点来看出错误是怎样产生的，做法是输入：

```
(gdb) watch string2[size – i]
Watchpoint 2: string2[size – i]
```

现在可以用 next 命令来逐步执行 for 循环：

```
(gdb) next
```

经过第 1 次循环后，gdb 告诉程序员 string2[size – i]的值是'h'。

```
Watchpoint 2, string2[size – i]
Old value = 0 '\000'
New value = 104 'h'
my_print2(string = 0xbfffdc4 "hello there") at greeting.c:3
3 for (i=0; i<size; i++)
```

这个值正是期望的，后来的数次循环的结果都是正确的。当 i=10 时，表达式 string2[size - i] 的值等于'e'，size – i 的值等于 1，最后一个字符已经复制到新串里。如果再把循环执行下去，则没有值分配给 string2[0]，而它是新串的第 1 个字符。由于 malloc( )函数在分配主存时把它们初始化为空(null)字符，因此 string2 的第 1 个字符是空字符。这解释了为什么在打印 string2 时没有任何输出。找到问题所在之后，只要把代码里写入 string2 的第 1 个字符的偏移量改为 size–1，而不是 size。下面是修正后的代码：

```
#include   <stdio.h>
int main(int argc, char *argv[ ])
{
   char my_string[ ] = "hello there";
   my_print (my_string);
   my_print2 (my_string);
   return 0;
}

void my_print (char *string)
{
   printf ("The string is %s\n", string);
}
```

```
void my_print2 (char *string)
{
   char *string2;
   int size, size2, i;
   size = strlen (string);
   size2 = size –1;
   string2 = (char *) malloc (size + 1);
   for (i = 0; i < size; i++)
      string2[size2 – i] = string[i];
   string2[size] = '\0';
   printf ("The string printed backward is %s\n", string2);
}
```

第2章　进程与线程

2.1　实 验 目 的

- 加深理解进程和程序、进程和线程的联系与区别。
- 深入理解进程和线程的重要数据结构及实现机制。
- 熟悉进程及线程的创建、执行、阻塞、唤醒、终止等控制方法。
- 学会使用进程及线程开发应用程序。

2.2　背 景 知 识

2.2.1　进程与线程的概念

进程和线程都是现代操作系统中程序运行的基本单位，多用户、多任务操作系统利用进程和线程来实现系统对应用任务的并发性。通俗地讲，程序是一个包含可执行代码的文件，是一种存在于系统中的静态资源；进程是一个具有独立功能的程序关于某个数据集合上的一次并发执行的运行活动，是一种有生命周期的动态实体，是支持程序执行的一种系统机制。在单线程结构进程中，进程作为构成系统的基本实体，既是内部独立的执行单元，又是独立竞争资源的基本单元。在多线程结构进程中，进程是系统进行资源分配和保护的基本单元，而线程是进程内独立的执行单元，即一条执行路径。线程包含独立的堆栈和处理器及寄存器状态，每个线程共享其所附属进程的所有资源，包括打开的文件、页表(因此也就共享整个用户地址空间)、信号等。一个程序可对应多个进程，而每个进程又可以有许多子进程，依次循环下去，产生子孙进程，形成进程族系。

线程和进程的关系主要包括以下几个方面。

① 进程是资源分配和管理的基本单位，线程是程序执行的独立单位。

② 进程在执行过程中拥有独立的主存空间，而线程不能够独立存在，必须运行在所属进程的地址空间内。

③ 线程属于进程的组成部分，进程可包含多个线程。当进程被撤销时，该进程所产生的线程都会被强制撤销。

在操作系统设计上，从进程演化出线程的最主要目的就是更好地支持多处理器系统，多线程程序设计的优点是使系统性能获得提高，具体表现在快速线程切换、节省主存空间、减少管

理开销、通信易于实现、并发程度提高等几方面。

图 2-1 和图 2-2 分别给出同一应用任务基于并发进程和并发线程实现的系统内部结构。对比分析两图可见，进程之间的关系比较疏远，各进程均运行于自己独立的地址空间内，不但寄存器和堆栈是独立的，动态堆、静态数据和程序代码也相互独立。而线程之间的关系则要紧密得多，虽然各线程为保持自己的控制流而独自具有寄存器和堆栈，但由于两个线程从属于同一进程，它们共享该进程的地址空间，所以动态堆、静态数据及程序代码为各线程共享。从这个意义来看，进程是独立的资源分配和保护的实体，为线程提供运行所需的资源并构成静态环境，线程则是处理器调度的独立单位。

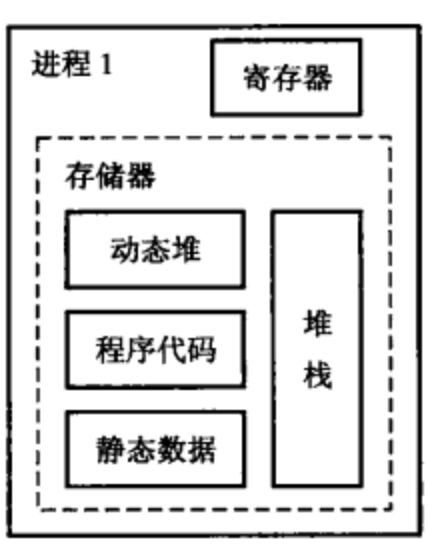
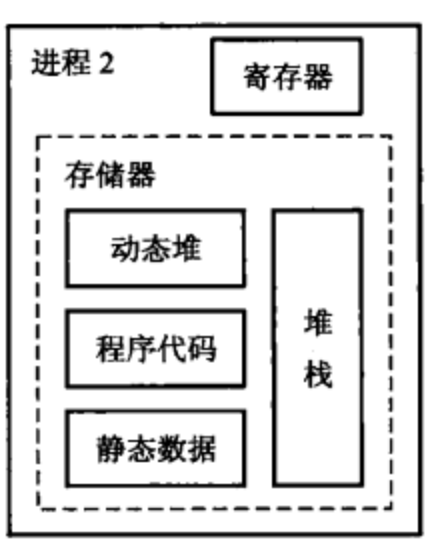


图 2-1 并发进程结构



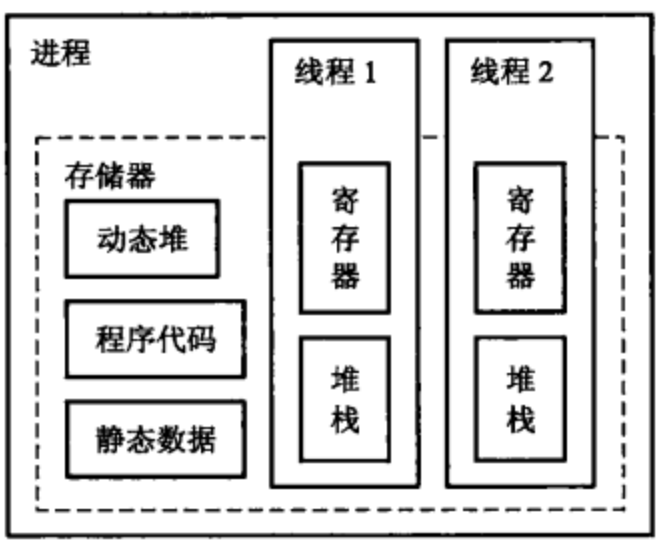


图 2-2 并发线程结构

可执行程序通常以文件形式存储在磁盘上，当程序被调入到主存后，系统会给它分配一定数量的资源(主存和设备等)，然后进行一系列复杂操作。其中最重要的是，由内核为其分配一个称为进程控制块(Process Control Block，PCB)的内核数据结构，填入所需各种信息，从而创建一个执行此程序的进程。进程是操作系统中“计算活动的独立实体”，具备在多道程序环境中运行的条件，可被调度程序调度执行。

与传统的进程概念一致，Linux 进程也由 4 个要素组成。

① 进程控制块。它是内核数据结构，每个进程捆绑一个，用来存储进程标志信息、现场信息和控制信息。进程创建时建立进程控制块，进程撤销时回收进程控制块，它与进程一一对应，v2.4 以前，它被包含在内核栈中。

② 进程程序块。它存放进程执行的指令代码，未必是某一进程所专有的，可与其他进程共享，故程序和进程具有一对多的对应关系，具有可读、可执行、不可写属性。

③ 进程内核栈。它又称为核心栈，每个进程捆绑一个，进程在内核态下工作时使用，用来保存中断/异常现场以及执行函数调用时存放参数和返回地址等，具有可读、可写、不可执行属性。

④ 进程数据块。它是进程专用地址空间，存放各种私有数据，用户栈也位于数据块中，

用于函数调用时存放栈帧和局部变量等参数，具有可读、可写、不可执行属性。

如果缺少其中一项，就不成为一个“进程”。如果只具备前 3 项而缺少第 4 项，则称之为“线程”；如果完全没有用户空间，则称为“内核线程”；如果共享用户空间，则称为“用户线程”。

Linux 支持传统 UNIX 进程概念，每个进程用 task_struct 描述。进程不但拥有资源，而且也参与调度。最初的进程由系统在初始化时生成，而此后的进程均由已有进程通过进程创建函数来创建。Linux 中的线程机制十分独特，从内核角度来说并没有线程概念，故没有为线程单独定义数据结构。通过 clone( )创建的 Linux 线程，仅被看做是一个与其他进程共享某些资源的特殊进程而已。线程(子进程)也使用 task_struct 描述，它与父进程共享同一个地址空间及打开文件表等信息。进程 ID 号就是其所拥有的线程组的线程组号(TGID)，而每个线程的 ID 号为用 ps 命令所看到的子进程号。此外，将线程调度等同于进程调度，也交给内核完成。当然，子进程可建立自己独立的地址空间，与父进程分离，真正成为传统意义上的进程。至于线程取消和线程同步等工作，都由在核外用户态下运行的 pthread 线程库完成。实质上，Linux 中线程只是一种进程间共享资源的手段。Linux 也支持内核线程，它们由内核在初始化时创建，永远在内核态下运行，没有用户空间且共享内核地址空间，它们完成页面换出和磁盘缓冲刷新等系统任务。

注意区分以下两种情况，不要与 Linux 线程相混淆。

① 在用户空间中，同一进程内由线程库支持与实现的“用户线程”。它不拥有独立地址空间和专用内核栈，也不作为一个独立单元接受内核调度，内核只调度用户进程，这些进程再通过线程库函数来调度应用程序的线程。由于 Linux 内核已提供对线程的支持，一般来说应用程序没有必要再在进程内派生这种用户线程。

② 在 Windows 中，使用 CreateProcess( )创建进程，操作系统为新进程分配新的地址空间，并创建一个基线程。基线程可以通过 CreateThread( )创建其他线程，这些线程共享进程的资源和地址空间，并被独立调度。Windows 线程通常又被称作轻量进程，相对于重量进程来说，线程被抽象成耗费资源少、可被调度执行的独立单元。与 Linux 不相同，Windows 内核不但提供对进程的支持，也提供对进程内的线程的支持。

2.2.2 多进程编程

2.2.2.1 Linux 进程结构

1. Linux 进程结构简介

Linux 操作系统使用一个称为进程控制块的数据结构 task_struct 来代表一个进程。该结构包含进程的所有信息，是系统掌握的进程的重要信息，是系统对进程进行管理和控制的有效手段，也是实现进程调度的主要依据。当一个进程被创建时，系统为该进程建立一个 task_struct 结构；当进程运行结束时，系统撤销该进程的 task_struct 结构。Linux 在操作系统的内核空间设置一个 task 数组，该数组的每个元素是一个指向任务结构的指针。因此，task 数组又称为 task 向量。

该 task 数组结构定义如下：

```
struct task_struct *task[NR_TASKS] = {&init_task};
#define NR_TASKS 512
```

全局变量 NR_TASKS 记录系统可容纳的进程数，默认大小是 512，表明在 Linux 系统中最多能同时运行 512 个进程。task_struct 在 include\Linux\sched.h 文件中定义，其主要成员说明如下：

```
struct task_struct {
    volatile long state;                        /*进程状态*/
    struct thread_info *thread_info;            /*指向进程的 thread_info 指针*/
    atomic_t usage;
    unsigned long flags;                        /*进程标志共 23 种，如正被创建或被信号杀死等*/
    unsigned long ptrace;                       /*跟踪标志*/

    int lock_depth;                             /*锁的深度*/

    int prio, static_prio, normal_prio;         /*动态、静态和正常优先级*/
    struct list_head run_list;                  /*链接优先级数组 prio_array 中的元素*/
    struct prio_array *array;                   /*当前 CPU 活动的就绪队列*/

    unsigned long sleep_avg;                    /*进程的平均等待时间(以毫微秒为单位)*/
    long interactive_credit;                    /*进程发生调度事件的时间(以毫微秒为单位)*/
    unsigned long long timestamp, last_ran;
    int activated;                              /*进程进入就绪态的原因*/

    unsigned long policy;                       /*调度策略*/
    cpumask_t cpus_allowed;
    unsigned int time_slice, first_time_slice;/*时间片余额：0/1 表示是否第 1 次分得时间片*/

    struct list_head tasks;                     /*任务队列*/

    struct list_head ptrace_children;           /*跟踪进程使用情况*/
    struct list_head ptrace_list;
    struct mm_struct *mm, *active_mm;           /*进程虚拟主存信息；内核线程借用的地址空间*/
    struct Linux_binfmt *binfmt;
    int exit_code, exit_signal;                 /*退出信号，系统强行退出时发出的信号*/
    int pdeath_signal;                          /*父进程死亡时发出该信号*/

    pid_t pid;                                  /*进程 ID 和组 ID*/
    pid_t tgid;                                 /*sessionID*/
    struct task_struct *real_parent;            /*调试时的真父进程*/
    struct task_struct *parent;                 /*父进程*/
```

```
struct list_head children;                  /*子进程链表*/
struct list_head sibling;                   /*兄弟进程链表*/
struct task_struct *group_leader;           /*线程组的领导者*/

struct pid_link pids[PIDTYPE_MAX];          /*pid 哈希表链*/
wait_queue_head_t wait_chldexit;            /*wait4( )使用*/
struct completion *vfork_done;              /*vfork( )使用*/
int _ _user *set_child_tid;                 /*CLONE_CHILD_SETTID*/
int _ _user *clear_child_tid;               /*CLONE_CHILE_CLEARED*/

struct list_head thread_group;              /*线程队列链*/
unsigned long rt_priority;                  /*实时进程优先级*/
unsigned long it_real_value，it_prof_value，it_virt_value;
unsigned long it_real_incr，it_prof_incr，it_virt_incr;
struct timer_list real_timer;
unsigned long utime,stime;
unsigned long nvcsw, nivcsw;                /*上下文切换计数*/
struct timespec start_time;                 /*进程创建时间*/

cputime_t it_prof_expires, it_virt_expires;/*进程的 ITIMER_PROF 和 ITIMER_VIRTUAL 定时器*/
unsigned long long it_sched_expires;
struct list_head cpu_timers[3];

uid_t uid,euid,suid,fsuid; /*用户的标识符、有效标识符、备份标识符、网络环境下文件标识符*/
gid_t gid,egid,sgid,fsgid; /*组的标识符、有效标识符、备份标识符、网络环境下文件标识符*/
struct group_info *group_info;
kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
unsigned keep_capabilities:1;
struct user_struct *user; /*进程定义的用户信息，包括该用户使用的进程数目、打开文件数等*/
struct rlimit rlim[RLIM_NLIMITS];
unsigned short used_math;
char comm[16];

int link_count, total_link_count;           /*文件链接数*/
struct thread_struct thread;                /*该任务与特定 CPU 相关的状态*/
struct fs_struct *fs;                       /*文件系统信息*/
struct files_struct *files;                 /*打开文件表指针*/
struct nsproxy *nsproxy;                    /*命名空间*/

struct sysv_sem sysvsem;                    /*ipc 信息*/
```

```
struct signal_struct *signal;        /*信号结构，对每种信号，各进程由 signal 属性选择处理函数。
                                     信号的检查在函数结束后或在“慢中断”中断服务程序结束后进行*/
struct sighand_struct *sighand;
sigset_t blocked, real_blocked;      /*进程接收信号的位掩码。置位表示屏蔽，复位表示不屏蔽*/
sigset_t saved_sigmask;              /*与 TIF_RESTORE_SIGMASK 共同恢复*/
struct sigpending pending;

unsigned long asa_ss_sp;
size_t sas_ss_size;
int (*notifier) (void &priv);
void *notifier_mask;

void security;
struct audit_context *audit_context;
u32 parent_exec_id;                  /*线程组跟踪*/
u32 self_exec_id;

spinlock_t alloc_lock;               /*分配 mm、files、fs、tty 等的自旋保护锁*/
spinlock_t proc_lock;                /*保护 proc_dentry*/
spinlock_t switch_lock;              /*上下文切换的自旋保护锁*/

        void *journal_info;
        struct reclaim_state *reclaim_state;
        struct dentry *proc_dentry;
        struct backing_dev_info *backing_dev_info;

        struct io_context *io_context;
        unsigned long ptrace_message;
        siginfo_t *last_siginfo;             /*跟踪使用情况*/
        wait_queue_t *io_wait;
        …
    }
```

2. 进程控制块中的主要成员

(1) 进程状态(state)

进程可有运行状态(0)、不可运行状态(-1)和暂停状态(>0)。

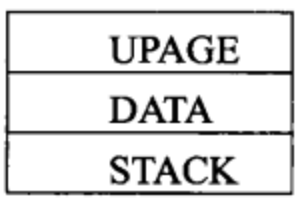
(2) 进程标志(flags)

进程标志描述进程管理信息或过渡状态，如进程正在创建、开始关闭、被信号杀死等，有以下几种定义：

```
#define PF_STARTING          0x00000002    /*进程被创建*/
```

```
#define PF_EXITING          0x00000004    /*进程开始关闭*/
#define PF_FORKNOEXEC       0x00000040    /*刚创建还没开始运行*/
#define PF_SUPERPRIV        0x00000100    /*超级用户特权*/
#define PF_DUMPCORE         0x00000200    /*标志进程是否清空 core 文件*/
#define PF_SIGNALED         0x00000400    /*进程被信号终止*/
#define PF_MEMALLOC         0x00000800    /*进程正在分配主存*/
#define PF_VFORK            0x00001000    /*对于用函数 vfork( )创建的进程，退出前正在唤醒
                                            的父进程*/
#define PF_USEDFPU          0x00100000    /*该进程使用 FPU，此标志只在 SMP 时使用*/
```

core 文件结构如下：



UPAGE 是包含用户结构的一个页面，gdb 调试程序的调试内容及所有寄存器的值都存在 UPAGE 中。DATA 是数据区，STACK 是堆栈区。最小 core 文件长度是 3 页。

(3) 跟踪标志(ptrace)

跟踪标志包括以下几种状态：

```
#define PF_PTRACED          0x00000010          /*进程跟踪标志*/
#define PF_TRACESYS         0x00000020          /*正在跟踪函数*/
#define PF_DTRACE           0x00200000          /*进程延期跟踪标志*/
```

(4) 进程优先级(prio、static_prio、rt_priority)

进程优先级包括静态优先级、动态优先级、实时优先级、优先级数组等信息。与其他操作系统类似，Linux 也支持普通进程和实时进程两类。实时进程具有时间敏感性，要求对外部事件作出快速响应，而普通进程则没有这种限制。所以，调度程序要区分对待两类进程，通常，实时进程要比普通进程优先运行。

prio_array 中包含一个就绪队列数组，数组的索引是进程优先级(共 140 级)。相同优先级的进程放置在相应数组元素的链表 queue 中。调度时直接查找就绪队列 active 中具有最高优先级的链表中的第 1 项作为候选进程，而优先级的计算过程则分散到各个进程执行过程中完成。在 v2.6 中，动态优先级独立计算，通过 prio_array 结构按优先级排序，存储在进程的 task_struct 中。此外，v2.6 中不再存储 nice 值，而代之以 static_prio。prio_array 将以进程优先级为序号组成数组。有关优先级的进一步信息请参见第 15 章。

(5) 进程进入就绪态原因(activated)

该原因会影响到调度优先级的计算，其取值包括以下几种。

- –1：进程从 TASK_UNINTERRUPTIBLE 状态唤醒。
- 0：默认值，进程原本就处于就绪状态。

• 1：进程从 TASK_INTERRUPTIBLE 状态被唤醒，且不在中断上下文中。

• 2：进程从 TASK_INTERRUPTIBLE 状态被唤醒，且在中断上下文中。

activiated 的初值为 0，在两处修改：一是在 schedule( )中被恢复成 0；另一处是通过 activate_task( )激活休眠进程。如果是中断服务程序调用 activate_task( )，即进程由中断激活，则该进程最有可能是交互式的，因此，置 activated=2；否则置 activated=1。如果进程是从 TASK_UNINTERRUPTIBLE 状态中被唤醒，则 activated= –1(在 try_to_wake_up( )函数中)。

(6) 进程平均等待时间(sleep_avg)

该时间在 0 到 NS_MAX_SLEEP_AVG 之间取值，初值为 0，相当于进程等待时间与运行时间的差值。sleep_avg 所代表的含义比较丰富，既可用于评价该进程的“交互程度”，又可用来表示该进程需要运行的紧迫性。这个值是动态优先级计算的关键因子，sleep_avg 值越大，计算出来的优先级值越小，则相应进程的优先级也越高。

(7) 进程交互程度(interactive_credit)

该变量记录本进程的“交互程度”，在-CREDIT_LIMIT 到 CREDIT_LIMIT+1 之间取值。进程被创建时，初值为 0，随后根据不同条件加 1 或减 1。一旦超过 CREDIT_LIMIT(只可能等于 CREDIT_LIMIT+1)，它就不会再降下来，表示进程已通过交互式测试，被认为是交互式进程。

(8) 运行列表(run_list)

优先级数组 prio_array 结构中按顺序排列各个优先级下的所有进程，但实际上数组中每个元素都是 list_head 结构，以它为表头的链表中的每一个元素也是 list_head，其中链接的是 task_struct 中的 run_list 成员。这是一个节省空间、加速访问的小技巧。调度程序在 prio_array 中找到相应的 run_list，然后通过 run_list 在 task_struct 中的固定偏移量找到对应的 task_struct。

(9) 进程运行环境信息(thread_info)

它包含 task_struct 指针、运行域、底层标志、线程同步标志、当前 CPU 号、内核抢占计数器、虚地址空间上限等。其中有两个结构成员与调度关系紧密：

• preempt_count：初值为 0 的非负计数器，大于 0 表示内核不宜被抢占。

• flags：其中有一个 TIF_NEED_RESCHED 位，相当于 v2.4 中的 need_resched 属性，如果当前运行中的进程此位为 1，则表示应该尽快启动调度程序。

Linux 进程的内核栈和 task_struct 数据结构之间存在紧密联系，二者在物理存储空间中也关联在一起。在 v2.4 中，内核为每个进程分配 task_struct 结构的主存空间时，一次分配两个连续主存页帧(8 KB)。其底部约 1 KB 空间用于存放 task_struct 结构，而上面约 7 KB 空间用作进程核心栈。task_struct 存放在内核栈尾端，目的是让像 x86 这样寄存器较少的硬件体系结构只要通过栈指针就能计算出 task_struct 的位置，从而避免使用额外寄存器专门进行记录。在 v2.6 中，仍然需要频繁访问宏 current，但已经改变为通过 slab 分配器动态为进程生成 task_struct，如图 2-3 所示。每个进程的运行环境信息 thread_info 结构在内核栈尾端分配，该结构中的 task 成员是指向该进程的 task_struct 指针。这样做的好处是仅将最关键的、访问最频繁的运行环境保存在内核栈里，而将 task_struct 大部分内容通过 thread_info 的 task 指针保存在内核栈之外，以方

便扩充。thread_info 的分配方式和访问方式与 v2.4 中的 task_struct 完全相同。宏 current 所对应的指针是通过堆栈指针 esp 计算出来的，首先去掉 esp 的末尾 13 位得到 thread_info 的地址，再通过 thread_info->task 得到 task_struct 的地址。current 的实现如下：

```
static inline struct task_struct * get_current(void)
{
    return current_thread_info( )->task;
}
#define current get_current( )
```

其中 current_thread_info( )定义为：

```
/*[include/asm-i386/thread_info.h]*/
static inline struct thread_info *current_thread_info(void)
{
    struct thread_info *ti;
    __asm__("andl %%esp,%0; ":"=r" (ti) : "0" (~8191UL));
    return ti;
}
```

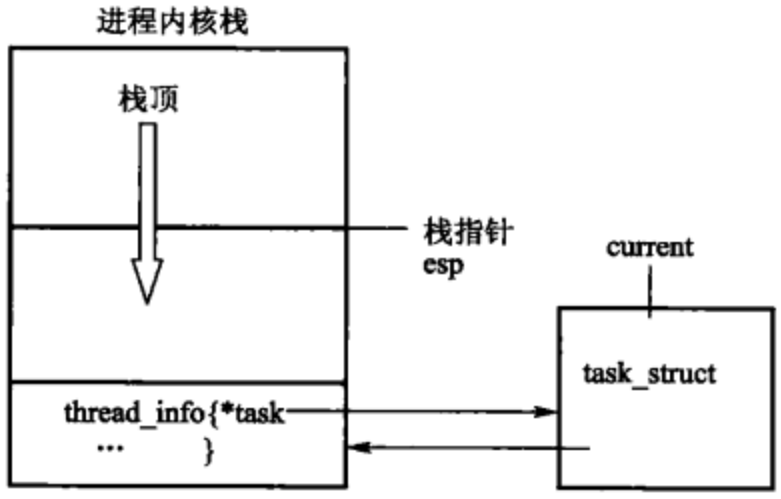


图 2-3 进程内核栈、*task 和 task_struct 结构

(10) 运行时间片剩余大小(time_slice)

在 v2.6 中，time_slice 变量代替 v2.4 中的 counter 变量来表示进程剩余运行时间片，time_slice 尽管拥有和 counter 相同的含义，但在内核中的表现行为已经大相径庭。主要表现在以下 3 个方面。

1) time_slice 基准值

和 counter 类似，进程的默认时间片与进程的静态优先级(在 v2.4 中是 nice 值)相关，使用如下公式得出：

MIN_TIMESLICE + ((MAX_TIMESLICE–MIN_TIMESLICE)×
(MAX_PRIO–1–(p)->static_prio) / (MAX_USER_PRIO–1))

内核将 100～139 的优先级映射到 800 ms～5 ms 的时间片上，优先级数值越大，则分配的时间片越小。

2) time_slice 的变化

进程的 time_slice 值代表进程的运行时间片剩余大小，在进程创建时与父进程平分时间片，在运行过程中递减。一旦归 0，则按 static_prio 值重新赋予上述基准值并请求调度。时间片的递减和重置在时钟中断中进行(sched_tick( ))，除此之外，time_slice 值的变化主要在创建进程和进程退出过程中。

① 进程创建。与 v2.4 类似，为防止进程通过反复调用 fork( )来偷取时间片，子进程被创建时并不另外获得时间片，而是与父进程平分父进程的剩余时间片。也就是说，fork( )结束后，两者时间片之和与原先父进程的时间片相等。

② 进程退出。进程退出时(sched_exit( ))，根据 first_time_slice 的值判断自己是否从未重新分配过时间片，如果是，则将自己的剩余时间片返还给父进程(保证不超过 MAX_TIMESLICE)。这个动作使进程不会因创建短期子进程而受到惩罚(与不至于因创建子进程而受到“奖励”相对应)。如果进程已经用完从父进程处分得的时间片，就没有必要返还。

3) time_slice 对调度的影响

在 v2.4 中，进程剩余时间片是除 nice 值以外对动态优先级影响最大的因素，也是休眠次数多的进程。它的时间片会不断叠加，从而计算出的优先级也更大，调度程序正是用这种方式来体现对交互式进程的优先策略。但实际上，休眠次数多并不表示该进程就是交互式的，只能说明它是 I/O 密集型的，因此这种方法精度很低。有时因为误将频繁访问磁盘的数据库应用当作交互式进程，反而造成真正的用户终端响应迟缓。v2.6 调度程序以时间片是否耗尽为标准将就绪进程分成 active 和 expired 两大类，分别对应不同的就绪队列。前者相对于后者拥有绝对的调度优先权，仅当 active 进程时间片都耗尽，expired 进程才有机会运行。但在 active 中挑选进程时，调度程序不再将进程剩余时间片作为影响调度优先级的一个因素。此外，为了满足内核可剥夺要求，时间片太长的非实时交互式进程还会被人为地分成好几段(每段称为一个运行粒度)运行。每段运行结束后，它都从 CPU 上被剥夺下来，放置到对应的 active 就绪队列的末尾，为其他具有相同优先级的进程提供运行机会。这一操作在 schedule_tick( )对时间片递减之后进行。此时，即使进程的时间片没耗完，只要该进程同时满足以下 4 个条件，就会被强制从 CPU 上剥夺下来，重新入队等候下一次调度。

① 进程当前在 active 就绪队列中。

② 该进程是交互式进程(TASK_INTERACTIVE( )返回真，nice 大于 12 时，该宏返回恒假)。

③ 该进程已经耗掉的时间片(时间片基准值减去剩余时间片)正好是运行粒度的整数倍。

④ 剩余时间片不小于运行粒度。

(11) 调度策略(policy)

它的定义值如下：

```
#define SCHED_OTHER 0        /*非实时进程，基于优先级的轮转法*/
```

```
#define SCHED_FIFO 1          /*实时进程，用先进先出算法*/
#define SCHED_RR 2            /*实时进程，用基于优先级的轮转法*/
```

(12) 信号管理成员

信号管理成员包括 blocked、signal、sighand、pending、exit_code、exit_signal 和 pdeath_signal 等，用于实现进程间通信。

(13) 进程标识成员

进程标识成员包括 uid、suid、fsuid、gid、egid、sgid、fsgid、pid、pgrd、session 和 tgid 等信息，用于标记进程及所属用户信息。

(14) 进程族系关系成员

在 Linux 系统中，除初始化进程外，每个进程都有父进程，该链接信息包括指向父进程、兄弟进程和子进程的指针，它们是 children、sibling 和 parent。

(15) 时间和定时器成员

内核要记录进程创建时间和进程运行所占用 CPU 的时间。Linux 系统支持进程的逻辑定时器，包括 start_time、real_timer、utime、stime、it_real_value、it_prof_value 和 it_virt_value 等。

(16) 文件系统成员

进程在运行时可以打开和关闭文件。task_struct 结构中包括指向每个打开文件的文件描述字指针 files，还包含进程的可执行映像所在的文件系统信息 fs，它包括两个指向 VFS 索引节点的指针，第 1 个索引节点是进程的根目录，第 2 个索引节点是当前工作目录。两个 VFS 索引节点都有一个计数字段用来计算指向节点的进程数。

(17) 虚拟主存信息成员

进程都使用虚拟主存空间，Linux 系统必须知道如何将虚拟主存映射到系统的物理主存，使用的成员有 mm 和 active_mm。

2.2.2.2 进程标识符信息

用户输入任何 Linux 命令时，对应的 Shell 通常会建立子进程来运行该命令。因此，可把任何在 Linux 系统中运行的程序叫做进程，并通过使用 ps(process status)命令来查看用户空间的当前进程。

```
$ ps
PID        TTY        TIME        CMID
1120       pts/1      00:00:12    bash
1232       pts/1      00:00:00    ps
```

上述命令显示，有两个进程正在运行，即 bash(shell)和 ps。每个进程都有一个标志它的数字，称为进程 ID。进程 ID 是一个非负数，在任何时刻都是唯一的，被系统用来区分和管理不同进程。进程 ID 由系统分配，不能被修改。当某个进程结束时，其进程 ID 可回收，以分配给新进程使用。通过函数 getpid( )可以得到进程 ID，通过函数 getppid( )可以得到父进程(创建调用

该函数进程的进程)的进程 ID。为对具有某些类似特性的进程统一管理，Linux 引入进程组概念，以组标识符来区别进程是否同组。进程的组 ID 是从父进程继承的，因此，通常进程的组 ID 就是和它相关联的注册进程的 ID。组 ID 可通过函数 setpgrp( )修改，通过函数 getpgrp( )或 getpgid( )获取。

在操作系统中，进程为程序服务，而程序为用户服务，因此，进程通常从属于特定用户。为在进程与用户之间建立联系，系统还为每个进程提供进程所有者 ID，可通过函数 getuid( )得到进程所有者 ID。此外，为保护系统资源，Linux 系统还为每个进程提供一个有效用户 ID，用于控制进程的资源访问权限，通过函数 geteuid( )可以得到进程的有效用户 ID。与用户 ID 相对应，进程还有一个有效组 ID，可通过函数 getegid( )得到有效组 ID。

所有进程标识符信息都记录在 task_struct 中，相关函数的原型定义如下：

```
#include <sys/types.h>
#include <unistd.h>
uid_t getpid(void)          /*获取进程 ID*/
uid_t getppid(void)         /*获取父进程 ID*/
pid_t getpgrp(void)         /*获取进程组 ID*/
pid_t getpgid(pid_t pid);   /*获得指定 pid 进程所属组的 ID*/
uid_t getuid(void)          /*获取进程所有者 ID*/
uid_t geteuid(void)         /*获取进程的有效用户 ID*/
git_t getegid(void);        /*获取进程的有效组 ID*/
```

注意，tgid 的赋值通过以下代码完成：

```
p->tgid = p->pid;
if (clone_flags & CLONE_THREAD)
    p->tgid = current->tgid;
```

对于单线程进程来说，TGID 和 PID 相等。对于多线程进程来说，同一线程组内的所有线程的 TGID 都相等且等于父进程的 PID。

以下代码(getpidtest.c)实例显示获取进程 ID 的方法：

```
/*****getpidtest.c*****/
#include <sys/types.h>
int main(int argc,char **argv)
{
    pid_t my_pid,parent_pid;
    uid_t my_uid,my_euid;
    gid_t my_gid,my_egid;

    my_pid=getpid( );
    parent_pid=getppid( );
    my_uid=getuid( );
    my_euid=geteuid( );
```

```
        my_gid=getgid( );
        my_egid=getegid( );

        printf("Process ID:%ld\n",my_pid);
        printf("Parent ID:%ld\n",parent_pid);
        printf("User ID:%ld\n",my_uid);
        printf("Effective User ID:%ld\n",my_euid);
        printf("Group ID:%ld\n",my_gid);
        printf("Effective Group ID:%ld\n",my_egid):
    }
```

该代码在某计算机上的运行结果如下：

```
Process ID:6634
Parent ID:6593
User ID:1000
Effective User ID:1000
Group ID:1000
Effective Group ID:1000
```

2.2.2.3 Linux 进程状态

Linux 将进程分成不同进程状态，并通过 task_struct 中 state 成员来表示，Linux 定义了 7 种进程状态。

① TASK_RUNNING 0：正在运行的进程或等待被调度执行的进程。task_struct 中的 run_list 成员把所有处于 TASK_RUNNING 状态的进程链接在一起，称为可运行队列或就绪队列。current 指向当前正在 CPU 上执行的进程的 task_struct 指针，通过 current–>pid 就可以获得当前执行进程的 ID。

② TASK_INTERRUPTIBLE 1：处于等待队列中的进程，待资源满足时唤醒，也可被信号唤醒，然后变成就绪状态。

③ TASK_UNINTERRUPTIBLE 2：处于等待队列中的进程，直接等待硬件条件，待资源有效时唤醒，不可通过信号唤醒。

④ TASK_STOPPED 4：进程被暂停，通过其他进程的信号才能唤醒。正在调试的进程收到信号 SIGSTOP 会处在停止状态，收到 SIGCONT 信号时恢复运行。

⑤ TASK_TRACED 8：进程处于跟踪状态。

⑥ TASK_ZOMBIE 16：终止的进程，是进程结束前的过渡状态(僵死状态)。虽然此时已经释放主存、文件等资源，但是在 Task 向量表中仍有 task_struct 数据结构项，它不进行任何调度或状态转换，等待父进程将它注销。

⑦ TASK_DEAD 32：该进程已经退出，且不需要父进程来回收。

注意，TASK_TRACED 和 TASK_DEAD 是 v2.6 中新增的进程状态。

TASK_INTERRUPTIBLE 和 TASK_UNINTERRUPTIBLE 两种状态的进程被细分为多类。每类对应一种特定事件，并采用链表方法形成多个等待队列，以方便在事件发生时遍历等待队列，唤醒相应进程，并将该进程改变为 TASK_RUNNING 状态。图 2-4 描述 Linux 系统中进程状态及转换函数图。

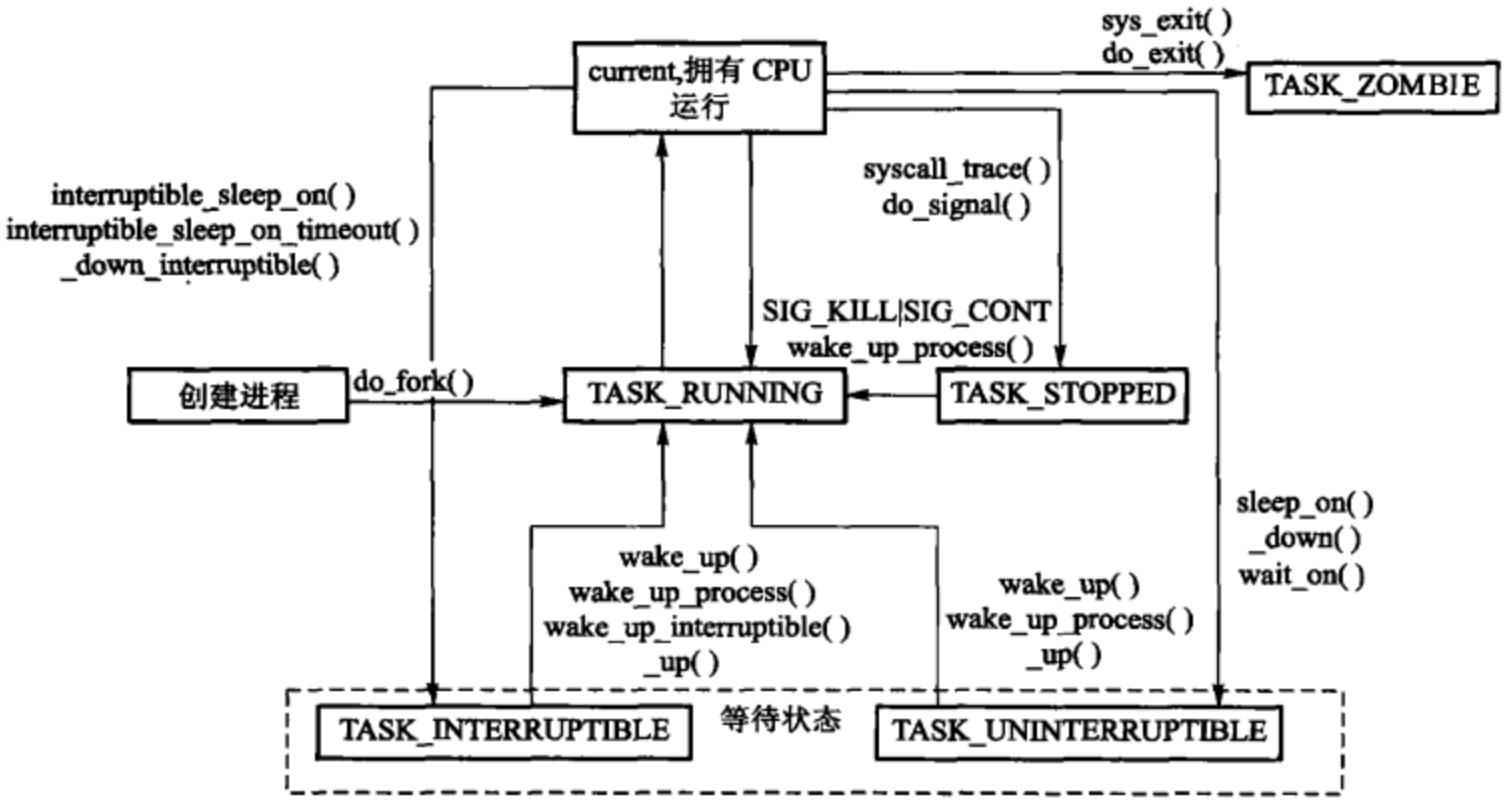


图 2-4 Linux 系统中进程状态及转换函数

kernel/sched.c 中的代码提供一组函数，用来管理进程队列。当拥有 CPU 的进程申请资源不能满足时，会通过 sleep_on( )或_down( )，将进程从 TASK_RUNNING 状态切换到 TASK_UNINTERRUPTIBLE 状态。一般来说引起状态变成 TASK_UNINTERRUPTIBLE 的资源申请都是对一些硬件资源的申请，如果得不到这些资源，进程将不能执行下去。因此，处于 TASK_UNINTERRUPTIBLE 状态的进程不能由 signal 信号或时钟中断唤醒，只能由 wake_up( )、_up( )或 wake_up_process( )唤醒。wake_up( )函数的作用是将 wait_queue 中的所有状态为 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE 的进程状态都置为 TASK_RUNNING，并将它们都放到 running 队列中去，即唤醒所有等待在该队列上的进程。

与 sleep_on( )非常相似，当拥有 CPU 的进程申请资源不能满足时，会通过 interruptible_sleep_on( )或_down_interruptible( )，将进程从 TASK_RUNNING 切换到 TASK_INTERRUPTIBLE 状态。进行这种转换的原因基本上与 sleep_on( )相同，申请资源无效时进程切换到等待状态。但与之不同的是，处于 interruptible_sleep_on 等待状态的进程是可以接受信号或中断而重新变为 running 状态。处于 TASK_INTERRUPTIBLE 状态的进程可以在资源有效时被 wake_up( )、wake_up_interruptible( )、_up( )或 wake_up_process( )唤醒，或收到信号以及时间中断后被唤醒。

TASK_STOPPED 是暂停状态，需要和进程标志 PF_PTRACED 及 PF_TRACESYS 配合使用，分别表示被跟踪和正在跟踪函数，一个是被动的，一个是主动的。进程可通过两种途径进入 TASK_STOPPED 状态。一种途径是受其他进程的 syscall_trace( )的控制而暂时将 CPU 交给控制进程。另一种进入 STOPPED 状态的方法是信号，SIGSTOP 信号使 current 进程打上 PF_PTRACED 标记，并将它转入 STOPPED 状态。do_signal( )在检查进程收到的信号时，若发现 current 进程已打上 PF_PTRACED 标记，则除收到 SIGKILL 信号外，current 进程都将马上进入 STOPPED 状态。从 TASK_STOPPED 状态转到 TASK_RUNNING 状态通过“信号唤醒”。当有 SIGKILL 或 SIGCONT 信号发给 TASK_STOPPED 状态下的进程时，进程将被 wake_up_process( )唤醒。

进程终止由可终止进程的函数通过调用内核函数 do_exit( )实现，它终止 current 进程，首先为 current 进程置 PF_EXITING 标记，释放 current 进程的存储管理信息、文件信息、信号响应函数指针数组，将状态置为 TASK_ZOMBIE，通知 current 指向进程的父进程，最后进行重新调度。do_exit( )带一个参数，用于传递终止进程的原因。

2.2.2.4 进程控制

1. 创建进程

Linux 中在下列服务请求情况下，需要创建新进程。

- 从后备批处理作业中选择新作业时，将创建一个新进程处理该作业。
- 终端用户登录到系统时，将为其创建一个新进程。
- 用户请求系统服务时，如打印文件，将为其创建一个系统(打印)进程。
- 一个进程显式要求创建另一个进程，以完成并发操作。

系统允许一个进程创建新进程，新进程即为子进程，而生成子进程的进程称为父进程。子进程还可以创建新的子进程，形成进程树结构模型。树根是系统自动构造的、在内核态下执行的 0 号进程，它是所有进程的祖先。由 0 号进程创建的在内核态执行的 1 号进程，负责执行内核的部分初始化工作及进行系统配置，并创建若干个用于高速缓存和虚拟主存管理的内核线程，如 kswapd 和 bdflush 等。随后，内核态 1 号进程调用 execve()运行可执行程序 init，并演变成用户态 1 号进程，即 init 进程，它按照配置文件/etc/initab 的要求，完成系统启动工作，创建编号为 1 号、2 号、……的若干个终端注册进程 getty。每个 getty 进程设置其进程组标识号，并监视配置到系统终端的接口线路。当检测到来自终端的连接信号时，getty 进程将通过函数 execve( )执行注册程序 login，此时用户就可输入注册名和密码进入注册过程。如果注册成功，由 login 程序再通过函数 execve( )执行 Shell，该 Shell 进程接收 getty 进程的 pid，即 Shell 进程取代原来的 getty 进程，该用户就可使用计算机工作了。如果因用户名或密码不符而注册失败，login 因超时而退出，并关闭打开的终端线路，然后 1 号用户进程将为该终端创建新的 getty 进程。所以，1 号进程 init 是 1 号 getty 进程(或 2 号、3 号……)的父进程；而 1 号 getty 进程是 1 号进程 init 的子进程，再由 1 号 getty 进程(或 2 号、3 号……)直接或间接生成用户空间的所有其他进程，

进程树中的所有进程都在系统中被并发执行。图 2-5 所示为 Linux 进程间的关系。

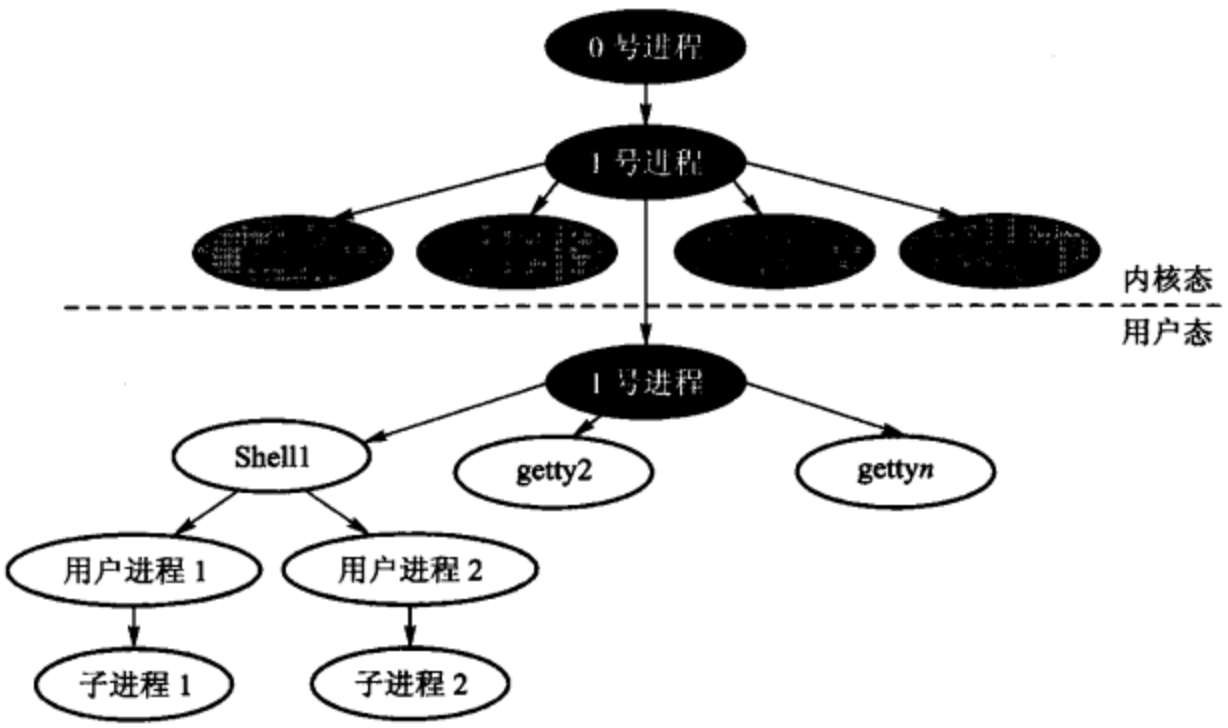


图 2-5 Linux 进程间的关系

由于执行 init( )函数的内核线程和 init 进程的进程标识符都是 1，它们又都叫 init，因此 init( )函数和 init 进程容易造成概念上的混淆。两者主要有以下 3 点区别。

- init( )函数是内核代码的一部分，在内核态运行。
- init 进程在 Linux 操作系统中是一个具有特殊意义的进程，它是由内核启动并运行的第 1 个用户进程，因此运行在用户态。它的代码不是内核本身的一部分，而是存放在磁盘上的可执行文件映像中，和其他用户进程没有什么两样。
- init( )函数调用 execve( )从文件/etc/inittab 中加载可执行程序 init 并执行，从此执行 init( )函数的 1 号内核进程演变成为 init 进程。也就是说由 init( )函数产生 init 进程，在这个演变过程中没有使用 fork( )。因此，init 进程的进程标识符仍然是 1 号内核进程的标识符 1。

从 0 号进程到 1 号用户进程的衍生过程可描述如下：

0 号进程→1 号内核线程→1 号内核进程→1 号用户进程(init 进程)

① 0 号进程：这是系统引导时自动形成的进程，实际上就是内核本身。它是系统中后来产生的所有进程的祖先。当内核系统完成自身初始化工作后，由内核本身调用函数 kernel_thread( )，它使用 int $0x80 创建第 1 个内核线程。

② 1 号内核线程：内核线程在 Linux 系统中是指没有虚拟存储空间的进程，它可以直接使用物理地址空间，运行在内核态中。在函数返回后，通过比较 ESP 和 ESI 寄存器的值来判断父、子进程。如果 ESP 寄存器的值就等于 ESI 寄存器的值，系统认为是父进程，就是内核本身，它就是 0 号进程。否则就是内核创建的第 1 个内核线程。所以此线程就是 1 号线程。

③ 1号内核进程：1号内核线程的程序控制该子程序直接去执行init( )函数，随后，1号线程将演变成1号内核进程。

④ init进程：init( )函数调用execve( )从文件/etc/inittab中加载可执行程序init并执行，从此执行init( )函数的1号内核进程演变成init进程。也就是说由init( )函数产生了init进程。在这个演变过程中没有使用fork( )，因此，init进程的进程标识符仍然是1号内核进程的标识符1。

用户如何从当前进程创建一个新进程呢?在Linux操作系统中，除初始化进程外，其他进程都用函数fork( )、vfork( )和clone( )创建新进程，这3个函数原型定义如下：

```
pid_t fork(void);
pid_t vfork(void);
int clone(int (*fn)(void * arg), void *stack, int flags, void * arg);
```

3个函数都调用同一个内核函数do_fork( )：

```
do_fork(unsigned long clone_flag, unsigned long usp, struct pt_regs);
```

其中clone_flag包括CLONE_VM、CLONE_FS、CLONE_FILES、CLONE_SIGHAND、CLONE_PID及CLONE_VFORK等标志位，任何一位被置1，则表明创建的子进程和父进程共享该位对应的资源。上述标志位的含义如下。

- CLONE_VM：父进程、子进程共享进程空间。
- CLONE_FS：父进程、子进程共享文件系统信息。
- CLONE_FILES：父进程、子进程共享打开的文件。
- CLONE_SIGCHLD：子进程终结或者暂停时给父进程发信号。
- CLONE_SIGHAND：父子进程共享信号处理函数。
- CLONE_PID：父进程、子进程共享进程标识符。
- CLONE_VFORK：父进程在子进程释放空间时被唤醒。

上述函数创建新进程时，对clone_flag的设置不同，内核函数do_fork( )完成的工作也不相同，因此，三者创建进程的实现机制不同。

do_fork( )的执行过程大致如下：

① 调用alloc_pidmap( )为子进程分配pid。

② 调用copy_process( )，它将完成创建的大部分工作(如调用dup_task_struct( )，而该函数又调用alloc_task_struct( )为子进程的task_struct结构分配内核栈)。

③ 调用alloc_thread_info( )为子进程分配thread_info结构。

④ 调用kmem_cache_alloc( )从slab中分配task_struct结构，并复制父进程的task_struct信息和thread_info信息。

⑤ 调用audit_alloc( )分配和初始化进程记账信息。

⑥ 依次调用copy_semundo( )、copy_files( )、copy_fs( )、copy_sighand( )、copy_signal( )、copy_mm( )、copy_namespace( )，分别复制并继承父进程的信号量信息、文件信息、信号处理信息、进程地址空间和命名空间，同时以上复制操作取决于clone标志的设置。

⑦ 调用copy_thread( )初始化子进程的内核栈。

⑧ 调用 sched_fork( )设置进程调度信息，将子进程状态设置为 TASK_RUNNING，并将父进程时间片余额的一半分给它，使用 SET_LINKS 将子进程插入运行队列。

⑨ 调用 attach_pid( )把子进程插入哈希队列，如果 clone_flags 包含 CLONE_STOPPED 标志，把子进程状态改为 TASK_STOPPED；否则调用 wake_up_new_task( )，它又调用 _activate_task( )将子进程加入运行队列。

至此，子进程创建完毕并在可运行队列中等待被调度执行；如果 clone_flags 包含 CLONE_VFORK 标志，则将父进程挂起，直到子进程释放进程地址空间，返回子进程的 pid 值，该值就是函数调用后父进程的返回值。创建成功后，内核会让子进程首先投入运行，通常情况下子进程将立即调用 exec( )，这样能避免写时复制的额外开销，让父进程执行的话，有可能会向地址空间写入信息。

(1) fork( )函数

调用 fork( )时，clone_flag = SIGCHLD，此时 fork( )将父进程所有的资源通过数据结构的复制“传”给子进程。当调用 fork( )时，系统将为一个新进程准备进程组成的 4 个要素。首先，让新进程与旧进程使用同一个代码段，因为它们的程序相同。对于数据段和堆栈段，系统则复制一份给新进程。这样，父进程的所有数据都可以留给子进程。但是，一旦子进程开始运行，虽然它继承父进程的所有数据，但实际上两个进程的地址空间已经分开，不再共享任何数据，因此相互之间不再有影响。由此可见，用 fork( )创建进程时，子进程只是完全复制父进程的资源，子进程的执行独立于父进程，因而具有良好的并发性。但两进程之间的数据共享需要通过专门的通信机制，如 pipe、fifo、System V 进程间通信机制来实现。以下代码(forktest.c)给出使用 fork( )的基本框架。

```
/***** forktest.c *****/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

  void main( )
  {
     int i;
     int p_id;
     if ( (p_id = fork( )) = = 0 ) {
        /*子进程程序*/
        for ( i = 1; i < 3; i ++ )
              printf("This is child process\n");
     } else if (p_id = = –1){
              printf(“fork new process error!\n”);
              exit(–1);
     } else {
```

```
        /*父进程程序*/
        for ( i = 1; i < 3; i ++ )
            printf("This is parent process\n");
    }
}
```

程序运行结果为：

```
This is child process
This is child process
This is parent process
This is parent process
```

fork( )创建子进程需要将父进程的每种资源都复制一个副本，系统开销很大，但是这些开销并不是所有的情况下都是必需的。比如某进程调用 fork( )函数创建一个子进程后，子进程接下来调用 exec( )执行另一个可执行文件，那么在调用 fork( )过程中对于虚存空间的复制将是多余的。也许读者会问，如果一个大程序在运行，其数据段和堆栈都很大，用 fork( )一次就要复制一次，那么 fork( )函数的系统开销不是很大吗？事实上，Linux 中是采取 copy-on-write 技术，无论是数据段还是堆栈段都是由许多“页”构成的。fork( )函数复制这两个段，只是“逻辑”上的，并非“物理”上的。也就是说，实际执行 fork( )函数时，物理空间上两个进程的数据段和堆栈段都还是共享着的，当有一个进程写了某个数据时，这时两个进程之间的数据才有了区别，系统就将有区别的“页”从物理上分离。因此，系统在空间上的开销就可以达到最小。

当调用 fork( )执行成功时，调用一次却返回两次，该调用向父进程返回子进程的 ID，向子进程返回 0。父进程工作时执行父进程代码，子进程工作时执行子进程代码。调用失败时，向父进程返回–1，没有子进程创建。下面是发生错误时，可能设置的错误代码 errno。

① EAGAIN：fork( )不能得到足够的主存来复制父进程页表。原因可能是用户为超级用户但进程表满，或者用户不是超级用户，但达到单个用户能执行的最大进程数。

② ENOMEM：对创建新进程来说没有足够的空间，该错误是指没有足够的空间分配给必要的内核数据结构。

(2) vfork( )函数

vfork( )函数不同于 fork( )函数，vfork( )函数创建的子进程共享地址空间，即子进程完全运行在父进程的地址空间上，子进程对虚拟地址空间任何数据的修改同样为父进程所见。但是用 vfork( )函数创建子进程后，父进程会被阻塞，直到子进程执行 exec( )或 exit( )。这样处理的优点是：当创建子进程的目的仅仅是为了调用 exec( )执行另一个程序时，子进程不会对父进程的地址空间有任何引用。因此，此时对地址空间的复制是多余的，通过 vfork( )可以减少不必要的开销。在 vfork( )函数的实现中，clone_flags = CLONE_VFORK | CLONE_VM | SIGCHLD，表示子进程和父进程共享地址空间。与此同时，do_fork( )会检查 CLONE_VFORK，如果该位被置 1，子进程会把父进程的地址空间锁住，直到子进程退出或执行 exec( )时才释放该锁。

以下代码(vforktest.c)展示 fork( )与 vfork( )的区别：

```
/***** vforktest.c *****/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{
   int data = 0 ;
   pid_t pid ;
   int choose = 0 ;
   while((choose = getchar( ))!='q') {
    switch(choose) {
    case '1':
      pid = fork( );
      if(pid < 0 ) {
        printf("Error !\n");
      }
      if(pid == 0 ) {
        data++;
      exit(0);
      }
      wait(pid);
      if(pid > 0 ) {
         printf("data is %d\n",data);
      }
      break;
  case '2' :
    pid = vfork( );
    if(pid < 0 ){
      perror("Error !\n");
    }
    if(pid == 0 ) {
      data++;
      exit(0);
    }
    wait(pid);
    if(pid > 0 ) {
      printf("data is %d\n",data);
    }
    break;
  default :
```

```
            break;
          }
        }
      }
```

编译运行结果如下：

```
1
data is 0
2
data is 1
```

程序运行结果说明，fork( )函数创建的子进程不改变父进程创建子进程之前定义的变量值，但是，vfork( )函数创建的子进程改变父进程创建子进程之前定义的变量的值。从该程序也可看出，子进程从父进程继承控制终端、信号标志位、可访问的主存区、环境变量和其他资源分配。但是，它们有不相同的进程堆栈区，正是在这个区域，父子进程完成各自的功能。

(3) clone( )函数

clone( )是创建 Linux 线程的函数，其中参数 fn 是线程所执行的程序，stack 是线程所使用的内核栈，flags 由用户指定，其值可以是前面提到的 CLONE_VM、CLONE_FS、CLONE_FILES、CLONE_SIGHAND 和 CLONE_PID 的组合。

以下给出使用 clone( )的示例代码：

```
void * func(int arg)
{
    …
}

int main( )
{
int clone_flag, arg;
    …
clone_flag = CLONE_VM | CLONE_SIGHAND | CLONE_FS | CLONE_FILES;
stack = (char *)malloc(STACK_FRAME);
stack += STACK_FRAME;
retval = clone((void *)func, stack, clone_flag, arg);
    …
}
```

2. 进程运行新程序

进程控制的另一个主要内容就是对其他程序的引用，在 Linux 中该功能通过使用 exec( )类函数实现。进程一旦调用 exec( )，将一个可执行程序文件读入，代替发出调用进程的原先程序执行。此时，系统把代码段替换成新程序的代码，废弃原有的数据段和堆栈段，并为新程序分

配新的数据段与堆栈段，唯一留下的就是进程号。换言之，对系统而言，还是同一个进程，只不过运行另一个可执行程序。所以，fork( )/exec( )组合是典型的 Linux 新进程产生模式，通常先用 fork( )创建新进程，然后新进程通过调用 exec( )类执行自己的任务。

exec( )类函数不止一个，但功能大致相同。在 Linux 中，它们分别是 execl( )、execlp( )、execle( )、execv( )、execve( )和 execvp( )，函数原型声明格式如下：

```
#include<unistd.h>
int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int execle( const char *path, const char *arg , ..., char* const envp[ ]);
int execv( const char *path, char *const argv[ ]);
int execve( const char *filename, char *const argv [ ], char *const envp[ ]);
int execvp( const char *file, char *const argv[ ]);
```

exec( )函数类装入并运行程序 pathname，并将参数 arg0(arg1,arg2,argv[],envp[])传递给程序，出错返回–1。

在 exec( )函数类中，后缀 l、v、p、e 添加到 exec( )后，表示将具有某种操作能力：

① 后缀为 p 时，可以利用 DOS 的 PATH 变量查找可执行文件。例如，PATH 变量中包含路径/bin，那么可以直接通过输入“cat”执行/bin/cat。

② 后缀为 l 时，希望接收以逗号分隔的参数列表，列表以 NULL 指针作为结束标志。例如：

```
execl( "/bin/cat","/etc/passed","/etc/group",NULL);
```

③ 后缀为 v 时，希望接收到以 NULL 结尾的字符串数组的指针。如：

```
char* argv[] = {"/bin/cat","/etc/passed","/etc/group",NULL}
execv( "/bin/cat", argv );
```

④ 后缀为 e 时，传递指定参数 envp，允许改变子进程的环境。无后缀 e 时，子进程使用当前程序的环境。envp 也是一个以 NULL 结尾的字符串数组指针。

exec( )函数类对应的底层内核函数是 do_execve( )，它的功能是加载新程序并跳转执行。该内核函数首先调用 open_exec( )，把要装入的可执行文件的路径名转换成该执行文件所在的索引节点；随后设置结构变量 bprm，以保存装入时需要的参数和环境变量；接着调用 pre_pare_binprm( )，进行安全性检查，验证调用者有否执行该文件的权限；再把可执行文件的头信息读入主存；最后调用 search_binary_handler( )，寻找与新程序相对应的二进制处理程序，以便加载和执行该程序。可执行文件有很多种(如 ELF(Executable and Linkable Format)、脚本文件 script 等)，它们都是二进制可执行文件，但存储格式各不相同，Linux 内核使用不同二进制处理程序来加载不同二进制格式的可执行文件。

假如一个程序想启动另一程序，并仍希望自身能继续运行，此时如何解决呢？以下代码(execlptest.c)结合 fork( )/execlp( )，给出实现该功能的示例程序：

```
/***** execlptest.c *****/
#include <errno.h>
```

```
#include <error.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
char command[256];

void main( )
{
  int rtn;                                  /*子进程的返回数值*/
  while(1) {                                /*从终端读取要执行的命令*/
   printf( ">" );
   fgets( command, 256, stdin );
   command[strlen(command)-1] = 0;
   if ( fork( ) == 0 ) {                    /*子进程执行此命令*/
     execlp( command, command );
                  /*如果 exec( )返回，表明没有正常执行命令，打印错误信息*/
     perror( command );
     exit( -1);
   } else {        /*父进程， 等待子进程结束，并打印子进程的返回值*/
        wait ( &rtn );
        printf( " child process return %d\n", rtn );
      }
    }
  }
```

运行结果：

```
>exit
child process return –1078552168
>ls
child process return –1078552168
```

3. 进程等待

与 vfork( )不同，fork( )创建的子进程与父进程的执行顺序是无法控制的。如果想控制，必须由父进程使用函数 wait( )或 waitpid( )。其中 wait( )返回任一终止状态的子进程，而 waitpid( )等待特定的子进程。实质上，waitpid( )提供了一个非阻塞版本的 wait( )，这两个函数原型定义如下：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

当进程终止时，会向其父进程发送 SIGCHLD 信号，这个异步事件可以在父进程运行的任何时候发生，包括正常和异常终止两种。因此，调用 wait( )和 waitpid( )的进程可能会有以下 3 种情况。

- 阻塞(如果其所有的子进程都还在运行)。
- 带子进程的终止状态正常返回(其中一个子进程终止)。
- 出错返回(没有子进程)。

(1) wait( )函数

进程通过 wait( )与其子进程同步。当子进程结束时，将发送 SIGCHLD 信号到父进程，子进程进入僵死状态，等待父进程调用 wait( )取出子进程的进程状态。进程一旦调用 wait( )，将立即阻塞自己，由 wait( )自动分析当前进程是否有某个子进程已经退出。如果找到一个已经僵死的子进程，wait( )将收集这个子进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait( )将一直阻塞，直到有一个僵死子进程出现为止。参数 status 用来保存被收集进程退出时的状态，它是一个指向 int 类型的指针。但如果不关心子进程是如何死掉的，而只想把该僵死进程销毁(事实上绝大多数是这种情况)，则可以设置参数为 NULL，其调用形式如下：

```
pid = wait(NULL);
```

如果成功，wait( )会返回被收集的子进程的进程 ID；如果调用进程没有子进程，则会失败，此时 wait( )返回–1，同时 errno 被置为 ECHILD。

以下代码(waittest1.c)给出 wait( )调用示例：

```
/***** waittest1.c *****/
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
main( )
{
    pid_t pc,pr;
    pc=fork( );
    if(pc<0)                              /*如果出错*/
        printf("error ocurred!\n");
    else if(pc==0){                       /*如果是子进程*/
        printf("This is child process with pid of %d\n",getpid( ));
        sleep(10);                        /*睡眠 10 秒钟*/
    }else{                                /*如果是父进程*/
        pr=wait(NULL);                    /*在这里等待*/
        printf("I catched a child process with pid of %d\n"),pr);
    }
    exit(0);
```

```
}
```

编译并运行：

```
$ cc waittest1.c-o waittest1
$ ./waittest1
This is child process with pid of 1508
I catched a child process with pid of 1508
```

从上述代码可以看到，在第 2 行结果打印出来前有 10s 等待时间，即设置的让子进程睡眠的时间。只有子进程从睡眠中苏醒过来，它才能正常退出，也才能被父进程捕捉到。事实上，不管设置子进程睡眠时间有多长，父进程都会一直等待下去。

如果参数 status 值不是 NULL，wait( )就会把子进程退出时的状态取出并存入其中，指出子进程是否正常退出，一个进程也可以被其他进程用信号结束。

由于这些信息被存放在一个整数的不同二进制位中，所以用常规的方法读取会非常麻烦，Linux 系统提供以下 6 个宏用来检查子进程的返回状态。其中前 3 个用来判断退出的原因，后 3 个是对应不同原因的返回状态值。

① WIFEXITED(status)：如果进程通过_exit( )或调用 exit( )正常退出，该宏的值为真。此时可执行 WEXITSTATUS(status)获取子进程传送给 exit( )或_exit( )参数的低 8 位。

② WIFSIGNALED(status)：如果子进程由于得到的信号(signal)没有被捕捉而导致退出，该宏的值为真。此时可执行 WTERMSIG(status)取出使子进程终止的信号编号。

③ WIFSTOPPED(status)：如果子进程没有终止，但停止并可以重新执行时，该宏返回真。这种情况仅出现在 waitpid( )调用中使用了 WUNTRACED 选项，此时可执行 WSTOPSIG(status)取出使子进程暂停的信号编号。

④ WEXITSTATUS(status)：如果 WIFEXITED(status)返回真，该宏返回由子进程调用_exit(status)或 exit(status)时设置的调用参数 status 值。

⑤ WTERMSIG(status)：如果 WIFSIGNALED(status)返回为真，该宏返回导致子进程退出的信号(signal)的值。

⑥ WSTOPSIG(status)：如果 WIFSTOPPED(status)返回真，该宏返回导致子进程停止的信号(signal)值。该调用返回退出的子进程的 ID；或者发生错误时返回–1；或者设置了 WNOHANG 选项，没有子进程退出就返回 0；发生错误时，可能设置的错误代码如下：

- ECHILD　该调用指定的子进程 pid 不存在，或者不是发出调用进程的子进程。
- EINVAL　参数 options 无效。
- ERESTARTSYS　WNOHANG 没有设置且捕获到 SIGCHLD 或其他未屏蔽信号。

请注意，这里的参数 status 并不同于 wait( )里的参数——指向整数的指针 status，而是那个指针所指向的整数值。以下代码(waittest2.c)给出上述宏调用的一个示例：

```
/***** waittest2.c *****/
#include <sys/types.h>
#include <sys/wait.h>
```

```
#include <unistd.h>
main( )
{
    int status;
    pid_t pc,pr;
    pc=fork( );
    if(pc<0)                          /*如果出错*/
        printf("error ocurred!\n");
    else if(pc==0){                   /*子进程*/
        printf("This is child process with pid of %d.\n",getpid( ));
        exit(3);                      /*子进程返回 3 */
    }
    else{                             /*父进程*/
        pr=wait(&status);
        if(WIFEXITED(status)){        /*如果 WIFEXITED 返回非零*/
            printf("the child process %d exit normally.\n",pr);
            printf("the return code is %d.\n",WEXITSTATUS(status));
        }else                         /*如果 WIFEXITED 返回零*/
            printf("the child process %d exit abnormally.\n",pr);
    }
```

编译并运行：

```
$ cc waittest2.c -o waittest2
$ ./waittest2
This is child process with pid of 1538.
the child process 1538 exit normally.
the return code is 3.
```

从运行结果可以看出，父进程准确捕捉到子进程的返回值 3，并把它打印出来。

(2) waitpid( )函数

从本质上讲，waitpid( )函数和 wait( )函数的作用是完全相同的，但 waitpid( )函数多出两个可由用户控制的参数 pid 和 options，从而为编程提供另一种更灵活的方式。

① pid：对于 waitpid( )函数，如果指定的进程或进程组不存在，或者调用进程没有子进程都会出错。从参数的名字 pid 和类型 pid_t 中就可以看出，这里需要的是一个进程 ID。但当 pid 取不同的值时，在这里有不同的意义。

- pid>0 时，只等待进程 ID 等于 pid 的子进程，不管已经有多少子进程运行结束退出了，只要指定的子进程还没有结束，waitpid( )函数就会一直等下去。
- pid= –1 时，等待任何一个子进程退出，没有任何限制，此时 waitpid( )函数和 wait( )函数的作用相同。

• pid=0 时，等待同一个进程组中的任何子进程，如果子进程已经加入了其他进程组，waitpid( )函数不会对它做任何理睬。

• pid< –1 时，等待一个指定进程组中的任何子进程，这个进程组的 ID 等于 pid 的绝对值。

② options：该参数可以让用户进一步控制 waitpid( )函数的操作，参数值可为 0，也可以为 WNOHANG 和 WUNTRACED 两个选项。

• WNOHANG：若由 pid 指定的子进程并不立即可用，则 waitpid( )函数不阻塞，此时其返回值为 0。

• WUNTRACED：若某版本实现支持作业控制，则由 pid 指定的任一子进程状态已暂停，且其状态自暂停以来还未报告过，则返回其状态 status。WIFSTOPPED 宏确定返回值是否对应于一个暂停子进程。

可以用“|”运算符把它们连接起来使用，如下例：

```
ret=waitpid(–1, NULL, WNOHANG | WUNTRACED);
```

如果使用 WNOHANG 参数调用 waitpid( )函数，即使没有子进程退出，它也会立即返回，不会像 wait( )函数那样永远等下去。实际上，wait( )函数是经过包装的 waitpid( )函数。从 /include/unistd.h 文件中可以发现有关 wait( )函数的程序代码段：

```
static inline pid_t wait(int * wait_stat)
{
    return waitpid(–1, wait_stat, 0);
}
```

waitpid( )函数的返回值比 wait( )函数稍微复杂一些，共有 3 种情况：

• 正常返回时，waitpid( )函数返回收集到的子进程的进程 ID。

• 如果设置选项 WNOHANG，而调用中 waitpid( )函数发现没有已退出的子进程可收集，则返回 0。

• 如果调用中出错，则返回–1，这时 errno 会被设置成相应的值以指示错误所在。

当 pid 所指示的子进程不存在，或此进程存在，但不是调用进程的子进程，waitpid( )就会出错返回，这时 errno 被设置为 ECHILD。

以下代码(waitpidtest.c)给出 waitpid( )函数调用的示例：

```
/***** waitpidtest.c *****/
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
main( )
{
    pid_t pc, pr;
    pc=fork( );
```

```
if(pc<0)                    /*如果 fork( )函数出错*/
    printf("Error occured on forking.\n");
else if(pc==0){             /*如果是子进程*/
    sleep(10);              /*睡眠 10 秒*/
    exit(0);
}
/*如果是父进程*/
do{
    pr=waitpid(pc, NULL, WNOHANG);   /*使用 WNOHANG 参数，waitpid 不会在这里
                                       等待*/
    if(pr==0){                       /*如果没有收集到子进程*/
    printf("No child exited\n");
    sleep(1);
    }
}while(pr==0);                       /*没有收集到子进程，就继续尝试*/
if(pr==pc)
    printf("successfully get child %d\n", pr);
else
    printf("some error occured\n");
}
```

编译并运行：

```
$ cc waitpidtest.c -o waitpidtest
$ ./waitpid
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
successfully get child 1526
```

该程序让父进程和子进程分别睡眠 10 s 和 1 s，代表它们分别做了 10 s 和 1 s 的工作。由于父子进程都有工作要做，父进程利用工作的简短间歇查看子进程是否退出，如退出就收集它。

waitpid( )函数和 wait( )函数的主要区别是，waitpid( )函数提供 wait( )函数不能实现的 3 个功能：

- waitpid( )函数等待特定的子进程，而 wait( )函数则返回任一终止状态的子进程。
- waitpid( )函数提供了一个 wait( )函数的非阻塞版本。
- waitpid( )函数支持作业控制(使用 WUNTRACED 选项)。

4. 进程终止

在 Linux 系统中，进程有生命周期，由 fork( )函数而创建，通过 schedule( )函数而执行，由 wait( )函数而等待。可通过给进程发送信号强行终止运行进程，也可当完成任务后进程执行函数自动退出而消亡。使进程自动退出系统的函数是 exit ( )或_exit( )。exit( )函数的原型是：

```
#include<stdlib.h>
void exit(int status);
```

而_exit( )的函数原型是：

```
#include<unistd.h>
void _exit(int status);
```

两个函数均带有一个整数类型的参数 status，用来传递进程结束时的状态(如进程正常结束或因出现某种意外而结束)。一般来说，0 表示正常结束，否则表示出现错误，进程非正常结束。在实际编程时，可以用 wait( )接收子进程的返回值，从而针对不同情况做不同处理。

作为函数，_exit( )和 exit( )是一对“孪生兄弟”，它们都可以正常结束进程的执行。而进程终止最终都要调用内核函数 do_exit( )，它的实现代码在文件 exit.c 中。其执行流程大致如下：

① 设置标志，表明进程正在被销毁。

② 如果进程在定时器队列或信号量队列中等待，则将其移出。

③ 调用_exit_mm( )、_exit_files( )、_exit_fs( )、exit_sem( )、exit_sighand( )、exit_thread( )释放进程所占用的各种资源。其中_exit_mm( )的作用是释放进程地址空间，执行过程中会检测该进程是否由 vfork( )创建，如果是则唤醒父进程。

④ 设置进程的退出状态，调用 exit_notify( )处理该进程与其父进程和子进程的各种关系。在该函数中，会将该进程状态置为 TASK_ZOMBIE，使其成为僵死进程。

⑤ 调用 schedule( )函数切换到其他进程。

do_exit( )执行后，进程处于 TASK_ZOMBIE 状态，只留下 task_struct 结构。之所以不立即释放该结构是因为其中存放着 pid，父进程往往要通过 wait( )来检查子进程是否结束，而调用之前显然不能让子进程消失。wait( )会回收处于 TASK_ZOMBIE 状态的子进程的 task_struct 结构及 pid 号，此时子进程才会退出僵死状态，并从系统中消失。对于父进程先于子进程结束的情况，这时 init 进程就变成子进程的父进程，从而保证 task_struct 结构总能得到回收。

图 2-6 进一步描述两个函数结束进程的处理逻辑。从图中可以看出，_exit( )的作用最为简单，直接使进程停止运行，做清除和销毁工作；而在调用 exit( )之前要检查文件的打开情况，把文件缓冲区中的内容写回文件，即图中“清理 I/O 缓冲”一项。这也是 exit( )与_exit( )的最大区别。

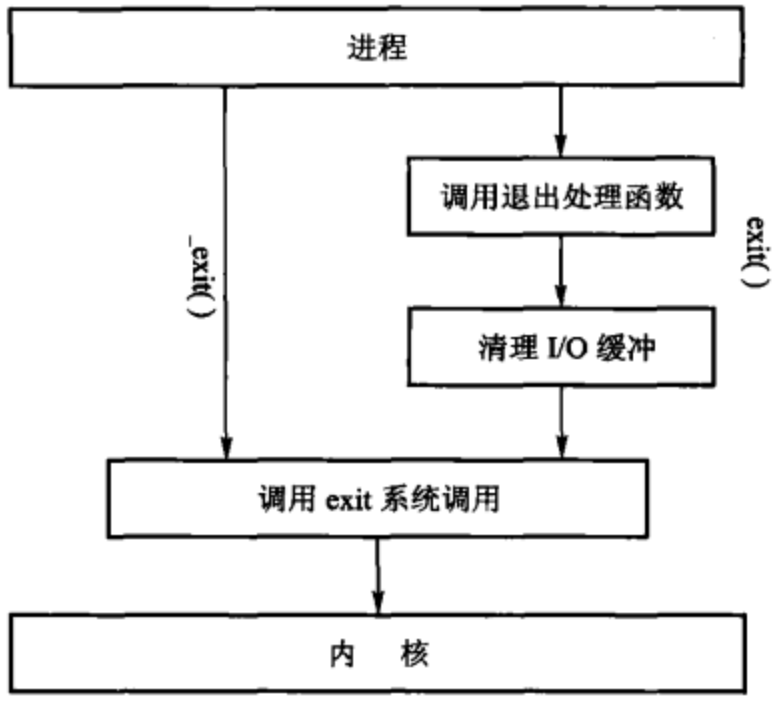


图 2-6 exit( )与_exit( )的区别

在 Linux 标准函数库中，有一套“高级 I/O”函数，C 语言中常用的 printf( )、fopen( )、fread( )、fwrite( )都在此列，它们也被称作缓冲 I/O(buffered I/O)。其特征是对应每个打开的文件，在主存中都有一个缓冲区。每次读文件时，会多读出若干条记录，这样下次读文件时就可以直接从主存的缓冲区中读取。每次写文件的时候，也仅仅是写入主存中的缓冲区，等满足一定的条件(达到一定数量，或遇到特定字符，如换行符\n 和文件结束符 EOF)，再将缓冲区中的内容一次性写入文件。这样就能大大加快文件读写的速度，但也为编程带来一些麻烦。如果有一些数据，误认为已经写入了文件，实际上因为没有满足特定的条件，它们还只是保存在缓冲区内。此时，用_exit( )直接将进程关闭，缓冲区中的数据就会丢失；反之，如果想保证数据的完整性，就一定要使用 exit( )。试比较以下两段代码的运行结果。

下列代码(exittest1.c)调用 exit(0)结束进程。

```
/***** exittest1.c *****/
#include<stdlib.h>
main( )
{
    printf("output begin\n");
    printf("content in buffer");
    exit(0);
}
```

编译并运行：

```
$gcc exittest.c –o exittest
$./exittest
output begin
```

```
content in buffer
```

下列代码(exittest2.c)调用_exit(0)结束进程。

```
/*****exittest2.c*****/
#include<unistd.h>
main( )
{
      printf("output begin\n");
      printf("content in buffer");
    _exit(0);
  }
```

编译并运行：

```
$gcc _exittest.c –o _exittest
$./_exittest
output begin
```

2.2.3 多线程编程

2.2.3.1 线程类型

线程在进程的基础上作进一步抽象，也就是说一个进程分为两部分：线程集合和资源集合。线程是进程中的动态对象，它是一个独立的控制流，进程中的所有线程将共享进程拥有的资源。但是线程应该有自己的私有对象，如程序计数器、堆栈和寄存器上下文。在 Linux 中，可把线程分为内核线程、内核支持的用户线程和线程库支持的用户线程等 3 种类型。

1. 内核线程

内核线程的创建和撤销由内核的内部需求来决定，负责实现一个指定系统功能。内核线程没有用户地址空间，它共享内核的正文段和内核全局数据，具有自己的内核栈。内核线程被单独调度并使用标准的内核同步机制，可以被单独分配到一个 CPU 上运行。内核线程的调度由于不需要经过 CPU 状态的转换并进行地址空间的重新映射，因此在内核线程间进行上下文切换比在用户进程间进行上下文切换快得多。

2. 内核支持的用户线程

Linux 内核支持的用户线程都有一个用来维护它的 task_struct 结构，该用户线程实质上是一个具备单独 task_struct 结构的子进程，都有自己的程序计数器、寄存器集合、内核栈和用户栈。但同属于一个父进程的这组子进程(内核支持的用户线程)共享父进程的所有资源，包括共享地址空间和其他资源。内核支持的用户线程实质上是特殊的进程，能被单独调度和运行。

3. 线程库支持的用户线程

用户线程是通过线程库实现的，内核不参与调度。它们可以在没有内核参与下创建、释放和管理，线程库提供同步和调度方法，这样进程可以使用大量线程而不消耗内核资源，节省系

统开销。每个用户线程都可以有自己的用户栈，即用来保存用户级寄存器上下文以及信号屏蔽等状态信息的主存区。线程库通过保存当前线程的堆栈和寄存器内容，并加载新调度线程的相应内容来实现用户线程之间的调度和上下文切换。内核仍然负责进程的切换，因为只有内核具有修改主存管理寄存器的权力。线程库支持的用户线程不是真正的调度实体，内核对它们一无所知，而只是调度用户线程所属的进程，这些进程再通过线程库函数来调度进程内的用户线程。当一个进程被抢占时，其所有用户线程都被抢占；当一个用户线程被阻塞时，它会阻塞对应的用户进程。由于 Linux 已提供内核支持的用户线程，所以 Linux 下的应用程序通常没有必要再使用线程库支持的用户线程。

在许多实现多线程的操作系统中(如 Windows 和 Digital UNIX 等)，线程和进程通过进程表项和线程表项两种数据结构来抽象表示。一个进程表项可以指向若干个线程表项，调度程序在进程的时间片内再调度线程。但是在 Linux 中没有做这种区分，而是统一使用 task_struct 来管理所有进程/线程。只是线程与线程之间的资源是共享的，这些资源可以是虚存、文件和信号等。也就是说，Linux 的每个线程都有一个 task_struct，所以线程和进程可以使用同一调度程序来调度。图 2-7 给出了 Linux 系统与其他操作系统中线程概念的对比。其中，P 表示进程，R 表示资源，T 表示线程。

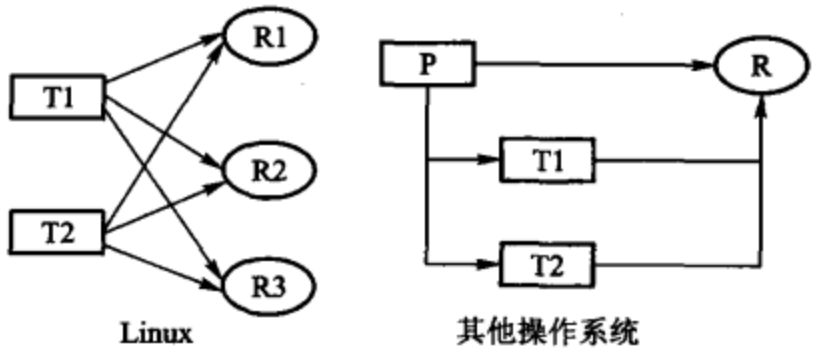


图 2-7 Linux 系统与其他操作系统中的线程对比

2.2.3.2 线程控制

1. POSIX 线程

POSIX 线程包提供一组函数，用来创建、删除和同步同一进程内的线程。为了使用这些函数，必须对_REENTRANT 宏进行定义，该宏包括在头文件 pthread.h 中。编译程序时，需要用选项-lpthread 来链接线程库。POSIX 线程可在用户空间内实现，也可由内核提供支持，然后通过 pthread API 供应用程序使用。

2. 线程的创建

在 Linux 系统中，可通过函数 clone( )及 pthread_create( )创建线程。这两个函数的原型定义如下：

```
int clone(int (*fn)(void * arg), void *stack, int flags, void * arg);
#include <pthread.h>
```

```
int pthread_create(pthread_t *restrict tidp, const pthread_attr_t *restrict attr,
    void *(*start_rtn)(void), void *restrict arg);
```

pthread_create( )在调用线程的地址空间内创建一个新线程来运行。参数 tidp 为指向线程标识符的指针，可以使用 attr 参数来为新线程定义不同属性(如栈尺寸)。使用默认属性时，将 NULL 赋给 attr 参数。第 3 个和第 4 个参数指定执行的函数 start_rtn 和传递给函数的参数 arg。函数返回 0 表示成功，非 0 表示失败。

pthread_create( )与 clone( )的调用形式有些相似，两者最根本的差别在于 clone( )是创建一个内核支持的用户线程，对内核是可见的且由内核调度。而 pthread_create( )通常只是创建一个线程库支持的用户线程，对内核是不可见的，由线程库调度。在 Linux 线程中，专门为每一个进程构造了一个管理线程，负责处理线程相关的管理工作。当进程第 1 次调用 pthread_create( )创建一个线程的时候就会创建并启动管理线程。

其中形式参数说明如下：

- pthread_t *restrict tidp：要创建的线程的线程 id 指针。
- const pthread_attr_t *restrict attr：创建线程时的线程属性。
- void* (start_rtn)(void)：返回值是 void 类型的指针函数。
- void *restrict arg：start_rtn 的行参。

以下代码(threadcreatetest.c)给出创建线程的示例。

```
/***** threadcreatetest.c *****/
#include <pthread.h>
#include <stdio.h>
void *create(void *arg)
{
   printf("new thread created ..... ");
}

int main(int argc,char *argv[ ])
{
   pthread_t tidp;
   int error;
   error=pthread_create(&tidp,NULL,create,NULL);
   if(error!=0) {
       printf("pthread_create is not created ... ");
       return -1;
   }
   printf("prthread_create is created... ");
   return 0;
}
```

使用下述命令编译该程序：

```
#gcc -Wall-lpthread threadcreatetest.c
```

因为 pthread 库不是 Linux 系统的库，所以在进行编译的时候要加上-lpthread，否则编译通不过。

3. 线程等待

父线程可以使用 pthread_join( )等候子线程终止，其函数原型定义如下：

```
#include<pthread.h>
int pthread_join(pthread_t thread，void **status);
```

其中参数 thread 指定将要等待的线程，线程通过 pthread_create( )返回的标识符来指定；参数 status 是一个指向另一个指针的指针，用于指定返回值；如果线程使用 pthread_exit(void*)返回一个值，它会经由 status 参数返回到父进程，线程的资源(如线程描述表)会直到父线程调用 pthread_join( )后才释放。函数返回 0 表示成功，非 0 表示失败。

4. 线程同步

pthread 线程库中有许多种同步原语，包括信号量、互斥锁、条件变量和读/写锁。

线程信号量同步原语的函数名都以 sem_开头，而不以 pthread_开头。线程中使用的基本信号量函数有 4 个，其原型定义如下：

```
#include <semaphore.h>
int sem_init(sem_t*sem,int pshared,unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_destroy(sem_t *sem);
```

sem_init( )初始化由 sem 指向的信号量对象，设置它的共享选项，并置初始化整数值。参数 pshared 控制信号量的类型，如果其值为 0，就表示该信号量是当前进程的局部信号量，否则，这个信号量就可以在多个进程之间共享。

sem_wait( )和 sem_post( )都以一个指针为参数，该指针指向的对象是由 sem_init( )初始化的信号量。sem_post( )的作用是以原子操作的方式给信号量的值加 l，sem_wait( )以原子操作的方式将信号量的值减 l。

sem_destroy( )用于清理信号量，若清理的信号量正被线程等待，就会收到错误消息。

线程互斥量同步原语，主要解决临界区的使用，线程可以通过相应函数来创建、销毁和操作互斥量。它们的函数原型定义如下：

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread__mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

这些函数中的 mutex 参数是指向 pthread mutex 互斥量的指针，函数返回 0 表示成功，非 0 表示失败。

以下代码(threadsyntest.c)给出线程同步的一种实现。

```
/***** threadsyntest.c *****/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
pthread_mutex_t work_mutex;    /*互斥量，保护工作区及额外变量 time_to_exit*/

#define WORK_SIZE 1024
char work_area[WORK_SIZE];   /*工作区*/
int time_to_exit = 0;

int main( )
{
    int res;
    pthread_t a_thread;
    void *thread_result;
res = pthread_mutex_init(&work_mutex, NULL); /*初始化工作区*/
if (res != 0) {
    perror("Mutex initialization failed");
    exit(EXIT_FAILURE);
}
res = pthread_create(&a_thread, NULL, thread_function, NULL); /*创建新线程*/
if (res != 0) {
    perror("Thread creation failed");
    exit(EXIT_FAILURE);
}
/*主线程*/
pthread_mutex_lock(&work_mutex);
printf("Input some text. Enter 'end' to finish\n");
while(!time_to_exit) {
    fgets(work_area, WORK_SIZE, stdin);
    pthread_mutex_unlock(&work_mutex);
    while(1) {
        pthread_mutex_lock(&work_mutex);
        if (work_area[0] != '\0') {
            pthread_mutex_unlock(&work_mutex);
```

```
                sleep(1);
            }
            else {
                break;
            }
        }
    }
    pthread_mutex_unlock(&work_mutex);
    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined\n");
    pthread_mutex_destroy(&work_mutex);
    exit(EXIT_SUCCESS);
}
/*新线程*/
void *thread_function(void *arg)
{
    sleep(1);
    pthread_mutex_lock(&work_mutex);
    while(strncmp("end", work_area, 3) != 0) {
        printf("You input %d characters\n", strlen(work_area)-1);
        work_area[0] = '\0';
        pthread_mutex_unlock(&work_mutex);
        sleep(1);
        pthread_mutex_lock(&work_mutex);
        while (work_area[0] == '\0' ) {
            pthread_mutex_unlock(&work_mutex);
            sleep(1);
            pthread_mutex_lock(&work_mutex);
        }
    }
    time_to_exit = 1;
    work_area[0] = '\0';
    pthread_mutex_unlock(&work_mutex);
    pthread_exit(0);
}
```

运行结果如下：

```
$./threadsyntest
Input some text. Enter 'end' to finish
Testing
You input 7 characters
A Test Routine
You input 14 characters
end

Waiting for thread to finish...
Thread joined
```

在该程序中，新线程首先试图对互斥量加锁。如果它已经被加锁，这个调用将被阻塞直到锁被释放为止。一旦获得访问权，新线程将检查是否有申请退出程序的请求。如果有，就简单设置 time_to_exit 变量，再把工作区的第 1 个字符设置为'\0'，然后退出。如果不想退出，就对字符个数进行统计。然后把 work_area 数组中的第 1 个字符设置为 NULL。通过将第 1 个字符设置为 NULL 通知读取输入的线程已完成字符统计。随后解锁互斥量并等待主线程继续运行。程序将周期性地尝试给互斥量加锁，如加锁成功，则检查是否主线程又有字符需要统计。若没有，则解锁互斥量继续等待；否则将统计字符个数，并再次进入循环。

主线程代码结构和新线程类似。首先给工作区加锁，向其中读入文本。然后解锁允许其他线程访问它并统计字符数目。程序将周期性地尝试给互斥量加锁，检查字符数目是否已统计完毕(work_area[0]=NULL)。如果还需要等待，则释放互斥量。

5. 线程终止

如果进程中任何一个线程调用函数 exit( )或_exit( )，那么整个进程就会终止。与此类似，如果信号的默认动作是终止进程，那么，把该信号发送到线程会终止进程。

线程的正常退出的方式包括以下几种。

- 线程只是从启动例程中返回，返回值是线程中的退出码。
- 线程可以被另一个线程终止。
- 线程自己调用 pthread_exit( )。

与线程终止相关的两个函数原型定义如下。

```
#include <pthread.h>
void pthread_exit(void *rval_ptr);  /*rval_ptr  线程退出返回的指针*/
int pthread_join(pthread_t thread,void **rval_ptr);
                                        /*成功结束进程为 0，否则为错误编码*/
```

以下代码(threadexittest.c)给出线程正常退出的代码示例：

```
/***** threadexittest.c *****/
#include <stdio.h>
#include <pthread.h>
```

```
#include <unistd.h>
void *create(void *arg)
{
   printf("new thread is created ...\n");
   return (void *)2;
}

int main(int argc,char *argv[])
{
   pthread_t tid;
   int error;
   void *temp;
   error=pthread_create(&tid, NULL, create, NULL);
   if(error!=0)  {
      printf("thread is not created ... ");
      return -1;
   }
   error=pthread_join(tid,&temp);
   if(error!=0){
   printf("thread is not exit ... ");
   return -2;
   }
   printf("thread is exit code %d  \n",(int )temp);   sleep(1);
   printf("thread is created... ");
   return 0;
   }
```

运行结果如下：

```
new thread is created ...
thread is exit code 2
thread is created...
```

6. 线程标识

与进程类似，线程也有线程标识，可通过 pthread_self( )获取，其函数原型是：

```
#include <pthread.h>
pthread_t pthread_self(void);
```

以下代码(threadselftest.c)给出线程正常退出的代码示例：

```
/***** threadselftest.c *****/
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void *create(void *arg)
```

```
{
   printf("new thread ....\n ");
   printf("thread_tid==%u \n ",(unsigned int)pthread_self( ));
   printf("thread pid is %d   \n",getpid( ));
   return (void *)0;
}
int main(int argc,char *argv[])
{
   pthread_t tid;
   int error;
    printf("Main thread is starting ... \n");
   error=pthread_create(&tid,NULL,create,NULL);
   if(error!=0) {
      printf("thread is not created ...\n ");
      return -1;
   }
   printf("main pid is %d   ",getpid( ));
   sleep(1);
   return 0;
}
```

该代码的执行结果：

```
Main thread is starting ...
main pid is 6860
new thread ....
thread_tid==3084954544
thread pid is 6860
```

2.3 实 验 内 容

2.3.1 实验 1 创建进程

2.3.1.1 实验说明

学会通过基本的 Linux 进程控制函数，由父进程创建子进程，并实现协同工作。创建两个进程，让子进程读取一个文件，父进程等待子进程读完文件后继续执行。

2.3.1.2 解决方案

进程协同工作就是要协调好两个或两个以上的进程，使之安排好先后次序并依次执行，可

以用 wait( )或 waitpid( )函数来实现这一点。当只需要等待任一子进程运行结束时，可在父进程中调用 wait( )函数。若需要等待某一特定子进程的运行结果时，需调用 waitpid( )函数，它是非阻塞型函数。

2.3.1.3 程序框架

```
#include <sys/types.h>
#include <sys/wait.h>
main( )
{
      /*创建子进程*/
      if(创建失败)
            /*打印“创建进程失败”提示信息*/
     else if(子进程){   /*子进程*/
            /*打印子进程相关信息*/
            /*退出子进程*/
 }else{     /*父进程*/
            /*等待子进程信息*/
            /*继续父进程的执行*/
 }
}
```

当 fork( )调用成功后，父子进程完成各自的任务，但当父进程的工作告一段落，需要用到子进程的结果时，它就停下来调用 wait( )，一直等到子进程运行结束，然后利用子进程的结果继续执行，这样就圆满地解决了父子进程间的协同工作问题。

2.3.2 实验 2 线程共享进程中的数据

2.3.2.1 实验说明

了解线程与进程之间的数据共享关系。创建一个线程，在线程中更改进程中的数据。

2.3.2.2 解决方案

在进程中定义共享数据，在线程中直接引用并输出该数据。

2.3.2.3 程序框架

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
static int shdata=4;
```

```
void *create(void *arg)
{
    printf("new pthread ... \n");
    printf("shared data = %d   \n", shdata);
    return (void *)0;
}

int main(int argc,char *argv[])
{
    /*用 pthread_create( )创建新线程*/
if(创建失败) {
    /*打印“创建线程失败”提示信息*/
    /*返回-1*/
    }
    /*休眠一段时间*/
    /*打印“创建线程成功”提示信息*/
    /*返回 0*/
}
```

2.3.3 实验 3 多线程实现单词统计工具

2.3.3.1 实验说明

多线程实现单词统计工具。

2.3.3.2 解决方案

区分单词原则：凡是一个非字母或数字的字符跟在字母或数字的后面，那么这个字母或数字就是单词的结尾。

允许线程使用互斥锁来修改临界资源，确保线程间的同步与协作。如果两个线程需要安全地共享一个公共计数器，需要把公共计数器加锁。线程需要访问称为互斥锁的变量，它可以使线程间很好地合作，避免对于资源的访问冲突。

2.3.3.3 程序框架

```
#include <stdio.h>
#include <pthread.h>
#include <ctype.h>

pthread_mutex_t counter_clock = PTHREAD_MUTEX_INITIALIZER;
```

```
int main(int ac,char *av[ ])
{
        void *count_words(void *);
        if(ac !=3) {
                printf("Usage:%s file1 file2\n",av[0]);
                exit(1);
        }
        /*分别以 av[1]和 av[2]作为参数，创建两个线程 t1 和 t2*/
        /*让线程 t1 和 t2 进入等待态*/
        /*输出统计出来的单词总数*/
}

void *count_words(void *f)
{
        char *filename=(char *)f;
        FILE * fp;
        int c,prevc='\0';

        if((fp=fopen(filename,"r"))!=NULL) {
                while((c=getc(fp))!=EOF)
                {
                        if(!isalnum(c)&&isalnum(prevc)) {
                                pthread_mutex_lock(&counter_clock);
                                total_words++;
                                pthread_mutex_unlock(&counter_clock);
                        }
                        prevc=c;
                }
                fclose(fp);
        } else {
                perror(filename);
        }
        return NULL;
}
```

编译：

```
$gcc wc.c -o wc-static-lpthread
```

运行：

```
$./wc file1 file2
```

第3章 传统的进程间通信

3.1 实 验 目 的

- 理解信号和管道的概念及实现进程间通信的原理。
- 掌握信号通信机制，学会通过信号实现进程间通信。
- 掌握匿名管道及命名管道通信机制，学会通过管道实现进程间通信。

3.2 背 景 知 识

3.2.1 进程间通信的方式

进程间通信(InterProcess Communication，IPC)是多用户、多任务操作系统必不可少的基本功能和基础设施。在操作系统中，遇到下列情况之一时，进程之间必须通过通信才能保证正确地工作。

① 共享资源。共享临界资源时，进程之间必须互斥，实质上相关进程之间可通过简单通信方式(如锁)来通知对方。

② 协同工作。进程之间协作完成任务，必须确保进程同步，进程同步是进程间通信的一种。

③ 并发控制。多任务系统中，有些进程之间并不相互独立，如调试程序将控制另一个进程的执行，控制进程和受控进程之间离不开进程通信。

④ 通知进程。当事件发生，一个进程应该向其他进程或进程组发出消息(如子进程终止)时，它必须通知其父进程，自己已是僵死进程。

⑤ 传递数据。协作进程之间通过进程间通信来传递数据，以便协同工作。

因此在设计一个操作系统时，必须设计各种有效的进程间通信工具。Linux 环境下的进程间通信来源于 UNIX 平台上的进程间通信手段和设施，对 UNIX 发展作出重大贡献的两大主力分别是 AT&T 的贝尔实验室和加州大学伯克利分校的伯克利软件发布中心。但两者在进程间通信方面的研究侧重点有所不同。前者对 UNIX 早期的进程间通信手段进行系统的改进和扩充，形成 System V IPC，通信进程局限在单台计算机内。后者则突破该限制，形成基于套接字(socket)的进程间通信机制。Linux 继承了这两种通信机制，如图 3-1 所示。

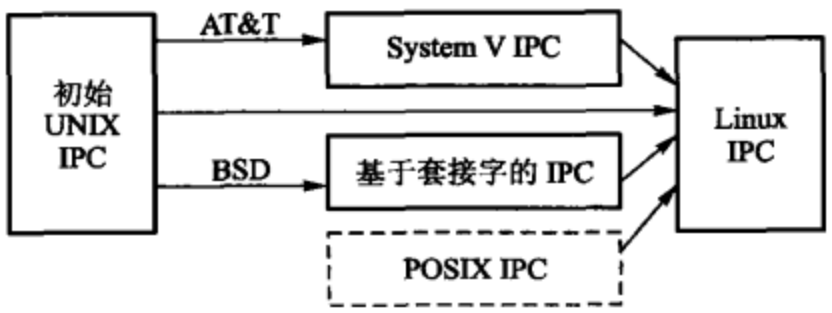


图 3-1 Linux 继承的进程间通信

其中，最初 UNIX IPC 包括信号和管道(匿名管道又称管道，命名管道又称 FIFO)，System V IPC 包括 System V 消息队列、System V 信号量、System V 共享主存区，POSIX IPC 包括 POSIX 消息队列、POSIX 信号量、POSIX 共享主存区。POSIX 是由美国电子电气工程协会(IEEE)开发的一个独立 ANSI UNIX 标准，该标准被称为计算机环境的可移植性操作系统界面。现有大部分 UNIX 都遵循 POSIX 标准，而 Linux 设计从一开始就遵循 POSIX 标准。

Linux 进程间通信主要包括以下几种。

(1) 信号(signal)

信号是一种简洁的通信方式，进程或内核均可使用信号通知一个进程有某种事件发生。除用于进程间通信外，进程也可以发送信号给进程自身(如函数 kill( ), raise( )等)。Linux 除支持 UNIX 早期信号语义函数 signal( )外，还支持符合 POSIX.1 标准的信号函数 sigaction( )。实际上，该函数是基于 UNIX BSD 的，UNIX BSD 为了实现可靠信号机制，又能够统一对外接口，用 sigaction( )函数重新实现了 signal( )函数。

(2) 管道(pipe)及命名管道(named pipe)

管道用于具有亲缘关系的进程之间的通信。父子进程之间或兄弟进程之间，可以通过函数建立单向管道。管道两端的进程将该管道视作一个文件，一个进程向管道中写入数据，另一个进程从管道中读出数据。数据传递遵循“先进先出”规则。由于管道都是基于单向通信的，因此，若需支持双向通信，则需要建立两个匿名管道。

命名管道能克服管道没有名字的限制，以 FIFO 文件的形式出现在文件系统中。因此，任何进程都可以使用其文件名来打开文件，然后进行读写。除具有管道所具有的功能外，它还具有允许无亲缘关系进程之间进行通信，命名管道实际是管道的推广。

(3) 消息(message)队列

消息队列是消息的链接表，在进程之间以传递消息的形式进行通信，包括 POSIX 消息队列和 System V 消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读取队列中的消息。消息队列克服信号承载信息量少、管道只能承载无格式字节流及缓冲区大小受限等缺点。

(4) 共享主存(shared memory)

该方式是针对其他通信机制运行效率较低而设计的，使得多个进程可以访问同一块主存空

间。共享主存方式通常与其他通信机制(如信号量)结合使用，以解决进程通信中的同步与互斥问题。

(5) 信号量(semaphore)

信号量主要用作用户空间中进程之间及同一进程内不同线程之间的同步手段，是内核信号量机制的一种推广。

(6) 套接字(socket)

套接字是更为一般的、统一的进程间通信机制。它既可用于同一台计算机上的进程间通信，也可为不同计算机上的进程建立基于网络的通信。如果说命名管道把原来只适用于亲缘关系进程间通信的“管道”推广到同一台计算机中的任意进程之间，那么套接字又将进程间通信进一步推广到网络环境中的任意进程之间。因此，套接字成为目前最流行和广泛应用的进程间通信手段。它最初从 UNIX BSD 分支中开发出来，但现在已被移植到其他 UNIX 系统上，Linux 和 System V 的变种都支持套接字。

本章将介绍基于管道和信号的 UNIX IPC 传统进程间通信机制，第 4 章将介绍基于 System V IPC 的进程间通信机制。

3.2.2 信号通信

3.2.2.1 信号的基本概念

信号全称为软中断信号，也称作软中断，它实质上是在软件层次上对中断机制的一种模拟，一个进程收到一个信号与 CPU 收到一个中断请求可以说是一样的。信号是进程间通信机制中唯一的异步通信机制，用来通知进程有异步事件发生。进程之间可以通过函数，如 kill( )和 alarm( )等传递软中断信号。内核也可在发生内部事件时向进程发送信号，通知进程发生了某个事件。需要注意，信号只是用来通知某进程发生了什么事件，并不给该进程传递任何数据。信号事件的发生有以下两种来源。

- 硬件来源：如用户按 Ctrl+C 键，或其他硬件故障。
- 软件来源：常用的发送信号的函数包括 kill( )、raise( )、alarm( )、setitimer( )及 sigqueue( )函数，软件来源还包括一些非法运算等操作。

进程收到信号后，通常有 3 种不同处理方法。

- 忽略信号：进程忽略接收到的信号，不做任何处理，像未发生过一样。但 SIGKILL 和 SIGSTOP 信号不能忽略。
- 捕获信号：该方法类似中断的处理程序，进程本身可以在系统中为需要处理的信号定义信号处理函数。一旦相应信号发生，执行对应的信号处理函数。
- 默认操作：信号由内核的默认处理程序处理。Linux 为每种信号都定义默认操作，如 SIGINT 的默认处理是进程消亡。进程可通过 signal( )来指定进程对某个信号的处理行为。

在进程表的表项中有一个信号域，该域中每一位对应一个信号。当内核向一个进程发送信号时，信号接收进程在其所在的进程表项的信号域设置对应于该信号的位。需要注意，Linux内核中不存在任何机制用来区分不同信号的优先级。也就是说，当同时接收到多个信号时，进程可能会以任意顺序接收信号并进行处理。

进程通过 task_struct 中的 sigpending 维护本进程中的未决信号，包括当前挂起的信号及当前阻塞的信号。该结构定义如下：

```
struct sigpending {
    struct sigqueue *head, **tail;
    sigset_t signal;
};
```

该结构的成员 signal 是进程中所有未决信号集，head 和 tail 分别指向一个 sigqueue 类型的结构链(称“未决信号信息链”)的首尾和尾部。信息链中的每个 sigqueue 结构刻画一个特定信号所携带的信息，并指向下一个 sigqueue 结构。该结构定义如下：

```
struct sigqueue {
    struct sigqueue *next;
    siginfo_t info;
}
```

未决信号集中的信号可以分成两类：挂起信号和阻塞信号。其中挂起信号指尚未进行处理的信号，阻塞信号指当前不处理的信号。如果产生某个当前被阻塞的信号，则该信号会一直保持挂起，直到该信号不再被阻塞为止。除 SIGKILL 和 SIGSTOP 信号外，所有信号均可以被阻塞，信号的阻塞通过函数实现。

此外，每个 task_struct 结构还包含一个指向 sigaction 结构数组的指针，该结构数组中的信息实际指定进程处理所有信号的方式。如果某个 sigaction 结构中包含处理信号的例程地址，则由该处理例程处理该信号；否则，则根据结构中的一个标志或者由内核做默认处理，或忽略该信号。通过函数，进程可以修改 sigaction 结构数组的信息，从而指定进程处理信号的方式。

进程不能向系统中所有的进程发送信号。一般而言，除系统和超级用户外，普通进程只能向具有相同 uid 和 gid 的进程，或处于同一进程组的进程发送信号。产生信号时，内核将信号发送到进程 task_struct 的 signal 成员的相应位，并设置为 1，表明产生该信号。如果信号发送给一个正在睡眠的进程，那么要看该进程进入睡眠的优先级。如果进程睡眠在可被中断的优先级上，则唤醒进程；否则仅设置进程表中信号域相应的位，而不唤醒进程。这一点比较重要，因为进程检查是否收到信号的时机是：一个进程在即将从内核态返回到用户态时，或者在一个进程要进入或离开适当的低调度优先级睡眠状态时。此时，内核检查它的 signal 和 block 字段，如果收到任何一个未被阻塞的信号，内核将根据 sigaction 结构数组中的信息进行处理。

3.2.2.2 信号的种类

可以从两个角度对信号进行分类：从可靠性角度，可分为可靠信号与不可靠信号；从实时

性角度，可分为实时信号与非实时信号。在 Linux 环境下，可通过运行“kill-l”命令获取 Linux 支持的信号列表。

```
$ kill –l
1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
9) SIGKILL      10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX–14
51) SIGRTMAX–13 52) SIGRTMAX–12 53) SIGRTMAX–11 54) SIGRTMAX–10
55) SIGRTMAX–9  56) SIGRTMAX–8  57) SIGRTMAX–7  58) SIGRTMAX–6
59) SIGRTMAX–5  60) SIGRTMAX–4  61) SIGRTMAX–3  62) SIGRTMAX–2
63) SIGRTMAX–1  64) SIGRTMAX
```

Linux 信号机制基本上是从 UNIX 系统中继承来的，早期 UNIX 系统中的信号机制比较简单和原始，在实践中暴露出一些问题，因此，把建立在早期机制上的信号称为“不可靠信号”。信号值小于 SIGRTMIN 的信号都是不可靠信号，其主要问题是：进程每次处理信号后，就将对信号的响应设置为默认动作，在某些情况下，将导致对信号的错误处理。用户如果不希望这样操作，就要在信号处理函数结尾再一次调用 signal( )，重新安装该信号。因此，早期 UNIX 下的不可靠信号主要指的是进程可能对信号作出错误反应，以及信号可能丢失。Linux 支持不可靠信号，但是对不可靠信号机制做了改进，在调用完信号处理函数后，不必重新调用该信号的安装函数(信号安装函数是在可靠机制上的实现)。因此，Linux 下的不可靠信号问题主要指的是信号可能丢失。

随着时间推移，实践证明有必要对原始信号机制加以改进和扩充，力图实现“可靠信号”。由于原来定义的信号已有许多应用，不宜再做改动，最终只好添加一些新信号，并将它们定义为可靠信号，这些信号支持排队，不会丢失。信号值位于 SIGRTMIN 和 SIGRTMAX 之间的信号都是可靠信号。Linux 提供新版本的信号安装函数 sigaction( )和信号发送函数 sigqueue( )。与此同时，Linux 仍支持早期信号安装函数 signal( )和信号发送函数 kill( )。

需注意的是，不要误认为所有由 sigqueue( )发送、sigaction( )安装的信号就是可靠的。事实上，信号可靠与不可靠只与信号值有关，与信号的发送及安装函数无关。目前 Linux 中的 signal( )是通过 sigation( )函数实现的，即使通过 signal( )安装的信号，也不必在信号处理函数的结尾再次调用信号安装函数。此外，由 signal( )安装的实时信号支持排队，同样不会丢失。signal( )与 sigaction( )的最大区别在于：经过 sigaction( )安装的信号都能传递信息给信号处理函数(对所有信号这一点都成立)，而经过 signal( )安装的信号却不能向信号处理函数传递信息，对于信号发送函数来说也是一样。

非实时信号都是不可靠信号，不支持排队；实时信号都是可靠信号，支持排队。

3.2.2.3 信号生命周期

一个完整的信号生命周期从信号发送开始，结束于相应的处理函数执行完毕。整个生命周期分为 3 个阶段，并通过 4 个事件来刻画，相邻两个事件的时间间隔构成信号生命周期的一个阶段。图 3-2 描述信号的生命周期。

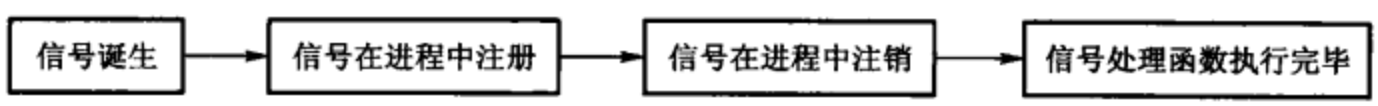


图 3-2 信号的生命周期

1. 信号诞生

信号诞生指的是触发信号的事件发生，如检测到硬件异常、定时器超时，以及调用信号发送函数 kill( )或 sigqueue( )等。

2. 信号在目标进程中注册

将信号值加入到进程的未决信号集中(sigpending 结构的第 2 个成员 sigset_t signal)，并将信号所携带的信息保存到未决信号信息链的某个 sigqueue 结构中。只要信号在进程的未决信号集中，表明进程已经知道这些信号的存在，但还没来得及处理，或者该信号被进程阻塞。

需要注意，当一个实时信号发送给一个进程时，不管该信号是否已经在进程中注册，都会被再注册一次。因此，信号不会丢失。这意味着同一个实时信号可以在同一个进程的未决信号信息链中占有多个 sigqueue 结构。这是因为进程每收到一个实时信号，都会为它分配一个结构来注册该信号信息，并把该结构添加在未决信号链尾，即所有诞生的实时信号都会在目标进程中注册。

当一个非实时信号发送给一个进程时，如果该信号已经在进程中注册，则该信号将被丢弃，造成信号丢失。这意味着同一个非实时信号在进程的未决信号信息链中，至多占有一个 sigqueue 结构。一个非实时信号诞生后，如果发现相同的信号已经在目标结构中注册，则不必再注册，对于进程来说，相当于不知道本次信号发生，信号丢失；如果进程的未决信号中没有相同信号，则在进程中注册该信号。

3. 信号在进程中注销

在目标进程执行过程中，会检测是否有信号等待处理(每次从内核空间返回到用户空间时都会检查)。如果存在未决信号等待处理且该信号没有被进程阻塞，则在运行相应的信号处理函数前，进程会把信号在未决信号链中占有的结构卸掉。

是否将信号从进程未决信号集中删除对于实时与非实时信号是不同的。对于非实时信号来说，由于在未决信号信息链中最多只占有一个 sigqueue 结构，因此该结构被释放后，应该把信号在进程未决信号集中删除(信号注销完毕)；而对于实时信号来说，可能在未决信号信息链中占有多个 sigqueue 结构，因此应该针对占有 sigqueue 结构的数目区别对待：如果只占有一个 sigqueue 结构(进程只收到该信号一次)，则应该把信号在进程的未决信号集中删除(信号注销完毕)。否则，不应该在进程的未决信号集中删除该信号。

4. 信号生命终止

进程注销信号后，立即执行相应的信号处理函数，执行完毕后，信号的本次发送对进程的影响彻底结束。

3.2.2.4 信号处理函数

Linux 信号处理函数可分为信号安装函数、信号发送函数和信号集操作函数。

1. 信号安装函数

信号安装函数用来设置某个信号的处理函数，主要有 signal( )和 sigaction( )。sigaction( )主要用于与 sigqueue( )配合使用，处理实时信号。使用这两个函数时，必须包含以下头文件：

```
#include <signal.h>
```

(1) signal( )函数

```
void (*signal(int signum, void (*handler)(int)))(int);
```

参数 signum 指出要设置处理函数的信号。参数 handler 是一个处理函数，也可以是 SIG_IGN 或 SIG_DFL。SIG_IGN 是使用忽略该信号的操作函数，而 SIG_DEL 是使用默认的信号操作函数。

传递给信号处理函数的整数参数是信号值，signal( )返回值是指定信号 signum 前一次的处理函数，或者错误时返回错误代码 SIG_ERR。下例(signaltest.c)给出 signal( )函数调用示例：

```
/***** signaltest.c *****/
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
void sigroutine(int dunno)
{      /*信号处理函数，其中 dunno 将会得到信号的值*/
   switch (dunno) {
      case 1:
         printf("Get a signal-SIGHUP\n ");
```

```
        break;
      case 2:
        printf("Get a signal -SIGINT\n ");
        break;
      case 3:
        printf("Get a signal -SIGQUIT\n");
        break;
    }
    return;
    }

    int main( )
    {
     printf("process id is %d ", getpid( ));
     signal(SIGHUP, sigroutine);          /*下面设置 3 个信号处理函数*/
     signal(SIGINT, sigroutine);
     signal(SIGQUIT, sigroutine);
     for (;;);
}
```

其中信号 SIGINT 由按 Ctrl+C 键发出，信号 SIGQUIT 由按 Ctrl+\组合键发出。该程序执行的结果如下：

```
localhost:~$ ./sigaltest
process id is 463
Get a signal -SIGINT                  /*按 Ctrl+C 组合键得到的结果*/
Get a signal –SIGQUIT                 /*按 Ctrl+\组合键得到的结果*/
[1]+ Stopped ./signaltest
localhost:~$ bg
[1]+ ./signaltest &
localhost:~$ kill -HUP 463            /*向进程发送 SIGHUP 信号*/
localhost:~$ Get a signal – SIGHUP
kill -9 463                           /*向进程发送 SIGKILL 信号，终止进程*/
localhost:~$
```

(2) sigaction( )函数

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oact);
struct sigaction {
   void (*sa_handler)(int signum);
   void (*sa_sigaction)(int siginfo_t *info, void *act);
   sigset_t sa_mask;
   int sa_flags;
```

```
    void (*sa_restore)(void);
}
```

参数 signum 指出要设置处理函数的信号。参数 act 包含对该信号进行如何处理的信息。参数 oact 指以前对这个信号的处理信息，主要用来保存信息，一般选 NULL。需要注意，SIGKILL 和 SIGSTOP 不能通过 sigaction 注册使用。

sa_handler 是一个函数型指针，指向与信号处理对应的处理函数。sa_sigaction、sa_restore 的功能与 sa_handler 类似，但参数不同。sa_flags 用来设置各种信号操作，但一般设置为 0。实际应用中，一般只需将 sa_handler 指向相应的信号处理函数，也可以是 SIG_IGN 或 SIG_DFL。

以下代码(signalactiontest.c)显示如何利用 sigaction( )函数捕捉用户从键盘按 Ctrl+C 组合键的信号，并输出一个提示语句。

```
/***** signalactiontest.c *****/
#define PROMPT"你想终止程序吗?"
char *prompt=PROMPT;
void ctrl_c_op(int signo)
{
  write(STDERR_FILENO,prompt,strlen(prompt));
}

int main( )
{
  struct sigaction act;
  act.sa_handler=ctrl_c_op;
  sigemptyset(&act.sa_mask);
  act.sa_flags=0;
  if(sigaction(SIGINT,&act,NULL)<0){
   fprintf(stderr,"Install Signal Action Error:%s\n\a",strerror(errno));
   exit(1);
  }
  while(1);
}
```

2. 信号发送函数

信号事件的发生有两个来源：一个是硬件原因(如按键盘键)，一个是软件原因(如函数或命令发出信号)。最常用的 5 个发出信号的函数是 kill( )、raise( )、alarm( )、setitimer( )和 pause( )。使用这些函数时，需包含以下两个头文件：

```
#include <sys/types.h>
#include <signal.h>
```

(1) kill( )函数

kill( )函数用来向进程或进程组发送一个信号。该函数声明的格式如下：

```
int kill(pid_t pid, int sig);
```

参数 pid 指接收信号的进程(组)的进程号。如果参数 pid>0，该函数将信号 sig 发送到进程号为 pid 的进程。如果 pid=0，信号 sig 将发送给当前进程所属进程组里的所有进程。如果参数 pid= –1，信号 sig 将发送给除了 1 号进程和自身以外的所有进程。如果参数 pid< –1，信号 sig 将发送给属于进程组–pid 的所有进程。

参数 sig 指发送的信号。如果参数 sig 为 0，将不发送信号。该函数执行成功时，返回值为 0；错误时，返回–1，并设置相应错误代码 errno。常见的错误代码包括以下 3 种。

• EINVAL：指定的信号 sig 无效。

• SRCH：参数 pid 指定的进程或进程组不存在。注意，在进程表项中存在的进程，可能是一个还没有被 wait( )函数收回，但已经终止执行的僵死进程。

• EPERM：进程没有权力将这个信号发送到指定接收信号的进程。因为，一个进程被允许将信号发送到进程 pid 时，必须拥有 root 权力，或者是发出调用的进程的 UID 或 EUID 与指定接收的进程的 UID 或保存用户 ID(savedset-user-ID)相同。如果参数 pid 小于–1，即该信号发送给一个组，则该错误表示组中有成员进程不能接收该信号。

(2) raise( )函数

raise( )函数用于向自身发送信号。该函数声明格式如下：

```
int raise(int sig);
```

调用成功时返回 0，出错时返回–1。

从原型上可以看出，raise( )可以通过 kill( )实现。raise(signo)等价于：

```
kill(getpid( ), signo);
```

(3) alarm( )函数

alarm( )函数的功能是设置一个定时器，当定时器计时到达时，将发出信号 SIGALRM 给进程。该函数声明格式如下：

```
unsigned int alarm(unsigned int seconds);
```

如果参数 seconds 为 0，则之前设置的闹钟会被取消，并将剩下的时间返回。返回值返回之前闹钟的剩余秒数，如果之前未设闹钟则返回 0。注意，在使用时，alarm( )只设置为发送一次信号，如果要多次发送，就要多次调用 alarm( )函数。

(4) setitimer( )函数

现在的系统中很多程序不再使用 alarm( )函数，而使用 setitimer( )函数来设置定时器，用 getitimer( )来得到定时器状态，这两个函数的声明格式如下：

```
int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);
```

在使用这两个函数的进程程序中加入以下头文件：

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

该函数给进程提供 3 个逻辑定时器，各自有其独有的计时域，当其中任何一个到达，就发送一个相应信号给进程，并使得定时器重新开始工作。3 个定时器由参数 which 指定。

- TIMER_REAL：按实际时间计时，计时到达将向进程发送 SIGALRM 信号。
- ITIMER_VIRTUAL：仅当进程执行时才进行计时。计时到达将向进程发送 SIGVTALRM 信号。
- ITIMER_PROF：当进程执行时和系统为该进程执行动作(如调度)时都计时。与 ITIMER_VIRTUAL 是一对，该定时器经常用来统计进程在用户态和内核态花费的时间，计时到达将向进程发送 SIGPROF 信号。

定时器中的参数 value 用来指明定时器的时间，其结构如下：

```
struct itimerval {
    struct timeval it_interval;      /*下一次取值*/
    struct timeval it_value;         /*本次设置值*/
};
```

该结构中 timeval 结构定义如下：

```
struct timeval {
    long tv_sec;                     /*秒(s)*/
    long tv_usec;                    /*微秒(μs)，1s=10^6μs*/
};
```

在 setitimer( )函数中，参数 ovalue 如果不为空，则其中保留的是上次调用设置的值。定时器将 it_value 递减到 0 时，产生一个信号，并将 it_value 的值设置为 it_interval 的值，然后重新开始计时，如此往复。当 it_value 设置为 0 时，定时器停止，或者当它计时到期，而 it_interval 为 0 时停止。调用成功时，返回 0；错误时，返回–1，并设置相应错误代码 errno。错误代码可有以下两种。

- EFAULT：参数 value 或 ovalue 是无效指针。
- EINVAL：参数 which 不是 ITIMER_REAL、ITIMER_VIRT 或 ITIMER_PROF 中的一个。

以下代码(settimertest.c)给出 setitimer( )函数示例，每隔 1s 发出一个 SIGALRM，每隔 0.5s 发出一个 SIGVTALRM 信号。

```
/***** settimertest.c *****/
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/time.h>
int sec;
void sigroutine(int signo)
{
   switch (signo) {
      case SIGALRM:
```

```
            printf("Catch a signal-SIGALRM ");
            break;
        case SIGVTALRM:
            printf("Catch a signal-SIGVTALRM ");
            break;
    }
    return;
}
int main( )
{
    struct itimerval value,ovalue,value2;
    sec = 5;
    printf("process id is %d ",getpid( ));
    signal(SIGALRM, sigroutine);
    signal(SIGVTALRM, sigroutine);
    value.it_value.tv_sec = 1;
    value.it_value.tv_usec = 0;
    value.it_interval.tv_sec = 1;
    value.it_interval.tv_usec = 0;
    setitimer(ITIMER_REAL, &value, &ovalue);
        value2.it_value.tv_sec = 0;
        value2.it_value.tv_usec = 500000;
        value2.it_interval.tv_sec = 0;
        value2.it_interval.tv_usec = 500000;
        setitimer(ITIMER_VIRTUAL, &value2, &ovalue);
        for (;;) ;
    }
```

该例的运行结果如下：

```
localhost:~$ ./settimertest
process id is 579
Catch a signal –SIGVTALRM
Catch a signal –SIGALRM
Catch a signal –SIGVTALRM
Catch a signal –SIGVTALRM
Catch a signal –SIGALRM
Catch a signal –SIGVTALRM
```

(5) pause( )函数

pause 使当前进程暂停，进入睡眠状态，直到被信号所中断。该函数声明格式如下：

```
int pause(void);
```

该函数只返回–1。

下例(alarmtest.c)给出使用 alarm( )和 pause( )实现 sleep( )功能的示例：

```
/*****alarmtest.c*****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
void my_alarm_handle(int sign_no)
{
 if (sign_no == SIGALRM) {
  printf("I have been waken up.\n");
 }
}

int main( )
{
 printf("sleep for 5s ... ...\n");
 signal(SIGALRM, my_alarm_handle);
 alarm(5);
 pause( );
 return 0 ;
}
```

3. 信号操作函数

有时用户希望进程连续地执行，不受到信号的影响，此时需要对信号执行信号操作。信号操作最常用的函数是信号屏蔽，主要操作函数包括 sigemptyset( )、sigfillset( )、sigaddset( )、sigdelset( )、sigismember( )和 sigprocmask( )。使用这些函数时，需包含以下头文件：

```
#include<signal.h>
```

这些函数的声明格式如下：

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(sigset_t *set, int signo);
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

sigemptyset( )初始化信号集合 set，将 set 设置为空。sigfillset( )也初始化信号集合，但只将信号集合设置为所有信号的集合。sigaddset( )将信号 signo 加入到信号集合之中，sigdelset( )将信号从信号集合中删除，sigismember( )查询信号是否在信号集合之中。

sigprocmask( )是最为关键的一个函数，在使用之前要先设置好信号集合 set。该函数将指定

的信号集合 set 加入到进程的信号阻塞集合中去。如果提供 oset，那么当前的进程信号阻塞集合将会保存在 oset 中。参数 how 决定函数的操作方式，取值有以下几种。

- SIG_BLOCK：添加一个信号集合到当前进程的阻塞集合之中。
- SIG_UNBLOCK：从当前的阻塞集合之中删除一个信号集合。
- SIG_SETMASK：将当前的信号集合设置为信号阻塞集合。

下例(siginttest.c)展示这些函数的使用方法：

```
/***** siginttest.c *****/
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
/*自定义 SIGINT 的处理函数，如果按 Ctrl+C 键，则会打印提示，而不是默认的退出*/
void my_func(int sigo_num)
{
   printf("If you want to quit, please try 'ctrl+\\' .\n");
}

int main( )
{
sigset_t set;
struct sigaction action1, action2;
 /*初始化信号集为空*/
 if (sigemptyset(&set) < 0) {
    perror("sigemptyset");
    exit(1);
  }
  /*将相应的信号加入信号集*/
  if (sigaddset(&set, SIGQUIT) < 0) {
    perror("sigaddset SIGQUIT");
    exit(1);
 }
 if (sigaddset(&set, SIGINT) < 0) {
   perror("sigaddset SIGINT");
   exit(1);
}
/*设置信号屏蔽字*/
if (sigprocmask(SIG_BLOCK, &set, NULL) < 0) {
  perror("sigprocmask SIG_BLOCK");
  exit(1);
```

```
} else {
    printf("blocked,and sleep for 5s ...\n");
    sleep(5);
 }
 if (sigprocmask(SIG_UNBLOCK, &set, NULL) < 0) {
    perror("sigprocmask SIG_UNBLOCK");
    exit(1);
 } else {
       printf("unblock\n");
       /*此处可以添加函数功能模块 process( )*/
       sleep(2);
       printf("If you want to quit this program, please try ...\n");
 }      /*对相应的信号进行循环处理*/
while (1) {
       if (sigismember(&set, SIGINT)) {
         sigemptyset(&action1.sa_mask);
         action1.sa_handler = my_func;
         sigaction(SIGINT, &action1, NULL);
       }else
       if (sigismember(&set, SIGQUIT)) {
          sigemptyset(&action2.sa_mask);
          /*SIG_DFL 采用默认的方式处理*/
          action2.sa_handler = SIG_DFL;
          sigaction(SIGTERM, &action2, NULL);
        }
      }
   return 0;
  }
```

编译后运行该程序，其结果如下：

```
localhost:~$./siginttest
blocked,and sleep for 5s ...
unblock
If you want to quit this program, please try ...
If you want to quit, please try 'ctrl+\' .
localhost:~$
```

3.2.3 管道通信

管道技术是 Linux 操作系统中由来已久的一种进程间通信机制，分为匿名管道(pipe)和命名管道(FIFO)，它们都通过内核缓冲区按先进先出的方式传输数据。对于管道，可以形象地当作

是连接两个命令或应用程序的一个单向连接器。例如，下面的命令行：

```
ls -1 | wc -l
```

该命令行首先创建两个进程，一个对应于 ls –1，另一个对应于 wc –l。然后，把第 1 个进程的标准输出设为第 2 个进程的标准输入，它的作用是计算当前目录下的文件数量。此例实际上是在两个命令之间建立一个管道，其中第 1 个命令 ls 执行后产生的输出作为第 2 个命令 wc 的输入，这是一个半双工通信，因为通信是单向的。两个命令之间连接的具体工作是由内核来完成的。下面将会看到，除了命令之外，应用程序也可以使用管道进行连接。

本节将分别介绍这两种通信机制。

3.2.3.1 匿名管道

匿名管道是 Linux 支持的最初 UNIX 通信机制之一，具有以下特点。

• 匿名管道是半双工的，数据只能向一个方向流动；要求双向通信时，需要建立两个匿名管道。

• 匿名管道只能用于具有亲缘关系的进程间通信，亲缘关系指的是具有共同祖先，如父子进程或者兄弟进程之间。

• 匿名管道对于管道两端的进程而言，就是一个文件，但它不是普通文件，而是一个只存在于主存中的特殊文件。

• 一个进程向管道中写入的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。

匿名管道的创建通过函数 pipe( )实现，其声明格式如下：

```
#include <unistd.h>
int pipe(int fd[2])
```

当调用成功时，函数 pipe 返回值为 0，否则返回值为–1。成功返回时，数组 fd 被填入两个有效的文件描述符。数组第 1 个元素中的文件描述符供应用程序读取，数组的第 2 个元素中的文件描述符可用来供应用程序写入。

对文件系统来说，匿名管道是不可见的，它的作用仅限于在父进程和子进程两个进程间进行通信。因此，一个进程在使用 pipe( )创建管道后，将会再用 fork( )函数复制一个子进程，然后通过管道实现父子进程间的通信。

匿名管道两端可分别用描述字 fd[0]及 fd[1]来描述，管道两端的任务是固定的。习惯上，描述符 fd[0]表示管道读端，描述符 fd[1]表示管道写端，按惯例编程，会使程序的可移植性更好。一般文件的 I/O 函数都可以用于管道，如 close( )、read( )和 write( )等。

从管道中读取数据时，如果管道的写端不存在，则认为已经读到数据的末尾，读函数返回的读出字节数为 0。当管道的写端存在时，此时如果请求的字节数目大于 PIPE_BUF，则返回管道中现有的数据字节数；如果请求的字节数目不大于 PIPE_BUF，则返回管道中现有数据字节数(此时管道中数据量小于请求的数据量)，或者返回请求的字节数(此时管道中数据量不小于

请求的数据量)。PIPE_BUF 在 include/Linux/limits.h 中定义，不同的内核版本可能会有所差异。

当向管道中写入数据时，Linux 不保证写入的原子性，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读出管道缓冲区中的数据，那么写操作将一直阻塞。需要注意，只有在管道的读端存在时，向管道中写入数据才有意义。否则，向管道中写入数据的进程将收到内核传来的 SIGPIPE 信号，应用程序可以处理该信号，也可以忽略(默认动作则是进程终止)。

下面给出一个在包含多个进程的应用程序中使用匿名管道的示例。该程序中，进程通过 fork( )分叉，变成一个父进程和一个子进程。在子进程中，尝试从管道的输入描述符读取。此时，子进程将被挂起，直到管道中有可以读取的内容为止。读完后，用 NULL 作为读取的内容的结束符，这样可确保使用 printf( )函数正确打印读取的内容。父进程先是利用存放在 thePipe[1]中的“写文件标识符”向管道写入测试字符串，然后使用 wait( )函数来等待子进程退出。

需要注意，fork( )函数一旦执行，子进程会继承父进程的功能和管道的文件描述符。但对于内核来说，父进程和子进程是平等的，它们是独立运行的。也就是说，两个进程分别具有独立的主存空间，它们正是通过管道来互通有无的。

下例(pipetest.c)展示 pipe( )的使用方法：

```
/***** pipetest.c *****/
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <wait.h>
#define MAX_LINE 80
   int main( )
{
      int thePipe[2], ret;
      char buf[MAX_LINE+1];
      const char *testbuf="a test string.";
      if ( pipe( thePipe ) == 0 ) {
        if (fork( ) == 0) {
          ret = read( thePipe[0], buf, MAX_LINE );
          buf[ret] = 0;
          printf( "Child read %s\n", buf );
        } else {
          ret = write( thePipe[1], testbuf, strlen(testbuf) );
          ret = wait( NULL );
        }
    }
    close(thePipe[0] );
```

```
        close(thePipe[1] );
        return 0;
    }
```

匿名管道虽然为进程间通信提供解决途径，但在实际运用中也有一些局限性。

- 只支持单向数据流。
- 只能用于具有亲缘关系的进程之间。
- 没有名字。
- 缓冲区有限，管道只存在于主存中，大小为一个页面。
- 所传送的是无格式字节流，这就要求管道的读出者和写入者必须事先约定好数据的格式，如消息(或命令、记录)长度等。

3.2.3.2 命名管道

命名管道(named pipe 或 FIFO)克服了匿名管道只能用于具有亲缘关系的进程之间通信的限制。命名管道不同于匿名管道之处在于它提供一个与路径名的关联，以命名管道的文件形式存在于文件系统中。这样，与命名管道的创建进程不存在亲缘关系的进程，只要可以访问该路径，即可彼此通过命名管道相互通信。因此，通过命名管道，不相关的进程也能交换数据。需注意的是，命名管道严格遵循“先进先出”原则，对管道及命名管道的读总是从开始处返回数据，对它们的写则把数据添加到末尾。命名管道不支持 lseek( )等文件定位操作。

命名管道的创建通过函数 mkfifo( )实现，该调用的函数声明格式如下：

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char * pathname, mode_t mode)
```

mkfifo( )的作用是在文件系统中创建一个文件，该文件用于提供命名管道功能。第 1 个参数(pathname)是将在文件系统中创建的一个专用文件，也就是创建后命名管道的名字。第 2 个参数(mode)用来规定命名管道的读写权限。mkfifo( )如果调用成功，返回值为 0；如果调用失败，返回值为–1。如果 mkfifo( )的第 1 个参数是一个已经存在的路径名，则返回 EEXIST 错误。因此，典型的调用代码首先会检查是否返回该错误。一般文件的 I/O 函数都可以用于命名管道，如 close( )、read( )和 write( )等。下例给出使用 mkfifo( )创建命名管道的代码框架：

```
int ret;
ret = mkfifo( "/tmp/cmd_pipe", S_IFIFO | 0666 );
 if (ret = = 0) {
        /*成功建立命名管道*/
 } else {
        /*创建命名管道失败*/
}
```

该代码利用/tmp 目录中的 cmd_pipe 文件建立一个命名管道。随后，可打开该文件进行读

写操作，并以此进行通信。命名管道一旦打开，可以利用典型的 I/O 输出函数从中读取内容。

命名管道比匿名管道多了一个打开操作 open( )。命名管道的打开规则如下：如果当前打开操作是为读而打开命名管道，若已经有相应进程为写而打开该命名管道，则当前打开操作将成功返回；否则，可能阻塞，直到有相应进程为写而打开该命名管道(当前打开操作设置了阻塞标志)；或者成功返回(当前打开操作没有设置阻塞标志)。当前打开操作是为写而打开命名管道时，如果已经有相应进程为读而打开该命名管道，则当前打开操作将成功返回；否则，可能阻塞，直到有相应进程为读而打开该命名管道(当前打开操作设置了阻塞标志)；或者返回 ENXIO 错误(当前打开操作没有设置阻塞标志)。

从命名管道中读取数据时，如果一个进程为了从命名管道中读取数据而阻塞打开命名管道，那么称该进程内的读操作为设置了阻塞标志的读操作。如果有进程为写而打开命名管道，且当前命名管道内没有数据，则对于设置了阻塞标志的读操作来说，将一直阻塞。对于没有设置阻塞标志的读操作来说则返回–1，当前 errno 值为 EAGAIN，提醒以后再试。

对于设置阻塞标志的读操作来说，造成阻塞的原因有两种：当前命名管道内有数据，但有其他进程在读这些数据；另外，就是命名管道内没有数据。解除阻塞的原因则是命名管道中有新的数据写入，不论写入数据量的大小，也不论读操作请求多少数据量。

读打开的阻塞标志只对本进程第 1 个读操作起作用，如果本进程内有多个读操作序列，则在第 1 个读操作被唤醒并完成读操作后，其他将要执行的读操作将不再阻塞，即使在执行读操作时，命名管道中没有数据也一样(此时，读操作返回 0)。如果没有进程为写而打开命名管道，则设置了阻塞标志的读操作会阻塞。需要注意，如果命名管道中有数据，则设置阻塞标志的读操作不会因为命名管道中的字节数小于请求读的字节数而阻塞，此时，读操作会返回命名管道中现有的数据量。

向命名管道中写入数据时，如果一个进程为了向命名管道中写入数据而阻塞打开命名管道，那么称该进程内的写操作为设置了阻塞标志的写操作。对于设置了阻塞标志的写操作，当要写入的数据量不大于 PIPE_BUF 时，内核将保证写入的原子性。如果此时管道空闲缓冲区不足以容纳要写入的字节数，则进入睡眠，直到缓冲区中能够容纳要写入的字节数时，才开始进行一次性写操作。要写入的数据量大于 PIPE_BUF 时，内核将不再保证写入的原子性。命名管道缓冲区一有空闲区域，写进程就会试图向管道写入数据，写操作在写完所有请求写的数据后返回。

对于没有设置阻塞标志的写操作，当要写入的数据量大于 PIPE_BUF 时，内核将不再保证写入的原子性。在写满所有命名管道空闲缓冲区后，写操作返回。当要写入的数据量不大于 PIPE_BUF 时，内核将保证写入的原子性。如果当前命名管道空闲缓冲区能够容纳请求写入的字节数，写完后成功返回；如果当前命名管道空闲缓冲区不能够容纳请求写入的字节数，则返回 EAGAIN 错误，提醒以后再写。

下面给出读写命名管道的两个程序，可通过修改程序或者更改程序执行的命令行参数实现

对各种命名管道读写规则的验证。

代码段 1(fifotest1.c)：写命名管道的程序。

```
/***** fifotest1.c *****/
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#define FIFO_SERVER "/tmp/fifoserver"

main(int argc,char** argv)
{       /*参数为即将写入的字节数*/
    int fd;
    char w_buf[4096*2];
    int real_wnum;
    memset(w_buf,0,4096*2);
    if((mkfifo(FIFO_SERVER,O_CREAT|O_EXCL)<0)&&(errno!=EEXIST))
        printf("cannot create fifoserver\n");
    if(fd= =-1)
        if(errno= =ENXIO)
            printf("open error; no reading process\n");
        fd=open(FIFO_SERVER,O_WRONLY|O_NONBLOCK,0);
    /*设置非阻塞标志*/
    /*若需要使用阻塞模式，则用语句 fd=open(FIFO_SERVER,O_WRONLY,0); */
    /*设置阻塞标志*/
    real_wnum=write(fd,w_buf,2048);
    if(real_wnum= = -1) {
        if(errno= =EAGAIN)
            printf("write to fifo error; try later\n");
    } else
        printf("real write num is %d\n",real_wnum);
    real_wnum=write(fd,w_buf,5000);
    /*5000 用于测试写入字节大于 4096 时的非原子性*/
    /*若需要测试写入字节不大于 4096 时原子性，则用语句 real_wnum=write(fd,w_buf,4096); */
    /*4096 用于测试写入字节不大于 4096 时的原子性*/
    if(real_wnum= = -1)
        if(errno= =EAGAIN)
            printf("try later\n");
}
```

代码段 2(fifotest2.c)：与代码段 1 一起测试写命名管道的规则，第 1 个命令行参数是请求从

命名管道读出的字节数。

```
/***** fifotest2.c*****/
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#define FIFO_SERVER "/tmp/fifoserver"

main(int argc,char** argv)
{
    char r_buf[4096*2];
    int  fd;
    int  r_size;
    int  ret_size;
    r_size=atoi(argv[1]);
    printf("requred real read bytes %d\n",r_size);
    memset(r_buf,0,sizeof(r_buf));
    fd=open(FIFO_SERVER,O_RDONLY|O_NONBLOCK,0);
    /*在此处可以把读程序编译成两个不同版本：阻塞版本及非阻塞版本*/
    if(fd= = –1) {
        printf("open %s for read error\n");
        exit( );
    }
    while(1) {
        memset(r_buf,0,sizeof(r_buf));
        ret_size=read(fd,r_buf,r_size);
        if(ret_size= = –1)
            if(errno= =EAGAIN)
                printf("no data available\n");
        printf("real read bytes %d\n",ret_size);
        sleep(1);
    }
    pause( );
    unlink(FIFO_SERVER);
}
```

用户可结合程序代码的注释说明，将代码段 1 编译成非阻塞且请求写的字节数大于 PIPE_BUF 的版本 nbwg、非阻塞且请求写的字节数不大于 PIPE_BUF 的版本 nbw、阻塞且请求写的字节数大于 PIPE_BUF 的版本 bwg、阻塞且请求写的字节数不大于 PIPE_BUF 的版本 bw

等 4 个版本，将代码段 2 编译成阻塞读版本 br 及非阻塞读版本 nbr 两个版本。用户可通过运行编译后的程序来验证阻塞及非阻塞写操作中数据写入的原子性及非原子性。

需要注意，不管写打开的阻塞标志是否设置，在请求写入的字节数大于 4096 时，都不保证写入的原子性，但两者有本质区别。对于阻塞写来说，写操作在写满命名管道的空闲区域后，会一直等待，直到写完所有数据为止，请求写入的数据最终都会写入命名管道；而非阻塞写则在写满命名管道的空闲区域后，就返回(实际写入的字节数)，所以有些数据最终不能够写入。

在 Linux 环境下，也可以在命令行中通过执行 mkfifo 命令来建立一个命名管道，它的功能与 mkfifo( )函数相似。mkfifo 命令的一般用法如下所示：

```
mkfifo [options] name
```

其中 options 一般为–m，即模式，用以指出读写权限，默认值为 0644。name 是要创建的管道的名称，必要时可加上路径。例如，若需在/tmp 目录下建立一个名为 cmd_pipe 的命名管道，可执行命令：

```
$ mkfifo-m 0644 /tmp/cmd_pipe
```

管道一旦建立，就能够在命令行下通过此管道进行通信。比如，可在一个终端上利用 cat 命令来读取管道：

```
$ cat cmd_pipe
```

当输入该命令后，进程就会被挂起，等待写入程序打开此管道。此时，在另一个终端上利用 echo 命令向这个命名管道写入：

```
$ echo Hi > cmd_pipe
```

该命令结束后，读取该管道的程序(即 cat)将被唤醒，随后结束。此时读取进程将得到如下结果：

```
$ cat cmd_pipe
Hi
$
```

除了 mkfifo 命令外，mknod 命令也可以用来创建命名管道，其用法如下：

```
$ mknod cmd_pipe p
```

该命令执行后，将在当前目录下创建一个命名管道 cmd_pipe，p 用于指出建立的是命名管道。

3.3 实 验 内 容

3.3.1 实验 1 信号通信

3.3.1.1 实验说明

利用信号通信机制在父子进程及兄弟进程间进行通信。

3.3.1.2 解决方案

父子进程之间的通信可分为阻塞型通信和非阻塞型通信两种情形。

在阻塞型通信情形中，父进程可以使用 wait( )或 waitpid( )函数等待子进程结束。为避免产生僵死子进程，也可以循环调用 wait( )或 watipid( )函数来等待所有子进程的结束。但此时最好的方法是让子进程在结束时，向父进程发送 SIGCHLD 信号，父进程通过 signal( )或 sigaction( )函数来响应子进程的结束。

3.3.1.3 阻塞型通信程序框架

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <wait.h>

void sigchld_handler(int sig)
{
  pid_t pid;
  int status;
  for(;(pid=waitpid(-1,&status,WNOHANG))>0;) {
    printf("child %d died :%d\n",pid,WEXITSTATUS(status));
    printf("hi,parent process received SIGHLD signal successfully!\n");
  }
  return;
}

void main( )
{
  /*创建子进程*/
  if(子进程) {
    /*输出子进程相关信息*/
    /*休眠一段时间*/
    /*退出*/
  }else
  if (父进程){
    /*调用信号处理函数*/
    /*暂停*/
  }else
```

```
    if(创建进程出错) {
        /*打印“创建进程出错”提示信息*/
        /*退出*/
    }
}
```

非阻塞型通信情形下父子进程共存，当产生相关信号，如 SIGINT 时，父子进程都能接收到此信号，只是先由父进程响应，再由父进程把此信号传递给子进程。但需要注意，如果父进程没有对该信号的自定义处理，则父子进程都接受信号的默认处理。例如，在没有对 SIGINT 自定义处理时，产生此信号，则父子进程都马上中止。如果父进程自定义了信号处理函数，则子进程一样接受此信号和其信号处理函数。此时，若子进程成为孤儿进程，则此时的子进程不会再接受此信号和其信号处理函数。

3.3.1.4 非阻塞型通信程序框架

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
void sigint_handler(int sig)
{
   printf("received SIGINT signal successed!\n");
   return;
}

void   main( )
{
      /*创建子进程*/
      if(子进程) {
            /*打印子进程号*/
            /*休眠一段时间*/
            /*打印休眠一段时间后的子进程号*/
            /*休眠一段时间*/
            /*打印第 2 次休眠一段时间后的子进程号*/
            /*退出*/
      }else
        if (父进程){
            /*调用信号处理函数*/
            /*暂停*/
      }else
        if(创建进程出错) {
```

```
            /*打印“创建进程出错”提示信息*/
            /*退出*/
        }
    }
```

3.3.2 实验2 匿名管道通信

3.3.2.1 实验说明

学习使用匿名管道在两进程间建立通信。

3.3.2.2 解决方案

匿名管道只能用于具有亲缘关系的两进程间通信。一个进程在由 pipe( )创建管道后，一般再用 fork( )函数复制一个子进程，然后通过管道实现父子进程间的通信，也可通过 fork( )函数复制多个进程实现兄弟进程之间的通信。管道两端可分别用描述字 fd[0]以及 fd[1]来描述，其中一端只能用于读，由描述字 fd[0]表示，称为管道读端；另一端则只能用于写，由描述字 fd[1]来表示，称为管道写端。

关于读写匿名管道的注意事项，请参见 3.2.3.1 节的有关内容。

以下给出匿名管道读写数据的关键代码，可对其进行扩充实现兄弟进程的通信。

3.3.2.3 程序框架

```
#include <wait.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define MAX_LINE 80
void main( )
{
    /*创建一个匿名管道*/
    if(管道创建成功) {
        /*创建子进程*/
        if((进程创建成功) {
            /*关闭写端*/
            /*休眠一段时间*/
            /*从管道读端读取数据并放入缓冲区*/
            /*打印“子进程读取数据成功”提示信息，并输出缓冲区数据*/
            /*关闭读端*/
            /*退出*/
```

```
            }else
              if(进程创建成功){
                /*关闭读端*/
                /*向管道写端写入数据*/
                /*打印“父进程写管道成功”提示信息*/
                /*关闭写端*/
                /*打印“父进程关闭写管道成功”提示信息*/
                /*休眠一段时间*/
            }
        }
        return 0;
    }
```

3.3.3 实验 3 命名管道通信

3.3.3.1 实验说明

学会使用命名管道在多进程间建立通信。

3.3.3.2 解决方案

命名管道以“先进先出”形式的文件存在于文件系统中。因此，只要可以访问该文件路径，就能够彼此通过命名管道相互通信。关于读写命名管道的注意事项，请参见 3.2.3.2 节的有关内容。

为实现多进程间基于命名管道的通信，首先使用 mkfifo( )创建一个命名管道。随后可使用一般的文件 I/O 函数，如 open( )、close( )、read( )、write( )等，来对它进行操作，从而实现多进程间的通信。

3.3.3.3 程序框架

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#define FIFO_SERVER "/tmp/fifoserver"
#define BUFFERSIZE 80

void main( )
{
```

```
if(创建命名管道失败) {
        /*打印“无法创建命名管道”错误提示信息*/
        /*退出*/
 }
 /*打印“成功创建命名管道”提示信息*/
 /*创建子进程*/
 if(子进程创建成功) {
        /*以写方式打开命名管道*/
        if(打开失败) {
              /*打印“无法打开命名管道”错误信息*/
              /*退出*/
        }
        /*向命名管道写入数据*/
        if(写入失败) {
              /*打印“写数据出错”提示信息*/
              /*退出*/
        }
        /*打印“成功写入数据”提示信息*/
        /*关闭命名管道*/
 }else
        if(父进程) {
              /*以只读方式打开命名管道*/
              /*输出读数据前缓冲区信息*/
              /*从命名管道读取数据到缓冲区*/
              /*输出读数据后缓冲区信息*/
              /*关闭命名管道*/
        }else {
              /*打印“创建进程出错”提示信息*/
              /*退出*/
        }
}
```

3.3.4 实验4 使用命名管道建立客户/服务器关联程序

3.3.4.1 实验说明

使用命名管道建立一个客户与服务器程序的关联，以便实现数据共享。

3.3.4.2 解决方案

为支持客户与服务器程序共享数据，程序首先在两端建立一个命名管道。随后一方可通过

该命名管道写入数据，另一方可通过该管道获取数据。程序设计的关键在于，协调好两端的读写过程，避免同时读或写。具体而言，可在程序中设置标记符来标识共享数据的结束。

3.3.4.3 程序框架

1. 服务器端程序

```
#include "csfifo.h"

int main(int argc,char *argv[ ]) {
char *clientfifo = "/tmp/cfifo";

while(1) {
    if(创建命名管道失败){
        /*打印“客户端尚未创建命名管道”提示信息*/
        /*休眠一段时间*/
        continue;
    }
    /*以只读方式打开命名管道*/
        if(打开失败) {
            /*打印“无法打开命名管道”错误提示信息*/
        }
        if(从管道读数据失败) {
            /*打印“进程无法从命名管道读取数据”错误提示信息*/
            /*返回*/
        } else {
            /*输出从管道获得的数据*/
            /*关闭管道*/
            /*统计读取字符的个数*/
            /*分配一个长度等于读取字符总数的字符串空间*/
            /*将读取的字符串写入新分配的字符串空间*/
            /*将新分配字符串空间字符转换成数字*/
            if(转换后数字等于 0) {
                /*打印“未从客户端获取到数据”提示信息*/
                /*返回*/
            }
            /*输出转换后的数字*/
            /*将命名管道名及接收到的字符数写入管道，发送给服务器*/
        }
    }
}
```

2. 客户程序

```
#include "csfifo.h"

int main(int argc,char *argv[ ])
{
    char r_msg[BUFSIZ];
    char *temp = " client message to server";
    char* cf = "/tmp/cfifo";
    char * s_msg ;
    char * cfifo;

    if(argc !=2) {
        printf("Usage: ./client    n\n ");
        return 0;
    }
    s_msg = malloc(strlen(temp)+strlen(argv[1])+sizeof(char));
    cfifo = malloc(strlen(cf)+strlen(argv[1]));
    strcpy(s_msg,argv[1]);
    strcpy(cfifo,cf);
    s_msg[strlen(argv[1])]='#';
    strcat(s_msg,temp);
    strcat(cfifo,argv[1]);

     if(创建命名管道 csfifo 失败) {
        /*打印“无法创建命名管道”错误信息*/
        /*返回*/
    }
    /*以读写模式打开命名管道 csfifo*/
    if(打开失败) {
        /*打印“无法打开命名管道”错误信息*/
        /*返回*/
    }
    /*将 s_msg 写入管道*/
    if(若写入失败) {
        /*打印“无法向命名管道写入数据”错误信息*/
        /*返回*/
    } else {
        /*打印“向命名管道写入数据成功”提示信息*/
    }
```

```
while(1) {
    /*休眠一段时间*/
    if(打开命名管道 cfifo 失败) {
        /*打印“命名管道 cfifo 尚未被创建，请稍候”提示信息*/
        /*休眠一段时间*/
        continue;
    }
    /*以只读模式打开命名管道 cfifo; */
        if(打开失败) {
            /*打印“无法打开命名管道”错误信息*/
        }
        /*从命名管道读取数据*/
        if(若读取数据出错) {
            /*打印“从命名管道读取数据”提示信息*/
            /*返回*/
        } else {
            /*输出从命名管道读取的数据*/
            /*返回*/
        }
    }
```

3. csfifo.h

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#define CSFIFO    "/tmp/csfifo"
```

第4章 System V 的进程间通信

4.1 实 验 目 的

- 理解 System V 的进程间通信机制。
- 掌握和使用共享主存实现进程间通信。
- 掌握和使用消息队列实现进程间通信。
- 掌握和使用信号量实现进程同步。

4.2 背 景 知 识

4.2.1 System V 的进程间通信机制

4.2.1.1 System V 的进程间通信共性描述

1. Linux 的 System V 的进程间通信机制

Linux 继承了 UNIX System V 的进程间通信机制，即消息队列、信号量和共享主存。通常把消息、信号量和共享主存统称为 System V IPC 对象或 IPC 资源，每个对象都具有同样类型的接口——函数。这组函数为应用进程提供 3 种服务：

- 通过信号量实现与其他进程的同步。
- 通过消息队列以异步方式为通信频繁但数据量少的进程间通信提供服务。
- 通过共享主存，为大数据量的进程间通信提供服务。共享主存是 IPC 机制中最快、最简单的一种，但为防止出现竞争条件，需要配合使用信号量或锁机制。

System V 的进程间通信机制的主要特点包括：

- 进程间需要通过 IPC 对象通信时，必须在函数中传递该对象的唯一 IPC 标识符。
- 对 IPC 对象的访问，必须经过权限验证。对象访问权限的设置，由对象的创建者利用函数来实现。
- IPC 通信机制都把 IPC 对象的 IPC 标识符作为对系统资源表的索引。

2. Linux 提供的 IPC 函数

(1) IPC 对象创建函数

每个 IPC 对象都有一个 32 位的 IPC 键(IPC key)和一个 32 位的 IPC 标识符(IPC identifier)。

IPC 标识符由内核分配给 IPC 对象，在系统内部是唯一的，而 IPC 键是 IPC 对象的外部表示，可由程序员选择。如果键是公用的，则系统中所有进程通过权限检查后，均可找到和访问相应 IPC 对象；如果键是私有的，则键值为 0。每个进程都可建立一个键值为 0 的私有 IPC 对象。

进程使用函数 ipc( )进行通信，并传入参数。执行 ipc( )时，通过中断机制进入内核，该函数位于函数表的第 117 个表项。被调用时，系统从中取出进程通信函数的总入口地址，并跳转到内核函数 sys_ipc( )的起始处。再根据用户传入的参数，用若干 case 语句决定具体调用哪个内核函数。可通过以下库函数创建所需类型的 IPC 对象。

- semget( )：获得信号量的 IPC 标识符。
- msgget( )：获得消息队列的 IPC 标识符。
- shmget( )：获得共享主存的 IPC 标识符。

上述函数将从调用进程传递的 IPC 键传递给以 sys_开头的内核函数，并为用户分配一个与 IPC 对象相对应的数据结构，然后返回一个 32 位的 IPC 标识符，进程使用此标识符对该资源进行访问。

(2) IPC 资源控制函数

创建 IPC 对象之后，用户可通过以下库函数对 IPC 对象进行控制。这些函数为用户提供一组用于获得和设置资源状态信息的调用接口。

- semctl( )：对信号量资源进行控制。
- msgctl( )：对消息队列进行控制。
- shmctl( )：对共享主存进行控制。

(3) IPC 资源操作函数

除可对 IPC 对象进行控制处理之外，还可以通过专用函数对其进行其他功能性操作。

- semop( )：用于对信号量资源进行操作，获得或释放一个 IPC 信号量。
- msgsnd( )及 msgrcv( )：分别发送和接收一个 IPC 消息。
- shmat( )及 shmdt( )：分别将一个 IPC 共享主存区附加到进程的虚拟地址空间，以及把共享主存区从进程的虚拟地址空间剥离出去。

3. IPC 机制的公共元素及属性

3 种 IPC 对象的编程接口及实现方法类似，为避免重复，将共有的元素和属性称为公共元素及公共属性。IPC 数据结构是在进程请求 IPC 对象时动态创建的，任何进程都可以使用它。即使祖先进程创建 IPC 对象，但不属于该祖先进程派生而来的进程也可以使用该 IPC 对象。每个 IPC 对象都有一个 ipc_perm 数据结构，用于描述其属性。ipc_perm 在 Linux/ipc.h 中的定义如下：

```
struct ipc_perm {
    __kernel_key_t key;      /*IPC 键*/
    __kernel_uid_t uid;      /*资源所有者的 UID*/
    __kernel_gid_t gid;      /*资源所有者的 GID*/
```

```
    __kernel_uid_t cuid;    /*创建这个资源的进程 UID */
    __kernel_gid_t cgid;    /*创建这个资源的进程 GID*/
    __kernel_mode_t mode; /*文件系统类型的权限*/
    unsigned short seq;     /*位置使用序号*/
};
```

该结构中的成员 key 即为 IPC 键，其值为整型，由用户提供，用于申请一个 IPC 标识符。成员 mode 指该资源的所有者、组以及其他用户对资源的读、写访问权限。成员 seq 表示位置使用序号，在计算 IPC 标识符时使用。

需要访问 IPC 对象时，需根据所请求的资源类型调用相对应的 XXXget( )函数来创建 IPC 对象，这些函数再通过系统调用机制分别调用 sys_XXXget( )内核函数，它们是：

```
int sys_semget(key_t key, int nsems, int semflg);
int sys_msgget(key_t key, int msgflg);
int sys_shmget(key_t key, int size, int shmflg);
```

这 3 个函数的形式极为相似，其功能是根据用户传递的第 1 个参数键 key 来获得或创建 IPC 对象，并返回代表该 IPC 对象的标识符。此后进程使用这个标识符访问对象。

这里需要解决两个问题：其一是使用共同 IPC 对象的一组进程如何得到一个与其他对象无关的、唯一的键，用以创建唯一的 IPC 对象；其二是这组进程如何知道自己需要共享的键。可通过以下方法产生键，并解决这两个问题。

- 服务器程序创建 IPC 对象时使用常量 IPC_PRIVATE，它保证创建一个系统中唯一的 IPC 对象和新键。随后，服务器程序把键写入特定文件中，所有客户机程序都从该文件中获取所需键。也可从父进程中得到键，从而避免文件操作。
- 使用 ftok( )函数。它允许从一个目录出发，加上一个群组标识，共同创建一个键。如果所有进程都遵守相同规则，一般而言，不同应用程序不会使用相同目录，也不会使用相同群组标识。因此，该方法可基本保证创建的键的唯一性，且便于相关进程共享，即只需所有共享进程都用相同规则产生。

通常，IPC 对象由服务器程序产生，只有当与键相关联的 IPC 对象不存在时才创建它。客户机程序只是使用 IPC 对象，因此，必须在 IPC 对象存在的情况下打开。

System V IPC 对象是全局的，一旦被创建，除非显式删除或系统重新引导，否则将一直存在于系统中而不会消失，已经发送的数据也不会被破坏。这和命名管道不同，如果使用命名管道的所有进程都关闭，其数据结构也就从主存中消失，不会因为还有没有取走的数据而保留，即使 PIPE 作为文件系统的一个 inode 一直存在于文件系统中。

此外，System V IPC 对象独立于文件系统而存在，不为文件系统所知。因此，不能用文件描述符进行相关的操作，也不能使用文件系统的多工操作(如 select( )、poll( )等函数)。

这些函数中使用的最后一个参数是 XXXflg 标志位，这个参数包括两个标志：IPC_CREAT 和 IPC_EXCL。用户可根据自己的需要使用这些标志，创建一个 IPC 资源。

3 个函数的主要操作步骤如下：

① 如果调用进程传递的 IPC 键是 IPC_PRIVATE，系统将调用 newque( )函数，创建一个新的 IPC 资源，并返回新 IPC 资源的 IPC 标识符。

② 如果 key 是由用户所选择的一个非 0 整数值，那么将调用 findkey( )来检查此键是否正在被使用。如果是，将返回与该键相对应的 IPC 标识符，否则返回–1。若 findkey( )返回–1，并且用户在使用上述 3 个函数时设置 IPC_CREAT 标志，系统将调用 newque( )函数创建一个新的 IPC 资源，并返回新创建的 IPC 标识符。否则返回 ENOENT 错误。

③ 如果 findkey( )返回值不是–1，而是一个 IPC 标识符，则说明用户给出的键正在被使用。因此，继续检查用户是否联合设置 IPC_CREAT 和 IPC_EXCL 两个标志，如果是，则返回错误 EEXIST。

④ 如果键 key 虽然被使用，但调用进程没有联合设置 IPC_CREAT 和 IPC_EXCL 标志，那么系统将检查调用者是否有访问它的权限。若没有访问权限，将返回错误 EACCES。否则，调用进程可以使用这个 IPC 资源，并返回资源的 IPC 标识符。

如果这 3 个函数运行正常，则返回一个正的 IPC 标识符。否则，将返回一个错误码。

4.2.1.2 System V 进程间通信的基本操作

1. key_t 键和 ftok( )函数

建立 IPC 通信(如消息队列、共享主存时)时，必须指定一个 id 值。通常情况下，该 id 值通过 ftok( )函数得到，也可以由用户直接指定。ftok( )函数把一个已存在的文件名(该文件必须是存在而且可以访问的)和一个整数标识符 id 转换成一个 key_t 值，称为 IPC 键或键值。该函数定义如下：

```
key_t ftok( char * filename, int id );
```

在 Linux 系统实现中，调用该函数时，系统将文件的索引节点号取出，并在前面加上子序号，从而得到 key_t 的返回值。当删除并重建该文件后，索引节点号由操作系统根据当时文件系统的使用情况重新分配，因此可能与原来的不同。为确保 key_t 值不变，应保证传给函数 ftok( )的文件不被重建，或者直接指定一个固定的 key_t 值来取代调用 ftok( )函数。例如：

```
#define IPCKEY 0x111
char path[256];
sprintf( path, "%s/etc/config.ini", (char*)getenv("HOME") );
msgid=ftok( path, IPCKEY );
```

以上代码可保证不同用户下的程序获得互不干扰的 IPC 键值。

由于 etc/config.ini(假定)是应用系统的关键配置文件，因此不存在被轻易删除的问题，即使被删，也会很快被发现并重建(此时应用系统也将被重启)。

2. 创建与打开 IPC 对象

创建或打开一个 IPC 对象时，需要提供一个类型为 key_t 的 IPC 键，应用进程可选择以下

两种方式之一创建 IPC 键。

- 调用 ftok( )，给它传递 pathname 和 id。
- 指定 IPC_PRIVATE，它保证创建一个新的、唯一的 IPC 对象。

创建或打开一个 IPC 对象函数的另一个共同参数是 oflag，它指定 IPC 对象的读写权限位(ipc_perm 结构中的 mode 成员)，并选择是创建一个新的 IPC 对象还是访问一个存在的 IPC 对象。表 4-1 给出了选择键的规则。

表 4-1 选择键的规则

oflag 标志	IPC 对象不存在	IPC 对象已存在
无特殊标志	出错，errno=ENOENT	成功，引用已存在对象
IPC_CREAT	成功，创建新对象	成功，引用已存在对象
IPC_CREAT|IPC_EXCL	成功，创建新对象	出错，errno=EEXIST

需要注意，若只设置 IPC_EXCL 而不设置 IPC_CREAT，则对 IPC_EXCL 的设置没有意义。权限位的设置用八进制表示如下：

- 0400：由用户(属主)读。
- 0200：由用户(属主)写。
- 0040：由(属)组成员读。
- 0020：由(属)组成员写。
- 0004：由其他用户读。
- 0002：由其他用户写。

oflag 由选择参数和权限参数组合而成。

ipc_perm 结构的 cuid 和 cgid 成员分别设置创建者进程的有效用户 ID 和有效组 ID，这两个成员合称为创建者 ID。而 ipc_perm 结构的 uid 和 gid 成员分别设置为所有者进程的有效用户 ID 和有效组 ID，这两个成员合称为属主 ID。ipc_perm 结构中的 seq 成员是一个位置使用情况序号。该变量是一个由主存为在系统中的每个潜在的 IPC 对象维护的计数器。每当删除一个 IPC 对象时，主存就递增相应的序号，若溢出则循环回 0。这避免在短时间内重用 IPC 标识符。

4.2.2 消息队列

4.2.2.1 消息队列的基本概念

消息(message）是一个格式化的可变长信息单位。消息机制允许一个进程向任何其他进程发送一个消息。当一个进程收到多个消息时，可将它们排成一个消息队列。消息使用以下两种重要数据结构。

- 消息首部：其中记录与消息有关的信息，如消息类型、消息大小、指向消息数据区指针、

消息队列链接指针等。

• 消息队列头表：其每一表项是作为一个消息队列的消息头，记录消息队列的有关信息，如指向消息队列中第 1 个消息和指向最后一个消息的指针、队列中消息的数目、队列中消息数据的总字节数、队列所允许消息数据的最大字节总数，以及最近一次执行发送操作的进程标识符和时间、最近一次执行接收操作的进程标识符和时间等。

System V 消息队列驻留于主存中，只有在主存重启或显式删除一个消息队列时，该消息队列才会真正被删除。因此，记录消息队列的数据结构(struct ipc_ids msg_ids）位于主存中，系统中的所有消息队列都可以在结构 msg_ids 中找到访问入口。消息队列就是一个消息链表。每个消息队列都有一个队列头，用结构 struct msg_queue 来描述。队列头中包含该消息队列的大量信息，包括消息队列键值、用户 ID、组 ID 和消息队列中的消息数目等，甚至记录最近对消息队列读写进程的 ID。用户既可访问这些信息，也可修改其中某些信息。

图 4-1 说明主存与消息队列之间的联系。其中 struct ipc_ids msg_ids 是主存中记录消息队列的全局数据结构。从图中可以看出，通过全局数据结构 struct ipc_ids msg_ids 可以访问到每个消息队列头的第 1 个成员 struct kern_ipc_perm。而每个 struct kern_ipc_perm 能够与具体的消息队列对应起来，这是因为在该结构中，有一个 key_t 类型成员 key，而 key 可唯一确定一个消息队列。

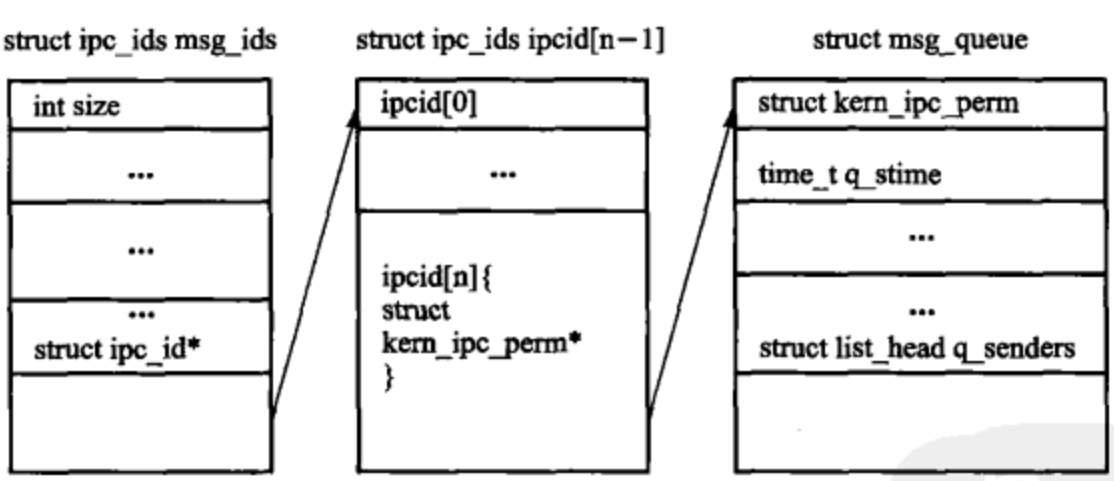


图 4-1 消息队列全局数据结构

图 4-2 描述了消息队列链表结构 struct msg_queue，其中每个元素指向一个能完全描述消息队列的 msgid_ds 数据结构。消息队列允许一个或多个进程向队列中写入消息，然后由一个或多个读进程读出。内核维护一个消息队列表。一旦一个新的消息队列被创建，在系统主存中会为其分配一个新的 msgid_ds 数据结构，并把它插入到数组中。

图 4-3 给出了 msgid_ds 数据结构的示意图，结构定义如下：

```
struct msgid_ds{
    struct ipc_perm msg_perm;
    struct msg *msg_first;        /*队列中的第 1 条消息，即链表头*/
    struct msg *msg_last;         /*队列中的最后一条消息，即链表尾*/
```

```
    time_t msg_stime;        /*发送给队列的最后一条消息的时间*/
    time_t msg_rtime;        /*从队列接收到的最后一条消息的时间*/
    time_t msg_ctime;        /*最后一次修改队列的时间*/
    ushort msg_cbytes;       /*队列中所有消息的总字节数*/
    ushort msg_qnum;         /*当前队列中消息的条数*/
    ushort msg_qbytes;       /*队列的最大字节数*/
    ushort msg_lspid;        /*发送最后一条消息的进程的 PID*/
    ushort msg_lrpid;        /*接收最后一条消息的进程的 PID*/
};
```

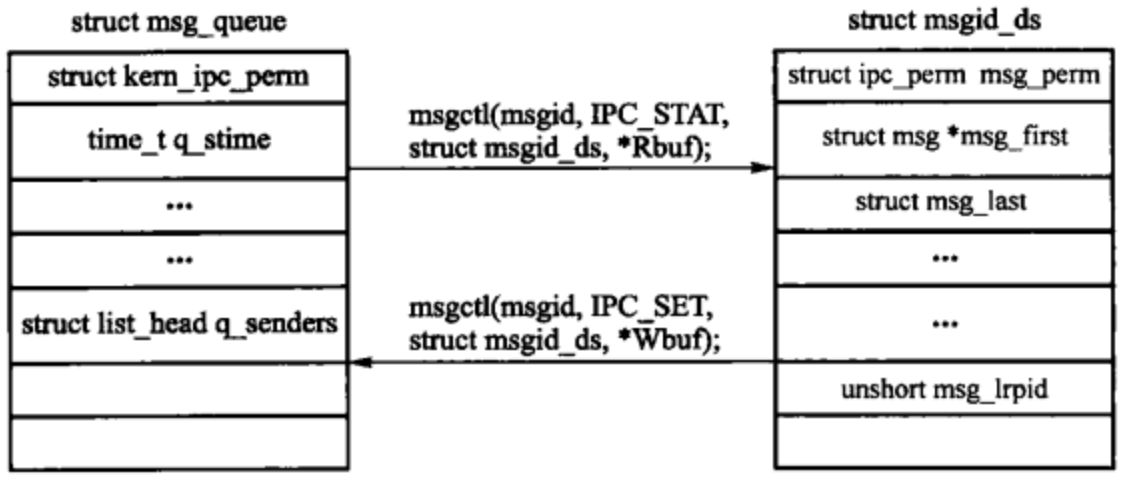


图 4-2　Linux 消息队列链表结构

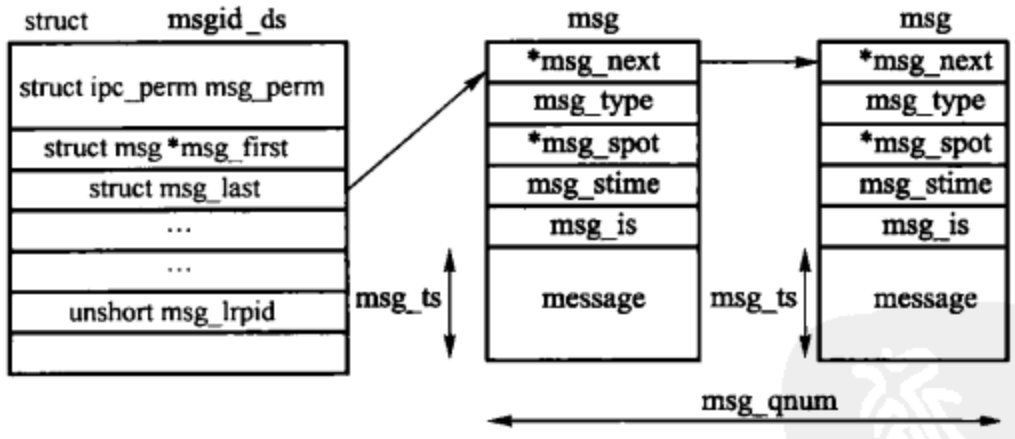


图 4-3　消息队列结构

每个 msgid_ds 结构都包含 ipc_perm 数据结构及指向进入该队列的消息的指针。msgid_ds 结构还包括两个等待队列，一个由队列写入进程使用，而另一个由队列读取进程使用。

每次进程要向写队列写入消息时，系统都要把它的有效用户标识及组标识与该队列的 ipc_perm 数据结构中的访问模式进行比较。如果进程可以写队列，消息会从进程的地址空间复制到一个 msg 数据结构中，然后系统把该 msg 数据结构放在消息队列的尾部。由于内核限制写消息的数量和消息的长度，所以可能会出现没有足够的空间来存放消息的情况。这时，当前进程会被放入对应消息的写等待队列中，函数进程调度程序选择合适的进程运行。在该消息队列

中有一个或多个消息被读出时，睡眠的进程会被唤醒。

从队列中读消息的过程与前面相似，进程对写队列的访问权限会再次被核对。一个读进程可以选择获得队列中的第一个消息而不考虑消息的类型，或者选择读取某种特别类型的消息。如果没有符合要求的消息，读进程会被加入到该消息的读等待队列中，系统唤醒进程调度程序调度新进程运行。一旦有新消息被写入消息队列，睡眠的进程被唤醒，并再次运行。

4.2.2.2 消息队列基本操作

消息队列基本操作包括打开或创建消息队列、对消息队列执行读写操作、获取或设置消息队列属性。与这些操作相关的函数包括 msgget( )、msgsnd( )、msgrcv( )和 msgctl( )等。这些函数包含在 3 个头文件中：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

1. 打开或创建消息队列

msgget( )可以创建一个新消息队列或打开一个存在的消息队列，该函数原型定义如下：

```
int msgget(key_t key, int flag);
```

其中，参数 key 是创建/打开队列的键值，由 ftok( )函数产生，也可以直接用常量指定。flag 参数指定创建/打开方式，可以是 IPC_CREAT、IPC_EXCL、IPC_NOWAIT 或三者的或结果，通常是 flag=IPC_CREAT|IPC_EXCL|0666。意思是，若不存在 key 值的队列，则创建；否则，如果存在则打开队列，0666 表示与一般文件权限一样，指本用户、同组用户、其他用户的读写权限。

在以下两种情况下，该函数将创建一个新的消息队列：

- 如果没有消息队列与键值 key 相对应，并且 flag 中包含 IPC_CREAT 标志位。
- key 参数为 IPC_PRIVATE。

需要注意，IPC_CREAT 位表示创建，一般由服务器程序创建消息队列时使用。如果是客户程序，必须打开现有的消息队列，且不使用 IPC_CREAT。

调用成功返回消息队列描述字，否则返回–1。

2. 读写操作

消息队列的读写操作比较简单，每个消息都有类似如下的数据结构：

```
struct msgbuf {
   long mtype;     /*消息类型*/
   char mtext[ ];   /*消息文本*/
};
```

mtype 成员代表消息类型，从消息队列中读取消息的一个重要依据就是消息类型；mtext 是消息内容，当然长度不一定为 1。因此，对于发送消息来说，首先预置一个 msgbuf 缓冲区并

写入消息类型和内容，调用相应的发送函数。对读取消息来说，首先分配一个 msgbuf 缓冲区，然后把消息读入该缓冲区。

读取消息的函数原型定义如下：

```
ssize_t msgrcv(int msgid, struct msgbuf *msgp, size_t size, long type, int flag);
```

该函数从 msgid 代表的消息队列中读取一条消息，并把消息存储在 msgp 的 msgbuf 结构中。其中，msgid 是返回消息队列的描述符。msgp 是指向用户消息缓冲区的一个结构体指针。size 指示由 msgp 指向的数据结构中字符数组的长度，即消息的长度。这个数组的最大值由 MSG_MAX 来确定。type 是消息类型，当等于 0 时，将读取队列中第 1 个数据。flag 规定倘若该队列无消息，内核应做的操作，即进程是等待还是立即返回。读取标志通常使用 IPC_NOWAIT，若没有满足条件的消息，则立即返回。此时，错误代码 errno=ENOMSG;若在 flag 中设置 MS_NOERROR，且所接收的消息大于 size，则内核截断所接收的消息；IPC_EXCEPT 常与 type>0 配合使用，此时返回队列中第 1 个类型不为 type 的消息。

对于 msgrcv( )，内核必须完成下述工作。

① 对消息队列的描述符和许可权等进行检查。若合法，继续执行；否则返回。

② 根据 type 值的不同分为 3 种情况处理：

- type=0：接收该队列的第 1 条消息，并将它返回给调用者。
- type>0：接收类型 type 的第 1 条消息。
- type<0：接收小于等于 type 绝对值的最低类型的第 1 条消息。

③ 当所返回消息大小等于或小于用户的请求时，内核便将消息正文复制到用户区，并从消息队列中删除此消息，然后唤醒睡眠的发送进程。但如果消息长度比用户要求的大时，则返回出错。

发送消息的函数原型定义如下：

```
int msgsnd(int msgid, struct msgbuf *msgp, size_t size, int flag);
```

参数说明与 msgrcv( )中的对应参数相似。其中，flag 规定当内核用尽内部缓冲空间时应执行的动作，即进程是等待还是立即返回。若在标志 flag 中未设置 IPC_NOWAIT 位，则当该消息队列中的字节数超过最大值时，或系统范围的消息数超过某一最大值时，调用 msgsnd 进程睡眠。若是设置 IPC_NOWAIT，则在此情况下，msgsnd 立即返回。

对于 msgsnd( )函数，内核必须完成以下工作。

① 对消息队列的描述符、许可权及消息长度等进行检查。若合法才继续执行，否则返回。

② 内核为消息分配消息数据区。将用户消息缓冲区中的消息正文复制到消息数据区。

③ 分配消息首部，并将它链入消息队列的末尾。在消息首部中必须填写消息类型、消息大小和指向消息数据区的指针等数据。

④ 修改消息队列头中的数据，如队列中的消息数和字节总数等。最后，唤醒等待消息的进程。

3. 消息队列属性操作

消息队列的信息基本上都保存在消息队列头中，因此，可以分配一个类似于消息队列头的

结构，来返回消息队列的属性。相应地，也可以设置该数据结构。Linux 消息队列链表结构如图 4-3 所示。

消息队列的控制操作指读取消息队列的状态信息并进行修改，如查询消息队列描述符、修改它的许可权及删除该队列等。其函数原型定义如下：

```
int msgctl(int msgid, int cmd, struct msgid_ds *buf);
```

其中，msgid 是打开的消息队列 id。buf 是用户缓冲区地址，供用户存放控制参数和查询结果。cmd 是规定的命令。命令可分为如下 3 类。

① IPC_STAT：查询有关消息队列情况的命令。如查询队列中的消息数目、队列中的最大字节数、最后一个发送消息的进程标识符、发送时间等。

② IPC_SET：设置 buf 指向的结构中的值，设置与消息此队列相关的结构中的下列 4 个字段，即 msg_perm.uid、msg_perm.gid、msg_perm_mode 和 msg_qbytes。此命令只能由两种进程执行，一种是其有效用户 ID 等于 msg_perm.cuid 或 msg_perm.uid；另一种是具有超级用户特权的进程。只有超级用户才能增加 msg_qbytes 的值。

③ IPC_RMID：从系统中删除该消息队列以及仍在该队列上的所有数据。这种删除立即生效。仍在使用这一消息队列的其他进程，在它们下一次试图对此队列进行操作时，将出错返回 EIDRM。此命令只能由两种进程执行，一种是其有效用户 ID 等于 msg_perm.cuid 或 msg_perm.uid；另一种是具有超级用户特权的进程。

4. 消息队列应用

消息队列应用相对较简单，下例(msgtest.c)基本上覆盖了对消息队列的所有操作：

```
/*****msgtest.c*****/
#include <sys/types.h>
#include <sys/msg.h>
#include <unistd.h>
void msg_stat(int,struct msgid_ds );

main( )
{
int gflags,sflags,rflags;
key_t key;
int msgid;
int reval;
struct msgsbuf{
      int mtype;
      char mtext[1];
}msg_sbuf;
struct msgmbuf {
      int mtype;
```

```
    char mtext[10];
}msg_rbuf;
struct msgid_ds msg_ginfo,msg_sinfo;
char* msgpath="/UNIX/msgqueue";
key=ftok(msgpath,'a');
gflags=IPC_CREAT|IPC_EXCL;
msgid=msgget(key,gflags|00666);
if(msgid== -1) {
    printf("msg create error\n");
    return;
}
/*创建一个消息队列后，输出消息队列默认属性*/
msg_stat(msgid,msg_ginfo);
sflags=IPC_NOWAIT;
msg_sbuf.mtype=10;
msg_sbuf.mtext[0]='a';
reval=msgsnd(msgid,&msg_sbuf,sizeof(msg_sbuf.mtext),sflags);
if(reval== -1) {
    printf("message send error\n");
}
/*发送一个消息后，输出消息队列属性*/
msg_stat(msgid,msg_ginfo);
rflags=IPC_NOWAIT|MSG_NOERROR;
reval=msgrcv(msgid,&msg_rbuf,4,10,rflags);
if(reval== -1)
    printf("read msg error\n");
else
    printf("read from msg queue %d bytes\n",reval);
/*从消息队列中读出消息后，输出消息队列属性*/
msg_stat(msgid,msg_ginfo);
msg_sinfo.msg_perm.uid=8;
msg_sinfo.msg_perm.gid=8;
msg_sinfo.msg_qbytes=16388;
/*此处验证超级用户可以更改消息队列的默认 msg_qbytes*/
/*注意这里设置的值大于默认值*/
reval=msgctl(msgid,IPC_SET,&msg_sinfo);
if(reval== -1) {
    printf("msg set info error\n");
    return;
}
```

```
msg_stat(msgid,msg_ginfo);
/*验证设置消息队列属性*/
reval=msgctl(msgid,IPC_RMID,NULL);/*删除消息队列*/
if(reval== -1) {
    printf("unlink msg queue error\n");
    return;
}
}
void msg_stat(int msgid,struct msgid_ds msg_info) {
int reval;
sleep(1);/*只是为了方便后面输出时间*/
reval=msgctl(msgid,IPC_STAT,&msg_info);
if(reval== -1) {
    printf("get msg info error\n");
    return;
}
printf("\n");
printf("current number of bytes on queue is %d\n",msg_info.msg_cbytes);
printf("number of messages in queue is %d\n",msg_info.msg_qnum);
printf("max number of bytes on queue is %d\n",msg_info.msg_qbytes);
    /*每个消息队列的容量(字节数）都有限制 MSGMNB，值的大小因系统而异*/
    /*在创建新的消息队列时，msg_qbytes 的默认值就是 MSGMNB*/
    printf("pid of last msgsnd is %d\n",msg_info.msg_lspid);
    printf("pid of last msgrcv is %d\n",msg_info.msg_lrpid);
    printf("last msgsnd time is %s", ctime(&(msg_info.msg_stime)));
    printf("last msgrcv time is %s", ctime(&(msg_info.msg_rtime)));
    printf("last change time is %s", ctime(&(msg_info.msg_ctime)));
    printf("msg uid is %d\n",msg_info.msg_perm.uid);
    printf("msg gid is %d\n",msg_info.msg_perm.gid);
    }
```

程序输出结果如下：

```
current number of bytes on queue is 0
number of messages in queue is 0
max number of bytes on queue is 16384
pid of last msgsnd is 0
pid of last msgrcv is 0
last msgsnd time is Thu Jan    1 08:00:00 1970
last msgrcv time is Thu Jan    1 08:00:00 1970
last change time is Sun Dec 29 18:28:20 2002
msg uid is 0
```

```
msg gid is 0
/*上面刚刚创建一个新消息队列时的输出*/
current number of bytes on queue is 1
number of messages in queue is 1
max number of bytes on queue is 16384
pid of last msgsnd is 2510
pid of last msgrcv is 0
last msgsnd time is Sun Dec 29 18:28:21 2002
last msgrcv time is Thu Jan  1 08:00:00 1970
last change time is Sun Dec 29 18:28:20 2002
msg uid is 0
msg gid is 0
read from msg queue 1 bytes     /*实际读出的字节数*/
current number of bytes on queue is 0
number of messages in queue is 0
max number of bytes on queue is 16384     /*每个消息队列最大容量(字节数)*/
pid of last msgsnd is 2510
pid of last msgrcv is 2510
last msgsnd time is Sun Dec 29 18:28:21 2002
last msgrcv time is Sun Dec 29 18:28:22 2002
last change time is Sun Dec 29 18:28:20 2002
msg uid is 0
msg gid is 0
current number of bytes on queue is 0
number of messages in queue is 0
max number of bytes on queue is 16388 /*可看出超级用户可修改消息队列最大容量*/
pid of last msgsnd is 2510
pid of last msgrcv is 2510                /*对操作消息队列进程的跟踪*/
last msgsnd time is Sun Dec 29 18:28:21 2002
last msgrcv time is Sun Dec 29 18:28:22 2002
last change time is Sun Dec 29 18:28:23 2002 /*msgctl( )调用对 msg_ctime 有影响*/
msg uid is 8
msg gid is 8
```

4.2.3 信号量

4.2.3.1 信号量的基本概念

信号量与其他进程通信方式大不相同，它是一种进程之间的访问控制机制，相当于标志变量。进程可以根据它判定是否能够访问某些共享资源或协同工作，同时也允许进程修改标志变

量。信号量实际是一个整型数，其值由多个进程进行测试和设置。进程执行的测试和设置操作是不可中断的，称为“原语”操作，即一旦开始，便确保两个操作全部完成。测试和设置操作的结果是：信号量的当前值和设置值相加，其和或者为正或者为负。根据测试和设置操作的结果，一个进程可能处于睡眠状态，直到有另一个进程改变信号量值。信号量有以下两种类型。

- 二值信号量：最简单的信号量形式，信号量的值只能取 0 或 1。请注意，二值信号量能够实现互斥锁的功能，但两者的关注内容不同。信号量强调共享资源，只要有共享资源可用，其他进程同样可以修改信号量值；互斥锁强调相关进程，占用资源的进程使用完资源后，必须由进程本身来解锁。
- 一般信号量：信号量的值可以取任意非负整型值。

System V 的信号量存储在主存中，只有在主存重启或者显式删除一个信号量集时，该信号量集才会真正被删除。因此，系统中记录信号量的数据结构(struct ipc_ids sem_ids)位于主存中，系统中的所有信号量都可以在结构 sem_ids 中找到访问入口。

图 4-4 说明主存中信号量的关联关系。struct ipc_ids sem_ids 是主存中记录信号量的全局数据结构，描述一个具体的信号量及其相关信息。从图中可以看出，全局数据结构 struct ipc_ids sem_ids 可以访问到 struct kern_ipc_perm 的第 1 个成员 struct kern_ipc_perm。而每个 struct kern_ipc_perm 能够与具体的信号量对应起来，这是因为 struct kern_ipc_perm 中有一个 key_t 类型成员 key，该 key 唯一确定一个信号量集。与此同时，结构 struct kern_ipc_perm 的最后一个成员 sem_nsems 确定该信号量在信号量集中的顺序，这样主存可以记录每个信号量的信息。

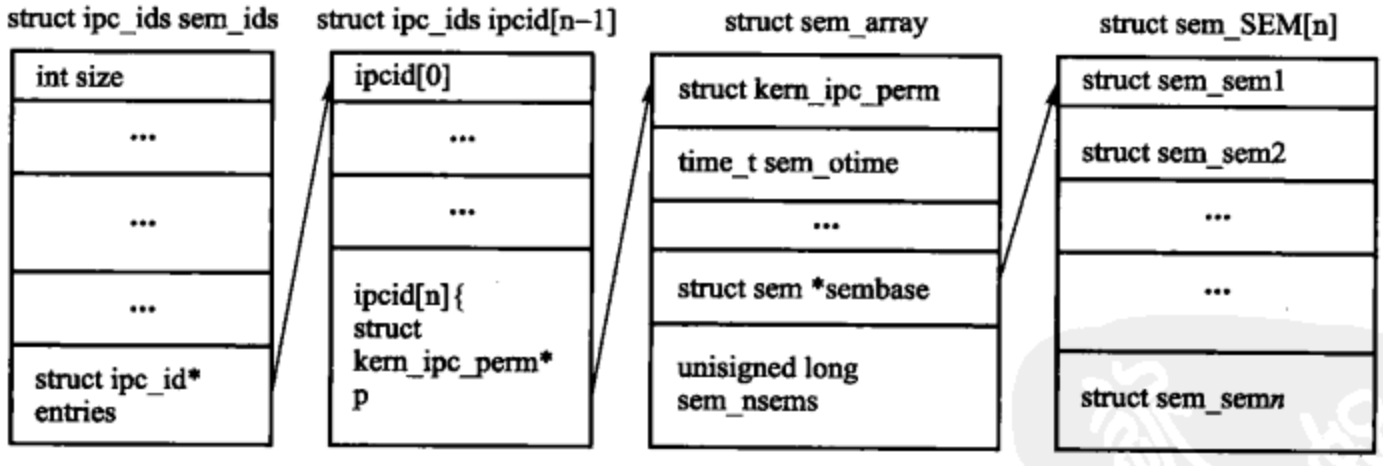


图 4-4 System V 信号量数据结构

从图 4-4 可以看出，System V 的每个 IPC 信号量对象都对应一个 struct sem_array 结构，该结构是信号量集合，如图 4-5 所示用 semid_ds 数据结构来表示，semid_ds 结构定义如下：

```
struct semid_ds {
  struct ipc_perm sem_perm;          /*IPC 权限*/
  long sem_otime;                    /*最后一次对信号量操作(semop)的时间 */
  long sem_ctime;                    /*最后一次修改该结构的时间*/
  struct sem *sem_base;              /*在信号量数组中指向第 1 个信号量的指针 */
```

```
struct sem_queue *sem_pending;          /*待处理的挂起操作*/
 struct sem_queue **sem_pending_last;   /*最后一个挂起操作 */
 struct sem_undo *undo;                 /*在这个数组上的 undo 请求 */
 ushort sem_nsems;                      /*在信号量数组上的信号量号 */
};
```

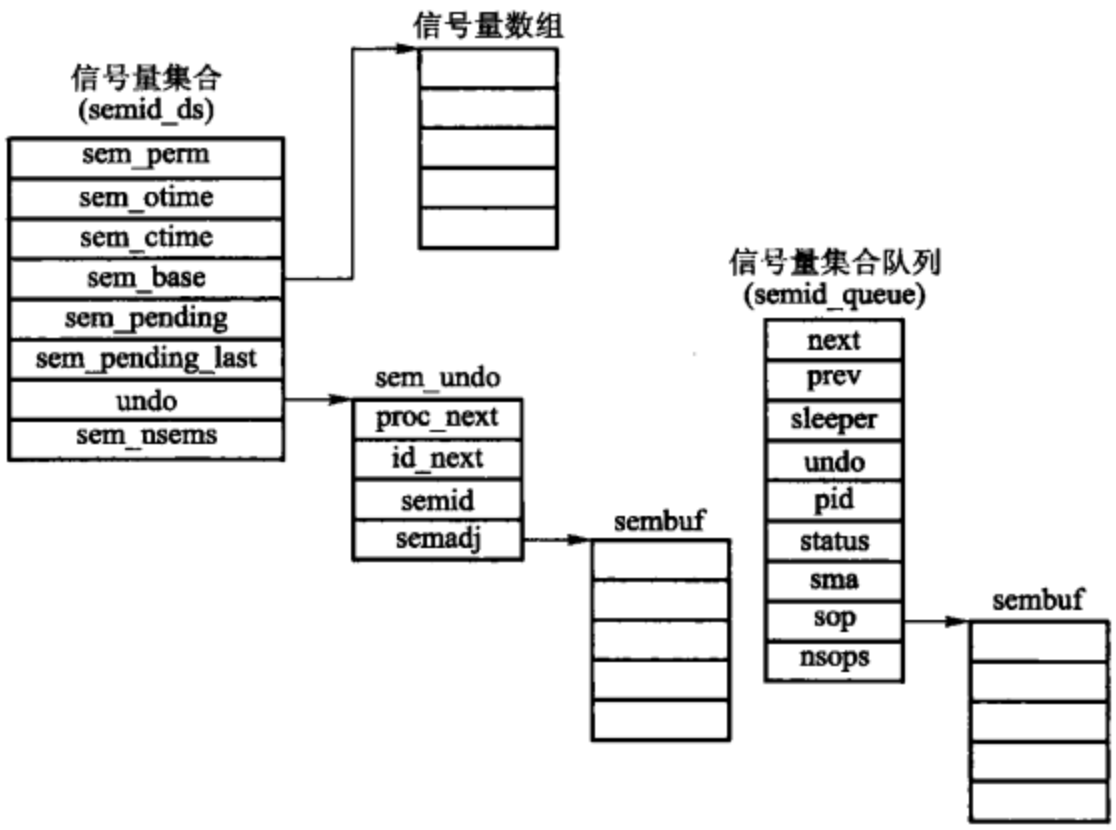


图 4-5　Linux 中的信号量集合结构

其中系统中每一信号量集队列(sem_queue)的结构定义如下：

```
struct sem_queue {
 struct sem_queue *next;          /*  队列中下一个节点 */
 struct sem_queue **prev;         /*  队列中前一个节点, *(q->prev) == q */
 struct wait_queue *sleeper;      /*  正在睡眠的进程 */
 struct sem_undo *undo;           /* undo 结构*/
 int pid;                         /*  请求进程的进程识别号 */
 int status;                      /*  操作的完成状态 */
 struct semid_ds *sma;            /*有操作的信号量集合数组 */
 struct sembuf *sops;             /*  挂起操作的数组 */
 int nsops;                       /*  操作的个数 */
};
```

semid_ds 结构有一个 sem_nsems 域，该域由 sem_base 指向的 sem 数据结构来描述，给出了一个信号量数组，其定义如下：

```
struct sem {
```

```
    ushort semval;
    pid_t sempid;
    ushort semncnt;
    ushort semzcnt;
};
```

请注意信号量与信号量集合的区别，从上面可以看出，信号量用“sem”结构描述，而信号量集合用“semid_ds”结构描述。

struct sem 的数据成员分别描述一个信号量相关的不同属性。semval 是信号量值，不能为负数。simpid 是最后使用该信号量的进程号。semncnt 是系统中等待信号量值大于当前值的进程的个数。semzcnt 是等待信号量值等于 0 的进程个数。允许操作这些信号量集合的进程可以利用函数执行操作。

4.2.3.2 信号量的工作机制

所有允许对 System V IPC 信号量数组中的信号量对象进行操作的进程，都必须通过函数来执行这些操作。每个操作包含 3 个输入项：信号量索引、操作值和一组标志位。信号量索引是对信号量数组的索引值，而操作值是加到当前信号量值上的数值。首先内核会测试是否所有的操作都会成功(操作成功指操作值加上信号量当前值的结果大于 0，或者操作值和信号量的当前值都是 0)。如果信号量操作中有任何一个操作失败，内核在操作标志没有指明函数为非阻塞状态时，会挂起当前进程。如果进程被挂起，系统会保存要执行的信号量操作的状态，并把当前进程放入等待队列中。内核通过在栈中建立一个 sem_queue 数据结构，并填入相应的信息的方法来实现前面的保存信号量操作状态。新的 sem_queue 数据结构被放在对应信号量对象的等待队列的末尾，当前进程被放在 sem_queue 数据结构的等待队列中，然后系统唤醒进程调度程序选择其他进程执行，原进程挂起。

如果所有信号量操作都成功，当前进程就会继续运行，对信号量数组中的对应成员执行相应操作。接着内核会查看那些处于等待队列中被挂起的进程，测试它们现在能否成功地执行信号量操作。如果有进程可以成功执行，则会删除未完成操作列表中对应的 sem_queue 数据结构，对信号量数组执行信号量操作，然后唤醒睡眠进程，将其放入就绪队列中。内核不断地查找等待队列成员，直到没有可成功执行的信号量操作并且也没有可唤醒的进程为止。

为避免发生死锁，内核通过为信号量数组维护一个调整项列表来防止死锁。其基本思想是：在使用调整项后，信号量会被恢复到一个进程的信号量操作集合执行前的状态。每一个单独的信号量操作都要求建立相应的调整项。内核为每个进程的每个信号量数组至多维护一个 sem_undo 数据结构，该结构在 include/Linux/sem.h 中定义，具体描述如下：

```
struct sem_undo {
  struct sem_undo *proc_next;      /*在该进程上的下一个 sem_undo 节点*/
  struct sem_undo *id_next;        /*在该信号量集上的下一个 sem_undo 节点*/
  int   semid;                     /*信号量集的标识号*/
```

```
    short *semadj;                    /*信号量数组的调整，每个进程一个*/
};
```

sem_undo 数据结构被加入到该进程的 task_struct 数据结构和信号量数组的 semid_ds 数据结构队列中。一旦对信号量数组中某些信号量执行相应操作，那么该操作数的负值会被加入到该进程 sem_undo 结构中与该信号量对应的记录项中。当进程被删除、退出系统时，会用这些 sem_undo 数据结构集合对信号量数组进行调整。如果信号量集合被删除，这些 sem_undo 数据结构还存在于进程的 task_struct 结构的队列中，而仅把信号量数组标识标记为无效。在这种情况下，信号量清理程序仅仅丢掉这些数据结构而不释放它们所占用的空间。

4.2.3.3 信号量基本操作

信号量的基本操作包括打开或创建信号量、信号量值操作、获取或设置信号量属性。与这些操作相关的函数包括 semget( )、semop( )和 semctl( )等，它们被包含在如下 3 个头文件中。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

1. 打开或创建信号量

semget( )可以创建一个新信号量或打开一个现有的信号量。该函数原型定义如下：

```
int semget(key_t key, int nsems, int flag);
```

其中参数 key 和 flag 的取值，以及何时打开已有信号量集或者创建一个新的信号量集，与 msgget( )中的对应部分相同。参数 nsems 指定打开或者新创建的信号量集中将包含信号量的数目，其最大值在“Linux/sem.h”中定义：

```
#define SEMMSL   250
```

注意，如果显式地打开一个现有的信号量集合，则 nsems 参数可以忽略。

调用成功返回信号量描述字，否则返回–1。

2. 信号量值操作

进程可以增加或减小信号量值，相应于对共享资源的释放和占有，操作函数的原型定义如下：

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

其中参数 semid 是信号量集 ID，参数 sops 指向类型为 sembuf 的一个数组，nsops 为 sops 指向数组的大小。

调用成功返回 0，否则返回–1。

sops 在/inclide/linux/sem.h 中声明，是主存中的一个数据结构，描述如下：

```
struct sembuf {
  ushort   sem_num;        /*在数组中信号量的索引值*/
  short    sem_op;         /*信号量操作值(正数、负数或 0) */
  short    sem_flg;        /*操作标志，为 IPC_NOWAIT 或 SEM_UNDO*/
```

};

如果 sem_op 为负数，就从信号量的值中减去 sem_op 的绝对值，这意味着进程要获取资源。这些资源是由信号量控制来存取的。如果没有指定 IPC_NOWAIT，那么调用进程睡眠到请求的资源数得到满足(其他的进程可能释放一些资源)。如果 sem_op 是正数，把它的值加到信号量，这意味着归还资源。如果 sem_op 为 0，那么调用进程将睡眠到信号量的值也为 0，这相当于一个信号量达到 100%的利用。

sem_flg 可取 IPC_NOWAIT 及 SEM_UNDO 两个标志。如果设置 SEM_UNDO 标志，在进程结束时，相应的操作将被取消，这是比较重要的一个标志位。如果设置该标志位，那么在进程没有释放共享资源就退出时，主存将代为释放。如果为一个信号量设置该标志，主存都要分配一个 sem_undo 结构来记录它，为的是确保以后资源能够安全释放。事实上，如果进程退出，那么它所占用的资源就会释放，但信号量值却没有改变。此时，信号量值反映的已经不是资源占有的实际情况。这有点像僵死进程，进程虽然退出，资源也都释放，但主存进程表中仍然有它的记录，此时需要父进程调用 waitpid( )函数来解决问题。

根据 sem_op 的值大于 0、等于 0 及小于 0，确定对 sem_num 指定的信号量进行的 3 种操作。这里需要强调的是，sem_op 同时操作多个信号量，在实际应用中，对应多种资源的同时申请或释放。因此，保证 sem_op 操作的原子性尤为重要。尤其对于多种资源的申请来说，要么一次性获得所有资源，要么放弃申请，要么在不占有任何资源情况下继续等待。这样，一方面避免资源浪费；另一方面避免进程之间由于申请共享资源造成死锁。也许从实际含义上更好理解这些操作。信号量的当前值记录相应资源目前可用数目，sem_op>0 对应相应进程要释放 sem_op 数目的共享资源；sem_op=0 可以用于对共享资源是否已用完的测试；sem_op<0 相当于进程要申请–sem_op 个共享资源。

综上所述，内核按如下的规则判断是否所有的操作都可以成功：操作值和信号量的当前值相加大于 0，或操作值和当前值均为 0，则操作成功。如果函数中指定的所有操作中有一个操作不能成功时，则内核会挂起该进程。但是，如果操作标志指定这种情况下不能挂起进程，函数返回并指明信号量上的操作没有成功，而进程可以继续执行。如果进程被挂起，必须保存信号量的操作状态并将当前进程放入等待队列。为此，内核在堆栈中建立一个 sem_queue 结构并填充该结构。新的 sem_queue 结构添加到集合的等待队列中(利用 sem_pending 和 sem_pending_last 指针)。当前进程放入 sem_queue 结构的等待队列中(sleeper)后，调用进程调度程序选择其他进程运行。

为了进一步解释 semop( )，先来看一个例子。假设系统有一台打印机，一次只能打印一个作业，创建一个只有一个信号量的集合(仅一个打印机)，并且给信号量的初值为 1(因为一次只能有一个作业)。每当希望把一个作业发送给打印机时，首先要确定这个资源是否可用，可通过从信号量中获得一个单位而达到此目的。为此，需加载一个 sembuf 数组来执行这个操作：

```
struct sembuf sem_lock = { 0,–1, IPC_NOWAIT };
```

从这个初始化结构可以看出，0 表示集合中信号量数组的索引，即在集合中只有一个信号

量，–1 表示信号量操作(sem_op)，操作标志为 IPC_NOWAIT，表示或者调用进程不用等待可立即执行，或者失败(另一个进程正在打印)。下面是以 sembuf 结构执行 semop( )函数的例子：

```
if(semop(sid, &sem_lock, 1) == -1)
  fprintf(stderr,"semop\n");
```

参数 nsops 表明仅仅执行一个操作(在操作数组中只有一个 sembuf 结构)，sid 参数是集合的 IPC 识别号。使用完打印机，必须把资源返回给集合，以便其他的进程使用。

```
struct sembuf sem_unlock = { 0, 1, IPC_NOWAIT };
```

上面这个初始化结构表示，把 1 加到集合数组的第 0 个元素，换句话说，一个单位资源返回给集合。

3. 信号量属性操作

系统中的每一个信号量集都对应一个 struct sem_array 结构，该结构记录信号量集的各种信息，存在于系统空间。为了设置、获得该信号量集的各种信息及属性，在用户空间有一个重要的联合结构与之对应，即 union semun。这个特殊的联合结构在 include/linux/sem.h 中声明，其结构如图 4-6 所示，对它的描述如下：

```
union semun {
  int val;                  /* SETVAL 的值*/
  struct semid_ds *buf;     /* IPC_STAT & IPC_SET 的缓冲*/
  ushort *array;            /* GETALL & SETALL 数组*/
  struct seminfo *__buf;    /* IPC_INFO 的缓冲*/
  void *__pad;
};
```

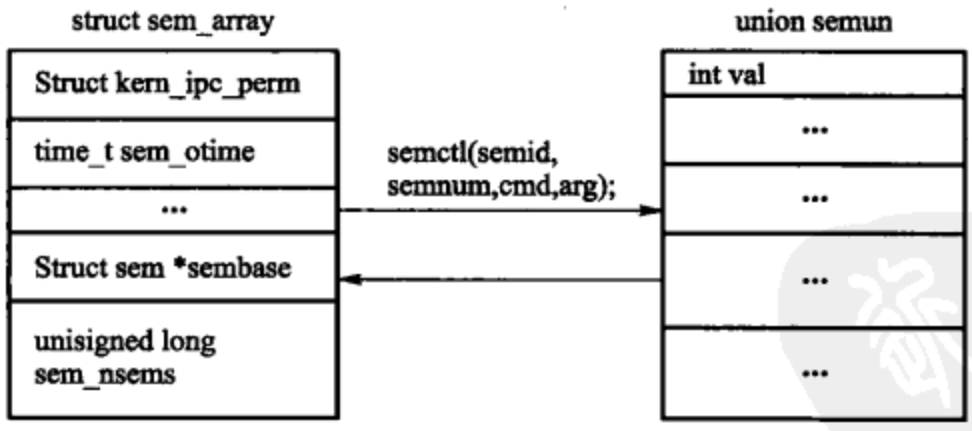


图 4-6　union semun 结构

信号量属性操作的函数原型定义如下：

```
int semctl(int semid, int semnum, int cmd, union semun arg );
```

该函数实现对信号量的各种控制操作。其中参数 semid 指定信号量集。参数 semnum 是将要操作的信号量个数，从本质上说，它是集合的一个索引，对于集合上的第 1 个信号量，则该值为 0。参数 arg 用于设置或返回信号量信息，类型为 semun。参数 cmd 指定具体的操作类型，

可选值见表 4-2。

表 4-2 cmd 命令及解释

命　令	解　释
IPC_STAT	获取信号量信息，信息由 arg.buf 返回
IPC_SET	设置信号量信息，待设置信息保存在 arg.buf 中(在 manpage 中给出了可以设置哪些信息)
IPC_RMID	从主存中删除信号量集合
GETALL	返回所有信号量的值，结果保存在 arg.array 中，忽略参数 sennum
GETNCNT	返回等待 semnum 所代表信号量的值增加的进程数，相当于目前有多少进程在等待 semnum 代表的信号量所代表的共享资源
GETPID	返回最后一个对 semnum 所代表信号量执行 semop 操作的进程 ID
GETVAL	返回 semnum 所代表信号量的值
GETZCNT	返回等待 semnum 所代表信号量的值变成 0 的进程数
SETALL	通过 arg.array 更新所有信号量的值；同时，更新与本信号集相关的 semid_ds 结构的 sem_ctime 成员
SETVAL	设置 semnum 所代表信号量的值为 arg.val

函数调用成功的返回结果与 cmd 的值相关，调用失败返回-1。

4.2.3.4 信号量应用

信号量机制能够有效地控制多个进程对资源的访问，实现进程同步。通常所说的 System V 信号量实际上是一个信号量集合，可用于多种共享资源的进程同步，每个信号量都有一个值，可以用来表示当前该信号量代表的共享资源可用数量，如果一个进程要申请共享资源，那么就从信号量值中减去要申请的数目，如果当前没有足够的可用资源，进程可以选择睡眠等待，也可以立即返回。

以下代码段(semtest.c)展示如何获取各种信号量信息及利用信号量实现共享资源的申请和释放。

```
/*****semtest.c*****/
#include <linux/sem.h>
#include <stdio.h>
#include <errno.h>
#define SEM_PATH "/UNIX/my_sem"
#define max_tries 3
int semid;

main( )
{
   int flag1,flag2,key,i,init_ok,tmperrno;
```

```
struct semid_ds sem_info;
struct seminfo sem_info2;
union semun arg;
struct sembuf askfor_res, free_res;
flag1=IPC_CREAT|IPC_EXCL|00666;
flag2=IPC_CREAT|00666;
key=ftok(SEM_PATH,'a');
/* ftok( )出错处理*/
init_ok=0;
semid=semget(key,1,flag1);      /*创建一个仅包含一个信号量的信号量集*/
if(semid<0) {
   tmperrno=errno;
   perror("semget");
   if(tmperrno==EEXIST) {
      semid=semget(key,1,flag2);
      /*flag2 只包含 IPC_CREAT 标志，参数 nsems(这里为 1)必须与原来的信号量数目一致*/
      arg.buf=&sem_info;
      for(i=0; i<max_tries; i++) {
           if(semctl(semid, 0, IPC_STAT, arg)== -1)
             {perror("semctl error"); i=max_tries;}
           else {
             if(arg.buf->sem_otime!=0){ i=max_tries;   init_ok=1;}
             else sleep(1);
            }
      }
      if(!init_ok) {
      /*部分初始化，这里假定建立 sem 的第 1 个进程将完成初始化 sem 的工作，并且在 max_tries
      时间内运行 semop*/
           arg.val=1;
           if(semctl(semid,0,SETVAL,arg)== -1) {perror("semctl setval error");
      }
      } else {
      perror("semget error, process exit");  exit( );}
} else {                  /*semid>=0; 初始化*/
   arg.val=1;
   if(semctl(semid,0,SETVAL,arg)== -1)
     perror("semctl setval error");
}
   /*获取部分信号量信息*/
   arg.buf=&sem_info;
```

```
if(semctl(semid, 0, IPC_STAT, arg)== -1)
   perror("semctl IPC STAT");
printf("owner's uid is %d\n",        arg.buf->sem_perm.uid);
printf("owner's gid is %d\n",        arg.buf->sem_perm.gid);
printf("creater's uid is %d\n",      arg.buf->sem_perm.cuid);
printf("creater's gid is %d\n",      arg.buf->sem_perm.cgid);
arg.__buf=&sem_info2;
if(semctl(semid,0,IPC_INFO,arg)== -1)
   perror("semctl IPC_INFO");
printf("the number of entries in semaphore map is %d \n",  arg.__buf->semmap);
printf("max number of semaphore identifiers is %d \n", arg.__buf->semmni);
printf("mas number of semaphores in system is %d \n", arg.__buf->semmns);
printf("the number of undo structures system wide is %d \n", arg.__buf->semmnu);
printf("max number of semaphores per semid is %d \n", arg.__buf->semmsl);
printf("max number of ops per semop call is %d \n", arg.__buf->semopm);
printf("max number of undo entries per process is %d \n", arg.__buf->semume);
printf("the sizeof of struct sem_undo is %d \n", arg.__buf->semusz);
printf("the maximum semaphore value is %d \n", arg.__buf->semvmx);

/*now ask for available resource: */
            askfor_res.sem_num=0;
            askfor_res.sem_op= -1;
            askfor_res.sem_flg=SEM_UNDO;
            if(semop(semid,&askfor_res,1)== -1)/*ask for resource*/
               perror("semop error");
            sleep(3); /*等待 3s，以执行共享资源处理*/
            printf("now free the resource\n");
            /*释放资源*/
            free_res.sem_num=0;
            free_res.sem_op=1;
            free_res.sem_flg=SEM_UNDO;
            if(semop(semid,&free_res,1)== -1)
                if(errno==EIDRM)
                printf("the semaphore set was removed\n");
            if(semctl(semid, 0, IPC_RMID)== -1)
                perror("semctl IPC_RMID");
            else printf("remove sem ok\n");
    }
```

编译并运行该程序，运行结果如下：

```
owner's uid is 0
```

```
owner's gid is 0
creater's uid is 0
creater's gid is 0
the number of entries in semaphore map is 32000
max number of semaphore identifiers is 128
mas number of semaphores in system is 32000
the number of undo structures system wide is 32000
max number of semaphores per semid is 250
max number of ops per semop call is 32
max number of undo entries per process is 32
the sizeof of struct sem_undo is 20
the maximum semaphore value is 32767
now free the resource
remove sem ok
```

4.2.4 共享主存

4.2.4.1 System V 共享主存的原理

由于进程的虚拟地址可以映射到任意一处物理地址，因此，如果两个进程的虚拟地址映射到相同物理地址上，则这两个进程就可以利用该物理单元进行通信。System V 共享主存将需要共享的数据放在 IPC 共享主存区(IPC shared memory region)，所有需要访问该共享区的进程都要把该共享区映射到本进程的虚拟地址空间。共享主存本质上是从主存中获取一个主存块，让使用进程保存一个指向该主存块的指针，各使用进程对它的读写操作与使用普通主存指针一样方便和高效。若两个不同进程 A、B 共享主存，则同一块物理主存被映射到进程 A、B 各自的进程虚拟地址空间。进程 A 可以即时看到进程 B 对共享主存中数据的更新，反之亦然。由于多个进程共享同一块主存区，对共享主存的访问必然要使用某种同步机制，如互斥锁和信号量。Linux 中的共享主存通过访问键来访问，并且进行访问权限的检查。共享主存对象的创建者负责控制访问权限，以及访问键的公有或私有特性。如果具有足够权限，也可以将共享主存锁定到物理主存中。一旦某一主存区被共享，系统就无法检查进程如何使用这部分主存区域。

共享主存是一种简单而高效的进程间通信方法。对于像管道和消息队列等通信方式，需要在主存和用户空间进行 4 次数据复制。而共享主存只需要复制两次数据，一次从输入文件到共享主存区，另一次从共享主存区到输出文件。使用共享主存的进程可以直接读写主存，而不需要任何数据复制。实际上，进程之间在共享主存时，并不总是读写少量数据后就解除映射，当有新的进程通信时，再重新建立共享主存区；而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享主存中，并没有写回磁盘文件，共享主存中的内容往往是在解除映射时才写回文件。

System V 通过映射特殊存储块 shm 中的文件实现进程间的共享主存通信，亦即每个共享主存区对应特殊存储块 shm 中的一个文件(通过 shmid_kernel 结构关联)。System V 共享主存通过 shmget( )获得(或创建)一个 IPC 共享主存区，并返回相应的标识符。主存在保证 shmget( )获得(或创建)一个共享主存区并初始化相应 shmid_kernel 结构的同时，还将在特殊存储块 shm 中创建并打开一个同名文件，并在主存中建立与该文件对应的 dentry 及 inode 结构。需要注意，新打开的文件不属于任何一个进程，任何进程都可以访问该共享主存区，所有这一切都是函数 shmget( )完成的。

每个共享主存区都有一个控制结构 struct shmid_kernel，它是共享主存区中重要的数据结构，是让存储管理和文件系统结合起来的桥梁，其定义如下：

```
struct shmid_kernel {
  struct kern_ipc_perm   shm_perm;
  struct file *shm_file;
  int id;
  unsigned long shm_nattch;
  unsigned long shm_segsz;        /*共享主存的大小(bytes） */
  time_t shm_atim;                /*最后一次 attach 此共享主存的时间*/
  time_t shm_dtim;                /*最后一次 detach 此共享主存的时间*/
  time_t shm_ctim;                /*最后一次更改此共享主存结构的时间*/
  pid_t shm_cpid;                 /*建立此共享主存的进程识别码*/
  pid_t shm_lpid;                 /*最后一个操作此共享主存的进程识别码*/
};
```

shm_file 是该结构中最重要的一个域，它存储了将被映射的文件的地址。每个共享主存区对象都对应特殊存储块 shm 中的一个文件，一般情况下，不能使用 read( )、write( )等函数访问 shm 中的文件。当采取共享主存的方式把其中的文件映射到进程地址空间后，可直接采用访问主存的方式对其访问。

图 4-7 描述 System V 共享主存中的相关数据结构。与消息队列和信号量一样，主存通过数据结构 struct ipc_ids shm_ids 维护系统中的所有共享主存区。图中的 shm_ids.entries 变量指向一个 ipc_id 结构数组，而每个 ipc_id 结构数组中有个指向 kern_ipc_perm 结构的指针。对于 System V 共享主存区来说，kern_ipc_perm 的宿主是 shmid_kernel 结构。shmid_kernel 用来描述一个共享主存区，于是主存就能够控制系统中所有的共享区。同时，在 shmid_kernel 结构的 file 类型指针 shm_file 指向文件系统 shm 中相应的文件。这样，共享主存区就与 shm 中的文件对应起来。

图 4-8 给出共享主存对象的结构。每个新创建的共享主存区由 shmid_ds 数据结构来表示，并被记录在 shm_segsz 变量中。shmid_ds 数据结构中包含共享存储区的大小、当前使用该共享存储区的进程数目以及共享存储区如何映射到进程地址空间等信息。下面介绍 shmid_ds 结构的组成元素。

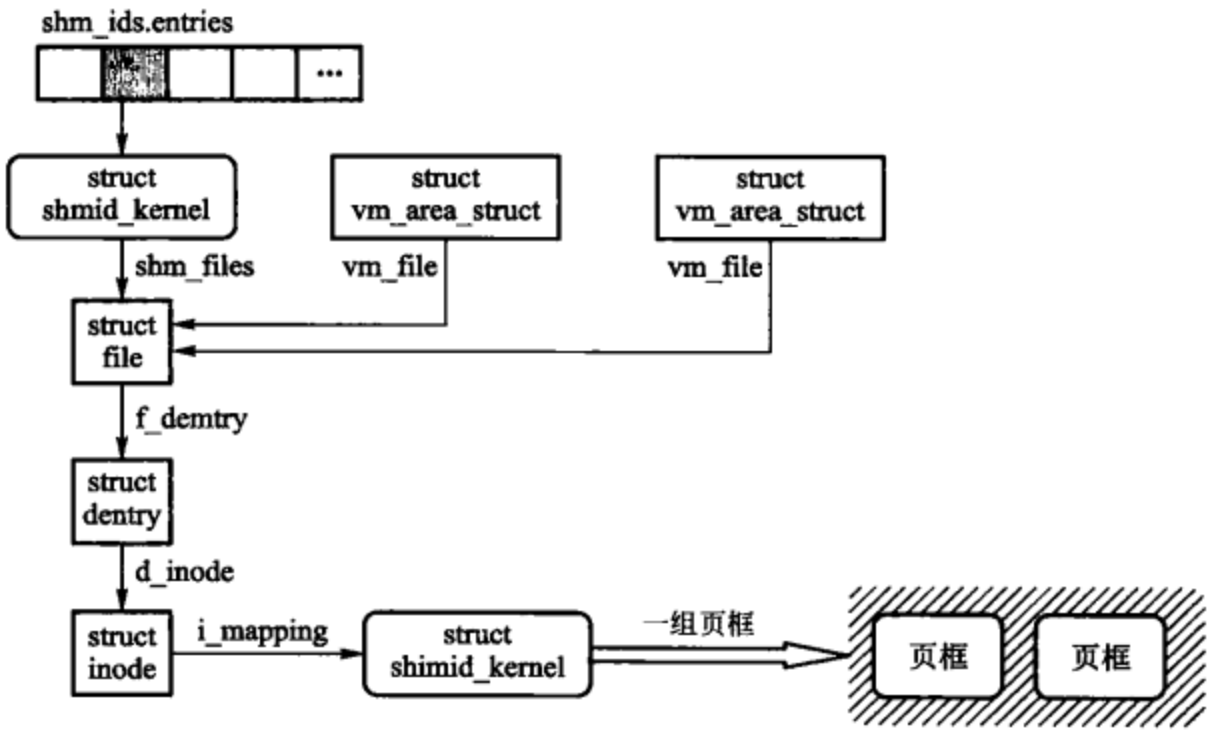


图 4-7　System V 共享主存中的相关数据结构

- shm_segsz：共享主存的大小。
- times：使用共享主存的进程数目。
- attaches：描述被共享的物理主存映射到各进程的虚拟主存区。
- shm_npages：共享虚拟主存页的数目。
- shm_pages：指向共享虚拟主存页的页表项。

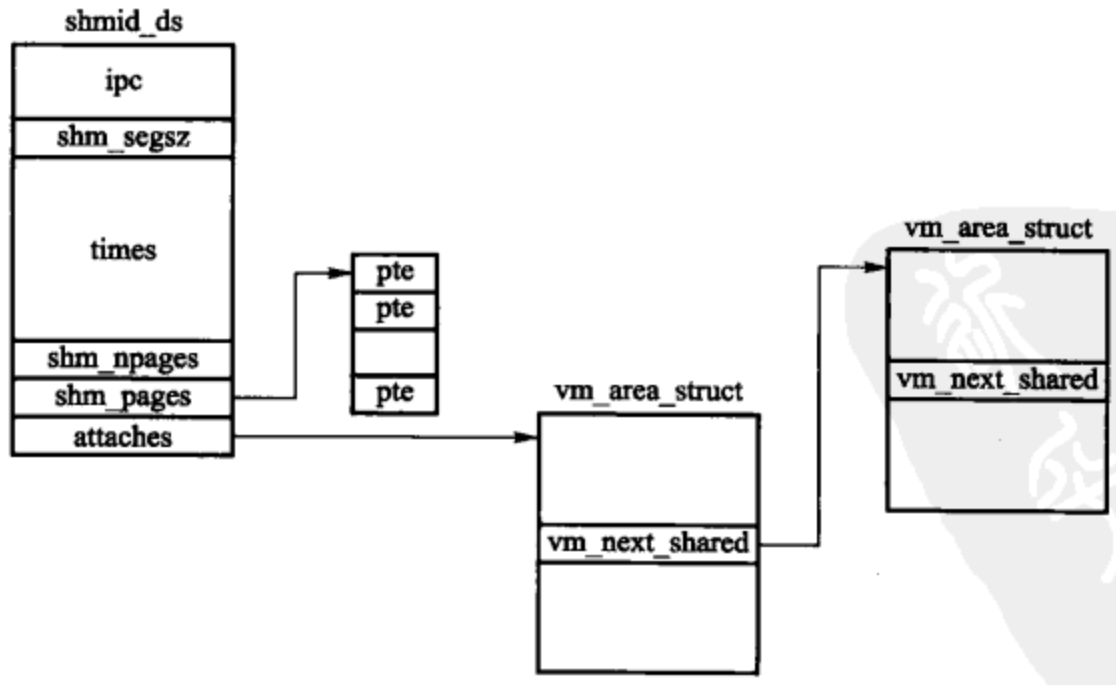


图 4-8　System V IPC 共享主存对象结构

创建一个共享主存区后，还要将它映射到进程虚拟地址空间，函数 shmat( )可完成此项任

务。由于在调用 shmget( )时，已经创建 shm 中的一个同名文件与共享主存区相对应，因此，调用 shmat( )的过程相当于映射 shm 中的同名文件过程。

每个想访问共享主存区的进程必须先通过函数，将该共享主存区连接到其虚地址空间中。该操作会创建一个描述该进程共享主存区的 vm_area_struct 数据结构。进程既可以指定共享主存区放在其虚地址空间的位置，也可以由内核自动选择一个足够大的自由空间。新的 vm_area_struct 数据结构被放入由 shmid_ds 指向的 vm_area_struct 结构的双向链表中。这个双向链表由 vm_area_struct 结构中的 vm_next_shared 指针和 vm_prev_shared 指针链接在一起。在执行连接操作时，系统实际上还没有创建该共享主存区，只有在第 1 个进程要访问共享主存区时，系统才会执行实际的创建工作。

当某一进程第 1 次访问共享主存区的某一页时，系统会产生一个页面失效，内核在处理页失效时，会找到描述该页的 vm_area_struct 数据结构。在 vm_area_struct 结构中包含处理这种共享主存区页面失效的例程的句柄。共享主存储区页面失效处理例程会为 shmid_ds 结构查找页表项的列表，以确定共享主存区中的这个页是否存在。如果不存在，内核分配一个物理页框，并为该页创建页表项。这个新的页表项会被同时保存到当前进程的页表和 shmid_ds 结构中。这种处理方法使下一个进程访问该页面时产生页面失效，共享主存区页失效处理例程会再次使用被分配的物理页。因此，第一个访问共享主存区某个页的进程会导致系统创建该共享页，而其他访问该共享页的进程仅仅会把该页增加到它们的虚地址空间中。

当进程不再使用共享主存区时，需执行分离操作。只要还有其他进程仍在使用这块存储区，分离操作就只会影响当前进程。进程的 vm_area_struct 结构会被从 shmid_ds 结构中删除并释放掉，系统更新当前进程的页表以使原来被共享的虚地址区无效。在最后一个使用共享主存区的进程执行分离操作时，处在物理存储器中的共享页面才会被释放掉，同时该共享主存区对应的 shmid_ds 数据结构也会被释放。

4.2.4.2 共享主存基本操作

System V 共享主存涉及的函数包括 shmget( )、shmat( )、shmdt( )及 shmctl( )，这些函数包含在以下两个头文件中：

```
#include <sys/ipc.h>
#include <sys/shm.h>
```

shmget( )用来获得共享主存区的 ID，如果不存在指定的共享区，则创建相应的区域。shmat( )把共享主存区映射到调用进程的地址空间中去，这样，进程就可以方便地对共享区进行访问操作。shmdt( )调用用来解除进程对共享主存区的映射。shmctl( )实现对共享主存区的控制操作。

需要注意，shmget( )的内部实现包含许多重要的 System V 共享主存机制。shmat( )在把共享主存区映射到进程虚拟地址空间时，并不真正改变进程的页表，当进程第 1 次访问主存映射区访问时，会因为没有物理页表的分配而导致一个缺页异常，然后主存会根据相应的存储管理机制为共享主存映射区分配相应的页表项。

应当指出，共享主存区机制只为进程提供用于实现通信的共享主存区和对共享主存区进行

操作的手段，然而并未提供对该区进行互斥访问及进程同步的措施。因而当用户需要使用该机制时，必须自己设置同步和互斥措施才能保证实现正确的进程间通信。

1. shmget( )函数

该函数用于创建、获得一个共享主存区，函数原型定义如下：

```
int  shmget(key_t key, int size, int flag);
```

其中，key 是共享主存区的名字。size 是其大小(以字节计)，如果正在创建一个新区，则必须指定其 size；如果正在存访一个现存的区，则将 size 指定为 0。flag 是用户设置的标志，如 IPC_CREAT。IPC_CREAT 表示若系统中尚无指名的共享主存区，则由内核建立一个共享主存区；若系统中已有共享主存区，便忽略 IPC_CREAT。

该函数成功返回 0，失败则返回–1。

例如，以下调用将创建一个关键字为 key，长度为 size 的共享主存区。

```
shmid=shmget(key,size,(IPC_CREAT|0400));
```

2. shmat( )函数

该函数将共享主存区连接到进程的虚拟地址空间上，函数原型定义如下：

```
void *shmat(int shmid, char *addr, int flag);
```

其中，shmid 是共享主存区的标识符。addr 是指针变量，一般情况下为 NULL，这是系统用进程中的首个适用地址来映射共享主存，并把地址赋值给 addr。除非特殊情况，一般不会给 addr 赋初值。flag 规定共享主存区的读、写权限，以及系统是否应对用户规定的地址做舍入操作。其值为 SHM_RDONLY 时，表示只能读；其值为 0 时，表示可读、可写；其值为 SHM_RND(取整)时，表示操作系统在必要时舍去这个地址。

函数调用成功返回 0，失败返回–1。

3. shmdt( )函数

该函数把一个共享主存区从指定进程的虚拟地址空间断开，函数原型定义如下：

```
int shmdt(char *addr);
```

其中，addr 是要断开连接的虚拟地址，亦即以前由连接的函数 shmat( )所返回的虚拟地址。对于成功映射的指向共享主存的指针，当进程结束后，即使没有调用 shmdt( )函数，系统也会自动取消这种映射，不会给主存系统带来什么影响。

函数调用成功返回 0 值，失败则返回–1。

4. shmctl( )函数

该函数控制一个共享主存区的属性，读取或修改其状态信息，函数原型定义如下：

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

其中，buf 是用户缓冲区地址。cmd 是操作命令，包括以下几种类型。

- IPC_STAT：对此段取 shmid_ds 结构，并存放在由 buf 指向的结构中。
- IPC_SET：按 buf 指向的结构中的值设置与此段相关结构中的 3 个字段，即 shm_perm.uid、shm_perm.gid 以及 shm_perm.mode。使用此命令的用户必须是共享主存的所有者、创建者或超

级用户。

• IPC_RMID：从系统中删除该共享存储段。不管此段是否仍在使用，该段标识符立即从调用进程被删除，所以不能再用 shmat 与该段连接。该命令的执行和 IPC_SET 需要相同的用户权限。

• SHM_LOCK：锁住共享存储段。此命令只能由超级用户执行。

• SHM_UNLOCK：解锁共享存储段。此命令只能由超级用户执行。因为每个共享存储段有一个连接计数(shm_nattch 在 shmid_ds 结构中)，所以除非使用该段的最后一个进程终止或与该段脱接，否则不会实际上删除该存储段。锁住或解锁一个共享主存的目的是，阻止或允许共享主存区被系统交换到 swap 中去。

4.2.4.3 共享主存应用

使用共享主存和使用 malloc( )来分配主存区很相似。使用共享主存的基本步骤是：

① 对一个进程/线程使用 shmget( )分配主存区。

② 使用 shmat( )连接一个或多个进程/线程到共享主存中，也可用 shmctl( )来获取信息或控制共享存区。

③ 使用 shmdt( )从共享存区中分离。

④ 使用 shmctl( )解除分配空间。

以下代码给出使用共享主存的一般框架：

```
/*****使用共享主存区*****/
/*建立共享主存区*/
intshared_id;
char *region;
const intshm_size = 1024;
shared_id = shmget(IPC_PRIVATE,/*保证使用唯一 ID*/
shm_size,
IPC_CREAT | IPC_EXCL          /*创建一个新的共享主存区*/
S_IRUSR | S_IWUSR);           /*使当前用户可以读写该区*/
/*生成进程*/
/*将新建的主存区放入进程/线程*/
region = (char*) shmat(segment_id, 0, 0);
/*其他程序代码*/
...
/*将各个进程/线程分离出来*/
shmdt(region);
/*破坏掉共享主存区*/
```

以下两程序展示共享主存区的用法。其中，shmwritetest.c 创建一个共享主存区，并写入数据；shmreadtest.c 打开已经存在的共享主存区，然后从共享主存区中读取并显示数据。程序代

码如下：

```
/***** shmwritetest.c *****/
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
typedef struct {
    char name[4];
    int age;
} people;
main(int argc, char** argv)
{
        int shm_id,i;
        key_t key;
        char temp;
        people *p_map;
        char* name = ".";
        key = ftok(name,0);
        if(key== -1)
            perror("ftok error");
    shm_id=shmget(key,4096,IPC_CREAT);
    if(shm_id== -1) {
        perror("shmget error");
        return;
    }
    p_map=(people*)shmat(shm_id,NULL,0);
    temp='a';
    for(i = 0;i<10;i++) {
        temp+=1;
        memcpy((*(p_map+i)).name,&temp,1);
        (*(p_map+i)).age=20+i;
    }
    if(shmdt(p_map)== -1)
        perror(" detach error ");
}

/***** shmreadtest.c *****/
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
```

```
typedef struct{
    char name[4];
    int age;
} people;

main(int argc, char** argv)
{
    int shm_id,i;
    key_t key;
    people *p_map;
    char* name = ".";
    key = ftok(name,0);
    if(key == -1)
        perror("ftok error");
    shm_id = shmget(key,4096,IPC_CREAT);
    if(shm_id == -1) {
        perror("shmget error");
        return;
    }
    p_map = (people*)shmat(shm_id,NULL,0);
    for(i = 0;i<10;i++) {
        printf( "name:%s\n",(*(p_map+i)).name );
        printf( "age %d\n",(*(p_map+i)).age );
    }
    if(shmdt(p_map) == -1)
        perror(" detach error ");
}
```

编译两程序后，先执行写入程序，然后执行读出程序。运行结果如下：

```
name: b
age 20;
name: c
age 21
```

4.3 实 验 内 容

4.3.1 实验 1 消息队列实现进程间通信

4.3.1.1 实验说明

利用消息队列解决客户及服务进程之间的通信问题。

4.3.1.2　解决方案

为实现客户进程与服务进程之间基于消息队列的通信，服务进程首先需要创建一个消息队列，随后客户进程可以打开消息队列，从而实现消息的发送和接收。涉及的函数包括 msgget( )、msgsnd( )、msgrev( )及 msgctl( )。这里将以编制一个长度为 1 KB 的消息队列为例，给出客户进程与服务进程的参考程序。

4.3.1.3　程序框架

1. 服务端程序

```
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#define MSGKEY 75
struct  msgform{                    /*定义消息格式*/
   long  mtype;
   char  mtext[1000];
}msg;

void server( )
  {
    /*创建标识号为 MSGKEY 的消息队列*/
    do {
        /*接收消息*/
        /*打印“接收到消息”提示信息*/
    }while(msg.mtype!=1);
    /*删除消息队列，归还资源*/
    /*退出*/
  }

main( )
{
  server( );
}
```

2. 客户端程序

```
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#define MSGKEY 75
```

```
struct msgform{
    long mtype;
    char mtext[1000];
}msg;

void client( )
  {
  /*打开标识号为 MSGKEY 的消息队列*/
  for(i=10;i>=1;i--) {
      msg.mtype=i;
      printf("(client)sent\n");
      /*发送消息 msg*/
  }
  /*退出系统*/
}

main( )
{
 client( );
}
```

4.3.2 实验 2 信号量实现进程同步

4.3.2.1 实验说明

利用信号量解决生产者-消费者问题。

4.3.2.2 解决方案

进程同步是操作系统多进程/多线程并发执行的关键之一，进程同步指为完成共同任务的并发进程基于某个条件来协调它们的活动，这是进程之间发生的一种直接制约关系。生产者-消费者问题是典型的进程同步问题，其本质是如何控制并发进程对有界共享主存区的访问。生产者进程生产产品，然后将产品放置在一个空缓冲区中供消费者进程消费。消费者进程从缓冲区中获得产品，然后释放缓冲区。当生产者进程生产产品时，如果没有空缓冲区可用，那么生产者进程必须等待消费者进程释放出一个空缓冲区。当消费者进程消费产品时，如果没有满的缓冲区，那么消费者进程将被阻塞，直到新的产品被生产出来。

4.3.2.3 程序框架

下面以生产者进程不断向数组添加数据(写入 100 次)，消费者从数组读取数据并求和为例，

给出基于信号量解决生产者-消费者问题的程序框架。该程序假设有一个生产者进程和两个消费者进程，创建了 fullid、emptyid 和 mutxid 共 3 个信号量，供进程间同步访问临界区。同时，还建立 4 个共享主存区，其中 array 用于维护生产者、消费者进程之间的共享数据，sum 保存当前求和结果，而 set 和 get 分别记录当前生产者进程和消费者进程的读写次数。

```
#include <sys/mman.h>
#include <sys/types.h>
#include <linux/sem.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <time.h>
#define MAXSEM 5
/*声明 3 个信号量 ID*/
int fullid;
int emptyid;
int mutxid;

int main( )
{
 struct sembuf P,V;
 union semun arg;
 /*声明共享主存*/
 int *array;
 int *sum;
 int *set;
 int *get;
 /*将 array、sum、set 和 get 映射到共享主存*/
 /*创建信号量 fullid,emptyid 和 mutxid*/
  /*为信号量赋值*/
 arg.val = 0;
 if(semctl(fullid , 0 , SETVAL , arg) == –1) perror("semctl setval error");
       arg.val = MAXSEM;
 if(semctl(emptyid , 0 ,SETVAL , arg) == –1) perror("semctl setval error");
       arg.val = 1;
 if(semctl(mutxid , 0 ,SETVAL , arg) == –1) perror("setctl setval error");
/*初始化 P、V 操作*/
/*创建生产者进程*/
 if(子进程)  {
```

```
    while( i < 100) {
    /*对 emptyid 和 mutxid 执行 P 操作*/
    array[*(set)%MAXSEM] = i + 1;
    (*set)++;
     /*对 emptyid 和 mutxid 执行 V 操作*/
    i++;
    }
/*休眠一段时间*/
 /*打印“生产者结束”提示信息*/
 /*退出*/
}else {
   /*创建消费者进程 A*/
   if(子进程) {
       while(1){
        /*对 emptyid 和 mutxid 执行 P 操作*/
        if(*get == 100)
           break;
       *sum += array[(*get)%MAXSEM];
       printf("The ComsumerA Get Number %d\n", array[(*get)%MAXSEM] );
       (*get)++;
       if( get ==100)
          printf("The sum is %d \n ", sum);
          /*对 emptyid 和 mutxid 执行 V 操作*/
          /*休眠一段时间*/
   }
   /*打印“消费者结束”提示信息*/
   /*退出*/
}
   }else {
          /*创建消费者进程 B*/
          if(子进程) {
              while(1){
              /*对 emptyid 和 mutxid 执行 P 操作*/
               if(*get == 100)
                  break;
                  *sum += array[(*get)%MAXSEM];
                   printf("The ComsumerB Get Number %d\n", array[(*get)%MAXSEM] );
                    (*get)++;
                    if( get ==100)
                       printf("The sum is %d \n ", sum);
```

```
                    /*对 emptyid 和 mutxid 执行 V 操作*/
                    /*休眠一段时间*/
                }
                printf("ConsumerB is over");
                exit(0);
            }
        }
    }
}
```

4.3.3 实验 3 基于信号量采用多线程技术实现进程同步

4.3.3.1 实验说明

利用信号量解决理发师问题，可采用 pthread 线程库的 sem_wait( )及 sem_post( )函数实现。

4.3.3.2 解决方案

理发店里有一位理发师、一把理发椅和 n 把供等候理发的顾客坐的椅子。如果没有顾客，理发师便在理发椅上睡觉。一个顾客到来时，它必须叫醒理发师。如果理发师正在理发时又有顾客来到，则如果有空椅子可坐，就坐下来等待，否则就离开。理发师问题是一个经典的进程同步问题，共需要 3 个信号量和一个控制变量来协调理发师、理发椅及顾客之间的活动。

- 信号量 customers(不包括正在理发的顾客)用来记录等候理发的顾客数，并用作阻塞理发师进程，初值为 0。
- 信号量 barbers 用来记录正在等候顾客的理发师数，并用作阻塞顾客进程，初值为 0。
- 信号量 mutex 用于互斥，初值为 1。
- 控制变量 waiting 用来记录等候理发的顾客数，实际上是 customers 的一份副本，初值均为 0。之所以使用 waiting 是因为无法读取信号量的当前值。

在理发师问题中，进入理发店的顾客必须先看等候的顾客数，如果少于椅子数，留下来等，否则就离开。整个活动主要受理发师和顾客的行为协同完成。具体处理流程请参见《操作系统教程(第 4 版)》(高等教育出版社，2008)第 3 章。

4.3.3.3 程序框架

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <semaphore.h>
```

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

#define CHAIRS 5 /*椅子数*/
sem_t customers;   /*等待服务的顾客信号量*/
sem_t barbers;     /*等待顾客的理发师信号量*/
pthread_mutex_t mutex; /*互斥变量*/
int waiting = 0;    /*  正在等待的顾客数*/

void barber (void);
void customer (void *num);
void cut_hair(void);

double timediff(struct timeval i, struct timeval j);
void seed_random(void);
double flat(void);
double normal(void);
double bursty(void);

int main( )
{
   int i;
   seed_random( );
   /*创建理发师线程 barber_t*/
   /*每隔一定时间间隔创建一个顾客线程 customer_t*/
}

double timediff(struct timeval now, struct timeval earlier) {
   if (now.tv_sec == earlier.tv_sec)
      return (now.tv_usec - earlier.tv_usec) / 1000000.0;
   else
      return (1000000 * (now.tv_sec - earlier.tv_sec) +
         now.tv_usec - earlier.tv_usec) / 1000000.0;
}

void barber(void) {
   while (1) {
      /*将顾客信号量减 1*/
      /*共享变量加锁*/
       waiting = waiting–1;
```

```
        /*对信号量 barbers 加 1*/
        /*共享变量解锁*/
        cut_hair( );/*理发*/
    }
}

void cut_hair (void) {
    printf("    Barber: I am cutting the customer's hair...\n ");
    usleep(100000);   /*理发时间*/
    printf("    Barber: done.\n");
}

void customer(void *num) {
    /*共享变量加锁*/
    if (等待顾客个数小于椅子个数) {
        waiting = waiting + 1;
        /*对信号量 customers 加 1*/
        /*共享变量解锁*/
        /*令信号量 barbers 减 1*/
    } else {
        /*打印“等待人数过多”提示信息*/
        /*共享变量解锁*/
    }
     /*释放占用资源*/
}

void seed_random(void) {
    struct timeval randtime;
    unsigned short xsub1[3];
    gettimeofday (&randtime, (struct timezone *)0);
    xsub1[0] = (ushort) randtime.tv_usec;
    xsub1[1] = (ushort)(randtime.tv_usec >> 16);
    xsub1[2] = (ushort)(getpid( ));
    seed48(xsub1);
}

double flat( )
{
    return drand48( ) / 5.;
}
```

```
double normal( )
{
   return sin(M_PI * drand48( )) / M_PI / 2.006999;
}

double bursty( )
{
   return (sin(M_PI + drand48( ) * M_PI) + 1.) / 3.591;
}
```

4.3.4 实验 4 共享主存实现进程间通信

4.3.4.1 实验说明

利用共享主存解决读者-写者问题。要求由写者创建一个共享主存，并向其中写入数据，读者进程随后从该共享主存区中访问数据。

4.3.4.2 解决方案

为基于共享主存解决读者-写者问题，需要由写进程首先创建一个共享主存，并将该共享主存区分别映射到读者进程和写者进程的虚拟地址空间。随后，写进程开始向共享主存写数据，读进程从共享主存区获取数据。以下两程序分别给出了读者进程与写者进程的代码框架。

4.3.4.3 程序框架

```
/***** write.c *****/
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
typedef struct{
 char name[4];
 int age;
} people;

main(int argc, char** argv)
{
  /*调用 ftok( )函数创建一个键值*/
  if(创建键值失败)
     /*打印“创建键值失败”提示信息*/
```

```
    /*调用 shmget 创建一块共享主存区*/
if(创建共享主存失败) {
    /*打印“创建共享主存失败”提示信息;*/
    /*返回*/
 }
/*将共享主存区附加到自己的主存段*/
/* 向共享主存中写入数据*/
/*将其从自己的主存段中“删除”出去*/
 if(删除失败)
   /* 打印“删除失败”提示信息*/
  /* 返回*/
}

/***** read.c *****/
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
typedef struct{
 char name[4];
 int age;
} people;
main(int argc, char** argv)
{
 /*调用 ftok( )函数创建一个键值*/
  if(创建键值失败)
     /*打印“创建键值失败”提示信息*/
    /*调用 shmget 获取一块共享主存区*/
  if(获取共享主存失败) {
    /*打印“获取共享主存失败”提示信息*/
     /*返回*/
  }
  /*将共享主存区附加到自己的主存段*/
  /*向共享主存中写入数据*/
  /*将其从自己的主存段中“删除”出去*/
  if(删除失败)
   /*打印“删除失败”提示信息*/
   /*返回*/
 }
}
```

第5章　Shell 程序设计

5.1　实 验 目 的

- 了解 Shell 在操作系统中的作用。
- 理解 I/O 重定向和管道。
- 学会编写简单的 Shell 脚本程序。
- 学会运行 Shell 命令文件。

5.2　背 景 知 识

5.2.1　Shell 简介

操作系统提供两种基本用户接口：第 1 种是程序接口，以函数(系统调用)形式向程序员提供支持，通过 fork( )、read( )、write( )等函数为上层软件提供服务，函数是用户应用程序与操作系统之间的一种接口；第 2 种是命令接口，当用户在控制作业运行、浏览文件信息和访问硬件资源时，使用此接口和操作系统进行交互。Shell 为用户和操作系统之间的命令交互提供最基本的接口，在使用者和操作系统之间架起一座桥梁，它的名字 Shell(外壳)形象地表示它在用户与操作系统内核之间的关系。早期 Shell 主要用作命令解释器，经过不断扩充和发展，现在已是命令语言、命令解释器、程序设计语言的统称，被泛指为一个提供人机交互界面和接口的程序。

最简单的 Shell 程序是基于字符的，用户通过输入一个字符串到 Shell 中来与操作系统进行交互，并且操作系统的响应结果是输出一行行的字符到屏幕上，更方便的 Shell 程序使用图符点击来操作界面。在 UNIX/Linux 系统中，有许多 Shell 可以使用，它们大多数都是从最初的 Bourne Shell 演变而来的，表 5-1 列出各种 Shell 的起源。

表 5-1　各种 Shell 的起源

Shell 名称	起　　源
sh(Bourne Shell)	最初的 Shell，由 Bell 实验室的 Steven Bourne 开发
csh	Bill Joy 在 BSD UNIX 上开发的 Shell，与 sh 不兼容
ksh	Bell 实验室的 David Korn 开发，界面友善、效率很高、商业版 UNIX 中默认配置
bash(Bourne Again Shell)	Linux 的标准 Shell，是 GNU 项目，由 Brain fox 开发。具有命令行历史、编辑、补全和别名扩展功能

续表

Shell 名称	起　源
tcsh	C Shell 的加强版，提供编辑命令行的功能
zsh	与 sh 兼容，提供编辑命令行的功能

在默认情况下，Shell 安装在路径/bin/sh 下。在 Linux 中，/bin/sh 是一个指向/bin/bash 的链接，即实际使用的 Shell 是 bash，该 Shell 包括了 C Shell 和 Korn Shell 的特性，并向下兼容 Bourne Shell 的语法，在 Linux 系统中被广泛地使用，当然也可以安装并使用 sh、csh、ksh、zsh 和 tcsh。

5.2.2　Shell 的主要功能

当用户登录到计算机系统时，会启动 Shell 程序，它不属于 Linux 内核部分，而是在内核之外，以用户态方式运行。其基本功能是解释并执行用户输入的各种命令，把相应命令程序加载到主存，改造成进程并启动它们运行，从这一点上来看，Shell 是一个程序启动器(program launcher)。系统初启后，内核为每个终端建立一个进程去执行 Shell 解释程序。它的执行过程按照如图 5-1 所示步骤进行。

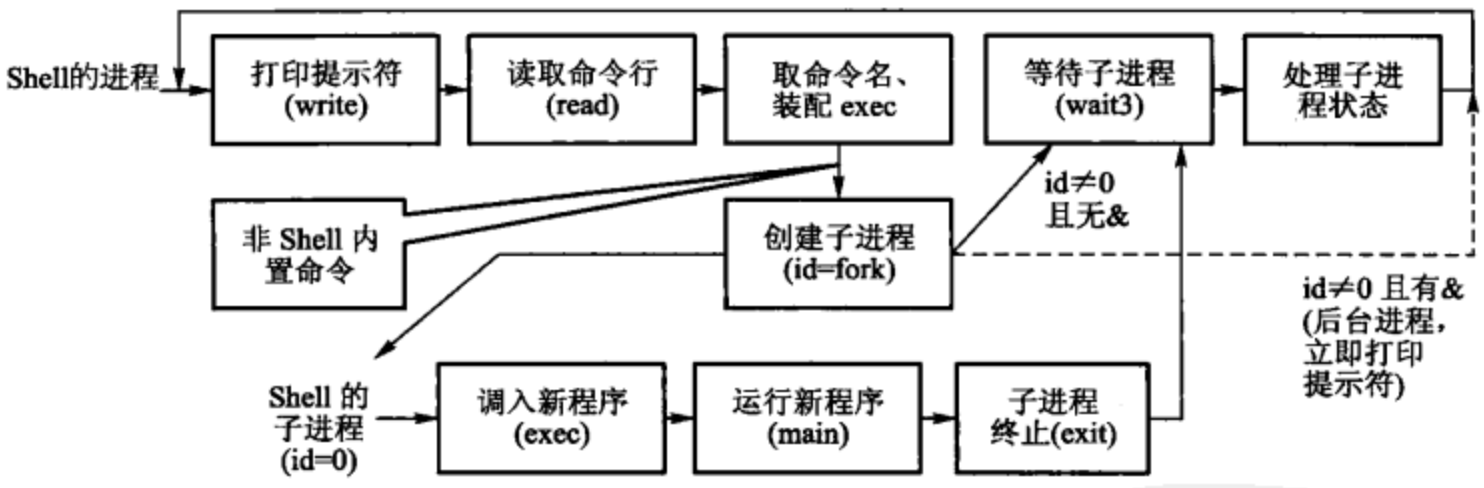


图 5-1　Shell 的执行步骤

从图 5-1 中可以看到，Shell 进程从自己的标准输入流(stdin)中读取要执行的命令或用户的可执行程序，根据输入规则对读取的字符串进行解析。在 Linux 系统中，标准输入、标准输出及标准错误输出流是指用户注册所使用的终端。如果解析后 Shell 识别出输入的字符串是一个内置命令，如 cd、ls、ps 等，Shell 就在自己的程序内部执行该命令，不用创建子进程；否则，就认为该字符串是一个可执行命令文件，此时，Shell 进程使用 fork( )和 exec( )组合，生成一个执行该程序的子进程；而 Shell 进程作为父进程，在子进程运行期间，执行 wait( )函数，处于等待状态，直到子进程执行 exit( )函数退出，它再继续循环读取用户键入的下一条命令。

除了解释执行用户命令外，Shell 也是一种高级程序设计语言，它的其他功能还包括：

- 设置环境变量。对用户工作环境进行修改和设定，根据规则选择设定的环境变量。

• 提供内置命令：提供一批内置命令，如 cd、ls、ps、echo、exit、pwd、exec、kill、wait、unset、eval、shift 等，直接在 Shell 进程地址空间执行，收到这些命令时并不需要创建新进程，从而提高了系统效率。

• 输入输出重定向：控制对系统标准流的修改。

• 建立和连通管道：利用简单的编程方法和命令组合，完成复杂的处理工作。

• 执行后台命令：安排命令在后台执行，前后台进程轮流执行可提高系统效率。

• Shell 程序设计：使用 Shell 脚本(script)，可将用户编写的程序与操作系统命令结合在一起，作为新命令使用，完成复杂的命令控制或用户环境设置。

5.2.3 Shell 主要功能的实现

5.2.3.1 命令解析

Shell 主要功能之一是解析用户输入的命令，用户登录时，会启动 Shell，它打印出提示符，通知用户进行命令输入。不同的 Shell 程序可能会有不同的默认提示符，常见的有单个符号：%、#、>。在 bash 中，通常命令提示符的形式为：

[username@servername]%

username 是当前用户的用户名，servername 是计算机的机器名。用户可以通过改变配置文件内容来更改提示符的格式，例如在提示符上显示当前的目录名。

当 Shell 输出提示符后，便执行一个阻塞读操作进入睡眠状态，直到用户输入一个完整命令行(命令行以一个 NEWLINE 结束)。用户输入结束后，命令行字符串返回给 Shell，并结束等待。

下一步是解析命令行，解析过程从命令行的左边开始向右扫描，直到遇到空白(如空格、制表符或 NEWLINE)为止。第 1 个单词为命令名，而后面的单词则是参数，以命令名作为文件名，其他参数改造为函数 exec( )内部处理所要求的形式。

如果该命令是内置命令，则直接跳转到代码区去执行，否则 Shell 进程会调用 fork( )创建子进程，本身则通过函数 wait( )等待子进程完成。子进程被启动时，调用 exec( )，根据命令指定的文件名到目录中查找有关文件(该命令解释程序构成的可执行二进制文件)，调入主存并执行该程序。当子进程完成处理并终止时，将唤醒 Shell 进程来做必要的判别工作，然后再次发提示符，通知用户输入新命令，重复上述处理过程。这种创建新进程来执行计算的思想，可能会导致进程过多，但却有一个很关键的好处，即对原进程进行保护，免遭可能在执行新计算时引起的致命错误的破坏，从而引起整个系统的瘫痪。

Shell 为每个用户提供一组环境变量，这些变量最初定义在用户的.login(bash 中为.bash_profile)文件中，可以通过 set 命令修改环境变量。在这些环境变量中，PATH 变量值指明 Shell 查找可执行文件的顺序，它是一组绝对路径列表，列表中的每一项为一个目录名，Shell 在这些目录中依次寻找可执行文件。例如，如果 PATH 环境变量的值为：

PATH=/bin:/usr/local/bin:/usr/bin

Shell将先搜索/bin目录，再搜索/usr/local/bin目录，然后搜索/usr/bin，如果以上的每个目录都找不到用户指定的可执行文件，Shell将提示用户无法找到命令。

5.2.3.2 命令执行

Shell可用两种方式执行命令：前台执行和后台执行。在默认情况下，Shell以前台方式执行命令，以该种方式执行时，Shell创建一个子进程，子进程加载可执行命令文件，Shell进程进入睡眠状态，等待子进程执行结束。在子进程执行命令的过程中，Shell进程将不再接受下一个命令输入，而用户此时输入的各种字符将作为子进程的输入信息，子进程的输出也将显示在屏幕上。在前台执行方式下，Shell进程和子进程并不是并发执行的，只有当子进程退出后，Shell进程才再次打印命令提示符，重新接受用户的命令输入。

命令也可以后台方式执行，如果命令行结尾以“&”结束，则表明用户希望Shell以后台方式执行此命令。在这种方式下，Shell在创建子进程之后并不等待它运行结束，而是直接显示命令提示符，等待接受用户输入其他命令。通过这种方式，可让父子进程并发执行。如果用户启动多个命令，每个命令都以“&”结尾，且每个命令相对都要花较长时间执行，那么系统中将同时运行许多进程。

当有多个进程并发执行时，每个进程都期望从键盘获得输入，那么用户就无法知道哪个进程将获取从键盘输入的数据。同样，这些并发进程如果都将自己的输出字符流输出到显示器上，那么当时光标在哪里，字符就只能输出到哪里。所以，如果一个程序需要从键盘输入数据，就不应该把它放在后台运行，以避免前后台程序对键盘访问的冲突。利用前后台进程轮流在CPU上执行，可充分利用资源、提高工作效率。通常规定后台进程的优先级低于前台进程的优先级。因此，只要有可运行的前台进程，就调度前台进程运行，仅当CPU空闲时，才让后台进程运行。

5.2.3.3 I/O重定向

通常，执行Shell命令行并创建一个进程时，会自动打开3个文件：标准输入(stdin)，标准输出(stdout)和标准错误输出(stderr)。默认情况下，Shell将命令的标准输入定为键盘，标准输出及标准错误输出定为显示器。当然，输入数据可不来自键盘，而来自指定文件；标准输出和错误输出可不送到屏幕上，而是输出到指定文件中，用户只要输入命令就可以重定义stdin或stdout。

用户在命令后用输入重定向符“<”连接一个文件，那么该文件就被定义为输入文件；如果用户通过输出重定向符“>”连接一个文件，那么该文件就被定义为输出文件。例如，wc命令可以统计文件包含的行数、单词数和字符数。如果用户输入：

[root@server]#wc <myinput

wc程序将统计myinput文件的行数、单词数和字符数，并将结果显示在屏幕上。如果用户输入：

[root@server]#wc <myinput > myoutput

执行效果是 wc 程序从 myinput 读取输入，并对文件信息进行统计，但是统计结果并不显示在屏幕上，而是写入 myoutput 文件。

此外，还有输出附加定向符“>>”，用于把输出附加到指定文件尾，即时文件定向符“<<”用于把 Shell 程序的输入行重定向到一个文件中。用户也可根据需要对错误输出进行重定向，不同 Shell 其修改错误输出流的方法略有不同。

5.2.3.4 Shell 管道

管道是 UNIX 最早的进程间通信机制，把一个进程连接到另一个进程的数据流称为一个“管道”。管道连接的每个程序都作为一个独立进程运行，于是 Shell 可以通过该机制将一个进程的输出连接到另一个进程的输入。管道命令方式体现了 UNIX/Linux 系统命令实现中的重要思想，可利用简单编程方法和命令组合功能完成比较复杂的处理工作。例如，用户在命令提示符后输入：

[root@server]# cmd1 | cmd2

Shell 负责安排两个命令的标准输入和标准输出，且使用管道符“|”分开：

- cmd1 的标准输入来自终端键盘。
- cmd1 的标准输出进入 cmd2，作为 cmd2 的标准输入。
- cmd2 的标准输出连接到终端屏幕。

Shell 所做的工作从最终效果上看是这样的：重新安排标准输入和标准输出流之间的连接，使数据从键盘输入流过两个命令再输出到屏幕，如图 5-2 所示。

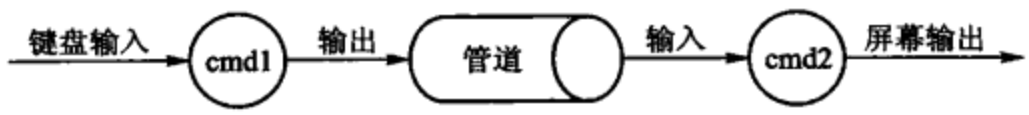


图 5-2 管道

再如，用户要求执行命令序列：

[root@server]#ls –l myfile
[root@server]#ws –l myfile
[root@server]#rm myfile

其功能是：ls 将当前目录下的文件及目录按行组织并存入 myfile；wc 计算 myfile 包含的行数，即当前目录下的文件及目录个数；rm 删除 myfile 文件。若使用管道，则只需一条命令：

[root@server]#ls –l | ws –l

显然，使用管道命令使用户不但减少工作量，而且使用十分方便。

在管道中，每个命令执行时都有一个独立的进程，前一个命令的输出正是下一个命令的输入。如果只需要前面命令执行结果的一部分信息，而输出信息内容又很多时，仅用管道命令不易实现选择，管道中有一类命令也称为“过滤器”，可解决此类问题。过滤器首先读取输入，然

后将输入进行简单的变换(过滤)，再将处理结果输出，例如 grep、tail、sort 和 wc 等命令都是过滤器。一个管道中可以包括多条命令，例如，grep 从指定文件或输出流中搜索指定模式，命令如下：

```
[root@server]#ls | grep fei?.c | wc –l
```

其功能是显示当前目录中以 fei 打头，后跟一个字符的所有 C 文件个数。

此外，管道还可以完成更复杂的处理工作，下列字符可以用做命令表的分隔符，达到命令执行中的不同效果。

- ;：在管道后面放分号，表示多个命令或多个管道方式的命令顺序执行。
- &：在命令行尾输入“&”，表示左边的管道的输出为后台进程，可与前台进程并发执行。
- &&：在两个命令之间用“&&”替换管道符号“|”时，表示当左边的命令输出为真值时，执行右边的命令。
- ||：在两个命令之间用“||”替换原管道符号“|”时，表示当左边的命令输出为假值时，执行右边的命令。

例如，输入下列命令：

```
[root@server]#write smith < letter || mail smith <letter
```

其作用是首先用 write 命令直接向 smith 用户发送 letter 中的信息，若发送操作失败时，再用 mail 命令向 smith 用户发送邮件 letter，其中“||”符号取代了管道符的作用。

5.2.3.5 环境变量和环境文件

在用户注册过程中，系统为其建立用户环境，也称为 Shell 环境，它由许多变量及变量值组成，这些变量和变量值决定用户环境的外观特性。Shell 环境包括用户所使用的 Shell 类型、主目录所在位置及正在使用的终端类型等多种内容。决定这些内容的变量有许多是在注册过程中定义的，一些为只读变量，另外一些为非只读变量，可由用户随意添加或修改。下面列举用户可以定义的主要环境变量。

- HOME：用户注册后的当前目录。不带参数的 cd 命令等价于 cd $HOME 命令。
- LOGNAME：用户注册名，系统通过它来确认某用户是文件主、有权执行某个命令或者是某邮件的作者等。
- MAIL：存放用户邮件的信箱。通过定时查询这个文件最近更新的时间来判断是否有新邮件到达信箱。系统设置的 MAIL 变量名为/usr/spool/mail/用户注册名。
- PATH：Shell 从中查找命令的目录列表。使用冒号分隔所有目录路径名，将按 PATH 变量中给出的顺序搜索目录，找到的第 1 个与命令名称一致的可执行文件将被执行，如/etc:/usr/bin。
- Shell：当前使用的 Shell，也指出 Shell 的存放路径。
- TERM：终端的类型，如 DEC 公司的 vt-100。
- UID：当前用户的标识符，取值是由数字构成的字符串。

• IFS：输入域分隔符。输入时分隔单词，通常为空格、制表符和换行符。

• PWD：当前工作目录的绝对路径名，该变量的取值随 cd 命令的使用而变化。

• PS1：命令主提示符，在特权用户下，默认的主提示符是“#”；在普通用户下，默认的主提示符是“$”。

• PS2：命令辅助提示符，在命令结束前遇到换行时，续行中使用的提示符，默认的主提示符是“>”。

• TZ：定义有关时区的信息。在中国，该变量设置为 BJT-8，即北京与格林尼治时差为 8 小时。

环境变量的使用和删除用以下方法：可以用 echo 命令察看一个环境变量的值，也可以在命令中将环境变量的值作为参数。使用环境变量的值时，需要在其名称前面加上“$”符号，如$echo $SHELL。删除环境变量使用 unset 命令，如$unset NAME。

当用户注册成功进入系统之后，Shell 会读取一些称为脚本的环境文件，并执行其中的每条命令。bash 的环境文件包括：.bash_profile 文件、.bashrc 文件、.bash_logout 文件等。在.bash_profile 中设置环境变量和文件掩码；在.bashrc 中，只含有针对 bash 的命令，可用来设置别名，.bashrc 在.bash_profile 之后执行；.bash_logout 仅在用户退出注册的时候运行，可以把清屏等命令放在这里，做退出注册时的扫尾工作。修改环境变量需特别小心，因为它相当于一种全局变量，会改变程序行为，产生不可预测的后果。

5.2.4 Shell 编程

Shell 是一种解释型程序设计语言，它支持大多数高级程序设计语言中能见到的程序成分，如函数、变量、数组和控制语句，具有极强的程序设计能力，可被用来编写 Shell 脚本。Shell 脚本包含一系列命令，运行脚本就是执行脚本中的每个命令，可用一个脚本在一次运行中执行许多个命令。Shell 脚本是由 Shell 解释执行的程序，能为用户提供强大的任务处理能力，可完成更为复杂和自动执行要求高的日常任务。下面以 bash 为例介绍 Shell 语言的主要成分：变量、数组、函数和控制语句，并给出简单例子。

5.2.4.1 特殊字符

Shell 语言中主要包含以下特殊字符。

• 星号*：通配符，可匹配任何字符串。

• 问号?：通配符，可匹配任何单个字符。

• 方括号[ ]：匹配括号内所有字符中的任一字符，可在括号内的字符间用短划线“-”表示字符范围，例如，myfile[23456]与 myfile[2-6]效果相同，可以匹配文件 myfile2 至 myfile6。

• 反斜杠\：为了将 Shell 的特殊字符*、?、[ ]、&和；等变成普通字符，必须在这些字符前加\，此外，\放行尾为续行符。

• 双引号“”：其所括字符中，除$、\及，外的其他特殊字符都失去特定含义。

- 单引号‘’：其所括字符中的所有特殊字符都失去特定含义。
- 倒引号``：其所括字符串被 Shell 解释为命令行。

5.2.4.2 变量

变量是用于保存值或正文的词，变量可以被创建、赋值和删除，可以现定义、现赋值。通常，所有变量都被看做字符串并以字符串来存储，即使它们被赋值为数值时也是如此，Shell 和软件工具会在需要时把数值型字符串转换为对应的数值以对它们进行操作。Shell 变量分为 5 类，下面逐一介绍。

1. 用户自定义变量

用户自定义变量也称局部变量，是用户在 Shell 脚本中定义的变量，可以对它赋值，形式为：

变量名=字符串

如 myfile=/usr/fei/fei1.c。使用范围仅限于定义它的程序，程序执行结束，其值也就不再存在。

2. 位置变量

位置变量类似于 C 语言的命令行参数，用在 Shell 程序名之后，共有 9 个，用$n 表示，n 为一个十进制数，在参数传递时必须使命令行中提供的参数与程序中的位置参数一一对应。$0 是一个特殊变量，它是当前 Shell 程序的文件名参数，第 1 个位置变量名为$1，第 2 个位置变量名为$2，以此类推。多于 9 个时可用 Shift 命令移动位置参数。

3. Shell 变量

Shell 变量的名字和含义是固定的，$?为得到上次命令执行后的十进制返回码，$$为得到当前 Shell 进程的进程号，$!为得到上个后台进程的进程号，$#为得到传递给 Shell 程序的位置参数个数(不限于 9 个，但不包含 Shell 文件名)，#为注释符，$-为当前 Shell(用 set)设置的执行标志名组成的字符串，$*为命令行中实际给出的实参字符串。Shell 变量是由 Shell 程序本身设置的特殊变量，根据实际情况赋值，不允许用户重新设置。

4. 参数替换

参数替换指一些变量值不是直接赋值或由传递参数来获得，而是该变量的值取决于其他变量的值。

5. 命令替换

一些变量的值取决于用一对倒引号 (`)括起来的命令执行的结果。

Shell 中的变量名只能包含大、小写字母、数字(0～9)和下划线，且只能以字母或下划线开头。定义变量可以使用等号(=)直接赋值，定义过的变量可以通过在变量名前加“$”符号访问其值，取消变量的定义使用 unset 变量名。

变量赋值的方法包括：使用赋值语句、使用内部命令 set、使用内部命令 read、利用命令行中的参数传值。

5.2.4.3 数组和函数

bash 仅提供一维数组，且不限定数组大小；下标从 0 开始，可以是整数或算术表达式，对

数组元素赋值的一般形式是：

数组名[下标]=值

函数与一般语言中的用法类似，应先定义后使用，允许 Shell 脚本与函数进行参数传递。

5.2.4.4 控制语句

命令在脚本中的执行顺序称脚本流，常常需要根据不同条件执行不同的脚本流，这就要求 Shell 提供各种流程控制语句。下面介绍常用的一些，细节请参考 Shell 命令手册。

1. 测试语句

测试语句 test 作为控制条件，通过测试表达式 expression 来实现一个条件测试，测试语句计算表达式的值返回真(0)或假(1)。test 在 Shell 脚本中常常缩写为[expression]的形式，表达式可分 3 类。

（1）文件测试

[-r file]表示若文件存在且为用户可读，则测试条件为真。

[-w file]表示若文件存在且为用户可写，则测试条件为真。

[-x file]表示若文件存在且为用户可执行，则测试条件为真。

[-b file]表示若文件存在且为块设备文件，则测试条件为真。

[-c file]表示若文件存在且为字符设备文件，则测试条件为真。

[-d file]表示若文件存在且为目录文件，则测试条件为真。

[-f file]表示若文件存在且为普通文件，则测试条件为真。

[-p file]表示若文件存在且为 FIFO 文件，则测试条件为真。

[-s file]表示若文件存在且文件长度>0，则测试条件为真。

[-t file]表示若文件描述符与终端相关，则测试条件为真。

（2）字符串比较

[-z s1]表示若字符串长度为 0，则测试条件为真。

[-n s1]表示若字符串长度>0，则测试条件为真。

[s1]表示若 s1 不是空字符串，则测试条件为真。

[s1=s2]表示若两个字符串相等，则测试条件为真。

[s1!=s2]表示若两个字符串不相等，则测试条件为真。

[s1<s2]表示若按字典顺序 s1 在 s2 之前，则测试条件为真。

[s1>s2]表示若按字典顺序 s1 在 s2 之后，则测试条件为真。

（3）整数比较

[int1 –gt int2]表示若 int1 大于 int2，则测试条件为真。

[int1 –eq int2]表示若 int1 等于 int2，则测试条件为真。

[int1 –ne int2]表示若 int1 不等于 int2，则测试条件为真。

[int1 –lt int2]表示若 int1 小于 int2，则测试条件为真。

[int1 –le int2]表示若 int1 小于或等于 int2，则测试条件为真。

[int1 –ge int2]表示若 int1 大于或等于 int2，则测试条件为真。

2. 判断语句

格式为：

if 条件表达式 then 命令 else 命令 fi

表达式为任意逻辑表达式，能给出返回值，大多数返回一个数值，返回 0 时执行 then 后的语句，否则执行 else 后的语句。两个特殊逻辑操作符 true 和 false，true 的返回值恒为 0。

3. 开关语句

格式为：

case 字符串 in 模式字符串 1) 命令表 1;; … 模式字符串 n) 命令表 n;; esac

case 允许多重条件选择，执行过程是：用字符串去匹配各模式字符串，发现某个匹配，便执行其后跟的各语句。

4. 循环语句

循环语句分三种：while、for 和 until。根据测试条件执行相应命令。

while 格式为：

while 测试命令 do 命令表 done

执行过程是：若测试命令返回 true，则进入循环体执行命令表，然后再做测试命令，直至返回 false 为止终止 while 语句。

for 格式为：

for 变量 in 值表 do 命令表 done

执行过程是：变量依次取值表中的各个字符串，然后进入循环体并执行命令表中的命令，变量变为空时结束 for 循环。

until 格式为：

until 测试命令 do 命令表 done

与 while 语句相似，但当测试条件为 false 时，进入循环体执行，直至测试条件为 true 时结束 until 语句。

5. 退出语句

break 语句为退出循环体，continue 语句返回本层循环头，开始新一轮循环。break[n]语句表示跳出几层循环，默认为 1。continue[n]语句表示退到包含本语句的第几层继续循环。

6. 选择语句

select 语句显示菜单项和接收用户输入。

7. 读入语句

read 语句从标准输入(键盘)上读取数据行，并给局部变量赋值。

8. 回显语句

echo 语句显示变量值或字符串。

9. 导出语句

export 语句将变量导出，使它们成为公共变量。

10. 运算语句

let 语句对后跟的算术表达式执行整数算术运算，因而 Shell 含有各种算术运算符。

11. 位置参数赋值

set 语句为 Shell 程序中的位置参数赋值。

12. 退出程序语句

exit 语句为退出执行的 Shell 脚本。

13. 计算语句

expr 语句用于表达式计算。如 expr 8+8，结果为 16。注意，只能整数操作，合法的操作符有+、–、*、/和%(求余数)。

14. 等待语句

wait 语句使 Shell 等待后台启动的所有子进程结束。

下面的程序代码编写显示给定目录下指定文件内容的 Shell 脚本。执行此 Shell 脚本时，如果第 1 个位置参数是合法的目录，那么就把后面给出的各个位置参数所对应的文件显示出来，若给出的文件名不正确，则显示出错信息。如果第 1 个位置参数不是合法的目录，则显示目录名不对。

```
/*****shellprogram1*****/
dir=$1;shift
if[ –d $dir ]
then
        cd $dir
        for name
        do
          if [ –f $name ]
         then    cat $name
                 echo "end of ${dir}/$name"
        else    echo "invaild file name:${dir}/$name"
        fi
     done
else echo "bad directory name:${dir}"
fi
```

编写从命令行输入简单的算术表达式并计算结果的 Shell 脚本。代码如下：

```
/*****shellprogram2*****/
if [ $# = 3 ]
```

```
then
    case $2 in
    +) let z=$1+$3;;
    -) let z=$1-$3;;
    /) let z=$1/$3;;
    x | X) let z=$1*$3;;
    *) echo "Warning-$2 invalied operator，only +、−、×、÷ "operator allowed.
       exit;;
    esac
        echo "Answer is $z"
    else
        echo "Usage-$0 value1 operator value2."
    fi
```

5.3 实 验 内 容

5.3.1 实验 1 编写一个简单的 Shell 程序——MyShell

5.3.1.1 实验说明

设计并实现一个简单的交互式 Shell——MyShell。MyShell 要具备如下功能：

- 支持程序后台运行。
- 支持重定向。
- 支持管道。
- 支持设置搜索路径。
- 支持内置命令：cd(切换目录)、exit(退出 Shell)和 path(设置搜索路径)。

5.3.1.2 解决方案

1. 总体结构

根据实验要求，MyShell 可以组织成图 5-3 所示的总体结构。

2. 交互界面显示

通常 Shell 会在命令提示符中根据用户的配置文件显示响应信息。在 MyShell 中，只需要固定显示当前用户关联的登录名、主机网络名、当前目录名即可。显示格式为：

```
[username@servername:pathname]#
```

要完成该功能，需要涉及的几个函数为：

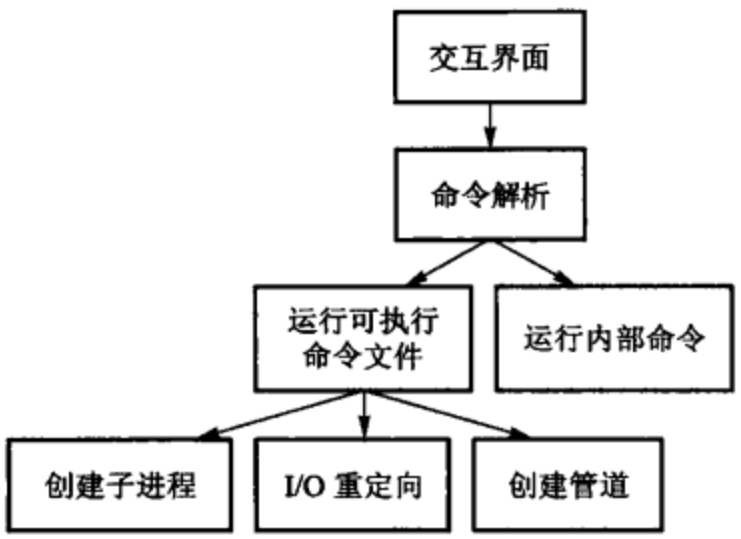


图 5-3 MyShell 总体结构

```
#include<unistd.h>
char *getlogin(void);
/*getlogin 函数返回的是与当前用户关联着的登录名*/
#include<unistd.h>
int gethostname(char *name, size_t namelen);
/*gethostname 函数把机器的网络名写到字符串 name 中。该字符串的长度至少要有 namelen 个字符。
gethostname 函数在成功时返回 0，否则返回-1*/
#include<unistd.h>
char *getcwd( char *buf, size_t size);
/*getcwd 函数的作用是把当前目录的名字写到给定的缓冲区 buf 里。如果子目录的名字超过参数 size
给出的缓冲区长度，返回 NULL。如果操作成功，它返回指针 buf*/
```

3. 命令解析

为了得到命令行，MyShell 执行一个阻塞型读操作，因此执行 MyShell 进程时将会被阻塞，直到用户根据提示符输入又一个命令行。MyShell 可以通过 gets( )获得用户输入的命令行，在获得用户的输入后，需要对输入进行解析，以获得命令和参数。在解析过程中，需要注意：

- 判断命令是否是内置命令。
- 是否包含“<”、“>”等字符，如包含，表明需要进行重定向。
- 是否包含“&”字符，如包含，表明命令要放入后台执行。
- 是否包含“|”字符，如包含，表明有多个命令，并要创建管道。

4. 内置命令

MyShell 需要处理 3 个内置命令：cd、path 和 exit。

cd 命令用于切换当前目录。可以通过 chdir()函数更改当前目录：

```
#include<unistd.h>
```

```
int chdir(const char *pathname);
/*pathname 为新目录名。如果切换成功，chdir 返回 0；否则返回–1 */
```

path 命令用于设置 MyShell 的搜索路径。在 MyShell 中，需要用一个全局变量 gpath 记录用户设置的搜索路径。当用户设置新搜索路径时，MyShell 对 gpath 变量进行更新。

exit 命令用于退出 MyShell：

如果用户输入“exit”命令，MyShell 可以通过 exit()函数退出程序：

```
#include <stdlib.h>
void exit(int status);
/*status 为程序的退出码*/
```

在 Linux 中，通常退出码为 0 时，表示程序正常退出。在 MyShell 中，用户输入 exit 命令是一个正常的退出请求，可以将退出码置为 0。

5. 执行命令

如果用户输入的不是内置命令，MyShell 需要在用户设置的路径中搜索命令，并测试该命令是否可执行。可以通过 access( )函数进行测试：

```
#include<unistd.h>
int access(const char *pathname, int mode);
```

access( )按照用户 ID 和组 ID 进行存取许可的测试。pathname 为待测试文件的路径名；mode 为需要进行的测试，常用的 mode 常数有 R_OK(测试读许可权)、W_OK(测试写许可权)、X_OK(测试执行许可权)和 F_OK(测试文件是否存在)。如果测试成功，access( )函数返回 0。在搜索时需要注意，用户可能已提供绝对路径名作为命令名单词或者根据 path 命令设置的搜索路径进行限定的相对路径名。此“/”、“./”和“../”开头的名称是可以用于启动执行的绝对路径名。否则 MyShell 需要在用户定义的搜索路径中进行查找。

在 Linux 系统中，执行一个命令通常使用 fork( )和 exec( )。MyShell 可以通过调用 fork( )创建一个子进程，通过 exec( )改变子进程当前执行的程序。

```
#include <sys/types.h>
#include<unistd.h>
pid_t fork(void);
```

由 fork( )函数创建的新进程被称为子进程。该函数被调用一次，但返回两次。两次返回的区别是子进程的返回值是 0，而父进程的返回值是新子进程的进程 ID。因为一个进程可以创建多个子进程，为了区分这些子进程，子进程需要将自己的进程 ID 返回给父进程。

新创建的进程是父进程的副本，子进程需要通过 exec( )类函数执行新功能。当进程调用一种 exec( )函数时，该进程完全由新进程替代，而新进程则从其 main( )函数开始执行。exec( )类函数并不创建新进程，而只是用另一个新进程替换当前进程的正文、数据和堆栈段。MyShell 通过调用 execve( )实现进程的替换：

```
#include <unistd.h>
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

pathname参数是将要被执行的新进程的文件的路径名，argv是一个参数字符串列表，而envp数组是一个环境变量字符串和值的列表，它们将在进程开始执行新进程时使用。execve( )如果成功不会返回，如果失败返回–1。

MyShell在执行命令时还需要考虑是否将该命令放入后台执行。如果将命令放入后台执行，父进程(MyShell)需要在fork( )成功之后，直接将命令提示符打印出来。如果需要将程序放在前台运行，则父进程需要等待子进程运行结束，可以通过执行waitpid( )函数等待子进程运行结束。

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *statloc, int options);
```

参数pid表示需要等待的子进程的进程ID。statloc为用户存储子进程退出时的退出状态，MyShell并不关心子进程的退出状态，因此可以将statloc设置为NULL。options参数一般设置为0。

执行命令的大体框架为：

```
if  ((pid=fork( ))==0 ){
    /*子进程*/
    execve(path, argv, envp);
}else{
    /*父进程*/
    if ( forground )                    /*前台运行*/
        waitpid( pid, &status, 0 );     /*等待子进程退出*/
}
```

6. I/O重定向

已经打开的文件描述符将会在fork( )和exec( )调用中保持下来，利用这一点可以实现文件的输入、输出重定向，为了说明如何实现这一点，需要了解内核用于I/O的数据结构。

每个进程描述符中包含一个files_struct结构，用来记录用户的文件打开情况，这个结构称为用户打开文件表。指向该结构的指针被保存在进程的task_struct结构的成员files中。用户打开的每一个文件都占用打开文件表中的一项，表中的每一项指向系统中的一个文件对象(file)。进程在进行I/O操作时，需要提供文件描述符。文件描述符是用户打开文件表的索引，内核通过文件描述符查找到相应的文件对象。例如，用户调用write(1,buf,count)时，内核会查找到文件描述符表中的第1项所对应的文件对象，并对相关文件进行写操作。

内核为系统中每个打开文件创建一个文件对象，每一个文件对象用一个file结构表示。每一个文件对象中包含文件偏移量，读写权限等信息。每当进程打开一个文件时，内核从files_struct结构中找一个空闲的文件描述符，使它指向文件对象file。当调用fork( )生成一个子进程时，子进程复制父进程的文件描述符，两者有相同的用户打开文件表，都有表项指向同一个file结构。当调用exec( )时，文件描述表也不会发生变化。因此子进程能够继承其父进程的文件描述符。图5-4显示父进程与子进程及打开文件表的关系。

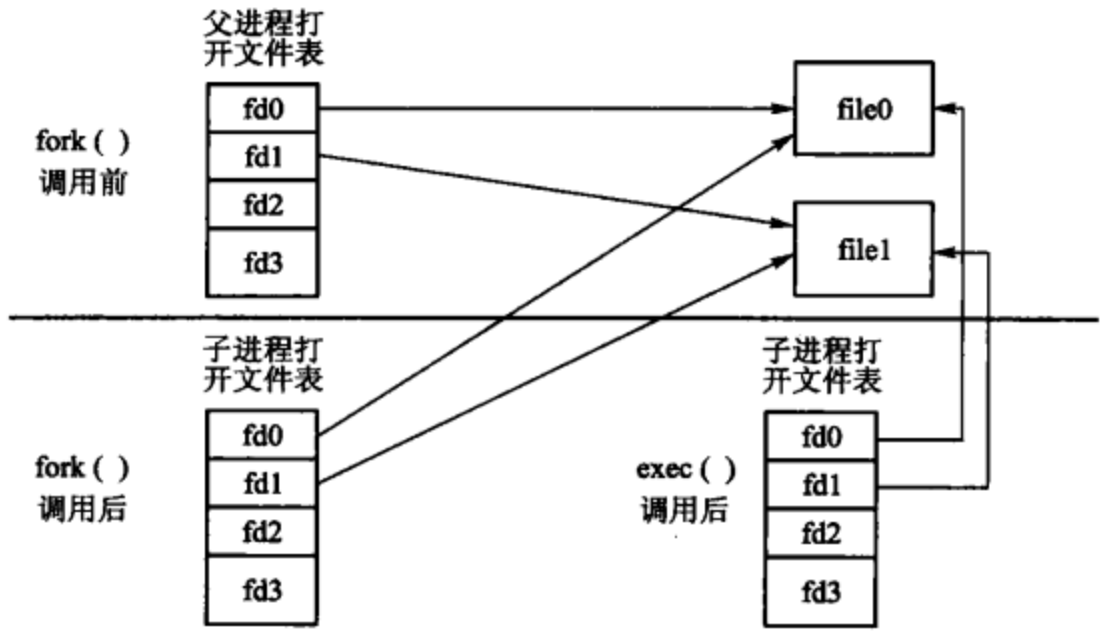


图 5-4 父进程、子进程与打开文件表关系

由于文件描述符是指向文件对象的指针，因此不同的文件描述符可以指向同一个文件对象，如图 5-5 所示。在没有进行 I/O 重定向时，文件描述符 0 指向标准输入，即指向键盘。在实现输入重定向时，程序需要先打开一个文件，内核会为该文件创建一个文件对象，然后，程序将文件描述符 0 指向已打开文件对应的文件对象。在这之后，程序通过标准输入(文件描述符 0)读取的数据便来自先前打开的文件。输出重定向的实现方法和输入重定向类似。

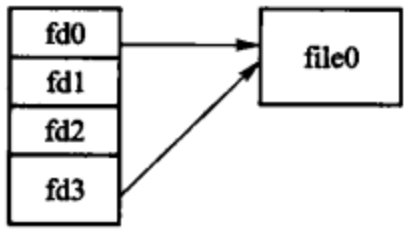


图 5-5 不同文件描述符指向同一文件对象

完成 I/O 重定向可以使用的函数有：

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

dup( )函数将指定的文件描述符 filedes 复制到当前第 1 个可用的文件描述符中。如果需要使用 dup( )函数完成 I/O 重定向，需要先将标准输入或输出关闭，例如：

```
fid = open( foo, O_WRONLY|O_CREAT);
close(1);
dup(fid);
```

在该段代码中，进程先打开准备用于输出重定向的文件，然后关闭标准输出。这时使用 dup( )

函数将把 fid 复制到标准输出的文件描述符中，其文件描述符表的变化如图 5-6 所示。

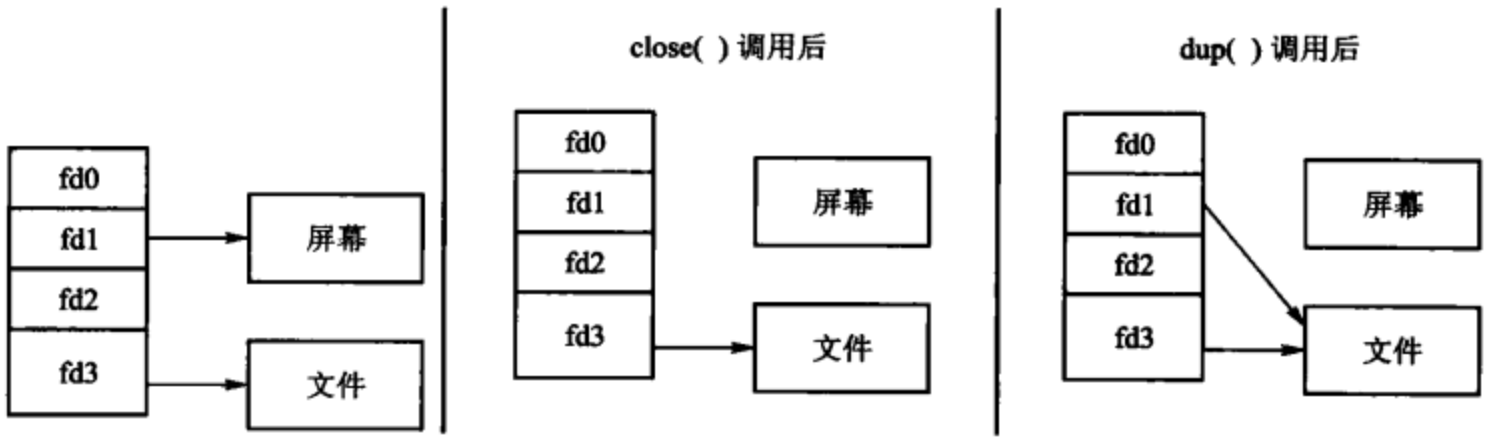


图 5-6 文件描述符变化过程

使用 close( )和 dup( )函数完成重定向是不安全的。因为在 close( )调用之后、dup( )调用之前，可能产生信号，在信号处理过程中可能修改文件描述符，这将导致 dup( )函数中的文件描述符复制错误。为了解决这个问题，需要使用 dup2( )函数，dup2( )函数将 filedes 文件描述符复制到 filedes2 文件描述符中。如果 filedes2 没有关闭，dup2( )函数将先把 filedes2 关闭。dup2( )函数是一个原子操作，因此在该函数的执行过程中不会被信号打断，能够避免使用 close( )和 dup( )函数可能产生的问题。

MyShell 实现 I/O 重定向的主要流程为：

```
if  ((pid=fork( ))==0 ){
        /*子进程*/
        /*实现输出重定向*/
        fid = open( foo, O_WRONLY|O_CREAT);
        close(1);
        dup(fid);
        execve(path, argv, envp);
}else{
        /*父进程*/
        if ( forground )                    /*前台运行*/
              waitpid( pid, &status, 0 );   /*等待子进程退出*/
}
```

7. 管道

管道在使用时有如下两种限制：

- 管道是半双工的，数据只能在一个方向上流动。
- 管道只能在具有公共祖先的进程之间使用。

通常，一个管道由一个进程创建，然后该进程调用 fork( )，此后父、子进程之间就可以使用管道，管道是调用 pipe( )函数创建的：

```
#include <unistd.h>
int pipe( int fd[2] );
```

经由参数 fd 返回两个文件描述符：fd[0]为读而打开，fd[1]为写而打开。fd[1]的输出是 fd[0]的输入。pipe( )函数调用成功后，函数返回 0，否则返回–1。

单个进程中的管道几乎没有任何用处。通常，调用 pipe( )的进程接着调用 fork( )，这样就创建了从父进程到子进程或反之的管道。图 5-7 显示了这种情况。

使用 fork( )函数之后做什么取决于所需要的数据流的方向。对于从父进程到子进程的管道，父进程关闭管道的读端(fd[0])，子进程则关闭写端(fd[1])，图 5-8 显示描述符的最后安排。

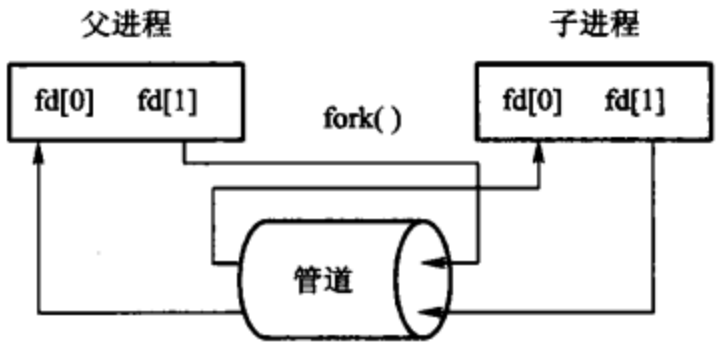


图 5-7 使用 fork( )函数之后的半双工管道

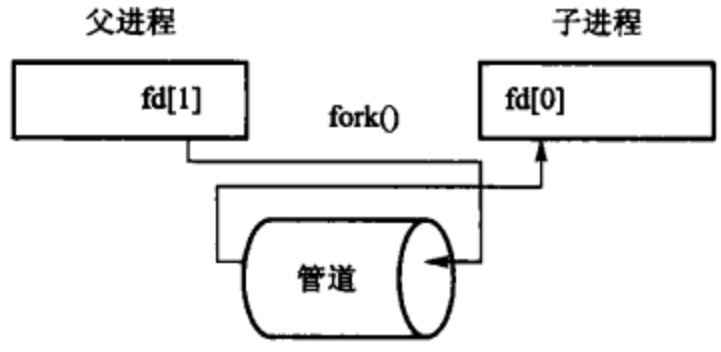


图 5-8 从父进程到子进程的管道

MyShell 需要实现的管道是将进程 1 的标准输出变成进程 2 的标准输入。实现该功能需要的步骤为：

① 创建进程 1。

② 在进程 1 中调用 pipe( )创建管道，并得到管道描述符 fd[0],fd[1]。

③ 在进程 1 中创建进程 2，使进程 2 成为进程 1 的子进程。

④ 进程 1 关闭描述符 fd[0]。

⑤ 进程 1 调用 dup2(fd[1],1)，将管道描述符复制到标准输出。

⑥ 进程 1 调用 exec( )执行新程序。

⑦ 进程 2 关闭描述符 fd[1]。

⑧ 进程 2 调用 dup2(fd[0],0)，将管道描述符复制到标准输入。

⑨ 进程 2 执行 exec( )执行新程序。

完成上述步骤后，进程 1 的标准输出将自动通过管道传递到进程 2 的标准输入中。代码的框架如下：

```
{   …
    int fd[2];
    pipe(fd);
    if ( fork( )>0 ) {
        /*父进程*/
        close(fd[0]);
```

```
            dup2(fd[1],1);
            exec(…);
        }else{
            close(fd[1]);
            dup2(fd[0],0);
            exec(…);
        }
        …
}
```

5.3.1.3 程序框架

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

void init( );
void setpath(char* newpath);                                    /*设置搜索路径*/
int read_command( );                                            /*获得用户输入*/
int is_internal_cmd(char*cmd, int cmdlen);                      /*解析内部命令*/
int is_pipe(char*cmd, int cmdlen);                              /*解析管道*/
int is_io_redirect(char*cmd, int cmdlen);                       /*解析重定向*/
int normal_cmd(char*cmd, int cmdlen, int infd, int outfd, int fork);   /*执行普通命令*/

void init( )
{
   /*设置命令提示符*/
   /*设置默认的搜索路径*/
   setpath("/bin:/usr/bin");
}

int read_command( )
{
   /*读取用户命令，如果用户输入 exit，程序退出*/
}

void setpath(char* newpath)
{
```

```
  /*解析路径，并设置搜索路径*/
}

int is_internal_cmd(char* cmd, int cmdlen)
{
  /*判断是否是内部命令(cd，path)*/
}

int is_pipe(char* cmd, int cmdlen)
{
  /*为命令创建管道*/
}

int is_io_redirect(char* cmd, int cmdlen)
{
   /*进行 IO 重定向*/
}

/*搜索命令*/
char* find_exe(const char* exepath)
{
   /*在已设置的目录中搜索命令*/
}
/*执行命令*/
void fork_and_exec( char* path, char* argv[], int is_background, int infd, int outfd)
{
    …
}
/*解析命令参数*/
char** parse_argv(char* path)
{
     /*解析命令参数*/
}
/*执行一个普通命令*/
int normal_cmd(char* cmd, int cmdlen, int infd, int outfd, int fork)
{
  …
}
int main( int argc, char *argv[] )
{
```

```
    init( );
    while ( read_command( ) ){
      if ( is_internal_cmd(cmdbuf, cmdlength) )
        continue;
      if ( is_pipe(cmdbuf, cmdlength) )
        continue;
      if ( is_io_redirect(cmdbuf, cmdlength) )
        continue;
      normal_cmd(cmdbuf, cmdlength, –1, –1, 1);
    }
    return 0;
}
```

5.3.2 实验 2 基于 Shell 的网络管理

5.3.2.1 实验说明

该实验要求实现两个基本功能：

- 通过 Shell 编程访问某个 URL(如 www.nju.edu.cn)*N* 次。
- 在网关节点捕获指定计算机的 MAC 地址(如 00:11:5B:A3:65:F8)，并限制其访问。

5.3.2.2 解决方案

对 URL 的访问可通过方法 get 实现，而访问次数可通过 while 语句实现。对 arp 地址的获取可通过 arp 实现，对指定 MAC 地址的访问则可通过 grep 得到。而对 IP 地址的提取可通过 awk 和 sed 工具实现。

5.3.2.3 程序框架

```
/*****基于 Shell 的网络管理程序框架*****/
url="http://cs.nju.edu.cn /"
looptime=0
loopcount=N
/*变量置初值*/
echo "$(date | cut –c 10-18)"
/*打印开始时间*/
while [ $looptime –lt $loopcount ]
do
        content=$(GET    $url )
        let "looptime = $looptime + 1"
done
```

```
/*通过一个 while 循环，访问 url 变量对应的网页 loopcount 次*/
echo "$(date | cut –c 10-18)"
echo "done"
/*打印结束时间*/
arp –a | grep "00:11:5B:A3:65:F8" > ctx
/*第 1 句是得到对应 MAC 地址为“00:11:5B:A3:65:F8”的包含其 IP 地址的字符串*/
awk '{printf $2}' ctx   > ipaddress
sed –n 's/(// w temp' ipaddress
sed –n 's/)// w pureip' temp
/*使用 awk 和 sed 工具，提取出 IP 对应的字符串*/
RES_IP= 'cat pureip'
echo $RES_IP
/*将 IP 地址赋值给 RES_IP 变量，并打印在屏幕上*/
iptables –A FORWARD –s $RES_IP –d $RES_IP   –j REJECT
/*使用 iptables 工具，将 FORWARD 链置位 reject(包括到该 IP 的和从该 IP 出来的包)*/
rm temp
rm ipaddress
/*删除两个临时变量*/
```

第6章　页面替换算法

6.1　实 验 目 的

- 了解存储管理的基本目的和功能。
- 理解实存管理的原理和实现技术。
- 理解虚存管理的原理和实现技术。
- 通过编程模拟实现请求分页式虚存管理和替换算法。

6.2　背 景 知 识

6.2.1　存储管理的目的和功能

计算机系统采用多道程序设计技术后，要在主存中同时存放多个作业的程序，而这些程序在主存中的位置是无法预先知道的，在编写程序时不能使用绝对地址。计算机的指令中地址部分所指示的地址通常是逻辑地址，可从0开始编号，用户则按逻辑地址编写程序。当要把程序装入计算机时，首先由操作系统为其分配一个能容纳的主存空间。由于逻辑地址通常与分配到的主存空间的绝对地址不一致，而CPU是按绝对地址执行指令的，所以必须把逻辑地址转换为绝对地址，才能得到信息的真实存放位置。此外，多个作业共享主存空间时必须对其中的程序和数据进行保护，并进行合理有效的移动，以充分发挥主存空间的利用率。为了方便用户编制程序，使用户编写程序时不受主存实际容量的限制，可以采用一些技术“扩充”物理主存容量，使用户得到比实际容量大的主存空间。可见，存储管理的目的是：方便用户使用和提高主存利用率。具体地说，存储管理的功能有：

- 主存分配。为作业的程序和数据分配主存单元，涉及逻辑地址到绝对地址的转换。使用完毕进行回收。
- 主存保护。隔离分配给作业的主存区，使之互不干扰，按存取权限把合法区与非法区分开，实现存储保护。
- 主存共享。允许多个作业共享主存空间或共享主存的某些区域，提高主存空间利用率。
- 主存扩充。允许用户编程空间大于物理主存容量，由系统实现主存“扩充”。

6.2.2　存储管理涉及的基本概念

6.2.2.1　程序的编译、链接、装入和执行

应用程序大都使用高级程序设计语言或汇编语言编写，所编写的程序称为源程序。源程序中的符号名集合限定的空间称程序命名空间，源程序是不能被计算机直接执行的，需要通过如图 6-1 所示的 3 个阶段处理后才能装入主存执行。

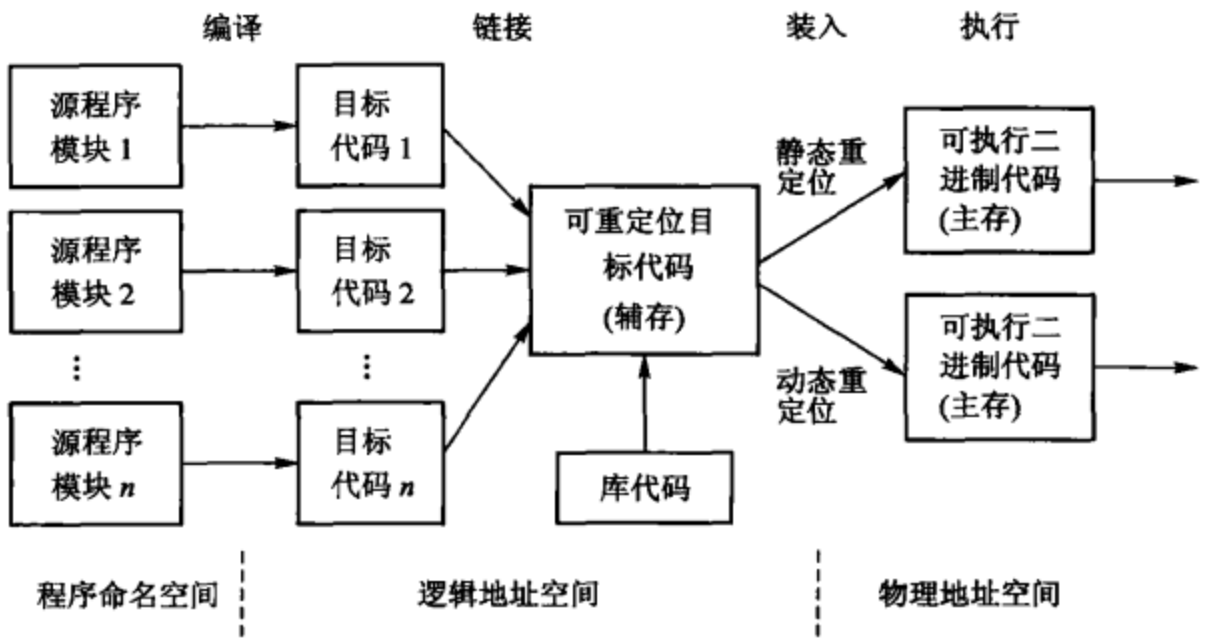


图 6-1　程序的编译、链接、装入和执行

1. 编译

源程序经过编译程序或汇编程序处理来获得目标代码(也称目标模块)，一个程序可由许多独立编写且具有不同功能的源程序模块组成。在 C 程序设计模型中，至少分 3 个程序模块：文本段、数据段和堆栈段。由于模块会包含外部引用，即指向其他模块中的数据或指令的操作数地址，或包含对库函数的引用，编译或汇编程序负责记录引用发生的位置，处理后产生相应的多个目标代码模块。每个都附有供引用使用的内部符号表和外部符号表。符号表中依次给出每个符号名及在本目标代码模块中的名字地址，在模块被链接时进行转换。

2. 链接

链接程序的作用是把多个目标代码模块链接成一个完整的可重定位目标程序(其中包括应用程序要调用的标准库中的函数，引用的其他模块中的子程序)，需要解析内部和外部符号表，把对符号名字的引用转换为数值引用，要将每个涉及名字地址的程序入口点和数据引用点转换为数值地址。可重定位目标程序又称装入代码模块，它被存放在磁盘中，由于程序在主存中的位置不可预知，链接时程序地址空间中的地址总是相对某个基准(通常为 0)开始编号的顺序地址，称为逻辑地址或相对地址。

3. 装入

在装入一个可重定位目标程序之前，存储管理程序总会分配一块实际主存给进程。装入程序根据指定的主存块首地址，再次修改和调整被装入模块中的每个逻辑地址，将逻辑地址绑定到物理地址，使之成为可执行二进制代码。这样，就可使用逻辑地址来引用分配到的主存物理块内相应的物理地址，将再次修改和调整的可重定位目标程序复制到指定主存块中，以便进程在物理地址空间中执行。

在磁盘中的可重定位目标程序使用的是逻辑地址，它的逻辑地址集合称为该进程的逻辑地址空间。逻辑地址空间可以是一维的，这时逻辑地址限制在从 0 开始顺序排列的地址空间内；也可以是二维的，这时整个进程被分为若干段，每段有不同的段号，段内地址从 0 开始顺序编址。进程运行时，它的可重定位目标程序将被装入物理主存地址空间中，此时程序和数据的实际地址一般不可能与原来的逻辑地址一致。物理主存中从统一的基地址开始顺序编址的存储单元称为物理地址或绝对地址，物理地址的总体构成物理地址空间，注意，物理地址空间是由存储器地址总线扫描出来的空间，其大小取决于实际安装的主存容量。

6.2.2.2 逻辑地址和物理地址

用户(目标)程序使用的地址空间中的每个地址单元称逻辑地址。由于逻辑地址通常相对于程序的起始地址，故也称相对地址；主存物理地址空间中的每个地址单元称物理地址或绝对地址，绝对地址可直接寻址。

6.2.2.3 静态分配和动态分配

作业所需主存空间是在装入时分配的，在其整个运行期间，一直占用且不能再申请新的主存空间，也不允许作业在主存中“移动”，称静态存储分配，简称静态分配。允许运行中的作业继续申请附加的主存空间，系统也可根据需要将程序或数据从主存的一个区域移动到另一个区域，以及从主存调至外存对换区或反之，称动态存储分配，简称动态分配。

6.2.2.4 静态重定位和动态重定位

作业的地址空间与物理空间不一致时，进行地址调整以便作业能够运行的过程称为重定位，其实质是地址映射，即作业地址空间中的逻辑地址转换为主存空间的物理地址。

由装入程序实现可重定位目标程序的装入和地址转换，把它装入分给进程的主存指定区域，其中的所有逻辑地址修改成主存物理地址，称静态重定位(static relocation)。地址转换工作在进程执行前一次完成，作业执行期间不再进行地址修改，也不允许作业在主存中移动。

由装入程序实现可重定位目标程序的装入，把它装入分给进程的主存指定区域，程序主存起始地址被置入硬件专用寄存器——重定位寄存器，程序执行过程中，每当 CPU 引用主存地址(访问程序和数据)时，由硬件截取该逻辑地址，并且在它被发送到主存之前加上重定位寄存器

的值，以实现地址转换，称动态重定位(dynamic relocation)，地址转换推迟到最后可能的时刻，即进程执行时才完成。

6.2.2.5 实存管理和虚存管理

如果在作业被启动运行之前，由操作系统为它分配足够的主存空间，来装入其全部信息以便运行，则称为物理存储管理（简称实存管理）。如果在作业被启动运行之前，仅将当前使用部分先装入主存，其余部分存放在磁盘，待用到时由系统自动把它们装入以便运行，则称为虚拟存储管理（简称虚存管理）。

6.2.3 实存管理的原理和实现技术

1. 固定分区

预先把可分配的主存空间分割成若干个连续区域，称分区。每个分区的大小可以不同，但划分后却固定不变，每个分区装入一个作业，这是支持多道程序的最简单的主存管理方法。

2. 可变分区

可变分区是指主存不预先划分，当作业装入时，根据其需求和主存空间的使用情况决定如何分配。若有足够的空间，则按需分割一部分分区给该作业，否则令其等待主存资源。可变分区的主存分配算法有最先适应算法、最佳适应算法、最坏适应算法和下次适应算法等。

3. 移动技术

把已在主存中的进程分区紧挨到一起，使分散的空闲区汇集成片，就是移动，也叫主存紧凑。

4. 覆盖技术

覆盖是一个作业的若干个程序段，或几个作业的某些部分共享同一主存空间，其基本思想是把主存的同一区域分配给一道程序的若干个子程序或数据段。开始时只有程序的一部分装入主存，在其执行过程中，根据请求动态地把其他部分装入到该程序原来已经占用过的存储区域中。

5. 对换技术

对换是把主存中的信息以文件形式写入磁盘，再将指定的信息从磁盘读入主存，并将控制转给它。

6. 分页存储管理

采用分页存储管理允许程序存放到若干不相邻接的空闲块中，既可免去移动信息工作，又可充分利用主存空间，消除动态分区法中的“碎片”问题，从而提高主存空间的利用率。下面介绍分页存储管理的基本原理。

- 页面：进程逻辑地址空间分成大小相等的区，每个区称页面或页，页号从 0 开始依次编号。
- 页框：又称页帧，把主存物理地址空间分成大小相等的区，区的大小与页面大小相等，每个区为一个物理块或页框，块号从 0 开始依次编号。

• 逻辑地址：逻辑地址由页号和页内位移两部分组成。

• 页表：操作系统为进程建立的、用来记录程序逻辑地址空间和程序在主存物理地址空间对应关系的对照表。使用页表的目的是把页面映射为页框。

• 地址转换：绝对地址=页框号×块长+页内位移。

• 快表：为提高分页式地址访问速度，在地址变换机制中增加的小容量联想存储器，用来存放部分页表。如果快表命中，只要访问主存一次即可存取一个数据。

7. 分段存储管理

系统允许用户对作业按逻辑关系进行分段，各段大小可以不同。逻辑段内的地址由“段号：段内位移”组成，形成一个用户定义的二维地址空间，以满足用户模块化程序设计的要求。主存分配以段为单位，一个作业的地址空间可分配在若干个互不连续的主存区域。分段存储管理的实现，需要有段表和地址转换机制的支持。

8. 段页式存储管理

结合分段管理在逻辑上的优点以及分页管理在物理上的优点，用分段来分配和管理逻辑地址空间，用分页来分配和管理物理存储空间，便形成段页式存储管理。具体做法是把作业分段，段内分成逻辑页面，主存分成物理页框，每一段不再占有连续的主存，而是被分为若干个页面，所以段页式存储管理实际上是对页面进行分配和管理。

在段页式存储管理系统中，作业的地址空间被分为 3 部分(S、P、W)。从逻辑地址空间到物理地址空间的变换是通过段表和页表共同来实现的。系统为每个作业建一张段表 ST，对应段表中的每一段再建立一张页表 PT。地址变换要 3 次访问主存：一次是访问 ST 段表，一次是访问 PT 页表，再一次是按位移 W 访问物理地址。为此，也可采用快表的方法来加快地址变换过程。

9. 存储共享和保护

存储共享与保护包括页面共享和段共享等方面。

• 页面共享。必须区分数据共享和程序共享。实现数据共享时，允许不同进程对共享的数据页用不同的页号，只要让各自页表中的有关表项指向共享的数据页框；实现程序共享时，由于指令包含指向其他指令或数据的地址，进程依赖于这些地址才能执行，所以，不同进程中要正确执行共享代码页面，必须为它们在所有逻辑地址空间中指定同样页号。

• 段的共享。通过两个作业段表中相应表项都指向被共享段的同一个物理副本来实现。

• 分页和分段存储保护。在多道程序运行的情况下，必须由硬件和软件相互配合，以保证每道程序只是使用自己的存储区域，防止用户之间的存储空间互相干扰，这种措施就是存储保护。分页和分段均可采用两种方式实现存储保护，一是地址越界保护，通过地址转换机制中的页表或段表长度的值与所访问的逻辑地址相比较完成；二是通过页表或段表中的访问控制位对主存中的信息提供保护。为了实现存储保护，硬件提供的基本保护支撑机制有：上、下界寄存器，基址、限长寄存器和钥匙/存储键。

6.2.4 虚存管理的原理和实现技术

6.2.4.1 虚拟存储器

在具有层次结构存储器的计算机系统中，利用大容量辅存(磁盘)来扩充主存，采用由系统自动实现“部分装入”和“部分替换”功能，能从逻辑上为用户提供一个比物理主存容量大得多的，可寻址的一种“主存储器”，称虚拟存储器(virtual memory)或虚拟主存，简称虚存。虚存允许用户程序以逻辑地址寻址，而不必考虑物理主存的大小，实际上虚存管理对用户隐蔽可用物理存储器的容量和操作细节，主存可看作是虚存的缓冲区，这种将物理空间和辅存空间分开编址但又统一使用的技术，既为用户编程提供了极大方便，又进一步提高了主存利用率和系统吞吐量。

6.2.4.2 虚存的基本概念

1. 程序局部性原理

一个进程的程序和数据的访问都有聚集成群的倾向。某存储单元被使用，其相邻存储单元很快也被使用，称空间局部性(Spatial Locality)，或者最近访问过的程序代码和数据，很快又被访问，称时间局部性(Temporal Locality)。基于这个原理，在任何时刻，只有进程的几个片段在主存中，因此主存中可以装入更多进程，甚至进程的程序总量比主存还要大；同样原理，也可预测在较短时间的将来会用到进程的哪些片段或不再使用哪些片段，以便作出调度。

2. 颠簸

颠簸是指 CPU 把大部分时间花在调入调出进程的片段上，而不是执行用户指令和访问数据。

3. 缺页中断率

如果作业执行中访问页面的总次数为 A，其中有 F 次访问的页面尚未装入主存，则有 F 次缺页中断，$f=F/A$，f 被称为缺页中断率。影响缺页中断的因素有：分配给作业的主存块数、页面大小、程序局部性程度和页面替换算法。

4. 工作集

工作集是指一个时间段中，作业运行所需访问的所有页面集合。

5. 虚实地址转换

逻辑地址是从进程角度看到的逻辑主存单元，逻辑地址空间是程序员的编程空间；物理地址是从 CPU 角度看到的物理主存单元，物理地址空间是程序的执行地址空间；进程中所有的主存访问都用逻辑地址，在执行时，逻辑地址被硬件动态地址转换机构 MMU 解释为虚拟地址，并再转化为物理地址。虚存空间的大小受到 CPU 地址线位数和辅存容量的限制，每个作业可用的虚存大小等同于实际物理主存大小加部分磁盘区域组成的存储空间容量。

支持虚存技术的两种基本方法是请求分页式和请求分段式，进程在主存中的存放，可分成若干片段(分页或分段)，这些片段不需要连续地存放。只要在主存中包含下一个将被取出的指

令和将要访问的数据的片段，那么至少在一段时间里进程可以继续执行下去，如果 CPU 访问到一个不在主存里的地址，将产生一个主存访问错误的缺页或缺段异常。所以虚拟存储器具有主存离散分配、作业部分装入、允许换进换出等特点。

6.2.4.3 请求分页虚存管理

1. 实现原理

把作业的所有分页副本存放在磁盘中，当它被调度投入运行时，首先把当前需要的页面装入主存，之后根据程序运行的需要，动态装入其他页面；当主存空间已满，而又需要装入新页面时，根据某种算法淘汰某个页面，以便装入新页面。因此，在页表中必须说明哪些页已在主存，存放在什么位置；哪些页不在主存，它们的副本在磁盘中的什么位置。还可设置页面是否被修改过，是否被访问过，是否被锁住等标志供淘汰页面使用。

在地址映射过程中，若页表中发现所要访问的页不在主存，则产生缺页异常，操作系统接到此信号后，就调出缺页异常处理程序，根据页表中给出的磁盘地址，将该页面调入主存，使作业继续运行下去。如果主存中有空闲块，则分配一个页框，将新调入页面装入，并修改页表中相应页表项的驻留位及相应的主存块号；若此时主存中没有空闲块，则要淘汰某页面，若该页在此期间被修改过，要将其先写回磁盘，这个过程称为页面替换。

2. 页面替换算法

页面替换算法是用来确定应该淘汰哪个页面的算法，一个好的算法应减少和避免“颠簸”现象。常见的页面替换算法有以下几种。

(1) 最佳页面替换算法(OPT)

淘汰以后不再需要的或最远的将来才会用到的页面。

(2) 先进先出页面替换算法(FIFO)

淘汰最先调入主存的页面，或者说在主存中驻留时间最长的那一页。

(3) 最近最少用页面替换算法(LRU)

淘汰在最近一段时间里较久未被访问的页面。它是根据程序执行时所具有的局部性来考虑的，即那些刚被使用过的页面可能马上还要被使用，而那些在较长时间里未被使用的页面一般可能不会马上使用。

(4) 最近未使用页面替换算法(NRU)

选择在最近一段时间内未使用过的页面并淘汰。

(5) 最不常用页面替换算法(LFU)

如果对应每个页面设置一个计数器，每当访问一页时就使它对应的计数器加 1。过一定时间后，将所有计数器全部清 0。当发生缺页异常时，可选择计数值最小的对应页面淘汰，显然它是在最近一段时间里最不常用的页面。

(6) 第二次机会页面替换算法(SCR)

最先进入主存的页面，如果最近还在被使用(其“引用位”总保持为 1)，仍然有机会像新调

入页面一样留在主存中。这是把 FIFO 算法与页表中的“引用位”结合起来使用的一种算法。

(7) 时钟页面替换算法(Clock)

类似 SCR 算法，仍然要使用页表中的“引用位”，把进程已调入主存的页面链接成循环队列，用指针指向循环队列中下一个将被替换的页面，形成类似于钟表面的环形表，队列指针相当于钟表面上的表针，这就是时钟页面替换算法的得名。

(8) 工作集页面置换算法(WSR)

引入滑动窗口概念，并不向前查看页面引用串，而是基于程序局部性原理向后看。这意味着，在任何给定时刻，一个进程在不久的将来所需主存页框数，可通过考查其过去最近的时间内的主存需求做出估计。

(9) 缺页频率页面替换算法(PFF)

根据连续的缺页之间的时间间隔来对缺页频率进行测量，每次缺页时，利用测量时间调整进程工作集尺寸。其规则是：如果本次缺页与前次缺页之间的时间超过临界值 τ，那么，所有在这个时间间隔内没有引用的页面都被移出工作集。

6.2.4.4 请求分段虚存管理

把作业的所有分段副本存放在磁盘中，当作业被调度投入运行时，首先把当前需要的一段或几段装入主存，在执行过程中访问到不在主存的段时，再把它们动态地装入。因此，在段表中必须说明哪些段已在主存，存放在什么位置，段长是多少；哪些段不在主存，它们的副本在磁盘中的位置。还可设置其是否被修改过，是否能移动，是否可扩充，能否共享等标志。

当被访问段不在主存时，将产生缺段异常。系统在处理缺段异常时，首先在主存中查找是否有足够大的空闲分区存放该段，如果有则将该段调入空闲分区，否则，检查空闲区的总和。如果能满足需要则对空闲分区进行拼接处理，以形成一个合适的空闲区，再调入该段。如果空闲区总和仍不能满足需要，则必须置换出主存中的一个或几个分段，再进行拼接，然后调入该分段。

6.3 实 验 内 容

6.3.1 实验 1 模拟实现动态分区存储管理

6.3.1.1 实验说明

编写程序实现动态分区存储管理方式的主存分配与回收。具体内容包括：首先确定主存空间分配表；然后采用最优适应算法完成主存空间的分配与回收；最后编写主函数对所做工作进行测试。

6.3.1.2 解决方案

动态分区管理方式预先不将主存划分成几个区域，而把主存除操作系统占用区域外的空间看作一个大的空闲区。当作业要求装入主存时，根据作业需要主存空间的大小查询主存内各空闲区。当从主存空间中找到一个大于或等于该作业大小的主存空闲区时，选择其中一个空闲区，按作业要求划出一个分区并装入该作业。作业执行完后，它所占用的主存空间被收回，成为一个空闲区。如果该空闲区的相邻分区也是空闲区，则需要将相邻空闲区合并成一个空闲区。

实现动态分区的分配与回收，主要考虑两个问题：第一，设计记录主存使用情况的数据结构，用来记录空闲区和作业占用的区域；第二，在该数据结构基础上设计主存分配算法和主存回收算法。

由于动态分区的大小是由作业需求量决定的，故分区的长度预先不能固定，且分区的个数也随主存分配和回收变动。总之，所有分区情况随时可能发生变化，数据表格的设计必须和这个特点相适应。由于分区长度不同，因此设计的表格应该包括分区在主存中的起始地址和长度。由于分配时，空闲区有时会变成两个分区(空闲区和已分配区)，回收主存分区时，可能会合并空闲区，这样如果整个主存采用一张表格记录已分配区和空闲区，就会使表格操作繁琐。主存分配时查找空闲区进行分配，然后填写已分配区表，主要操作在空闲区；某个作业执行完后，将该占用分区变成空闲区，并将其与相邻的空闲区合并，主要操作也在空闲区。由此可见，主存的分配与回收主要是对空闲区的操作。这样为了便于对主存空间的分配与回收，可建立两张分区表记录主存使用情况：“已分配区表”记录作业占用分区，“空闲区表”记录空闲区。

然后在数据结构上进行主存的分配，其主存分配算法采用最优适应算法或首次适应算法，即按作业要求挑选一个能满足作业要求的最小空闲区分配。具体实现时，把空闲区按长度以某种方式(如递增方式)登记在“空闲区表”中，分配时顺序查找“空闲区表”，查找到的第一个空闲区就是满足作业要求的最小分区。在实现回收时，先在“已分配区表”中找到将作业归还的区域，且变为空，检查“空闲区表”中未分配区域，查找是否有相邻空闲区，最后合并空闲区，修改“空闲区表”。设计程序时可选择进行主存分配或主存回收，所需参数为：若是主存分配，输入作业名和所需主存空间大小；若是回收，输入回收作业的作业名，以循环进行主存分配和回收。

6.3.1.3 程序框架

```
/*****可变分区分配回收模拟程序 memory.c*****/
    #define n 10        /*定义系统允许的最大作业数*/
    #define m 10        /*定义系统允许的空闲区表最大值*/
    #define minisize 100
    struct {            /*已分配区表的定义*/
      float address;
      float length;
```

```
   int flag;
}used_table[n];
struct {          /*空闲区表的定义*/
 float address;
 float length;
 int flag;
}free_table[m];

allocate(char J,float xk)
{                 /*主存分配函数*/
   int i,k;
   float ad;
   k=-1;
   /*遍历 free_table 查找可用区域*/
   /*检索 free_table 进行最优分配*/
   /*修改已分配区表 used_table*/
}

reclaim(char J)
{       /*主存回收函数开始*/
  /*从 used_table 中寻找已分配区表中对应项 s*/
   /*修改已分配区表 used_table*/
}

main( )
{
   /*空闲区表初始化*/
   /*已分配区表初始化*/
   while(1) {
   printf("选择功能项(0-退出,1-分配主存,2-回收主存,3-显示主存)\n");
   printf("选择功能项(0~3):");
   scanf("%d",&a);
   switch(a){
      case 0:
        exit(0);
      case 1:
         printf("输入作业名 J 和作业所需长度 xk:");
         scanf("%*c%c%f",&J,&xk);
         allocate(J,xk);
         break;
```

```
        case 2:
          printf("输入要回收分区的作业名\n");
          scanf("%*c%c",&J);
          reclaim(J);
          break;
        case 3:
          printf("输出空闲区表:\n 起始地址   分区长度   标志\n");
          for(i=0;i<m;i++)
            printf("%5.0f%10.0f%6d\n",free_table[i].address,free_table[i].length,
                  free_table[i].flag);
          printf("按任意键,输出已分配区表\n");
          getch( );
          printf("输出已分配区表:\n 起始地址   分区长度   标志\n");
          for(i=0;i<n;i++)
            if(used_table[i].flag!=0)
                printf("%6.0f%9.0f%6c\n",used_table[i].address,used_table[i].length,
                  used_table[i].flag);
            else
                printf("%6.0f%9.0f%6d\n",used_table[i].address,used_table[i].length,
                  used_table[i].flag);
break;
          default:
            printf("没有该选项\n");
          }
        }
      }
```

6.3.2 实验 2 模拟实现请求分页虚存页面替换算法

6.3.2.1 实验说明

实现虚存页面替换算法的模拟程序应该含有以下功能：

- 接收用户输入参数，包括程序长度(页面数)、页框个数及页面大小。
- 程序结果采用不同颜色区分命中、替换及直接加入空闲块。
- 实现 OPT、FIFO、LRU、LFU、SCR、Clock 等替换算法。

6.3.2.2 解决方案

在请求分页虚存页面替换算法中，为实现从请求页到主存块的置换，需要在模拟程序中维护两个数据结构，即请求页面队列和主存块队列。其中请求页面队列为进程所用，记录当前进

程请求的页面块信息。而主存块队列由系统维护，该队列保存当前系统中各主存块的状态(包括最后访问时间、闲忙状态等)。各种替换算法将以这两个数据结构为基础，在系统中为用户请求寻找物理块。

6.3.2.3 程序框架

```
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include <sys/time.h>
#include <unistd.h>

#define BUSY 1
#define IDLE 0
/*#define _DEBUG 1*/

typedef struct _Page{                    /*页面*/
      int pageID;                        /*页号*/
} Page;                                  /*进程*/

typedef struct _PageQueue{               /*页面队列*/
   Page page;
   struct _PageQueue* next;              /*下一页面*/
} PageQueue;

typedef struct _Block{                   /*块记录结构*/
   Page *page;                           /*页面*/
   long time;                            /*最后访问时间*/
   int state;                            /*页块是否空闲，0代表未使用，1代表已被使用*/
} Block;                                 /*页块链表*/

typedef struct _BlockQueue{              /*块队列*/
   Block block;
   struct _BlockQueue *next;
} BlockQueue;

typedef struct _process{                 /*进程结构*/
   PageQueue pages;                      /*页面*/
   unsigned int pageLength;              /*页面数*/
}process; /*进程
```

```
long GetSystemUtime( )
{                          /*获取系统当前时间的微秒数*/
  struct timeval nowit;
     gettimeofday(&nowit,NULL);
  return 1000000 * nowit.tv_sec + nowit.tv_usec;
}

void* InitializeBlockQueue(unsigned int size)
{            /*初始化主存块，把首地址返回。如果分配失败，返回 NULL*/
  for(i = 0; i < size; i++){
     /*申请大小等于 BlockQueue 的空间*/
     /*初始化块结构内部元素值*/
     /*建立 BlockQueue 链表*/
  }
  /*返回链表头*/
}

int GetBlockQueueSize(BlockQueue *blockQueue, int listType)
{              /*获取块长度*/
  if (单向链表){
     while (presentBlock != NULL){
           blockQueueSize++;
           presentBlock = presentBlock->next;
     }
  } else {
     while (presentBlock->next != blockQueue){
           blockQueueSize++;
           presentBlock = presentBlock->next ;
     }
     blockQueueSize++;
  }
  return blockQueueSize;
}

void ResetBlockQueue(BlockQueue *blockQueue)
{ /*清空块内容*/
  BlockQueue *presentBlock;
```

```
    presentBlock = blockQueue;
    while(presentBlock != NULL){
        presentBlock->block.page = NULL;
        presentBlock->block.state = IDLE;
        presentBlock->block.time = 0;
        presentBlock = presentBlock->next;
    }
}

void PrintBlockList(BlockQueue *blockQueue, int pageID, int color, int listType)
{
    BlockQueue *presentBlock;
    char str1[5], *str2;
    presentBlock = blockQueue;
    if (单向链表){
        while (presentBlock != NULL){
            if (presentBlock->block.state == IDLE)
                printf("|     ");
            else{
                if (presentBlock->block.page->pageID == pageID)
                    printf("| \033[1;40;%dm%d\033[0m ", color, presentBlock->block.page->pageID);
                else
                    printf("| %d ", presentBlock->block.page->pageID);
            }
            presentBlock = presentBlock->next ;
        }
    } else {
            while (presentBlock->next != blockQueue){
                if (presentBlock->block.state == IDLE)
                    printf("|     ");
                else{
                    if (presentBlock->block.page->pageID == pageID)
                        printf("| \033[1;40;%dm%d\033[0m ", color,
                            presentBlock->block.page->pageID];
                    else
                        printf("| %d ", presentBlock->block.page->pageID);
                }
                presentBlock = presentBlock->next ;
            }
```

```
            if (presentBlock->block.state == IDLE)
                printf(" | ");
            else{
                if (presentBlock->block.page->pageID == pageID)
                  printf("| \033[1;40;%dm%d\033[0m ", color, presentBlock->
                      block.page->pageID);
                else
                      printf("| %d ", presentBlock->block.page->pageID);
            }
        }
}

void* InitializePageQueue(unsigned int pageLength, int maxPageID)
{               /*初始化主存块，把首地址返回。如果分配失败，返回 NULL*/
  srand((unsigned) time(&t));
  head = NULL;
  printf("Page Serials: ");
  for(i = 0; i < pageLength; i++){
     p = malloc(sizeof(PageQueue));
     p->page.pageID = (int) (rand( ) % (maxPageID + 1));
     printf("%d ",p->page.pageID);
     p->next = NULL;
     if (head == NULL)
           head = p;
     else
           q->next = p;
     q = p;
  }
  printf("\n");
  return head;
}

void InitializeProcess(process *proc, unsigned int pageSize, unsigned int maxPageID)
{
  /* 初始化进程*/
  int i;
  proc->pageLength = pageSize;
  proc->pages.next = InitializePageQueue(pageSize, maxPageID);
}
```

```
void* SearchPage(BlockQueue *blockQueue, Page page, int listType)
{              /*搜索特定页面 */
    BlockQueue *p;
  int blockSize;

  p = blockQueue;
  if (单向链表){
      while(p != NULL){
            if(p->block.page != NULL){
              if(页面 ID 相等)
                  return p;
      }
      return NULL;
  }
  else{
      while(p->next != blockQueue){
            if(p->block.page != NULL){
              if(页面 ID 相等)
                  return p;
            }
            p = p->next;
      }
      if (p->next == blockQueue){              /*处理最后一个块*/
            if(p->block.page != NULL){
              if(页面 ID 相等)
                  return p;
            }
      }
      return NULL;
  }
}

void* SearchIdleBlock(BlockQueue *blockQueue, int listType)
{              /*搜索空闲块*/
    BlockQueue *p;
  p = blockQueue;
  if (listType == 1){
      while(p != NULL){
            if(p->block.state == IDLE)
```

```
                return p;
            else
                p = p->next;
        }
        return NULL;
    } else {
        while(p->next != blockQueue){
            if(p->block.state == IDLE)
                return p;
            else
                p = p->next;
        }
        if (p->next == blockQueue){ /*处理最后一个块*/
            if(p->block.state == IDLE)
                return p;
        }
        return NULL;
    }
}
BlockQueue* GetOldestBlock(BlockQueue *blockQueue)
{ /*取得在主存中停留最久的页面，虚存页面替换算法需要比较块队列中每个成员的时间，返回最老
    的块地址*/
}

void OPT(BlockQueue *blockQueue, process *proc)
{       /*最佳置换算法*/
    blockQueueSize = GetBlockQueueSize(blockQueue, 1);
    currentPage = proc->pages.next ;
    while(currentPage != NULL){
        if(页面已置换入块){
            /*输出页面及块相关信息*/
            /*输出命中率信息*/
        } else
        if(搜索到空闲块) {
            /*更改空闲块状态信息从 IDLE 到 BUSY*/
            /*输出页面及块相关信息*/
            /*输出命中率信息*/
        } else {
            /*搜索最长时间不会被访问的页面，并将其置换出去*/
```

```
            /*输出页面及块相关信息*/
            /*输出命中率信息*/
        }
        currentPage = currentPage->next;
    }
}

void FIFO(BlockQueue *blockQueue, process *proc)
{     /*先进先出算法*/
    blockQueueSize = GetBlockQueueSize(blockQueue, 1);
    currentPage = proc->pages.next ;
    count = 0;
    while(currentPage != NULL){
        if(页面已置换入块){
            /*输出页面及块相关信息*/
            /*输出命中率信息*/
        } else
            if(搜索到空闲块) {
            /*更改空闲块状态信息从 IDLE 到 BUSY*/
            /*记录被访问时间*/
            /*输出页面及块相关信息*/
            /*输出命中率信息*/
        } else {
            /*通过比较 Block->block.time 搜索最早进入的页面，并将其置换出去*/
            /*修改最近访问时间*/
            /*输出页面及块相关信息*/
            /*输出命中率信息*/
        }
        currentPage = currentPage->next;
    }
}

void LRU(BlockQueue *blockQueue, process *proc)
{     /*最近最少页面替换算法*/
    blockQueueSize = GetBlockQueueSize(blockQueue, 1);
    currentPage = proc->pages.next ;
    while(currentPage != NULL){
        if(页面已置换入块){
            /*更新最近访问时间*/
            /*输出页面及块相关信息*/
```

```
            /*输出命中率信息*/
        } else
          if(搜索到空闲块) {
            /*记录最近访问时间*/
            /*输出页面及块相关信息*/
            /*输出命中率信息*/
        } else {
            presentBlock = GetOldestBlock(blockQueue);
            /*通过比较 Block->block.time 搜索最近最久未被使用的页面，并将其置换出去*/
            /*修改最近访问时间*/
            /*输出页面及块相关信息*/
            /*输出命中率信息*/
        }
        currentPage = currentPage->next;
    }
}

void CLOCK(BlockQueue *blockQueue, process *proc)
{       /*时钟页面置换算法，将块链形成环*/
  while(currentPage != NULL){
if(页面已置换入块){
      /*修改相应 Block->block.time，将访问标记置为 1*/
      /*输出页面及块相关信息*/
      /*输出命中率信息*/
} else
  if (搜索到空闲块) {
      /*将 Block->block.state 更改为 BUSY*/
      /*页面初次载入主存，设其访问标记为 0*/
      /*输出页面及块相关信息*/
      /*输出命中率信息*/
} else {
while ( presentBlock->next != tempBlock){
      /*寻找 Block->block.time 等于 0 的块成员*/
      /*令遍历过程中遇到的 Block->block.time==1 的成员项修改为 Block->block.time=0 */
        presentBlock = presentBlock->next;
      if (presentBlock->next == tempBlock){
          if (presentBlock->block.time == 1){
              presentBlock->block.time = 0;
              presentBlock = presentBlock->next;
          }
```

```
                }
                /*修改找到的候选成员的最近访问时间*/
                /*输出页面及块相关信息*/
                /*输出命中率信息*/
            }
            currentPage = currentPage->next;
        }
        /*释放块链环*/
    }

    int main(int argc,char **argv)
    {
        if (argc != 4){
            printf("usage: vm BlockNumber PageNumber PageTypeNumbe\n");
            return 1;
        }
        blockNumber = atoi(argv[1]);
        pageNumber = atoi(argv[2]);
        pageTypeNumber = atoi(argv[3]);
        printf("Block Numer:%d, Page Number:%d, Page Type Number: %d\n",
        blockNumber, pageNumber, pageTypeNumber);
        /*初始化块*/
        /*初始化进程*/
        OPT(blocks, &proc);
        ResetBlockQueue(blocks);
        FIFO(blocks, &proc);
        ResetBlockQueue(blocks);
        LRU(blocks, &proc);
        ResetBlockQueue(blocks);
        CLOCK(blocks, &proc);
        ResetBlockQueue(blocks);
        return 0;
    }
```

第 7 章　文件系统的设计与实现

7.1　实 验 目 的

- 深入了解文件管理涉及的概念和功能。
- 理解文件系统如何组织和管理信息。
- 通过编程模拟实现一个文件系统。

7.2　背 景 知 识

7.2.1　文件系统的基本概念

7.2.1.1　文件和文件系统

文件是由文件名标识的一组信息的有序集合。文件是存储设备的一种抽象机制，它提供一种把信息保存在存储介质上，隐蔽操作细节，便于用户存取的方法。引入文件概念的主要优点是：接口统一、使用方便、易于共享、安全可靠。

文件系统是操作系统中负责存取和管理信息的模块，它用统一方法管理用户和系统信息的存储、检索、更新、共享和保护，并为用户提供一整套方便有效的文件使用和操作方法。文件系统的主要功能有：提供文件的逻辑组织方法；提供文件的物理组织方法；提供文件的存取和使用方法；实现文件的目录管理；实现文件的共享、保护和保密；实现文件的存储空间管理；提供与 I/O 子系统的统一接口等。

7.2.1.2　文件分类和属性

文件可按用途、保护级别、存放时限、数据类型、设备类型、逻辑结构和物理结构进行分类。UNIX/Linux 支持普通文件、目录文件和设备文件，并采用多重索引结构来保存磁盘文件。

文件属性用于文件的管理控制和安全保护，它们虽非文件的信息内容，但对于系统的管理和控制是十分重要的。文件的主要属性有基本属性、类型属性、保护属性、管理属性和控制属性等。

7.2.1.3　UNIX 类文件系统和非 UNIX 类文件系统

UNIX 类文件使用 4 种和文件系统相关的抽象概念：文件、目录项、索引节点和安装点。

• 文件(file)：文件是由文件名标识的有序字节串，典型的配套文件操作有读、写、创建和删除等。

• 目录项(dentry)：目录项是文件路径名中的一部分，例如/home/fei/fei1.c，其中/、home、fei 和 fei1.c 都是目录项。

• 索引节点(inode)：索引节点是存放文件控制信息的数据结构，又分磁盘块中的索引节点和主存中活动的索引节点。

• 安装点(mount point)：文件系统被安装在一个特定的安装点上，所有的已安装文件系统都作为根文件系统树中的叶子出现在系统中。

Linux 的 Ext2 和 Ext3 是 UNIX 类文件系统。Windows 的 FAT 和 NTFS 属于非 UNIX 类文件系统，它们的实现采用其他方法。

7.2.1.4 文件控制块和文件目录

文件控制块(file control block，FCB)是文件系统给每个文件建立的唯一管理数据结构，一个文件由两部分组成：FCB 和文件体(文件信息)。FCB 一般应该包括文件标识和控制信息、文件逻辑结构信息、文件物理结构信息、文件使用信息、文件管理信息等。

文件目录是 FCB 的有序集合，它包含许多目录项。目录项有两种，分别用于描述子目录和 FCB。通常将文件目录以文件形式保存在磁盘上，这种仅含目录项的文件就称目录文件。

文件目录可组织成一级目录结构、二级目录结构和树形目录结构。树形目录结构中包含根目录、子目录、当前目录、路径名、绝对路径名和相对路径名等概念。

文件存取方式包括顺序存取、直接存取和索引存取。

文件函数有建立文件、打开文件、关闭文件、读文件、写文件、删除文件和控制文件。

7.2.2 文件管理的数据结构

7.2.2.1 磁盘上文件空间的组织

文件是对设备的一种抽象，而且是对磁盘设备进行多层次抽象的结果。

① 第一层抽象，从磁盘到分区。一个物理磁盘可划分成分区，每个分区可从逻辑上看作是一个独立的磁盘，可安装和驻留一个文件系统。

② 第二层抽象，从分区到扇区。磁盘由柱面号、磁道号和扇区号来定位，扇区是磁盘上的基本存储单元，例如每个扇区存储 1 KB，可从外向里一个柱面接一个柱面，一个磁道接一个磁道给每个扇区编号。将磁盘扇区编号的系统使磁盘变成为一系列扇区的集合。

③ 第三层抽象，从扇区到簇。不同磁盘的扇区大小可能不同，通过系统软件屏蔽这一事实并向高层软件提供统一的数据块尺寸，将若干扇区合并为一个逻辑块，称簇，再按簇进行编号，这样高层软件就只和大小都相同的簇交互，而不管物理扇区的尺寸。

④ 第四层抽象，从簇到文件系统分区。内核再将簇序列分成超级块、索引节点区和数据块区

等，再加上各种组织、控制和管理信息的软件便形成文件和文件系统。扇区序列分成以下 3 个部分。

- 超级块：占用 1#号块，存放文件系统结构和管理信息。
- 索引节点区：2#～(k+1)#块，存放索引节点表。索引节点记录文件属性，每个索引节点都有相同的大小和唯一的编号；文件系统中的每个文件在该表中都有一个索引节点。
- 数据区：(k+2)#～n#为数据块，文件的内容保存在这个区域的块中。

文件系统内部结构如图 7-1 所示。

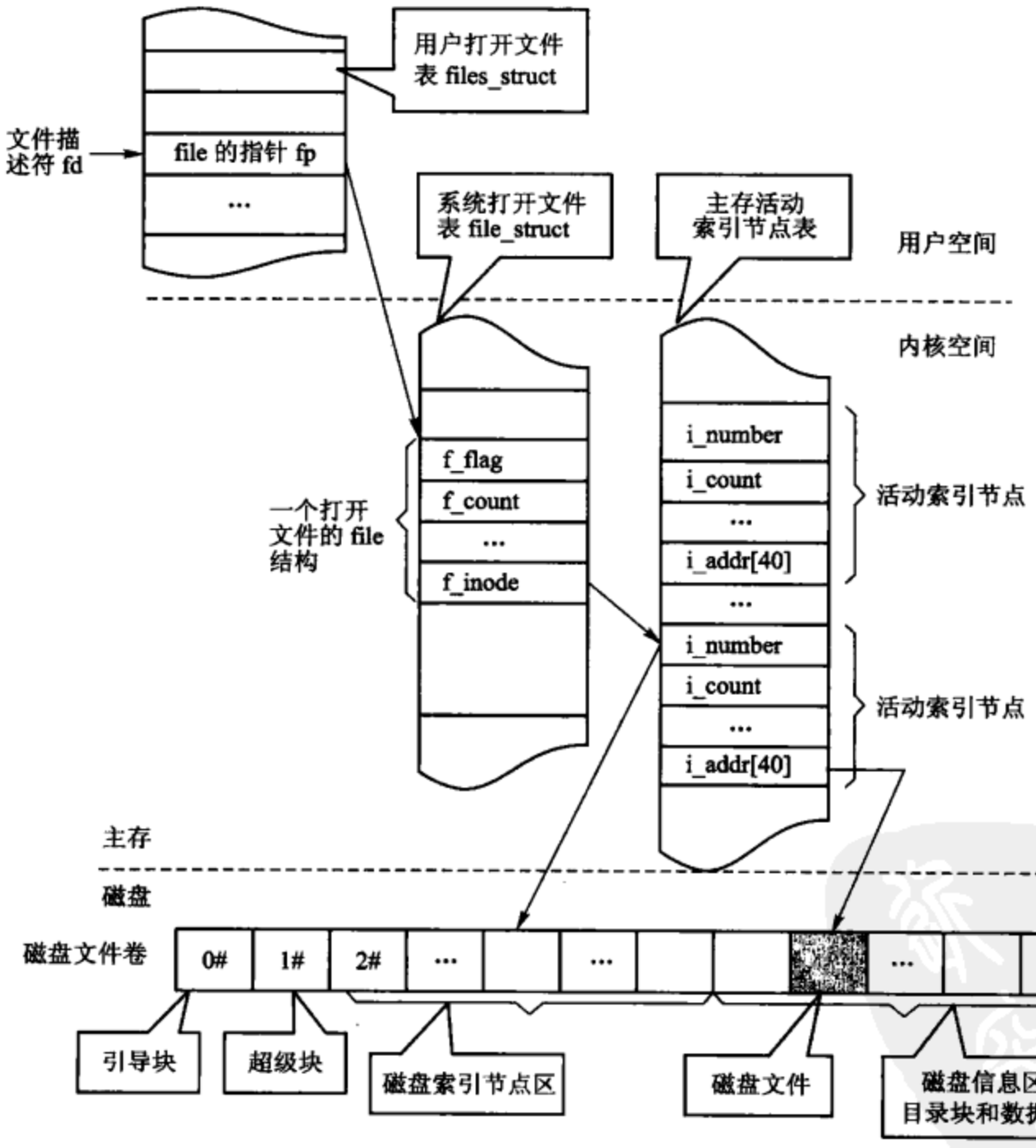


图 7-1 文件系统内部结构

7.2.2.2 主存中文件管理的数据结构

操作系统在主存中用于管理文件的数据结构包括以下 3 种。

(1) 系统打开文件表

这是为解决多用户进程共享文件、父子进程共享文件而设置的系统数据结构。打开一个文件时，通过系统打开文件表的表项把用户打开文件表的表项与文件活动索引节点链接起来，以实现数据的访问和信息的共享。

(2) 用户打开文件表

进程的 PCB 结构中建立一张用户打开文件表或称文件描述符表，表项的序号为文件描述符，该登记项内登记系统打开文件表的一个入口指针，通过此系统打开文件表的表项连接到打开文件的活动索引节点。

(3) 主存索引节点表

这是为解决频繁访问磁盘索引节点表的效率问题，系统开辟的主存区，正在使用的文件的索引节点被调入主存活动索引节点中，以加快文件访问速度。

7.2.2.3 文件空间管理方法

文件保存在光盘、磁盘、磁带等存储介质上。这些存储介质被分成物理块，全部物理块组成文件存储空间。文件存储空间的管理就是块空间的管理，包括空闲块的分配、回收和组织等几个问题。文件存储空间管理中常见的方法有以下几种。

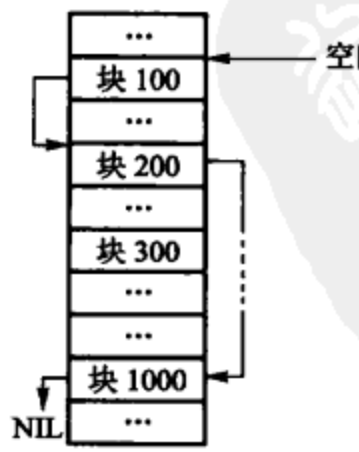
1. 空闲块表和空闲块链

空闲块表(见图 7-2a)适合于连续组织的文件，因为在建立文件时按文件尺寸申请一组连续的空闲块区，撤销文件时归还这组连续的空闲块区。与可变分区分配算法相似，可采用最先适应、最坏适应、最佳适应等算法。回收空闲块时，注意空闲块区的归并问题。由于空闲块区的个数是动态改变的，导致空闲块表目个数不能预先确定，因此可能会产生表目溢出(表较小时)或表目浪费(表较大时)。

文件存储空间的空闲物理块组成空闲块链(单向或双向链均可) (见图 7-2b)，适用于各种物理组织的文件。分配时从链头摘下一块，回收时把空闲块插入链头位置。注意，对空闲块链操作时应互斥。

首块	空闲块数	表目状态
106	4	已用
285	14	已用
…	…	未用
432	5	已用

(a) 空闲块表



(b) 空闲块链

图 7-2 空闲块表与空闲块链

2. 位示图

位示图(bit map) (见图 7-3)是反映整个文件存储空间分配情况的数据结构。其中 bit=0 表示该块是空闲块，bit=1 表示该块已分配。

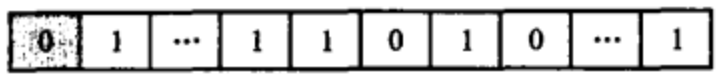


图 7-3 位示图

当申请空闲块时，从位示图上找出 bit=0 的字位，将其改为 1，返回对应的块号。归还空闲块时，在位示图中把该块所对应的字位改成 0 即可。

3. 空闲块成组链接法

UNIX/Linux 系统采用空闲块成组链接法对磁盘空间的空闲块加以组织。图 7-4 给出基于空闲块成组链接法的示例。该图中将每 100 个空闲块划归 1 组，将各组中的盘块号存放在其前组中的第 1 个空闲块中，但第 1 组的空闲盘块号放入系统专用控制块中。如图 7-4 所示，最后一组为 99 块，第 1 组不足 100 块，其他各组均为 100 块。文件存储空间不会恰好为 100 的整数倍，所以第 1 组小于 100。

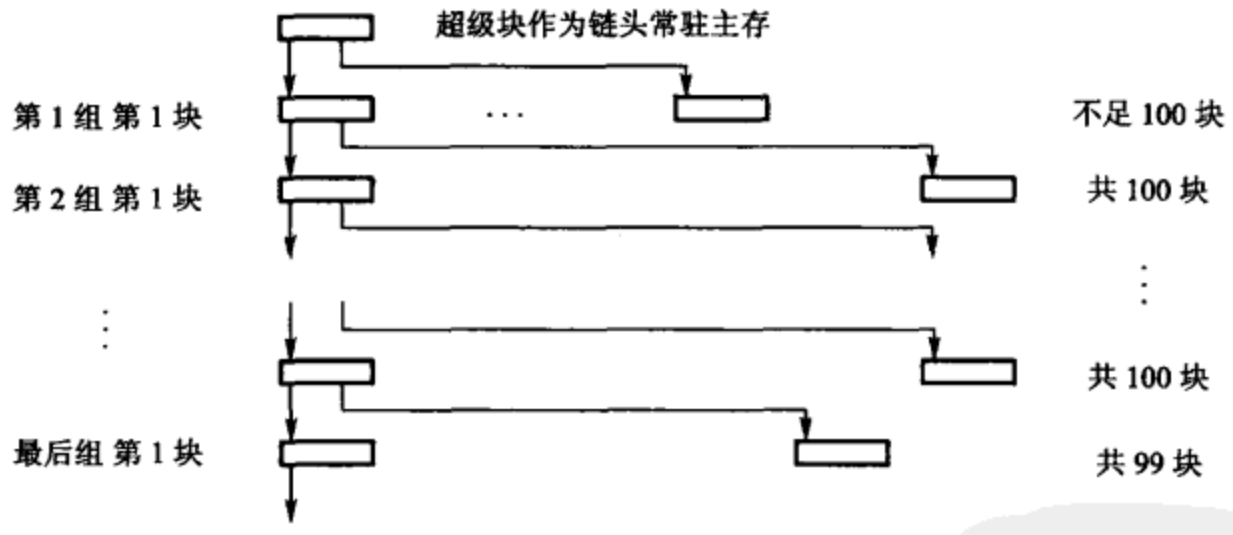


图 7-4 磁盘空间的空闲块示例

7.2.3 Ext2 文件系统

Ext2 文件系统(second extended file system)支持标准 UNIX 文件类型，包括普通文件、目录文件、特别文件和符号链接文件；Ext2 文件系统可管理特大磁盘分区，文件系统最大可达 4 TB；此外，还支持长文件名(最长 1 012 个字符)、可选的逻辑数据块大小、提供数据更新时同步写入磁盘和快速符号链接功能。

Ext2 文件系统的信息都保存在数据块中，数据块的长度相同，其物理结构如图 7-5 所示。Ext2 所占用的磁盘除引导块外，逻辑分区划分为块组，每个块组依次包括超级块、块组描述符表、块位示图、索引节点位示图、索引节点表及数据块区，重复保存有关文件系统的关键信息

及存储的文件和目录信息。文件系统保存逻辑块号，由块设备驱动程序将逻辑块号转换成块设备的物理存储位置。引导块是磁盘上第一个数据块，只有根文件系统才有引导程序放在这里，Linux 以 Ext2 作为其根文件系统，新的发行版本已默认使用 Ext3 文件系统。

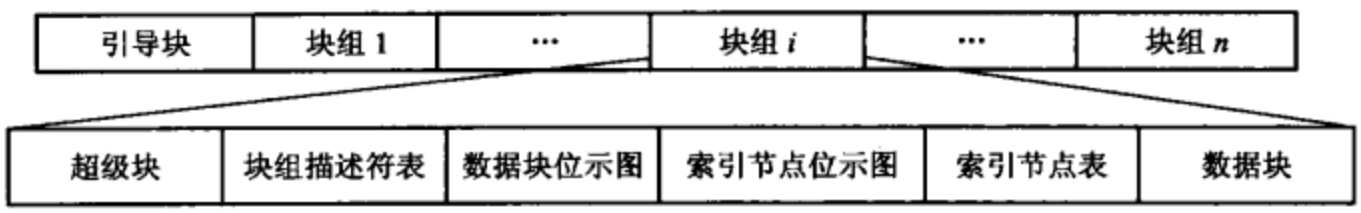


图 7-5 Ext2 文件系统结构

采用块组划分的目的一是提高文件系统可靠性，每个块组中都有管理信息的副本，当文件系统崩溃时可很容易地恢复；二是提高文件系统性能，由于块组内数据块靠近其索引节点，文件索引节点靠近其目录索引节点，从而，将磁头定位时间减到最少，加快磁盘访问速度。

7.2.3.1 超级块

Ext2 超级块用 ext2_super_block 结构表示，用来描述目录和文件在磁盘上的静态分布，包括尺寸和结构，每个块组都有一个超级块，只有块 1 的超级块才被读入主存工作，直至卸载，其他块组的超级块仅作为恢复备份。超级块主要包括：块组编号、块数量、块长度(1 KB～4 KB)、空闲块数量、索引节点数量、空闲索引节点数量、第一个索引节点号、第一个数据块位置、每个块组中的块数、每个块组的索引节点数以及安装时间、最后一次写时间、安装信息、文件系统状态信息等内容。

7.2.3.2 块组描述符

每个块组都有一个块组描述符 ext2_group_desc，记录该块组的以下信息。

- 数据块位示图：表示数据块位示图占用的块号，此位示图反映块组中数据块的分配情况，在分配或释放数据块时需使用数据块位示图。
- 索引节点位示图：表示索引节点位示图占用的块号，此位示图反映块组中索引节点的分配情况，在创建或删除文件时需使用索引节点位示图。
- 索引节点表：块组中索引节点占用的数据块数，系统中的每个文件对应一个索引节点，每个索引节点都由一个数据结构来描述。
- 空闲块数、空闲索引节点数和已用数目。

一个文件系统中的所有块组描述符结构组成一个块组描述结构表，每个块组在其超级块之后都包含一个块组描述结构表的副本，实际上，Ext2 文件系统仅使用块组 1 中的块组描述结构表。

7.2.3.3 索引节点

在 Ext2 中，每个文件都由一个 ext2_inode 来唯一描述，每个块组的索引节点集中存放在一个索引节点表中，每个索引节点有唯一的索引节点号，索引节点位示图记录索引节点的分配情况，索引节点起着文件控制块的作用，可用来对文件进行控制和管理。可区分为磁盘索引节点和主存活动索引节点。创建文件时就分给一个磁盘索引节点，文件被打开时，其对应的磁盘索引节点复制到活动索引节点中；当文件关闭时，其活动索引节点的内容写回磁盘索引节点，并释放该活动索引节点。

索引节点大小为 128B，其索引节点号为在索引节点表的存放位置，包含的与文件有关的内容有：用户 ID、用户组 ID、文件大小、文件创建时间、最近访问和修改时间、链接数、文件类型和访问权限、指向磁盘块的指针等。

7.2.3.4 文件目录

文件目录的数据块包含属于该目录的文件信息，用 ext2_dir_entry 结构描述，该结构内容有：索引节点号、本项长度、文件名长度、文件类型、文件名。文件目录的各项在数据块中依次存放，依靠本项长度计算每项的位置。

7.2.3.5 数据块分配策略

文件空间的碎片是每个文件系统都要解决的问题，它是指系统经过一段时间的读写后，导致文件的数据块散布在磁盘空间的各处，访问这类文件时，致使磁头移动急剧增多，访问速度大幅下降。操作系统提供“碎片合并”实用程序，定时运行可把碎片集中起来，Linux 的碎片合并程序叫 defrag(defragmentation program)。操作系统能够通过分配策略避免碎片的发生则更加重要，Ext2 采用两个策略来减少文件碎片。

(1) 原地先查找策略

为文件新数据分配数据块时，尽量先在文件原有数据块附近查找。首先试探紧跟文件末尾的数据块，再试探位于同一个块组的相邻的 64 个数据块，接着就在同一个块组中寻找其他空闲数据块；实在不得已才搜索其他块组，且首先考虑 8 个一簇的连续的块。

(2) 预分配策略

如果 Ext2 引入预分配机制，就从预分配的数据块取一块来用，这时紧跟该块后的若干个数据块空闲也被保留下来。文件关闭时，释放仍保留的数据块，这样保证尽可能多的数据块被集中成一簇。Ext2 的索引节点数据结构的 ext2_inode_info 域中包含两个属性 prealloc_block 和 prealloc_count，前者指向可预分配数据块链表中第一块的位置，后者表示可预分配数据块的总数。

7.3 实验 模拟实现一个 Linux 文件系统

7.3.1 实验说明

在磁盘空间模拟实现一个 Linux 文件系统，并提供基本的文件操作命令(如 mk、cp、mkdir、rmdir、cd、ls、cat、chmod、chown、chgrp、chnam 等)。文件系统的实现要采用混合索引式文件结构，包括使用空闲节点号栈管理空闲节点和成组链接管理空闲盘块。

7.3.2 解决方案

7.3.2.1 基本思路

在现有文件系统中创建一个文件，并将其模拟成一个物理磁盘，并在该文件上实现模拟磁盘块及文件系统的管理。在此基础上，设计并实现一组文件操作函数调用接口，支持对模拟文件系统的访问。

7.3.2.2 物理磁盘块设计

卷盘块数等于 100 块，每个磁盘块 512 字节，磁盘块之间用“/n”隔开，总共是 514 B。0#表示超级块，1#～10#放索引节点，每个索引节点占 64 B，共 80 个索引节点。初始化时存在根目录 root，占用 0#节点和 11#盘块。

7.3.2.3 空闲磁盘块管理

采用成组链接法管理。每组 10 块，12#～99#分为 9 组，每组的最后一个磁盘块里存放下一组的磁盘号信息。最后一组只有 8 块，加上 0 作为结束标志。在超级块中用一个一维数组(10 个元素)作为空闲磁盘块栈，放入第一组盘块。

7.3.2.4 空闲索引节点

采用混合索引式文件结构。索引节点结构中文件物理地址包括 6 项，即 4 个直接块号，一个一次间址，一个两次间址。其中一次间址和两次间址中一个磁盘块中存放 16 个磁盘号。在超级块中也用一维数组(80 个元素)作为空闲节点栈。与空闲磁盘块管理不同的是，这里不采用成组链接法，这一维数组中存放所有节点编号，而且一直保持同一大小秩序。根目录占 0#索引节点，由于根目录不会删改，一直占 0#索引节点。

7.3.2.5 超级块、索引节点及目录结构设计

```
struct SUPERBLOCK {      /*超级块*/
```

```
    int fistack[80];          /*空闲节点号栈 setw(3)×80*/
    int fiptr;                /*空闲节点栈指针 setw(3) */
    int fbstack[10];          /*空闲盘块号栈 setw(3)×10*/
    int fbptr;                /*空闲节点栈指针 setw(3) */
    int inum;                 /*空闲节点总数 setw(3) */
    int bnum;                 /*空闲盘块总数 setw(3) */
};
struct INODE {                /*节点(64 B)已保证每两个数据之间由空格隔开*/
    int fsize;                /*文件大小 setw(6) */
    int fbnum;                /*文件盘块数 setw(6) */
    int addr[4];              /*4 个直接盘块号(0～512×4–1) setw(3)×4*/
    int addr1;                /*一个一次间址( ) setw(3) */
    int addr2;                /*一个两次间址( ) setw(3) */
    char owner[6];            /*文件所有者 setw(6) */
    char group[6];            /*文件所属组 setw(6) */
    char mode[11];            /* 文件类别及存储权限 setw(12) */
    char ctime[9];            /*最近修改时间 setw(10) */
};
struct DIR{                   /*目录项(36 B) */
    char fname[14];           /*文件名(当前目录)setw(15) (0～14) */
    int index;                /*节点号 setw(3) (15～17) */
    char parfname[14];        /*父目录名 setw(15) (18～32) */
    int parindex;             /*父目录节点号 setw(3) (33～35) */
};
```

7.3.3 主要功能模块设计

7.3.3.1 节点相关操作

1. 节点操作函数

(1) 申请节点

int ialloc(void);

```
/*申请一个节点，返回节点号，否则返回–1。
返回的是空闲节点号栈中最小的节点号，节点用完时返回–1，申请失败*/
int ialloc( )
{
    if(superblock.fiptr>0) {
        int temp=superblock.fistack[80–superblock.fiptr];      /*当前可用*/
        superblock.fistack[80–superblock.fiptr]= –1;
        superblock.fiptr– –;
```

```
            return temp;
        }
        return –1;
    }
```

(2) 归还节点

void ifree(int index);

```
    /*指定一个节点号，回收一个节点。先清空节点，然后插入栈中合适位置(必须保持节点号的有序性)*/
    void ifree(int index)
    {
        disk.open("disk.txt",ios::in | ios::out | ios::nocreate ); /*清空节点*/
            disk.seekp(514+64*index+2*(index/8));
            disk<<setw(64)<<' ';
        disk.close( );
        for(int i=80–superblock.fiptr;i<80;i++)          /*节点号找到合适位置插入空闲节点号栈*/
        {
            if(superblock.fistack[i]<index)              /*若小于它，前移一位*/
            {
               superblock.fistack[i–1]=superblock.fistack[i];
            }else {/*放在第 1 个大于它的节点号前面*/
                superblock.fistack[i–1]=index;
                break;
            }
        }
        superblock.fiptr++;
    }
```

(3) 读节点

void readinode(int index,INODE &inode);

```
    /*读指定节点的索引节点信息(节点号为 index，读指针应定位到 514+64×index+2×(index/8) )，把索引
    节点信息保存到变量 inode 中，便于对同一节点进行大量操作*/
    void readinode(int index,INODE &inode)
    {
            disk.open("disk.txt",ios::in | ios::out | ios::nocreate);
            disk.seekg(514+64*index+2*(index/8));
            disk>>inode.fsize;          /*文件大小*/
            disk>>inode.fbnum;          /*文件盘块数*/
            for(int i=0;i<4;i++)        /*4 个直接盘块号*/
                disk>>inode.addr[i];
            disk>>inode.addr1;          /*一个一次间址( )*/
            disk>>inode.addr2;          /*一个两次间址( )*/
```

```
            disk>>inode.owner;          /*文件所有者*/
            disk>>inode.group;          /*文件所属组*/
            disk>>inode.mode;           /*文件类别及存储权限*/
            disk>>inode.ctime;          /*最近修改时间*/
        disk.close( );
    }
```

(4) 写节点

void writeinode(INODE inode,int index);

```
    /*把 INODE inode 写回指定的节点*/
     void writeinode(INODE inode,int index)
    {
        disk.open("disk.txt",ios::in | ios::out | ios::nocreate);
        disk.seekp(514+64*index+2*(index/8));
            disk<<setw(6)<<inode.fsize;                    /*文件大小*/
            disk<<setw(6)<<inode.fbnum;                    /*文件盘块数*/
            for(int i=0;i<4;i++)                           /*4 个直接盘块号*/
                disk<<setw(3)<<inode.addr[i];
            disk<<setw(3)<<inode.addr1;                    /*一个一次间址( )*/
            disk<<setw(3)<<inode.addr2;                    /*一个两次间址( )*/
            disk<<setw(6)<<inode.owner;                    /*文件所有者*/
            disk<<setw(6)<<inode.group;                    /*文件所属组*/
            disk<<setw(12)<<inode.mode;                    /*文件类别及存储权限*/
            disk<<setw(10)<<inode.ctime;                   /*最近修改时间*/
        disk.close( );
    }
```

2. 盘块操作函数

(1) 申请盘块

int balloc(void); /*成组链接法*/

```
    /*申请一个盘块，返回盘块号，否则返回–1*/
    int balloc( )
    {
        int temp=superblock.fbstack[10–superblock.fbptr];
        if(superblock.fbptr==1) {                          /*到栈底了*/
            /*是最后记录盘块号 0(保留作为栈底，分配不成功)*/
            if(temp==0) {
                return –1;
            }
            superblock.fbstack[10–superblock.fbptr]= –1;
            superblock.fbptr=0;
```

```
/*盘块内容读入栈*/
for(int i=0;i<10;i++) {
        int id,num=0;
        disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
          /*先计算盘块内容个数 num(最多 10)，最后盘块可能不到 10 个*/
          disk.seekg(514*temp);
          for(int i=0;i<10;i++) {
                disk>>id;
                num++;
                if(id==0)   break;
          }
          disk.seekg(514*temp);                          /*盘块内容读入栈*/
          for(int j=10–num;j<10;j++) {
              disk>>id;
              superblock.fbstack[j]=id;
          }
          superblock.fbptr=num;
        disk.close( );
  }
  disk.open("disk.txt",ios::in | ios::out | ios::nocreate );       /*清空回收盘块*/
                disk.seekp(514*temp);
                disk<<setw(512)<<' ';
                disk.close( );
                /*盘块使用掉*/
                return temp;
          } else {       /*不是记录盘块*/
                superblock.fbstack[10–superblock.fbptr]= –1;
                superblock.fbptr– –;
                return temp;
          }
    }
```

(2) 归还盘块

void bfree(int index);

```
    /*指定一个盘块号，回收一个盘块*/
    void bfree(int index)
    {
         disk.open("disk.txt",ios::in | ios::out | ios::nocreate );       /*清空回收盘块*/
              disk.seekp(514*index);
              disk<<setw(512)<<' ';
        disk.close( );
```

```
        if(superblock.fbptr==10) {    /*栈已满，栈中内容记入回收盘块，栈清空*/
            disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
                disk.seekp(514*index);
                for(int i=0;i<10;i++)
                {
                        disk<<setw(3)<<superblock.fbstack[i];
                        superblock.fbstack[i]= -1;
                }
            disk.close( );
            superblock.fbptr=0;
        }/*回收盘块压栈*/
        superblock.fbstack[10-superblock.fbptr-1]=index;
        superblock.fbptr++;
    }
```

3. 超级块操作函数

(1) 读超级块

void readsuper(void);

```
    /*读超级块到主存 SUPERBLOCK superblock*/
    void readsuper( )
    {    /*读超级块到主存*/
            disk.open("disk.txt",ios::in | ios::out | ios::nocreate);
            int i;
            for(i=0;i<80;i++) {                /*读空闲节点号栈*/
                disk>>superblock.fistack[i];
            }
            disk>>superblock.fiptr;            /*空闲节点号栈指针*/
            for(i=0;i<10;i++) {                /*读空闲盘块号栈*/
                disk>>superblock.fbstack[i];
            }
            disk>>superblock.fbptr;            /*空闲盘块号栈指针*/
            disk>>superblock.inum;             /*空闲节点号总数*/
            disk>>superblock.bnum;             /*空闲盘块号总数*/
        disk.close( );
    }
```

(2) 写超级块

void writesuper(void);

```
    /*主存 SUPERBLOCK superblock;写回超级块*/
    void writesuper( )
    {    /*主存写回超级块*/
```

```
            disk.open("disk.txt",ios::in | ios::out | ios::nocreate);
            int i;
            for(i=0;i<80;i++) {                    /*写空闲节点号栈*/
                disk<<setw(3)<<superblock.fistack[i];
            }
            disk<<setw(3)<<superblock.fiptr;       /*空闲节点号栈指针*/
            for(i=0;i<10;i++)  {                   /*写空闲盘块号栈*/
                disk<<setw(3)<<superblock.fbstack[i];
            }
            disk<<setw(3)<<superblock.fbptr;       /*空闲盘块号栈指针*/
            disk<<setw(3)<<superblock.inum;        /*空闲节点号总数*/
            disk<<setw(3)<<superblock.bnum;        /*空闲盘块号总数*/
        disk.close( );
    }
```

4. 目录项操作函数

(1) 读目录

void readdir(INODE inode,int index,DIR &dir);

```
    /*读指定目录项(索引节点的盘块，下标 index)进临时对象*/
    void readdir(INODE inode,int index,DIR &dir)
    {
        disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
        disk.seekg(514*inode.addr[0]+36*index);
        disk>>dir.fname;
        disk>>dir.index;
        disk>>dir.parfname;
        disk>>dir.parindex;

        disk.seekp(514*inode.addr[0]+36*index);
        disk<<setw(15)<<' ';
        disk<<setw(3)<<' ';
        disk<<setw(15)<<' ';
        disk<<setw(3)<<' ';
        disk.close( );
    }
```

(2) 写目录

void writedir(INODE inode,DIR dir,int index);

```
    /*目录项对象 DIR dir 写到指定目录项*/
    void writedir(INODE inode,DIR dir,int index)
    {
```

```
        disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
            disk.seekp(514*inode.addr[0]+36*index);
            disk<<setw(15)<<dir.fname;
            disk<<setw(3)<<dir.index;
            disk<<setw(15)<<dir.parfname;
            disk<<setw(3)<<dir.parindex;
        disk.close( );
    }
```

7.3.3.2 文件相关操作

1. 创建文件

void mk(char *dirname,char *content);

```
    /*当前目录下创建一个数据文件(规定目录文件只占 1～4 个盘块)。第 1 个参数表示文件名，第 2 个
      参数表示文件内容，可以在文件副本中使用这个函数*/
    void mk(char *filename,char *content)
    {
        INODE inode,inode2;
        readinode( road[num–1],inode );          /*把当前节点 road[num–1]内容读入索引节点*/
        if( havewpower(inode) ) {                /*判断权限*/
            if(512–inode.fsize<36) {             /*是否目录项已达到最多 14 个*/
                cout<<"当前目录已满，创建子目录失败!";
            } else {
                int i,index2;
                if( havesame(filename,inode,i,index2) ) {          /*有无重名存在*/
                    cout<<"该名已存在，创建失败!";
                } else {                                           /*可以创建目录*/
                    int size=strlen(content)+1;
                    /*cout<<"请输入所创文件的大小(1～2048)";       /*最多 4 个盘块*/
                    cin>>size;
                    if( size>2048 ) {
                        cout<<"\n 文件太大，创建失败!";
                    } else {
                        int bnum=(size–1)/512+1;                   /*计算盘块数(1～4)*/
                        int bid[4];
                        int iid=ialloc( );                         /*申请节点*/
                        if(iid!= –1) {
                            bool success=true;
                            for(int i=0;i<bnum;i++) {              /*申请盘块*/
                                bid[i]=balloc( );
```

```
        if(bid[i]== -1) {
            cout<<"盘块不够，创建数据文件失败!";
            success=false;
            ifree(iid);/*已申请的节点和盘块释放掉*/
            for(int j=i-1;j>=0;j--)
            {
                bfree( bid[j] );
            }
        break;
      }
}
if(success) {                          /*当前目录盘块的修改*/
      disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
      disk.seekp(514*inode.addr[0]+inode.fsize );    /*写目录名*/
      disk<<setw(15)<<filename;                      /*写节点*/
      disk<<setw(3)<<iid;
      disk<<setw(15)<<curname;
      disk<<setw(3)<<road[num-1];
      disk.close( );
      /*当前目录节点的修改*/
inode.fsize+=36;
char tmpbuf[9];
_strtime(tmpbuf);
strcpy(inode.ctime,tmpbuf);
/*新建目录节点的初始化*/
inode2.fsize=size;              /*int fsize;文件大小*/
inode2.fbnum=bnum;              /*int fbnum;文件盘块数*/
int i;
for(i=0;i<4;i++) {              /*4 个直接盘块号*/
      if(i<bnum) {
            inode2.addr[i]=bid[i];
      } else {
            inode2.addr[i]=0;
      }
}
inode2.addr1=0;                    /*int addr1;一个一次间址 */
inode2.addr2=0;                    /*int addr2;一个两次间址*/
strcpy( inode2.owner,auser );      /*char owner[6];文件所有者*/
strcpy( inode2.group,agroup );  /*char group[6];文件所属组*/
/*文件类别及存储权限(默认最高) */
```

```
                strcpy( inode2.mode," –rwxrwxrwx" );
                _strtime(tmpbuf);
                strcpy( inode2.ctime,tmpbuf );/*最近修改时间*/
                writeinode( inode2,iid );
                /*新建文件盘块的初始化(内容写入), 最后盘块不一定满*/
                char temp;
                disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
                    disk.seekp( 514*bid[0] );
                    for(i=0;i<size;i++) {
                            temp=content[i];
                            disk<<temp;
                            if(i/512==511)
                            {
                                    disk<<'\n';
                            }
                    }
                disk.close( );
                cout<<"文件已成功创建";
            }
        }
                                    else{
                                        cout<<"节点已用完，创建数据文件失败!";
                                    }
                                }
                            }
                        }
                    } else {
                        cout<<"你没有权限";
                    }
                    writeinode( inode,road[num –1] ); /*把 inode 写入指定节点*/
                }
```

2. 删除文件

void rm(char *filename);

```
    /*当前目录下删除指定数据文件*/
    void rm(char *filename)
    {
        INODE inode,inode2;
        DIR dir;
        readinode(road[num–1],inode);        /*当前节点写入节点对象*/
```

```
if( havewpower(inode) ) {                          /*判断权限*/
    int i,index2;/*i 为待删子目录目录项下标 index2 为目录项中的待删子目录的节点号*/
    if( havesame(filename,inode,i,index2) )        /*存在该子目录名*/
    {
      readinode(index2,inode2);                    /*待删子目录的节点写入节点对象*/
      if( havewpower(inode2) ) {                   /*判断权限*/
        if( inode2.mode[0]=='-' ) {                /*判断是数据文件而非目录文件*/
          /*回收盘块和节点*/
          for(int ii=0;ii<inode2.fbnum;ii++)
          {
                bfree( inode2.addr[ii] );
          }
          ifree( index2 );
/*对当前目录盘块的修改，inode.addr[0]为当前盘块号，i 为待删子目录的目录项下标*/
          disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
              disk.seekp(514*inode.addr[0]+36*i);  /*清空当前盘块第 i 个子目录的目录项
              内容*/
              disk<<setw(36)<<' ';
          disk.close( );
          for(int j=i+1;j<(inode.fsize/36);j++) {        /*后面的前移一位*/
              readdir(inode,j,dir);/*inode 指向盘块读入*/
              writedir(inode,dir,j -1);
          }
          /*对当前目录节点的修改*/
          inode.fsize- =36;
          char tmpbuf[9];
          _strtime(tmpbuf);
          strcpy( inode.ctime,tmpbuf );
          cout<<"文件已成功删除";
        } else {
          cout<<"目录文件应用 rmdir 命令删除";
        }
      } else {
          cout<<"你没有权限";
      }
} else {
    cout<<"目录中不存在该子目录!!!";
```

```
            }
        } else {
            cout<<"你没有权限";
        }
        writeinode(inode,road[num-1]);
    }
```

3. 复制文件

void cp(char*string);

```
    /*给定一个路径，找到该文件并复制到当前目录下*/
    void cp(char*string)
        { /*把指定目录下的指定文件复制到当前目录下*/
        bool getit=false;                    /*记录指定文件是否找到*/
        char content[2048];                  /*保存内容*/
        char fname[14];                      /*保存文件名*/

        char tcurname[14];                   /*保存当前路径*/
        int troad[20];
        int tnum=num;
strcpy(tcurname,curname);
for(int i=0;i<num;i++) {
    troad[i]=road[i];
}
if( find(string) )                           /*如果找到目标(当前路径跟着改变)*/
{
        INODE inode;
        readinode(road[num-1],inode);
        if(inode.mode[0]=='-')               /*确定是数据文件*/
        {
            getit=true;
            /*文件名复制到 fname 变量，盘块内容复制到字符串 content[2048] */
            strcpy(fname,curname);
            char temp=' ';
            disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
                    int i,j;
                    for(i=0,j=0;j<inode.fsize;i++,j++)
                     {
```

```
                    disk.seekg(514*inode.addr[0]+i);
                    disk>>temp;
                    content[j]=temp;
                    if(i/512==511)    /*跳过'\n'*/
                    {
                        i=i+2;
                    }
                }
                content[j+1]='\0';
            disk.close( );
        } else {
            cout<<"不能根据路径找到相关目录";
        }
    } else {
        cout<<"不能根据路径找到相关目录";
    }
strcpy(curname,tcurname);                    /*路径还原*/
num=tnum;
 for(int ii=0;ii<tnum;ii++){
        road[ii]=troad[ii];
        }
        if(getit) {                          /*文件复制*/

            mk(fname,content);
            cout<<"文件复制完毕";
        }
    }
```

4. 显示文件内容

void cat(char *filename);

```
    /*显示当前目录下指定数据文件的内容*/
    void cat(char *filename)
    { /*显示文件内容(当前目录下指定数据文件) */
        INODE inode,inode2;
        readinode(road[num -1],inode);    /*当前节点写入节点对象*/
            int i,index2;    /*i 为子目录待显目录项下标，index2 为目录项中的待显示目录的节点号*/
            if(havesame(filename,inode,i,index2)){
                readinode(index2,inode2);    /*要显示的数据文件的节点写入节点对象*/
                if(inode2.mode[0]=='-'){
                    cout<<"文件内容为：(用文件名和*填充来模拟文件内容)\n";
```

```
                    char content[512];
                    disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
                    for(int i=0;i<inode2.fbnum;i++)/*遍历盘块并输出盘块内容*/
                    {
                        disk.seekg(inode2.addr[i]*514);
                        disk>>content;
                        cout<<content;
                    }
                    disk.close( );
                } else {
                cout<<"显示失败(目标是目录文件，cat 是用来显示数据文件内容的)";
                }
            } else {
                cout<<"目录中不存在该数据文件!!!";
            }
    }
```

7.3.3.3 目录相关操作

1. 判断目录项是否存在

bool havesame(char *dirname,INODE inode,int &i,int &index2)

/*判断对象 inode 指向的目录文件盘块中有无该名(dirname)的目录项存在，有则返回 1，无则返回 0。同时，若有该目录项，则按引用调用的 i 为待删子目录目录项下标，index2 为目录项中的待删子目录的节点号*/

```
bool havesame(char *dirname,INODE inode,int &i,int &index2)
{
    bool have=false;
    char name[14];
    disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
        for(i=0;i<(inode.fsize/36);i++) {          /*遍历所有的目录项*/
            disk.seekg(514*inode.addr[0]+36*i);
            disk>>name;
            if( !strcmp(dirname,name) ){
                have=true;
                disk>>index2;
                break;
            }
        }
    disk.close( );
```

```
        return have;
    }
```

2. 查找文件或目录

bool find(char *string)

```
    /*根据路径找到指定文件或目录，路径至少有一个“/”以 root 开头，不以“/”结尾。需要注意，使
    用此函数时当前路径跟着改掉了，所以使用前必须保存当前路径。万一找不到目标，可以还原为当
    前路径*/
    bool find(char *string)
    {
        int ptr=0;
        char name[14]=" ";
        INODE inode;
        /*读根目录*/
       for(int i=0 ; string[ptr]!='/' ; ptr++ ,i++){
            if(i==15)   return 0;                    /*超过正常长度，肯定错*/
            name[i]=string[ptr];
        }
        if( !strcmp(name,"root") ) {                 /*第 1 个应该肯定是“root”*/
            strcpy(curname,"root");                  /*初始化当前路径*/
            road[0]=0;
num=1;
for(;;) {
        readinode(road[num-1],inode);      /*当前节点读入*/
        ptr++;
        char name[14]=" ";
        for(int i=0; (string[ptr]!='/')&&(string[ptr]!='\0') ; ptr++ ,i++) {
            /*从 string 读入一个名字*/
            if(i==15)   return 0;            /*超过正常长度，肯定错*/
            name[i]=string[ptr];
        }
        int ii,index2;                       /*当前目录查找该目录项*/
        if( havesame(name,inode,ii,index2) ) {
            char tname[14];
            /*路径下一步*/
            disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
                disk.seekg(514*inode.addr[0]+36*ii);
                disk>>tname;
            disk.close( );

            strcpy(curname,tname);
```

```
                    road[num]=index2;
                    num++;
                    /*判断字符串结尾*/
                    if( string[ptr]=='\0' ) {
                        return 1;
                    }
                } else {
                    return 0;
                }
            }
        } else {
            return 0;
        }
    }
```

3. 创建目录

void mkdir(char *dirname)

/*当前目录下创建目录。规定目录文件只占一个盘块。为了降低难度，已设置目录文件只占一个盘块*/

```
void mkdir(char *dirname)
{/*创建目录(规定目录文件只占一个盘块)当前目录下创建*/
INODE inode,inode2;
readinode( road[num-1],inode );                    /*把当前节点 road[num-1]的内容读入索引节点*/
if( havewpower(inode) ) {                          /*判断权限*/
    if(512-inode.fsize<36) {                       /*是否目录项已达到最多 14 个*/
        cout<<"当前目录已满，创建子目录失败!";
    }
    else{
        int i,index2;
        if( havesame(dirname,inode,i,index2) ) {   /*有无重名存在*/
            cout<<"该目录已存在，创建失败!";
        } else {                                   /*可以创建目录*/
            int iid=ialloc( );                     /*申请节点*/
            if( iid!=-1 ){
                int bid=balloc( );                 /*申请盘块*/
                if(bid!=-1){
                /*当前目录盘块的修改*/
                disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
                disk.seekp(   514*inode.addr[0]+inode.fsize );
                disk<<setw(15)<<dirname;           /*写目录名*/
                disk<<setw(3)<<iid;                /*写节点*/
```

```
disk<<setw(15)<<curname;
disk<<setw(3)<<road[num-1];
disk.close( );
/*当前目录节点的修改*/
inode.fsize+=36;
char tmpbuf[9];
_strtime(tmpbuf);
strcpy(inode.ctime,tmpbuf);
/*新建目录节点的初始化*/
inode2.fsize=0;                    /*int fsize;文件大小*/
inode2.fbnum=1;                    /*int fbnum;文件盘块数*/
inode2.addr[0]=bid;
for(int aaa=1;aaa<4;aaa++)         /*指向 0#表示没有指向*/
    inode2.addr[aaa]=0;
inode2.addr1=0;                    /*int addr1;一个一次间址 */
inode2.addr2=0;                    /*int addr2;一个两次间址*/
strcpy( inode2.owner,auser );      /*char owner[6];文件所有者*/
strcpy( inode2.group,agroup );     /*char group[6];文件所属组*/
strcpy( inode2.mode,"drwxrwxrwx" );
/*文件类别及存储权限(默认最高)
_strtime(tmpbuf);
strcpy( inode2.ctime,tmpbuf );     /*char ctime[9];最近修改时间*/
writeinode( inode2,iid );
disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
disk.seekp(514+64*iid+2*(iid/8));
disk<<setw(6)<<0;                  /*int fsize;文件大小*/
disk<<setw(6)<<1;                  /*int fbnum;文件盘块数*/
disk<<setw(3)<<bid;
disk<<setw(3)<<0;                  /*指向 0#表示没有指向*/
disk<<setw(3)<<0;
disk<<setw(3)<<0;
disk<<setw(3)<<0;                  /*int addr1;一个一次间址*/
disk<<setw(3)<<0;                  /*int addr2;一个两次间址*/
disk<<setw(6)<<auser;              /*char owner[6];文件所有者*/
disk<<setw(6)<<agroup;             /*char group[6];文件所属组*/
disk<<setw(12)<<"drwxrwxrwx";
/*文件类别及存储权限(默认最高) */
_strtime(tmpbuf);
disk<<setw(10)<<tmpbuf;            /*char ctime[9];最近修改时间*/
disk.close( ); */
```

```
                    cout<<"目录已成功创建";
                    } else {
                        ifree(iid);                    /*释放刚申请的节点*/
                        cout<<"盘块已用完，创建子目录失败!";
                    }
                } else {
                    cout<<"节点已用完，创建子目录失败!";
                }
            }
        }
    } else {
        cout<<"你没权限创建";
    }
    writeinode( inode,road[num –1] );/*把 inode 写入指定节点*/
}
```

4. 删除目录

void rmdir(char *dirname,int index)

/*当前目录下删除目录。将要删除的目录可能非空。有两种策略：一是规定只能删除空目录；二是递归地将非空目录的所有子目录删除，然后再删除自己。第一种实现较简单，这里使用了第二种策略。所以参数为 (子目录名,当前节点)。如果使用第一种策略，参数只要子目录名即可*/

```
void rmdir(char *dirname,int index)
{
/*删除目录，当前目录下删除，参数为 (子目录名,当前节点) */
  t++;
  INODE inode,inode2;
  DIR dir;
  readinode(index,inode);                          /*当前节点写入节点对象*/
  if( havewpower(inode) ) {                        /*判断权限*/
    int i,index2;              /*i 为待删子目录目录项下标，index2 为目录项中的待删子目录的节点*/
    if( havesame(dirname,inode,i,index2) ) {       /*存在该子目录名*/
      readinode(index2,inode2);  /*待删子目录的节点写入节点对象*/
      if( havewpower(inode2) ) {                   /*判断权限*/
        if( inode2.mode[0]=='d' ) {                /*判断要删除的是目录文件而非数据文件*/
          if( inode2.fsize!=0 ) {                  /*判断待删子目录有无子目录==>有 cout<<"该目
                                                     录非空，不能删除"*/
            char yes='y';
            if(t==1) {
              cout<<"该目录非空，如果删除的话，将失去目录下所有文件，要继续吗?(y/n) ";
              cin>>yes;
```

```
}
if(  (yes=='y') || (yes=='Y') ) {
     /*遍历待删子目录(inode2)所有子目录，递归将其删除*/
     char name[14];
     int index3;
     INODE inode3;
     for(int i=0;i<(inode2.fsize/36);i++) {      /*遍历所有的目录项*/
          disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
               disk.seekg(inode2.addr[0]*514+36*i);
               disk>>name;
               disk>>index3;
          disk.close( );
          readinode(index3,inode3);
          if(inode3.mode[0]=='d') {              /*是目录文件*/
               rmdir(name,index2);
          } else {                               /*是数据文件*/
               rm(name);
          }
     }
     rmdir(dirname,index);                       /*子目录空了后再删除自己*/
} else {
          cout<<"目录删除失败";
     }
} else {   /*空目录删除*/
     /*回收盘块和节点*/
     bfree( inode2.addr[0] );
     ifree( index2 );
     /*对当前目录盘块的修改，inode.addr[0]为当前盘块号，i 为待删子目录目录项
     下标*/
     disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
          disk.seekp(514*inode.addr[0]+36*i);    /*清空当前盘块第 i 个子目录的目
          录项内容*/
          disk<<setw(36)<<' ';
     disk.close( );
     for(int j=i+1;j<(inode.fsize/36);j++) {          /*后面的前移一位*/
          readdir(inode,j,dir);                       /*inode 指向盘块读入*/
          writedir(inode,dir,j-1);
     }
     /*对当前目录节点的修改*/
     inode.fsize-=36;
     char tmpbuf[9];
     _strtime(tmpbuf);
```

```
                    strcpy( inode.ctime,tmpbuf );
                }
                cout<<"目录成功删除   ";
                /*修改当前节点*/
                inode.fsize-=36;
                char tmpbuf[9];
                _strtime(tmpbuf);
                strcpy(inode.ctime,tmpbuf);
            } else {
                cout<<"数据文件应用 rm 命令删除";
            }
        } else {
                cout<<"你没有权限";
        }
    } else {
        cout<<"目录中不存在该子目录!!!";
    }
} else {
    cout<<"你没有权限";
}
    writeinode(inode,index);
    t--;
}
```

5. **显示目录**

void ls(void)

```
/*显示当前节点的所有子目录*/
void ls(void)
{/*显示当前节点的所有子目录*/
    INODE inode,inode2;
    readinode( road[num -1],inode );
        char name[14];
        int index;
        for(int i=0;i<(inode.fsize/36);i++) {   /*遍历所有的目录项*/
            disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
                disk.seekg(inode.addr[0]*514+36*i);
                disk>>name;
                disk>>index;
            disk.close( );
            readinode( index,inode2 );
            cout<<setw(15)<<name<<setw(6)<<inode2.fsize<<setw(6)<<inode2.owner;
```

```
        cout<<setw(6)<<inode2.group<<setw(12)<<inode2.mode<<setw(10)<<inode2.ctime<<endl;
                }
                cout<<"显示完毕";
    }
```

6. 改变目录

void cd(char *string)

/*改变当前目录。有 4 种处理过程： string="."时，表示切换到当前目录；string=".."时，表示切换到父目录；string= "/" 时，表示切换到根目录；string 为一路径时，则调用 bool find(char *string) 切换到指定目录*/

```
void cd(char *string)
{
    if( !strcmp(string,".") ) {                    /*切换到当前目录*/
        cout<<"已切换到当前目录";
        return;
    }
    if( !strcmp(string,"/") ) {                    /*切换到根目录*/
        strcpy(curname,"root");
        road[0]=0;
        num=1;
        cout<<"已切换到根目录";
        return;
    }
    if( !strcmp(string,"..") ) {                   /*切换到父目录*/
        if( strcmp(curname,"root") ) {             /*当前不是根目录才可切换*/
            INODE inode;
            readinode(road[num-2],inode);          /*父目录节点号*/
            char name[14];
            disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
                                                   /*得到父目录名(从父目录任一目录项) */
                disk.seekg(inode.addr[0]*514+18);
                disk>>name;
            disk.close( );
            strcpy(curname,name);
            num--;
            cout<<"已切换到父目录";
            return;
        }
        cout<<"当前已是根目录";
        return;
    }
```

```
char *per=strchr(string,(int)'/');
if(per==NULL) {                                    /*没有“/”的是子目录名，切换到某一子目录*/
    INODE inode,inode2;
    int i,index2;
    readinode(road[num-1],inode);
    char name[14];
    if( havesame( string,inode,i,index2) ) {
        readinode(index2,inode2);
        if(inode2.mode[0]=='d') {
            disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
                disk.seekg(514*inode.addr[0]+36*i);
                disk>>name;
            disk.close( );
            strcpy(curname,name);
            road[num]=index2;
            num++;
            cout<<"已切换到子目录";
            return;
        }
        cout<<"不能根据路径找到相关目录,因为 "<<string<<" 是数据文件";
    } else {
        cout<<"该子目录不存在，不能根据路径找到相关目录";
    }
} else {                                           /*根据指定路径切换目录*/
    char tcurname[14];                             /*保存当前路径*/
    int troad[20];
    int tnum=num;
    strcpy(tcurname,curname);
    for(int i=0;i<num;i++){
        troad[i]=road[i];
    }
    if( find(string) ) {                           /*如果找到目标(当前路径随之更改)*/
        INODE inode;
        readinode(road[num-1],inode);
        if(inode.mode[0]=='d') {                   /*确定是目录文件*/
            cout<<"已切换到该目录";
            return;
        }
    }
    cout<<"不能根据路径找到相关目录";              /*找不到目标，还原*/
```

```
                strcpy(curname,tcurname);
                num=tnum;
                for(int ii=0;ii<tnum;ii++){
                        road[ii]=troad[ii];
                }
        }
}
```

7.3.3.4 用户管理相关操作

1. 登录

bool login(void);

```
/*登录信息，要求输入用户信息，并判断是否合法*/
bool login( )
{                                   /*登录*/
        char auser2[6];             /*存放用户组内的信息*/
        char apwd2[6];
        cout<<"==your name: ";
        cin>>auser;
        cout<<"==your password: ";
        cin>>apwd;
bool have=false;            /*记录用户名和密码正确否*/
user.open("user.txt",ios::in);
        for( int n=0;n<usernum;n++ )
        /*用户名为 18n+0~18n+5，密码为 18n+6~18n+11，用户组为 18n+12~18n+17*/
        {
                user.seekg(18*n);

                user>>auser2>>apwd2;
                int a=strcmp(auser,auser2);
                int b=strcmp(apwd,apwd2);
                if( (!a)&&(!b) )
                {
                        have=true;
                        user>>agroup;
                        break;
                }
        }
user.close( );
if(have==true)
        return 1;
```

```
        return 0;
    }
```

2. 修改密码

void changepassword(void);

```
    /*改变当前用户的密码*/
    void changepassword( )
    {/*改变用户密码*/
        char auser2[6];
        char apwd2[6];
        cout<<"请输入原有密码：";
        cin>>apwd2;
        if(!strcmp(apwd,apwd2)){
            user.open("user.txt",ios::in | ios::out | ios::nocreate);
            int n;
            for(n=0;n<usernum;n++ ) {/*找到位置*/
                user.seekg(18*n);
                user>>auser2;
                if(!strcmp(auser,auser2)){
                    user>>agroup;
                    break;
                }
            }
            cout<<"请输入密码：";
            cin>>apwd2;
            user.seekp(18*n+6);
            user<<setw(6)<<apwd2;
            cout<<"密码修改成功";
            user.close( );
        } else {
            cout<<"输入错误";
        }
    }
```

3. 判别写权限

bool havewpower(INODE inode)

```
    /*判断当前用户对指定的节点有无写权限*/
    bool havewpower(INODE inode)
    {/*判断当前用户对指定的节点有无写权限*/
        if( !strcmp(auser,inode.owner) ) {          /*是文件所有者*/
            if(inode.mode[2]=='w')
                return true;
```

```
            return false;
        } else {
            if( !strcmp(agroup,inode.owner) )          /*在组内*/
            {
                if(inode.mode[5]=='w')
                    return true;
                return false;
            } else {                                   /*其他用户*/
                if(inode.mode[8]=='w')
                    return true;
                return false;
            }
        }
    }
```

4. 改变文件权限

void chmod(char *name)

```
    /*改变当前目录下指定文件的文件权限*/
    void chmod(char *name)
    {
        INODE inode,inode2;
        readinode(road[num-1],inode);          /*当前节点写入节点对象*/
        int i,index2;                          /*i 为目录项下标，index2 为目录项中节点号*/

        if( havesame(name,inode,i,index2) ){
            readinode(index2,inode2);
            if( havewpower(inode2) ){
                char amode[3];
                cout<<"1 表示所有者，4 表示组内，7 表示其他用户，a 表示-wx 模式，b 表示 r-x
                    模式，c 表示 rwx 模式\n";
                cout<<"请输入修改方案(例如 4c)：";
                cin>>amode;
                if( amode[0]=='1' || amode[0]=='4' || amode[0]=='7' ){
                    /*(int)amode[0]=49;强制类型转换是转成其 ASCII 码*/
                    if( amode[1]=='a' ) {
                        inode2.mode[ (int)amode[0]- 48 ]='-';
                        inode2.mode[ (int)amode[0]+1- 48 ]='w';
                        writeinode(inode2,index2);
                        cout<<"修改完毕";
                    } else {
                        if( amode[1]=='b' ) {
```

```
                    inode2.mode[ (int)amode[0]– 48 ]='r';
                    inode2.mode[ (int)amode[0]+1– 48 ]='–';
                    writeinode(inode2,index2);
                    cout<<"修改完毕";
                } else {
                    if( amode[1]=='c' )        {
                        inode2.mode[ (int)amode[0]– 48 ]='r';
                        inode2.mode[ (int)amode[0]+1– 48 ]='w';
                        writeinode(inode2,index2);
                        cout<<"修改完毕";
                    } else {
                        cout<<"输入不合法!!!";   goto end;
                    }
                }
            }
        } else {
            cout<<"输入不合法!!!";   goto end;
        }
    } else {
        cout<<"你无权修改该子目录或文件";   goto end;
    }
        } else {
            cout<<"不存在该子目录或文件";   goto end;
        }
        /*修改当前节点和子节点*/
        char tmpbuf[9];
        _strtime(tmpbuf);
        strcpy(inode.ctime,tmpbuf);
        strcpy(inode2.ctime,tmpbuf);
        writeinode(inode,road[num–1]);
        writeinode(inode2,index2);
    end:return;
    }
```

5. 改变文件的所有者

void chown(char *name)

```
    /*改变当前目录下指定文件的文件所有者(如所有者在另一个组，那么组也要改)*/
    void chown(char *name)
    {
      INODE inode,inode2;
      readinode(road[num–1],inode);          /*当前节点写入节点对象*/
```

```
int i,index2;                          /*i 为目录项下标，index2 为目录项中节点号*/
if( havewpower(inode) ){
    if( havesame(name,inode,i,index2) ){
        readinode(index2,inode2);
        if( havewpower(inode2) ){
            char owner2[6];
            char auser2[6];
            char group2[6];
            cout<<"请输入改后的文件所有者：";
            cin>>owner2;
            bool is=false;             /*判断输入的也是合法用户名*/
            user.open("user.txt",ios::in);
                for( int n=0;n<usernum;n++ ) {
                /*用户名 18n+0~18n+5，密码 18n+6~18n+11，用户组 18n+12~18n+17*/
                    user.seekg(18*n);
                    user>>auser2;
                    if( !strcmp(owner2,auser2) ) {
                        is=true;
                        user.seekg(18*n+12);
                        user>>group2;
                        break;
                    }
                }
            user.close( );
            if( is ) {
                strcpy(inode2.owner,owner2);
                strcpy(inode2.group,group2);
                writeinode(inode2,index2);
                cout<<"修改成功";
                /*修改当前节点和子节点*/
                char tmpbuf[9];
                _strtime(tmpbuf);
                strcpy(inode.ctime,tmpbuf);
                strcpy(inode2.ctime,tmpbuf);
                writeinode(inode,road[num-1]);
                writeinode(inode2,index2);
            } else {
                cout<<"不存在该用户，修改失败";
            }
} else {
```

```
                cout<<"你没有权限";
            }
        } else {
            cout<<"不存在该子目录或文件";
        }
    } else {
        cout<<"你没有权限";
    }
}
```

6. 改变文件的所属组

void chgrp(char *name)

```
/*改变当前目录下指定文件的所属组*/
void chgrp(char *name)
 {/*改变文件所属组*/
  INODE inode,inode2;
  readinode(road[num-1],inode);          /*当前节点写入节点对象*/
  int i,index2;                          /*i 为目录项下标，index2 为目录项中节点号*/

  if( havewpower(inode) ) {
     if( havesame(name,inode,i,index2) ) {
readinode(index2,inode2);
 if( havewpower(inode2) ){
     char group2[6];
     char agroup2[6];
     cout<<"请输入改后的文件所属组：";
     cin>>group2;
     bool is=false;                      /*判断输入的也是合法用户名*/
     user.open("user.txt",ios::in);
        for( int n=0;n<usernum;n++ ) {
        /*用户名 18n+0~18n+5，密码 18n+6~18n+11 用户组 18n+12~18n+17*/
                user.seekg(18*n+12);
                user>>agroup2;
                if( !strcmp(group2,agroup2) )   {
                        is=true;
                        break;
                }
            }
  user.close( );
  if( is ){
          strcpy(inode2.group,group2);
```

```
                writeinode(inode2,index2);
                cout<<"修改成功";
                /*修改当前节点和子节点*/
                char tmpbuf[9];
                _strtime(tmpbuf);
                strcpy(inode.ctime,tmpbuf);
                strcpy(inode2.ctime,tmpbuf);
                writeinode(inode,road[num-1]);
                writeinode(inode2,index2);
            } else {
                cout<<"不存在该组，修改失败";
            }
        } else {
            cout<<"你没有权限";
        }
    } else {
        cout<<"不存在该子目录或文件";
    }
} else {
    cout<<"你没有权限";
    }
}
```

7. 改变文件名

void chnam(char *name)

```
/*改变当前目录下指定文件的文件名*/
void chnam(char *name)
{/*改变文件名*/
  INODE inode,inode2;
  readinode(road[num-1],inode);                /*当前节点写入节点对象*/
  int i,index2;                                /*i 为目录项下标，index2 为目录项中节点号*/
  if( havewpower(inode) ) {
     if( havesame(name,inode,i,index2) ){
       readinode(index2,inode2);
        if( havewpower(inode2) ) {
            char name2[14];
            cout<<"请输入更改后的文件名：";
            cin>>name2;
            disk.open( "disk.txt",ios::in | ios::out | ios::nocreate );
                disk.seekp( 514*inode.addr[0]+36*i);
                disk<<setw(15)<<name2;
```

```
        disk.close( );
        /*如果是目录文件，其下的所有目录项都要改 parfname[14] */
        if(inode2.mode[0]=='d'){
              disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
                   for(int i=0;i<(inode2.fsize/36);i++)/*遍历所有的目录项*/
                    {
                          disk.seekg(514*inode2.addr[0]+36*i+18);
                          disk<<setw(15)<<name2;
                    }
              disk.close( );
        }
        cout<<"更改成功";
        /*修改当前节点和子节点*/
        char tmpbuf[9];
        _strtime(tmpbuf);
        strcpy(inode.ctime,tmpbuf);
        strcpy(inode2.ctime,tmpbuf);
        writeinode(inode,road[num-1]);
        writeinode(inode2,index2);
    } else {
                cout<<"你没有权限";
            }
        } else {
              cout<<"不存在该子目录或文件";
        }
    } else {
          cout<<"你没有权限";
    }
}
```

7.3.3.5 命令解析相关函数

命令解析主要面向终端用户，接受用户命令，并将其映射到具体的函数调用，从而实现对不同操作请求的执行。与之相对应的函数包括以下几个。

1. **取命令**

```
void getcommand( )
    /*命令解析函数*/
    void getcommand( )
    {
        char commond[10];
```

```
bool have;
for(;;){      /*接受并解释命令*/
      have=false;/*记录命令接受否*/
      cout<<'\n';
      printroad( );
      cout<<">";
      cin>>commond;
      if( !strcmp(commond,"cd") ) {     /*改变当前目录，若为文件名，则切换到子目录；若为“..”
                                          则切换到父目录；若为“/”则切换到根目录*/
            have=true;
            char string[100];        /*若是路径，至少有一个“/”以root开头，不以“/”结尾*/
            cin>>string;
            cd(string);
    }
    if( !strcmp(commond,"mksome") ) { /*构建基本文件结构*/
            have=true;
            cout<<"bin";       mkdir("bin");
            cout<<"\ndev";    mkdir("dev");
            cout<<"\nlib";    mkdir("lib");
            cout<<"\netc";    mkdir("etc");
            cout<<"\nusr";    mkdir("usr");
            cout<<"\ntmp";    mkdir("tmp");
}
/*当前目录下的操作*/
if( !strcmp(commond,"mkdir") ) {   /*在当前目录下创建目录(规定目录文件只占一个盘块)*/
      have=true;
      char dirname[14];
      cin>>dirname;
      mkdir(dirname);
}
if( !strcmp(commond,"rmdir") ) {   /*在当前目录下删除目录：1.只能删除空目录；2. 提醒+
                                     删除所有子目录*/
      have=true;
      char dirname[14];
      cin>>dirname;
      rmdir(dirname,road[num-1]);
  }
  if( !strcmp(commond,"mk") ) {     /*在当前目录下创建数据文件*/
      have=true;
      char filename[14];
```

```
        cin>>filename;                          /*可输入 1～4 盘块的内容*/
        char content[2048];
        cout<<"请先输入文件内容(1～2048 位)：";
        cin>>content;
        mk(filename,content);
    }
    if( !strcmp(commond,"cp") ) {       /*把指定目录下的指定文件复制到当前目录下*/

        have=true;
        char string[100];               /*路径至少有一个“/”以 root 开头，不以“/”结尾*/
        cin>>string;
        cp(string);
    }
    if( !strcmp(commond,"rm") ) {       /*删除当前目录下的数据文件*/
        have=true;
        char filename[14];
        cin>>filename;
        rm(filename);
    }
    if( !strcmp(commond,"cat") ) {/*显示当前目录下指定数据文件的内容*/
        have=true;
        char filename[14];
        cin>>filename;
        cat(filename);
    }
    if( !strcmp(commond,"chmod") ) {    /*改变文件权限*/
        have=true;
        char name[14];
        cin>>name;
        chmod(name);
     }
     if( !strcmp(commond,"chown") ) { /*改变文件所有者(如所有者在另一个组，那么组也要
                                        改变)*/
        have=true;
        char name[14];
        cin>>name;
        chown(name);
     }
     if( !strcmp(commond,"chgrp") ) { /*改变文件所属组(???)*/
        have=true;
```

```
        char name[14];
        cin>>name;
        chgrp(name);
    }
    if( !strcmp(commond,"chnam") ) {          /*改变文件名*/
        have=true;
        char name[14];
        cin>>name;
        chnam(name);
    }
    /*对当前目录的操作*/
    if( !strcmp(commond,"pwd") ) {            /*显示当前目录*/
        have=true;
        cout<<"您的当前目录为："<<curname;
    }
    if( !strcmp(commond,"ls") ) {             /*显示所有子目录(必须保证是目录文件) */
        have=true;
        ls( );
    }
    /*用户组操作*/
    if( !strcmp(commond,"login") ) {          /*用户登录*/
        have=true;
        cout<<"账号"<<auser<<"已注销\n";
        while( !login( ) )
            cout<<"wrong !!!\n";
        cout<<"login success"<<endl;
        cout<<"******Welcome "<<auser<<"******";
    }
    if( !strcmp(commond,"passwd") ) {         /*改变用户口令*/
        have=true;
        changepassword( );
    }
    if( !strcmp(commond,"reset") ) {          /*系统重置(调用初始化函数)前有非正常退出时，
                                                一定要用*/
        have=true;
        initial( );
        cout<<"系统已重置";
    }
    if( !strcmp(commond,"exit") ) {           /*退出*/
        have=true;
```

```
                return;
            }
            if( have==false ) {                    /*都不被接受*/
                cout<<commond<<" is not a legal command!!!";
            }
        }
    }
```

2. **输出路径**

```
void printroad(void)
    /*根据节点路径，输出路径*/
    void printroad(void)
    {
        cout<<"root";
        INODE inode;
        int nextindex;
        char name[14];

        for(int i=0;i+1<num;i++){
            readinode( road[i],inode );
            disk.open("disk.txt",ios::in | ios::out | ios::nocreate );
            for(int j=0;j<(inode.fsize/36);j++) {   /*遍历所有的目录项*/
                disk.seekg(514*inode.addr[0]+36*j);
                disk>>name;
                disk>>nextindex;
                if(nextindex==road[i+1]) {
                    cout<<'/';
                    cout<<name;
                    break;
                }
            }
          disk.close( );
        }
    }
```

7.3.4 程序框架

主函数首先会根据控制文件 control.txt 的头一个位(初始化确定位)来确定是否进行初始化。如果初始化，那么将只有一个根目录。因为一旦初始化后，初始化确定位就会置为 0，所以系统只会在第 1 次进入时初始化，以后都是在前面建立的文件结构上进行操作的。如果一定要进行系统初始化，可把 control.txt 的头一个位置为 1，或者使用命令 reset。

初始化过程包括对超级块(0#盘块)的初始化、对根目录文件节点(0#节点)的初始化和对数据盘块的初始化。

进入系统时，当前路径是根目录。然后是登录请求。登录后，超级块读入主存(由一个结构对象 superblock 模拟)，进入命令解析层，对用户的不同命令执行不同的操作。退出系统前，把主存的超级块再写回 disk.txt。

初始化代码框架如下：

```
void main( )
{
    control.open("control.txt",ios::in | ios::out | ios::nocreate);
    int i;
    control>>i;
    control.close( );
    if(i!=0) {                          /*不为 0 就初始化*/
        initial( );
    }
    control.open("control.txt",ios::in | ios::out | ios::nocreate);
    control.seekp(0);
    control<<0;                         /*默认是在上次基础上继续下去，不用再初始化*/
    control.close( );
    strcpy(curname,"root");             /*当前目录文件名为 root*/
    road[0]=0;                          /*当前目录路径(存放从根目录到这里的节点号) */
    num=1;                              /*最后的 road[num-1]为当前目录文件节点号*/
    cout<<"请登录系统\n";
    while( !login( ) )                  /*登录为止*/
        cout<<"wrong !!!\n";
    cout<<"      login success"<<endl;
    cout<<"******Welcome "<<auser<<"******";
    readsuper( );
   getcommand( );                       /*命令解析函数*/
    writesuper( );
}

void initial( )
{/*初始化*/
    /*用户组的初始化--->>>最多用户名 5 位，密码 5 位，用户组 5 位*/
    user.open("user.txt",ios::in | ios::out | ios::nocreate | ios::trunc);
    user<<setw(6)<<"adm";       /*用户名 18n+0 ~ 18n+5*/
    user<<setw(6)<<"123";       /*密码 18n+6 ~ 18n+11*/
    user<<setw(6)<<"adm";       /*用户组 18n+12 ~ 18n+17*/
```

```
user<<setw(6)<<"cnj";
user<<setw(6)<<"123";
user<<setw(6)<<"adm";
user<<setw(6)<<"jtq";
user<<setw(6)<<"123";
user<<setw(6)<<"guest";
user.close( );
/*disk 初始化*/
disk.open("disk.txt",ios::in | ios::out | ios::nocreate | ios::trunc);
if(!disk){
        cout<<"can't use the disk \n";
        exit(1);
 }
 int i;
for(i=0;i<100;i++) {            /*先全部填“空”*/
        disk<<setw(512)<<' ';
        disk<<'\n';
 }
/*超级块初始化 0#盘块*/
disk.seekp(0);                  /*空闲节点号栈 80×3=240 B (0～239) */
disk<<setw(3)<<-1;              /*0#的节点已使用，就赋-1(0～2) */
for(i=1;i<=79;i++){
        disk<<setw(3)<<i;
 }
 disk<<setw(3)<<79;             /*空闲节点栈指针=当前节点数(240～242)*/
 for(i=0;i<10;i++) {
/*空闲盘块号栈 10×3=30 B (243～270) 11#盘块已使用末尾盘块为 21#*/
        disk<<setw(3)<<i+12;
 }
 disk<<setw(3)<<10;             /*空闲节点栈指针=当前栈中盘块数(271～273) */
 disk<<setw(3)<<80;             /*空闲节点总数(274～276) */
 disk<<setw(3)<<89;             /*空闲盘块总数(277～279) */
                                /*空闲节点栈互斥访问标志(尚不用) */
                                /*空闲盘块栈访问标志(尚不用) */
/*根目录文件节点的初始化，是 1#盘块开始的*/
disk.seekp(514);
disk<<setw(6)<<0;               /*int fsize;文件大小*/
disk<<setw(6)<<1;               /*int fbnum;文件盘块数*/
disk<<setw(3)<<11;
disk<<setw(3)<<0;               /*指向 0#表示没有指向*/
```

```
disk<<setw(3)<<0;
disk<<setw(3)<<0;
disk<<setw(3)<<0;                    /*int addr1;一个一次间址*/
disk<<setw(3)<<0;                    /*int addr2;一个两次间址*/
disk<<setw(6)<<"adm";                /*char owner[6];文件所有者，根目录由超级用户所有*/
disk<<setw(6)<<"adm";                /*char group[6];文件所属组*/
disk<<setw(11)<<"drwxrwxrwx";        /*文件类别及存储权限*/
char tmpbuf[9];
_strtime(tmpbuf);
disk<<setw(9)<<tmpbuf;               /*char ctime[9];最近修改时间*/
/*根目录文件初始化，如果尚无子目录，不用初始化*/
/*空闲盘块初始化，只用初始化记录盘块 i# 成组链接最后记录盘块最后记录号后要压个 0*/
for(i=21;i<100;i++){
    if(i%10==1){
        disk.seekp(514*i);           /*定位记录盘块*/
        for(int j=0;j<10;j++) {
            int temp=i+j+1;
            if(temp<100) {
                disk<<setw(3)<<temp;
            }
        }
    }
    disk<<setw(3)<<0;/*最后记录盘块最后记录号后要压个 0*/
    disk.close( );
}
```

第 8 章　时钟与定时器

8.1　实 验 目 的

- 了解 Linux 时钟和定时器机制。
- 了解 Linux 中与时间相关的函数。
- 掌握如何使用 Linux 定时器。

8.2　背 景 知 识

8.2.1　定时器机制的概念

在计算机系统中，很多活动都是由定时测量来驱动的，例如，当用户停止使用计算机的控制台之后，屏幕会自动关闭，这得归因于定时器，它允许内核跟踪按键或移动鼠标后到现在过了多少时间。用户也可能接收到来自系统的警告信息，希望其删除一组不用的文件，这是由于有一个系统程序能够识别长时间未被访问的所有用户文件，为了进行这些操作，程序必须能从每个文件中检索到文件的最后访问时间。时钟也是操作系统进行调度工作的重要工具，如让交互进程进行时间片轮转、让实时进程定时发出或接收控制信号、定时唤醒或阻塞进程、控制两个进程的上下文切换、对进程进行计时和记账，利用定时器能确保操作系统必要时获得控制权，陷入死循环的进程最终会因时间片耗尽被迫出让 CPU。

这样的时间标记必须由内核自动设置。在 Linux 操作系统中，有两种定时测量：

- 维护系统当前时间和日期。
- 间隔定时器，又称定时器。

8.2.2　时间维护

计算机中有两个时钟：硬件时钟和软件时钟。硬件时钟是由电池供电的 CMOS 时钟，其作用是提供计时标准和产生时钟中断，是最底层的时钟信息；软件时钟的初始值在操作系统启动时从 CMOS 获得，以后由内核来维护。Linux 内核提供的系统时间服务是国际标准时间：公元 1970 年 1 月 1 日 00:00:00 以来经过的累计秒数，这种秒数是以数据类型 time_t 表示的，它实际是一个长整型数，用全局变量 jiffies 表示，其定义在头文件 time.h 中。jiffies 在每个时钟周期更新一次，系统时钟和定时器间隔都根据这个变量计算出来。这种时间也称为日历时间，日历时

间包括时间和日期。Linux 在这方面与其他操作系统的区别如下：

- 以国际标准时间而非本地时间计时。
- 可自动进行转换，如变换到夏令时。
- 将时间和日期作为一个量值保存。

time( )函数返回当前时间和日期。

```
#include <time.h>
time_t time( time_t *tloc);
```

时间值作为函数值返回。如果参数非 NULL，则时间值也存放在由 tloc 指向的主存单元内。下面这个简单程序 envtimetest.c 演示了 time( )函数的用法。

```
/*****envtimetest.c*****/
#include <time.h>
#include <stdio.h>
#include <unistd.h>

int main( )
{
        int i;
        time_t the_time;
        for ( i=1; i<=10; i++ ) {
                the_time = time( (time_t *) 0 );
                printf("The time is %ld\n ", the_time);
                sleep(2);
        }
        exit(0);
}
```

运行这个程序，它会在 20s 时间内每两秒钟打印一次底层的时间值。

```
$ ./envtime test
The time is 1044695820
The time is 1044695822
The time is 1044695824
The time is 1044695826
The time is 1044695828
The time is 1044695830
The time is 1044695832
The time is 1044695834
The time is 1044695836
The time is 1044695838
```

以从 1970 年开始计算的秒数来表示时间和日期，对测量某些事件持续的时间是很有用的，

可以把它考虑为只是简单地把两次调用 time( )得到的值相减就行了。但是考虑兼容性，应该采用 difftime( )函数，该函数用来计算两个 time_t 值之间的秒数并以 double 类型返回。

```
#include <time.h>
double difftime( time_t time1, time_t time2);
```

difftime( )函数计算两个时间值之间的差，并将 time1-time2 的值作为浮点数返回。对 Linux 来说，time( )函数的返回值是秒数，可以对它进行处理，但考虑到最大限度地增加可移植性，则最好使用 difftime( )。

在得到以秒计的时间值之后，通常需要调用另一个时间函数将其变换成可读的时间和日期。图 8-1 说明了各种时间函数之间的关系，图中以虚线表示的 4 个函数 localtime( )、mktime( )、ctime( )、strftime( )都受环境变量 TZ 的影响。

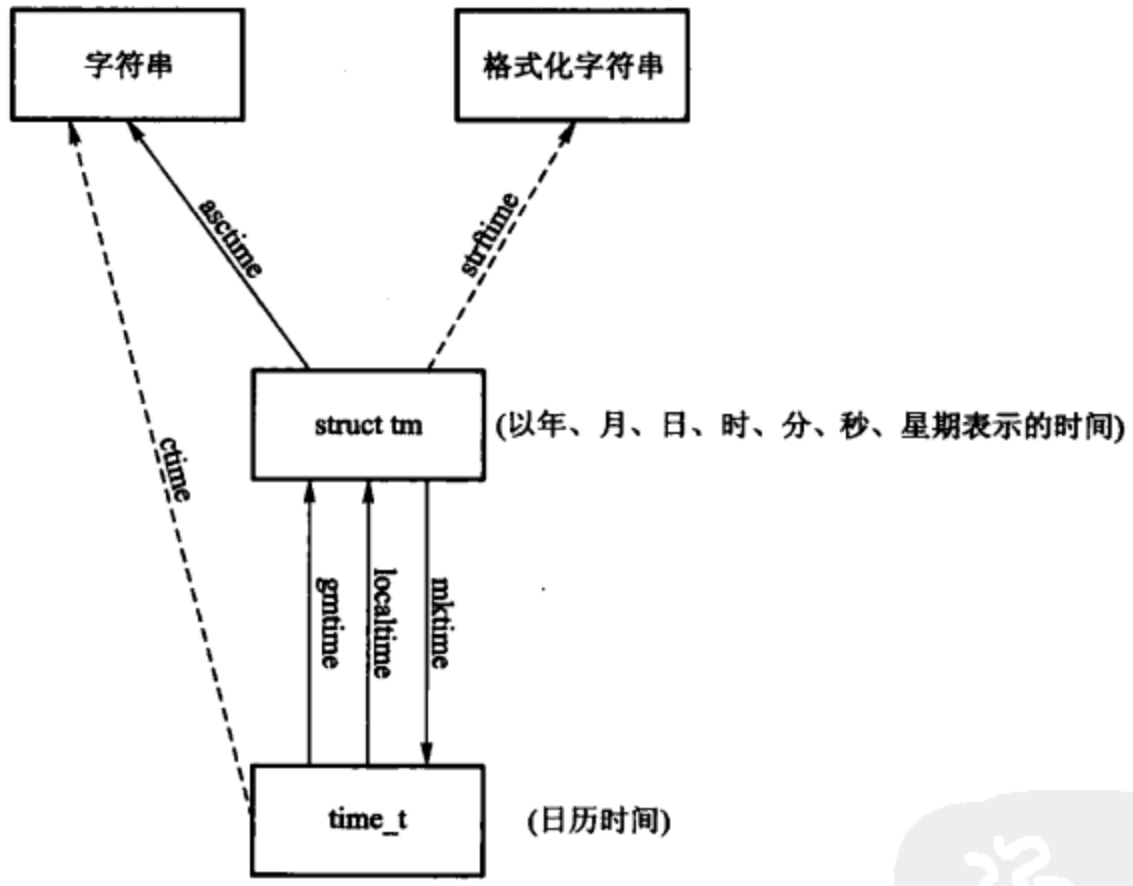


图 8-1 各种时间函数之间的关系

1. localtime( )和 gmtime( )函数

两个函数 localtime( )和 gmtime( )将日历时间变换成以年、月、日、时、分、秒、星期表示的时间，并将这些存放在一个 tm 结构中。

```
struct tm{
    int tm_sec;          /* 秒数*/
    int tm_min;          /* 分钟*/
    int tm_hour;         /*小时*/
    int tm_mday;         /*日*/
```

```
    int tm_mon;     /*月*/
    int tm_year;    /*年*/
    int tm_wday;    /*星期几*/
    int tm_yday;    /*从 1 月 1 日开始的时间*/
    int tm_isdst;   /*是否夏令时*/
}
```

秒数可以超过 59，因为可以表示润秒。注意，除月、日字段，其他字段的值都以 0 开始。如果夏令时生效，则夏令时标志值为正；否则，该标志值为负。函数 localtime( )和 gmtime( )的原型为：

```
#include <time.h>
struct tm *gmtime(const time_t *calptr);
struct tm *localtime(const time_t *calptr);
```

这两个函数都返回指向 tm 结构的指针。localtime( )和 gmtime( )之间的区别是：localtime( )将日历时间变换成本地时间(考虑到本地时区和夏令时标志)，而 gmtime( )则将日历时间变换成国际标准时间的年、月、日、时、分、秒、星期几。

2. mktime( )函数

函数 mktime( )以本地时间的年、月、日等作为参数，将其变换成 time_t 值。

```
#include <time.h>
time_t mktime(struct tm *tmptr);
```

函数在成功时会返回日历时间，失败时返回–1。

3. asctime( )与 ctime( )函数

asctime( )和 ctime( )函数产生长度为 26 的字符串，这以 date 命令的系统默认输出形式类似：

```
Sun Jun  6 12:30:34 1999
```

这两个函数的区别在于，asctime( )的参数是一个 struct tm 结构的指针，而 ctime( )的参数是一个 time_t 指针。

```
#include <time.h>
char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timeval);
ctime( )函数等效于调用 asctime(localtime(timeval));
```

4. strftime( )函数

strftime( )函数与 printf( )函数非常相似，该函数的原型为：

```
#include<time.h>
size_t strftime( char *buf, size_t maxsize, const char *format, const struct tm *tmptr);
```

该函数的最后一个参数要求格式化的时间值，格式化结果存放在一个长度为 maxsize 个字符的 buf 数组中。format 参数控制时间值的格式，如同 printf( )函数一样，变换说明的形式是百分号后跟一个特定字符。format 中的其他字符则按原样输出。表 8-1 列出 ANSI C 规定的变换说明。

表 8-1 strftime( )函数转换控制符

转换控制符	说　明	转换控制符	说　明
%a	星期几缩写	%S	秒
%A	星期几全称	%u	星期几，1~7
%b	月份缩写	%U	一年中的第几周(周日是一周的第 1 天)
%B	月份全称	%V	一年中的第几周(周一是一周的第 1 天)
%c	日期和时间	%w	星期几，0~6
%d	月份中的日期，01~31	%x	本地格式的日期
%H	小时	%X	本地格式的时间
%I	12 进制的小时，01~12	%y	年份减去 1900
%j	年份中的日期	%Y	年份
%m	年份中的月份	%Z	时区名
%M	分钟	%%	字符%
%p	a.m.(上午)或 p.m.(下午)		

ctime( )、localtime( )、mktime( )、strftime( )4 个函数受到环境变量 TZ 的影响，如果定义过 TZ，则这些函数将使用其值以代替系统默认时区，如果 TZ 定义为空串，则使用国际标准时间。

8.2.3 定时器

操作系统能保证调度准时进行的机制是定时器，从硬件上来讲必须包含一个可周期性中断、可编程的间隔定时器，这个周期性中断被称为系统时钟滴答，它像节拍器一样来组织系统任务。从软件上来讲必须有程序来实现定时器在硬件中断到来时处理任务调度。定时器(timer)是 Linux 提供的一种定时服务机制，它所起的作用是在某个特定的时刻唤醒某个进程来完成相关工作。

间隔定时器可由以下两个方面来刻画：

- 发送信号所必需的频率，如果只需要产生一个信号，则频率为空。
- 在下一个信号被产生前所剩余的时间。

需要注意，这些定时只保证在要求的时间已过去之后，定时器将被执行，但是不可能预知恰好在什么时候它们会被执行。

操作系统中分两类定时器：系统定时器和进程定时器。

8.2.3.1 系统定时器

Linux 的系统定时器又分两种，它们都具有对应的处理例程，必须在到达给定的系统时间时被进程调用，但实现方法有区别。第 1 种是老定时器机制，是由一个 32 位指针的静态数组定义的定时器，每个指针可指向一个 timer_struct 结构，而 timer_active 是活动定时器掩码，在系统初始化时入口被加到该数组中。

```
struct timer_struct {
```

```
    unsigned long expires;              /*定时器激活时间*/
    void (*fn)(void);                   /*定时器处理函数*/
};
```

第 2 种是新定时器机制，突破 32 个定时器的限制，使用一个 timer_list 数据结构的链表，按定时器到期时间的升序排列，定时器中 expires 给出该定时器被激活的时间，而* function( ) 指出定时器激活后的处理函数。

```
struct timer_list {
    struct list_head entry;             /*定时器链表*/
    unsigned long expires;              /*定时器激活时间*/
    unsigned long data;                 /*传给处理函数的参数*/
    void (*function)(unsigned long);    /*定时器处理函数*/
};
```

两种定时器都使用 jiffies 值作为到期比较时间，例如，某个定时器要在 2s 之后到期，则必须将 2s 转换成对应的 jiffies 值，加上当前的系统时间(也是以 jiffies 为单位)后，得到的便是该定时器到期的系统时间 expires。每次系统时钟滴答到来时，定时器 bottom half 处理例程被标记为活动状态，这样当调度程序下次运行时，定时器队列能获得处理。定时器 bottom half 处理例程要处理两种类型的系统定时器。对于老系统定时器，检查 timer_active 中被置位的位掩码，以便确定活动的定时器。如果一个活动的定时器到期，便调用对应的定时器例程，timer_active 对应位被清除。对于新系统定时器，检查链表中的 timer_list 数据结构。每个到期的定时器从链表中移出，对应的定时器例程被调用。新的定时器机制的优点是能传递参数 data 到定时器例程中。

8.2.3.2 进程定时器

由于进程运行时有用户态和内核态，所以进程执行时也有进程自身在用户模式下花费的执行时间，以及内核代表进程在内核模式下花费的执行时间，内核为每个进程累计时间并提供 3 种不同的进程定时器，因而就有与进程相关的 3 种时间间隔：ITIMER_REAL、ITIMER_VIRTUAL 和 ITIMER_PROF。进程可以运行一种或多种定时器，在进程的 task_struct 数据结构中记录所有的必要信息，可使用函数建立、启动、并停止它们或者读取当前还剩余的激活时间。ITIMER_VIRTUAL 和 ITIMER PROF 定时器的处理方式相同，每一次时钟周期，当前进程的定时器值递减，如果到期，就直接产生适当的定时信号。实现 ITIMER_REAL 定时器时，使用系统的 timer 链表，当它到期的时候，是由时钟 bottom_half 处理例程把它从队列中删除并调用内部定时器处理程序，这个处理例程完成的工作就是产生 SIGALRM 信号。

如果进程对定时器的要求不是很精确，用 alarm( )函数就可以实现定时器。使用 alarm( )函数先设置一个时间值(以秒为单位)，在将来的某个时刻该时间值会被超过，当所设置的时间值被超过后，产生 SIGALRM 信号。如果不忽略或不捕捉此信号，则其默认动作是终止该进程。

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

其中，参数 seconds 的值是秒数，经过指定的 seconds 秒后会产生信号 SIGALRM。这种情况下，每个进程只能有一个定时器时间。如果在调用 alarm( )时，以前已为该进程设置过定时器，而且它还没有超时，则该定时器时间的余留值作为本次 alarm( )函数的值返回，以前登记的定时器时间则被新值替换。如果拥有以前登记的尚未超过的定时器时间，而且 seconds 的值为 0，则取消以前的定时器时间，其余留值仍然作为函数的返回值。

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void sigalrm_fn(int sig)
{
    printf("alarm!\n");
    alarm(2);
    return;
}
int main(void)
{
    signal(SIGALRM, sigalrm_fn);
    alarm(1);
    while(1) pause( );
}
```

在上例中，进程设置了一秒后被激活的定时器，定时器被激活时，程序将打印“alarm!”。下面，利用 alarm( )和 pause( )函数模拟实现 sleep( )函数。

```
/*****alarmtest.c*****/
#include <signal.h>
#include <unistd.h>
static void sig_alrm( int sig no )
{
    return ;
}

unsigned int mysleep( unsigned int nsecs )
{
    if ( signal(SIGALRM, sig_alrm) == SIG_ERR)
        return (nsecs);
    alarm(nsecs);          /*设置定时器*/
    pause( );              /*等待信号*/
    return ( alarm(0) );
}
```

```
int main( )
{
    printf("sleep\n");
    mysleep(10);
    printf("wake up\n");
}
```

该程序利用 alarm( )和 pause( )两个函数模拟实现 sleep( )函数，pause( )函数会将进程挂起，直到捕捉到一个信号。这种简化的实现有多个问题，将在 8.3.2 小节中对该程序进行改进，实现一个可靠的类 sleep( )函数。

alarm( )函数实现的定时器精度较低，如果进程需要使用精度较高的定时器，可以通过 setitimer( )函数，函数原型为：

```
#include <sys/time.h>
int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);
```

该函数的第 1 个参数 which 指定定时器采用的策略，通常可选用下列 3 种定时器中的一种。

• ITIMER_REAL：这种定时器使用实时计数，反映进程走过的实际时间，不管进程在何种模式下运行，它总是在计数。当定时到达时，会发给进程一个 SIGALRM 信号。alarm( )函数实际上是采用 ITIMER_REAL 策略。

• ITIMER_VIRTUAL：这种定时器使进程在用户模式(进程本身执行)执行的过程中计数，反映进程走过的虚拟时间，当计数完毕时发送 SIGVTALRM 信号给进程。

• ITIMER_PROF：这种定时器是进程在用户模式(进程本身执行)和内核模式(系统代表进程执行)的时候都计时，反映进程处于活跃状态下走过的时间。与 ITIMER_VIRTUAL 比较，这个定时器记录的时间多于该进程在内核模式执行过程中消耗的时间。当定时到达时，会发送 SIGPROF 信号。

内核为每个进程累计时间并管理进程的不同定时器，调度程序需要使用每个进程有关定时器的值。这些定时器周期性地被初始化为指定值，都进行递减操作，来反映时间的流逝，当定时器为 0 时，就发出一个中断信号，这时系统或应用程序就会进行相应处理。

间隔定时器只能执行一次，但能周期性循环。setitimer( )的第二个参数指向一个 itimerval 类型的结构，该结构指定定时器初始的持续时间(以秒和微秒为单位)以及定时器被自动重新激活后使用的持续时间(对于一次性执行的定时器而言为 0)。setitimer( )的第 3 个参数是一个指针，它是可选的，指向一个 itimerval 类型的结构，函数将先前定时器的参数填充到该结构中。

```
struct itimerval {
    struct timeval it_interval;
    struct timeval it_value;
 };
struct timeval {
    long tv_sec;
    long tv_usec;
```

};

it_interval 指定间隔时间，it_value 指定初始定时时间。如果只指定 it_value，就是实现一次定时；如果同时指定 it_interval，则超时后，系统会重新初始化 it_value 为 it_interval，实现重复定时；两者都清零，则会清除定时器。tv_sec 提供秒级精度，tv_usec 提供微秒级精度，以值大的为先。

下面是 setitimer( )函数调用的一个简单示范，在该例中，每隔 1s 发出一个 SIGALRM，每隔 0.5s 发出一个 SIGVTALRM 信号。

```
/*****setitimertest.c*****/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>

void sigroutine(int signo)
{
    switch (signo){
        case SIGALRM:
            printf("Catch a signal -- SIGALRM \n");
            signal(SIGALRM, sigroutine);
            break;
        case SIGVTALRM:
            printf("Catch a signal -- SIGVTALRM \n");
            signal(SIGVTALRM, sigroutine);
            break;
    }
    return;
}

int main( )
{
    struct itimerval value, ovalue, value2;
    signal(SIGALRM, sigroutine);          /*设置信号处理函数*/
    signal(SIGVTALRM, sigroutine);        /*设置信号处理函数*/
    value.it_value.tv_sec = 1;            /*设置定时器时间*/
    value.it_value.tv_usec = 0;
    value.it_interval.tv_sec = 1;
    value.it_interval.tv_usec = 0;
```

```
        setitimer(ITIMER_REAL, &value, &ovalue);
        value2.it_value.tv_sec = 0;          /*设置定时器时间*/
        value2.it_value.tv_usec = 500000;
        value2.it_interval.tv_sec = 0;
        value2.it_interval.tv_usec = 500000;
        setitimer(ITIMER_VIRTUAL, &value2, &ovalue);
        for( ; ; );
    }
```

Linux 还支持 POSIX 1003.1b 标准定时器，该定时器为用户态程序引入一种新型定时器，尤其是针对多线程和实时应用程序，这些定时器常被称为 POSIX 定时器。要执行 POSIX 定时器，必须向用户态程序提供 POSIX 时钟，也就是说，虚拟时间源预定义分辨度和属性。只要应用程序想使用 POSIX 定时器，它就创建一个新的定时器资源并制定一个现存的 POSIX 时钟来作为定时基准。POSIX 定时器比传统间隔定时器更灵活、更可靠。它们之间有两个显著区别。

• 当传统间隔定时器到期时，内核会向进程发送一个 SIGALRM 信号来激活定时器。而当 POSIX 定时器到期时，内核可以向整个多线程应用程序发送各种信号，也可以向单个指定的线程发送信号。

• 如果传统间隔定时器到期了很多次，但用户态进程不能接收 SIGALRM 信号(例如由于信号被阻塞或者进程不处于运行态)，那么只有第 1 个信号被接收到，其他所有 SIGALRM 信号都丢失了。对于 POSIX 定时器来说会发生同样的情况，但进程可以调用 timer_getoverrun( )函数来得到自第 1 个信号产生以来定时器到期的次数。

8.3 实 验 内 容

8.3.1 实验 1 统计进程时间

8.3.1.1 实验说明

统计进程运行时在用户模式、内核模式及总的运行时间。需要使用 3 种定时器来测量进程的处理器使用率，同时使用 signal 机制建立信号处理程序，记录和统计进程的实际的、虚拟的和活跃的 3 种时间。

8.3.1.2 解决方案

1. 方案 1

在使用 setitimer( )函数设置定时器时，用户可以指定 3 种策略：ITIMER_REAL、ITIMER_VITRUAL 和 ITIMER_PROF，通过在一个进程中设置这 3 种定时器，就可以统计一个进程在用户模式、内核模式及总的运行时间。

首先需要定义 3 个计时变量，分别用于记录进程的 3 种运行时间，然后需要定义 3 种不同的定时器，设置这些定时器的发生时间，并将这些定时器的发生频率设置为一直发生。在设置完毕之后，每当定时器超时并产生一个信号之后，只需要增加相应计时变量的值便粗略地得到进程运行的 3 种时间。为了得到更加精确的运行时间，需要考虑如下情况：在定时器刚产生一个信号之后，程序便运行结束；在定时器信号产生后至程序结束前的时间也需要进行统计。这需要利用设置定时器时使用的 itimerval 结构。通过 setitimer( )设定定时器时，需要传入一个 itimerval 结构，当再次使用 getitimer( )取出该结构时，该结构的 it_value 成员指明当前至下次定时器超时所需要的时间，通过在进程结束前取出各个定时器的 itimerval 结构，就能够更精确地统计进程运行的 3 种时间。

```
struct itimerval {
        struct timeval it_interval;
        struct timeval it_value;
 };
struct timeval {
        long tv_sec;
        long tv_usec;
 };
```

程序框架如下：

```
#include <signal.h>
#include <sys/time.h>
#include <stdio.h>
static long real_secs=0, virt_secs=0, prof_secs=0;
static void sig_handler(int signo)
{
    switch(signo) {
        case SIGALRM:
            /*增加实际时间计数*/
            break;
        case SIGVTALRM:
            /*增加 VIRTUAL 时间计数*/
            break;
        case SIGPROF:
            /*增加 PROF 时间计数*/
            break;
    }
}
int main( )
{
    struct itimerval v;
```

```
        /*设置 SIGALRM、SIGVTALRM、SIGPROF 信号处理函数*/
        /*进程运行*/
        /*打印计数*/
    }
```

2. 方案 2

方案 1 中给出了一个测量本进程的 3 种运行时间的方法。如果需要测量任意其他进程的 3 种运行时间，方案 1 便不再适用。例如，在 Linux 中，time 命令可以测量任意进程的 3 种运行时间。用户可以思考如何完成此功能，并可以参考 time 命令的实现。time 命令的源代码可以从 http://ftp.gnu.org/pub/gnu/time/time-1.7.tar.gz 下载。该命令通过 times( )函数完成对进程的 3 种运行时间的测量。

8.3.2 实验 2 通过 alarm( )实现 sleep( )函数功能

8.3.2.1 实验说明

定时器 8.2.3 小节给出一个通过 alarm( )函数实现 sleep( )函数的例子。但是该例子还有多个问题需要解决。在本实验中，需要对该方案进行改进，实现一个可靠的类 sleep( )函数。

8.3.2.2 解决方案

sleep( )函数的原型为：

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

函数的作用是将进程挂起直到：已经经过 seconds 所指定的实际时钟时间；或者该进程捕捉到一个信号并从信号处理程序返回。

对于第 1 种情况，sleep( )函数的返回值为 0。当由于捕捉到某个信号 sleep( )提早返回时(第 2 种情况)，返回值是未睡足的秒数(所要求的时间减去实际睡眠时间)。

1. 简化实现的问题所在

在定时器一节给出的简化实现存在着以下 3 个问题。

① 如果程序已设置过针对 SIGALRM 信号的信号处理函数，在调用 mysleep( )函数后，该信号处理函数会被 mysleep( )的 SIGALRM 信号处理函数替代。

② 如果调用者已设置定时器，在 mysleep( )函数调用 alarm( )后，原先的定时器不再会被激活。

③ mysleep( )函数在调用 alarm( )和 pause( )之间有一个竞争条件。因为 alarm( )调用和 pause( )调用并不是一个原子操作。在一个繁忙的系统中，可能 alarm( )在调用 pause( )之前超时，并调用过信号处理程序。如果发生这种情况，在调用 pause( )后，如果没有捕捉到其他信号，则调用者将永远被挂起。

2. **问题解决方案**

针对第 1 个问题，解决方案是在使用 mysleep( )之前先保存原信号处理函数，在该函数返回前再恢复原有信号处理函数。通过可靠的信号处理调用 sigaction( )可以实现该需求。

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act, struct sigaction *&oact);
```

signo 为需要注册的信号，act 为新的信号处理函数，oact 为老信号处理函数。

针对第 2 个问题，解决方案如下：

- 检查第 1 次调用 alarm( )的返回值，如其小于本次调用 alarm( )的参数值，则只应等到前次设置的定时器超时。
- 如果前次设置定时器的超时时刻后于本次设置值，则在 mysleep( )函数返回之前，再次设置定时器，使其在预定时间再发生超时。

针对第 3 个问题，解决方案如下：

- 在调用 alarm( )函数前，通过 sigprocmask( )函数将 SIGALRM 信号屏蔽。
- 在调用 alarm( )函数后，需要通过一个函数解除 SIGALRM 信号的屏蔽，并进入睡眠等待信号的发生。但是解除信号屏蔽和进入睡眠等待信号发生需要是一个原子操作，否则就有可能丢失信号，使进程一直被挂起。sigsuspend( )函数可以实现该功能。

3 种解决方案的代码如下：

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
int sigsuspend(const sigset_t *sigmask);
```

在 sigprocmask 函数中，set 是需要屏蔽的信号集，oset 是原先设定的屏蔽信号集。进程需要记录该信号集，在 mysleep( )函数返回前进行恢复。sigsuspend( )函数中，sigmask 是需要解除屏蔽的信号集。

8.3.2.3 程序框架

```
#include <signal.h>
#include <stddef.h>

static void sig_alrm(void)
{
    return;
}

unsigned int mysleep(unsigned int nsecs)
{
    /*设置新的信号处理函数*/
    /*保存原有信号处理函数*/
```

```
        /*设置信号集，屏蔽 SIGALRM 信号*/
        /*调用 alarm( )*/
        /*解除 SIGALRM 屏蔽，并等待信号发生*/
        /*恢复原有信号处理函数*/
        /*如果原先设置的定时器尚未超时，则再次调用 alarm( )*/
        /*返回未睡足的时间*/
}
```

8.3.3 实验 3 基于单定时器实现任意数目的逻辑定时器

8.3.3.1 实验说明

在 8.3.2 小节(实验 2)中，仅仅只能使用一个定时器，在某些时候是不够的。为了让用户更加方便地使用定时器，在本实验中，需要基于原有的 Linux 定时器构造一组新的定时器函数，使进程可以设置任意数目的定时器，这些定时器的精度以秒为单位即可。

8.3.3.2 解决方案

1. 函数定义

可以定义这样的一组 API，供用户使用自定义的定时器，自定义定时器采用进程的总运行时间进行计时：

- unsigned long add_timer(unsigned long nsecs, timer_handler handler, void *data); 添加一个定时器，该定时器会在 nsecs 秒后被激活，handler 在定时器激活后会被调用。函数的返回值为自定义定时器的唯一标识符。
- unsigned long del_timer(int id); 删除一个定时器。函数的返回值为该定时器距离超时的秒数
- typedef void timer_handler(void *); 自定义的定时处理函数，会在函数被激活时调用。该处理函数能够在 add_timer( )时附带一个参数，当该函数被调用时，该参数会被传递给该函数。

2. 总体结构

实现自定义定时器的总体思想如图 8-2 所示。用户在进程空间中维护一个自定义定时器链表，在用户使用自定义定时器时：

① 先设置一个系统定时器，定时器每隔 1s 被激活。

② 为每一个自定义定时器维护一个数据结构。

③ 系统定时器被激活时，更新每一个自定义定时器的计数，如果发现一个自定义定时器超时，便执行其关联的定时处理函数。

3. 数据结构

对于每个定时器，需要定义的信息有：定时器剩余多少时间被激活；该定时器的唯一标识符是什么；该定时器的自定义定时处理函数是什么；定时处理函数被调用时需要被传递的参数

是什么；指向下一个定时器的指针。根据这几个要求，可以定义出自定义定时器需要的数据结构：

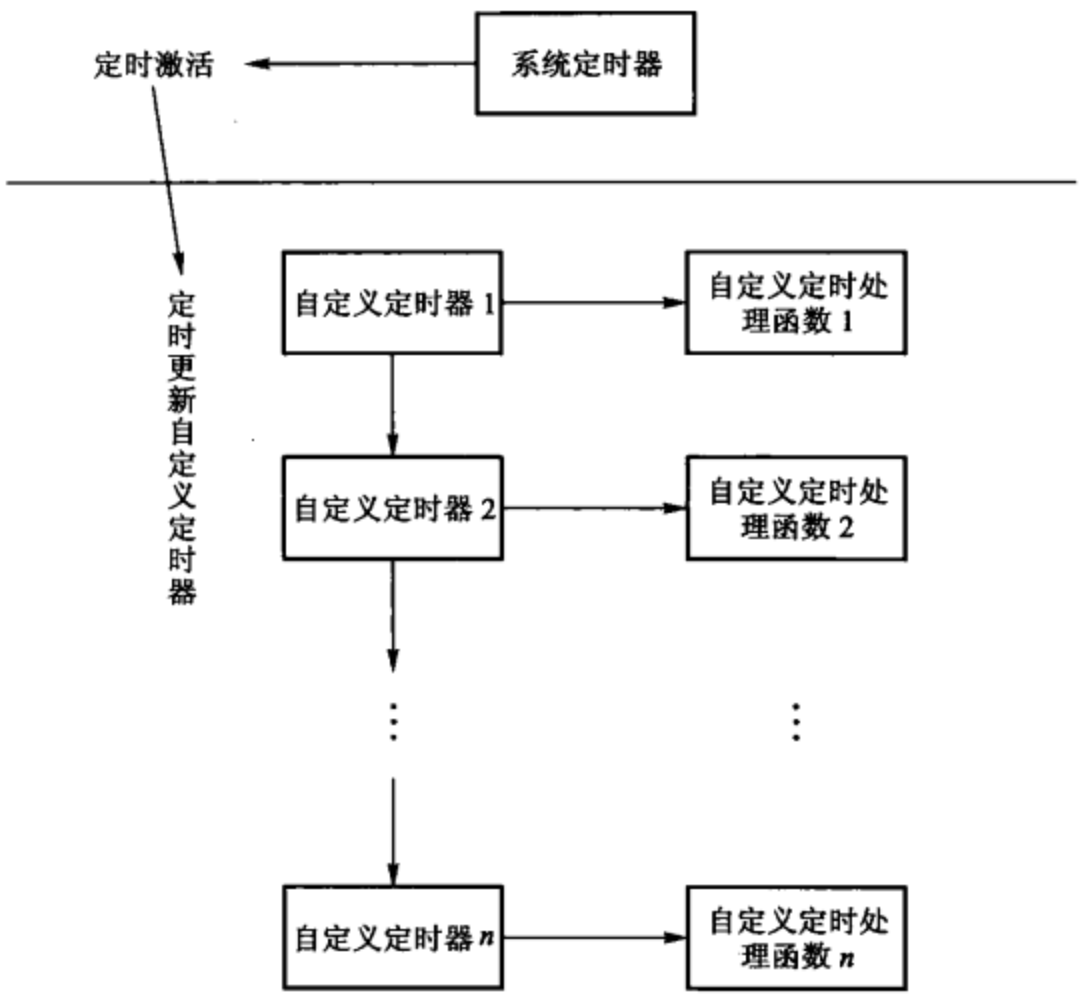


图 8-2 自定义定时器结构

```
struct my_timer{
    int id;
    long nsecs;
    timer_handler handler;
    void * data;
    my_timer *next;
}
```

为了给每个自定义定时器都能快速分配一个 ID，现定义变量 mytimer_id，每当用户调用 add_timer( )后，mytimer_id 的值便会自增，并将值赋予新添加的定时器。

4. 函数实现

（1）add_timer( )

用户第 1 次调用 add_timer( )时，需要对环境进行初始化：设置系统定时器，初始化 mytimer_id。在初始化之后，便可以将用户的定时器添加进链表，并设置一个唯一标识符。

（2）del_timer( )

该函数执行时，需要遍历用户维护的定时器链表。当找到该定时器时，需要将定时器从链表中删除，并将定时器中距离定时器超时的秒数返回给用户。

（3）SIGALRM( )信号处理函数

在 SIGALRM( )信号处理函数中，函数的任务是遍历自定义定时器，并将自定义定时器的计数值递减。当发现自定义定时器超时，调用该定时器关连的处理函数。在调用结束之后，需要删除该自定义定时器。

8.3.3.3 程序框架

```
#include <signal.h>
…
struct my_timer {
        int id;
        long nsecs;
        timer_handler handler;
        void * data;
        my_timer *next;
};

unsigned long mytimer_id;
struct my_timer *timer_head;
void init_timer( )
{
/*添加系统定时器，将定时器设定成间隔 1s 触发 1 次*/
}

/*系统定时器的信号处理函数*/
void sigroutine(int signo)
{
        /*扫描 timer_head 链表*/
        /*减少自定义定时器的计数*/
        /*激活超时的自定义定时器*/
        /*删除已经运行过的自定义定时器*/
}

unsigned long add_timer(unsigned long nsecs, timer_handler handler, void *data)
{
        /*如果是第 1 次调用 add_timer，进行初始化*/
        /*获得唯一标识符*/
        /*插入队列，并返回标识符*/
```

```
}

unsigned long del_timer(int id)
{
    /*扫描链表，找到自定义定时器*/
    /*删除该定时器*/
    /*返回距离超时时间的秒数*/
}
```

第 9 章　网络通信编程

9.1　实 验 目 的

- 加深对网络编程原理的理解。
- 深入了解客户/服务器网络编程的执行流程。
- 学会使用套接字建立客户/服务器程序。

9.2　背 景 知 识

9.2.1　网间进程通信概念

进程间通信的概念来源于单机系统，为保证单机系统内相互通信的两进程既互不干扰，又协调一致工作，操作系统提供相应的进程间通信设施，如 UNIX BSD 中的管道(pipe)、命名管道(named pipe)和软中断信号(signal)，UNIX System V 中的消息(message)、共享主存(shared memory)和信号量(semaphore)等。在开放系统互连(Open System Interconnection，OSI)参考模型中，网络层及其以下各层又称为通信子网，只提供点到点通信，没有程序或进程的概念。而传输层实现的是“端到端”通信，引进网间进程通信概念，将进程间通信扩展到网络环境。在互连网络中，参与通信的两进程驻留于不同主机上，这些主机可能会位于不同网络上，并安装不同的操作系统。因此，单机系统内的进程间通信设施不再适用。

为支持网间进程通信，网络系统必须解决两个问题。首先是网间进程标识问题。单机系统常通过进程号(process ID)来唯一标识同一主机上的不同进程，但在网络环境下，各主机独立分配的进程号不能唯一标识该进程。例如，主机 A 赋予某进程号 500，在 B 机中可存在进程号同为 500 的进程。因此，无法通过进程号 500 来区分这两个位于不同主机上的进程。其次是多重协议的识别问题。操作系统支持的网络协议众多，不同协议的工作方式不同、协议规程及地址格式也不同；此外，还需要解决差错控制、流量控制、数据排序(报文排序)、连接管理等问题。

为了解决上述问题，业界提出传输控制协议/网际协议(Transmission Control Protocol/Internet Protocol，TCP/IP)，用于支持 Internet 上的网间进程通信，并在此基础上，提供套接字(socket)编程机制，用于支持网间进程通信的实现。

9.2.2 网间进程通信协议

9.2.2.1 TCP/IP 协议族

TCP/IP 是网间进程通信的协议基础，它是由 TCP 协议、IP 协议、用户数据报文协议(UserDatagram Protocol，UDP)、因特网控制消息协议(Internet Control Message Protocol，ICMP)及其他协议组成的一个协议族。该协议族定义计算机通过网络互相通信及协议族各层次之间通信的规范，所有使用或实现某种 Internet 服务的程序都必须使用该协议族。其中 IP 协议、TCP 协议及 UDP 协议是最为根本的 3 种协议，是所有其他协议的基础。从协议分层来看，IP 是网络层协议，TCP 协议及 UDP 协议属于传输层协议。

IP 协议是 TCP/IP 协议族的心脏，也是网络层中最重要的协议。它定义数据按照数据报(Datagram)传输的格式和规则。IP 层接收由更低层(网络接口层，如以太网设备驱动程序)发来的数据报，并把该数据报发送到更高层——TCP 或 UDP 层；与此同时，IP 层也负责把从 TCP 或 UDP 层接收来的数据报传送到更低层。需要注意，IP 层不提供任何报文检测机制，无法确认报文是否按序传输或正确处理，因此 IP 数据报是不可靠的。IP 数据报中含有报文发送者的地址(源地址)及报文接收者的地址(目的地址)。利用 IP 层传输报文时，当目的方网际协议层收到 IP 报文后，必须识别出该报文所使用的上层协议(即传输层协议)，因此，在 IP 报头中，设有一个“协议”域，通过该域的值，即可判明其上层协议类型。

TCP 协议建立在 IP 协议之上，定义网络上程序与程序之间的数据传输格式和规则，提供 IP 数据报的流量控制、传输确认、丢失数据报的重传请求、将收到的数据报按照它们的发送次序重新装配的机制，保证数据传输的可靠性和有序性。TCP 协议是一个可靠的端到端的传输层协议，提供面向连接的数据报服务，该服务模式是电话系统服务模式的抽象，每一次完整的数据传输都要经过建立连接、使用连接、终止连接的过程。在数据传输过程中，各数据分组不携带目的地址，而使用连接号(connection ID)。本质上，连接是一个管道，收发数据不但顺序一致，而且内容相同。

与 TCP 协议相似，UDP 协议也建立在 IP 协议之上。但与 TCP 协议的有连接服务特征不同，UDP 协议是一种无连接协议，提供无连接的数据报服务。该服务模式是邮政系统服务的抽象，无需在数据传输前建立明确的连接。每个 UDP 报文都携带完整的目的地址，各报文可独立地从数据源传送到终点。UDP 协议不保证数据传输的可靠性，也不提供报文顺序重排或请求重传功能，无连接服务不能保证分组的先后顺序，不进行分组出错的恢复与重传。这种不可靠性一方面限制 UDP 协议的应用场合，但另一方面也使得它比 TCP 协议具有更好的传输效率。

图 9-1 描述 Linux 对 IP 协议族的实现机制。Linux 支持 BSD 的套接字和全部的 TCP/IP 协议，是通过网络协议将其视为一组相连的软件层来实现的，BSD 套接字由通用的套接字管理软件支持，该软件是 INET 套接字层，用来管理基于 IP 的 TCP 与 UDP 端口到端口的互联问题。

选择 TCP 协议还是 UDP 协议取决于应用程序的要求。可以从以下几方面考虑。

- 是否需要确认信息？若应用程序需要从客户端或服务器得到确认信息，则使用 TCP 协

议，并在收发数据之前先建立连接，如 FTP 服务。而 UDP 协议更适用于可容忍一定范围数据传输差错率的应用系统，如流服务。

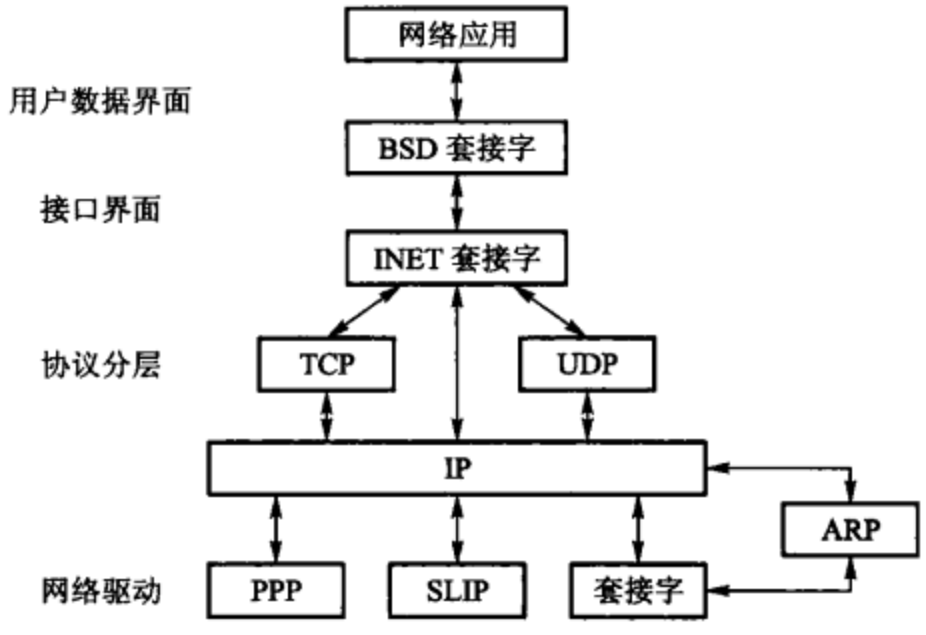


图 9-1 Linux 中网际协议的结构

• 数据传输量规模。由于 UDP 协议不提供端到端的流控制机制，因此基于 UDP 协议的大规模数据传输容易引起网络拥塞，并造成 UDP 流“饿死”现象。而 TCP 协议的面向连接服务可有效避免网络拥塞并确保数据的完整性，但面向连接需要更多的计算资源，因而代价较为“昂贵”。

• 数据发送频次，即数据传输是间歇的，还是在一个会话内完成？TCP 协议在连接双方发送数据之前，都需要首先建立一条连接。该连接的完成需要经过 3 次握手过程，耗时较长。因此，TCP 协议较适合可在一个会话内完成的数据通信，而 UDP 协议更适宜于间歇性数据传输。

9.2.2.2 网络地址

网络环境中相互通信的两个进程分别在不同的计算机上。在互连网络中，两台计算机可能位于通过网络互连设备(网关，网桥，路由器等)连接的异构网络中。因此，网间进程的识别与定位需要经过三级寻址。

• 某一主机可与多个网络相连，必须指定一特定网络地址。
• 网络上每一台主机应有其唯一的地址。
• 每一主机上的每一进程应有在该主机上的唯一标识符。

通常主机地址由网络 ID 和主机 ID 组成，在 TCP/IP 协议中，用 32 位整数值表示；TCP 协议和 UDP 协议均使用 16 位端口号标识用户进程。

9.2.2.3 协议端口

按照 OSI 七层协议的描述，传输层与网络层在功能上的最大区别是传输层提供进程间通信

能力。在进程通信的意义上，网络通信的最终地址不仅仅是主机地址，还包括可以描述进程的某种标识符，为此，TCP/UDP 提出了协议端口(protocol port)的概念，用于标识通信的进程。

端口是一种抽象的软件结构(包括一些数据结构和 I/O 缓冲区)，被客户程序或服务进程用来发送和接收信息。进程通过函数与某端口建立连接后，传输层传给该端口的数据都被相应进程所接收，相应进程发给传输层的数据都通过该端口输出。在 TCP/IP 协议的实现中，端口操作类似于一般的 I/O 操作，进程获取一个端口，相当于获取本地唯一的 I/O 文件，可以用一般的读写原语访问。服务进程通常使用一个固定的端口，类似于文件描述符，每个端口都拥有一个称为端口号(Port Number)的整数型标识符，用于区别不同端口。由于 TCP/IP 传输层的两个协议 TCP 和 UDP 是完全独立的两个软件模块，因此各自的端口号也相互独立，如 TCP 协议有一个 255 号端口，UDP 协议也可以有一个 255 号端口，两者并不冲突。

端口号的分配是一个重要问题。有两种基本分配方式：第 1 种叫全局分配，是一种集中控制方式，由一个公认的中央机构根据用户需要进行统一分配，并将结果公布于众；第 2 种是本地分配，又称动态连接，即进程需要访问传输层服务时，向本地操作系统提出申请，操作系统返回一个本地唯一的端口号，进程再通过合适的函数将自己与该端口号绑定起来。TCP/IP 端口号的分配中综合上述两种方式，TCP/IP 将端口号分为两部分，少量的作为保留端口，以全局方式分配给服务进程。因此，每一个标准服务器都拥有一个全局公认的端口，即使在不同计算机上，其端口号也相同。剩余的为自由端口，以本地方式进行分配。TCP 协议和 UDP 协议均规定，小于 256 的端口号才能作保留端口。

表 9-1 列出常见的 Internet 协议/服务及其默认使用的端口。

表 9-1 常见 Internet 服务的默认端口

服务/协议	端 口 号	说 明
echo	7	回显服务，回显另一主机发送的数据，以验证两主机之间连接的有效性
daytime	13	时钟服务，返回当前服务器时间的文本描述
ftp	20/21	ftp 协议用于文件传输。端口 21 用于命令；端口 20 用于数据
TELNET	23	用于以命令行方式与远程主机交互
WHOIS	43	为网络管理提供的简单目录服务
finger	79	用于获取有关主机用户的信息
HTTP	80	超文本传输协议(HyperText Transfer Protocol)，WWW 的基础协议
SMTP	25	简单邮件传输协议(Simple Mail Transfer Protocol)，用于发送邮件
POP3	110	Post Office Protocol Version 3，用于从邮件服务器传送电子邮件到客户机

综上所述，网络中用一个三元组可以在全局唯一标志一个进程：

(协议，本地地址，本地端口号)

这个三元组称为一个半相关(Half-Association)，它指定连接的一端。

而一个完整的网间进程通信需要由两个进程组成，并且只能使用同一种高层协议。也就是说，不可能通信的一端用 TCP 协议，而另一端用 UDP 协议。因此一个完整的网间通信需要一个五元组来标识：

(协议，本地地址，本地端口号，远地地址，远地端口号)

这个五元组称为一个相关(Association)，即两个协议相同的半相关才能组合成一个合适的相关，或完全指定组成一连接。

9.2.2.4 通信服务模式

在 TCP/IP 网络应用中，通信的两个进程间相互作用的主要模式是客户/服务器(Client/Server)模式，即客户向服务器发出服务请求，服务器接收到请求后，提供相应的服务。客户/服务器模式在操作过程中采取的是主动请求方式，服务器首先启动服务，客户在需要相应服务时向服务器提出服务请求。

服务器的工作流程如下：

① 打开一通信通道并告知本地主机，它愿意在本地主机地址的指定端口上(如 FTP 的服务端口为 21)接收客户请求，提供特定服务。

② 等待客户请求到达该端口。

③ 当接收到重复服务请求时，处理该请求并发送应答信号。当接收到并发服务请求时，将激活一新进程来处理新客户请求(如 Linux 系统中用 fork( )、exec( ))。新进程负责处理此客户请求，并不需要对其他请求作出应答。服务完成后，关闭此新进程与客户的通信连接，并终止。

④ 返回步骤②，等待另一客户请求。

⑤ 关闭服务器。

客户机的工作流程如下：

① 打开一通信通道，并连接到服务器所在主机的特定端口。

② 向服务器发服务请求报文，等待并接收应答。

③ 重复步骤②，直到服务请求结束。

④ 请求结束后关闭通信通道并终止。

从上面所描述过程可知，客户与服务器进程的作用是非对称的，服务器端提供服务实现所需的所有代码，而客户端代码主要负责向服务器发送服务请求，并根据应答结果进行相应处理。服务器进程一般先于客户请求而启动。只要系统运行，该服务进程一直存在，直到正常或强迫终止。

9.2.3 套接字编程

网络通信中的一个非常重要的概念就是套接字，简单地说，套接字就是网络进程的 ID，它是 IP 地址和相应 TCP/UDP 端口号的组合体。在一台计算机中，一个端口一次只能分配给一个

进程，即端口号与进程具有一一对应的关系，因此，端口号和网络地址就能唯一地确定 Internet 中的一个网络进程。

要在应用程序中使用 TCP/IP，就需要使用应用程序接口(API)。至于应用程序如何使用 TCP/IP，还没有业界标准对其进行限定。然而，支持 TCP/IP 的 BSD UNIX 系统中的 API 已成为许多系统的事实标准。在利用网络运行 TCP/IP 的实例中，套接字接口就是网络进程间通信的终点，它能够支持多个信息传输进程。利用调用套接字的方法编写出来的程序，经过少许改写或无需修改，即可用在不同的网络架构上和不同的本地网络进程间通信设备中。图 9-2 所示为套接字在 TCP/IP 网络中的位置。

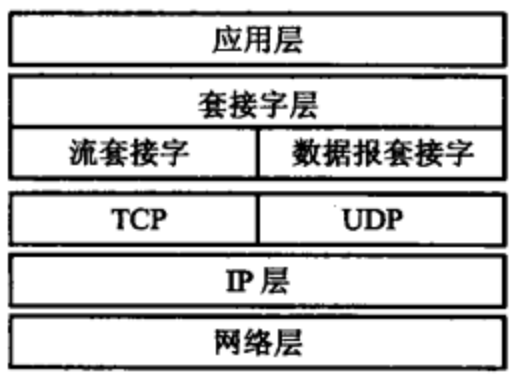


图 9-2 TCP/IP 网络连接套接字

9.2.3.1 套接字类型

套接字接口能够同时向其他进程发送或从其他进程接收数据，依照套接字的类型而调用语义。共有 3 种套接字：流格式、数据报格式和原始格式套接字，每一种类型分别代表一种不同类型的通信服务。

- 流格式套接字(SOCK_STREAM)：提供面向连接的通信，通信过程中所包含的两个进程必须建立相互间的逻辑连接。流格式套接字保证数据无差错、无重复地发送，且按发送顺序接收。该格式内设流量控制，避免数据流超限；数据被看做字节流，无长度限制。流格式套接字对应于 TCP/IP 协议中的 TCP 协议。
- 数据报格式套接字(SOCK_DGRAM)：提供无连接服务，通信进程相互之间并不存在着逻辑连接。数据报以独立包形式被发送，不提供无错保证，数据报可能会丢失或被复制，并且不能确保按照发送顺序被接收者接收。数据报格式套接字对应于 TCP/IP 协议中的 UDP 协议。
- 原始格式套接字(SOCK_RAW)：该套接字支持对底层协议(如 IP、ICMP)直接访问。常用于检验新的协议实现或访问现有服务中配置的新设备。

9.2.3.2 套接字工作原理

套接字接口是 TCP/IP 网络的 API，套接字接口定义许多函数或例程，程序员可以用它们来

开发 TCP/IP 网络上的应用程序，学习 Internet 上的 TCP/IP 网络编程，必须理解套接字接口。

套接字接口设计者最先是将接口放在 UNIX 操作系统里面的。如果了解 UNIX 系统的输入/输出的话，就很容易了解套接字。网络的套接字数据传输是一种特殊的 I/O，套接字也是一种文件描述符。套接字也具有一个类似于打开文件的函数调用 socket ( )，该函数返回一个整型的套接字描述符，随后的连接建立、数据传输等操作都是通过该套接字实现的。表 9-2 列出基本的套接字函数。

表 9-2 基本的套接字函数

socket( )	创建套接字
bind( )	将套接字和网络地址联系在一起
connect( )	将套接字和远程网络地址连在一起
listen( )	侦听传入的连接意图
accept( )	接受传入的连接意图

套接字接口的进程通常用于客户机/服务器编程。客户端进程由用户直接或间接操纵，而服务器端进程驻留在主机上等待连接进入。服务器进程能够自动连续地运行。在 UNIX 的环境中，这种进程被叫做端口监控进程。图 9-3 描述流格式套接字客户机/服务器通信程序的基本流程，图 9-4 描述数据报格式套接字客户机/服务器通信程序的基本流程。在流格式套接字客户机/服务器通信程序中，服务器程序首先创建一个套接字(socket( ))，然后将该套接字与本地地址/端口号绑定(bind( ))，成功之后将在相应的套接字上监听(listen( ))。当 accept( )函数捕捉到一个连接服务(connect( ))请求时，接受并生成一个新的套接字，并通过这个新的套接字与客户端通信(通过 recv( )获取客户机数据，调用 send( )向客户机发送数据)。服务结束后，服务器程序将关闭新建的套接字(close( ))。为请求服务，客户程序也要创建一个套接字，将该套接字与本地地址/端口号绑定。客户程序还需要指定服务器端的地址与端口号，并向服务器发出套接字连接请求(connect( ))。请求被服务器接受后，客户可以通过套接字与服务器通信(调用 recv( )/send( ))，服务结束后，客户程序关闭套接字连接(close( ))。由于数据报格式套接字通信无需建立连接，因此，该格式客户机/服务器通信程序的服务进程中不需要通过调用 listen( )函数准备好接收连接，并通过 accept( )将服务进程转入睡眠状态，等待客户连接请求。

9.2.3.3 套接字函数调用说明

套接字调用所涉及的函数及数据类型定义主要包含在头文件 sys/types.h 及 sys/socket.h 中，以下根据套接字客户机/服务器通信程序的初始化及执行过程，介绍其中涉及的主要函数。

1. 创建套接字——socket( )

程序在使用套接字前，首先必须拥有一个套接字，函数 socket( )向应用程序提供创建套接字的手段，该函数返回一个类似于文件描述符的句柄。socket( )函数原型为：

```
int socket(int domain, int type, int protocol)
```

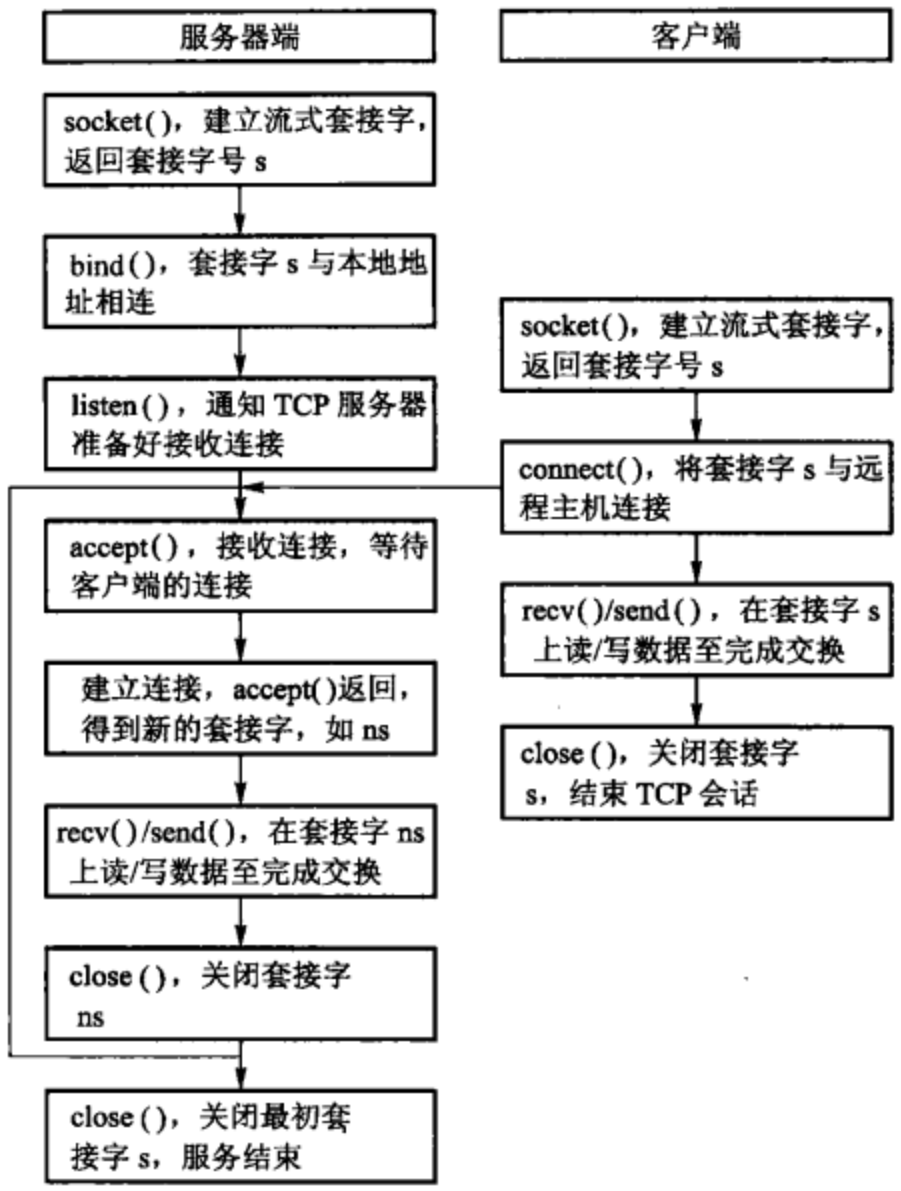


图 9-3 流格式套接字客户机/服务器通信程序基本流程

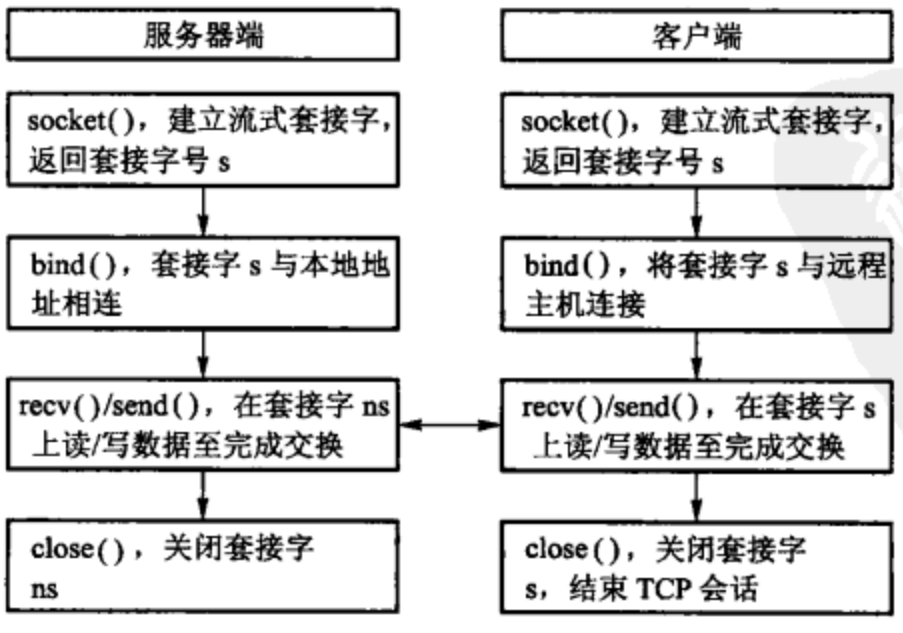


图 9-4 数据报套接字客户机/服务器通信程序基本流程

该调用要接收 3 个参数：domain、type、protocol。参数 domain 指明所使用的协议族，通常为 AF_INET，表示互联网协议族(TCP/IP 协议族)。参数 type 指定套接字的类型(SOCK_STREAM、SOCK_DGRAM 或 SOCK_RAW)。参数 protocol 说明该套接字使用的特定协议，如果调用者不希望特别指定使用的协议，则置为 0，使用默认的连接模式。

套接字描述符是一个指向内部数据结构的指针，它指向描述符表入口，“建立套接字”意味着为一个套接字数据结构分配存储空间。调用 socket( )函数时，套接字执行体将根据这 3 个参数建立一个套接字，并将相应的资源分配给它，同时返回一个整型套接字描述符。因此，socket( )函数实际上指定了相关五元组中的“协议”这一元。

调用成功时 socket( )返回一个正整型数，供在后面的调用中使用；调用失败时返回–1。

2. 配置套接字——bind( )

用 socket( )创建一个套接字后，存在一个命名空间(地址族)，但它没有被命名。因此，在使用套接字进行网络传输以前，必须配置该套接字。bind( )将套接字地址(包括本地主机地址和本地端口地址)与所创建的套接字号联系起来，即将名字赋予套接字，以指定本地半相关。bind( )函数原型为：

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen)
```

其中，参数 sockfd 是调用 socket( )函数返回的套接字描述符，my_addr 是一个指向包含有本机 IP 地址及端口号等信息的 sockaddr 类型的指针；addrlen 常被设置为 sizeof(struct sockaddr)。

struct sockaddr 结构类型用于保存套接字信息：

```
struct sockaddr {
    unsigned short sa_family;    /* 地址族，  AF_xxx */
    char sa_data[14];            /* 14 B 的协议地址 */
};
```

参数 sa_family 一般为 AF_INET，代表 Internet(TCP/IP)地址族。参数 sa_data 则包含该 socket 的 IP 地址和端口号。

另外还有一种结构类型：

```
struct sockaddr_in {
short int sin_family;          /*地址族*/
unsigned short int sin_port;   /*端口号*/
struct in_addr sin_addr;       /*IP 地址*/
unsigned char sin_zero[8];     /*填充 0 以保持与 struct sockaddr 同样大小*/
};
```

该结构更方便使用，其中参数 sin_zero 用来将 sockaddr_in 结构填充到与 struct sockaddr 同样的长度，可以用 bzero( )或 memset( )函数将其置为 0。

指向 sockaddr_in 的指针和指向 sockaddr 的指针可以相互转换，这意味着如果一个函数所需参数类型是 sockaddr，可以在函数调用的时候将一个指向 sockaddr_in 的指针转换为指向 sockaddr 的指针；反之亦然。

需要注意，在调用 bind( )函数时一般不要将端口号置为小于 1024 的值，因为 1～1024 是保留端口号，可选择大于 1024 中的任何一个没有被占用的端口号。也可以用下面的赋值自动获得本机 IP 地址和随机获取一个未被占用的端口号：

```
my_addr.sin_port = 0; /* 系统随机选择一个未被使用的端口号 */
my_addr.sin_addr.s_addr = INADDR_ANY; /* 填入本机 IP 地址 */
```

注意，使用 bind( )函数时需要将 sin_port 和 sin_addr 转换成为网络字节优先顺序，而 sin_addr 则不需要转换。

计算机数据存储有两种字节优先顺序，即高位字节优先和低位字节优先。Internet 上数据以高位字节优先顺序在网络上传输，所以对于在内部是以低位字节优先方式存储数据的计算机，在 Internet 上传输数据时就需要进行转换，否则就会出现数据不一致。常用字节顺序转换函数包括以下几种。

- htonl( )：把 32 位值从主机字节序转换成网络字节序
- htons( )：把 16 位值从主机字节序转换成网络字节序
- ntohl( )：把 32 位值从网络字节序转换成主机字节序
- ntohs( )：把 16 位值从网络字节序转换成主机字节序

调用成功时 bind( )函数返回 0；调用失败时返回–1，并将 errno 置为相应错误号。

3. 监听连接——listen( )

listen( )函数使套接字处于被动监听模式，并为该套接字建立一个输入数据队列，将到达的服务请求保存在此队列中，直到程序处理它们。listen( )函数原型为：

```
int listen(int sockfd, int backlog)
```

参数 sockfd 是调用 socket( )函数返回的套接字描述符，backlog 指定在请求队列中允许的最大请求数。listen( )函数并未开始接收连线，只是设置套接字为 listen 模式，真正接收客户端连接的是 accept( )。如果一个服务请求到来时，输入队列已满，该套接字将拒绝连接请求，客户将收到一个出错信息。

调用成功时，listen( )返回 0；调用失败时返回–1，并将 errno 置为相应的错误号。

4. 建立套接字连接——connect( )/accept( )

connect( )和 accept( )这两个函数用于完成一个完整相关的建立。面向连接的客户程序使用 connect( )函数来配置套接字并与远端服务器建立一个 TCP 连接。connect( )函数原型为：

```
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen)
```

参数 sockfd 是调用 socket( )函数返回的套接字描述符，serv_addr 是指向说明对方套接字地址结构的指针，addrlen 说明对方套接字地址长度。

面向无连接的客户进程也可以调用 connect( )，但此时在进程之间没有实际的报文交换，调用将从本地操作系统直接返回。

调用成功时，connect( )返回 0；调用失败时返回–1，并将 errno 置为相应的错误号。

accept( )使服务器处于睡眠状态，等待来自客户进程的连接请求。accept( )函数原型为：

```
int accept(int sockfd, void *addr, int *addrlen);
```

参数 sockfd 是被监听的套接字描述符；addr 通常是一个指向 sockaddr_in 变量的指针，用来存放提出连接请求的客户机信息(如客户地址或端口)；addrten 通常为一个指向值为 sizeof(struct sockaddr_in)的整型指针变量。

当 accept( )函数监视的套接字收到连接请求时，套接字执行体将建立一个与 sockfd 具有相同特性的新套接字，将该新套接字和请求连接进程的地址联系起来，并在新套接字上进行数据传输操作。收到服务请求的初始套接字仍可以继续监听。

调用成功时，accept( )返回 0；调用失败时返回–1 并将 errno 置为相应的错误号。

5. **数据传输——send( )/recv( )及 sendto( )/recvfrom( )**

send( )函数和 recv( )函数用于在面向连接的套接字上进行数据传输，sendto( )函数和 recvfrom( )函数用于在无连接的数据报套接字方式下进行数据传输。

(1) send( )函数

send( )函数用于将数据由指定的套接字传给对方主机，其函数原型为：

```
int send(int sockfd, const void *buf, int len, int flags);
```

参数 sockfd 是已连接的本地套接字描述符；buf 指向待发送数据缓冲区；len 是以字节为单位的缓冲长度；flags 指定传输控制方式；如是否发送带外数据等，一般情况下置为 0。

send( )函数返回实际上发送出的字节数，在程序中应该将 send( )的返回值与欲发送的字节数进行比较。当 send( )函数返回值与 len 不匹配时，应该对这种情况进行处理。

调用成功时，send( )函数返回实际发送的字节数；调用失败时返回–1。

(2) recv( )函数

recv( )函数用于在指定套接字上接收数据，其函数原型为：

```
int recv(int sockfd,void *buf,int len,unsigned int flags);
```

参数 sockfd 是接收数据的本地套接字描述符；buf 指向存放接收数据的缓冲区；len 是以字节为单位的缓冲长度；flags 指定传输控制方式，一般情况下置为 0。

recv( )函数返回实际接收的数据字节数，在程序中应该将 recv( )函数的返回值与欲发送的字节数进行比较，当 recv( )函数返回值与 len 不匹配时，应该对这种情况进行处理。

调用成功时，recv ( )函数返回实际发送的字节数；调用失败时返回–1。

(3) sendto( )函数

sendto( )函数用于将数据由指定的套接字传给对方主机，其函数原型为：

```
int sendto(int sockfd, const void *buf, int len, unsigned int flags, const struct sockaddr *to, int tolen)
```

该函数比 send( )函数多了两个参数，to 表示目的主机的 IP 地址和端口号信息，而 tolen 常常被赋值为 sizeof (struct sockaddr)。

调用成功时，sendto( )函数返回实际发送的数据字节长度，调用失败时返回–1。

(4) recvfrom( )函数

recvfrom( )函数用于在指定 socket 上接收数据，其函数原型为：

int recvfrom(int sockfd, void *buf,int len,unsigned int flags, struct sockaddr *from, int *fromlen)

参数 from 是一个 struct sockaddr 类型的变量，用于保存源机的 IP 地址及端口号。fromlen 常置为 sizeof (struct sockaddr)。当 recvfrom( )返回时，fromlen 包含实际存入 from 中的数据字节数。

调用成功时，recvfrom( )返回实际发送的数据字节长度，调用失败时返回–1。

从 sendto( )和 recvfrom( )函数原型可知，由于数据报套接字没有与远端计算机建立连接，因此，通过数据报套接字发送数据时需指明目的地址及端口。但若对数据报套接字调用了 connect( )函数时，则也可利用 send( )和 recv( )函数进行数据传输，但该套接字仍然是数据报套接字，并且利用传输层的 UDP 服务。但在发送或接收数据报时，内核会自动为其加上目的地和源地。

6. **结束传输**——close( )

当所有的数据操作结束以后，可以调用 close( )函数来释放该套接字，从而停止在该套接字上的任何数据操作。close( )函数原型为：

int close(int sockfd)

调用成功时，close ( )返回 0；调用失败时返回–1。

9.3 实 验 内 容

9.3.1 实验 1 UDP 通信

9.3.1.1 实验说明

创建一个套接字，然后用它发送消息到以命令行参数传入的特定的主机和端口号。

9.3.1.2 解决方案

UDP 编程的客户端一般步骤是：

① 使用函数 socket( )，创建一个套接字。

② 使用函数 setsockopt( )，设置套接字属性(可选)。

③ 使用函数 bind( )，将 IP 地址、端口等信息绑定到套接字上(可选)。

④ 设置对方的 IP 地址和端口等属性。

⑤ 使用函数 sendto( )发送数据。

⑥ 关闭网络连接。

服务器端启动时，接收用户输入，在用户指定端口打开服务监听程序，然后进入循环，接收和打印从客户端发来的数据报。客户端请求服务时，首先通过广播机制寻找服务器，随后建立通信连接。

9.3.1.3 程序框架

1. 客户程序

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define oops(m,x)    { perror(m);exit(x);}

int main(int ac, char *av[ ])
{

 if ( ac != 4 ){
  fprintf(stderr,"usage: dgsend host port 'message'\n");
  exit(1);
 }
 msg = av[3];
 if (创建 UDP socket 失败)
  oops("cannot make socket",2);
 if ( IP 地址转换失败)
  oops("make addr",4);
 if (发送数据失败)
  oops("sendto failed", 3);
 return 0;
}
```

2. 公用辅助函数

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>

#define     HOSTLEN   256
        int make_internet_address( );
```

```
int make_dgram_server_socket(int portnum)
{
 struct   sockaddr_in    saddr;                    /*构建地址信息*/
 char hostname[HOSTLEN];                           /*主机名   */
 int sock_id;                                      /*套接字号   */

 sock_id = socket(PF_INET, SOCK_DGRAM, 0);  /* 获取套接字*/
 if ( sock_id == -1 ) return -1;

                                                   /* 建立地址，绑定到套接字*/
 gethostname(hostname, HOSTLEN);                   /* 获取主机名*/
 make_internet_address(hostname, portnum, &saddr);

 if ( bind(sock_id, (struct sockaddr *)&saddr, sizeof(saddr)) != 0 )
         return -1;

 return sock_id;
}

int make_dgram_client_socket( )
{
 return socket(PF_INET, SOCK_DGRAM, 0);
}

int make_internet_address(char *hostname, int port, struct sockaddr_in *addrp) {

 /*通过主机名和端口号构建 Internet socket 地址*/
 struct hostent *hp;

 bzero((void *)addrp, sizeof(struct sockaddr_in));
 hp = gethostbyname(hostname);
 if ( hp == NULL ) return -1;
 bcopy((void *)hp->h_addr, (void *)&addrp->sin_addr, hp->h_length);
 addrp->sin_port = htons(port);
 addrp->sin_family = AF_INET;
 return 0;
}

int get_internet_address(char *host, int len, int *portp, struct sockaddr_in *addrp)
{
```

```
    /*从套接字地址中解析主机名和端口号*/
     strncpy(host, inet_ntoa(addrp->sin_addr), len );
     *portp = ntohs(addrp->sin_port);
     return 0;
    }
```

3. 服务器程序

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define oops(m,x)   { perror(m);exit(x);}
/*辅助函数 make_dgram_server( )和 get_internet_address( )*/
int   make_dgram_server_socket(int);
int   get_internet_address(char *, int,   int *, struct sockaddr_in *);
void say_who_called(struct sockaddr_in *);

int main(int ac, char *av[])
{
 int port;
 int sock;
 char buf[BUFSIZ];                  /*接收数据缓存*/
 size_t msglen;                     /*接收到的数据长度*/
 struct sockaddr_in saddr;          /*存放发送者地址*/
 socklen_t saddrlen;

 if ( ac == 1 || (port = atoi(av[1])) <= 0 ){
  fprintf(stderr,"usage: dgrecv portnumber\n");
  exit(1);
 }

 if( 建立 UDP 套接字失败)
  oops("cannot make socket",2);
saddrlen = sizeof(saddr);
 while(等待接收数据) {
  buf[msglen] = '\0';
  printf("dgrecv: got a message: %s\n", buf);
  say_who_called(&saddr);
 }
```

```
 return 0;
}
void say_who_called(struct sockaddr_in *addrp) {
 char host[BUFSIZ];
 int port;

 get_internet_address(host,BUFSIZ,&port,addrp);
 printf("    from: %s:%d\n", host, port);
}
```

9.3.2 实验 2 基于 TCP 的客户/服务器程序

9.3.2.1 实验说明

网络客户端程序，连接服务器，发送一行字符串数据。网络服务端程序，等待客户端的连接，接收一行字符串数据并打印到终端，然后关闭连接，等待下一个客户端的连接。

9.3.2.2 解决方案

与 UDP 模式类似。

9.3.2.3 程序框架

1. 客户端程序

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <errno.h>

int readline(int fd, void *pbuf, int maxlen);

int main(int argc,char **argv)
{
 int fd;
 int len,ret;
 struct sockaddr_in remoteaddr;
 char data[1024];
/*建立套接口*/
/*设置远程地址参数*/
```

```
if(连接到远程地址失败)  {
   printf("connect( ) error\n");
   return –1;
  }

  /*发送数据到套接字*/
if(发送失败) {
   printf("send( ) error\n");
   goto finish;
  }
  printf("sent line:%s", data);
  printf("client exit.\n");
  /*关闭套接字*/
return 0;
}

int readline(int fd, void *pbuf, int maxlen) {
  int n, ret;
  char c, *ptr;
  ptr = pbuf;
  for(n = 1; n < maxlen; n++) {
again:
   ret = recv(fd, &c, 1, 0);
   if( ret == 1) {
    *ptr++ = c;
    if(c == '\n') break;          /*读结束*/
   } else
   if(ret == 0) {
       if(n == 1)
          return 0;               /*文件尾，未读到数据*/
      else
      break;                      /*文件尾，读到部分数据*/
   }    else {
    if(errno == EINTR) goto again;
    return –1;                    /*出错*/
   }
  }
 *ptr = 0;
```

```
    return n;
}
```

2. 服务器程序

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <errno.h>

int readline(int fd,void *pbuf,int maxlen);
int main(int argc,char **argv)
{
        /*建立套接字*/
        if(创建失败) {
                printf("socket( ) error %d\n",errno);
                return -1;
        }
        /*绑定地址和端口*/
        if(绑定失败) {
                printf("bind( ) error\n");
                return -1;
        }

        /*建立套接字队列*/
        if(建立失败) {
                printf("listen( ) error\n");
                return -1;
        }
        while(1) {
        /*等待客户请求到达*/
        /*接收数据*/
                printf("server read line :%s",buf);
        /*关闭连接*/
        }
        return 0;
}

int readline(int fd,void *pbuf,int maxlen)
```

```
{
    int n,ret;
    char c,*ptr;
    ptr = pbuf;
    for(n = 1; n< maxlen; n++) {
again:
        ret = recv(fd,&c,1,0);
        if(ret ==1) {
            *ptr++ =c;
            if(c == '\n')break;        /*结束*/
        } else
          if(ret == 0){
            if(n == 1)
                return 0;              /*文件尾，未读到数据*/
            else
                break;                 /*文件尾，读到数据*/
        } else {
            if(errno == EINTR)goto again;
            return -1;                 /*出错*/
        }
    }
    *ptr = 0;
    return n;
}
```

第 10 章　事件驱动编程

10.1　实 验 目 的

- 了解 curses 库和 curses 库函数。
- 学习屏幕管理、使用定时器和信号实现进程的并发执行。
- 学会异步事件驱动编程。
- 利用上述知识编写一个视频动画游戏。

10.2　背 景 知 识

10.2.1　视频游戏的概念

设计一个视频游戏需要综合应用一些概念和原则。游戏程序必须在计算机屏幕的特定位置画出影像，这涉及空间位置处理；游戏程序要在特定的时间触发某些事件，让影像以不同的速度在屏幕上移动，以特定的时间间隔改变位置，这需要获知时间；影像在屏幕上平滑地移动和变化，用户在任意时刻输入信息希望游戏作出反应，这意味着游戏程序应该在保持几个影像移动的同时还要及时响应中断，也就是说，它必须并发执行多个事件。

对于操作系统来说，运行视频游戏程序同样要面临这些问题。内核将游戏程序加载到主存空间并维护不同功能的程序段在主存中的位置；在内核的调度下，程序段以时间片轮流方式运行，同时，内核也在特定的时刻运行特定的内核任务；内核必须在很短的时间内响应用户通过外部设备在任何时刻的输入。由于“同时”要并发执行多任务，内核必须确保这些任务的安全性、协调性和及时性。本章将利用 curses 库完成一个基于字符界面的动画游戏。

10.2.2　curses 库的历史

curses 是一个函数库，包含许多库函数，专门用来进行 UNIX 终端环境下的屏幕界面处理及 I/O 处理。通过这些库函数，C 和 C++程序就可以控制终端的视频显示以及输入/输出。使用 curses 库中的函数，用户可以非常方便地创建和操作窗口，使用菜单及表单，而且最为重要的一点是使用 curses 库编写的程序将独立于各种具体终端，这样一个直接的好处就是程序具有良好的可移植性。这一点在网络上显得尤其重要，因为面对的可能是上百种终端，如果为每一个

终端都专门编写一套新程序，复杂程度超乎想像，而且几乎不可能实现。为此，curses 库使用终端描述数据库 terminfo(TERMinal INFOrmation database)或 termcap(TERMinal CAPability database)，这两个数据库里存放不同终端的操作控制码、转义序列及其余相关信息。这样，使用每一个终端时，curses 库将首先在终端描述数据库中查找是否存在该类型的终端描述信息，如果找到则进行适当的处理；如果数据库中没有这种终端信息，则程序无法在该终端上运行，除非用户自己增加新的终端描述。curses 库还可以管理键盘，提供一种简单易用的非阻塞字符输入模式。curses 库可以看成是介于简单的文本行程序和完全图形化界面的 X 视窗系统程序之间的一个过渡。

curses 库的名称起源于“cursor optimization”，即光标优化的意思，它能够优化光标的移动并减少需要对屏幕进行的刷新，最早是由美国加州大学伯克利分校的 Bill Joy 和 Ken Arnold 研究和开发的，主要用于处理游戏 rogue 的屏幕界面。rogue 是一个古老的基于文本的冒险类游戏，在当时，仅仅控制游戏屏幕的外观显示就需要编写大量代码，因为它们使用的是早期的 termios 甚至是 tty 接口。巨大的工作量迫使 Bill Joy 和 Ken Arnold 将 rogue 游戏中的所有屏幕处理和光标移动的函数汇集到函数库中，这就形成早期简单的 curses 函数库的雏形。它最终随着 BSD UNIX 的早期版本发行开来，在这个版本中使用的是当时业已存在的 termcap 数据库来描述终端信息。后来美国 AT&T 公司贝尔实验室的 Mark Horton 在 System Ⅲ UNIX 中重新编写了 curses 库，它相对以前的版本有很大扩展和提高，增加了许多新的特性。首先将 termcap 数据库改进为 terminfo 数据库，terminfo 数据库完全由 Horton 开发编写，它是从 termcap 发展而来，而且更为重要的是其中引进参数化性能的概念，这样使得描述多视频属性以及彩色终端成为可能。在后来的 AT&T System V UNIX 版本中，curses 库扩展出更多功能和性能，包括对窗体、菜单、面板、表单等组件以及对鼠标的支持。至此 curses 库内容以及设计与最初 BSD 版本的 curses 库在功能和复杂性上已经相去甚远。

Linux 使用的 curses 库版是 ncurses(new curses)。对 curses 库的程序进行编译时，必须在程序中包含头文件 curses.h，同时需要在编译命令行中用-lcurses 选项对 curses 库进行链接。在许多 Linux 系统中，curses 头文件和库文件是对 ncurses 对应文件的链接。

10.2.3 使用 curses 库

10.2.3.1 curses 屏幕

curses 程序工作在屏幕、窗口和子窗口上。“屏幕”指的是正在写的设备(通常是终端屏幕、也可能是 xterm 屏幕)。curses 库将终端屏幕看成是由字符单元组成的网格，每一个单元由(行、列)坐标对标示，坐标系的原点是屏幕的左上角，行坐标自上而下递增，列坐标自左向右递增。图 10-1 所示为 curses 屏幕。无论何时，至少存在一个 curses 窗口，称为 stdscr，它与物理屏幕

的尺寸完全一样。除了 stdscr 之外，还可以拥有多个子窗口。这些子窗口可以互相重叠，并且还可以拥有各自的子窗口，各自的子窗口总是被包含在其父窗口内。

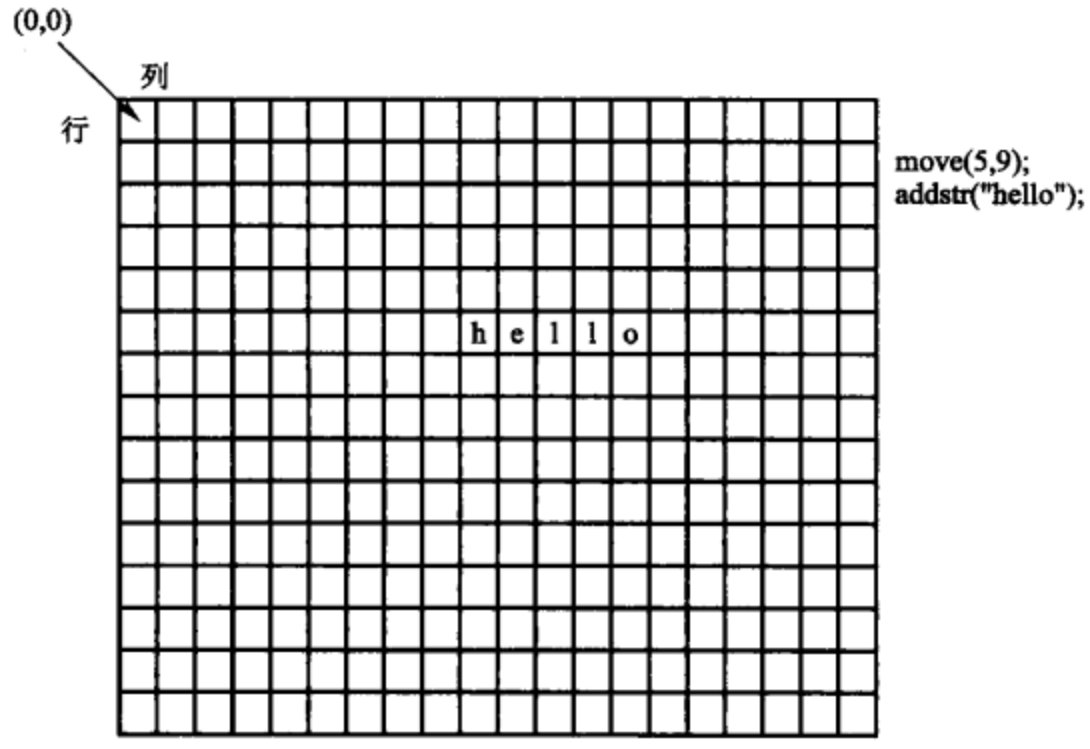


图 10-1 curses 屏幕

curses 库用两个数据结构来映射终端屏幕，分别是 stdscr 和 curscr。stdscr 结构表示是一个逻辑屏幕，它的显示内容在终端屏幕上不一定能够立即看得到，curscr 对应的是实际看到的物理屏幕。在使用 curses 库函数时，函数更新的是 stdscr 结构，即逻辑屏幕。逻辑屏幕是终端应该呈现的内容，它和当前终端的实际内容是有区别的。在调用 refresh( )函数之前，输出到 stdscr 上的内容不会显示在屏幕上。调用 refresh( )函数时，curses 库会比较 stdscr 和 curscr 之间的不同之处，然后通过两个数据结构之间的差异来刷新屏幕。一旦更新，curscr 将显示 stdscr 上的内容，这样当前屏幕上反映内容就与标准屏幕上的一样。图 10-2 和图 10-3 演示在屏幕上输出单词 Eye 和 Bulls 时 stdscr 和 curscr 的不同之处。综上所述，在使用 curses 库时，输出字符的过程如下：

① 用 curses( )函数刷新逻辑屏幕。

② 用 refresh( )函数刷新物理屏幕。

在 10.2.2 小节中曾经提到，curses 库产生是为了能够优化光标的移动并减少需要对屏幕进行的刷新。通过比较 stdscr 和 curscr，curses 库能够判断出两者之间的差异，并通过差异来刷新屏幕。通过这样的刷新步骤，curses 屏幕的刷新效率很高，虽然这点效率对于控制台屏幕来说意义不大，但是如果是通过慢速串行接口或者调制解调器连接到主机上来运行程序，则屏幕刷新效率的提高意义就很大了。

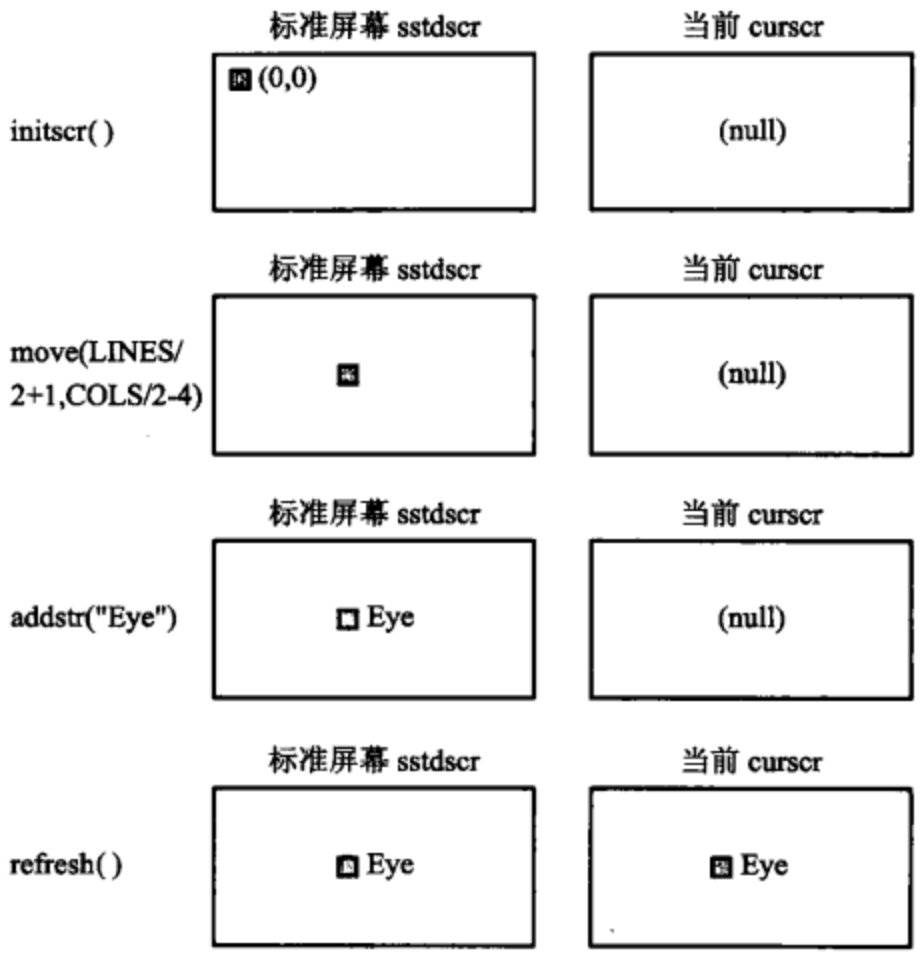


图 10-2 stdscr 和 curscr 关系(1)

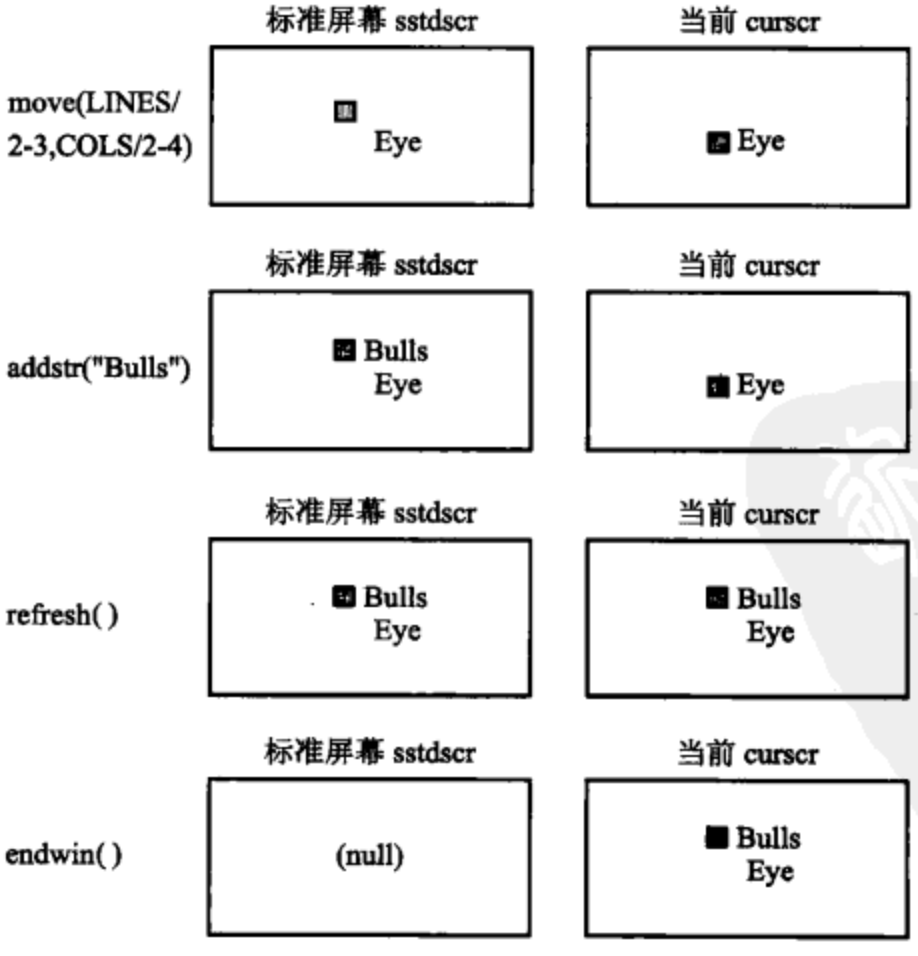


图 10-3 stdscr 和 curscr 关系(2)

一个 curses 程序会多次调用逻辑屏幕输出函数，例如在屏幕上移动光标到达正确的位置，然后输出文本、绘制线框。curses 库函数使用的坐标是 y 值(行号)在前、x 值(列号)在后。每个位置不仅包含该屏幕位置处的字符，还包含其属性，可实现的属性依赖于物理终端的性能指标，但一般至少会支持粗体和下划线这两个属性。在 Linux 控制台上，通常还支持反白显示和均匀彩色属性。

10.2.3.2 初始化和重置函数

由于 curses 库在使用时需要创建和删除一些临时数据结构，所以 curses 程序必须在开始使用时进行初始化，并在结束时删除这些资源。这两项工作通过 initscr( )和 endwin( )函数分别完成。在使用这些函数时，需要包含头文件 curses.h。大体框架如下：

```
#include <curses.h>
int main(int argc, char** argv)
{
        …
        initscr( );
        …
        endwin( );
        …
}
```

这两个函数的原型为：

```
#include <curses.h>
WINDOW *initscr(void);
int endwin(void);
```

initscr( )函数在一个程序中只能调用一次。如果成功，initscr( )函数返回一个指向 stdscr 结构的指针；如果失败，简单输出一条诊断错误信息并使程序结束。endwin( )函数在成功时返回 OK，失败时返回 ERR。endwin( )函数还有一个用法是能够强制 curses 库重新重写屏幕。调用 refresh( )函数时，curses 库只是比较 stdscr 和 curscr 的差异重写屏幕。如果要将 stdscr 的内容完整重写到 curscr 上，可以先调用 endwin( )函数退出 curses 库，然后通过调用 clearok(stdscr,1)和 refresh( )函数继续 curses 操作。这样实际上是首先让 curses 库忘记物理屏幕的内容，然后强迫它执行一次完整的屏幕刷新。

10.2.3.3 屏幕输出和字符属性

curses 库提供一组函数用于将字符输出到屏幕上。这些函数主要有：

```
#include <curses.h>
int addch( const chtype chart_to_add);
int addchstr(chtype *const string_to_add);
int printw(char *format, …);
```

```
int insch(chtype char_to_insert);
int inchstr (chtype *);
int insertln(void);
int delch(void);
int deleteln(void);
int beep(void);
int flash(void);
int refresh( );
```

curses 库使用的字符类型为 chtype。在 Linux 中的定义为：

```
typedef unsigned long chtype;
```

在这个定义下，chtype(32 位)能够表示的字符数量比 char(8 位)更多。addch( )、addchstr( )和 printw( )函数都从当前光标开始输出字符或字符串。addch( )用于输出单个字符，addchstr( )可以输出一个字符串。如果需要对输出的字符串进行格式化，达到类似 printf( )的效果，则需要使用 printw( )函数。该函数的使用方法和 printf( )函数一致。在使用这 3 个函数时，如果当前光标已经有字符，则字符会被新字符覆盖。如果不希望覆盖光标所在位置的字符，则可以使用 insch( )函数，该函数在光标所在位置插入一个字符，原先的字符则向右移。inchstr( )用于插入一个字符串、insertln( )函数可以插入一个空行，并将现有行下移。与 insch( )函数对应的一个函数是 delch( )，该函数能够删除光标所在位置的字符，并将光标右边的字符左移。insertln( )也有对应的函数 deleteln( )，该函数删除一行，并将现有行上移。

程序还可以利用 curses 库对用户进行提示。有两种提示方法：发声和闪烁屏幕。这两种提示方法对应的函数为 beep( )和 flash( )。有些终端并不支持发声，在这种情况下，beep( )函数会尝试让屏幕闪烁。如果使用 flash( )函数，但是终端无法产生效果，则 flash( )函数会尝试在终端上发出声音。

在往屏幕输出完毕后，程序必须通过 refresh( )函数让 curses 库根据 strscr 和 curscr 的差异刷新屏幕。

在输出时，每个 curses 字符都可以有特定的属性，该属性控制着字符在屏幕上的显示方式，前提是用于显示的硬件设备能够支持要求的属性。预定义的属性有 A_BLINK、A_BOLD、A_DIM、A_REVERSE、A_STANDOUT 和 A_UNDERLINE。curses 库提供了一组函数用于设置字符属性：

```
#include <curses.h>
int attron(chtype attribute);
int attroff(chtype attribute);
int attrset(chtype attribute);
int standout(void);
int standend(void);
```

通过这些函数能够设置 curses 的字符属性。attron( )函数用于开启指定属性；attroff( )函数用于关闭一个属性；attrset( )函数用于设置指定属性；而 standout( )函数开启“突出”模式，在

大多数终端上，它通常映射为反白显示；standend( )函数用于关闭“突出”模式。

当设置好 curses 字符属性后，接下来输出的字符都将以指定的属性输出。例如：

```
attron(A_BOLD);          /*打开粗体*/
printw("%s", "BOLD");    /*BOLD 将以粗体显示*/
attroff(A_BOLD);         /*关闭粗体*/
printw("%s", " hello:");  /*将不以粗体显示*/
```

10.2.3.4 移动光标、清除屏幕

curses 库在往屏幕输出字符时，是在当前光标位置进行操作。因此为了在指定位置输出字符，需定义 curses 光标移动函数：

```
#include <curses.h>
int move(int new_y, int new_x);
```

通过该函数可以将光标移动到指定的 new_y 和 new_x 位置。需要注意，在调用 move( )函数后，光标并不会立即出现在指定的位置，这是因为 curscr 尚未更新，在执行 refresh( )函数才能让光标出现在新的位置。new_y 和 new_x 的取值范围也受到限制，new_y 不能超过屏幕的行数，new_x 不能超过屏幕的列数，屏幕的这两个属性通过两个外部整数 LINES 和 COLS 定义。

通常在使用 move( )函数之后是调用屏幕输出的相关函数进行屏幕绘制。curses 库为这样的典型操作也提供了一系列简化函数，这些简化函数是在原来屏幕输出函数之前添加 mv 前缀，代表在输出前移动到指定坐标：

```
#include<curses.h>
int mvaddch( const chtype chart_to_add);
int mvaddchstr(chtype *const string_to_add);
int mvprintw(char *format, …);
…
```

curses 库还提供清除屏幕的方法：

```
#include <curses.h>
int erase(void);
int clear(void);
```

erase( )函数在每个屏幕位置写上空白字符。clear( )函数的功能类似 erase( )函数，但是 clear( )函数会调用 clearok( )函数来强制重现屏幕原文。clearok( )函数会强制执行清屏操作，并在下次调用 refresh( )函数时重现屏幕原文。clear( )函数并不是简单地尝试删除当前屏幕上每个非空白的位置，因此 clear( )函数是一种可以彻底清除屏幕的可靠方法。clear( )函数后跟 refresh( )函数这样的结合提供了一种有效的重新绘制屏幕的方法。

下面用一个例子(cursestest.c)说明如何使用以上介绍过的函数，图 10-4 显示这个程序的输出，在编译该程序时，需要通过-l 指明程序需使用 curses 库：

```
gcc cursestest.c –o test –lcurses
/***** cursestest.c*****/
```

```
#include <stdio.h>
#include <curses.h>

int main(int argc, char* argv[])
{
   initscr( );
   /*粗体显示*/
   move(10, 20);
   attron(A_BOLD);
   addstr("Hello, ");
   refresh( );
   sleep(1);
   /*粗体+下划线*/
   attron(A_UNDERLINE);
   addstr("world!");
   refresh( );
   sleep(1);
   /*下划线*/
   move(11,20);
   attroff(A_BOLD);
   addstr("Hello, ");
   refresh( );
   sleep(1);
   /*普通模式*/
   attroff(A_UNDERLINE);
   addstr("world!");
   refresh( );
   sleep(1);
   /*反白*/
   move(12,20);
   standout( );
   addstr("Hello, world!");
   standend( );
   refresh( );
   sleep(1);

   sleep(10);
   endwin( );
   return 0;
}
```

输出结果如图 10-4 所示。

图 10-4 curses 输出

10.2.3.5 键盘

程序可以通过 curses 库获得用户的输入，当程序开始使用 curses 库时，默认的输入模式是“行模式”。在行模式下，只有用户输入回车之后，用户输入的数据才会被一次性传递给 curses 程序。curses 库也支持“中断模式”输入，在这种模式下，用户的每一个输入都会被直接发送给程序。在使用“中断模式”时，一些简单的特殊字符，如退格键会被直接传递给程序处理，如果想让退格键保留原来的功能，就必须要在程序中实现。打开和关闭“中断模式”的函数为：

```
#include <curses.h>
int cbreak(void);
int nocbreak(void);
```

cbreak( )函数用于打开“中断模式”。nocbreak( )函数用于关闭“中断模式”，并进入“行模式”。curses 库还允许程序关闭字符的回显功能。当回显功能关闭后，用户的输入不会在屏幕上有输出。打开和关闭回显功能的函数为：

```
#include <curses.h>
int echo(void);
int noecho(void);
```

通过 curses 库中读取用户的输入和平常类似，主要函数有：

```
#include <curses.h>
int getch(void);
int getstr(char *string);
int getnstr(char *string, int number_of_characters);
```

```
int scanw(char *format, ...);
```

这些函数与 getchar( )、gets( )和 scanf( )函数非常相似。需要注意，推荐使用 getnstr( )函数，而不使用 getstr( )函数。getstr( )函数对用户输入的字符串长度没有限制。

下面通过一个程序(passwordtest.c)演示本节介绍的函数。该程序让用户输入用户名、密码。在用户输入密码时，程序将关闭字符回显进行保密。

```
/*****passwordtest.c*****/
#include <unistd.h>
#include <stdlib.h>
#include <curses.h>
#include <string.h>

int main(int argc, char *arg[])
{
    char name[20];
    char password[20];
    char *real_password = "123456";
    initscr( );
    move(5, 10);
    addstr("User name:");
getnstr(name, sizeof(name)); /*获得输入*/

move(7, 10);
addstr("Password:");
refresh( );

cbreak( );              /*中断模式*/
noecho( );              /*关闭回显*/
memset(password, 0, sizeof(password));

int len = sizeof(password);
for (int i=0; i<len ; i++){
    password[i] = getch( );
    move(7, 20+i);
    addch('*');
    refresh( );
    if ( password[i] == '\n' )
        break;
    if ( strcmp(password, real_password)==0 )
        break;
}
```

```
    echo( );/*打开回显*/
    nocbreak( );

    move(9, 10 );
    if ( strcmp(password, real_password)==0 )
        addstr("Correct");
    else
        addstr("Wrong");
    refresh( );
    endwin( );

    return 0;
}
```

该程序将输入方式设置成“中断模式”后，用一块主存接收用户的输入。每个输入的密码字符立即被记录并在屏幕上显示“*”，每次输出“*”后都必须通过 refresh( )函数写到屏幕上。再通过 strcmp( )函数比较密码是否正确。

curses 库提供一些处理键盘的功能，一般键盘都会包含方向键和功能键，curses 库提供一个精巧的用于管理功能键的结构。对于每个终端来说，它的每个功能键所对应的转义序列都被保存，通常是保存在一个 terminfo 结构中，而头文件 curses.h 通过一组以 KEY_为前缀的定义来管理逻辑键。该功能需要通过 keypad( )函数来启用。

```
#include <curses.h>
int keypad(WINDOW *window_ptr, bool keypad_on);
```

keypad_on 参数设置为 true，然后调用 keypad( )函数来启用 keypad 模式。在该模式中，curses 库将接管按键转移序列的处理工作，读键盘操作不仅能够返回用户按下的键，还将返回与逻辑按键对应的 KEY_定义。

下面的演示程序(keypadtest.c)将展示如何使用 keypad( )函数。程序一开始时，屏幕上将显示一个字母，用户可以通过方向键移动字母；用户也可以输入不同字母显示在屏幕上。如果用户输入“q”，程序将退出。

```
/*****keypad test.c*****/
#include <curses.h>
#define MIN(a,b) a<b?a:b
#define MAX(a,b) a>b?a:b

int main(int argc, char *argv[])
{
  int x = 10;          /*待显示的字母坐标*/
  int y = 10;
```

```
char ch = 'A';              /*待显示的字母*/

initscr( );
crmode( );                  /*中断模式*/
noecho( );                  /*关闭回显*/
clear( );
keypad(stdscr, TRUE);       /*打开 keypad*/
mvaddch(y, x, ch);
chtype input;
while ( (input=getch( )) && input!=ERR && input!='q' ) {
  if ( (input>='A' && input<='Z') ||
    (input>='a' && input<='z') )
    ch = input;
  else{
     /*通过方向键调整坐标*/
    switch(input) {
    case KEY_LEFT:
        x = MAX(x–1, 0);
        break;
    case KEY_RIGHT:
        x = MIN(x+1, COLS);
        break;
      case KEY_UP:
        y = MAX(y–1, 0);
        break;
      case KEY_DOWN:
        y = MIN(y+1, LINES);
        break;
      }
    }
    clear( );
    mvaddch(y, x, ch);
    refresh( );
  }
  endwin( );
  return 0;
}
```

程序通过 x 和 y 两个变量记录待显示的字母坐标，ch 变量记录待显示的字母。程序不断读入用户的输入，根据方向键对 x 和 y 变量进行相应的变化。如果用户输入的是非“q”字母，则更新待显示的字母变量。在新坐标显示字母前，需要通过 clear( )函数将老字母擦去。

10.3 实 验 内 容

10.3.1 实验 1 利用 curses 库实现弹球游戏

10.3.1.1 实验说明

弹球游戏的界面如图 10-5 所示，它由墙、球和挡板组成。游戏的主要规则为：

- 球以一定的速度移动。
- 球碰到墙壁或挡板会被弹回。
- 用户通过方向键来控制挡板左右移动。

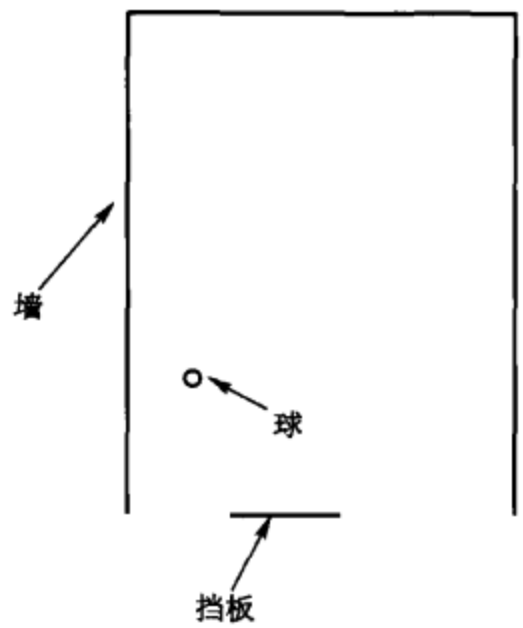


图 10-5 弹球游戏说明

10.3.1.2 解决方案

1. 用 usleep( )函数绘制动画

在弹球游戏中，球在没有碰到挡板或者墙时，会朝着一条直线一直运动，当碰到挡板或者墙的时候，球运动的方向会改变，这涉及如何使用 curses 库实现动画效果。

实现动画的过程是在一个地方画一个字符串，等待几毫秒，然后擦去旧的影像并在原来位置的边上重新绘制一个相同的字符串，通过这样的一个过程就能够形成动画效果。可以通过 usleep( )函数让进程进入等待，当进程被唤醒时，绘制下一个图像，动画效果便能形成。下面以一个简单的例子来说明如何让球沿着一条直线运动。

在本例中，球(用字母 O 表示)会先水平向右移动，当移动到屏幕的最右端，球会水平向左移动，当球碰到屏幕的左端，球会改变方向向右移动。

```
/*****exampleprogram1.c*****/
#include <curses.h>

int main(int argc, char *argv[])
{
   int x = 10;
   int y = 10;
   int direction = 1;
   char ball = 'O';
   /*初始化*/
   initscr( );
   crmode( );
```

```
noecho( );

while ( true ) {
   clear( );
   mvaddch(y, x, ball);
   refresh( );
   /*更新坐标*/
   x += direction;
   /*变换方向*/
   if ( x==COLS ) {
      direction = –1;
      x = COLS–1;
      beep( );          /*碰到墙时，发出声音*/
   }
   if ( x<0 ){
      direction = 1;
      x = 0;
      beep( );
   }
   usleep(100000);   /*睡眠*/
}
   endwin( );
   return0;
}
```

2. 通过定时器绘制动画

在 exampleprogram1.c 程序执行过程中不断地睡眠，在每次被唤醒后刷新屏幕，形成动画效果，还可以通过定时器不断发送 SIGALRM 信号对屏幕定时刷新。通过定时器刷新屏幕的好处在于程序可以方便地与用户交互，定时器的使用可以参见第 8 章。下面是一个通过定时器实现动画的例子，在该例中，用户可以通过 q 键退出程序，f 键加快球的运动速度，s 键降低球的运动速度。

```
/*****exampleprogram2.c*****/
#include <curses.h>
#include <time.h>
#include <sys/time.h>
#include <signal.h>
/*球的坐标*/
int x = 10;
int y = 10;
int direction = 1;
char ball = 'O';
/*定时器设置函数*/
```

```
int set_ticker(long n_msecs )
{
    struct itimerval new_timeset;
    longn_sec, n_usecs;
    n_sec = n_msecs / 1000 ;
    n_usecs = ( n_msecs % 1000 ) * 1000L ;
    new_timeset.it_interval.tv_sec = n_sec;
    new_timeset.it_interval.tv_usec = n_usecs;
    new_timeset.it_value.tv_sec = n_sec;
    new_timeset.it_value.tv_usec = n_usecs ;
    return setitimer(ITIMER_REAL, &new_timeset, NULL);
}
/*信号处理函数，在 SIGALRM 信号产生时，绘制图像*/
void paint( )
{
    clear( );
    mvaddch(y, x, ball);
    refresh( );

    x += direction;
    if ( x==COLS ) {
       direction = –1;
       x = COLS–1;
       beep( );
    }
    if ( x<0 ){
       direction = 1;
       x = 0;
       beep( );
    }
}

int main(int argc, char *argv[])
{
   chtype input;
   long delay = 100;
   /*初始化 curses*/
   …
   /*设置定时器*/
   signal(SIGALRM, paint);
   set_ticker( delay );
   while ( (input=getch( )) && input!=ERR && input!='q' ) {
```

```
        switch( input ){
        case 'f':{
        /*加速*/
            delay /= 2;
            set_ticker( delay );
            break;
        }
        case 's':{
        /*减速*/
            delay *= 2;
            set_ticker( delay );
            break;
        }
        }
    }
    endwin( );
    return 0;
}
```

3. 绘制球与挡板

在弹球游戏中，球的运动是自动的，球根据碰到的墙或挡板自动进行方向变换。而挡板的移动是由游戏者控制的，游戏者通过方向键控制挡板的左右移动。

```
/*---program3.c---*/
#include <curses.h>
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#include <signal.h>

#define MAX(a,b) a>b?a:b;
#define MIN(a,b) a<b?a:b;

int ballx = 10;
int bally = 10;
int direction = 1;
char ball = 'O';
int barx = 10;
int bary ;
char* bar="__________";
int barlength = 10;
```

```
int set_ticker(long n_msecs )
{

…
}

void paint( )
{
      clear( );
      mvaddch(bally, ballx, ball);
      mvaddstr(bary, barx, bar);
      refresh( );
      …
}

int main(int argc, char *argv[])
{
   chtype input;
   long delay = 100;
initscr( );
bary = LINES–1;
…
while ( (input=getch( )) && input!=ERR && input!='q' ) {
   switch( input ){
   case 'f':{
         /*加速*/
   }
   case 's':{
         /*减速*/
   }
   case KEY_RIGHT:{
      barx = MIN(barx+1, COLS–1–barlength);
      break;
   }
   case KEY_LEFT:{
      barx = MAX( barx–1, 0);
      break;
   }
```

```
            }
        }
        ...
    }
```

程序运行结果如图 10-6 所示。

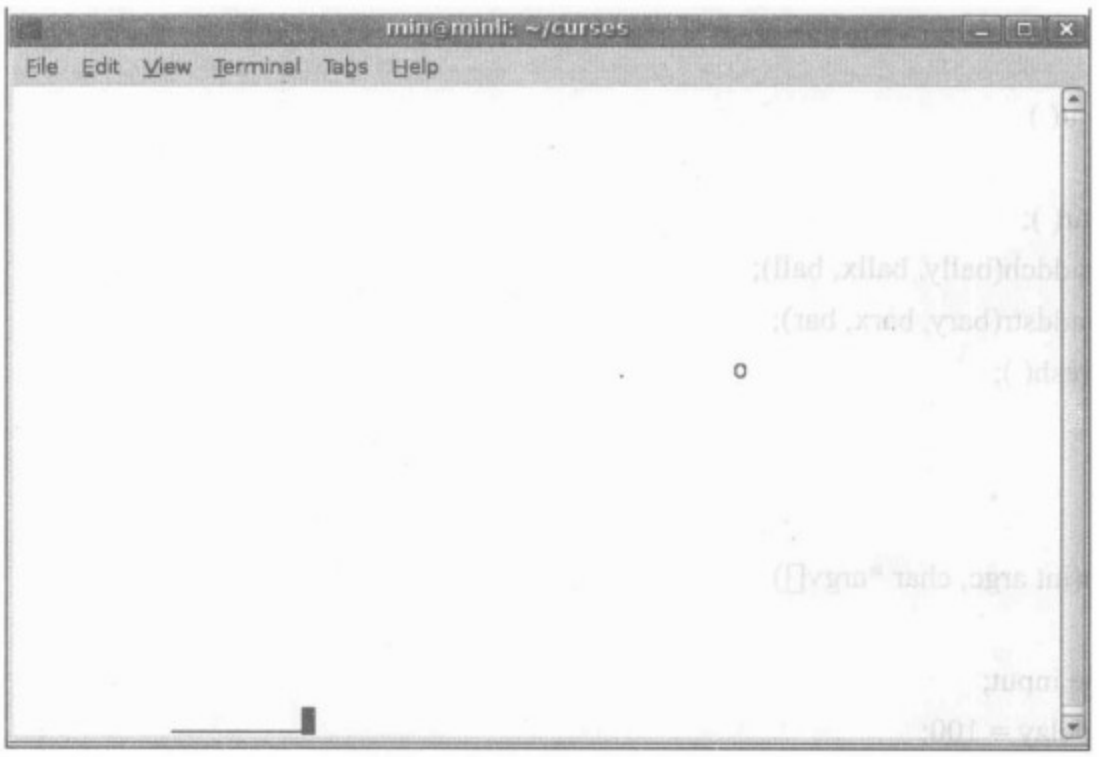


图 10-6 球与挡板

4. 斜线运动

弹球游戏中，球是以斜线方式运动的。在绘制斜线运动的动画时，需要考虑运动的平滑性。例如图 10-7 中，从 A 往 B 方向移动。如果 A 先向上移动一个单位，则必须同时向右移动 3 个单位。这样的移动跨度较大，移动不平滑，动画显示时有跳跃感。如果要平滑移动，比较好的方式是按照图 10-8 的移动方式，每次移动一个单位。当球遇到挡板或墙时，球运动的斜率也要进行相应的变化，该段代码框架如下所示并可由用户自行实现。游戏界面可以参考图 10-9 和图 10-10 所示。

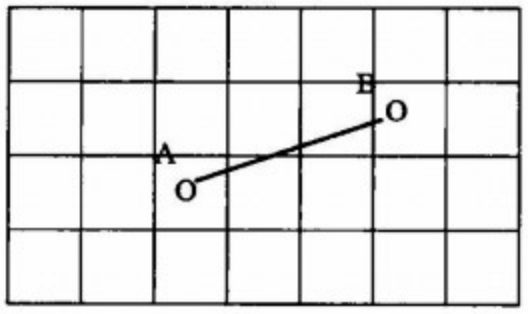


图 10-7 绘制斜线

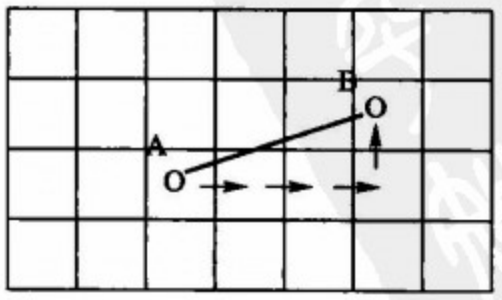


图 10-8 平滑移动

图 10-9 游戏启动

图 10-10 游戏结束

10.3.1.3 程序框架

```
#include <curses.h>
…

/*变量定义*/
/*设置定时器*/
int set_ticker(long n_msecs )
{
        …
}
```

```
/*绘制图像*/
void paint( ){
        …
}

int main(int argc, char *argv[ ])
{
   chtype input;
   long delay = 100;
   /*初始化*/
   while ( (input=getch( )) && input!=ERR && input!='q' ) {
      switch( input ){
      …
      }
   }
   …
}
```

10.3.2 实验 2 利用多线程实现弹球游戏

10.3.2.1 实验说明

在 10.3.1 小节的实验中，解决方案通过定时器实现动画的绘制。程序执行时通过不断响应 SIGALRM 信号，定时对屏幕进行刷新。在本实验中，需要通过多线程实现动画绘制。

10.3.2.2 解决方案

上面的解决方案实际上是对多线程的一种模拟，通过信号机制，程序可以同时进行用户输入响应和屏幕绘制。基于这样的考虑，可以通过显示的多线程编程完成动画的绘制。

在多线程结构中，主线程进行用户输入的响应，主线程创建出一个独立的线程进行屏幕的绘制。屏幕绘制线程可以通过 usleep 函数进行休眠，完成屏幕的定时重绘。游戏程序的框架如下：

```
void* paint_thread(void* arg);

int main( )
{
        …
        pthread_create(...)/*创建屏幕绘制线程*/
        …
}
```

```
void* paint_thread(void* arg){
    while (true) {
        /*绘制屏幕*/
        /*通过 usleep( )函数休眠*/
    }
}
```

关于 pthread_create( )的使用，可以参考本书第 2 章。

10.3.2.3 程序框架

```
#include <curses.h>
#include <string.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <pthread.h>

int main( )
{
  …
  while ( (input=getch( )) && input!=ERR && input!='q' ) {
    /*读入用户输入*/
    /*如果开始新游戏，进行初始化*/

  }
  …
}

/*开始一个新游戏的初始化工作*/
void new_game( ){
  /*进行游戏初始化*/
  /*创建绘图线程*/
  pthread_t tidp;
  pthread_create(&tidp, NULL, paint_thread, NULL);
}

/*绘制图像*/
void* paint_thread(void* arg) {
  while (true){
    /*绘图*/
    /*通过 usleep( )函数休眠*/
  }
}
```

第 11 章　综合实验：一个小型远程访问 FTP 服务系统

11.1　实 验 目 的

• 加深对进程、线程、进程互斥、同步、通信、文件系统及网络编程等概念的理解，能综合运用所学知识，并用来分析和验证有关理论问题和解决实际问题。

• 理解基于客户/服务器计算模型，掌握支持并发用户访问的分布式软件系统的设计方法及核心技术。

• 实现一个小型远程访问 FTP 服务系统，本实验需要综合运用前面几章介绍的概念、技术和方法，且有一定的程序量(约 4 000 行)，可作为操作系统的一个综合性实验或课程设计题。

11.2　背 景 知 识

11.2.1　客户/服务器计算模型

客户/服务器模型是一种基于局域网或广域网的分布式系统，提供数据和服务的计算机称为服务器(server)，向服务器提出数据请求和服务要求的计算机称为客户(client)，这样的模型称为客户/服务器模型。著名的万维网基本上是一个客户/服务器模型，客户通常为个人机或工作站，为终端用户提供友好的图形界面；Web 站点是服务器，为用户提供各种数据和服务，最常见的是数据库服务、邮件服务和文件服务等。该模型的思想是：把操作系统看做一组协议进程，用户称作客户，协作进程称作服务器，客户和服务器通常全部运行相同网络操作系统，一台计算机可运行一个客户进程，也可运行多个客户进程、多个服务器进程或两者的混合。为了避免面向连接协议所造成的大量开销，客户/服务器模型通常基于简单的、无连接的请求/应答协议，客户向服务器发一个请求消息，要求提供某些服务(如查询信息)，服务器工作并返回所需的数据或出错代码，请求和应答消息都要经过内核进行。

客户/服务器模型的优点是简洁和高效率，该模型所用协议集比 OSI 的小，因而效率高。如果所有计算机都相同，那么只需要 3 层协议：物理层、数据链路层和请求应答层(相当于 OSI 的会话层)，前两层协议处理客户与服务器之间来往的数据分组，这些是由硬件处理的，不需要路由选择，也不需要建立链接。在请求/应答层有相应的协议，它定义一组合法的请求和合法的应答，由于结构简单，操作系统的通信机制只需提供函数：发送消息和接收消息，这两个函数可通过函数库加以封装。

传统客户/服务器体系结构包括两层：客户层和服务器层。近年来，3 层结构模型变得日益普遍，在这种结构中，应用软件分布在 3 种类型的机器上：用户层计算机(客户)、中间层服务器(应用服务)及后端服务器(数据资源)。用户计算机一般是瘦客户机。中间层计算机基本上是位于客户和很多后端数据库之间的网关，它能够转换协议，将一种类型的数据库查询映射为另一种类型的数据库查询；另外，中间层计算机能够融合来自不同数据源的结果，它还因介于两个层次之间而可充当桌面应用程序和后端应用程序之间的网关。中间层计算机和后端服务器之间的交互也遵循客户/服务器模型，因此，中间层同时充当着客户机和服务器。

此外，基于多层客户/服务器模型结构的应用框架用得越来越广泛，SUN 公司的 J2EE 体系结构可用于构建复杂的 Web 应用，J2EE 采用 4 层结构：Web 浏览器、Web 服务器、应用程序服务器和数据服务器。用户通过 Web 页面提交请求和接收结果；Web 服务器负责接收来自用户的请求并转交给应用服务器，另一方面负责接收来自应用服务器的计算结果并组织成 Web 页面返回给客户端；应用服务器负责具体的逻辑计算，需要时访问数据资源服务器以获得计算所需数据资源；数据资源服务器只负责应用数据的保存与维护，并对外提供访问数据的接口。

11.2.2 中间件

11.2.2.1 中间件的概念

客户/服务器计算模型解决了单个系统的互通互连以及独立的应用系统的开发和使用，对分布式计算的影响日益显著，但由于不同系统往往基于不同开发环境和运行环境，因此系统之间的互操作性实现困难。于是对分布式计算的标准化提出了很高要求，缺乏标准化难以实现集成的、多厂商的、企业范围的客户/服务器配置，因为客户/服务器方式的很大优势是与其模块化及将平台和应用程序混合、协调来提供商业解决方案的能力紧密相连的，这种“互操作性”问题必须得到很好解决。

实现这一要求的一种方法是：在上层应用程序和下层通信软件及操作系统之间使用标准编程接口和协议，建立支持在异构网络、异构计算机和不同操作系统中访问系统资源的工具，这种标准化的接口、协议和工具被称为中间件(middleware)。

目前已经有很多中间件软件包，有些简单而有些非常复杂，共同特点是能隐蔽不同网络协议和操作系统的复杂性和不一致性，客户机和服务器厂商一般都提供很多流行的中间件软件包，以供选择，这样，用户可采取一种特定的中间件策略，然后从厂商那里集成设备来支持这种策略。

11.2.2.2 中间件体系结构

图 11-1 给出在客户/服务器体系结构中的中间件的作用。中间件的确切作用将取决于所使用的客户/服务器计算的类型，即应用程序的功能划分。中间件具有客户端组件和服务器端组件两个部分，它的基本作用是使位于客户端的应用程序或用户能够访问服务器上的各种服务，同

时无需考虑服务器之间的区别。例如，对于特定的应用领域，结构化查询语言 SQL 提供本地或远程用户访问关系数据库的标准访问方式，然而，很多数据库厂商都对 SQL 进行特定扩展，这就使众多产品有所差别，产生潜在的不兼容性问题。

中间件提供一个软件层，其目的是隐蔽下层网络操作系统和平台的异构性，支持不兼容系统的统一访问，提供一致的分布式服务接口，把底层服务综合起来，以一种更透明、更高级形式为上层应用提供服务。从这个意义上来说，中间件好像是分布式操作系统。从逻辑的角度而不是从实现的角度来看，中间件所扮演的角色如图 11-2 所示，分布式系统可看作一组应用程序和用户可用资源的集合，用户无需关心数据或应用程序的实际位置，所有应用程序操作建立在统一的 API 之上。中间件使分布式客户/服务器计算模式互操作成为可能，它贯穿所有客户和服务器平台，负责将客户请求定位到合适的服务器上，中间件软件是用于解决互操作性的，它的实现通常基于底层通信机制：消息传递和远程过程调用。

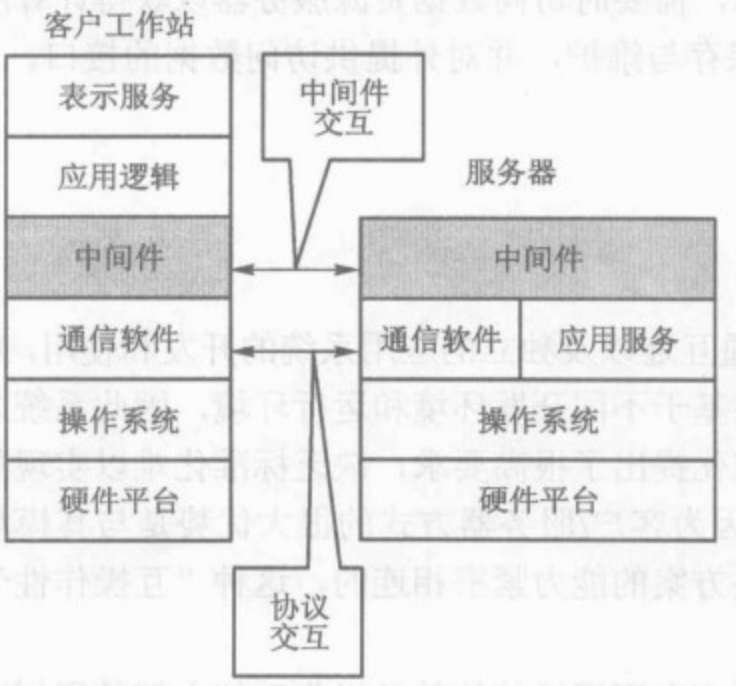


图 11-1　中间件在客户/服务器结构中的作用

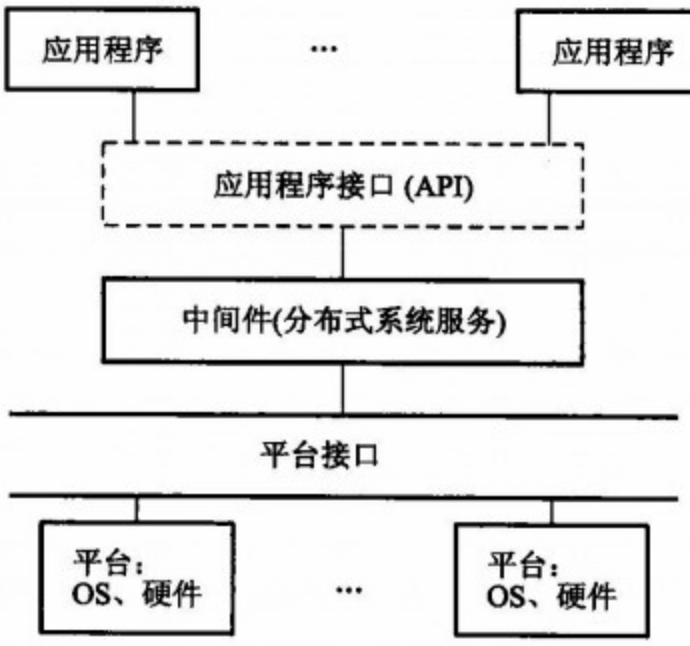


图 11-2　中间件的逻辑视图

大多数客户/服务器模型都采用中间件技术来实现，无论客户如何向服务器发送请求，无论客户访问哪一个数据库，都通过中间件来帮助客户沟通从用户界面到数据访问的通路；同时，中间件还起着增强透明性的作用，如隐蔽通信协议、隐蔽数据库查询语言、转换异构硬件的数据格式等。迄今，虽然没有一种标准的分布式计算环境，但已出现的各种中间件，如 Microsoft 的 COM(Component Object Model)、OSF 的 DCE(Distributed Computing Environment)和 OMG 的 CORBA(Common Object Request Broker Architecture)等一系列实现方法推动着不同的分布式环境的迅速发展。

中间件技术已在业界得到广泛应用，并有相应的技术开发支撑平台。因此，本小节只对相关概念做了简单介绍，本章实验内容并未涉及该技术。

11.2.3 FTP 技术简介

文件传输协议(File Transfer Protocol，FTP)是 TCP/IP 协议族中的协议之一，用于 Internet 上控制文件的双向传输。同时，FTP 协议也是一个分布式应用程序，使 Internet 用户可以通过自己的个人计算机与分散于分布式网络环境下、运行 FTP 协议的任一服务器相连，进而访问服务器上的大量程序和信息。FTP 协议的主要作用是，让用户通过 FTP 协议，连接到一个运行 FTP 服务器程序的远程计算机，查看远程计算机上的现有文件，然后将文件从远程计算机复制到本地计算机——下载(download)文件，或把本地计算机的文件传送到远程计算机去——上传(upload)文件。简单说，FTP 就是完成两台计算机之间的文件复制。在 TCP/IP 协议族中，FTP 标准命令 TCP 端口号为 21，PORT 方式数据端口为 20。

与大多数 Internet 服务一样，FTP 协议也是一个客户/服务器系统。客户通过一个客户机程序连接至在远程计算机上运行的服务器程序。依照 FTP 协议提供的服务，进行文件传送的计算机是 FTP 服务器；而连接 FTP 服务器、遵循 FTP 协议与服务器传送文件的计算机是 FTP 客户机。在通过 FTP 协议从远程计算机复制文件时，实际上需要启动两个程序：一个是 FTP 客户端上的 FTP 客户程序，用于向 FTP 服务器提出复制文件的请求；另一个是 FTP 服务器上的 FTP 服务程序，响应客户请求并将指定文件传送到 FTP 客户计算机。

使用 FTP 协议实现文件传输时，FTP 服务过程与 FTP 服务器及 FTP 客户机所处的位置、连接的方式、操作系统类型等无关。FTP 命令在不同操作系统上的格式有一些细微差别，不过每种协议基本的命令结构是相同的。当客户需要访问(或“登录”)FTP 服务器时，必须要有该 FTP 服务器授权的账号。也就是说，只有在拥有了一个客户标识和相应密码后才能登录 FTP 服务器，享受 FTP 服务器提供的服务。使用 FTP 访问远程服务器的命令格式如下：

ftp://客户名:密码@FTP 服务器 IP 或域名：FTP 命令端口/路径/文件名

以上参数中，除 FTP 服务器 IP 或域名为必要项外，其余都是可选项。如以下地址都是有效 FTP 访问命令格式：

- ftp://ftp.cs.nju.edu.cn
- ftp://user:ftp.cs.nju.edu.cn
- ftp://user:ftp.cs.nju.edu.cn:2008
- ftp://user:passwd@ ftp.cs.nju.edu.cn:2008/soft/list.txt

在 Internet 中，绝大多数 FTP 服务器都提供“匿名”(anonymous)服务功能，支持客户以匿名身份登录到 FTP 服务器并访问远程主机上公开的文件，而不必是该 FTP 服务器的注册客户。匿名 FTP 一直是 Internet 上获取信息资源的最主要方式，在 Internet 成千上万的匿名 FTP 主机中存储着无以计数的文件，客户只要知道特定信息资源的主机地址，就可以用匿名 FTP 登录获取所需的信息资料。但在实际应用中，许多系统要求客户将 E-mail 地址作为密码，以便更好地对客户访问进行跟踪。

从传输角度，FTP 有两种传输模式，即 ASCII 传输模式和二进制数据传输模式。

• ASCII 传输模式。当客户复制的文件包含 ASCII 码文本时，如果远程主机上运行的不是 UNIX，FTP 协议通常会自动地调整文件内容，以便于把文件解释成远程主机对应的文本文件存储格式。但是常常会有这样的情况：客户正在传输的文件包含的不是文本文件，它们可能是程序、数据库、字处理文件或压缩文件(尽管字处理文件中大部分是文本，其中也包含有指示页尺寸，字库等信息的非打印字符)。在这种情形下，可在复制非文本文件之前，用 binary 命令告诉 FTP 协议逐字复制，而不要对这些文件进行处理。该传输方式即为二进制传输模式。

• 二进制传输模式。该传输模式将保存文件的位序，以便本地副本与原始文件是逐位一一对应的。需要注意，有时在目的地计算机上包含位序列的文件是没意义的。例如，若以二进制方式传输模式将 Macintosh 系统中的可执行文件传送到 Windows 系统，则在 Windows 系统上将无法执行该文件。

FTP 协议有两种不同工作模式，一种是 Standard (即 PORT 模式，主动模式)，另一种是 Passive (也就是 PASV，被动模式)。Standard 模式 FTP 的客户端发送 PORT 命令到 FTP 服务器。Passive 模式 FTP 的客户端发送 PASV 命令到 FTP 服务器。这两种方式的工作原理如下。

在 Standard 模式下，FTP 客户端首先与 FTP 服务器的 TCP 21 端口建立连接，客户端需要接收数据的时候，在这个连接通道上发送 PORT 命令。PORT 命令包含客户端接收数据的端口。在传送数据的时候，服务器端通过自己的 TCP 20 端口连接至客户端的指定端口并发送数据。Passive 模式的控制通道的建立过程与 Standard 模式类似，但建立连接后发送的不是 PORT 命令，而是 PASV 命令。FTP 服务器收到 PASV 命令后，随机打开一个高端端口(端口号大于 1024)，并通知客户端在这个端口上传送数据请求。客户端将通过此端口建立到 FTP 服务器的连接。随后，FTP 服务器将通过该连接进行数据传送。

在实际应用中，很多防火墙在设置的时候都不允许接受外部发起的连接。因此，许多位于防火墙后或内网的 FTP 服务器不支持 Passive 模式，这是因为客户端无法穿过防火墙打开 FTP 服务器的高端端口；与此同时，许多内网的客户端也不能用 Standard 模式登录 FTP 服务器，其原因在于从服务器的 TCP 20 无法与内部网络的客户端建立一个新的连接，以致无法工作。

11.3　综合实验功能设计

借鉴 Internet 上 FTP 服务的功能设计，采用客户/服务器模型，实现一个支持远程文件传输与共享的小型 FTP 服务系统。与标准 FTP 服务软件类似，该小型远程访问 FTP 服务系统需要由服务器端软件和客户端软件两部分组成。其中服务器端软件在指定端口接受客户连接请求，并根据客户请求执行相应处理。客户端应用程序为客户提供访问此小型远程访问 FTP 服务系统的交互界面。具体而言，该 FTP 服务系统具备以下几个基本功能。

1. 基于套接字的客户/服务器通信模式

服务器在指定 TCP 端口 sport 启动 FTP 模拟系统的服务器软件，等待并接收客户请求。客户端通过 TCP 协议向服务器的端口 sport 发送连接请求，若通过服务器端的接入控制，将通过

该连接与服务器实现交互。

2. 远程登录功能

为支持身份验证，服务器端软件还需集成 Linux 系统的客户管理功能，对客户身份信息予以验证。客户在发送连接请求时，必须提供 FTP 服务器软件所驻留的 Linux 主机上客户名及密码。客户发送服务请求的命令格式如下：

ftps://客户名:密码@FTP 服务器 IP 或域名：FTP 命令端口

3. 并发执行及管理功能

为提高系统运行效率，服务器软件将采用多线程技术响应客户请求。当客户请求通过身份验证后，服务器将创建一个新线程来响应客户请求，为客户提供服务。

4. 活动客户计数功能

站点计数为客户提供统计当前系统中的活动客户个数的功能。当客户通过身份验证后，活动客户计数器加 1。当客户断开连接后，活动客户计数器减 1。

5. 文件管理功能

客户请求被调度后，支持客户执行以下操作。

- 在服务器端执行基本文件操作，包括创建/删除目录(mkdir/rmdir)、切换目录(cd)、查看当前目录下的所有文件(ls)。
- 在客户端执行基本文件操作，包括创建/删除目录(lmkdir/lrmdir)、切换目录(lcd)、查看当前目录下的所有文件(ldir)。
- 设定文件传输模式，包括文本模式(ascii)或二进制模式(bin)。
- 文件传输功能，包括上传/下载文件(upload/download)到指定目录。

11.4 综合实验解决方案

11.4.1 服务器端接收客户请求的套接字结构

为实现基于套接字的客户/服务器通信，服务器程序需与服务器 IP 地址绑定，并在指定端口监听客户请求。以下为服务器端接收客户请求的 SOCKET 代码结构。

1. 启动服务函数

```
SOCKET FTP_RunServer(int nPort){
    SOCKET nServerSocket = INVALID_SOCKET;
    struct sockaddr_in tAddress;
    nServerSocket = socket(AF_INET,SOCK_STREAM,0);
    if (nServerSocket == INVALID_SOCKET){
        return INVALID_SOCKET;
    }
    tAddress.sin_port = htons((unsigned short)nPort);
```

```
        tAddress.sin_family = AF_INET;
        tAddress.sin_addr.s_addr = htonl(INADDR_ANY);
        if (bind(nServerSocket,(const struct sockaddr*)&tAddress,sizeof(struct sockaddr_in)) == INVALID_
        SOCKET){
            return –1;
        }
        if (listen(nServerSocket,SOMAXCONN) == INVALID_SOCKET){
            return INVALID_SOCKET;
        }
        return nServerSocket;
    }
```

2. **接收客户请求函数**

```
    SOCKET FTP_Accept(SOCKET nSocket, in_addr_t *ptClientIP){
        struct sockaddr_in tAddr_in;
        socklen_t nLength = sizeof(struct sockaddr_in);
        SOCKET nAccSocket = accept(nSocket,(struct sockaddr*)&tAddr_in,&nLength);

        if (nAccSocket < 0){
            return INVALID_SOCKET;
        }
        if (ptClientIP)*ptClientIP = tAddr_in.sin_addr.s_addr;
        return nAccSocket;
    }
```

11.4.2　客户端发送套接字连接请求的核心代码

客户端连接函数如下：

```
    SOCKET FTP_Connect(in_addr_t nConnectTo, in_port_t nPort){
        SOCKET nClientSocket;
        struct sockaddr_in tServerAddress;
        nClientSocket = socket(AF_INET,SOCK_STREAM,0);
        if (nClientSocket == –1){
            printf("Failed to create socket at FTP_Connect\n");
            return INVALID_SOCKET;
        }
        tServerAddress.sin_port = nPort;
        tServerAddress.sin_addr.s_addr = nConnectTo;
        tServerAddress.sin_family = AF_INET;
        if (connect(nClientSocket,(struct sockaddr*)&tServerAddress,
            sizeof(struct sockaddr_in)) == INVALID_SOCKET){
            printf("failed to connect at FTP_Connect\n");
```

```
            return INVALID_SOCKET;
        }
        printf("connected to %s:%d\n",FTP_GetIP(nConnectTo),(int)ntohs(nPort));
        return nClientSocket;
    }
```

11.4.3 与线程处理相关的核心函数

根据上文关于服务器端程序的功能设计，为提高系统并发效率，服务器软件将采用多线程技术，为每个客户会话创建一个线程。与此同时，对活动客户计数器的访问是一个多线程的读/写同步问题，必须实现互斥访问。以下代码段给出了与线程处理相关的核心函数。

1. **启动线程**

```
int StartThread(tThread* pThread,ThreadFunc pFunc,void *pParam){
    if (pthread_create(pThread,NULL,(void*)pFunc,pParam) == 0)
        return 0;
    else
        return –1;
}
```

2. **线程阻塞**

```
int JoinThread(tThread hThread){
    if (pthread_join(hThread,NULL) != 0)
        return –1;
    else
        return 0;
}
```

3. **释放进程占用资源**

```
int DetachThread(tThread hThread){
    if (pthread_detach(hThread) != 0)
        return –1;
    else
        return 0;
}
```

4. **初始化互斥变量**

```
void InitializeMutex(tMutex* pMutex){
    if (pMutex){
        pthread_mutex_init(pMutex,NULL);
    }
}
```

5. **销毁互斥变量**

```
int DestroyMutex(tMutex* pMutex){
```

```
        pthread_mutex_destroy(pMutex);
        return 0;
    }
```

6. **互斥变量加锁**

```
    int LockMutex(tMutex *pMutex){
        if (pthread_mutex_lock(pMutex) != 0)
            return –1;
        else
            return 0;
    }
```

7. **互斥变量解锁**

```
    int UnlockMutex(tMutex* pMutex){
        if (pthread_mutex_unlock(pMutex) != 0)
            return –1;
        else
            return 0;
    }
```

11.4.4　接收客户请求与实现客户会话的线程

为优化系统效率，服务器软件将对客户接入的处理放置在一个线程中完成。该线程负责接收客户请求(接入控制)，实施身份验证。若通过验证，将创建一个新的线程，实现与客户的会话(客户会话)。

1. **接入控制线程**

```
unsigned int THREADCALL FTP_ServerThread(void *pParam){
    g_nServerSocket =FTP_RunServer(SPORT);
    if (g_nServerSocket == –1){
        printf("cannot start server. terminated.\n");
        return –1;
    }
    printf("server launched. waiting for users...\n");

    for (;;){
        in_addr_t tClientAddress;
        SOCKET nClientSocket = FTP_Accept(g_nServerSocket,&tClientAddress);
        if (nClientSocket == –1){
            printf("cannot accept new user. terminated.\n");
            return –1;
        }
        if (checkuser(username, password) != –1){
```

```
                    LockMutex(&g_tCounterMutex);
                    Counter++;
                    UnlockMutex(&g_tCounterMutex);
                    if (FTP_CreateNewClientThread(nClientSocket, tClientAddress) == -1){
                        printf("cannot add new thread. terminated.\n");
                        return -1;
                    }
                } else
                    printf("Invalid username or password!\n");
            }
            return 0;
        }
        int StartServer(){
            StartThread(&g_tFTP_ServerThread, FTP_ServerThread,NULL);
            DetachThread(g_tFTP_ServerThread);
            return 0;
        }
```

2. 创建客户会话线程的函数

```
        int FTP_CreateNewClientThread(SOCKET nSocket,in_addr_t tClientAddress){
            tClientInfo *pNewInfo = NULL;
            pNewInfo = (tClientInfo *)malloc(sizeof(tClientInfo));
            if (!pNewInfo){
                printf("cannot alloc memory for tThreadInfo\n");
                return -1;
            }
            pNewInfo->m_nSocket = nSocket;
            pNewInfo->m_tClientAddress = tClientAddress;
            StartThread(&g_tFTP_ClientThread, FTP_ClientThread, pNewInfo);
            return 0;
        }
```

3. 客户会话线程

```
        unsigned int THREADCALL FTP_ClientThread(void *pParam){
            tClientInfo *pClientInfo = (tClientInfo *)pParam;
            printf("Child Thread started.\n");
            if (FTP_RecvMessage(pClientInfo ->m_nSocket,&tRecvMessage) == -1){
                printf("failed to receive first message from client\n");
                return -1;
            }
            LockMutex(&g_tCounterMutex);
```

```
        counter++;
        UnlockMutex(&g_tCounterMutex);
        for( ; ; ; ){     /*与客户的交互处理*/
            performRoutine(tRecvMessage, pClientInfo);    /*根据客户交互信息类型，做相应处理*/
        }
        FTP_CloseConnection(pClientInfo->m_nSocket);
        printf("Child Thread ended.\n");
        return 0;
    }
```

11.4.5　文件管理

文件管理功能主要包括 4 部分。其中“服务器端基本文件操作”的实现可通过调用 exec( ) 函数来执行相应命令，并将结果通过套接字消息传递机制返回到客户端；“客户端基本文件操作”的实现亦可通过直接调用 exec( )函数来执行相应命令，并将输入结果重定向到客户界面。

1. 以 ASCII 模式传输文件

文件传输模式实质反映了文件的读写模式。当以 ASCII 模式传输文件的时候，涉及的读写函数包括 fget( )和 fputs( )。

(1) 读文本文件

```
if((fp=fopen(filename,"r"))==NULL) { /* 打开一个由 argv[1]所指的文件*/
    printf("not open");
    exit(0);
}
while ((strlen(fgets(str,1024,fp1)))>0) {
    …
}
   fclose(fp);
```

(2) 写文本文件

```
if((fp=fopen(filename,"w"))==NULL)  { /*以只写方式打开文件 2 */
    printf("cannot open file\n");
    exit(0);
}
…
fputs(str, fp );
```

2. 以二进制模式传输文件

当以二进制模式传输文件的时候，涉及的读写函数包括 fread 和 fwrite( )。

(1) 读二进制文件

```
unsigned char buf[MAXLEN];
int rc;
```

```
if( (fp = fopen(filename, "rb" )) == NULL){
        printf("cannot open file\n");
        exit(0);
}
while( (rc = fread(buf,sizeof(unsigned char), MAXLEN,fp)) != 0 ) {
        … /*通过网络发送文件*/
}
fclose(fp);
```

(2) 写二进制文件

```
unsigned char buf[MAXLEN];
int rc;

if( (fp = fopen(filename, "wb" )) == NULL){
        printf("cannot open file\n");
        exit(0);
}
…/*通过网络获得文件数据*/
fwrite( buf, sizeof(unsigned char), rc, outfile);
```

11.4.6 套接字通信

文件上传与下传的实现需要通过套接字通信来实现。在该 FTP 软件，客户与服务器之间将涉及多次交互与通信，并且涉及不同类型的消息通信。

1. 定义消息类型

为便于客户与服务器双方解析消息类型及参数，对消息类型做如下定义。

```
typedef enum _tMsgType{
        FTP_USERAUTH,
        FTP_DOWNLOAD,
        FTP_UPLOAD,
        FTP_CD,
        FTP_MKDIR,
        FTP_RMDIR,
        FTP_LS,
        FTP_ASCII,
        FTP_BIN
} tPtclType;

typedef struct _tPtclHeader{
        uint32_t m_nVersion;
```

```
    tPtclType m_nType;
    uint32_t m_nBytes;
} tPtclHeader;

typedef struct _tPtclMsg{
    tPtclHeader m_tHeader;
    void* m_pData;
} tPtclMsg;
```

2. 发送元数据

```
int FTP_Send(SOCKET nSocket, const void* pBuffer, int nSize){
    return (int)send(nSocket, pBuffer, nSize, 0);
}
```

3. 阻塞式发送数据

```
int FTP_SendUntilAll(SOCKET nSocket,const void *pBuffer,int nSize){
    int nSent = 0;
    int nSentBytes;
    for (;;){
        nSentBytes = FTP_Send(nSocket,(void*)&((char*)pBuffer)[nSent],nSize - nSent);
        if (nSentBytes <= 0)return –1;
        nSent += nSentBytes;
        if (nSent == nSize)return 0;
    }
}
```

4. 发送消息头

```
int FTP_SendHeader(SOCKET nSocket, const tPtclHeader* pHeader){
    uint32_t bufHeader[4];
    bufHeader[0] = htonl(pHeader->m_nVersion);
    bufHeader[1] = htonl(pHeader->m_nType);
    bufHeader[3] = htonl(pHeader->m_nBytes);
    if (FTP_SendUntilAll(nSocket,bufHeader,sizeof(uint32_t)*4) == –1){
        printf("Connection lost at sending packet header:FTP_SendMessage\n");
        return –1;
    }
    return 0;
}
```

5. 发送消息体

```
int FTP_SendMessage(SOCKET nSocket, const tPtclMsg* pMsg){
    uint32_t bufEndMarker =htonl(UMPP_END_MARKER);
    if (!pMsg || nSocket == –1 ||
        (pMsg->m_tHeader.m_nBytes != 0 && !pMsg->m_pData)){
```

```
        return -1;
    }
    if (FTP_SendHeader(nSocket,&pMsg->m_tHeader) == -1)return -1;
    if (pMsg->m_tHeader.m_nBytes != 0){
        if (FTP_SendUntilAll(nSocket,pMsg->m_pData,pMsg->m_tHeader.m_nBytes) == -1){
            printf("Connection Lost at sending packet message:FTP_SendMessage(%d)\n"
            ,nSocket);
            return -1;
        }
    }
    if (FTP_SendUntilAll(nSocket,&bufEndMarker,sizeof(uint32_t)) == -1){
        printf("Connection Lost at sending endmarker:FTP_SendMessage(%d)\n",nSocket);
        return -1;
    }
    return 0;
}
```

6. **接收元数据**

```
int FTP_Receive(SOCKET nSocket, void *pBuffer, int nSize){
    return (int)recv(nSocket, pBuffer,nSize,0);
}
```

7. **阻塞式接收数据**

```
static int ReceiveUntilFull(SOCKET nSocket,void *pBuffer,int nSize){
    int nReceived = 0;
    int nGotBytes;
    for (;;){
        nGotBytes = FTP_Receive(nSocket,(void*)&((char*)pBuffer)[nReceived],nSize - nReceived);
        if (nGotBytes <= 0)return -1;
        nReceived += nGotBytes;
        if (nReceived == nSize)return 0;
    }
}
```

8. **接收消息体**

```
int FTP_RecvMessage(SOCKET nSocket, tPtclMsg* pMsg){
    uint32_t bufHeader[4];
    uint32_t bufEndMarker;
    if (!pMsg || nSocket == -1){
        return -1;
    }
    pMsg->m_pData = NULL;
```

```
if (ReceiveUntilFull(nSocket,bufHeader,4*sizeof(uint32_t)) == -1){
        printf("Connection Lost at receiving packet header:FTP_RecvMessage(%d)\n",nSocket);
        return -1;
}

pMsg->m_tHeader.m_nVersion= ntohl(bufHeader[0]);
pMsg->m_tHeader.m_nType = ntohl(bufHeader[1]);
pMsg->m_tHeader.m_nBytes = ntohl(bufHeader[3]);

if (pMsg->m_tHeader.m_nBytes == 0){
        pMsg->m_pData = NULL;
}
else{
        pMsg->m_pData = malloc(pMsg->m_tHeader.m_nBytes);
        if (!pMsg->m_pData){
                printf("Cannot malloc %dbytes for getting data at FTP_RecvMessage(%d)\n",pMsg->m_
                  tHeader.m_nBytes,nSocket);
                return -1;
        }
        if (ReceiveUntilFull(nSocket,pMsg->m_pData,pMsg->m_tHeader.m_nBytes) == -1){
                printf("Connection Lost at receiving message:FTP_RecvMessage(%d)\n",nSocket);
                SAFE_RELEASE(pMsg->m_pData)
                return -1;
        }
}

        if (ReceiveUntilFull(nSocket,&bufEndMarker,sizeof(uint32_t)) == -1){
                printf("Connection Lost at receiving end marker:FTP_RecvMessage(%d)\n",nSocket);
                SAFE_RELEASE(pMsg->m_pData)
                return -1;
        }
        bufEndMarker = ntohl(bufEndMarker);
        if (bufEndMarker != UMPP_END_MARKER){
                printf("Recieved non-EndMarker %x :FTP_RecvMessage(%d)\n",bufEndMarker,nSocket);
                SAFE_RELEASE(pMsg->m_pData)
                return -1;
        }
        return 0;
}
```

11.5 综合实验程序框架

11.5.1 客户端代码框架

```
int main(int argc,char **argv){
if(argv[1]不符合 ftps 格式)
    return –1;
char server[30], user[20], passwd[20];
int modeFlag = 0 ; /*传输模式，默认为 ASCII 模式*/
int port, socket;
    /*从 conString 中解析服务器地址 server，服务端口 port, 用户名 user*/
If((socket =FTP_Connect(server, port)) == –1)
    return –1;
fgets(passwd);
/*用空格符连接 user 及 passwd，形成一个新串 userStr*/
PtclMsg * pSendMsg, *pRecvMsg;
pSendMsg -> m_nType = FTP_USERAUTH;
pSendMsg -> m_pData = userStr;
FTP_SendMessage(socket, pSendMsg);          /*发送用户信息给服务器验证*/
FTP_RecvMessage(socket, pRecvMsg);          /*获取服务器对用户信息的验证结果*/
char *comStr[40], command[10], para[30];
fgets(comStr);

typedef enum _tMsgType{
        FTP_USERAUTH,
        FTP_DOWNLOAD,
        FTP_UPLOAD,
        FTP_CD,
        FTP_MKDIR,
        FTP_RMDIR,
        FTP_LS,
        FTP_ASCII,
        FTP_BIN
} tPtclType;

while (comStr != "quit"){
    /*分析 comStr，获取 command 及 para 部分*/
    if (command == "get"){
```

```
    pSendMsg -> m_nType = FTP_DOWNLOAD;
    pSendMsg -> m_pData = command;
    FTP_RecvMessage(socket, pRecvMsg);
    /*根据结果执行响应处理*/
    fgets(comStr);
    /*根据 modeFlag 确定写文件的模式*/
    continue;
}
if (command == "put"){
    pSendMsg -> m_nType = FTP_UPLOAD;
    pSendMsg -> m_pData = command;
    FTP_RecvMessage(socket, pRecvMsg);
    /*根据结果执行响应处理*/
    fgets(comStr);
    /*根据 modeFlag 确定读文件的模式*/
    continue;
}
if (command == "cd"){
    pSendMsg -> m_nType = FTP_CD;
    pSendMsg -> m_pData = command;
    FTP_RecvMessage(socket, pRecvMsg);
    /*根据结果执行响应处理*/
    fgets(comStr);
    continue;
}
if (command == "mkdir"){
    pSendMsg -> m_nType = FTP_MKDIR;
    pSendMsg -> m_pData = command;
    FTP_RecvMessage(socket, pRecvMsg);
    /*根据结果执行响应处理*/
    fgets(comStr);
    continue;
}
if (command == "rmdir"){
    pSendMsg -> m_nType = FTP_RMDIR;
    pSendMsg -> m_pData = command;
    FTP_RecvMessage(socket, pRecvMsg);
    /*根据结果执行响应处理*/
    fgets(comStr);
    continue;
```

```
}
if (command == "ls"){
   pSendMsg -> m_nType = FTP_LS;
   pSendMsg -> m_pData = command;
   FTP_RecvMessage(socket, pRecvMsg);
   /*根据结果执行响应处理*/
   fgets(comStr);
   continue;
}
if (command == {"ascii"){
   modeFlag = 0;
   pSendMsg -> m_nType = FTP_ASCII;
   pSendMsg -> m_pData = command;
   FTP_RecvMessage(socket, pRecvMsg);
   /*根据结果执行响应处理*/
   fgets(comStr);
   continue;
}
if (command == "bin"){
   modeFlag = 1;
   pSendMsg -> m_nType = FTP_BIN;
   pSendMsg -> m_pData = command;
   FTP_RecvMessage(socket, pRecvMsg);
   /*根据结果执行响应处理*/
   fgets(comStr);
   continue;
  }
 }
}
```

11.5.2 服务端代码框架

```
int main(int argc,char **argv){
    signal(SIGTERM, SignalHandler);
    int count = 0;
    static tMutex g_tCounterMutex;
    InitializeMutex(g_tCounterMutex);
    if (StartScheduler( ) == -1)
        return -1;
    TerminalCommand( );
        return 0;
```

```
}

void SignalHandler(int nSignalNo){
    if (nSignalNo != SIGTERM)
        return;
    TerminateProgram( );
}

void TerminateProgram( ){
    exit(0);
}

void TerminalCommand( ){
    char szCommandBuffer[COMMAND_BUFFER];
    for (;;){
    printf("> ");
    fgets(szCommandBuffer,COMMAND_BUFFER,stdin);
    if (strncmp("exit",szCommandBuffer,4) == 0||
       strncmp("quit",szCommandBuffer,4) == 0||
       strncmp("bye",szCommandBuffer,3) == 0)
        break;
    }
}
```

第12章 内核模块

12.1 实验目的

- 了解内核模块的概念和特点。
- 学习如何编写一个内核模块。
- 掌握内核模块的实现机制，学会内核模块的加载和卸载。

12.2 背景知识

12.2.1 内核模块概述

模块是在内核空间运行的程序，实际上是一种目标文件，不能单独运行但其代码可在运行时链接到系统中作为内核的一部分运行或卸载。Linux 内核模块是一种特有的机制，它由一组函数和数据结构组成，可作为独立程序来编译，当模块被安装时，它被链接到内核中。可在系统启动时进行模块安装，称静态加载；也可在系统运行时进行模块安装，称动态加载。模块的主要作用是动态地增加或减少内核功能，许多情况下用户需要增加内核态程序，例如，添加一个文件系统或设备驱动程序，而这类程序运行在内核态，工作在内核空间，由于它们种类繁多、体积庞大，要求内核全部包含进去是十分困难的。通过精心设计，Linux 内核较好地解决了这个问题，提供一种称为可加载内核模块(Loadable Kernel Modules，LKM)机制，它可让一个文件系统或设备驱动程序加载到内核空间去运行，通过模块机制来实现系统运行时对内核功能的动态扩充，就能大大提高单内核操作系统的灵活性与可扩展性。

Linux 内核模块是一个编译好的、具有特定格式的独立目标文件，用户可通过系统提供的一组与模块相关的命令将内核模块加载进内核，当内核模块被加载后，它有以下特点：

- 与内核一起运行在相同的内核态和内核地址空间。
- 运行时具有与内核同样的特权级。
- 可方便地访问内核中的各种数据结构。

被加载到内核的内核模块代码与静态编译进内核的代码没有区别，内核模块与内核中的其他模块交互只需采用函数调用。此外，内核模块还可以很容易地被移出内核，当用户不再需要某功能模块时，可以自动地将它从内核卸载以节省系统主存开销，配置十分灵活。

Linux 内核需要对加载的内核模块进行管理。管理内核模块主要有两项任务：一是内核符

号表管理；二是维护内核模块的引用计数。内核将资源登记在符号表中，内核模块被加载后，模块可以通过符号表使用内核中的资源，新模块加载到内核时，系统把新模块提供的符号加进符号表中，这样新加载模块就可访问已加载模块提供的资源；在卸载一个模块时，系统释放分配给该模块的所有系统资源，如内核主存区等，同时将该模块提供的符号从符号表中删除。

由于所有内核模块在加载后都在同一地址空间中，内核模块之间可相互引用各自导出的符号，因此内核模块之间会产生依赖性，如果 A 模块需要用到 B 模块导出的符号，而 B 模块没有被加载到内核，A 模块的加载就会出错；同样，一个内核模块如果有其他内核模块引用它导出的符号，内核也不允许该内核模块被卸载。内核模块的引用计数器便是用来管理内核模块之间的依赖性的。如果一个内核模块被依赖，它的引用计数就会增加；当依赖减少时，相应的引用计数也会减少；一个内核模块只有在引用计数为 0 时候才允许被卸载。

通常，内核开发人员通过直接修改源代码的方式增加或修改内核功能。在 GPL 协议下，开发人员可方便地获得内核源代码，并做相应修改。此法的优点在于：开发人员的自由度高，能够最大限度地给内核增加或修改功能。缺点在于：内核需要被重新编译、重新加载，这就意味着需要通过重新启动并引导内核来使新内核生效；新内核生效以后，如果用户不再需要新增加的功能时，只能再次引导旧内核，将增加的功能去除，这样对于没有编程基础的用户来说是十分困难的任务。而通过模块机制，在内核运行时动态加载或卸载的方法，既方便又简单，在调试内核代码时，用户不必每次修改后都重新编译内核和引导系统。简而言之，如果要修改原有内核的功能一般采用修改源代码的方法，如果是增加或减少功能一般采用内核模块方法更佳。

12.2.2 内核模块编程

编写内核模块需要使用内核中预先定义的宏，宏的作用是帮助内核来管理内核模块，由于内核模块在内核空间中运行，无法与 C 函数库连接，因此它的编写也受到内核编程的限制，如不能使用 C 库函数，不能使用浮点计算。内核模块编程在不同内核版本中各不相同，本书以 Linux 2.6 为基础讨论内核模块编程。

12.2.2.1 内核模块的结构

现在以一个简单的“Hello,world!”模块为例来学习如何编写内核模块，该模块的功能是在被载入内核时会向系统日志中写入“Hello,world!”；当该模块被卸载的时候，它也会向系统日志写入一条“Goodbye, world!”消息。该内核模块的代码如下：

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("GPL");

static int hello_init(void)
{
        printk(KERN_ALERT "Hello, world!\n");
```

```
        return 0;
}

static void hello_exit(void)
{
        printk(KERN_ALERT "Goodbye, world!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

此代码框架是一个内核模块的典型结构，一个内核模块应包含如下几部分：

• 头文件声明。头两行是模块头文件，头文件 module.h 和 init.h 是必不可少的。module.h 包含加载模块需要的函数和符号定义；init.h 中包含模块初始化和清理函数的定义。如果在加载时允许用户传递参数，模块还应该包含 moduleparam.h 头文件。

• 模块许可声明。从内核 v2.4.10 版本开始，模块必须通过 MODULE_LICENSE 宏声明此模块的许可证，否则在加载此模块时，内核会显示“kernel tainted”(内核被污染)的警告信息。从 linux/module.h 文件中可看到，被内核接受的许可证有 GPL、GPL v2、GPL and additional rights、Dual BSD/GPL、Dual MPL/GPL、Dual MIT/GPL 和 Proprietary。

• 初始化和清理函数声明。内核模块必须调用宏 module_init 和 module_exit 去注册初始化与清理函数。在“hello world!”模块源代码的最后两行已声明该模块被加载时的初始化函数是 hello_init( )，模块被卸载时的清理函数是 hello_exit( )。需要注意，初始化与清理函数必须在宏 module_init 和 module_exit 使用前定义，否则会出现编译错误。这两个函数配对使用，例如，当 module_init( )申请一个资源，那么 module_exit( )中就应释放这个资源，使得模块不留下任何副作用。

一个基本的模块只要包含以上的 3 个基本部分就能够正常地工作，Linux 内核还提供其他一些函数与宏，帮助开发者开发更加强大的内核模块，这些在以下几节进行讲解。

12.2.2.2 编译与加载

Linux 内核 v2.6 中，内核模块编译有较大变化。在 v2.4 中，模块的编译只需要内核源代码的头文件，在包含 Linux/module.h 之前定义 MODULE，编译、链接后生成的内核模块后缀为.o；在 v2.6 中，模块的编译需要配置过的内核源代码，没有内核源代码无法进行内核模块的编译工作，编译、链接后生成的内核模块后缀为.ko，编译过程中首先会到内核源代码目录下，读取顶层 makefile 文件，然后返回模块源代码所在的目录继续编译。编译内核模块的 makefile 文件非常简单，对于本小节“hello world!”示例，只需要下面一行：

```
obj-m:=hello.o
```

生成的内核模块为 hello.ko。如果需要生成一个名为 mymodule.ko 的内核模块，并且该内

核模块的源代码来源于modulesrc1.c和modulesrc2.c两个文件，makefile文件应该写成如下形式：

```
obj -m:=mymodule.o
module-objs:=modulesrc1.o modulesrc2.o
```

如果用户采用这种makefile，在调用make命令时，需要将内核源代码所在目录作为一个参数传递给make命令，例如，如果v2.6的内核源代码位于/usr/src/linux-2.6目录下，用户模块源代码所在目录应该使用的make命令为：

```
make -C/usr/src/linux-2.6 M='pwd'modules
```

makefile还提供另一种形式，用户可指定内核源代码所在的目录，而不用每次都把该目录作为参数传递给make命令。对于“hello world!”示例，在makefile中指定内核源代码的方式为：

```
/* 如果定义了 KERNELRELEASE 宏，则可直接通过配置内核的源代码目录来编译文件*/
ifneq($(KERNELRELEASE),)

obj -m:= hello.o
/*否则，需要从命令行获取配置内核的源代码目录信息，并编译文件*/
else

KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD:= $(shell pwd)
default:
$(MAKE) -C$(KERNELDIR) M=$(PWD) modules
endif
```

在这个makefile文件中，KERNELDIR指定内核的源代码目录，该目录通过当前运行内核使用的模块目录中的build符号链接指定。

当编译好内核模块后，用户以超级用户身份就可将内核模块加载到内核中，内核提供modutils软件包，供用户对内核模块进行管理，该软件包安装后会在/sbin目录下安装insmod、rmmod、ksyms、lsmod、modprobe等实用程序。

- insmod命令。该命令调用insmod程序，把需要加载的模块以目标代码形式加载进内核中，insmod自动调用modules_init( )函数中定义的过程运行，超级用户使用这个命令，其格式为：

```
# insmod [path]modulename
```

- rmmod命令。调用rmmod命令，将已经加载到内核的模块从内核中卸载，rmmod命令自动调用modules_exit( )函数中定义的过程运行，命令格式为：

```
# rmmod [path]modulename
```

- ksyms命令。该命令用来显示内核符号和模块符号信息。
- lsmod命令。该命令显示已经加载到内核的所有模块信息，包括被加载模块的模块名、大小和引用计数等，命令格式为：

```
# lsmod
```

对于本节中的 hello 模块，超级用户可用以下命令加载模块：

```
#insmod hello.ko
```

该模块被载入内核时，会向系统日志写入一条“Hello,world!”的消息，在执行完 insmod 命令后，用户可用 dmesg 命令查看内核系统日志，日志的最后会显示“Hello,world!”；执行完 rmmod 命令后，日志的最后会显示“Goodbye, world!”。

12.2.3.3 内核符号表

内核符号表是一个用来存放所有模块可以访问的符号及对应地址的特殊数据结构。模块的链接是将模块插入到内核的过程，模块所导出的符号都将成为内核符号表的一部分。模块根据符号表从内核空间获取主存地址，从而确保在内核空间中正确地运行，对于从模块中导出的符号，在符号表中会包含第 3 列“所属模块”，在 v2.6 内核中，用户可从/proc/kallsyms 中以文本方式读取内核符号表。

在 v2.4 内核中，默认情况下，模块中的非静态全局变量及函数在模块加载后会输出到内核符号表；而在 v2.6 内核中，默认情况下，这些符号不会被输出到内核符号表中。如果模块需要导出符号供其他模块使用，应该使用下面定义的两个宏：

```
EXPORT_SYMBOL(name)
EXPORT_SYMBOL_GPL(name)
```

它们都可以将给定的符号导出到内核符号表。EXPORT_SYMBOL_GPL 与 EXPORT_SYMBOL 的区别在于：前者把导出的符号标记为仅对遵循 GPL 许可证或其他兼容许可证的模块是可使用的。模块的许可证由 MODULE_LICENSE 宏指定，这个宏插入到模块的源代码中，其参数通常为“GPL”或“Proprietary”。为了便于维护，模块中不需要输出到内核空间，且不需模块中其他文件所用的全局变量及函数，最好显式声明为 static 类型，需要导出的内核符号以模块名为前缀。

12.2.2.4 初始化与清理函数

内核模块必须调用宏 module_init 与 module_exit 去注册初始化与清理函数。初始化函数通常定义为：

```
static int _init init_func (void)
{
    /*初始化代码*/
}
module_init(init_func);
```

从模块易于维护的角度出发，初始化函数应当声明成静态的，使它们不会在特定文件之外可见。声明中的_init 告诉内核：该函数只在初始化时使用，模块加载者在模块加载后会丢弃该函数，内核就把初始化函数占用的主存释放掉。_init 用于声明只在初始化时使用的函数，_initdata 用于声明只在初始化时使用的数据，内核可以在初始化完模块后，释放该数据占用的主存。

在模块初始化过程中，需要密切注意初始化过程中可能发生的各种错误，错误主要发生在资源的申请过程中，如分配主存失败、打开设备失败，模块代码必须检查各个函数的返回值，及时发现出现的错误保证系统的稳定性。对于一些不严重的错误，模块代码可能将模块功能减弱就能够继续运行；而对于一些模块不能处理的严重错误，模块可能就无法继续运行，此时需要把已经申请的资源都及时释放。模块与内核处于同一地址空间，模块的不稳定可能引起整个内核的不稳定。

与初始化函数相对应的是清理函数，大部分模块都需要设置一个清理函数，该函数在模块卸载时被调用；如果一个模块没有定义清理函数，内核将不会允许它被卸载，模块在清理函数中需要将已申请的资源归还给系统。清理函数的定义为：

```
static void_exit exit_func(void)
{
    /*清理代码*/
}
module_exit(exit_func);
```

清理函数没有返回值，故它被声明为 void。如果清理函数只在模块卸载时被使用，可以在函数前添加_exit 修饰符，该修饰符的作用与初始化函数中的_init 修饰符类似，都是给内核一个提示，告诉内核该函数只在模块卸载时使用。如果该模块被直接编译在内核里，或者内核被配置成不允许模块卸载，标识为_exit 的函数将直接被丢弃，因为如果一个函数被标识成_exit，该函数除在模块被卸载时，不能够在任何其他地方被调用。

12.2.2.5 模块参数

用户在加载内核模块之前，可能需要传递参数给内核模块，与内核模块进行一些简单的交互，用户和内核模块能有多种交互方式，例如通过/proc 文件系统或内核模块参数。如果用户只需要在模块加载时与模块发生交互，则采用内核模块参数方法较好。用户在 insmod、modprobe 命令中直接制定参数，命令形式为：

```
#modprobe modname var=value
```

modname 是要加载的模块名，var 是要传递的变量名，value 是传递的参数值。

如果用户每次加载模块时传递的参数都相同，每次在命令行中输入参数比较繁琐，可以在/etc/modprobe.conf 配置文件中预先写入参数，每次当模块被加载时，该配置文件中的参数就会被自动传递给模块。

模块要使用用户传递的参数，应该采用以下定义的宏：

```
module_param(name,type,perm)
module_param_array(name,type,nump,perm)
```

module_param 是对单个参数进行定义，module_param_array 是对数组型参数进行定义，在这两个宏中，name 指明参数在模块中的变量名，用户采用 name=value 形式传递参数给模块。下面介绍其他 3 个参数的含义。

（1）type

type 指明参数类型，模块直接支持的参数类型有以下几种。

- bool、invbool：bool 为布尔型，invbool 类型是颠倒的布尔类型，真值为 false，假值为 true。
- charp：字符串指针，指针指向用户传入的字符串。
- int、long、short、uint、ulong、ushort：基本变长整型值，以 u 开头的是无符号值。

以上类型可在 module_param 和 module_param_array 两个宏中直接使用。如果用户需要使用没有出现在上面列表中的类型，模块代码允许用户来使用自定义类型，读者可以查看 moduleparam.h 了解详细信息。

（2）perm

perm 为参数在 sysfs 文件系统中所对应的文件节点的属性。在 v2.6 内核中，系统使用 sysfs 文件系统，该文件系统可以动态、实时、有组织、有层次地反映当前系统中的硬件和驱动程序等情况。加载模块后，在/sys/module/目录下将出现以此模块名命名的目录，如果此模块存在 perm 不为 0 的命令行参数，则在此模块的目录下将出现 parameters 目录，包含一系列以参数名命名的文件节点，这些文件的权限值等于 perm，文件的内容为参数的值。使用 S_IRUGO 权限说明该参数对于所有用户是只读的；S_IRUGO|S_IWUSR 权限允许 root 修改参数，允许其他用户读取参数。如果一个参数在 sysfs 文件系统中被用户修改，那么模块看到该参数的值是修改后的新值。内核在参数被修改后并不会通知模块，除非模块能够自动检测这个参数的变动，在大多数情况不需要使模块参数可写。

（3）nump

nump 为一个指针。该指针所指的变量保存输入的数组元素个数，当不需保存实际输入的数组元素个数时，该指针可设置为 NULL。用户传递数组参数给模块时，使用逗号分隔输入的数组元素。设置好 nump 后，如果用户传递的数组大小大于指定的 nump，模块加载者会拒绝加载更多的数组元素。

需要注意，所有模块参数应当给定一个默认值，如果用户没有传递参数给模块，参数会使用预先设置好的默认值。

现在以本章的“hello world!”模块(称为 hellop)为例，说明模块参数的使用方法，模块使用两个参数：一个是名为 howmany 的整型值；一个是名为 whom 的字符串。当模块加载时，将对 whom 说 hello 不止一次，而是 howmany 次。用户可采用以下命令行加载：

```
#insmod hellop howmany=10 whom="Mom"
```

当模块以这样的方式加载后，模块会显示“hello,Mom”10 次。为了使用 howmany 和 whom 两个参数，在模块源代码中应该插入如下代码：

```
static char *whom="world";
static int howmany=1;
module_param(howmany, int,S_IRUGO);
module_param(whom, charp,S_IRUGO);
```

module_param 定义两个参数，一个是整型的 howmany，另一个参数是字符串类型的 whom，两个参数在 sysfs 文件系统中均是只读的。

12.2.3 内核模块机制的实现

12.2.3.1 模块在内核中的表示

内核在管理模块时使用的管理数据结构为 struct module，每一个内核模块被加载时，都要为其分配一个 module 对象，用一个双向链表把所有 module 对象组织起来，该链表的第一个元素为 modules，开发者能够通过该元素依次访问内核中所有的 module 对象。

内核通过 module 对象主要是为了记录模块的依赖，并进行模块导出符号的管理，一些重要的 module 成员在表 12-1 中列出。

表 12-1 module 结构重要成员

类 型	成 员 名	作 用
enum module_stat	stat	模块当前状态
char [60]	nam	模块名
const struct kernel_symbol *	syms	导出符号数组
unsigned int	num_syms	导出符号数组大小
const struct kernel_symbol *	gpl_syms	GPL 导出符号数组
unsigned int	num_gpl_syms	GPL 导出符号数组大小
struct module_ref	ref[NR_CPUS]	每个 CPU 上对该模块的引用计数

stat 成员表明当前模块的状态：MODULE_STATE_LIVE(处于激活状态)、MODULE_STATE_COMING(正在被初始化状态)或 MODULE_STATE_GOING(正在被卸载状态)。

每个 module 对象都包含有多个引用计数，每个 CPU 都有一个引用计数。每次当模块被使用时，模块的引用计数都会加 1；相反，当模块不被使用时，模块的引用计数就会相应减少，仅当引用计数为 0 时，该模块才能被内核卸载。例如，假设一个 MS-DOS 的文件系统被编译成为模块，当模块被加载后，模块的引用计数为 0；当用户用 mount 命令挂上一个 MS-DOS 的分区后，模块引用计数就变成 1；只有在用户使用 umount 命令后，引用计数变成 0，该模块才能被卸载。

编写模块时可使用内核导出的全局符号，也可使用其他模块导出的符号；编译模块时，这些符号在主存中的位置并不知道，只有在加载模块时，内核才能知道这些符号的具体主存地址，并把模块使用的这些符号替换成主存地址，这个过程与加载进程时的链接过程类似。

在内核代码段中，有 3 个段保存导出符号的相关信息：_kstrtab 保存导出符号的名字；_ksymtab 保存供所有模块使用的符号的地址；_ksymtab_gpl 保存仅供 GPL 协议模块使用的符号的地址。只有被 EXPORT_SYMBOL 和 EXPORT_SYMBOL_GPL 宏导出的符号才会被 C 编译

器写入内核代码的相应段中，在加载内核时，会根据代码段中的符号信息创建符号表。当前内核符号表的内容可通过/proc/kallsyms 查看：

```
c0400294 T _stext
c0400294 t run_init_process
c0400294 T stext
c04002d0 t init
c04005c7 t rest_init
c04005e8 t try_name
c0400765 T name_to_dev_t
c04009ac T calibrate_delay
c0400c40 T hard_smp_processor_id
c0400c50 t target_cpus
...
```

该文件中，第 1 列是该符号在内核地址空间中的地址；第 2 列是符号属性，小写表示局部符号，大写表示全局符号，具体含义可通过 man nm 查看；第 3 列表示符号字符串。

模块可通过 EXPORT_SYMBOL 和 EXPORT_SYMBOL_GPL 宏将符号导出，在模块的目标代码中，也有相同的 3 个段，用于存储该模块导出的符号，当加载模块时，内核会根据模块目标代码中的信息，创建相应符号表。

如果模块 A 使用模块 B 导出的符号，那么 A 和 B 之间就产生模块的依赖性，内核除记录模块的引用计数，还通过 modules_which_use_me 成员记录模块的依赖关系，modules_which_use_me 指向的是一个双向链表，该链表中的元素是 module_use 结构，该结构描述模块之间的依赖关系。

12.2.3.2 模块的加载与卸载

1. 模块的加载

用户可通过 insmod 命令将模块加载到内核，该命令的主要操作如下：

① 从命令行读入要被载入的模块名。

② 获得模块代码，它通常放在/lib/modules 目录下。

③ 调用 init_module( )函数，将包含模块代码缓存的指针、模块代码长度和用户参数传递给函数，该函数将完成模块的加载工作。

init_module( )函数的主要工作如下：

① 检查用户是否有权限加载模块(具有 CAP_SYS_MODULE 权限)。

② 在内核空间中为模块申请主存，并将模块目标代码复制到内核空间。

③ 检查模块代码是否是有效的 ELF(Executable and Linking Format，可执行连接格式)，如果不是，报错。

④ 在内核空间为用户传递的模块参数申请主存，并将参数复制到内核空间。

⑤ 内核通过模块名检查模块是否已经被加载到内核。

⑥ 为模块的可执行代码分配空间，并将模块目标代码中的相应段复制到该空间。

⑦ 为模块的初始化代码分配空间，并将模块目标代码中的相应段复制到该空间。

⑧ 获得模块的 module 对象的位置，在模块目标代码的正文段中存储着该对象。

⑨ 根据步骤⑥、⑦，初始化 module 对象中的 module_code 和 module_init 成员。

⑩ 初始化 modules_which_use_me，并把模块的引用计数置成 0。

⑪ 根据模块的授权协议，设置 license_gplok，如果该模块不符合 GPL 协议，该标志置为 0。

⑫ 根据模块和内核符号表，对模块代码进行重定位，将代码中的符号解析成主存地址。

⑬ 设置 module 对象中的 syms 和 gpl_syms 成员，这两个值被设置成模块导出符号表的地址。

⑭ 解析用户传递过来的参数，并将值赋给模块中相应的符号。

⑮ 注册 module 对象中的 mkobj 成员。注册后，在 sysfs 文件系统的 module 目录中会增加一个该模块的目录，该目录下包含有该模块的信息。

⑯ 将步骤②中申请的主存释放。

⑰ 将 module 对象加入全局的模块对象双向链表。

⑱ 将模块状态设置为 MODULE_STATE_COMING。

⑲ 如果模块自定义初始化函数，调用模块的初始化函数。

⑳ 设置模块状态为 MODULE_STATE_LIVE。

㉑ 结束。

2. 模块的卸载

用户可以通过 rmmod 命令将内核模块卸载，该命令的操作如下：

① 读取要被卸载的模块名。

② 打开/proc/modules 文件，查看该模块是否已经被卸载。

③ 调用 delete_module( )，把模块名传递给该函数，该函数将完成模块的卸载工作。

sys_delete_module( )函数的主要工作如下：

① 检查用户权限，只有具有 CAP_SYS_MODULE 权限的用户才可以卸载模块。

② 将模块名复制到内核空间。

③ 在全局模块对象的双向链表中查找到该模块的 module 对象。

④ 通过 module 对象的 modules_which_use_me 成员检查是否有其他模块依赖该模块。只有在没有依赖的情况下，函数才能继续完成卸载。

⑤ 检查模块状态，只有处于 MODULE_STATE_LIVE 状态的模块才能被卸载。

⑥ 如果该模块有自定义初始化函数，该模块只有在也定义了清理函数的情况下才允许被卸载。

⑦ 为了防止竞争，将系统中的其他 CPU 停止。

⑧ 将模块的状态设置成 MODULE_STATE_GOING。

⑨ 如果模块的引用计数大于 0，模块不能被卸载。
⑩ 如果模块定义了清理函数，调用清理函数。
⑪ 将模块加载时注册的 mkobj，取消掉。
⑫ 如果有其他模块被该模块使用，修改其他模块的引用关系。
⑬ 释放该模块的 module 对象。
⑭ 释放模块占用的主存(代码、符号表、异常表)。
⑮ 结束。

12.3 实 验 内 容

实验 通过内核模块显示进程控制块信息

1. 实验说明

在内核中，所有进程控制块都被一个双向链表连接起来，该链表中的第一个进程控制块为 init_task。编写一个内核模块，模块接收用户传递的一个参数 num，num 指定要打印的进程控制块的数量；若用户不指定 num 或者 num<0，模块则打印所有进程控制块的信息。需要打印的进程控制块信息有：进程 PID 和进程的可执行文件名。

2. 解决方案

（1）定义模块参数

该模块需要接受用户传递的参数，在使用该参数之前，需要在代码中预先定义好该参数，将该参数的类型设置为整型，并且在 sysfs 文件系统中的权限是只读的。定义的方法为：

```
static int num=-1;
module_param(num, int, S_IRUGO);
```

该参数的初始值被设置为-1。-1 将作为打印所有进程控制块的标记，默认值为-1，意味着当用户不传入任何参数时，模块将打印所有的进程的信息。

（2）访问进程控制块链表

在内核中，进程控制块被组织成多个双向链表，其中有一个双向链表包含所有的进程控制块，只需要访问该双向链表，就可以访问到所有进程控制块。Linux 内核中几乎所有双向链表都采用相同的数据结构来实现，内核中定义 list_head 通用数据结构，其定义如下：

```
struct list_head {
        struct list_head *next, *prev;
};
```

list_head 中，next 指向链表中的下一个 list_head 数据结构，prev 指向链表中的前一个 list_head 数据结构。之所以说该结构是用于实现一个通用的双向链表是因为：如果一个数据结构包含 list_head 结构，开发者就可以通过内核提供的一组宏创建并操作一个双向链表，而该链表中元素的类型为该数据结构，如图 12-1 所示。

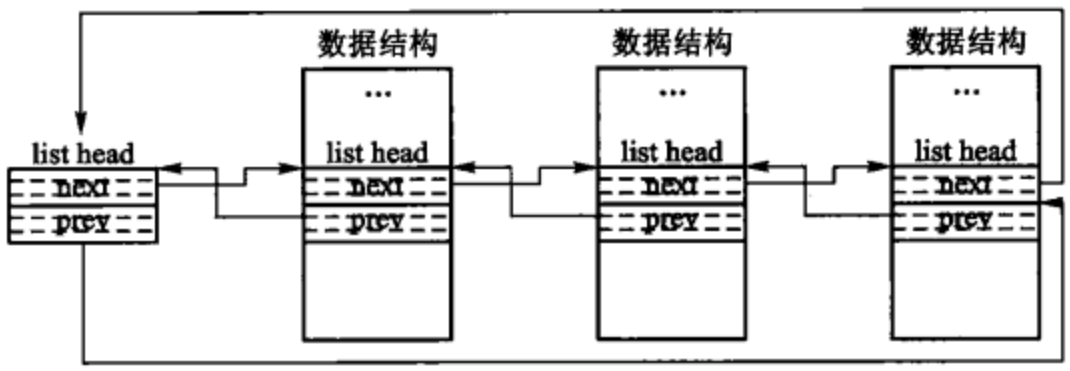


图 12-1 Linux 通用双向链表

在进程控制块 task_struct 中，包含一个名为 tasks 的成员，该成员的类型为 list_head，这意味着进程控制块能够通过该成员将进程控制块串成一个双向链表。Linux 内核通过该成员将所有的进程都放入同一个双向链表，因为 list_head 结构中的 next 和 prev 指针并不是指向包含 list_head 的数据结构，而是指向另一个 list_head 数据结构。为了访问包含 list_head 的数据结构，内核提供一个宏：

```
list_entry(ptr,type,member);
```

在该宏中，ptr 是一个指向 list_head 的指针，type 是包含 list_head 的数据结构类型，而 member 是 list_head 在该数据结构中的成员名。例如，若一个进程控制块中的 tasks 的地址为 p，为了访问该进程控制块，可以采用：

```
list_entry(p,struct task_struct,tasks);
```

该宏便会返回该进程控制块的地址。

知道如何使用双向链表后，就可以方便地访问内核中所有的进程控制块，因为它通过 tasks 成员串成一个双向链表，如果得到一个进程控制块的地址 p，开发者可以通过：

```
list_entry(p->tasks.next, struct task_struct, tasks);
```

访问该双向链表中的下一个进程控制块。在该双向链表中，第一个进程控制块为 init_task，如果开发者发现下一个进程控制块为 init_task 时，说明已经完整地遍历过所有进程控制块。

内核定义宏 for_each_process 用于遍历所有的进程控制块，开发者通过该宏就能将所有的进程控制块访问一遍，该宏展开的形式为：

```
for (p = &init_task ; (p = list_entry((p)->tasks.next, struct task_struct, tasks) != &init_task ; )
```

（3）输出进程控制块信息

进程控制块中包含进程大部分信息，根据实验要求，模块需要打印进程的 pid 和可执行文件名，在进程控制块的数据结构中，成员 pid 为进程的 PID，而成员 comm 包含进程的可执行文件名。在内核中，模块可以通过 printk( )内核函数将这些信息打印到系统日志中。

3. 程序框架

```
#include <linux/init.h>
#include <linux/module.h>
```

```
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

MODULE_LICENSE("GPL");

static int num= –1;
module_param(num, int, S_IRUGO);

/*模块初始化函数 */
static _init int exp_init( )
{
    struct task_struct *p = NULL;

    for_each_process(p){
        if ( num==0 )
            break;
        printk("pid=%d, path=%s\n", p->pid, p->comm );

        num--;
     }
    return 0;
}

/*模块清理函数*/
static _exit void exp_exit( )
{
   …
}

/*注册模块初始化与清理函数*/
module_init(exp_init);
module_exit(exp_exit);
```

第 13 章　中断与系统调用

13.1　实 验 目 的

- 加深对中断机制和系统调用原理的理解。
- 深入了解系统调用的执行流程。
- 学会增加系统调用及向内核添加内核函数。

13.2　背 景 知 识

13.2.1　中断机制

13.2.1.1　中断及其分类

系统调用的实现依赖于中断机制，本节简单介绍与中断相关的概念。中断机制是现代计算机系统的重要组成部分之一，每当应用程序执行系统调用要求获得操作系统服务、I/O 通道及设备报告传输情况、或产生形形色色的内部和外部事件时，都需要通过中断机制产生中断信号，启动内核工作。所以，可以说操作系统是由“中断驱动”的。中断是指程序执行过程中，遇到急需处理的事件时，暂时中止 CPU 上现行程序的运行，转去执行相应的事件处理程序，待处理完成后再返回原程序被中断处或调度其他程序执行的过程。

按中断事件的来源和实现手段可把中断源分成：硬中断和软中断。硬中断又分外中断和内中断。外中断又称中断或异步中断，是指来自 CPU 之外的中断信号，外中断还可分屏蔽中断和不可屏蔽中断，不同中断具有不同的中断优先级，表示事件的紧急程度，在处理高级中断时，往往会屏蔽部分或全部低级中断。内中断又称异常或同步中断，是指来自 CPU 内部，在程序执行中，发现的与当前指令关联的、不正常的或错误的事件。Linux 中的异常可被细分为 4 种：故障(fault)、陷阱(trap)、终止(abort)和编程异常(programmed excption)。编程异常用来实现系统调用，进程自愿进入内核态以请求系统服务；故障指程序运行中系统捕获出现的潜在可恢复错误，处理后可返回到当前指令再次执行，页面故障是典型的例子；终止是发现致命的不可恢复错误，通常不会返回原程序而转向内核特殊函数处理；陷阱是执行特定调试指令触发的，被调试的进程遇到所设置的断点，会暂停等待。所有这些事件均由异常处理程序处理，异常处理用来响应 CPU 中状态的变化且一般依赖于执行程序的当前现场，不能被屏蔽掉，一旦出现应立即

响应并进行处理。

中断是由与当前执行程序无关的中断信号触发的，系统不能确定中断事件发生的时间，所以，它与 CPU 是异步的，CPU 对中断的响应完全是被动的；中断的发生与 CPU 模式无关，既可发生在用户态，也可发生在内核态，通常在两条机器指令之间才可以响应中断。一般来说，中断处理程序提供的服务不是当前进程所需要的，而且它在系统的中断上下文中执行。异常是由 CPU 控制单元产生的，它的发出缘于现行程序在执行指令过程中检测到例外，它与 CPU 是同步的，一条指令执行期间允许响应异常，而且允许多次响应异常，大部分异常发生在用户态。通常，异常处理程序提供的服务是为当前进程所需要的，如处理程序出错或页面故障，异常处理程序一般在当前进程的上下文中执行。此外，允许中断的嵌套发生，但异常多数情况为一重，异常处理过程中可能产生中断，但反之则不会发生。

中断和异常要通过硬件设施来产生中断请求，这些都是硬中断，与其相对应的不必由硬件产生中断源而能引发的一种中断称为软中断，软中断是利用硬中断的概念，用软件方法对中断机制进行模拟，实现宏观上的异步执行效果。软中断分两种："信号"是一种软中断机制，信号的发送者相当于中断源，而信号的接收者必为一个进程 (相当于 CPU)；"软件中断"是另一种软中断机制，它的典型应用例子是 Linux 中的 bottom half，而下半部分的升级就是 Linux 中复杂庞大的软中断子系统 softirq 机制。

上述几种中断的通常用法如下：

- "中断"(硬中断)用于外部设备对 CPU 的中断(中断的是正在运行的任何程序)，转向中断处理程序上半部分执行。
- "异常"(硬中断)因指令执行不正常而中断 CPU(中断的是正在执行这条指令的程序)，转向异常处理程序。
- "软件中断"(软中断)用于硬中断服务程序对内核的中断，在上半部分中发出软件中断(即标记 bottom half)，使得中断下半部分在适当时刻获得处理。
- "信号"(软中断)用于内核或进程对某个进程的中断，通知进程某个特定事件发生或迫使进程执行信号处理程序。

操作系统中，除一部分低层硬件异常由内核异常处理程序处理外，大部分异常均转化为信号(软中断)再进行处理，由于异常与当前运行进程紧密相关，每当执行指令产生异常事件，可通过信号处理程序向当前运行进程发出一个信号。

13.2.1.2 中断和中断处理程序

1. 上半部分和下半部分

中断通常由硬件设备引发，与异常的处理有一定差异，在响应一个特定中断的时候，内核会执行中断处理程序(或称中断服务例程)。产生中断的每个设备都有一个相应的中断处理程序，例如，时钟中断处理程序、键盘中断处理程序。一个设备的中断处理程序是它的设备驱动程序的一部分，而设备驱动程序是用于对设备进行管理的内核代码。

在 Linux 中，中断处理程序必须按照特定的类型声明，以便内核能够以标准方式传递处理程序的信息，中断处理程序与其他内核函数的真正区别在于：中断处理程序是被内核调用来响应中断的，它们运行于称为中断上下文的特殊上下文中。

中断可能随时发生，因此中断处理程序也就随时可能执行，必须让中断处理程序能够快速执行，这样才能保证尽可能快地恢复被中断代码的执行。为此把中断处理程序分成上半部分(top half)和下半部分(bottom half)，上半部分接收到一个中断后，它就立即开始执行，只做有严格时间限制的工作，如对接收的中断进行响应或复位硬件，这些工作都是在所有中断被禁止的情况下完成的；能够被允许稍后完成的工作会推迟到下半部分中去，在合适的时机，下半部会被开中断执行。

2. 注册和释放中断处理程序

中断处理程序是管理硬件的驱动程序的组成部分，每类设备都有对应的设备驱动程序，如果设备使用中断，那么相应的驱动程序就注册一个中断处理程序，下面的函数用于注册并激活一个中断处理程序，以便处理中断：

```
int request_irq(unsigned int irq,
irqreturn_t( *handler)(int, void *, struct pt_regs*),
unsigned long irqflags,
const char *devname,
void *dev_id)
```

该函数申请分配一条给定的中断线，参数 irq 表示要分配的中断号；参数 handler 是一个指针，指向处理这个中断的实际中断处理程序，只要系统接收到中断，就调用该函数；参数 irqflags 可以为 0，也可能是多个标志的位掩码；参数 dev_id 用于共享中断线。

卸载驱动程序时，需要注销相应中断处理程序，并释放中断线。可以调用 void free_irq(unsigned int irq.void *dev_id)来释放中断线。如果指定的中断线不是共享的，那么，该函数删除中断处理程序的同时将禁用这条中断线；如果中断线是共享的，则仅删除 dev_id 所对应的中断处理程序，而这条中断线本身只有在删除了最后一个中断处理程序时才会被禁用。

如何编写中断处理程序呢?以下是一个典型的中断处理程序声明：

```
static irqreturn_t intr_handler(int irq,void *dev_id,struct pt_regs *regs)
```

它的类型与 request_irq( )参数中 handler 所要求的参数类型相匹配。第 1 个参数 irq 就是该中断处理程序要响应的中断的中断号；第 2 个参数 dev_id 是一个通用指针，它与在中断处理程序注册时传递给 request_irq( )的参数 dev_id 必须一致；第 3 个参数 regs 是一个指向结构的指针，该结构包含处理中断之前处理器的寄存器和状态，主要用于调试。中断处理程序的返回值是一个特殊类型 irqreturn_t，可能返回两个特殊的值：IRQ_NONE 和 IRQ_HANDLED。当中断处理程序检测到一个中断，但该中断对应的设备并不是在注册处理函数期间指定的产生源时，返回 IRQ_NONE；当中断处理程序被正确调用，且确实是它所对应的设备产生了中断时，返回 IRQ_HANDLED。

3. 中断上下文

曾经讨论过进程上下文，这是内核可能处于的一种操作模式，此时内核代表进程工作，例如执行系统调用。在进程上下文中，可以通过 current 宏关联当前进程。此外，因为进程是以进程上下文形式连接到内核中的，因此，在进程上下文中内核可以睡眠，也可以调用调度程序。

执行中断处理程序或下半部时，内核处于中断上下文模式，中断上下文和进程并没有关系，与 current 宏也不相干，尽管它会指向被中断的进程。既然没有进程的背景，中断上下文中却不可以阻塞，因此，不能从中断上下文中调用某些会发生阻塞的函数，这是对什么样的函数可以在中断处理程序中使用的一项限制。

中断上下文具有较为严格的时间限制，因为它会打断其他代码，中断上下文中的代码应当能快速执行，尽量不去处理繁重的工作。有一点非常重要：中断处理程序可能打断其他代码，甚至可能打断另一个中断处理程序的执行，正是因为这种异步执行特性，内核尽量把一些工作从中断处理程序中分离出来，放在下半部分来执行，因为下半部可以被延时运行或在运行时被打断。

中断处理程序堆栈的设置是一个配置选项，以前的中断处理程序不具有自己的堆栈，相反它共享被中断进程的内核栈。内核栈的大小是两页(8 KB)，因为在这种设置中，中断处理程序共享别人的堆栈，所以应该非常节省地使用栈空间。在 v2.6 早期内核中，增加了选项，把堆栈的大小可从两页减少到一页(4 KB)。这就减轻了主存的压力，因为系统中每个进程原先都需要两页不可换出的内核主存空间。为了应对堆栈大小的减少，中断处理程序拥有了自己的堆栈，每个 CPU 一个，大小为一页，这个栈就称为中断堆栈，尽管中断堆栈的大小是原先共享栈的一半，但平均可用堆栈空间大得多，因为中断处理程序把这一整页拥为己有。

4. 中断和异常的一般处理过程

Linux 中断机制在保护模式下的实现采用中断描述符表 IDT(Interrupt Descriptor Table)，该表包含 256 个中断描述符，每个中断或异常对应一个。描述符的作用是把程序控制转给中断/异常服务程序，每个均为 8 B，通过它就能找到中断/异常服务程序的起始地址、属性及程序特权级等信息。IDT 的位置由硬件中断描述符表寄存器 IDTR 指定，它是个 48 位的寄存器，高 32 位是 IDT 的基地址，低 16 位限定 IDT 的长度。当产生中断时，CPU 响应中断请求后，硬件将自动清除中断允许位，以禁止其他硬件中断；但对于异常，启动异常处理程序后系统并不关中断，允许响应中断并暂时中止异常处理。

每个中断/异常都有一个向量号，该号的值在 0～255 之间，该值是中断在 IDT 表中的索引，每个中断/异常均有其相应处理程序，中断在使用前，必须在 IDT 中注册以保证发生中断时能找到相应处理程序。IDT 在系统初始化时创建，向量号的使用情况为：0～31 号对应异常或硬件非屏蔽中断，32～47 号分配给可屏蔽硬件中断，48～255 号中断向量分配给软件中断，其中，128 号(0x80)用来实现系统调用。

中断处理的执行流程大致如下：

① 当设备发出中断请求时，中断信号由设备发送到中断控制器，中断控制器根据 IRQ 号

转换成中断量号传给 CPU；CPU 响应中断后，自动保护现场信息，把 PSW(EFLAGS、CS 和 EIP)、中断向量号、用户栈段寄存器 SS 和栈指针 ESP 压入内核栈，并根据向量号查 IDT，生成指向中断描述符表的指针，找到对应中断服务程序 IRQn_interrupt 的地址，中断处理程序开始执行。

② 进入中断公共代码段 common_interrupt 处执行 SAVE_ALL，保护所有硬件未保护的寄存器内容到内核栈。

③ 调用 do_IRQ( )函数，对中断控制器进行确认，设置中断源状态等。

④ 根据 IRQ 为发出中断请求的设备提供服务，调用服务程序执行相关中断处理任务，标记中断下半部分；如果有多个设备共享该中断 IRQ，需执行该中断线上所有设备的中断服务程序。

⑤ 检查 softirq_active 和 softirq_mask 是否有标记的下半部分，有就调用 do_softirq( )执行之。

⑥ 跳转到 ret_from_intr 退出，恢复中断前的现场。

异常处理分 3 步：一是当前进程执行指令产生异常；二是进入并执行异常处理程序；三是从异常处理程序返回。当异常产生后，将触发 0～31 号对应的异常，并自动转向异常处理程序公共入口执行，执行下列操作：

① 将硬件错误码和异常向量号存入当前进程 PCB 中。

② 判别异常产生于内核态还是用户态，对于前者，将简单地转向内核预定义服务程序处理，没有处理的内核态异常是操作系统的致命错误。

③ 对于用户态异常，终止当前进程运行，调用 force_sig( )函数给当前进程发信号。

④ 从 ret_from_exception 处返回用户空间时，将会检查进程是否有信号等待处理，如果有则根据信号类型调用相应函数进行处理。

当然，也有些异常处理很特殊，如页面故障异常，它是实现分页虚存管理的硬件支撑，由系统来处理这种异常，异常处理程序最终会转向 do_page_fault( )执行页面调度。

中断和异常处理过程大致相同，但产生异常时，硬件并不清除中断标志位，此时还允许外部硬件中断；而产生中断时，硬件立即清除中断标志位，以禁止其他硬件中断。

5. Linux 中断处理机制

当中断事件发生时，运行进程执行完当前指令后，CPU 响应中断，转入相应中断服务程序 ISR 处理，区分两类中断事件：快中断和慢中断。两者主要区别为：处理慢中断前需保存所有寄存器的内容，而处理快中断仅要保存那些被常规 C 函数修改的寄存器；在处理慢中断时，通常不屏蔽其他中断信号，而处理快中断时会屏蔽所有其他中断。

慢中断要做很多工作，如果全部在屏蔽中断下来完成，会影响及时响应处理期间到达的其他中断信号，于是，如前面已经讨论过的，Linux 中断服务程序被分成上半部分(top half)和下半部分(bottom half)两个处理函数。上半部分函数每当接收到中断，立即开始工作，在关中断状态下只做严格限时且与硬件相关的任务，然后“标记中断”，通知下半部分做剩余工作，这种部分工作在关中断状态下处理，另外的工作由可中断代码来处理，称为中断下半部分处理，可见这

是一种任务延迟处理机制。

和上半部分只能通过中断处理程序实现不同，Linux 中断处理程序下半部分可用多种机制来实现，不同机制分别由不同子系统与接口组成，它们是 bottom half、task queue、tasklet、work queue 和 softirq。

1) bottom half(下半部分)

Linux 最早提供的任务延迟执行机制称为 bottom half(称 BH)，用于实现下半部分中断处理程序，提供静态创建的下半部分处理数据结构，建立一个函数指针数组，采用数组索引访问，最多有 32 个不同下半部分处理函数。BH 机制有两个局限性：一是 BH 限制为 32 个，且每个 BH 上只能挂接一个函数，随着系统设备越来越多，BH 应用范围越来越广，这个数目不够用；二是每个 BH 在全局范围内同步，即使属于不同 CPU，也不允许任何两个 BH 同时执行，这种机制使用方便但不灵活，安全简单但有性能瓶颈，所以在开发 v2.5 内核版本时，抛弃了 BH 接口。

2) task queue(任务队列)

进一步改进是引入任务队列机制，来实现对各种任务的延迟执行，为此内核定义一组队列，每个队列包含一个由等待调用的函数组成的链表，不同队列中的函数会在某个时刻被触发执行，可用它来替代 BH，当驱动程序或内核有关部分要将任务排队进行延时处理时，可将任务添加到相应任务队列，然后，用适当方式通知内核执行任务队列函数。

典型的下半部分处理均有任务队列相关联，如定时器队列、即时队列等。由于任务队列的灵活性差，无法代替整个 BH 接口，也不能胜任像网络等性能要求较高的子系统，该接口已经从 v2.5 内核版本中除去。

3) tasklet(小任务)

Linux 2.3 版本后，有两种机制可用来实现任务延迟执行：tasklet 和 softirq。tasklet 也是一种下半部分机制，能更好支持 SMP，它基于软中断来实现，但比软中断接口简单，锁保护要求低；softirq 保留给执行频率及时间要求特高的下半部分使用(如网络和 SCSI)，多数场合下可使用 tasklet。

因为 BH 全局串行处理，不适应 SMP 环境，引入 tasklet 后，不同 tasklet 可同时运行于不同 CPU 上，当然，系统保证相同 tasklet 不会同时在不同 CPU 上运行，在这种情形下，tasklet 就不需要是可重入的。在新版 Linux 中，tasklet 是建议的异步任务延迟执行机制。

4) work queue(工作队列)

Linux 2.5 内核引入另外一种任务延迟执行机制——工作队列，它与 tasklet 的工作原理不相同，tasklet 在软中断上下文中运行，所有 tasklet 代码必须是原子的；而 work queue 把一个任务延迟，并交给内核线程去完成，且该任务总是在进程上下文中执行，故有更多的灵活性。这样，通过 work queue 执行的代码能占尽进程上下文的优势，最重要的是工作队列允许重新调度及阻塞。如果延迟执行的任务需要阻塞，需要获取信号量或需要获得大量主存时，那么，可选择 work queue，否则可使用 tasklet 或 softirq。

5) softirq(软中断)

迄今，Linux 沿用最早的 BH 思想，但在该机制上已实现了庞大和复杂的软中断子系统 softirq，它是一种软中断机制，又是一个框架，最多可注册 32 个软中断，目前版本已预定义 6 个，包括 tasklet、定时器下半部分、发送网络数据包、接收网络数据包、SCSI 下半部分等。softirq 保留给对时间要求严格的下半部分使用，其中，网络和 SCSI 属于这种情况，它们直接使用软中断。

13.2.2 系统调用的概念

操作系统为应用程序提供的与内核进行交互的一组接口称为系统调用。通过此接口，用户态应用程序被切换到内核态，由对应该系统调用的内核函数代表该进程运行，从而可以访问系统的各种资源，实现应用程序与内核的交互。系统服务之所以需要通过系统调用来提供用户空间的根本原因是为了对系统进行“保护”，因为 Linux 的运行空间分为内核空间与用户空间，它们各自运行在不同级别中，逻辑上相互隔离。通常不允许应用进程访问内核数据，也无法使用内核函数，它们只能在用户空间操作用户数据，调用用户空间函数。很多情况下，应用进程需要获得操作系统服务，这时就必须利用系统提供的“特殊接口”——系统调用，其特殊性主要在于规定了应用进程进入内核的具体位置，即用户访问内核的路径是事先规定好的，而不允许肆意进入，有了这样的陷入内核的统一访问路径限制才能保证内核安全无虞，图 13-1 是两个空间与模式切换示意。

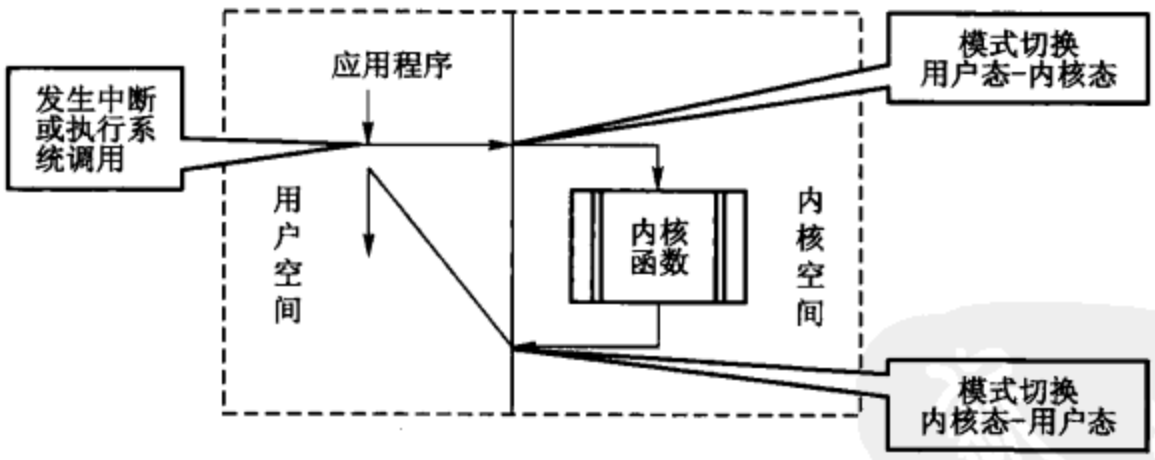


图 13-1 两个空间与模式切换

什么功能应该在内核空间而不是在用户空间实现？这个问题并没有明确的答案，有些服务可以选择在内核空间完成，但也可以在用户空间完成。如果选择前者，通常基于以下考虑：该项服务必须获得内核数据，如获得中断或系统时间等；从安全角度考虑，在内核中提供的服务比用户空间提供的安全性会更好，因为它不能被非法访问；从效率角度出发，在内核空间实现系统服务能避免和用户空间来回传递数据，效率往往比在用户空间实现高许多。

程序员或系统管理员通常并非直接和系统调用打交道，在实际使用中，程序员调用的是

函数，或称为应用程序接口(Application Programm Interface，API)，管理员使用的则是更高层次的系统命令。操作系统为每个系统调用在标准 C 函数库中构造一个具有相同名字的封装函数(wrapper function)，由它来屏蔽下层的复杂性，负责把操作系统提供的服务接口——系统调用——封装成应用程序能够直接调用的函数(库函数)。

系统调用通过中断机制向内核提交请求，它的功能由内核函数实现，进入内核后不同系统调用找到各自对应的内核函数，这些内核函数就是系统调用的“服务例程”。API 实质上是一个函数定义，说明如何获得一个给定服务，如 read( )、malloc( )、free( )等。它有可能和系统调用形式上一致，如 read( )函数就和 read( )系统调用对应，但未必总是一一对应，往往会出现不同函数的内部用到同一个内核函数，如 malloc( )、free( )内部均利用 sys_brk( )内核函数来扩大或缩小进程堆；一个函数也可利用几个内核函数组合完成服务，有些函数不需要任何内核函数，它的实现与内核无关。系统命令相对编程接口有更高的层次，它们是内部引用函数的可执行程序，如常用的系统命令 ls、hostname 等，而这些命令的实现大多数依靠系统调用。

下面以 read( )为例来了解系统调用的执行流程，当应用程序调用 read(fd,buffer,nbytes)函数时，该函数在 Linux/GNU 提供的标准 C 库，即 libc 中，对应的封装函数由下面汇编指令实现：

```
movel   $3 , %eax
movel   fd, %ebx
movel   buffer, %ecx
movel   nbytes, %edx
int   $0x80
```

在 Linux 中规定 int $0x80 指令是系统调用的总入口，若干个寄存器中存放应用程序传递给内核的参数。内核为了区分不同的系统调用，需要给每个都分配唯一的系统调用号，而相对应的内核函数的入口地址都放在系统调用表中。内核在获得系统调用号后根据寄存器 EAX 中调用号的值跳转到系统调用表相应的内核函数，以完成应用程序请求的服务。系统调用处理程序 system_call( )的入口地址放在系统的中断描述符表 IDT 中。当 Linux 系统在初始化时，由 trap_init( )将该描述符表填写完整，其设置系统调用处理程序的语句为：

```
set_system_gate(0x80,&system_call)
```

经过初始化以后，当用户调用“int $0x80”指令便会跳转到 system_call( )。

13.2.3 系统调用的执行流程

系统调用与普通 C 语言函数调用最大的区别在于系统调用需要陷入内核态，发生特权级的转换。

图 13-2 是 Linux 系统调用执行流程：首先应用程序准备参数并发出一个函数请求，它由标准 C 库的封装函数来引导；接着把封装函数包含的宏展开成含有 int $0x80 汇编指令的汇编代码段，int $0x80 通过陷阱处理机制使该系统调用进入内核的入口点 system_call( )；最后由 system_call( )找到指定内核函数，该内核函数被执行并返回结果。

Linux 操作系统提供 200 多个系统调用，系统调用的执行比 C 语言函数复杂得多。entry.S 文件中的 system_call( )是所有系统调用的公共入口，其功能是保护现场，根据系统调用号跳转

到具体的内核函数，该内核函数执行完毕时需调用 ret_from_sys_call( )，完成返回用户空间前的最后检查，检查通过后系统将恢复现场并返回到用户空间。system_call( )的简要处理流程如图 13-3 所示。

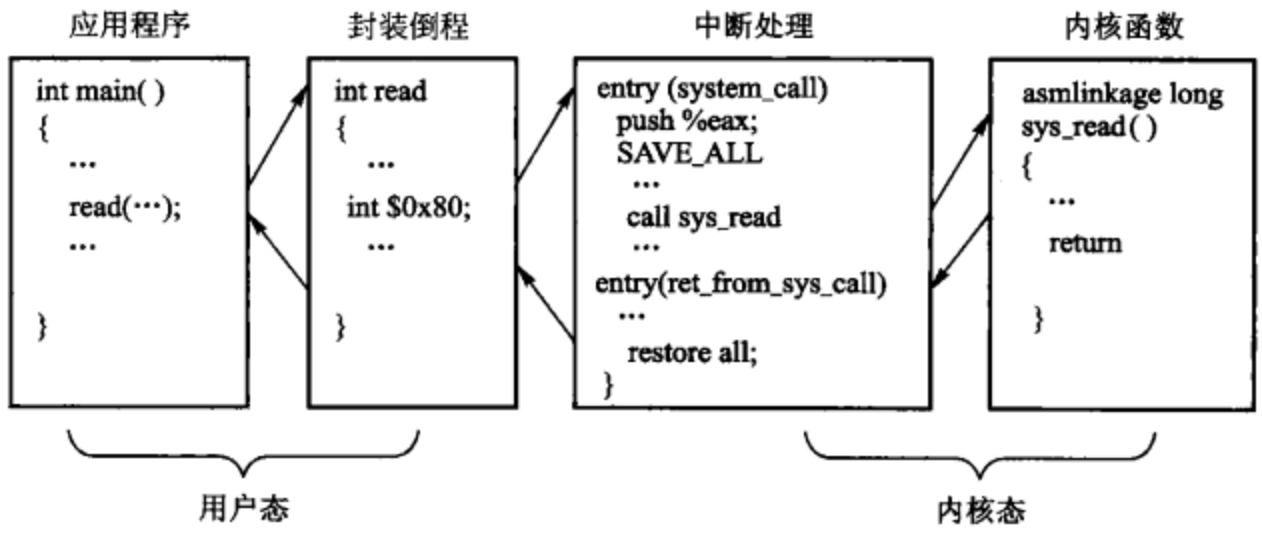


图 13-2　Linux 系统调用执行流程

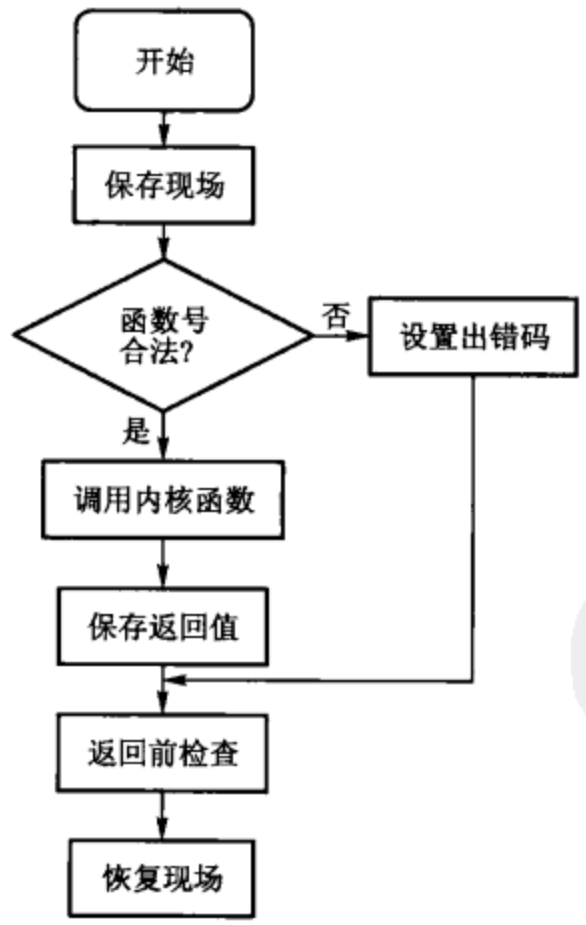


图 13-3　系统调用处理流程

13.2.3.1　保存现场

每个系统调用都对应一个完成其功能的内核函数，为了使得该内核函数能够方便使用寄存

器，system_call( )在转跳到具体的内核函数之前需要对所有寄存器先进行保存。用户在执行陷阱指令后，CPU 硬件自动地依次将 SS、ESP、EFLAGS、CS 和 EIP 寄存器原有内容压入进程内核栈。所以在 system_call( )中，只需将余下的寄存器压入内核栈即可。压入过程对应 SAVE_ALL 宏：

```
#define SAVE_ALL
        cld;
        pushl %es;
        pushl %ds;
        pushl %eax;
        pushl %ebp;
        pushl %edi;
        pushl %esi;
        pushl %edx;
        pushl %ecx;
        pushl %ebx;
        movl $(_KERNEL_DS),%edx;
        movl %edx,%ds;
        movl %edx,%es;
```

SAVE_ALL 宏将寄存器压入内核栈，加载内核的 DS 和 ES，往 EDX 寄存器中放入 $(_KERNEL_DS)以指明使用内核数据段，把内核数据段选择符装入 DS 和 ES 段寄存器。该宏压入寄存器的顺序并不是随意的，该顺序和系统调用的参数传递密切相关。

13.2.3.2 跳转到内核函数

每个系统调用对应一个系统调用号，系统调用的数量由 NR_syscalls 给定。每个系统调用所对应的编号在 unistd.h 中定义，且都用一个宏表示，对于名为 xyz( )的系统调用，其相应的系统调用号所对应的宏为_NR_xyz,其定义有如下的形式：

```
#define _NR_exit        1
#define _NR_fork        2
#define _NR_read        3
#define _NR_write       4
#define _NR_open        5
…
#的 fine_NR_xyz         i
 …
```

当用户请求一个系统调用时，必须将请求的系统调用号写入 EAX 寄存器，system_call( )将根据系统调用号跳转到对应的内核函数。Linux 的系统调用号和内核函数映射关系的系统调用表也被定义在 entry.S 文件中，具有如下的形式：

```
.data
ENTRY(sys_call_table)                    /*入口*/
```

```
.long SYMBOL_NAME(sys_ni_syscall)      /*0、空项*/
.long SYMBOL_NAME(sys_exit)            /*1*/
.long SYMBOL_NAME(sys_fork)            /*2*/
.long SYMBOL_NAME(sys_read)            /*3*/
.long SYMBOL_NAME(sys_write)           /*4*/
.long SYMBOL_NAME(sys_open)            /*5*/
.long SYMBOL_NAME(sys_close)           /*6*/
  …
.long SYMBOL_NAME(sys_ni_syscall)      /*保留*/
.rept NR_syscalls-(.-sys_call_table)/4
  .long SYMBOL_NAME(sys_ni_syscall)    /*空项*/
.endr                                  /*结束*/
```

倒数第 3 项中，“.”代表当前地址，sys_call_table 代表数组首地址，相减便得到相差的字节数，即系统调用表的大小，再除以 4，得到目前已定义的系统调用个数。用 NR_syscalls 减去系统调用个数，能计算出尚未定义的系统调用个数，故可用空项.long SYMBOL_NAME(sys_ni_syscall)去填充系统调用表。

由此可以确定系统调用号与内核函数的关系是：系统调用号就是系统调用表中各表项的相对偏移量，即系统调用的处理函数 system_call( )执行时，根据其传入 EAX 寄存器中的系统调用号，就可以定位该系统调用的内核函数在系统调用表 sys_call_table 中的准确位置，从而确定对应内核函数 sys_name 的入口地址。

表中第 1 项定义 exit 系统调用的内核函数 sys_exit( )入口地址，表中第 2 项定义 fork( )系统调用的内核函数 sys_fork( )入口地址，以此类推，而 sys_ni_syscall 表示空表项，即该系统调用号还没有被使用。有了上述数据结构，system_call( )就能很容易按系统调用号跳转到对应的内核函数，具体的转跳语句为：

```
call *SYMBOL_NAME(sys_call_table)(,%eax,4)
```

该语句计算出 eax 值所对应内核函数的入口地址(eax*4+sys_call_table)，并用 call 语句转跳并执行。

13.2.3.3 参数传递

有些系统调用不需要传递给内核任何参数，如 getpid( )系统调用；而有些系统调用需要应用进程向内核传递参数，如 exit( )系统调用。

普通 C 函数调用采用堆栈传递参数的方法，这在系统调用中行不通，因为涉及用户栈到内核栈的转换，用户态进程无法往内核栈写入数据。虽然内核态程序可以从用户栈中读取数据，但这样做效率太低，较好的方法是通过寄存器传递参数。应用进程在调用系统调用前，向约定寄存器写入参数，system_call( )运行时，再将寄存器中的参数写入内核栈。将内核栈设置好后，系统调用对应的内核函数就能和普通 C 函数一样处理，都从各自的堆栈中获得参数。

Linux 把系统调用号装入 EAX 寄存器，传递给内核函数的其他参数最多为 5 个，依次存放在寄存器 EBX、ECX、EDX、ESI 及 EDI 中。超过 6 个参数的情况不多见，可通过寄存器存放指向参数在用户空间的地址指针来传递，然后执行中断指令 int $0x80 陷入内核态。返回值则放入 EAX 中。

system_call( )的开头便用 SAVE_ALL 把所有的寄存器都保存在内核栈里，这样内核函数便能够通过内核栈访问应用进程提供的参数。在这之前提到 SAVE_ALL 保存寄存器的顺序并不是任意的，当 SAVE_ALL 宏执行完毕之后，从栈顶往下依次为 EBX、ECX、EDX、ESI 和 EDI。这与传递参数使用的寄存器的顺序一致，因为只有 SAVE_ALL 以这样的顺序将寄存器压入堆栈才能满足 C 函数的调用约定。

参数传递还涉及另一个问题，即如何将系统调用的返回值返回给应用程序。根据 C 函数的调用约定，整型的返回值应存放在 EAX 寄存器中，因此当内核函数返回到 system_call( )时，EAX 中存放着内核函数的返回值。要将这个返回值传递给应用进程，内核先将 EAX 放入原先 SAVE_ALL 宏保存 EAX 的位置，其代码是：

```
movl %eax,EAX(%esp)
```

这样当 system_call( )调用 RESTORE_ALL 恢复寄存器时，EAX 便被恢复成系统调用的返回值，完成了返回值从内核空间到用户空间的传递。

13.2.3.4 系统调用封装

每个系统调用的执行流程都类似，主要步骤为：

① 将系统调用号写入 EAX。

② 将参数写入相应寄存器。

③ 调用 int $0x80。

④ 错误检查。

为了简化系统调用的封装，Linux 定义了 7 个宏：

- _syscall0(type,name)
- _syscall1(type,name,type1,arg1)
- _syscall2(type,name,type1,arg1,type2,arg2)
- _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3)
- _syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4)
- _syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5)
- _syscall6(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4,type5,arg5,type6,arg6)

这组宏能够自动将系统调用封装成 C 库函数供应用程序调用，每个宏名字中的数字表示系统调用所需要传递参数的个数。通过这些宏，用户就可以给操作系统添加新的系统调用。以 exit( )系统调用为例，该系统调用的封装可用如下的语句完成：

```
_syscall1(int, exit, int, error_code)
```

该宏在内核中可以自动展开成：

```
int exit(int error_code){
    long _res;
    asm("int $0x80": "=a" (_res)
                     : "0" (_NR_exit), "b" ((long)error_code));
    …
    return _res;
}
```

该段函数中嵌入的汇编语句的含义是：

- 将_NR_exit(系统调用号)放入 EAX。
- 将参数 error_code 放入 EBX。
- 调用 int $0x80 进入系统调用的内核函数。
- 将系统调用的返回值写入_res。

通过这种方式，exit( )系统调用便被封装好了，在用户态下调用系统调用就变得很简单，既不需要关心系统调用号，也不需要提供复杂的参数，方便地让内核为自己提供服务。虽然系统调用一般在应用程序中使用，但在内核中同样可调用封装后的系统调用，只不过二者有一些区别：在用户态执行系统调用时，转换到内核态后，系统调用处理程序要进行用户栈到内核栈的切换，即从 int $0x80 指令转换到内核态的 system call( )时，要保存寄存器 SS 和 ESP 的值，而当 ret 指令从 system call( )返回用户态时要取回 SS 和 ESP 的值；在内核中执行系统调用时，不用进行堆栈切换，因为这时始终在内核态工作。

应用程序发出的系统调用是同步事件，虽然它在内核态执行，但却在调用进程的上下文中，所以既可以访问进程地址空间，也可以访问内核空间。system_call( )在内核栈上保存硬件上下文，然后使用系统调用号作为系统调用表 system_call_table 的索引，以确定 system_call( )应该执行哪个系统调用对应的内核函数。当该系统调用执行结束后，system_call( )在对应的寄存器 EAX 中放入返回的 int 值或错误码，并从内核心栈恢复硬件上下文，回到用户态并把控制权交给库函数，库函数再返回给应用程序。

13.2.3.5 系统调用上下文

内核在执行系统调用时处于进程上下文中，current 指针指向当前进程，即引发系统调用的进程。在进程上下文中，内核可以阻塞或被抢占。首先，能够阻塞说明系统调用可以使用内核提供的绝大部分功能，从而给内核编程带来方便；其次，在进程上下文中能够被抢占，其实表明像用户空间内的进程一样，当前进程同样可以被其他进程抢占，当然因为新进程可以使用相同的系统调用，所以必须保证该系统调用是可重入的。当系统调用返回时，控制权仍然在 system_call( )中，它最终会负责切换到用户空间并让应用进程继续执行。

13.2.4 新系统调用机制 sysenter/sysexit

13.2.4.1 sysenter/sysexit 系统调用机制

在 Linux 2.4 内核中，系统调用是通过中断指令 int $0x80 实现的。在 x86 保护模式中，CPU 处理 int 时，首先从中断描述表 IDT 取出对应的描述符，判断描述符的种类，然后检查描述符的级别 DPL 和 int 指令调用者的级别 CPL，当 CPL<=DPL，即 int 调用者级别高于描述符指定级别时，才能成功调用，最后再根据描述符内容，进行压栈、跳转、权限级别提升。内核代码执行完毕后，调用 iret 指令返回，该指令恢复用户栈，并跳转回低级别的代码。

其实，在发出系统调用，由特权级 3 进入特权级 0 的过程会浪费 CPU 周期，系统调用必然需要由特权级 3 进入特权级 0(由内核调用 int 指令的方式除外)，权限提升之前和之后的级别是固定的，CPL 肯定是 3，而 int $0x80 的 DPL 肯定也是 3，这样 CPU 检查描述符的 DPL 和调用者的 CPL 就完全没必要。所以，x86 从 PII300(Family6、Model3、Stepping3)之后，开始支持新的系统调用指令 sysenter/sysexit。sysenter 指令用于由特权级 3 进入特权级 0，sysexit 指令用于由特权级 0 返回特权级 3。由于没有特权级别检查处理，也没有压栈操作，执行速度比 int $0x80/iret 快了不少。对 sysenter/sysexit 指令的支持最早在 2002 年，由 Linus Torvalds 编写并首次加入 v2.5 版内核中，经过多方测试和多次修补，最终正式加入到 v2.6 版内核中。

sysenter 指令可以在 3、2、1 这 3 个特权级别调用(Linux 中只用特权级 3)，而 sysexit 指令只能从特权级 0 调用。sysenter/sysexit 和 int $0x80/iret 指令的一个区别在于：sysenter/sysexit 指令并不成对，sysenter 指令并不会把 sysexit 所需的返回地址压栈，sysexit 返回的地址并不一定是 sysenter 指令的下一个指令地址。调用 sysenter/sysexit 指令地址的跳转是通过设置一组特殊寄存器实现的，这些寄存器包括：

- SYSENTER_CS_MSR：用于指定要执行的特权级 0 代码的代码段选择符，由它还能得出目标特权级 0 所用堆栈段的段选择符。
- SYSENTER_EIP_MSR：用于指定要执行的特权级 0 代码的起始地址。
- SYSENTER_ESP_MSR：用于指定要执行的特权级 0 代码所使用的栈指针。

这些寄存器可以通过 wrmsr 指令来设置，执行该指令时，通过寄存器 EDX、EAX 指定设置的值，EDX 指定值的高 32 位，EAX 指定值的低 32 位，在设置上述寄存器时，EDX 都是 0，通过寄存器 ECX 指定填充的 MSR 寄存器，SYSERTER_CS_MSR、SYSENTER_ESP_MSR、SYSENTER_EIP_MSR 寄存器分别对应 0x174、0x175、0x176。需要注意，wrmsr 指令只能在特权级 0 执行。

在特权级 3 中的代码调用 sysenter 指令之后，CPU 会执行如下操作：

① 将 SYSENTER_CS_MSR 的值加载到 CS 寄存器。

② 将 SYSENTER_EIP_MSR 的值加载到 EIP 寄存器。

③ 将 SYSENTER_CS_MSR 的值加 8(特权级 0 的堆栈段描述符)加载到 SS 寄存器。

④ 将 SYSENTER_ESP_MSR 的值加载到 ESP 寄存器。
⑤ 将特权级切换到特权级 0。
⑥ 如果 EFLAGES 寄存器的 VM 标志被置位，则清除该标志。
⑦ 开始执行指定的特权级 0 代码。

在特权级 0 代码执行完毕，调用 sysexit 指令退回特权级 3 时，CPU 会执行如下操作：

① 将 SYSENTER_CS_MSR 的值加 16(特权级 3 的代码段描述符)加载到 CS 寄存器。
② 将寄存器 EDX 的值加载到 EIP 寄存器。
③ 将 SYSENTER_CS_MSR 的值加 24(特权级 3 的堆栈段描述符)加载到 SS 寄存器。
④ 将寄存器 ECX 的值加载到 ESP 寄存器。
⑤ 将特权级切换到特权级 3。
⑥ 继续执行特权级 3 的代码。

由此可知，在调用 sysenter 进入特权级 0 之前，一定需要通过 wrmsr 指令设置好特权级 0 代码的相关信息，在调用 sysexit 之前，还要保证寄存器 EDX 和 ECX 的正确性。

13.2.4.2 Linux 对 sysenter/sysexit 系统调用方式的支持

系统调用均被封装成库函数提供给应用程序使用，应用程序调用库函数后，由 glibc 库负责执行 int 指令进入内核调用相应内核函数。在 v2.4 内核加上老版 glibc 的情况下，库函数所做的就是通过 int 指令来完成系统调用，而内核提供的接口很简单，只要在 IDT 中提供 int $0x80 的入口，库函数就可以完成中断调用。

在 v2.6 内核中，内核代码同时包含对 int $0x80 中断方式和 sysenter 指令方式调用的支持，因此内核会给用户空间提供一段入口代码，内核启动时根据 CPU 类型，决定这段代码采取哪种系统调用方式。对于 glibc 来说，无需考虑系统调用方式，直接调用这段入口代码，即可完成系统调用。这样做还可以尽量减少对 glibc 的改动，在 glibc 的源代码中，只需将“int $0x80”指令替换成“call 入口地址”即可。

前面说到的这段入口代码，根据调用方式分为两个文件，支持 sysenter 指令的代码包含在文件 arch/i386/kernel/vsyscall-sysenter.S 中，支持 int 中断的代码包含在 arch/i386/kernel/vsyscall-int80.S 中，入口名都是_kernel_vsyscall，这两个文件编译出来的二进制代码由 arch/i386/kernel/vsyscall.S 所包含，并导出起始地址和结束地址。

v2.6 内核在启动时，调用新增加的内核函数 sysenter_setup( )，在这个函数中，内核将虚拟主存空间的顶端一个固定地址页面(从 0xffffe000 开始到 0xffffefff 的 4 KB 大小)映射到一个空闲的物理页框。然后通过之前执行 CPU ID 指令得到的数据，检测 CPU 是否支持 sysenter/sysexit 指令。如果 CPU 不支持，那么将采用 int 调用方式的入口代码复制到该页框中，然后返回。相反，如果 CPU 支持 sysenter/sysexit 指令，则将采用 sysenter 调用方式的入口代码复制到该页框中。使用宏 on_each_cpu 在每个 CPU 上执行 enable_sep_cpu( )。

在 enable_sep_cpu( )中，内核将当前 CPU 的 TSS 结构中的 SS1 设置为当前内核使用的代码

段，ESP1 设置为该 TSS 结构中保留的一个 256 B 大小的堆栈。在 x86 中，TSS 结构中 SS1 和 ESP1 本来是用于保存特权级 1 进程的堆栈段和堆栈指针的。由于内核在启动时，并不能预知调用 sysenter 指令进入特权级 0 后 ESP 的确切值，而应用程序又无权调用 wrmsr 指令动态设置，所以此时就借用 ESP1 指向一个固定的缓冲区来填充这个 MSR 寄存器，由于特权级 1 根本没被启用，所以并不会对系统造成任何影响。在下面的介绍中会涉及进入特权级 0 之后，内核如何修复 ESP 来指向正确的特权级 0 堆栈。关于 TSS 结构更细节的应用可参考代码 include/asm-i386/processor.h。

然后，内核通过 wrmsr(msr,val1,val2)宏调用 wrmsr 指令对当前 CPU 设置 MSR 寄存器，可以看出调用宏的第 3 个参数(即 EDX)都被设置为 0。其中 SYSENTER_CS_MSR 的值被设置为当前内核用的所在代码段；SYSENTER_ESP_MSR 被设置为 ESP1，即指向当前 CPU 的 TSS 结构中的堆栈；SYSENTER_EIP_MSR 则被设置为内核中处理 sysenter 指令的接口函数 sysenter_entry。这样，sysenter 指令的准备工作就完成。

为了配合内核使用新的系统调用方式，glibc 中要做一定的修改，新的 glibc-2.3.2(及其以后版本)中已经包含这个改动。在 glibc 源代码的 sysdeps/UNIX/sysv/Linux/i386/sysdep.h 文件中，处理系统调用的宏 INTERNAL_SYSCALL 在不同的编译选项下有不同结果。在打开支持 sysenter/sysexit 指令的选项 I386_USE_SYSENTER 下，系统调用会有两种方式，在静态链接(编译时加上-static 选项)情况下，采用 call *_dl_sysinfo 指令；在动态链接情况下，采用 call *%gs:0x10 指令。这两种情况不论由 glibc 库采用哪种方法链接，实际上最终都相当于调用某个固定地址的代码。下面通过一个小程序，配合 gdb 来验证。

首先是一段静态编译的程序，代码很简单：

```
int main( )
{
    getuid( );
    return 0;
}
```

将代码加上 static 选项用 gcc 静态编译，然后用 gdb 装入并反编译 main( )函数。

```
[root@test opt]# gcc test.c -o ./static -static
[root@test opt]# gdb ./static
(gdb) disassemble main
0x08048204 <main+0>:     push   %ebp
0x08048205 <main+1>:     mov    %esp,%ebp
0x08048207 <main+3>:     sub    $0x8,%esp
0x0804820a <main+6>:     and    $0xfffffff0,%esp
0x0804820d <main+9>:     mov    $0x0,%eax
0x08048212 <main+14>:    sub    %eax,%esp
0x08048214 <main+16>:    call   0x804cb20 <_getuid>
0x08048219 <main+21>:    leave
```

```
0x0804821a <main+22>:    ret
```

可以看出，main( )中调用_getuid( )函数，接着反编译_getuid( )函数。

```
(gdb) disassemble 0x804cb20
0x0804cb20 <_getuid+0>:          push    %ebp
0x0804cb21 <_getuid+1>:          mov     0x80aa028,%eax
0x0804cb26 <_getuid+6>:          mov     %esp,%ebp
0x0804cb28 <_getuid+8>:          test    %eax,%eax
0x0804cb2a <_getuid+10>:         jle     0x804cb40 <_getuid+32>
0x0804cb2c <_getuid+12>:         mov     $0x18,%eax
0x0804cb31 <_getuid+17>:         call    *0x80aa054
0x0804cb37 <_getuid+23>:         pop     %ebp
0x0804cb38 <_getuid+24>:         ret
```

上面只是_getuid( )函数的一部分。可以看到_getuid( )函数将 EAX 寄存器赋值为 getuid 系统调用的功能号 0x18，然后调用另一个函数，这个函数的入口在哪里呢？接着查看位于地址 0x80aa054 的值。

```
(gdb) X 0x80aa054
0x80aa054 <_dl_sysinfo>:         0x0804d7f6
```

_dl_sysinfo 指针的指向的地址就是 0x80aa054。现在尝试着启动这个程序，然后停在程序第 1 条语句，再查看这个地方的值。

```
(gdb) b main
Breakpoint 1 at 0x804820a
(gdb) r
Starting program: /opt/static
Breakpoint 1, 0x0804820a in main ( )
(gdb) X 0x80aa054
0x80aa054 <_dl_sysinfo>:     0xffffe400
```

可以看到，_dl_sysinfo 指针指向的数值已经发生变化，指向 0xffffe400，如果继续运行程序，_getuid( )将会调用地址 0xffffe400 处的代码。

接下来，将上面的代码编译成动态链接方式，即默认方式，用 gdb 装入并反编译 main( )函数。

```
[root@test opt]# gcc test.c -o ./dynamic
[root@test opt]# gdb ./dynamic
(gdb) disassemble main
0x08048204 <main+0>:     push    %ebp
0x08048205 <main+1>:     mov      %esp,%ebp
0x08048207 <main+3>:     sub     $0x8,%esp
0x0804820a <main+6>:     and     $0xfffffff0,%esp
0x0804820d <main+9>:     mov      $0x0,%eax
0x08048212 <main+14>:    sub     %eax,%esp
```

```
0x08048214 <main+16>:    call    0x8048288
0x08048219 <main+21>:    leave
0x0804821a <main+22>:    ret
```

由于 libc 库是在程序初始化时才被加载，所以先启动程序，并停在 main 第一条语句，然后反汇编 getuid 库函数。

```
(gdb) b main
Breakpoint 1 at 0x804820a
(gdb) r
Starting program: /opt/dynamic
Breakpoint 1, 0x0804820a in main ( )
(gdb) disassemble getuid
Dump of assembler code for function getuid:
0x40219e50 <_getuid+0>:          push   %ebp
0x40219e51 <_getuid+1>:          mov     %esp,%ebp
0x40219e53 <_getuid+3>:          push   %ebx
0x40219e54 <_getuid+4>:          call    0x40219e59 <_getuid+9>
0x40219e59 <_getuid+9>:          pop     %ebx
0x40219e5a <_getuid+10>:         add     $0x84b0f,%ebx
0x40219e60 <_getuid+16>:         mov    0xffffd87c(%ebx),%eax
0x40219e66 <_getuid+22>:         test    %eax,%eax
0x40219e68 <_getuid+24>:         jle     0x40219e80 <_getuid+48>
0x40219e6a <_getuid+26>:         mov     $0x18,%eax
0x40219e6f <_getuid+31>:         call    *%gs:0x10
0x40219e76 <_getuid+38>:         pop     %ebx
0x40219e77 <_getuid+39>:         pop     %ebp
0x40219e78 <_getuid+40>:         ret
```

可以看出，函数 getuid( )将 EAX 寄存器设置为 getuid 系统调用的调用号 0x18，然后调用%gs:0x10 所指向的函数。在 gdb 中，无法查看非 DS 段的数据内容，所以无法查看%gs:0x10 所保存的实际数值，不过可以通过编程方法，内嵌汇编语句将%gs:0x10 的值赋予某个局部变量来得到这个数值，而这个数值也是 0xffffe400，具体代码这里就不再赘述。

由此可见，无论是静态还是动态方式，最终都到达 0xffffe400 的一段代码，这里就是内核提供的映射系统调用入口代码。在 gdb 中，可以直接反汇编来查看这里的代码。

```
(gdb) disassemble 0xffffe400 0xffffe414
Dump of assembler code from 0xffffe400 to 0xffffe414:0xffffe400:      push    %ecx
0xffffe401:      push    %edx
0xffffe402:      push    %ebp
0xffffe403:      mov      %esp,%ebp
0xffffe405:      sysenter
0xffffe407:      nop
```

```
0xffffe408:     nop
0xffffe409:     nop
0xffffe40a:     nop
0xffffe40b:     nop
0xffffe40c:     nop
0xffffe40d:     nop
0xffffe40e:     jmp    0xffffe403
0xffffe410:     pop    %ebp
0xffffe411:     pop    %edx
0xffffe412:     pop    %ecx
0xffffe413:     ret
End of assembler dump.
```

这段代码正是 arch/i386/kernel/vsyscall-sysenter.S 文件中的代码。其中，在 sysenter 之前的是入口代码，在 0xffffe410 开始的是内核返回处理代码(后面提到的 SYSENTER_RETURN 即指向这里)。在入口代码中，首先是保存当前的 ECX、EDX(由于 sysexit 指令需要使用这两个寄存器)及 EBP。然后调用 sysenter 指令，跳转到内核特权级 0 代码，即 sysenter_entry 入口处。

sysenter_entry 整个的实现可以参见 arch/i386/kernel/entry.S。内核处理 sysenter 的代码和处理 int 的代码不太一样。通过 sysenter 指令进入特权级 0 之后，由于当前的 ESP 并非指向正确的内核栈，而是当前 CPU 的 TSS 结构中的一个缓冲区，所以首先要解决的是修复 ESP，幸运的是，TSS 结构中 ESP0 成员本身就保存有特权级 0 状态的 ESP 值，所以在这里将 TSS 结构中 ESP0 的值赋予 ESP 寄存器。将 ESP 恢复成指向正确的堆栈之后，由于 sysenter 不是通过调用门进入特权级 0，所以在堆栈中的上下文和使用 int 指令的不一样，int 指令进入特权级 0 后堆栈中会保存如下值：

- 返回用户态的 EIP。
- 用户态的 CS。
- 用户态的 EFLAGS。
- 用户态的 ESP。
- 用户态的 SS(和 DS 相同)。

因此，为了简化和重用代码，内核会用 pushl 指令往堆栈中放入上述各值，值得注意的是，内核在堆栈中放入的对应用户态 EIP 的值，是一个代码标签 SYSENTER_RETURN，在 vsyscall-sysenter.S 可以看到，它就在 sysenter 指令的后面。接下来，处理系统调用的代码就和中断方式的处理代码一模一样，内核保存所有寄存器，然后由系统调用表找到对应内核函数的入口，完成调用。最后，内核从内核栈中取出前面保存的用户态的 EIP 和 ESP，存入 EDX 和 ECX 寄存器，调用 sysexit 指令返回用户态。返回用户态之后，从堆栈中取出 ESP、EDX 和 ECX，最终返回 glibc 库。

13.3 实 验 内 容

实验 记录系统调用的使用次数

1. 实验说明

本实验要求通过修改 system_call( )，使内核能够记录每个系统调用被使用的次数。同时为了使应用进程能够查询到这些数据，本实验要求提供一个系统调用供应用进程来查询某个特定系统调用被使用的次数。

2. 解决方案

（1）处理步骤

在内核中添加新的系统调用步骤如下：

① 编写系统调用的内核函数。根据需要编写一个系统调用的内核函数，它将要被加到内核中去，因此该内核函数在内核中的标准名称应该是在函数名前面加上“sys_”标志。

② 连接新的系统调用。先将新系统调用的源代码添加到/usr/Linux/kernel/sys.c 文件中，再通知内核的其余部分：内核增加了新的系统调用的内核函数，需要重新编辑/usr/src/Linux/include/asm/unistd.h 和/usr/src/Linux/arch/asm/kernel/entry.S 两个文件。

③ 编译新的 Linux 内核，获得内核映像文件 bzImage。

④ 启动新内核的操作系统。

⑤ 尝试使用新系统调用。用户使用新的系统调用之前，因为在已有的标准 C 库中没有包含新系统调用的封装函数，应该先用 Linux 提供的预处理宏指令 syscalln( )对该系统调用进行封装。

（2）数据结构

为了能够记录每个系统调用的使用次数，需要有一个大小为 NR_syscalls 的整型数组。该数组的第 n 项代表系统调用号为 n 的系统调用被使用的次数，可以采用类似定义 sys_call_table 的方法在 entry.S 汇编文件中定义该整型数组。但是更为简单的方法是在 C 语言文件中定义该数组，并将该数组导出为全局符号，这样 entry.S 就能够方便地访问该数组。假如在(arch/i386/kernel/sys_i386.c)中添加一个数组 sys_call_ctable，用于记录各个系统调用被调用的次数，代码如下：

```
int sys_call_ctable[NR_syscalls];
EXPORT_SYMBOL(sys_call_ctable);      /*全局符号*/
```

（3）修改 system_call( )

在 system_call( )中，必须在调用内核函数之前增加系统调用的使用计数，增加计数只需获得该系统调用所对应的计数在 sys_call_ctable 中的位置便可。由于一个计数占用 4 个字节，所以系统调用所对应计数的位置为 sys_call_ctable+eax*4。具体的代码如下：

```
incl sys_call_ctable(,%eax,4) ;                /*在转入服务例程前增加计数*/
call *sys_call_table(,%eax,4);                 /*转入服务例程*/
```

（4）定义系统调用

为了查询系统调用被使用的次数，需要增加一个新的系统调用，添加系统调用的第 1 步是定义系统调用。将该系统调用定义为：

```
int syscall_audit(int syscallno);          /*syscallno 为被查询的系统调用号*/
```

定义系统调用后，系统调用号所对应的宏和内核函数的定义也都被确定。系统调用号为__NR_syscall_audit，服务的内核函数的定义为：

```
asmlinkage int sys_syscall_audit(int syscallno);
```

（5）添加系统调用号

系统调用号是在 include/asm-i386/unistd.h 文件中定义的，那么在该文件中添加一个名称为__NR_syscall_audit 的系统调用号，并需要增加 NR_syscalls 的计数。在 Linux 2.6.18 中，修改如下：

```
#define __NR_restart_syscall          0
#define __NR_exit                     1
...
#define __NR_tee                      315
#define __NR_vmsplice                 316
#define __NR_move_pages               317
#define __NR_syscall_audit            318
#define NR_syscalls                   319
```

只有在定义好系统调用号后，system_call( )才能根据系统调用号跳转到相应的内核函数，用于封装系统调用的宏才能正确地展开。

（6）修改 sys_call_table

下一步是添加 sys_call_table 中的相应表项，添加系统调用号与内核函数的对应关系。在 Linux 2.6.18 中，sys_call_table 被放入单独的 arch/i386/kernel/syscall_table.S 文件中，只要在该文件中添加相应的对应即可，修改如下：

```
ENTRY(sys_call_table)
    .long sys_restart_syscall
    .long sys_exit
    ...
    .long sys_move_pages
    .long sys_syscall_audit
```

（7）添加内核函数

添加系统调用已经到最后一步，把内核函数添加到内核。该内核函数定义中的 asmlinkage 表示它将从堆栈获得参数，其原因已在前面有过阐述，现在可在 arch/i386/kernel/sys_i386.c 文件中添加该内核函数，它的结构如下：

```
asmlinkage int sys_query_syscall(int syscallno) {
        ...
        return sys_call_ctable[syscallno];
}
```

至此，已经完成添加新系统调用任务，只要重新编译内核代码，再引导新内核之后，新的系统调用便可以使用。

3. 程序框架

由于该系统调用是新添加到内核中的，标准的 C 库中并没有相应的库函数定义，为了使用该系统调用，需要使用在系统调用一节所提到的宏。在测试程序中，必须引用<Linux/unistd.h>头文件，并用_syscall1 展开新定义的系统调用。这两个步骤结束后，新的系统调用就可以在测试程序中使用，测试程序结构如下：

```
#include <linux/unistd.h>
_syscall1(int, syscall_audit, int, syscallno)
int main( )
{
        /*执行程序其他部分*/
        /*调用新添加的系统调用*/
        printf("syscall_audit ( ) has been called %d times.",
                syscall_audit(_NR_ syscall_audit) );
        /*执行程序剩余部分*/
}
```

因为大部分系统调用的调用次数都不受用户控制，因此用户可以通过测试新添加的系统调用的使用次数来验证程序的正确性。

第14章 同步机制

14.1 实验目的

- 深入理解程序并发执行的本质以及进程间的互斥和同步概念。
- 了解 Linux 内核并发性和各种同步机制。
- 了解信号量实现中涉及的关键技术：进程状态转换和等待队列。
- 学会向内核添加新的同步机制。

14.2 背景知识

14.2.1 进程同步和同步机制

14.2.1.1 进程互斥和同步的概念

操作系统中引入并发程序设计技术，并发执行的进程由于共享系统资源和协同完成任务需要交互，从而产生进程间的相互依赖及相互制约关系。并发进程的交互如果不加以控制，可能出现与时间有关的错误和不正确的运算结果。为了确保系统正常运转，必须提供某种机制来解决并发进程之间的制约和依赖，即互斥和同步关系，本章对进程间通信机制中的一种——信号量机制进行实验。

并发进程之间的交互存在两种基本关系：竞争关系和协作关系。进程之间的一般交互关系是互斥关系，这是由于计算机中的资源有限，众多进程需要共享资源，当两个进程访问同一个独占型资源时，一个进程向操作系统提出资源申请请求，另外一个进程只能等待资源被释放后再申请资源。在竞争关系中，由于进程间共享资源而产生制约关系，这是间接制约关系，又称互斥关系。互斥机制是解决进程间竞争关系的手段，进程使用互斥机制向系统提出资源申请，谁先向系统提出申请，谁就能先执行。

同步是并发进程之间共同完成一项任务时直接发生的制约关系，又称协作关系。具有协作关系的进程在执行的时间次序上必须遵循确定规律，需要相互协作的进程在某些关键点上协调各自的工作。当其中的一个到达关键点后，在尚未得到其协作进程发来的消息或信号之前应阻塞自己，等待协作进程发来信号或消息后方被唤醒并继续执行。同步机制是解决进程间同步关系的手段，让并发进程基于某个条件来协调它们的活动，通过合适的方法排定执行的先后次序。

进程互斥关系是一种特殊的进程同步关系，即逐次使用互斥共享资源，也是对进程使用资源次序上的一种协调。信号量机制是最常用的一种进程同步机制，它能够很好地解决系统态或用户态进程间的竞争关系和协作关系，即同步与互斥关系。

这里再通过一个例子进一步阐明什么是进程的同步和同步机制。著名的生产者—消费者问题(Producer Consumer Problem)是计算机操作系统中并发进程内在关系的一种抽象，是典型的进程同步问题，在操作系统中，生产者进程可以是计算进程、发送进程；而消费者进程可以是打印进程、接收进程等。解决好生产者—消费者问题就解决了一类并发进程的同步问题。生产者—消费者问题表述如下：有 n 个生产者和 m 个消费者，连接在一个有 k 个单位缓冲区的有界环形缓冲上，故又叫有界缓冲问题。其中，pi 和 cj 都是并发进程，只要缓冲区未满，生产者 pi 生产的产品就可投入缓冲区；类似地，只要缓冲区不空，消费者进程 cj 就可从缓冲区取走并消耗产品。

但是如果缓冲区满了，生产者进程必须等待；如果缓冲区空了，消费者进程必须等待，否则便会出现不正确的结果或产生系统死锁。导致这类情况不是因为并发进程共享缓冲区，而是因为它们访问缓冲区的速率不匹配，或者说 pi 和 cj 的相对速度不协调，需要调整并发进程的推进速度，并发进程间的这种制约关系称进程同步，交互的并发进程之间通过交换信号或消息来达到调整相互速率，保证进程协调运行的目的。操作系统实现进程同步的机制称为同步机制，它通常由同步原语组成，不同的同步机制采用不同的同步方法，迄今已设计出许多种同步机制，最常用的同步机制有：信号量及 PV 操作和消息传递等。

14.2.1.2 信号量和 PV 操作

信号量由荷兰计算机科学家 Edsger.Dijkstra 在 1965 年提出，并发进程通过信号量展开交互，一个进程在某一关键点上被迫停止执行直到发现信号量值被改变，通过这一设施，任何复杂的进程交互要求可得到满足。信号量实际上是一个整型数，进程在信号量上的操作仅有两种，一种称为 P 操作，一种称为 V 操作。在 Linux 中，P 操作也被称为 Down 操作，V 操作也被称为 Up 操作。Down 操作让信号量的值减 1，Up 操作让信号量的值加 1。在进行 Down 操作时，进程会先检测信号量的值。如果值大于 0，则进程将信号量的值减 1，否则进程进入该信号量等待队列进行睡眠。进程在进行 Up 操作时，信号量的值被加 1，如果此时信号量大于 0，睡眠在该信号量等待队列上的进程会被唤醒。

通过信号量，进程能够实现进程间的互斥与同步，信号量的值通常用来表示系统中可用资源的数量，进程通过 Down 操作来获取资源，通过 Up 操作来释放资源。当系统资源不够时，得不到资源的进程会进入等待队列去睡眠，以等待其他进程释放资源后将其唤醒。进程通过对信号量的操作实现进程之间对资源的互斥访问和对同一任务的协同工作。

信号量在 Linux 内核中的实现中涉及两个关键技术：进程状态转换和等待队列。

14.2.1.3 进程状态转换

进程是一个程序的执行过程，它的动态性质是由其状态变化决定的，例如，当进程使用信

号量申请资源时，如果所需资源得不到满足，它就需要放弃 CPU，而进入等待状态，这充分说明进程的状态是会发生变化的，状态及其转换体现了进程的动态性。为了便于系统管理，一般来说，按进程在执行过程中的不同情况至少要定义 3 种不同的进程状态。

- 运行(running)态：进程占有 CPU 正在运行的状态。
- 就绪(ready)态：进程具备运行条件，等待系统分配 CPU 以便运行的状态。
- 等待(wait)态：又称阻塞(blocked)态或睡眠(sleep)态，指进程不具备运行条件，正在等待某个事件完成的状态。例如，当进程通过信号量得不到需要的资源时，便转变成等待态，等待其他进程释放资源。

进程在它的生命周期中不断地在这些状态之间进行转换。

在 Linux 操作系统中，进程在 3 态模型基础上进行扩展，就绪态和运行态被合并成一个状态——可运行态，内核将处于可运行态的进程放入同一队列，调度程序从该队列中选取一个进程占有 CPU 运行。当进程需要的资源得不到满足时，便转变为等待态，Linux 将等待态细分为两种，于是不同等待状态的进程放入不同的等待队列加以管理。除此之外，还为进程添加了暂停态和僵死态。

- 可运行态(TASK_RUNNING)：正在运行或准备运行，处于该状态的所有进程组成可运行队列。
- 可中断睡眠态(TASK_INTERRUPTIBLE)：进程正在睡眠，等待资源可用时被唤醒，也可以由其他进程通过信号或时钟中断唤醒后，进入运行队列。
- 不可中断睡眠态(TASK_UNINTERRUPTIBLE)：与可中断睡眠态类似，但是有一个例外，不能由其他进程通过信号或时钟中断唤醒。
- 暂停态(TASK_STOPPED)：进程暂时停止执行来接受某种状态。例如，当进程接受到 SIGSTOP、SIGTSTP、SIGTTIN 和 SIGTTOU 等信号时，进程进入该状态。
- 僵死态(TASK_ZOMBIE)：进程执行结束但尚未消亡的状态。此时，进程已经运行结束并释放大部分资源，但尚未释放进程控制块。这个状使其他进程可以在进程消亡之前获得该进程的一些信息。

14.2.1.4 等待队列

内核在管理进程时，需要把处于不同状态的进程进行分类组织。处于可运行态的进程被组织在一起。暂停态和僵死态的进程不链接在专门的链表中。等待态的进程再分成很多类，每一类对应一个特定的事件。处于同一类等待态的进程通过等待队列链接在一起。等待队列实现了在事件上的条件等待：希望等待特定事件的进程把自己放进合适的等待队列，并放弃 CPU。等待队列表示一组睡眠的进程，当某一条件变成真时，由内核唤醒睡眠的进程。

1. 数据结构

等待队列由循环链表实现，其元素包括指向进程描述符的指针。每一个等待队列都有一个等待队列头(wait queue head)，等待队列头是一个类型为 wait_queue_head_t 的数据结构，内核中

定义在<linux/wait.h>文件中：

```
struct _wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
typedef struct _wait_queue_head wait_queue_head_t;
```

在该数据结构中，lock 成员为自旋锁，它保证多个 CPU 不会同时修改等待队列；task_list 是用于实现双向链表的数据结构。开发者可以通过 DECLARE_WAIT_QUEUE_HEAD 静态定义并初始化一个 wait_queue_head_t，也可以使用 init_waitqueue_head( )动态地初始化。

等待队列中的每个元素为 wait_queue_t 类型，该数据结构也定义在<linux/wait.h>文件中：

```
struct _wait_queue {
    unsigned int flags;
    #define WQ_FLAG_EXCLUSIVE   0x01
    void *private;
    wait_queue_func_t func;
    struct list_head task_list;
};
typedef struct _wait_queue wait_queue_t;
```

等待队列链表中的每个元素代表一个睡眠进程，该进程等待某一事件发生，其描述符地址存放在 private 成员中。在同一等待队列中的所有进程都在等待同一个事件的发生，当该事件发生时，内核唤醒所有进程是合理的。但是在有些情况下，如果多个进程等待的资源只能互斥地被访问，唤醒所有进程会浪费系统资源；因为只有一个进程能够获得资源，被唤醒的其他进程仍必须进入睡眠。wait_queue_t 数据结构中的 flag 用于区分这两种情况，当 flag 设置为 1 时，内核有选择性地唤醒进程；如果 flag 设置为 0，内核会将所有进程唤醒。func 成员指定进程被唤醒后执行的函数。开发者可以使用 DECLARE_WAIT_QUEUE 静态定义并初始化一个 wait_queue_t，也可以使用 init_waitqueue_entry( )来初始化。默认的初始化将 flag 设为 0，private 设为要睡眠的进程的进程描述符，而 func 则初始化为函数 default_wake_func( )的指针，该函数只是调用 try_to_wake_up( )来唤醒等待队列项对应的进程。

等待队列中的元素还可以采用 DEFINE_WAIT 宏进行声明。该宏将 private 设置为当前进程，将 func 设置成 autoremove_wake_function( )。autoremove_wake_function( )将调用 default_wake_function( )，并将该等待队列元素从等待队列中删除。开发者在定义好等待队列元素后，可以使用内核提供的函数直接对等待队列进行操作。

- add_wait_queue( )：将一个非互斥进程插入等待队列。
- add_wait_queue_exclusive( )：将一个互斥进程插入等待队列。
- remove_wait_queue( )：将一个进程从等待队列中删除。
- waitqueue_active( )：检查等待队列是否为空。

2. 睡眠进程

内核提供一组函数让进程可以睡眠到某一个事件发生，这些函数使得开发者不需要直接对等待队列进程插入、删除操作，方便开发者使用。

- sleep_on( )：该函数对当前进程起作用，并需要使用一个等待队列头作为参数。该函数在内核中的代码为：

```
void sleep_on(wait_queue_head_t *wq)
{
    wait_queue_t wait;
    init_waitqueue_entry(&wait, current);
    current->state = TASK_UNINTERRUPTIBLE;
    add_wait_queue(wq,&wait);
    schedule( );
    remove_wait_queue(wq, &wait);
}
```

该函数将当前进程的状态设置成 TASK_UNINTERRUPTIBLE，并将该进程插入等待队列。然后，它调用调度程序，调度程序开始执行其他可运行程序。当该进程被唤醒时，该进程将从等待队列中被删除。

- interruptible_sleep_on( )：该函数与 sleep_on( )类似，但是该函数将进程的状态设置成 TASK_INTERRUPTIBLE。处于这个状态的进程除了在等待条件满足时会被唤醒，也会被发送到该进程的信号或时钟中断唤醒。

- sleep_on_timeout( )和 interruptible_sleep_on_timeout( )：这两个函数与 sleep_on( )和 interruptible_sleep_on( )类似，但它们允许调用者设置一个时间间隔，过了这个时间间隔之后，进程将被内核唤醒。为了做到这一点，它们调用的是 schedule_timeout( )而不是 schedule( )。

Linux 2.6 中引入 prepare_to_wait( )、prepare_to_wait_exclusive( )和 finish_wait( )函数。程序员通过这几个函数也可以使进程睡眠，它们的通常的用法是：

```
DEFINE_WAIT(wait);
prepare_to_wait_exclusive(&wq, &wait, TASK_INTERRUPTIBLE);
...
if (!condition)
    schedule( );
finish_wait(&wq, &wait);
```

prepare_to_wait( )和 prepare_to_wait_exclusive( )函数均通过函数的第3个参数设置进程是否可被信号唤醒，并将进程插入等待队列。这两个函数的差异在于 prepare_to_wait( )将进程设置成非互斥的，prepare_to_wait_exclusive( )将进程设置成互斥的。在进程被唤醒后，finish_wait( )函数会将进程状态重新设置成 TASK_RUNNING，并将进程从等待队列中移出。

从 Linux 2.4 开始，引入 wait_event 和 wait_event_interruptible 宏。这两个宏将进程置于等待队列中睡眠，直到给定的条件满足。wait_event_interruptible(wq,condition)实际产生的代码与

如下类似：

```
DEFINE_WAIT(_ _wait);
for (;;) {
    prepare_to_wait(&wq, &_ _wait, TASK_UNINTERRUPTIBLE);
    if (condition)
        break;
    schedule( );
}
finish_wait(&wq, &_ _wait);
```

这两个宏应该用来代替以前的 sleep_on( )和 interruptible_sleep_on( )，因为以前的函数在条件未被验证时不能测试条件，也不能原子地将进程置于睡眠，因此是一个著名的竞争条件源。采用这两个宏将进程置于睡眠时，进程都是非互斥的。要把进程互斥地插入等待队列，需要使用 prepare_to_wait_excluse( )函数或 add_wait_queue_excluse( )函数直接操作等待队列。

3. 唤醒进程

内核通过 wake_up( )、wake_up_nr( )、wake_up_all( )、wake_up_sync( )、wake_up_sync_nr( )、wake_up_interruptible( )、wake_up_interruptible_nr( )、wake_up_interruptible_all( )、wake_up_interruptible_sync( )和 wake_up_interruptible_sync_nr( )来唤醒进程。可以从其名字知道每个宏的含义。

- 所有宏都考虑到处于 TASK_INTERRUPTIBLE 状态的睡眠进程；如果宏的名字中不含字符串“interruptible”，则还将考虑处于 TASK_UNITERRUPTIBLE 状态的睡眠进程。
- 所有宏唤醒具有所需状态的所有非互斥进程。
- 名字中含有“nr”字符串的宏唤醒给定数字的具有所需状态的互斥进程；这个数字是宏的一个参数。名字中含有“all”字符串的宏唤醒具有所需状态的所有互斥进程。最后，名字中不含“nr”或“all”字符串的宏只唤醒具有所需状态一个互斥进程。
- 名字中不含有“sync”字符串的宏检查唤醒进程的优先级是否高于系统中正在运行进程的优先级，并在必要时调用 schedule( )。

例如，wake_up 宏等价于下面的代码片断：

```
void wake_up(wait_queue_head_t *q)
{
        struct list_head *tmp;
        wait_queue_t *curr;

        list_for_each(tmp, &q->task_list) {
            curr = list_entry(tmp, wait_queue_t, task_list);
            if (curr->func(curr, TASK_INTERRUPTIBLE|TASK_UNINTERRUPTIBLE,
                        0, NULL) && curr->flags)
                break;
```

```
        }
    }
```

代码扫描所有的等待队列项。对每一项，代码将进程描述符指针传递给 wake_up_process( ) 函数。如果唤醒的进程是互斥的，循环结束。被唤醒的进程并不从等待队列中删除。一个进程可能被唤醒，而等待条件仍然为假；在这种情况下，进程可能又一次把自己挂起在同一等待队列中。

14.2.2 Linux 内核的并发性和同步机制

操作系统内核执行过程中，以下原因会造成并发执行：

- 中断可能在任何时刻异步发生，随时可能打断正在执行的内核代码。
- 内核调度 tasklet 和 softirq，打断当前正在执行的代码。
- 如果内核具有抢占性，运行的内核任务会被另一个任务抢占。
- 进程被阻塞时，会唤醒调度程序工作，引起其他进程运行。
- SMP 平台上，两个或多个 CPU 同时执行内核代码，可能访问同一共享数据结构。

这就要求有同步机制来保证不出现竞争条件，在中断处理程序中能避免并发访问的安全代码称作中断安全代码、在 SMP 的计算机中能避免并发访问的安全代码称 SMP 安全代码、在内核抢占时能避免并发访问的安全代码称抢占安全代码。Linux 提供多种内核同步机制避免上述竞争条件，实现各种内核安全代码。

14.2.2.1 原子操作

原子操作保证在执行过程中不被打断，以防止发生简单的竞争条件，确保操作结果的正确性，复杂的锁机制能在原子操作基础上构建。Linux 内核定义两类原子操作。

1. 原子整数操作

针对整型变量，定义特殊数据类型 atomic_t 和专门的原子整型操作函数，典型用途是实现计数器。以下是部分原子整型操作：ATOMIC_INT(int i)(初始化原子变量为 i)、atomic_read(atomic_t *v)(读整数值 v)、atomic_set(atomic_t *v,int i)(把 v 置成 i)、atomic_add(int i, atomic_t *v)(把 v 增加 i)、atomic_sub(int i, atomic_t *v)(把 v 减去 i)、atomic_sub_and_test(int i, atomic_t *v)(从 v 中减去 i，结果为 0 时，则返回 1，否则返回 0)、atomic_add_negative(int i, atomic_t *v)(将 v 中增加 i，结果为 0 时，则返回 1，否则返回 0)、atomic_inc(atomic_t *v)(将 v 加 1)和 atomic_dec(atomic_t *v)(将 v 减 1)等。

2. 原子位图操作

针对指针变量指定的任意一块主存区域的位序列进行操作。以下是部分原子位图操作：set_bit(int nr,void *addr)(设置位图地址 addr 的 nr 位)、clear_bit(int nr,void *addr)(清除位图地址 addr 的 nr 位)、change_bit(int nr,void *addr)(反转位图地址 addr 的 nr 位)和 test_bit(int nr,void *addr)(返回位图地址 addr 的 nr 位的值)等。

14.2.2.2 内核信号量

在 Linux 内核中，也使用等待队列来实现信号量机制，内核信号量 semaphore 定义为：

```
struct semaphore {
    atomic_t count;
    int sleepers;
    wait_queue_head_t wait;
};
```

资源计数器 count 表示可用的某种资源数，若为正整数则尚有这些资源可用；若为 0 或负整数则资源已用完，且因申请资源而等待的进程有 count 绝对值个。sema_init 宏用于初始化 count 为任何值，可以是二元信号量，也可以是一般信号量；在 Down 操作中，count 减 1 后的值若小于 0，进程便进入等待队列。Up 操作中，count 加 1 后的值如果大于 0，立即唤醒等待队列中的所有进程。

- sleepers 对等待当前临界资源的进程个数进行辅助计数。
- wait 用于存放等待队列链表的地址，该链表包含当前正在等待该信号量(资源)而被阻塞的所有进程。

内核信号量上定义的函数有：DOWN( )、DOWN_interruptible( )、DOWN_trylock( )和 UP( )。此外，还提供读者—写者信号量，相应的操作有：down_read( )（读者 Down 操作）、up_read( )（读者 Up 操作）、down_write( )（写者 Down 操作）及 up_write( )（写者 Up 操作）。

14.2.2.3 内核等待队列

Linux 内核信号量采用非忙式等待实现，当进程执行 Down 操作而等待时，将放入等待信号量队列；事实上，并发进程同步时，只要等待条件不满足，就必须挂起，放入相应等待队列。所以，等待队列是支持进程同步的重要数据结构，其定义为：

```
struct wait_queue {
 unsigned int compiler_warning;
 struct task_struct *task;                /*指向等待进程的 PCB*/
 struct list_head task_list;              /*等待队列链表*/
};
```

内核等待队列也是临界资源，修改时应加锁，以避免竞争条件。内核为每个事件都设立一个等待队列，若进程等待一个指定事件发生时，它必须调用 add_wait_queue( )把自己加入该事件的等待队列，并通过 schedule( )交出 CPU。等待事件发生后，内核调用 remove_wait_queue( )将一个进程从等待队列中移出。

14.2.2.4 关中断

一个正在对内核数据结构操作的进程可以被 I/O 设备中断，如果设备中断处理程序恰好也

要访问该数据结构，就会引起系统的不一致状态，解决这类问题的有效方法是关中断。关中断是把内核态执行的程序段作为临界区来保护的一种手段，主要保护中断处理程序也要访问的数据结构，如磁盘中断就需要被屏蔽，此外，关中断还能禁止内核抢占。为了防止死锁，关中断期间内核不能执行阻塞操作。在 SMP 环境中，关中断只能防止来自本机其他中断处理程序的并发访问，需要引入自旋锁在禁止本地中断的同时，防止来自它机的并发访问。

14.2.2.5 自旋锁

原子操作能保证对变量操作的原子性，例如，原子加操作，把读取和增加变量的动作包含在一个单步中执行，从而防止发生竞争。可是临界区却不像增加变量这样简单，有时它可以跨越多个函数。例如，先从一个数据结构中读出数据，对其进行格式转换和解析，再把它写到另一个数据结构中。整个执行过程必须是原子的，在数据被更新完毕前，不能有其他代码读取这些数据。显然、简单的原子操作对此无能为力，这就需要使用更为复杂的同步方法——锁来提供保护。

Linux 内核中最常见的锁是自旋锁，自旋锁最多只能被一个可执行线程持有。如果一个执行线程试图获得一个已经被锁住的自旋锁，那么该线程就会一直进行忙式等待(旋转)，等待锁重新可用，期间该 CPU 不能再处理其他工作，同时等待其他 CPU 上运行的进程执行解锁操作，要是锁未被争用，请求锁的执行线程便能立刻锁住它，继续执行。在任意时刻，自旋锁都可以防止多于一个的执行线程同时进入临界区。

自旋锁是最简单的一种锁原语，锁的取值为 0 表示资源可用，锁的取值为 1 表示资源加锁。自旋锁很像二元信号量，但在实现上有区别，若一个资源得到自旋锁的保护，另一个试图取得该资源的内核例程将保持忙式等待，直到资源被解锁。而在实现二元信号量时，不必循环等待信号量，而是进入等待队列，暂时放弃对资源的请求。

Linux 中，自旋锁变量 lock 被定义成 spinlock_t 类型，只有 lock 成员的最低位被使用，如果锁可用，lock 的最低位为 0；如果上锁 lock 的最低位为 1，初始化时，lock 成员被设为 0、即未上锁。自旋锁定义如下：

```
typedef struct {
   volatile unsigned int lock;
}spinlock_t;

void spin_lock(spinlock_t *plock)
{
      int flag;
      do{
      flag=plock->lock&1;          /*取出 lock 的第 0 个 bit，若为 1，已上锁*/
      plock->lock=1;               /*上锁，将 lock 的第 0 个 bit 置为 1*/
      }while(flag!=0);             /*若已上锁则循环测试，直到成功*/
```

```
}

void spin_unlock(spinlock_t *plock)
{
    plock->lock&=~1;            /*开锁,将 lock 的第 0 个 bit 置为 0*/
}
```

在 SMP 系统中，自旋锁最重要的特点是内核例程在等待锁被释放时一直占据着某个 CPU，一般用来保护那些只需简短访问数据结构的操作，如在双向队列链表增加或删除一个元素，因此，在内核需要进程同步的位置，自旋锁用得十分频繁。

自旋锁的上锁或解锁通过以下宏实现：spin_lock(spinlock_t *lock)/spin_unlock(spinlock_t *lock)(获得/释放自旋锁)、spin_lock_irq(spinlock_t *lock)/spin_unlock_irq(spinlock_t *lock)(获得自旋锁并关闭中断/释放自旋锁并开放中断)、spin_lock_bh(spinlock_t *lock)/spin_unlock_bh(spinlock_t *lock)(获得自旋锁并关闭 bh 的执行/释放自旋锁并开放 bh 的执行)和 spin_lock_init(spinlock_t *lock)(初始化自旋锁)等。

Linux 中的读者—写者自旋锁机制允许在内核中实现比基本自旋锁更大的并发度。每个读者—写者自旋锁包含一个 24 位的读者计数器和一个解锁标记，有 3 种状态：当计数器=0，标记=1 时，自旋锁释放，可以使用；当计数器=0，标记=0 时，自旋锁被一个写线程获得；当计数器>0，标记=0 时，自旋锁已被若干个读线程获得。读者—写者自旋锁有：获得/释放读锁 read_lock( )/read_unlock( )、禁止/开放本地中断并获得/释放读锁 read_lock_irq( )/ read_unlock_irq( )。类似地有：write_lock( )、write_unlock( )、write_lock_irq( )、write_unlock_irq( )、试图获得指定写锁 write_trylock( )、初始化指定锁 rw_lock_init( )等。

因为自旋锁在同一时刻至多被一个执行线程持有，所以一个时刻只能有一个线程位于临界区内，这就为 SMP 计算机提供防止并发访问所需的保护机制。在单 CPU 计算机上，编译操作系统时并不会加入自旋锁，它仅仅被当作一个设置内核抢占机制是否被启用的开关，如果禁止内核抢占，那么在编译时会被完全剔出内核。

14.3 实 验 内 容

实验 设计并实现新的同步原语

1. 实验说明

设计一组新的同步原语，多个进程可以在同一个事件上等待。当其他进程产生该事件时，等待在该事件上的进程被唤醒。如果事件产生时，没有进程因这个事件而阻塞，则该事件无效。内核需要为该同步原语提供 4 个系统调用。

• int event_open(int arg)：打开一个事件。如果参数 arg 等于 0，内核创建一个新事件，并且返回新事件的 ID；如果参数 arg 大于 0，内核打开一个已有事件 ID(为 arg 的事件)，如果该事件不存在，内核返回–1。

• int event_close(int)：关闭已有事件。关闭成功返回 0，关闭失败返回–1。如果在关闭时还有进程等待在该事件上，内核将这些进程唤醒。

• int event_wait(int arg)：进程在事件 arg 上等待，直到其他进程产生该事件。如果事件不存在，内核返回–1。

• int event_sig(int arg)：进程产生事件 arg，等待在该事件上的进程被唤醒。如果事件不存在，内核返回–1。

2. 解决方案

（1）数据结构

用户在使用该组同步原语时，可以动态地创建多个事件。用内核提供的双向链表将这些数据结构组织在一起是比较好的方法。对于链表中的每一个事件，内核首先需要为事件定义一个事件 ID；由于进程可以等待在事件上，因此每个事件都需要有一个单独的等待队列，睡眠于该事件的进程便可以放入进程的该等待队列中。在事件数据结构中，还需要记录该事件是否已经发生。根据这些要求，可以将事件的数据结构定义为：

```
typedef struct _event{
    int id;
    wait_queue_head_t *wq;
    bool occur;
    struct list_head events;
} event_t;
```

id 表示事件 ID；wq 为等待队列头；occur 表示事件是否已经发生过；events 是用于将事件链在一起的双向链表元素。同时，内核需要记录事件链表的头元素，这样内核才能对链表进行搜索。事件链表的头元素定义为内核中的一个全局变量：

```
event_t *event_head = {0, NULL, 0, LIST_HEAD_INIT(event_head.events)};
```

Linux 双向链表的使用可以参考本书前面的相关内容。当定义好链表头后，事件便可以插入链表中。为了加快事件的搜索速度，事件在插入链表时，按照事件 ID 从小到大进行排序。

（2）event_open( )

event_open( )系统调用是为了创建或者打开已有的一个事件。该系统调用的主要流程如图 14-1 所示。

（3）搜索事件

在 event_open( )中，根据用户提供的事件 ID，内核需要在事件链表中搜索到相应的事件元素。由于链表采用的是 Linux 内核提供的双向链表，并且链表中元素按事件 ID 排序，搜索事件可以采用以下的代码：

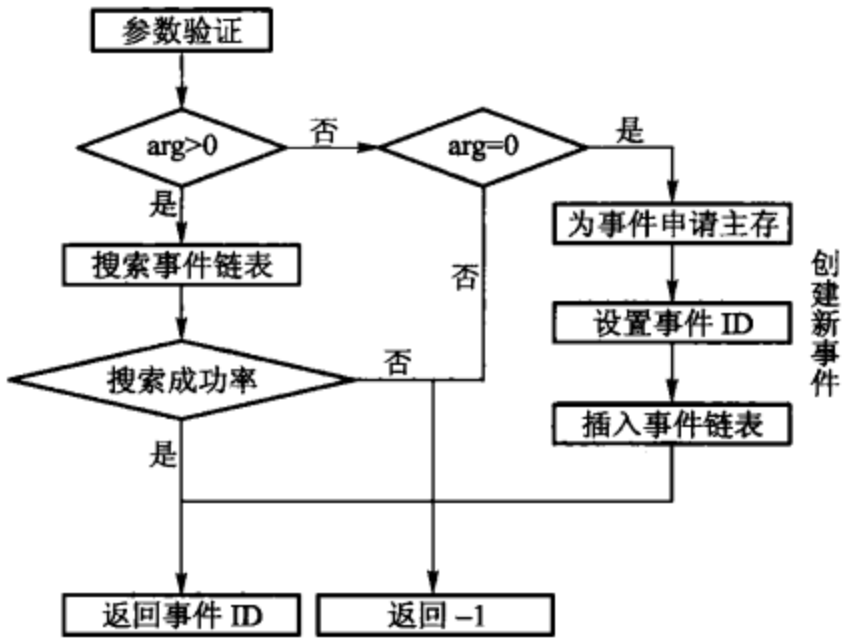


图 14-1 event_open( )流程图

```
event_t *search_event( int id )
    {
        event_t *event = NULL;
        list_head *p = NULL;
        list_for_each(p, event_head) {
            event = list_entry(p,event_t, events);
            if ( event->id==id )   /*找到元素*/
                return event;
            if ( event->id>id )    /*链表有序排列，因此不需要再继续查找*/
                return NULL;
        }
        return NULL;
    }
```

search_event( )如果找到需要的事件，将事件的指针返回，否则返回 NULL。调用者可以根据返回值判断事件的查找是否成功。

（4）创建新事件

创建新事件主要涉及 3 个流程：为事件申请主存、为事件申请 ID 和将事件插入事件链表。

在内核中，申请主存可以采用 kmalloc( )，该内核函数的原型为：

```
void *kmalloc(size_t size, int flags);
```

size 为所申请的主存字节数，flags 为要分配的主存类型，返回值为申请到的主存地址。flags 参数取值如下：GFP_USER 代表用户分配主存，可以睡眠；GFP_KERNEL 分配内核中的主存，可以睡眠；GFP_ATOMIC 表示分配但不睡眠，一般在中断处理程序中使用。为事件申请主存可以使用如下代码：

```
newevent = (event_t *) kmalloc(sizeof(event_t), GFP_KERNEL);
newevent->wq = (wait_queue_head_t *) kmalloc(sizeof(wait_queue_head_t), GFP_KERNEL);
init_waitqueue_head(newevent_wq);
```

该段代码先为事件申请主存，在为事件的等待队列申请主存，最后将等待队列进行初始化。

新事件在申请好主存之后，内核需要为新事件设置一个全局唯一的 ID。事件链表中的事件按照事件 ID 从小到大进行排序，因此只需要将事件链表中最后一个事件的 ID 加 1 赋值给新事件即可。最后只需要将新事件插入事件链表的队尾便完成新事件的创建过程。

（5）event_wait( )、event_sig( )与 event_close( )

1）event_wait( )

事件创建好以后，进程可以通过 event_wait( )系统调用在事件上等待。Linux 提供 wait_event 宏让进程可以在指定的等待队列中等待。该宏的使用方法是：

```
wait_event(wq, condition);
```

wq 是使用到的等待队列，condition 是一个布尔值的表达式。进程调用该宏后，会在指定的等待队列中睡眠，直到指定的条件变为真。在事件的数据结构中，occur 成员表明事件是否已经发生，因此该成员便是 wait_event 需要使用到的条件。event_wait( )的主要结构如下：

```
int event_wait( int id)
{
    …
    event_t *e = search_event(id);
    wait_event(e->wq, e->occur);
    …
}
```

2）event_sig( )

event_sig( )系统调用的作用是唤醒在指定事件上睡眠的进程，它的主要工作流程是：

① 设置事件的 occur 成员，表明事件已经发生。

② 唤醒睡眠进程。

与 wait_event( )宏类似，内核提供 wake_up(wq)宏唤醒在指定等待队列中的睡眠进程，wq 是指定的等待队列。event_sig( )的结构如下：

```
int event_sig(int id)
    {
        …
        event_t *e = search_event(id);
        e->occur = true;
        wake_up(e->wq);
        …
}
```

3）event_close( )

event_close( )系统调用需要将事件删除，并且在事件被删除前将睡眠在该事件上的进程唤醒。删除事件的主要流程是：

① 唤醒睡眠在该事件的进程。

② 将事件从事件链表中删除。

③ 将事件所占用的主存释放。

该流程涉及的内核函数有包括 list_del( )和 kfree( )。list_del( )将指定的双向链表节点从链表中删除，kfree( )用于释放用 kmalloc( )申请的主存。这两个内核函数的原型是：

```
void list_del(struct list_head *entry);
void kfree(const void *objp);
```

在 list_del( )内核函数中，参数 entry 是要删除的双向链表节点；在 kfree( )内核函数中，参数 objp 是需要释放的主存地址。

（6）封装系统调用

最后一步是将这些系统调用封装成函数，可以参考系统调用一章的内容。

（7）使用新的同步原语

在定义新的系统调用及封装成函数以后，程序中就可以使用它们了。在本节中，将通过编写几个测试程序来验证。程序用 SYS_eventopen、SYS_eventclose、SYS_eventwait 和 SYS_eventsig 这 4 个宏表示这几个系统调用的系统调用号。测试程序包括 4 个文件：

```
/*****open.c、close.c、sig.c、wait.c*****/
/*open.c*/
#include <linux/unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
        if ( argc!=2 )
                return –1;
        int id = syscall(SYS_eventopen, aroi(argv[1]);
        printf("%d\n", id);
        return 0;
}

/*close.c*/
#include <linux/unistd.h>
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char** argv)
{
        if ( argc!=2 )
                return –1;
        int r = syscall(SYS_eventclose, aroi(argv[1]);
        printf("%d\n", id);
        return 0;
}

/*wait.c*/
#include <linux/unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
        if ( argc!=2 )
                return –1;
        i = syscall(SYS_eventwait, atoi(argv[1]));
        printf("%d\n",i);
        return 0;
}

/*sig.c*/
#include <linux/unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
        if ( argc!=2 )
                return –1;
        i = syscall(SYS_eventsig, atoi(argv[1]));
        printf("%d\n",i);
        return 0;
}
```

编译源代码后获得可执行程序 open、close、wait 和 sig。open 程序用于打开一个指定的事件，如果事件 ID 为 0，则创建一个新事件，并将事件 ID 打印到控制台，例如：

```
#open 0
1
```

这表明 open 程序创建了事件 ID 为 1 的新事件。当一个事件创建好以后，可以通过 wait 命令在该事件上等待：

```
#wait 1
```

执行该命令后，进程将在该事件上等待，并进入睡眠状态。如果要将该进程唤醒，需要切换到其他控制台，并通过 sig 程序唤醒该进程。

```
#sig 1
```

执行该命令之后，等待事件 1 上的进程将被唤醒。在实验的最后，用户需要通过 close 程序将已经打开的事件关闭，将占用的资源释放。

```
#close 1
```

3. 程序框架

```
#include <linux/List.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/unistd.h>
#include <linux/wait.h>

typedef struct _event{
        int id;
        wait_queue_head_t *wq;
        bool occur;
        struct list_head events;
} event_t;

event_t *event_head = {0, NULL, 0, LIST_HEAD_INIT(event_head.events)};

event_t *search_event( int id )
{
        /*搜索事件，如果查找不到该事件，返回 NULL*/
}

int event_open( int id )
{
        /*查找该事件*/
        /*如果事件不存在，创建新事件*/
}

int event_wait( int id)
{
        /*通过 wait_event 函数将进程放入等待队列*/
```

```
}

int event_sig(int id)
{
    /*通过 wake_up 函数将进程唤醒*/
}

int event_close(int id)
{
    /*关闭事件，并释放相关内存资源*/
}
```

第15章 进程调度

15.1 实验目的

- 深入理解进程调度的基本原理。
- 了解 Linux 2.4 调度算法及其不足。
- 掌握 Linux 2.6 调度算法的原理与实现。
- 能够根据要求修改 Linux 2.6 调度算法。

15.2 背景知识

15.2.1 调度策略和调度机制

CPU 调度按照层次可分 3 级：高级调度、中级调度和低级调度，这里仅讨论低级调度。低级调度的最初对象是进程，随着多线程机制的引入，一些操作系统中进程演变成资源分配和保护单位，从而进程只作为中级调度对象，内核级线程则替代进程成为低级调度对象。适用于进程调度的算法一般都适用于内核级线程调度。

CPU 调度指的是在一组就绪的进程/线程中进行 CPU 分配，采用多道程序设计的操作系统中，在一段时间间隔内，允许多个进程加载到主存中，并通过时分复用(时间片)技术让在主存的进程/线程共享 CPU。当一个进程/线程执行结束、或等待某事件发生而进入阻塞状态时，由调度程序将 CPU 分配给另一个进程/线程运行。在操作系统控制下，通常有许多进程/线程处于就绪状态，当 CPU 变成可用时，调度程序从就绪队列中选择一个来使用 CPU。操作系统的调度程序(scheduler)有两项任务：调度和分派。前者实现调度策略(scheduling policy)确定就绪进程/线程竞争使用 CPU 的次序的裁决原则，即进程/线程何时应放弃 CPU 和选择哪个来执行；后者实现调度机制(scheduling mechanism)确定如何时分复用 CPU，处理上下文交换细节，完成进程/线程和 CPU 的绑定与放弃的具体工作。

从概念上看，调度机制由 3 个逻辑功能程序模块组成。

- 队列管理程序。当一个进程/线程变为就绪时，会更新其进程控制块(PCB)来反映这种变化，队列管理程序(queuer)将 PCB 指针放入等待 CPU 的进程列表中，每当把进程/线程移入就绪队列时，可计算为该进程/线程分配 CPU 的优先级。
- 上下文切换程序。当调度程序把 CPU 从正在执行的进程/线程使剥夺切换到另一个进程/

线程执行时，上下文切换程序(context switcher)将当前执行进程/线程的上下文信息保存到它的PCB中，恢复选中进程/线程的上下文信息，从而，让它占用CPU运行。

• 分派程序。当一个进程/线程出让CPU后，分派程序(dispatcher)被激活，当然，为了运行分派程序，需要把它的上下文装入CPU，分派程序从就绪队列中选择进程/线程，而后，完成从它自己到选择的进程/线程间的又一次上下文切换，把CPU出让给选中的进程/线程。

低级调度的基本类型指调度策略中选择下一个运行进程的时间点和仲裁方式，在其他时间不能改变已获得CPU的进程/线程的分派情况，据此有两类基本调度方式。

• 剥夺式(Preemptive)：当进程/线程正在CPU上执行时，系统可根据规定的原则剥夺分配给它的CPU，把其移入就绪队列，选择其他进程/线程运行。常用的有两种CPU剥夺原则，一是高优先级进程/线程可剥夺低优先级进程/线程，二是当运行进程/线程时间片用完后被剥夺，在动态改变进程/线程优先级的系统中，经常出现这种情况。

• 非剥夺式(Nonpreemptive)：一旦某个进程/线程开始执行后便不再出让CPU，除非该进程/线程运行结束、或主动放弃CPU、或发生某个事件不再能继续执行。

Linux 2.6与Linux 2.4采用的调度策略有所不同。Linux 2.4的调度程序有很多不足之处，Linux 2.6内核使用新的调度算法，称为*O*(1)调度算法，它在高负载情况下的执行极其出色，且在SMP中也可以很好地扩展。

15.2.2 Linux 2.4的调度算法及其不足

Linux 2.4进程调度函数是schedule( )，它将选出一个可运行进程，并且让该进程占用CPU运行。进程task_struct结构中，存放与进程调度有关的重要成员，供进程调度使用。

① policy：标识进程调度策略，有以下3种类型。

• SCHED_OTHER：普通进程。采用基于优先级的时间片轮转法调度，只要有实时进程就绪，这类进程便不能运行。

• SCHED_FIFO：先进先出实时进程。可以一直使用CPU运行，除非自己出现等待事件或被具有更高优先级的实时进程替代。

• SCHED_RR：轮转法实时进程。除时间片是个定量外，和SCHED_FIFO类似，当时间片耗尽后，使用相同优先级排到原队列的队尾。

② priority：进程静态优先级，内核v2.4后版本已取消该变量。

③ nice：进程可控优先级因子，其值是–20～19的整数，可用于改变进程的静态优先级。默认值为0，增加nice值会降低进程的优先级。

④ rt_priority：实时进程优先级，仅被实时进程使用，是0～99的一个整数，区分实时进程的等级，较高权值的进程总优先于较低权值的进程，实时进程的优先级总高于普通进程的优先级。

⑤ counter：进程目前时间片配额，也称进程动态优先级。

schedule( )函数的执行过程大致如下。

① 检查是否有软中断服务请求，如果有，则先执行这些请求。

② 若当前进程调度策略是 SCHED_RR 且 counter 为 0，则将该进程移到可运行进程队列尾部并对 counter 重新赋值。

③ 若当前进程状态为 TASK__INTERRUPTIBLE 且它有信号接收，则将进程状态置为 TASK_RUNNING；若当前进程状态不是 TASK_RUNNING，则将其从可执行进程队列中移出，然后将当前进程描述符的 need_resched 恢复成 0。

④ 调度程序的 goodness( )函数为可运行进程队列的每个进程计算出一个权值，最终最大的权值保存在变量 c 中，与之对应的进程描述符保存在变量 next 中。

⑤ 检查 c 是否为 0。若为 0，则表明所有可运行进程的时间配额都已用完，此时对所有进程的 counter 重新赋值，然后重新执行第 5 步。

⑥ 如果 next 进程就是当前进程，则结束 schedule( )函数的运行。否则进行进程切换，CPU 改由 next 进程占据。

goodness( )函数用来计算进程的当前权值，它的第 1 个参数是待估进程的描述符，返回值 c 比较真实地反映了待估进程“值得运行的程度”。c 的取值范围如下：

- c= –1 000，表示永远不必选择待估进程。当运行队列里只有一个进程时，选择该值。
- c=0，表示待估进程的时间片用完，在其他进程的时间片用完之前不会选择它。
- 0<c<1 000，表示待估进程的时间片还没有用完，剩余的时间片可以看做优先级。
- c ≥ 1 000，表示待估进程是实时进程，应该优先执行。

如果该进程是实时进程，其权值为 1 000 + rt_priority，普通进程权值无法到达 1 000，因而实时进程可以优先得到执行。对于普通进程，它的权值为 counter + 20 – nice，如果它又是内核线程，由于无须切换用户空间，则将权值加 1 作为奖励。找出切换进程后，调用 switch to( )宏，让新进程开始执行。

在 v2.4 调度算法中，所有处于就绪态的进程都被放入同一个队列中，即使在 SMP 环境下也只有一个就绪队列，当进行 CPU 调度时，需要遍历进程就绪队列，执行循环直到找到优先级最高的进程。如果所有准备进行调度的进程时间片耗尽，需要对就绪队列中的每个进程都重新计算时间片，然后返回前面的调度过程，重新在就绪队列中查找优先级最高的进程，让它占有 CPU 执行。该算法明显的不足处有以下几方面。

- 每次调度时都需要遍历所有就绪态进程，需要耗费 $O(n)$的时间。如果处于就绪态的进程非常多，遍历的效率十分低。
- 多个 CPU 共享一个就绪队列，每个 CPU 在访问就绪态队列时都需要进行加锁操作，当一个 CPU 在调度进程，而其他 CPU 处于空闲时，其他 CPU 也只能等待。这样大大降低了调度算法在 SMP 环境中的效率。
- 进程可能在多个 CPU 上切换，由于多 CPU 共享同一个就绪态队列，一个进程可能在一个时间片用完之后被调度到另外 CPU 上执行，进程在不同 CPU 上切换降低 CPU 缓存的效率，从而降低了系统性能。

由此可见，在 v2.4 调度算法中，如果系统中有大量的就绪态进程，调度进程需要花费较多时间，并且不能很好地发挥 SMP 的优势。基于以上考虑，Linux 2.6 对 v2.4 的调度算法进行了重大改进。

15.2.3 Linux 2.6 调度算法的设计与实现

15.2.3.1 Linux 2.6 调度算法概述

Linux 2.6 调度算法的设计主要是为了克服 v2.4 调度算法的不足，它有以下特性：

① 调度器的开销恒定，调度开销不随着系统中就绪态进程和 CPU 数目的增多而增大。

② 对 SMP 有良好的可伸缩性，每个 CPU 都有自己的锁和可运行队列。

③ 提高对 SMP CPU 的亲和性，在负载均衡的情况下避免进程在 CPU 之间来回切换。

④ 加强交互性和公平性，在系统负载较重的情况下，确保系统响应时间，及时调度交互型进程。在合理设置的时间范围内，没有进程会饥饿，也没有进程能获得过量 CPU 时间。

v2.6 内核中，进程描述符 task_struct 主要含有如下与调度有关的成员。

① policy：进程调度策略，有以下 3 种类型：

- SCHED_NORMAL　非实时进程。
- SCHED_FIFO　实时进程，采用先进先出的调度算法。
- SCHED_RR　实时进程，采用轮转法。

② rt_priority：实时进程的优先级。MAX_RT_PRIO 定义为 100，故其 rt_priority 范围为 0～99，且不参与优先级计算。

③ static_prio：非实时进程静态优先级。由 nice 值转换而来，nice 值为 -20～19，公式为：static_prio=MAX_RT_PRIO+nice-20，故其范围为 100～139。

④ sleep_avg：进程平均等待时间。相当于进程等待时间与运行时间的差值，既反映该进程的交互程度，又表示进程需要运行的紧迫程度，该值越大，算出来的数据越小，进程的优先级就越高。

⑤ prio：进程动态优先级。在进程运行过程中动态计算，主要影响因素为 sleep_avg。创建时子进程继承父进程的动态优先级、唤醒等待进程时对它进行优先级修正、时钟中断中重新计算进程优先级并进入相应队列、负载平衡/修改 nice 值/修改调度策略(setscheduler( ))都有可能改变进程动态优先级。

⑥ prio_array_t* array：进程优先级数组。以进程优先级为序号排列。

⑦ time_slice：进程时间片余额。进程默认时间片与 static_prio 有关，内核将 100～139 的优先级映射为 800 ms～5 ms 的时间片区间。

⑧ load_weight：平衡负载用的权重。解决可运行队列出现的负载不均现象。

⑨ CONFIG_PREEMPT：内核可剥夺编译选项，当该开关开启时，v2.6 内核将会在更多内核安全点上检测 TIF_NEED_RESCHED 位，从而让刚被唤醒的高优先级进程减少延迟而尽快获

得 CPU 运行。

15.2.3.2 O(1)调度算法设计和实现

1. 运行队列

Linux 2.6 的调度算法称为 *O*(1)算法，不论就绪进程与 CPU 的个数多少，调度程序的调度开销总是恒定的常数。每个 CPU 都有一个运行队列(runqueue)，它是给定 CPU 上的可执行进程的链表，每个就绪进程都唯一归属于一个运行队列。此外，运行队列中还包含每个 CPU 的调度信息，运行队列的定义如下：

```
struct runqueue {
  spinlock_t          lock;                    /*本 CPU 上的运行队列自旋锁*/
  unsigned long       nr_running;              /*本 CPU 上活跃、过期队列就绪进程总数*/
  unsigned long       cpu_load;                /*各处理器上的负载*/
  unsigned long long nr_switches;              /*本 CPU 上发生的上下文切换数*/
  unsigned long long timestamp_last_tick;      /*本就绪队列发生调度时间*/
  unsigned long       expired_timestamp;       /*进程耗完时间片事件的时间*/
  unsigned long       nr_uninterruptible;      /*本 CPU 上不可中断阻塞态的进程数*/
  struct mm_struct    *prev_mm;                /*前面运行进程的 active_mm 指针*/
  prio_array_t    *active,*expired;            /*指向活动、过期优先级数组的指针*/
  prio_array_t    arrays[2];                   /*活跃、过期两类优先级数组*/
  int best_expired_prio;                       /*过期队列中进程最高优先级(数值最小) */
  int active_balance;                          /*平衡 CPU 负载使用*/
  int push_cpu;
  task_t              *curr,*idle;             /*该处理器当前运行和空进程*/
  task_t *migration_thread;                    /*处理器上的迁移进程*/
  struct list_head *migration_queue;           /*处理器上的迁移进程队列*/
  atomic_t            *nr_iowait;              /*本 CPU 上等待 I/O 的进程数*/
}
```

内核为每个运行队列维持调度用的数据结构，称优先级数组 arrays[ ]：一个是活跃的(active)、一个是过期的(expired)，每个成员内含 140 个双向链表，相同优先级的进程链接在同一个双向链表里，指针 active 和 expired 分别指向 arrays 的成员。进程调度时从指针 active 指向的 arrays[ ]的成员里选择下个要运行的进程。当非实时进程的时间片用完后，就会被移到指针 expired 指向的 arrays[ ]的成员的双向链表里。prio_array 的定义如下：

```
struct prio_array {
  unsigned int        nr_active;                /*数组中的进程数*/
  unsigned long     bitmap[BITMAP_SIZE];        /*优先级位图*/
  struct list_head    queue[MAX_PRIO];          /*优先级队列*/
}
```

在优先级数组中，有一个长度为 MAX_PRIO(默认值为 140)的数组，数组中的每一个元素

是一个链表。链表中放入的是具有相同优先级的就绪态进程。优先级为 n 的进程将被放入queue[n]中。系统中定义的优先级为0～139，0为最高优先级，139为最低优先级。Linux在进行调度时，将从活跃优先级队列中挑选一个优先级最高的进程执行。为了不扫描活跃优先级队列中的所有进程，优先级数组中提供了一个优先级位图，它能够帮助调度器在*O*(1)时间内找到优先级最高的进程。位图中的每一位对应一个优先级链表，如果位图中该位为 0，则说明链表为空；反之说明链表中有处于就绪态的进程。在优先级位图的定义中，BITMAP_SIZE 的默认值为5，所以bitmap共含有5个长整型，在32位计算机上共有160位(其中20位被放弃不用)。最后，nr_active指示该优先级数组内可执行的进程的个数。

初始化时，位图都被设置为 0，并且所有的运行队列都为空，当一个进程就绪时，将它放到合适的优先级队列，位图中相应位就被置 1，这样，查找系统中最高优先级就变成了查找位图中被设置的第1个位。因为优先级个数是个定值，所以查找时间恒定，并不受系统到底有多少进程的影响，Linux 对它支持的每一种体系结构都提供了对应的快速查找算法sched_find_first_bit( )。

在运行队列中分别有活跃与过期优先级队列，这两个队列中放置的都是就绪态进程，它们的不同之处在于：时间片尚未用完的就绪态进程被放置在活跃优先级队列中，因此调度器只要在活跃优先级队列中取得的进程都是可运行、并且还有可用时间片的进程。当进程的时间片用完后，它会被放入过期优先级队列，并重新计算进程应该分得的时间片，当活跃优先级队列中的进程都执行完毕以后，所有的就绪态进程都进入过期优先级队列，而在过期优先级队列中的进程都是计算完时间片的。调度器这时只需要将活跃队列与过期队列的指针交换，即原来的过期队列变成活跃队列，原来的活跃队列变成过期队列，重新在活跃队列上进行调度。

通过以上算法，调度器能够在 *O*(1)时间完成进程调度，且对于 SMP 有良好的可伸缩性和亲和性，达到了 Linux 2.6 调度算法的设计目标。

2. 进程优先级

Linux 2.6 中一共有0～139个优先级，在这140个优先级中，优先级0～99提供给实时进程使用，一个进程为实时进程，如果该进程的调度策略被设置为SCHED_FIFO或SCHED_RR。进程的调度策略是记录在进程控制块的policy成员中，用户可通过sched_setscheduler( )函数设置进程的优先级，sched_setscheduler( )函数的原型为：

```
int sched_setscheduler (pid_t pid, int policy, struct sched_param *param);
```

pid为进程的ID号；policy表示进程的调度策略；param为指向一个int的指针，表示要设置的优先级。除SCHED_FIFO和SCHED_RR调度策略外，系统还有一种称为SCHED_NORMAL的调度策略，进程创建时的默认调度策略为SCHED_NORMAL，它可适用的优先级为100～139。这3种调度策略的含义如下：

- SCHED_FIFO　表示先进先出的实时进程。当CPU调度这种进程时，该进程将一直运行，除非有更高优先级的进程进入就绪态或该进程主动放弃CPU；和该进程具有相同优先级的进程也无法被调度。

• SCHED_RR：当 CPU 调度这种进程时，如果进程的时间片用完，进程不会进入过期优先级数组，而是再次进入活跃数组。这意味着处于同一优先级的其他进程将被调度，但是任何低于该优先级的进程都不会被调度。

• SCHED_NORMAL：一般进程采用的调度策略。当进程被调度，并且时间片用尽以后，进程将被放入过期优先级数组，其他具有时间片的低优先级数组将被调度。

由此可知，只要系统中存在就绪态的实时进程，普通进程就没有办法被调度到。实时进程的优先级是通过 sched_setscheduler( )设置的，而普通进程的优先级是没法直接设置的。普通进程的优先级为两部分组成：静态优先级和动态优先级。用户能够设置进程的静态优先级，但是动态优先级是进程在运行过程中动态调整的，用户无法直接设置。

通过函数 nice( )设置静态优先级，它接受的参数为–20～19，默认值为 0。调用 nice( )后，内核将进程的静态优先级设置成 120+nice，并放入进程控制块的 static_pio 成员中，即普通进程的静态优先级为 100～139。静态优先级只是用户提供给内核的一个参考值，并不是真正用于调度的进程优先级，真正参与调度的是进程的动态优先级。

动态优先级是在静态优先级的基础上，结合进程的运行状态和进程的交换性进行计算的。进程交互性强说明进程经常进入睡眠等待用户的操作，而如果进程的交互性弱则进程基本不需要进入睡眠等待用户的输入。对于进程交互性强的进程，调度器需要给它赋予一个较高的优先级，这是因为用户希望交互性强的进程能够有较快的反应速度，满足用户的需要。例如，用户同时打开一个文字编辑器和一个视频编码程序，文字编辑器是一个交互性强的进程，而视频编码器却不是。在这种情况下，希望文字编辑器能够有较高的优先级。如果在视频编码器工作时，用户进行一次输入，在文件编辑器优先级较高的情况下，文字编辑器能够抢断视频编码器的运行，即时给用户一个反馈，对用户比较友好。

进程的交互性是通过进程的睡眠时间判断的，因此动态优先级的计算是通过静态优先级与睡眠时间进行计算的。进程控制块中的 sleep_avg 成员记录进程睡眠时间，它的范围从 0 到 MAX_SLEEP_AVG，默认值为 l0 ms。当一个进程从睡眠状态恢复到运行状态时，sleep_avg 会根据它睡眠时间的长短而增加，直至达到 MAX_SLEEP_AVG 为止。相反，进程每运行一个时钟滴答，sleep_avg 就做相应的递减，直到 0 为止。动态优先级的计算式为：

```
static int effective_prio(task_t *p)
{
    int bonus,prio;
    if(rt_task(p))
        return p->prio;                    /*若为实时进程直接返回其动态优先级*/
        bonus=(NS_TO_JIFFIES(p->sleep_avg)*MAX_BONUS/MAX_SLEEP_AVG)
        -MAX_BNOUS/2;
/*sleep_avg 越大，则 bonus 越大，动态优先级数值越小，优先级越高*/
    prio=p->static_prio-bonus;
/*得到进程动态优先级，数值越大，优先级越低*/
```

```
        if(prio<MAX_RT_PRIO)
            prio= MAX_RT_PRIO;
        if(prio>MAX_PRIO-1)
            prio= MAX_PRIO-1;
        return prio;
        …
    }
```

它的计算不仅仅基于睡眠时间有多长，还要计算运行时间的长短。所以，尽管一个进程睡眠了不少时间，但它如果把自己的时间片用得一干二净，那么它就不会得到大额的优先级奖励。这种推断机制不仅会奖励交互性强的进程，还会惩罚 CPU 耗费大的进程。如果一个进程发生了变化，开始大量占用 CPU 时间，那么，它很快就会失去曾经得到的优先级提升。

3. 时间片

时间片是进程一次最多可运行的时间长度。确定时间片的长度是比较困难的。如果时间片太长，进程切换速度太慢，系统的交互性就变得比较差；如果时间片太短，进程的频繁切换将花费系统较多的开销。在一般操作系统中，交互性较强的进程通常不需要较长的时间片，而交互性弱的进程希望能够得到较长的时间片(这样 CPU 能够有比较高的缓存命中率，可以提高系统性能)，但较长的时间片又会影响整个系统的交互性。

在 Linux 2.6 中，较好地解决了这个问题。根据进程优先级的不同，内核会给进程赋予不同的时间片长度；优先级高的时间片长，优先级低的时间片短。在 Linux 中，最短的时间片也是比较长的，这可以满足交互性弱的进程的要求。虽然采用的时间片长度较长，但是 Linux 并不会因此降低系统的交互性。在 Linux 中高优先级进程会剥夺低优先级进程的运行，同时交互性强的进程会自动提升优先级，因此交互性强的进程并不会因时间片长而受到影响。这样的时间片分配也保证高优先级进程能够有较多机会而得到运行。

当一个进程的时间片用完后，就要根据进程的动态优先级重新计算时间片。task_timeslice( )函数为给定进程返回一个新的时间片。时间片的计算只需要把优先级按比例缩放，使其符合时间片的数值范围要求即可。进程的优先级越高，它每次执行得到的时间片就越长。优先级最高的进程能获得的最大时间片长度(MAX_TIMESLICE)是 800 ms，而优先级最低的进程获得的最短时间片(MIN_TIMESLICE)是 5 ms。默认优先级(nice=0)的进程得到时间片长度为 100ms。

调度程序还提供另一种机制用于支持交互进程：如果一个进程的交互性非常强，那么当它时间片用完后，它会被再放置到活动数组而不是过期数组中。重新计算时间片是通过活动数组与过期数组之间的切换来进行的，一般进程在用尽它们的时间片后，都会被移至过期数组，当活动数组中没有剩余进程时，这两个数组就会被交换；活动数组变成过期数组，过期数组替代活动数组。这种操作提供了时间复杂度为 $O(1)$的时间片重新计算方法。但在这种交换发生之前，交互性很强的一个进程有可能已经处于过期数组中，当它需要交互时却无法执行，因为必须等到数组交换发生为止才可执行。将交互式进程重新插入到活动数组可以避免这种问题。在时钟中断处理函数 update_process_time( )中会调用 scheduler_tick( )函数，它包含以下有关排队操作：

```
void scheduler_tick(int user_ticks,int sys_ticks)
 {
    int cpu = smp_processor_id( );                    /*当前 CPU 序号*/
    task_t *p  =  current;                            /*当前进程 task_struct*/
    runqueue_t *rq  =  this_rq( );                    /*当前 CPU 运行队列*/
    …
    spin_lock(&rq->lock);                             /*封锁当前运行队列*/
    if(!--p->time_slice) {
        dequeue_task(p,rq->active);                   /*当前进程出运行队列*/
        set_tsk_need_resched(p);                      /*置 need_resched 标志*/
        p->prio = effective_prio(p);                  /*计算优先级*/
        p->time_slice = task_timeslice(p);            /*优先级换算成时间片*/
        if (!task_INTERACTIVE(p) ‖ EXPIRED_STARVING(rq)){
            enqueue_task(p,rq->expired);
            if (p->static_prio < rq->best_expired_prio)
               rq->best_expired_prio) = p->static_prio;   /*修改过期队列最高优先级*/
            }
            else
                enqueue_task(task，rq->active);
    spin_unlock(&rq->lock);                           /*开放当前 run queue*/
        …
    }
```

这段代码首先减小进程时间片的值，再看它是否为 0。如果是 0，说明进程的时间片已经用完，需要把它插入到一个数组中，所以该代码先通过 TASK__INTERACTIVE( )宏来查看这个进程是不是交互型进程，该宏主要基于进程的 nice 值来判定，nice 值越小(优先级越高)，越能说明该进程交互性越高。接着，EXPIRED_STARVING( )宏负责检查过期数组内的进程是否处于饥饿状态，即是否已经有相对较长的时间没有发生数组切换。如果最近一直没有发生切换，那么再把当前的进程放置到活动数组会进一步拖延切换，过期数组内的进程会越来越饥饿。只要不发生这种情况，进程就会被重新放置在活动数组里。否则，进程会被放入过期数组里，这也是最普通的一种操作。

调度程序在进程的创建、退出时也作了比较细致的考虑。在进程创建时，子进程被创建时并不分配自己的时间片，而是与父进程平分父进程的剩余时间片。也就是说，fork( )结束后，两者时间片之和与原先父进程的时间片相等。这样防止进程通过不断创建子进程来窃取时间片。进程退出时，根据 first_time_slice 的值判断自己是否从未重新分配过时间片，如果是，则将自己的剩余时间片返还给父进程(保证不超过 MAX_TIMESLICE)。这个动作使进程不会因创建短期子进程而受到惩罚。

4. *O*(1)调度算法

Linux 中进程调度分为直接调度和被动调度。

（1）直接调度

当进程因等待资源而需要转为阻塞状态时，直接调用 schedule( )进入调度，该函数的执行结果往往是当前进程放弃 CPU。这时的执行步骤如下：

① 把 current 指向的当前进程投入相应等待队列。

② 把当前进程的状态改为 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE。

③ 调用 schedule( )函数选择新进程占有 CPU。

④ 检查并发现所需资源可用后，把当前进程从等待队列里删除。

（2）被动调度

当前进程时间片用完后，或当一个进程被唤醒且其优先级高于当前进程的优先级时，通过将 TIF_NEED_RESCHED 位置 1，来告诉内核在适当的时刻需要重新调度。此外，进程执行函数，状态发生变化，将转入调度，这类函数有 set_scheduler( )、yield( )、pause( )、sleep( )、wait( )和 exit( )。

内核必须知道在什么时候调用 schedule( )。为了解决这个问题，系统设计一个 need_resched 标志来表明是否需要重新执行一次调度。为什么不立即调度呢?因为进程在内核里运行时，可能需要申请共享资源(如自旋锁)，若此时立即让该进程出让 CPU，如果新进程也需要相同的共享资源，则会导致系统死锁。这里仅设标志，让进程在合适的时候，即已经释放共享资源，且系统处于安全的时候，才可以调用 schedule( )函数进行进程调度。当某个进程耗尽它的时间片时，scheduler_tick( )就会设置这个标志；当一个优先级高的进程进入可执行状态的时候，try_to_wake_up( )也会设置这个标志。当内核完成工作返回用户空间或从中断返回时，内核将检查 need_resched 标志，如果设置，此时便会调用调度程序工作。每个进程都包含一个 need_resched 标志，这是因为通过 current 宏访问进程描述符内的数值要比访问一个全局变量快，在 Linux v2.2 以前的内核版本中，该标志曾经是一个全局变量；Linux v2.2～Linux v2.4 版内核中它放在进程的 task_struct 中；而在 Linux v2.6 版中，它被移到 thread_info 结构体里，用一个特别的标志变量中的一位 TIF_NEED_RESCHED 来表示。Linux 中用于访问和操作 need_resched 的函数为：set_tsk_need_resched( )(设置指定进程中的 need_resched 标志)，clear_tsk_need_resched ( ) (清除指定进程中的 need_resched 标志)，need_resched( )(检查 need resched 标志的值，如果被设置就返回真，否则返回假)。

调度算法由 schedule( )函数实现，既简单又有效。对给定的 CPU，调度程序选择在活跃优先级数组中找到第 1 个被设置的位，该位对应着优先级最高的就绪进程队列，然后，选择该优先级链表里的表头进程，它就是马上会被调度执行的进程。图 15-1 是 Linux 2.6 CPU 的调度数据结构示意，其中仅画出活跃队列的 140 位的优先级数组及相应的优先级队列；过期队列的 140 位的优先级数组及相应的优先级队列完全类似，不再重复。

新调度程序减少对循环的依赖，取而代之的是它为每个 CPU 维护两个优先级数组，活跃数组内的可执行队列上的进程都还有时间片剩余；过期数组内的可执行队列上的进程都耗尽时间片。当一个进程的时间片耗尽时，它会被移至过期数组，但在此之前，时间片已经给它重新计

算过，只要在活跃和过期数组之间来回切换，因为数组是通过指针进行访问的，切换它们所用的时间就是交换指针需要的时间。

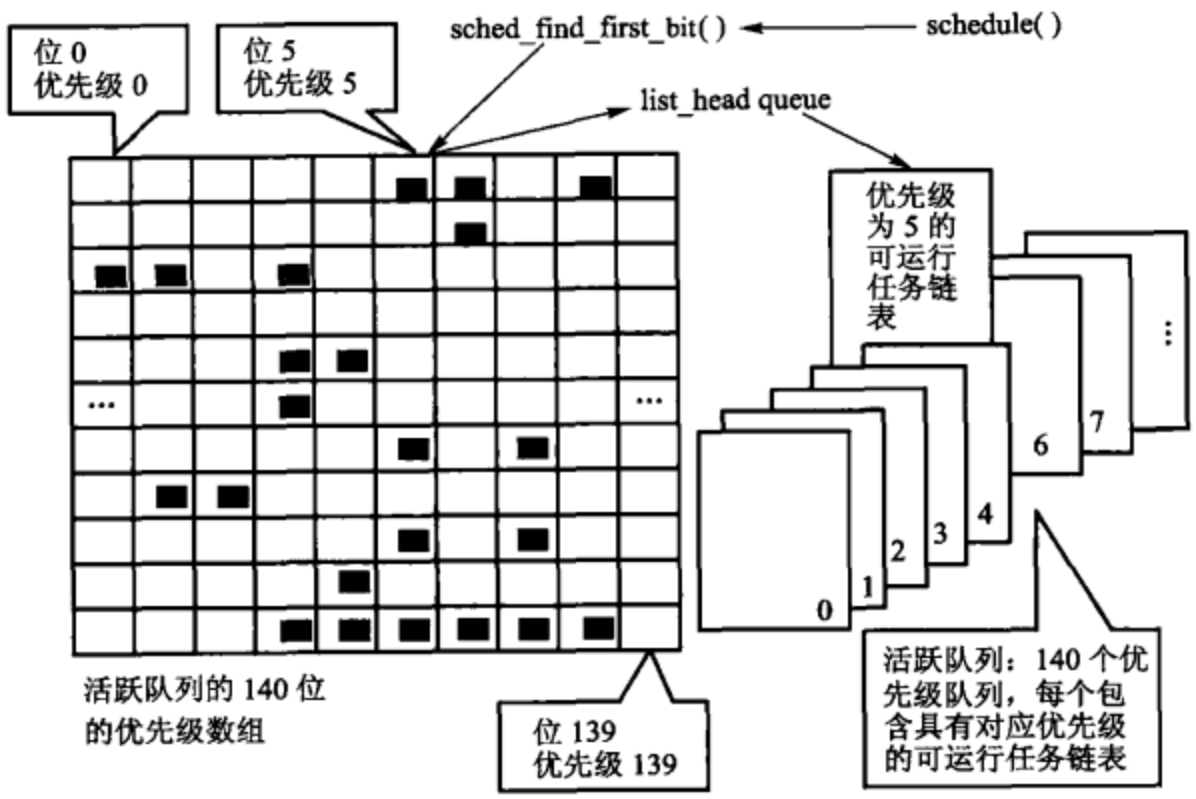


图 15-1 $O(1)$调度算法

schedule( )函数进入调度，寻找和调度最高优先级进程的执行过程由以下代码段实现。

```
asmlinkage void _sched schedule(void)
{
   task_t *prev,*next;
   runqueue *rq;
   prio_array_t *array;
   int idx;

   preempt_disable( );                  /*内核抢占锁计数加 1*/
   prev = current;                      /*保存当前进程*/
   rq = this_rq( );                     /*当前运行队列*/
   now = sched_clock( );                /*取当前时间*/

   if(prev->state && !(preempt_count( ) & PREEMPT_ACTIVE)){
      switch_count = &prev->nvcsw;
      if (unlikely((prev->state & TASK_INTERRUPTIBLE) &&
            unlikely(signal_pending(prev))))
            prev->state = TASK_RUNNING;          /*恢复到 TASK_RUNNING*/
      else
```

```
            deactivate_task(prev,rq);                    /*从运行队列中删除进程*/
}
cpu = smp_processor_id( );                               /*取 CPU 号*/
If (unlikely(!rq->nr_running) ) {
        idle_balance(cpi,rq);                            /*启动负载平衡函数*/
        …
}
array = rq->active;
if (unlikely(!array->nr_active)) {                       /*运行队列的 active 变成空*/
    schedstat_inc(rq,sched_switch);
    rq->active = rq->expired;                            /*活跃、过期数组切换*/
    rq->expired = array;
    array = rq->active;
    rq->expired_timestamp = 0;                           /*进程耗完时间片事件的时间清 0*/
    rq->best_expired_prio = MAX_PRIO;                    /*修改过期队列最高优先级*/
} else
    schedstat_inc(rq,sched_noswitch);
/*从 active 队列选择进程*/
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next,task_t,run_list);
    if (!rt_task(next) && next->activated>0) {
        unsigned long long delta = now–next->timestamp;
/*activated = 1 表示进程从可中断阻塞态被唤醒，且不在中断上下文中*/
if (next->activated == 1)
    delta = delta * (ON_RUNQUEUE_WEIGHT * 128 / 100)/128
array = next->array;
dequeue_task(next,array);                                /*从 array 取出 next*/
recale_task_prio( next,next->timestamp + delta);         /*重新计算优先级*/
enqueue_task(next,array);                                /*next 加入 array*/
  }
activated = 0;
/*任务切换*/
…
}
```

下面是 schedule( )执行调度的主要操作说明。

① 检查当前进程是否处于原子状态，如果是原子状态，表示当前进程不可以被调度。每个进程的 thread_info 中有一个抢占计数字段 preempt_count 和抢占标志位 PREEMPT_ACTIVE。如果内核编译时打开 CONFIG_PREEMPT 项，当 PREEMPT_ACTIVE 被复位且 preempt_count 不为零时，表示当前进程处于原子状态，系统不可以切换进程。

② 计算当前进程的运行时间，使用该数据调整进程的 sleep_avg、timestamp 和 last_run 等变量值。

③ 如果进程的状态不是 TASK_RUNNING，而且当前进程没有被抢占(PREEMPT_ACTIVE 标志位是零)，内核调用函数 deactivate_task( )从运行队列中删除进程。但是如果进程处于 TASK_INTERRUPTIBLE，并且有信号(signal_pending)等待处理，则进程的状态立即恢复到 TASK_RUNNING。

④ 检查自己 CPU 的运行队列是否只剩下 IDLE 进程，如果是，执行 idle_balance( )，启动 CPU 运行队列之间的负载平衡，动态转移任务到本 CPU 的运行队列。

⑤ 如果运行队列的 active 变成空，则交换 active 和 expire。

⑥ 从运行队列根据优先级选取下一个要运行的任务，重新计算它的优先级，根据新的优先级调整它在运行队列中的位置。

⑦ 如果 prev 和 next 不等，即前一个进程不是新选出的进程，必须进行任务切换，函数 prepare_task_switch( )、context_switch( )和 finish_task_switch( )负责这项工作。

context_switch( )完成大部分切换操作：如果新进程有自己的 task_struct->mm，则调用 switch_mm( )切换虚拟主存空间，要把新进程的页表起始物理地址写入页目录表起址寄存器 CR3。如果新进程和前一个进程共享同一个虚拟主存空间，则不切换 mm；硬件上下文切换，如通用寄存器 EAX、EBX、ECX、EDX、ESP、EBP 等；任务状态段切换，任务状态段是 x86 硬件提供的数据结构，每个 CPU 有一个任务状态段，内核栈地址保存在任务状态段里，不同的进程使用不同内核栈，进程切换时，必须切换任务状态段中的堆栈指针，同时实现内核堆栈指针寄存器的切换，把当前堆栈指针寄存器的内容保存到 task_struct->thread.esp，然后把新进程的堆栈指针地址从它的 task_struct->thread.esp 中取出并复制到堆栈指针寄存器里等，从前一个进程的 CPU 状态切换到新进程的 CPU 状态。

⑧ 进程切换完成后，通过检查新进程的 TIF_NEED_RESCHED 确认是否需要再次调度进程，如果需要，则回到开头，重新开始选择下一个进程。

5. 负载平衡

系统中每个 CPU 具有自己的运行队列，它只对属于自己的这些进程进行调度，如果运行队列出现负载不均时，由负载平衡程序 load_balance 来解决这个问题。它有两种调用方法，在 schedule( )执行时，只要当前可运行队列空就会被调用，找到一些就绪进程并且插入到这个队列里；它还会被定时器调用，每隔适当时间调用一次，这时需要解决所有运行队列间的失衡问题。

load_balance( )函数工作时需要锁住当前 CPU 的运行队列并屏蔽中断，以避免出现竞争条件，负载平衡进程完成后，需解除对当前运行队列的锁定并开放中断。它的操作步骤大致如下。

① 找最繁忙的可运行队列，即该队列中的就绪进程数目最多。如果没有哪个可运行队列中就绪进程的数目比当前运行队列中的就绪进程数目多 25%或 25%以上，就结束本次平衡负载工作。

② 找最繁忙的可运行队列，从中选择一个优先级数组以便抽取就绪进程。最好是过期数

组，因为那里面的就绪进程已经有相对较长的一段时间没有运行了，很可能不在 CPU 的高速缓存中。如果过期数组为空，那就只能选活动数组。

③ 找到含有进程并且优先级最高的链表，因为把优先级高的就绪进程平均分散开来才是最重要的。

④ 分析找到的所有优先级相同的就绪进程，选择一个不是正在执行，也不会因为 CPU 相关性而不可移动，并且不在高速缓存中的进程。如果有进程满足这些条件，便将其从最繁忙的队列中抽取到当前队列。

⑤ 只要可运行队列之间仍然不均衡，就重复上面两个步骤，继续从繁忙的队列中抽取进程到当前队列，最终会消除不平衡局面。

6. 用户抢占和内核抢占

内核即将返回用户空间时，如果 TIF_NEED_RESCHED 标志被设置，会导致调用 schedule( )，此时就会让刚被唤醒的高优先级进程尽快获得 CPU，发生了用户抢占。这时内核知道自己是安全的，因为既然可以继续去执行当前进程，那么也可以选择一个新的进程运行。用户抢占发生在从系统调用返回用户空间时，以及从中断处理程序返回用户空间时。

Linux 2.6 完整地支持内核抢占，只要重新调度是安全的，内核就可以在任何时间抢占正在运行的进程。那么何时重新调度是安全的呢？只要没有持有锁，内核就可以进行抢占。锁是非抢占区域的标志。由于内核是支持 SMP 的，所以，如果没有持有锁，正在运行的代码是可重入的、也就是可以抢占的。

为了支持内核抢占所作的修改是在每个进程的 thread_info 中引入 preempt_count 计数器，其初始值为 0，每当使用锁的时候数值加 1，释放锁的时候数值减 1。当数值为 0 时，内核就可执行抢占。从中断返回内核空间的时候，内核会检查 TIF_NEED_RESCHED 和 preempt_count 的值，如果 TIF_NEED_RESCHED 被设置，并且 preempt_count 为 0，这说明有一个更为重要的进程需要运行并且可以安全地抢占，此时就会调用调度程序。如果 preempt_count 不为 0，说明有当前进程持有锁，抢占是不安全的。这时，就会像通常那样直接从中断返回当前执行进程。如果当前进程持有的所有的锁都被释放了，那么 preempt_count 就会重新为 0。此时，释放锁的代码会检查 TIF_NEED_RESCHED 是否被设置，如果是的话，就会调用调度程序。内核抢占会发生在：

- 当从中断处理程序返回内核空间的时候。
- 当内核代码再一次具有可抢占性的时候。
- 如果内核中的进程显式调用 schedule( )。
- 如果内核中的进程阻塞，同样会调用 schedule( )。

有些内核代码需要允许或禁止内核抢占，可通过 preempt_disable( )和 preempt_enable( )实现。由于内核是可抢占的，内核中的进程在任何时刻都可能停下来以便另一个具有更高优先级的进程运行。这意味着一个进程与被抢占的进程可能会在同一个临界区内运行。为了避免这类情况，内核抢占代码使用自旋锁作为非抢占区域的标记。如果一个自旋锁被持有，内核便不能

进行抢占。因为内核抢占和 SMP 面对相同的并发问题，并且内核已经是 SMP 安全的，因此，这种简单的变化使内核也是抢占安全的。

15.3 实 验 内 容

实验 将 Linux 2.6 调度算法修改成随机调度算法

1. 实验说明

Linux 2.6 *O*(1)算法在进行进程调度时，达到了较好的效果，在本实验中，将把 Linux 2.6 的调度算法修改成为随机调度算法。该算法的要求是从可运行队列中随机抽取一个进程来运行。

2. 解决方案

（1）修改 schedule( )函数

首先要确定完成该算法可能要涉及的代码。随机调度的思想较为简单，完成该算法也不需要有额外信息，只需要从可运行队列种任意抽取一个进程即可。在 Linux 2.6 的 *O*(1)算法中，schedule( )函数负责找到最高优先级进程，并进行进程切换工作。因此在完成随机调度算法中，可以通过修改 schedule( )函数，将其选择最高优先级进程的代码修改成随机选择进程。

在 schedule( )函数中，可以找到如下代码：

```
array = rq->active;
…
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list_entry(queue->next, struct task_struct, run_list);
…
```

在这段代码中，array 指向 rq->active，即活跃的优先级数组。array->bitmap 为该优先级数组的位图。在 15.2 节中，介绍过该位图能够帮助调度器在 *O*(1)时间内找到优先级最高的进程。位图中的每一位对应一个优先级链表，如果位图中该位为 0，则说明链表为空；反之说明链表中有处于就绪态的进程。在该段代码中，sched_find_first_bit( )函数将返回 array->bitmap 中第 1 个为 1 的位置，当取到该 idx 时，程序就能够找到该优先级队列，并取出第 1 个进程。

为了实现随机调度，需要对该段代码进行修改。第一个想法是将通过内核提供的随机函数，随机生成一个数字(该数字必须在内核所支持的优先级范围内)，并测试在位图中该位是否被置为 1；如果置为 1，表明找到一个可以运行的进程，调度工作继续进行；如果不为 1，表明找不到可运行进程，生成下一个随机数并再次进行测试。根据这个思路，可实现下面一段代码：

```
get_random_bytes(&idx, sizeof(idx));
idx %= MAX_PRIO;
while ( !test_bit( idx, (void*) &array->bitmap) ){
        idx = (idx+1) % MAX_PRIO;
```

```
}
queue = array->queue + idx;
next = list_entry(queue->next, struct task_struct, run_list);
```

在这个函数中，通过 get_random_bytes( )函数实现随机数生成，并从该随机数所指的位置开始，不断测试是否有可运行进程。使用 get_random_bytes( )需要包含 Linux/random.h 头文件。当将修改后的代码编译后，发现内核不能正常地启动因此这样的修改是有问题的，需要分析可能产生错误的原因。

（2）是否有死循环

仔细查看代码后，发现了代码

```
while ( !test_bit( idx, (void*) &array->bitmap) ){
    idx = (idx+1) % MAX_PRIO;
}
```

有产生死循环的可能。在这段代码中，如果 test_bit( )一直不成功，即队列中没有可运行进程，就有陷入死循环的可能。有没有可能在运行该段代码时，队列中没有可运行进程呢？通过查看 schedule( )函数，得出的答案是不可能，只要运行到这段代码，队列中就一定有可运行进程。在 schedule( )函数中，有如下代码：

```
if (unlikely(!rq->nr_running))
  {
    idle_balance(cpu, rq);
    if (!rq->nr_running) {
        next = rq->idle;
        rq->expired_timestamp = 0;
        wake_sleeping_dependent(cpu);
        goto switch_tasks;
    }
}
```

其中，程序判断当前是否有可运行进程，如果没有可运行进程(rq->nr_running==0)，则程序调用 idle_balance( )将其他 CPU 上的进程搬过来。如果其他 CPU 上也没有可运行进程，则程序会将系统的 idle 进程设置成为下一个运行的进程，并转跳到任务切换 switch_tasks( )执行。如果当前 CPU 上有可运行进程，则程序继续执行以下代码：

```
array = rq->active;
if (unlikely(!array->nr_active)) {
    /*活动优先级数组与过期优先级数组指针对换*/
    schedstat_inc(rq, sched_switch);
    rq->active = rq->expired;
    rq->expired = array;
    array = rq->active;
    rq->expired_timestamp = 0;
```

```
            rq->best_expired_prio = MAX_PRIO;
        }
```

在这段代码中，程序判断当前活跃的优先级队列上是否有可运行进程；如果当前优先级队列上没有可运行进程，则说明当前的过期优先级队列上有可运行进程。在这种情况下，内核需要将过期队列和活跃队列的指针进行交换，保证活跃队列上有可运行进程。

通过对这两段代码观察，当程序运行到本实验中提出的随机调度程序时，就能保证 array[ ] 上一定有可运行进程。因此 while 循环一定可以退出，死循环不是该段代码的错误。

（3）是否与实时进程调度有关

再次回顾背景知识可以发现：Linux 调度算法对于实时进程和普通进程的调度方法是不一样的。在 Linux 2.6 内核下，如果实时进程的调度策略是 SCHED_FIFO，进程可以一直使用 CPU 运行，除非自己出现等待事件或被具有更高优先级的实时进程替代；如果调度策略是 SCHED_RR，当进程时间片耗尽后，使用相同优先级排到原队列的队尾。根据这种策略，以上代码段就有可能产生错误。例如，如果一个进程是 SCHED_FIFO 策略，该进程就会假定：在该进程执行完毕之前不会有低优先级的进程被执行。如果采用随机调度，这个假设就不再成立，因此可能导致进程运行出错。因此需要对实时进程和普通进程分开考虑。

对于实时进程，还采用原来的调度方式；而对于普通进程，则采用新的随机调度算法。修改后的代码为：

```
        idx = sched_find_first_bit(array->bitmap);
        if ( idx>=MAX_RT_PRIO ){
            get_random_bytes(&idx, sizeof(idx));
            idx %= MAX_PRIO;
            while ( !test_bit( idx, (void*) &array->bitmap) ){
                idx = (idx+1) % MAX_PRIO;
            }
        }
```

代码先通过 sched_find_first_bit( )函数找到最高优先级的进程位置，如果 idx<MAX_RT_PRIO 表明目前有实时进程，则忽略执行随机调度的代码段；否则，通过随机调度获得拥有可运行进程的优先级。

编译新代码，内核在启动过程中，还是发生错误，看来还有其他情况。

（4）随机算法是否有错

随机算法是否有错？随机算法中，最重要的是随机数的生成。本实验中，通过 get_random_bytes( )获得随机数，该函数是否有错误？查看 get_random_bytes( )的实现可以发现，该函数也是通过/dev/random 设备产生随机数的。显然，在进程调度过程中，不能使用这样的函数。因此尝试实现一个简单的随机数生成算法。

在该随机算法中，可以利用 jiffies 变量来实现伪随机数，因为 jiffies 变量用于记录时间，每次 schedule( )函数被调用时，该变量都很可能发生变化，它可以作为随机数的一个种子。因

此可以通过它实现如下的随机算法：

```
idx = sched_find_first_bit(array->bitmap);
if ( idx>=MAX_RT_PRIO ){
   int seed = jiffies;
   int mod = MAX_PRIO - MAX_RT_PRIO;
   seed = (seed+7)%mod;
   while ( !test_bit(MAX_RT_PRIO+seed, (void*) &array->bitmap) ){
      seed = (seed+7) % mod;
   }
   idx = MAX_RT_PRIO+seed;
}
```

再次编译修改后的代码，内核成功运行，随机调度算法修改成功。

3. 程序框架

```
/*****sched.c*****/

idx = sched_find_first_bit(array->bitmap);
if ( idx>=MAX_RT_PRIO ){
   int seed = jiffies;
   int mod = MAX_PRIO - MAX_RT_PRIO;
   seed = (seed+7)%mod;
   while ( !test_bit(MAX_RT_PRIO+seed, (void*) &array->bitmap) ){
      seed = (seed+7) % mod;
   }
   idx = MAX_RT_PRIO+seed;
}
```

第16章 存储管理

16.1 实验目的

- 了解 Linux 物理主存管理机制。
- 了解 Linux 进程虚存管理机制。
- 初步掌握 Linux 缺页异常的处理过程。

16.2 背景知识

Linux 是一个多用户、多任务操作系统，在它运行时，多个进程共享系统主存，所有进程要求的主存容量远远大于物理主存的容量，为此采用虚拟主存技术，通过部分对换、部分装入方法让每个进程都有大小 4 GB(x86 平台)的逻辑主存地址空间。引入主存寻址机制后，可以隔离各应用进程，并保护内核不受恶意或无意的破坏，且用户编程不受物理主存大小的限制。Linux 的存储管理包括物理主存管理和进程虚存管理两部分。在对 x86 平台的分段机制作简短讨论后，再对两个部分进行介绍。

16.2.1 x86 的分段机制

在 x86 CPU 中，分段机制是实现虚拟主存的基本单位。在保护模式下，通过硬件地址主存管理单元(Memory Management Unit，MMU)将虚拟地址转换为物理地址的过程如图 16-1 所示，分页机制再将线性地址转换为物理地址。

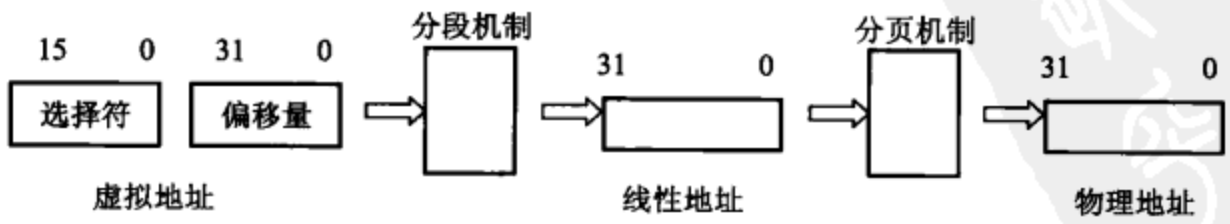


图 16-1 MMU 将虚拟地址转换为物理地址

在 x86 CPU 中，虚拟地址指应用程序编程地址空间中的地址，一般用“选择符：偏移量”的形式来规定，通过选择符可得到段基地址，由段基地址和段内偏移量组成的地址也称为逻辑地址；线性地址是指一段连续且不分段的、由 32 位无符号整数表示的、范围为 0～4 GB 的地

址空间中的一个绝对地址；物理地址指硬件主存提供的物理主存空间中的主存单元地址，用于芯片级主存单元寻址。

16.2.1.1 分段机制

MMU 的分段机制将虚拟地址转换为线性地址，为了实现这种映射，仅仅用段寄存器来确定一个基地址是不够的，为此设计两张表：局部描述符表(Local Descriptor Table，LDT)和全局描述符表(Global Descriptor Table，GDT)，每个描述符占 8 个字节，其中包括基地址 32 位、界限 20 位和属性 12 位。由此可以想到，段寄存器中应该存放索引，索引表示段描述符在描述符表中的位置，因此段寄存器的内容也称选择符，它的 3 个域用来指定：索引值、全局描述符表还是局部描述符表、特权级。

程序中的虚拟地址表示为“选择符：偏移量”形式，通过以下步骤把一个虚拟地址转换为线性地址：

① 在段寄存器中加载段选择符，同时把 32 位地址偏移量加载到某个寄存器，如 ESI 或 EDI 中。

② 根据选择符中的索引值、及全局描述符表还是局部描述符表选项找到描述符，再根据相应描述符中的段地址和段限长，进行合法性检查。如果该段无问题，就取出相应的描述符放入“段描述符高速缓冲寄存器”中。

③ 将描述符中的 32 位段基地址和放在 ESI 或 EDI 等寄存器中的 32 位偏移量相加，形成 32 位线性地址。

寻址过程如图 16-2 所示。

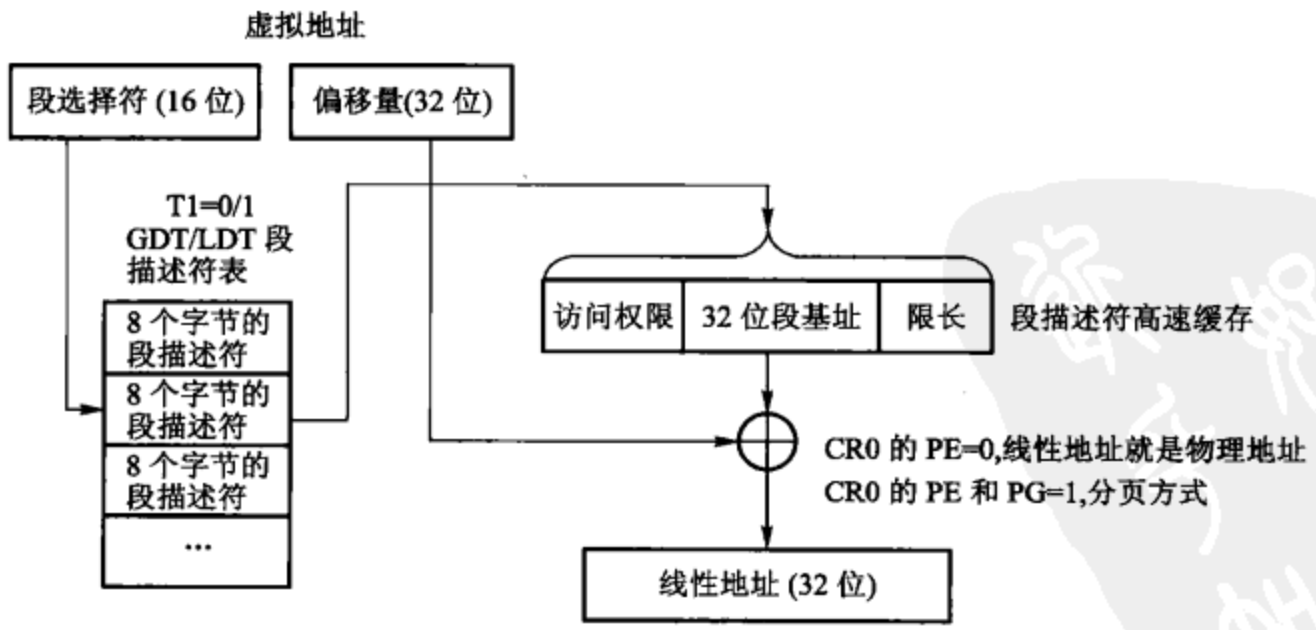


图 16-2 保护模式下的寻址过程

注意，在地址转换过程中，对段进行保护：在一个段内，如果偏移量大于段界限，虚拟地

址将没有意义，系统会产生异常；如果要对一个段进行访问，系统会根据段的保护属性检查访问者是否具有访问权限，如果没有则产生异常。

16.2.1.2 分页机制

分页机制在分段机制之后工作，以完成从线性地址到物理地址的转换。分段机制把虚拟地址转换为线性地址，分页机制再进一步把该线性地址转换为物理地址。如果不允许分页(CR0的最高位置 0)，那么经分段机制转换而来的 32 位线性地址就是物理地址；如果允许分页(CR0的最高位置 1)，就要将 32 位线性地址通过 MMU 地址转换部件转换成物理地址。

16.2.2 物理存储管理

16.2.2.1 管理区和页框

物理主存是系统中十分宝贵的资源。在操作系统的启动过程中，内核需要对主存进行初始化，并对其建立相应的管理数据结构。初始化完成之后，物理主存的分布如图 16-3 所示。

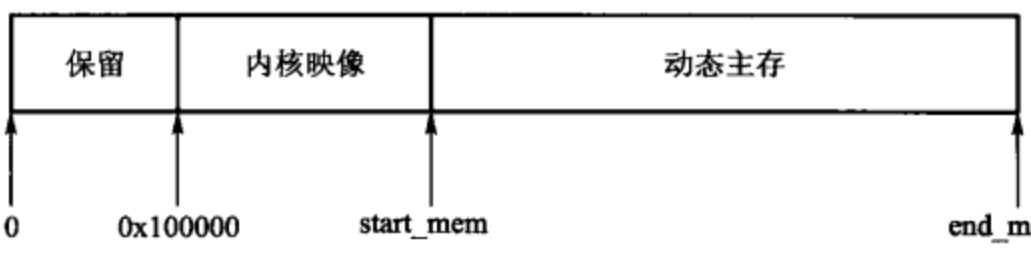


图 16-3 物理主存分布

从图 16-3 中可知，在初始化结束后，系统中可分配的主存是从 start_mem 开始，到 end_mem 结束。

在 Linux 中，主存的分配与管理以页框为单位，一个页框的大小为 4 KB。在理想情况中，物理主存中的每个页框都应该一样，使用上不应该有限制，但在实际的计算机体系结构中，页框的使用方式是受限制的。例如，在 Intel 32 体系结构中，ISA 总线的直接存储器存取(DMA)有严格限制：只能对主存的前 16 MB 寻址，为了应对这样的限制，Linux 把物理存储器分成以下 3 个管理区。

- ZONE_DMA：包含低于 16 MB 的存储器页框，用于 DMA 方式访问主存。
- ZONE_NORMAL：包含高于 16 MB 且低于 896 MB 的存储器页框，直接被内核映射。
- ZONE_HIGHMEM：包含高于 896 MB 的存储器页框，不能直接被内核映射。

需要申请一个页框时，必须指明是从主存哪个管理区申请，这样就可以解决刚才提及的问题。

内核需要对每个页框的状态进行管理，为每个页框分配一个 struct page 结构，该结构记录页框的使用状态。在系统初始化时，所有的 page 结构都被统一存放在 mem_map 数组中，整个

数组就代表系统中的全部物理页框，数组下标就是物理页框的序号。该结构定义如下：

```
struct page {
    unsigned long flags;                    /*表示页框状态的标志位*/
    atomic_t _count;                        /*引用计数，-1 表示页框空闲*/
    atomic_t _mapcount;                     /*引用改页框的 PTE 数量*/
    struct address_space *mapping;          /*页框所在的线性空间*/
    pgoff_t index;          /*表示页中的数据在磁盘中的偏移量，或表示换出页的标识符*/
    struct list_head lru;                   /*LRU 队列指针*/
    void *virtual;                          /*页框所在的线性空间*/
};
```

16.2.2.2 页框分配

Linux 采用伙伴算法进行页框的分配与释放，用以解决页框的“外部碎片”问题：

(1) 设计以 page 结构为数组元素的 mem_map[ ]数组；

(2) 设计以 free_area_struct 结构为数组元素的 free_area 数组：

```
struct free_area_struct {
  struct page *next;
  struct page *prev;
  unsigned int *map;
};
static free_area_struct free_area[NR_MEM_LISTS];
```

该数组记录空闲主存页框，共 11 个元素(NR_MEM_LISTS 默认值)。对于每个管理区，都要把该管理区中的所有空闲页框分成 11 个链表，每个链表中的一个块包含 2 的幂次个页框，这种块称为“页块”，即大小为 2^0=1～2^{10}=1 024，第 i 个元素代表 mem_map 数组中第 i 组空闲块链表头。当内核需要页框时，要先进行申请；页框使用完毕后，再进行释放。每个链表在内核中用一个 struct free_area 表示，在管理区的数据结构 struct zone 中，free_area 成员是包含这些链表的数组。

(3) 位示图(bitmap)，共 11 个，每个空闲页框块数的链表对应一张，用二进制表示主存页框使用情况，第 0 组的每一位表示单个页框的使用情况，为 1 表示该页框正在使用，为 0 表示空闲；第 1 组的每一位表示相邻的两个页框的使用情况，如果其中有一个位置 1，表示所对应的两个页框正在使用，依此类推；第 i 组中的每一位表示相邻的 2^i 个页框被使用的情况。直接向伙伴系统申请空间和释放空间的函数是 alloc_pages( )和 free_pages( )。

通过一个简单的例子来说明该算法的工作原理。假设用户需要请求一个大小为 128 个页框的页块。算法首先在 128 个页框的链表中检查是否有一个空闲块。如果没有这样的块，算法会查找下一个更大的空闲块，即大小为 256 个页框的块。如果存在这样的块，就把 256 个页框的块分成两等份，一半用于满足用户请求，另一半插入到 128 个页框的链表中。如果在 256 个页框的块链表中也没有找到空闲块，就继续找更大的块——512 个页框的空闲块。如果这样的空

闲块存在，内核就把 512 个页框的块分成两个各有 256 个页框的块，其中一个插入 256 页框的块链表；另一个再分成两个各有 128 页框的块，其中一个插入 128 页框的块链表，一个用于满足用户请求。页框的释放过程是以上过程的逆过程，它将分解的动作变成合并动作。在内核中，分配和释放页框的函数为__rmqueue( )和__free_pages_bulk( )，用户可以参考代码深入理解。

16.2.2.3 页面换出

当物理页框不够用时，Linux 存储管理系统必须释放部分物理页框，把其中的页面写到交换空间。内核态后台守护线程 kswapd( )专门完成这项任务，它每隔 10 s 被激活一次，负责把页面换出到交换空间，保证系统中有足够的空闲页框，确保存储管理系统高效运行。

kswapd( )在系统初启时由 init( )创建，然后调用 init_swap_timer( )函数设置时间间隔，并马上转入睡眠。当定时器时间到后，kswapd( )被 process_timeout( )函数唤醒，它首先查看系统中空闲页面 nr_free_pages 是否变得太少，利用两个控制变量 free_pages_high 和 free_pages_low 进行判断。如果空闲页框数小于 free_pages_high，就要有页面被交换出去；如果空闲页框数小于 free_pages_low，kswapd( )不仅换出部分页面，还要把睡眠时间减为平时的一半，更频繁地换出页面；当空闲页框数大于 free_pages_low 时，睡眠时间又会恢复原样。页框回收利用由 try_to_free_pages( )函数实现，它调用的主要函数是 shrink_caches( )，其中又要调用 kmem_cache_reap( )减少 slab 机制管理的空闲块，内核数据结构占用的其他空闲块一并回收，然后，循环地依次从 3 条途径缩减系统使用的物理页框：

① 缩减 page_cache 和 buffer_cache，调用 shrink_mmap，采用 clock 算法实现。

② 换出 System V 共享主存占用的页面，调用 shm_swap 实现。

③ 换出或丢弃进程占用的页面，调用 swap_out，并采用 clock 算法实现。

如果系统中空闲页框数低于 free_pages_low，kswapd( )将在下次运行之前释放 6 个页面，否则只释放 3 个页面。

16.2.3 进程虚拟存储管理

16.2.3.1 页表

Linux 进入保护模式以后，进程对主存的访问都必须采用虚拟地址，不能使用物理地址。在 x86 上，虚拟地址空间的大小是 4 GB。4 GB 被分成两个部分，0～3 GB 为进程的私有空间，称为用户空间，进程可直接访问；3～4 GB 供内核使用，称为内核空间，存放内核的代码和数据，它们虽然可被所有应用进程共享，但用户态进程只能通过中断或函数，经过 CPU 模式转换，从用户态切换到内核态执行和完成相应功能。

进程虽然用虚拟地址对主存进行访问，但是最终硬件会将虚拟地址转换成物理地址再进行存取，MMU 负责地址转换工作，进程虚实地址转换工作的数据结构就是页表。页表机制是虚拟主存管理的核心，32 位线性地址分成 3 个部分：页目录表项，置于高 10 位，存放页目录的

索引；页表项，占据中间 10 位，存放页表的索引；偏移量，占据低 12 位，表示在 4KB 的页框中的偏移量。

每个进程都有一个页目录表，当它投入运行时，寄存器 CR3 指向该页目录的基址。图 16-2 已经获得线性地址，图 16-4 是从线性地址到物理地址的映射过程。

① 从 CR3 取得页目录基址。页目录用一个物理页框存储，用来保存页表的基址。每个数据项占 4B，因而页目录表有 1 024 个数据项。

② 以页目录项为索引，在页目录中找到某个页表的基址。页表也使用一个物理页框存储，用来保存物理页框号。每个数据项占 4 B，因而页表也有 1 024 个数据项。

③ 以页表项为索引，在页表中找到相应的物理页框号。

④ 物理页框号加上偏移量得到对应的物理地址。

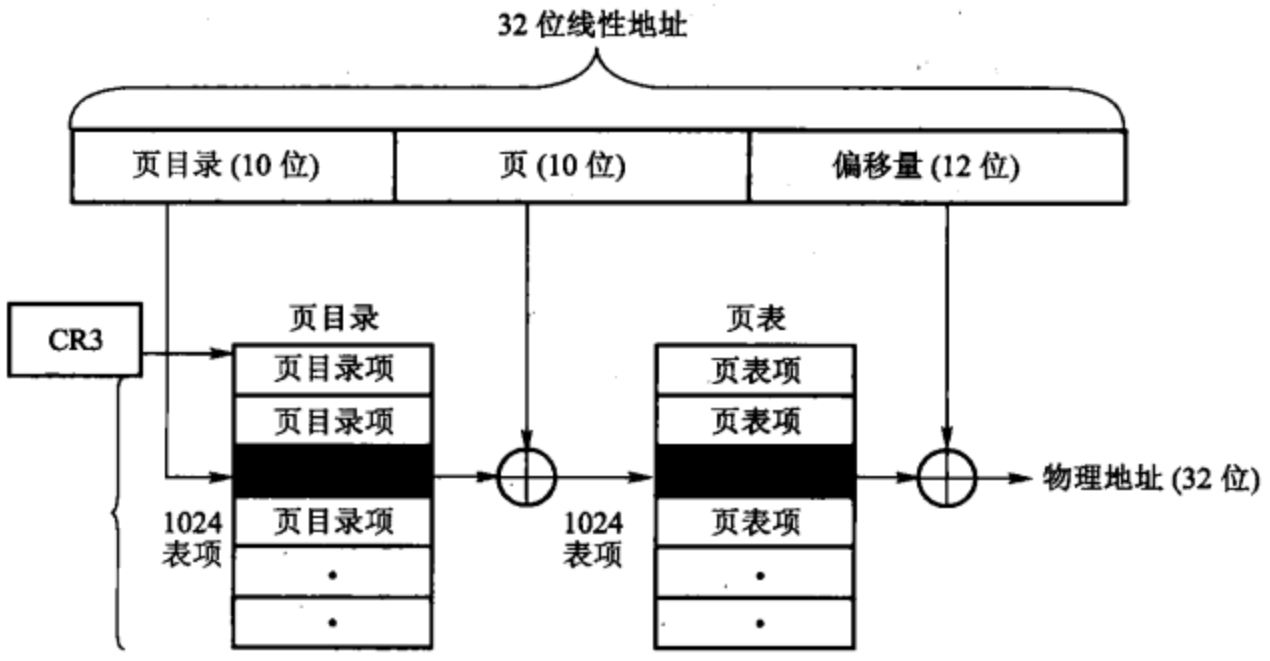


图 16-4 Linux 页式地址转换过程

因为每个物理页框的大小总是 4 KB，所以页目录项和页表项一共只需要 20 位，占用物理地址的 12～31 位，还有 12 位用作标志位，可以用于控制或其他目的。这些标志位中比较重要的有以下几个。

- 存在位：表明项对应的物理页是否加载到主存。
- 读/写位：表明该页是只读还是可写，起保护作用。
- 用户/系统位：选择用户级访问许可或内核级访问许可。
- 访问位：表明该页是否被访问过。
- 脏标志位：表明该页是否被写过。
- 直写位：置位表明页面 cache 采用“直写”(既写主存也写缓存)，否则回写缓存。

这几个标志位对于完成虚拟主存管理十分重要。例如，进程需要访问的一个页面不在物理主存中，而存储在磁盘上，则该页的页表项的存在位为 0；当进程访问该页时，会触发一个缺

页异常，Linux 便将该页载入物理主存，并修改页表项，进程便可以访问在物理主存中的页面了。在 Linux 中，所有进程共享同一个内核空间，因此所有进程的 3～4 GB 的页表项是一致的。

16.2.3.2 进程地址空间信息

Linux 内核将与进程地址空间有关的所有信息都存放在 struct mm_struct 数据结构中，在每个进程的 task_struct 结构中包含一个指向 mm_struct 结构的指针。mm_struct 结构中与主存管理有关的成员如下：

- 指向页目录表和局部描述符表的指针。
- 分配给进程的页框数、进程地址空间含有的总页数及被锁定页面数。
- 共享同一个 struct mm_struct 结构的进程数目。
- 指向进程虚存区链表的表头指针，该链表中的每个元素都是 vma(struct vm_area_struct)结构。
- 虚存区的 AVL 树。进程拥有的虚存区个数，最多不超过 65 536 个。
- 对页表操作的自旋锁。
- 代码段、数据段的起始地址和结束地址。
- 未初始化数据段和堆栈的起始地址和结束地址。

16.2.3.3 虚存区

进程实际可以使用的地址空间是 0～3 GB，但绝大多数程序不会全部用满，而且组成进程空间的各个部分性质并不相同，如代码段是只读的、数据段是可读可写的。在 Linux 中，这样的一个段被称为虚存区(virtual memory area，VMA)，一个进程通常有若干个虚存区，而且两个虚存区绝不会重叠，但却可以不连续。内核将进程的每个虚存区作为一个单独的主存对象管理，每个虚存区都拥有一致的属性，如访问权限等，另外，相应的操作也都一致。虚存区是由起始地址、长度和一些存取权限来描述的。为了效率起见，起始地址和虚存区的长度都必须是 4 096 的整数倍，以便每个线性区所识别的数据完全填满分配给它的页框。一个进程的所有虚存区被串在一个链表中，在虚存区较多时，通过链表搜索会较慢，为了加快搜索速度，内核把它们组织成一个 AVL 树。表示虚存区的数据结构为 struct vm_area_struct：

```
struct vm_area_struct {                    /*虚存区结构*/
struct mm_struct *vm_mm;                   /*虚存区所在的 mm_struct 指针*/
unsigned long vm_start;                    /*虚存区的起始地址*/
unsigned long vm_end;                      /*虚存区的结束地址*/
pgprot_t vm_page_prot;                     /*该虚存区存取权限*/
unsigned short vm_flags;                   /*该虚存区操作标志*/
struct vm_area_struct * vm_next;           /*按地址降序排列的链接下一虚存区的指针*/
unsigned long vm_avl_height;               /*虚存区的 AVL 树的高度*/
struct vm_area_struct *vm_avl_left;        /*虚存区的 AVL 树的左节点*/
```

```
struct vm_area_struct *vm_avl_right;          /*虚存区的 AVL 树的右节点*/
struct vm_operations_struct *vm_ops;          /*虚存区上定义的操作集* /
struct vm_area_struct *vm_next_share;         /*共享同一文件的下一个虚存区*/
struct vm_area_struct **vm_pprev_share;       /*共享同一文件的上一个虚存区*/
unsigned long vm_pte;                         /*pte 表*/
unsigned long_vm_pgoff;                       /*vm_file 中的偏移量*/
struct file *vm_file;                         /*该虚存区被映射的文件，如果有的话*/
unsigned long vm_raend;                       /*存放预读信息*/
void *vm_private_data;                        /*共享主存*/
…
}vma;
```

进程虚拟地址空间中的访问权限就是其虚存区的访问权限，权限标志放在虚存区的 vm_flags 中，定义的页面访问权限及有关虚存区所在页的全部信息有：只读页、可写页、可执行页、可共享页、虚存区的页面被锁住、虚存区可作共享主存、虚存区可向下增长、虚存区可向上增长、虚存区映射一个不可写文件、虚存区映射一个可执行文件、虚存区不能换出、虚存区是 I/O 设备的地址空间、页被连续访问、页被随机访问等。页的存取权限可以任意组合，当产生缺页异常时，内核将根据标志来查找发生缺页的原因，以实施不同方式处理。

每个虚存区可能有不同的来源，有的可能来自可执行映像，有的可能来自共享库，有的可能是动态分配的主存区。不同来源的虚存区可能会有不同的操作，因此 Linux 通过虚存区结构体中的 vm_ops 成员来指向与指定虚存区相关的操作函数表，其中的函数可用于操作虚存区。vm_area_struct 作为通用对象代表了任何类型的虚存区，而操作函数表描述针对特定的对象的特定操作方法。vm_operation_struct 结构的定义为：

```
struct vm_operations_struct {
  void(*open)(struct vm_area_struct *area);                  /*打开映射的虚存区*/
  void(*close)(struct vm_area_struct *area);                 /*关闭映射的虚存区*/
  void (*unmap) (struct vm_area_struct *area,                /*去映射并释放虚存区*/
                 unsigned long,size_t);
  void (*protect) (struct vm_area_struct *area,              /*设置和检查虚存区的保护权限 */
                 unsigned long,size_t,unsigned int newprot);
  struct page *(*nopage)(struct vm_area_struct *area,        /*处理缺页异常*/
                         unsigned long address, int write_access);
  int (*swapout) (struct vm_area_struct *,struct page *);    /*换出虚存区*/
  pte_t (*swapin) (struct vm_area_struct *,unsigned long,    /*换入虚存区*/
                                           unsigned long);
};
```

进程中的每个页面都必定属于某个虚存区，但是属于虚存区中的页面不一定在页表中有数据。因为只有在需要用到该页面时才会申请页框并填充相应的页表项，这是通过缺页异常实现的。

对虚存区进行处理的底层函数包括：

- find_vma( )：搜索给定地址区最近的虚存区。
- find_vma_prev( )：类似 find_vma( )，但返回值不同。
- find_vma_intersection( )：搜索与给定地址区相重叠的虚存区。
- get_unmapped_area( )：搜索一个空闲的地址区。
- insert_vm_struct( )：向虚存区简单链表插入虚存区。
- merg_segments( )：把给定地址区内的虚存区合并在一起。

16.2.3.4 进程虚存区映射

进程的虚拟地址空间中包括一组主存对象，实际上这些对象代表后备存储(如磁盘交换区、文件系统的文件)与进程地址空间之间的映射，每个这样的映射被称为一个虚存区，或者说在虚存系统中，一个虚存区就是一个主存对象(也称映像)，它是对应于一个文件、共享主存、交换设备或其他特殊对象而建立的一段连续的虚拟地址区。在 Linux 内核中，创建并初始化一个虚存区映像是由被称为主存映射的函数 mmap( )来完成的，通过它来实际执行后备存储与进程地址空间的映像。

mmap( )函数通过内核函数 do_mmap( )为当前进程在主存生成一个 vm_area_struct 结构体的虚存区并连接到对应链表上。如果该虚存区填充的是指令代码并标记为可执行，系统就跳转到代码段的首地址处开始执行。因为只有代码的一少部分被调入主存，在 MMU 进行虚实地址转换时会产生缺页异常，然后系统通过异常调用执行 do_page_fault( )函数来完成调入可执行文件的剩余页面。也可以这样讲，mmap( )为可执行文件创造了一个可执行的条件，即为可执行文件分配一段进程的虚拟地址空间，这个进程的地址空间称为虚存区。

传递给 do_mmap( )的参数有：需要建立虚拟映射的文件指针、具体映射的相对于文件起始地址的偏移量、映射的地址空间长度、指定虚存区包含页面的访问权限(可读、可写、可执行和不可访问)等。

释放进程虚存区使用 mummap( )函数，它调用 do_mummap( )内核函数从指定进程的地址空间中删除一个虚存区。该函数的参数为虚存区的起始地址和长度。要释放的区间并不总是对应一个虚存区，它可以是虚存区的一部分，也可以删除两个或多个虚存区。该函数执行可分为两步，首先，从进程所拥有的虚存区链表中，删除与指定地址区相重叠的所有虚存区；其次，更新进程页表，并重新调整虚存区链表。

16.2.3.5 缺页异常

进程执行时，CPU 访问的地址是用户空间的虚拟地址，Linux 仅仅把用户空间的少量页面加载到主存，当访问的虚存页面尚未加载到物理主存时，CPU 将产生缺页异常。缺页异常产生的原因有以下几种。

- 编程错误。例如用户进程访问内核空间，在这种情况下，Linux 将向进程发送一个信号

并终止进程的运行。

• 虚地址有效，但是所对应的页当前不在物理主存中。在这种情况下，操作系统必须从磁盘或交换文件中将页面加载到物理主存。

• 程序试图写一个只读页面。如果进程写的是本进程空间的一个页面，说明是编程错误，操作系统将终止进程运行。如果进程写的是一个共享页，则操作系统需要将该页复制一份，完成“写时复制”。

在 Linux 中，处理缺页异常的函数是 do_page_fault( )，当发生缺页异常时，该函数将被调用，它的执行过程为：

① 首先通过 CR2 寄存器得到发生缺页异常的地址。

② 检查异常是否发生在中断或内核线程中，如果是，则进行出错处理。

③ 检查该地址是否属于进程的某个虚存区中。如果不属于某个虚存区，则检验该地址是否处于栈的合理可扩展区间。栈虚存区的 vm_flags 被设置成 VM_GROWSDOWN，即该虚存区是可向下扩展的。Linux 通过缺页异常来扩展栈。一旦用户态产生异常的地址正好处于栈 vm_start 前面的合理位置，则调用 expand_stack( )函数扩展该区间。

④ 根据错误码确定下一个步骤。如果错误码的值表示为写错误，则检查该区间是否允许写。若不允许，进行出错处理；如果允许，就进行“写时复制”。如果错误码的值表示为页面不存在，则进行按需分页。

“写时复制”的处理流程为：首先改写对应页表项的访问标志位，表明其刚被访问过，这样在页面调度时，该页面就不会被优先考虑淘汰。如果该页框目前只被一个进程单独使用，则只需把页表项设置成可写。如果页框被多个进程共享，则申请一个新物理页框并将其标记成可写并复制原来物理页框的内容，更改当前进程相应的页表项，同时原来物理页框的共享计数减 1。

按需分页的处理流程是：首先确认产生页面不在物理主存中的原因。一个原因是页面从来未被进程访问，对于这种情况，页表项的值全为 0。另一个原因是该页面被进程访问过，但是目前已被写到交换区，对于这种情况，页表项的存在标志位为 0，其他位被用来记录页面在交换分区中的信息。第一种情况下，又要区分该页面是否是映射到一个文件，如果所属区间的 vm_ops->nopage 不为空，则表示该区间映射到一个文件，并且 vm_ops->nopage 指向加载页面的函数，此时调用此函数加载该页面；如果 vm_ops 或 vm_ops->nopage 为空，则应该调用 do_anonymous_page( )申请一个页面。第二种情况调用 do_swap_page( )函数从交换区调入该页面。

16.2.3.6 与主存相关的函数

进程使用的与主存相关的函数有以下几种。

• fork( )：创建进程，为子进程复制一个虚拟地址空间。

• clone( )：创建线程，多线程共享虚拟地址空间。

- vfork( )：创建进程，父进程出让虚拟地址空间给子进程。
- exec( )：执行新程序，抛弃原来的虚拟地址空间，并根据可执行文件内容生成相应的新虚拟地址空间。
- exit( )：进程终止，销毁进程虚拟地址空间。
- mmap( )：映射虚拟主存页，扩展进程虚拟地址空间。
- munmap( )：去除主存页映射。
- shmget( )：创建一个共享主存区。
- shmmat( )：连接共享主存区。
- shmdt( )：释放共享主存区。
- mprotect( )：设置主存映射保护。
- mlock( )：主存页面加锁。
- munlock( )：主存页面解锁。
- malloc( )：从堆里分配主存。
- calloc( )：类似 malloc( )。
- free( )：释放分配的主存。
- brk( )：修改堆的大小。
- sbrk( )：类似 brk( )。

16.2.4 slab 分配器

16.2.4.1 slab 的结构

分配和释放主存是内核中最频繁和最普遍的操作，在很多情况下，需要的主存量远远小于页框大小，如为 inode、vma、task_struct 等数据结构分配空间。此外，无法预知运行中各种不同数据结构对主存的需求量。为了更经济地使用和全局性地动态控制主存资源，Linux 引入了 1994 年在 SunOS 操作系统中首创的 slab 主存分配器，它比传统分配器有更好的性能和主存利用率。slab 是内核的主存空间与页面级分配接口，为经常使用的数据结构建立专用的高速缓存(cache)，空间的申请与释放都通过 slab 分配器来管理；每个高速缓存存放不同类型对象(object)，而每种对象类型对应一个高速缓存；每个高速缓存被划分为一串 slab，每个 slab 由一个或多个(最多 32 个)物理上连续的页框组成，slab 的大小因对象而异，初始化时将计算出最合适的大小；每个 slab 包含若干个同类型对象，这些对象可被缓存数据结构，例如由一个页框组成的 slab，大约可以放 8 个 inode 对象。

每个 slab 可能处于 3 种状态之一：满、半满或空，并按该次序排列。满的 slab 没有空闲对象(对象已被分配并缓存数据结构)；空 slab 没有分配出任何对象(所有对象都空闲)；半满的 slab 有些对象已分配出去，有些对象还空闲着。当内核的某一部分需要一个新对象时，先从半满 slab

中进行分配，如果没有半满 slab，就从空 slab 中进行分配，如果没有空 slab，这时就要向伙伴系统申请并创建一个 slab。由于满的 slab 已没有空闲对象，这种策略能减少主存碎片。

高速缓存的控制结构为 kmem_cache_t，每种高速缓存都由一个 kmem_cache_t 结构来管理。kmem_cache_t 是 kmem_cache_s 类型，用来描述高速缓存，如第 1 个空闲 slab 对象指针、第 1 个和最后一个 slab 对象指针、slab 对象大小、一个 slab 中的对象个数、一个 slab 中包含的连续页框数的对数、高速缓存的名字和下一个高速缓存指针等。

描述高速缓存中每个 slab 使用的数据结构为 kmem_slab_s 类型，该结构包含 3 个链表：slabs_full、slabs_partial 和 slabs_empty，这些链表包含高速缓存中所有 slab：

```
struct kmem_slab_s {
  struct list_head     list;          /*逻辑上分满、半满或空链表*/
  unsigned long        s_colouroff;   /*slab 着色的偏移量、即 slab 块之间的距离*/
  unsigned long        s_magic;       /*检查 slab 状态一致性的魔数*/
  void                 *s_mem;        /*slab 中的第 1 个对象的地址*/
  unsigned int         s_inuse;       /*已分配的对象数*/
  kmem_bufctl_t        *s_freep;      /*第 1 个空闲对象指针*/
}kmem_slab_t;
```

系统建立变量名为 cache_cache 的高速缓存池，其中存放着类型为 kmem_cache_t 的对象，全局 cache_cache 高速缓存池的结构如图 16-5 所示。

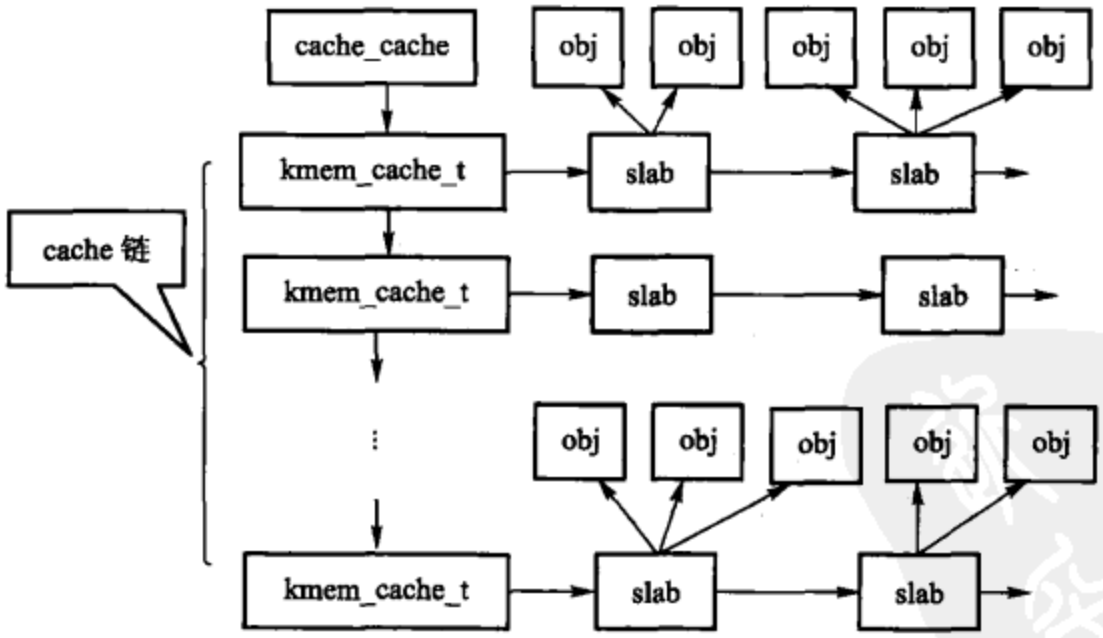


图 16-5 全局 cache_cache 高速缓存池结构

除这些专用对象的 cache 外，Linux 还提供 13 种通用高速缓存，其存储对象的大小分别为 32 B、64 B、128 B、256 B、512 B、1 KB、2 KB、4 KB、8 KB、16 KB、32 KB、64 KB 和 128 KB，这些高速缓存用来满足特定对象之外的普通主存需求，大小呈 2 的幂数增长。

16.2.4.2 slab 操作

1. kmem_getpages( )与 kmem_freepages( )

这两个函数是 slab 与页框级分配器的接口，当 slab 分配器要创建一个新的 slab 或 cache 时，通过 kmem_getpages( )向内核提供的伙伴算法来获得一组连续的页框。如果释放分配给 slab 分配器的页框，调用 kmem_freepages( )函数。

2. kmem_cache_create( )

该函数给一个对象空间分配一个专用 cache，它从 cache_cache 高速缓存池中得到一个空闲 kern_cache_t 对象，对成员进行初始化，确定最佳的 slab 构成，包括每个 slab 由几个页框组成，包含几个对象，以及安排 slab 控制结构的存放位置。

3. kmem_cache_alloc( )与 kmem_cache_free( )

当需要分配一个拥有专用高速缓存的对象时，通过 kmem_cache_alloc( )函数完成，如分配一个 task_struct、mm_struct、vm_area_struct、inode、dentry、或 file。申请到的对象不再使用时通过 kmem_cache_free( )函数释放。

4. cache_grow( )与 cache_reap( )、kmem_cache_destroy( )与 kmem_cache_shrink( )

kmem_cache_create( )函数用于建立所需的专用高速缓存，但是此时高速缓存为空。slab 的创建则要等到 kmem_cache_alloc( )函数被调用时，发现高速缓存中无空闲对象供分配，此时通过 cache_grow( )向伙伴系统申请一个 slab 空间，初始化 slab 中的各个对象，每隔一定时间 cache reap( )函数被调用回收对象全空闲的 slab。kmem_cache_destroy( )用于销毁 cache，回收 slab 占用的空间；kmem_cache_shrink( )用于收缩 cache。

5. kmalloc( )与 kfree( )

对于不频繁的主存请求操作，可通过通用 cache 来处理，通用 cache 中的对象大小，呈 2 的整次幂，这两个函数分别用来向通用高速缓存池申请和释放空间。

16.2.4.3 slab 应用例子

考察一个实例，将 slab 分配器应用于进程描述符 task_struct 结构。首先，内核用一个全局变量存放指向 task_struct 高速缓存的指针：kmem_cache_t *task_struct_cache；在系统初始化时，执行 fork_init( )函数：

```
static kmem_cache_t *task_struct_cachep;
void_init_ fork_init(unsigned long mempages)
{ /*mempages 为物理主存大小/PAGE_SIZE*/
 /*创建分配 task_struct 对象的 slab*/
 task_struct_cachep ＝ kmem_cache_create( " task struct " , sizeof(struct task_struct),
                          ARCH_MIN_TASKALIGN,SLAB_PANIC,NULL,NULL);
 /* THREADS_SIZE=2*PAGE_SIZE，PAGE_SIZE=2^12 */
 max_threads = mempages/(THREAD_SIZE/PAGE_SIZE)/8;
 /*启动系统时最少要允许 20 个线程*/
```

```
    if(max_thread<20)
       max_threads=20;
    /*每个用户的进程总数不能多于一半物理主存*/
    init_task.rlim[RLIMIT_NPROC].rlim_cur = max_threads /2;
    init_task.rlim[RLIMIT_NPROC].rlim_max = max_threads /2;
}
```

该函数初始化线程数，并创建一个名为 task_struct 的高速缓存，进程创建时 task_struct 对象从这里分配空间。kmem_cache_create( )的第 1 个参数为高速缓存名字，第 2 个参数为对象大小，第 3 个参数为高速缓存中存放的第一个对象的偏移量(在 ARCH_MIN_TASKALIGN 个字节的地方)，确保页内特定对齐；标志是为高速缓存的特殊行为而设置的，例如若分配失败，slab 分配器就调用 panic( )函数，SLAB_PANIC 标志用在这儿是因为这是系统操作必不可少的高速缓存；构造与析构函数通常在高速缓存中不用，这里置两个 NULL。

每当进程调用 fork( )时，会创建一个新的进程描述符，这是在 dup_task_struct( )中完成的，而该函数会被 do_fork( )调用：

```
struct task_struct *tsk;
tsk = kmem_cache_alloc(task_struct_cachep,GFP_KERNEL);
if(!tsk)
return NULL;
```

进程执行完后，如果没有子进程在等待，其进程描述符就会被释放，并返回给 task_struct_cachep slab，这是在 free_task_struct( )中执行的：

```
kmem_cache_free(task_struct_cachep, tsk);
```

由于进程描述符是内核的组成部分，时刻都要使用，因此 task_struct_cachep 高速缓存绝不会被销毁。

在系统初始化过程中将执行 proc_caches_init( )函数，它给进程的各种资源管理结构创建能分配对象空间的 slab，以备使用：

```
kmem_cache_t *signal_cachep;        /*信号结构 cache*/
kmem_cache_t *sighand_cachep;       /*信号处理结构 cache*/
kmem_cache_t *files_cachep;         /*文件结构 cache*/
kmem_cache_t *fs_cachep;            /*文件系统结构 cache*/
kmem_cache_t *vma_area_cachep;      /*虚拟主存区结构 cache*/
kmem_cache_t *mm_cachep;            /* 虚拟主存结构 cache*/
```

16.3 实 验 内 容

实验 统计系统和单个进程的缺页次数

1. 实验说明

修改系统的缺页异常处理程序，使之能够记录缺页次数：一是系统缺页次数，二是单个进

程缺页次数，并提供新的系统调用供用户查询缺页次数。

2. 解决方案

(1) 统计方法

当每次发生缺页时，缺页中断服务内核函数 do_page_fault( )都要被调用，所以可以认为执行该函数的次数就是系统发生缺页的次数。需要修改 do_page_fault( )，使它在每次被调用时对一个计数器进行自增，该计数器的值便是整个系统发生缺页的次数。为了统计进程发生缺页的次数，需要在进程的进程控制块中添加一个成员，用于记录进程发生缺页的次数。调用 do_page_falut( )时，函数通过 current 宏获得当前进程的进程控制块，并将其中用于记录缺页次数的成员进行自增。

(2) 记录全局缺页次数

先需要添加一个全局变量 global_pf 作为计数变量，将该变量申明在 include/Linux/mm.h 文件中：

```
extern unsigned long volatile global_pf;
```

在 arch/i386/mm/fault.c 中定义该变量：

```
unsigned long volatile global_pf;
```

在定义好之后，修改 arch/i386/mm/fault.c 中的 do_page_fault( )，global_pf 在该函数的开始时自增：

```
fastcall void_ _kprobes do_page_fault(struct pt_regs *regs, unsigned long error_code) {
    global_pf++;
    …
}
```

随着 do_page_fault( )每一次被调用，global_pf 都会加 1，系统发生的缺页次数也就统计好了。

(3) 记录进程缺页次数

要记录进程的缺页次数，首先要在进程的进程控制块中添加一个成员 pf。进程控制块 task_struct 定义在文件 include/Linux/Sched.h 中：

```
struct task_struct {
    …
    unsigned long pf;
}
```

记录进程缺页次数与记录系统缺页次数类似；在定义好之后，修改 arch/i386/mm/fault.c 中的 do_page_fault( )函数，当前进程的 pf 在该函数开始时自增：

```
fastcall void_ _kprobes do_page_fault(struct pt_regs *regs, unsigned long error_code) {
    global_pf++;
    current->pf++;
    …
}
```

统计进程缺页次数与统计系统缺页次数有一点不同在于：在进程创建时需要将进程控制块中的 pf 设置为 0。在 Linux 中，进程创建是采用 fork( )操作，在进程创建过程中，子进程会将父进程的进程控制块复制一份。实现该复制过程的函数是 kernel\fork.c 文件中的 dup_task_struct( ) 函数，修改该函数将子进程的 pf 设置成 0：

```
static struct task_struct *dup_task_struct(struct task_struct *orig) {
    …
    *tsk = *orig;
    tsk->pf = 0;
    …
}
```

完成这几个步骤就可以实现对当前进程的缺页次数的统计。

(4) 添加系统调用

为了能够读出系统缺页次数和当前进程缺页次数，需要添加两个系统调用：sys_global_pf( ) 和 sys_pf( )。添加系统调用的方法可以参见系统调用相关章节。这两个系统调用只要将 global_pf 和当前进程控制块中的 pf 值返回即可：

```
int sys_global_pf( ) {
    …
    return global_pf;
}

int sys_pf( ) {
    …
    return current->pf;
}
```

3. 程序框架

完成添加系统调用后，可以通过编写测试程序测试新添加的系统调用。在使用新的系统调用时需要用系统提供的_syscall0 宏将展开新的系统调用，该宏需要引用<Linux/unistd.h>文件。测试程序结构如下：

```
#include <linux/unistd.h>
_syscall0(int, global_pf)
_syscall0(int, current_pf)
int main( )
{
    …
    long gpf = global_pf( );
    long cpf = current_pf( );
    …
}
```

```
write(outfile,buf,charnum)
close(infile);
close(outfile);
```

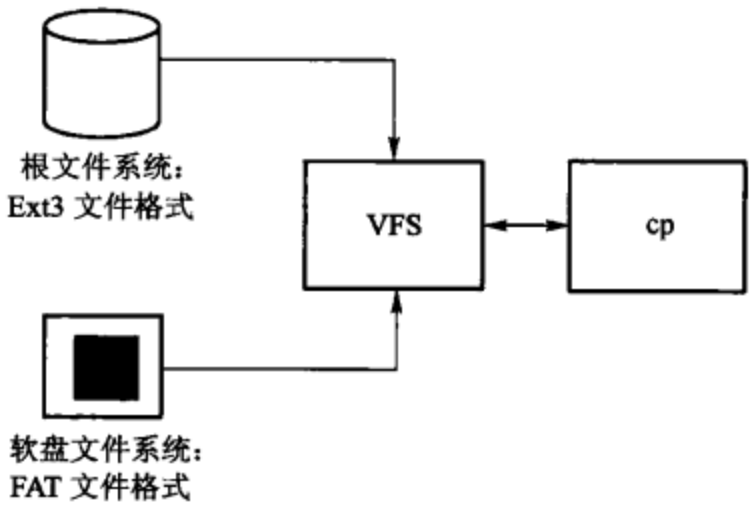


图 17-1 跨文件系统的文件复制示意图

从上述的 cp 代码片段可知，cp 操作调用 open( )、read( )、write( )和 close( )等函数。以 write( )为例，在用户空间执行的操作：

```
write(outfile, buf, charnum);
```

将 buf 指针指向的、长度为 charnum 字节的数据写入文件描述符 outfile 对应的文件的当前位置。该用户操作首先被一个通用内核函数 sys_write( )处理，sys_write( )会找出 outfile 所在文件系统实际给出的是哪个写操作，然后再执行该操作。实际的写操作是具体文件系统的一部分，数据最终通过该写操作被写入磁盘介质中去。图 17-2 描述了上述操作的流程。可以看到，一方面函数是 VFS 提供给用户空间的接口，另一方面调用文件系统的具体实现方式来达到这个文件系统希望完成的动作。

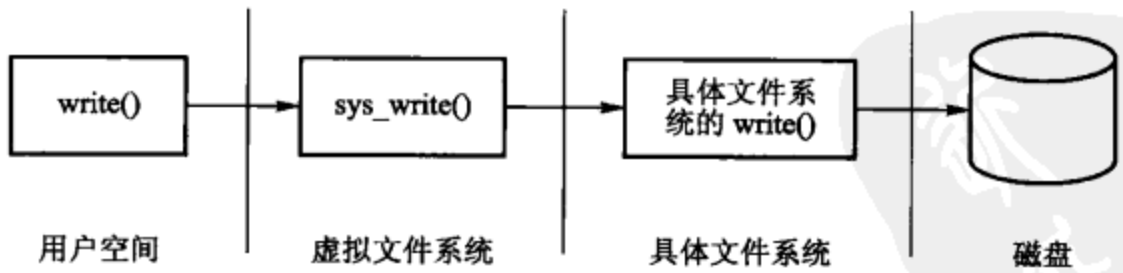


图 17-2 文件写入磁盘的工作流程示意图

为了深入理解两个文件系统中的文件如何被交叉进行访问，进一步来讨论读写操作的执行过程。打开的文件在内核中用系统打开文件表 file 数据结构来表示，它有一个成员指向文件操作表的指针 struct file_operation *f_op，其类型为 file_operation 结构。该文件操作表中包含指向各种函数，如 read( )、write( )、open( )、close( )等的入口，每种具体文件系统都有自己的 file operation 结构，即有自己的操作函数。当进程使用 open( )打开文件时，将与具体文件建立连接，

这种连接以 file 结构作为纽带，而将其中的 f_op 设置成指向某个具体的 file_operation 结构，也就指定了该文件所属的文件系统。因此，当应用程序调用 read( )函数时，就会陷入内核而调用 sys_read( )内核函数，而 sys_read( )就会通过 file 结构中的指针 f_op 去调用 DOS 文件系统的读函数，即 file→f_op→read( )函数。同样，write( )操作也会引发一个与输出文件相关的 Ext3 的 file→f_op→write( )写函数的执行。具体的数据结构和操作接口将会在下面部分中介绍。

可见，为了使多个文件系统能够共存于一个操作系统中，并向用户屏蔽差异，VFS 需要从各种不同的具体文件系统中抽象出共性，即基本的、概念上的接口和数据结构，并进行定义，以明确统一的界面(如 file_operation 及其他接口)。这样，用户应用程序看到的是相同的文件操作方式。当然，这也需要具体文件系统将自身的诸如“如何打开文件”、“目录是什么”等概念在形式上与 VFS 的定义保持一致。

VFS 定义一系列标准的数据结构与操作，基于这些数据结构和操作实现 VFS 框架。在该框架中，VFS 提供独立于具体文件系统的通用文件操作，提供索引节点缓存、目录高速缓存等用于提高文件系统效率的缓存模块。由于 VFS 基本上是按 UNIX 类文件抽象的，对于非 UNIX 类文件系统，如 Windows 的 FAT 或 NTFS 等，要想能被 VFS 支持，它们的文件系统代码必须提供这些概念的模拟形式。例如，即使文件系统不支持索引节点，它也必须在主存中装配索引节点结构体(如同本身固有一样)。或者如果某文件系统将目录看做是一种特殊对象，那么要想使用 VFS，必须将目录重新表示为文件形式。通常，这种转换需要在使用现场引入特殊处理，使非 UNIX 类文件系统能够兼容 UNIX 类文件系统的使用规则和满足 VFS 的需求。通过这些处理，非 UNIX 类文件系统便可以和 VFS 一同工作，但性能上多少会受些影响。当一个具体文件系统接入 VFS 后，就能享受到 VFS 提供的各种缓存带来的好处。

17.2.2 文件系统的安装和挂载

VFS 支持的文件可划分为 3 种主要类型。

(1) 磁盘文件

基于磁盘的文件系统管理本地磁盘分区中可用的存储空间，这种文件的“文件数据”存放在磁盘的数据块中，“文件管理信息”存放在磁盘的索引节点中。这类文件系统中，Linux 文件系统有 Ext2、Ext3、ReiserFS；UNIX 文件系统有 SystemV、UFS、MINIX 及 VERITAS VxFS；微软文件系统有 MS-DOS、VFAT 及 NTFS；ISO9660CD-ROM 和通用磁盘格式的 DVD 文件系统；其他有专利权的文件系统有 HPFS、HFS、AFFS、ADFS；日志文件系统有 JFS 和 XFS。

(2) 设备文件

设备文件同样有“文件管理信息”，而且也存储在磁盘的索引节点中，但却不一定在磁盘上存储数据。根据设备类型和性质的不同，它可以是用于存储/读出的(如磁盘)、也可以是接收/发送的(如网络卡)、还可以是供采集/控制的(如机电设备)等，不管什么设备，在操作过程中总要伴随着一定程度的数据采集和控制，这可通过设备接口上的“控制/状态寄存器”进行。用于访问其他网络计算机所包含的文件系统，如 NFS、Coda、AFS、SMB、NCP 等，也作为设备文

件处理。

(3) 特殊文件

特殊文件有“文件管理信息”，但不一定在磁盘上有索引节点，即不管理具体的磁盘空间，且它一般都与外部设备无关，涉及的介质通常为主存或 CPU。当从一个特殊文件“读”时，所读出的数据都是由系统内部按一定规则临时生成，或从主存中收集、加工出来的，反之亦然。特殊文件例子有/proc、/dev/null、命名管道和套接字等。

各种不同的文件系统通过 mount 挂载和安装到根文件系统中，在 Linux 中，根文件系统通常是 Ext2 或 Ext3 文件系统。当进程或 Shell 命令访问目录和文件时，先调用库函数，再调用对应内核函数，进入内核态后将遍历 VFS 的 VFS 索引节点，而 VFS 索引节点将会指向具体文件系统的索引节点，再通过底层块 I/O 函数来调用块驱动程序访问块设备(磁盘)，从而得到文件数据。VFS 对文件系统中具有共性的上层和底层进行处理，对上层进行处理的工作有：文件路径查找、文件读写共性操作处理；对底层进行处理的工作有：各种高速缓存(如页缓存)处理。

VFS 要与具体的文件系统相联系，需要将这个具体文件系统注册并挂载到 VFS 上。文件系统注册是一种文件系统被安装到 VFS 上时需要做的首要工作。向内核注册一个新文件系统有两种途径：一种是在编译内核时确定，并在系统初始化时通过内嵌的函数调用向注册链表登记；另一种则是利用 Linux 的内核模块机制，把文件系统当作一个内核模块。这里主要介绍通过内核模块注册文件系统的方式，因为这种方式简单、便捷，无需重新编译内核就可以挂载和使用自己的文件系统。而文件系统的挂载，主要是通过 mount 命令把具体文件系统的根目录挂载到系统目录树的某个目录下(一般是/mnt)，这个挂载的系统目录被称为安装点(mount point)。

VFS 提供标准的数据结构来描述某个特定文件系统的类型和一个已安装文件系统的实例，分别为 file_system_type 结构体和 vfsmount 结构体。每种文件系统不管有多少个实例安装到系统中，还是根本就没有安装到系统中，都只有一个 file_system_type 结构。而安装一个文件系统时，将在安装点创建一个 vfsmount 结构体。该结构体用来代表文件系统的实例，即代表一个安装点。file_system_type 和 vfsmount 的具体结构会在后面的部分里结合实例介绍。

17.2.3 虚拟文件系统的结构和通用文件模型

从面向对象的角度来说，VFS 定义一个文件系统的基类，并基于该基类实现一个文件系统的框架。如果一个文件系统(如 Ext3)需要接入 VFS 中，该文件系统只需要提供继承 VFS 标准基类的一个子类即可。这里需要说明的是，因为 Linux 内核是纯粹使用 C 语言实现的，没有使用面向对象语言来支持，所以内核中的数据结构都使用 C 结构体实现。这些结构体包含数据的同时，也包含操作这些数据的函数指针，这些函数指针指向的具体函数由具体的文件系统来实现。VFS 结构如图 17-3 所示。

VFS 实质上是一种存在于主存中的，支持多种类型具体文件系统运行环境，其功能包括：

- 记录安装的文件系统类型。
- 建立设备与文件系统的联系。

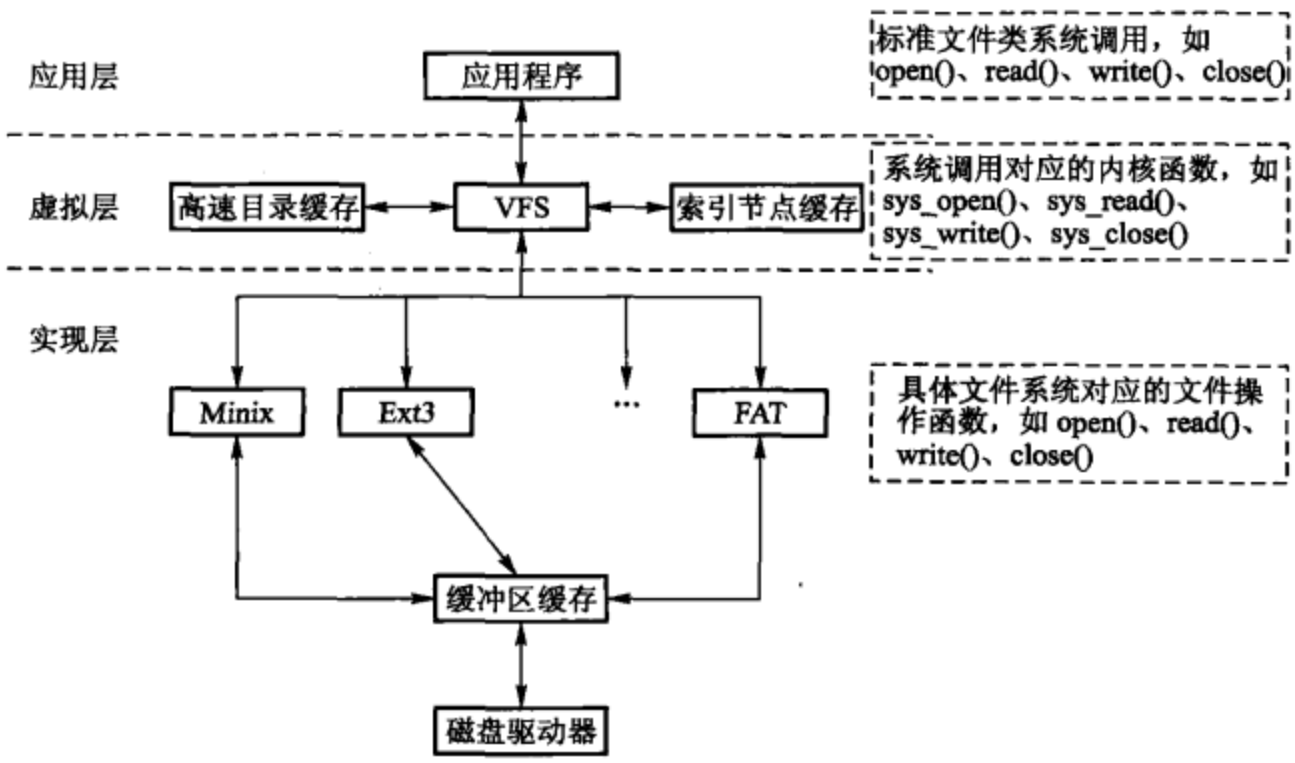


图 17-3 虚拟文件系统的结构

- 处理面向文件的通用操作。
- 涉及具体文件系统的操作时，把这些操作映射到具体文件系统中执行。

VFS 有自己的通用文件模型，该模型主要定义以下 4 种对象。

- 超级块(SuperBlock)对象：代表一个已安装的文件系统，存放已安装文件系统的信息。如果是基于磁盘的文件系统，该对象便对应于存放在磁盘上的文件系统控制块，每个文件系统都对应一个超级块对象。
- 索引节点(inode)对象：代表一个文件。存放通用的文件信息，如果是基于磁盘的文件系统，该对象通常对应于存放在磁盘上的文件控制块，每个文件都有一个索引节点对象，而每个索引节点都有一个索引节点号，该号唯一地标识某个文件系统中的指定文件。
- 目录项(Dentry)对象：代表路径中的一个组成部分。存放目录项与对应文件进行链接的各种信息。VFS 把最近最常使用的目录项对象放在目录项高速缓存中，加快文件路径名搜索过程，以提高系统性能。
- 文件(File)对象：代表由进程已打开的一个文件。存放已打开的文件与进程的交互信息，这些信息仅当进程访问文件期间才存于主存中。

图 17-4 描述超级块、索引节点、目录项和文件这 4 种对象的关系：超级块对象包含所有已经打开的文件对象的链表以及所有“脏”索引节点的链表。

以上每个主要对象都包含一个操作对象，它描述内核针对主要对象可以使用的方法，这些操作对象是：

- super_operation 对象，其中包括内核针对超级块所能调用的方法。
- inode_operation 对象，其中包括内核针对索引节点所能调用的方法。

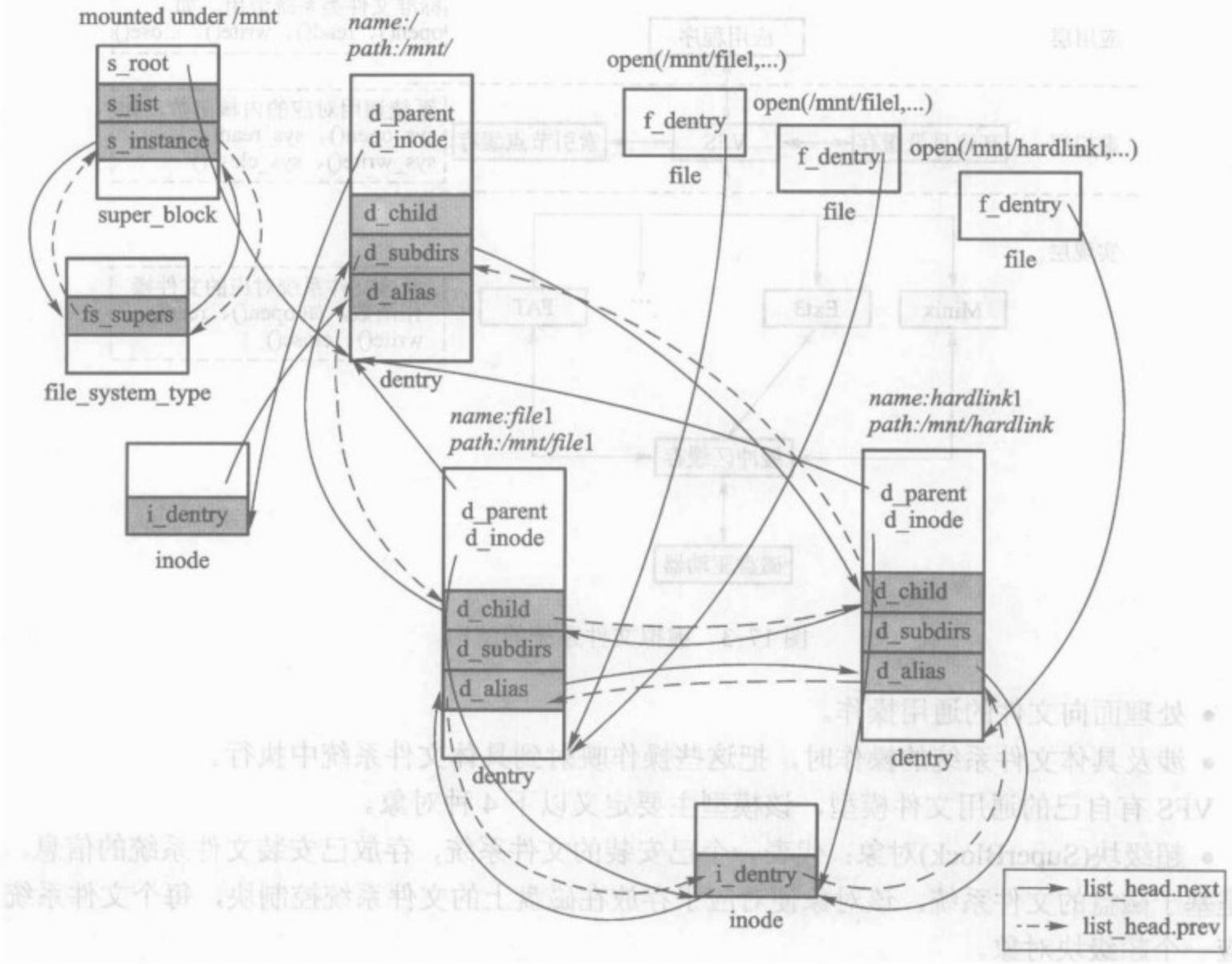


图 17-4 VFS 中各个数据对象的关系图

- dentry_operation 对象，其中包括内核针对目录项所能调用的方法。
- file_operation 对象，其中包括进程针对已打开文件所能调用的方法。

这些操作对象作为一个指针结构体来实现，此结构体中包含指向操作其父对象的函数指针。对于其中许多方法而言，VFS 提供通用的函数实现。当然，如果这些通用的函数实现无法满足具体的需要，那么具体的文件系统必须让这些函数指针指向它们提供的独有的方法。下面是各个操作对象中需要实现的函数指针之间的关系(以 read( )、write( )函数为例)：

Super Operation	File Operation	Inode Operation	Address Space Operation
read_inode( )	read( )	look_up( )	readpage( )
write_inode( )	write( )		prepare_write( )
	readaddr( )		commit_write( )

图 17-5 展示了函数与各个数据对象以及各个对象的 operation 操作对象的关系。

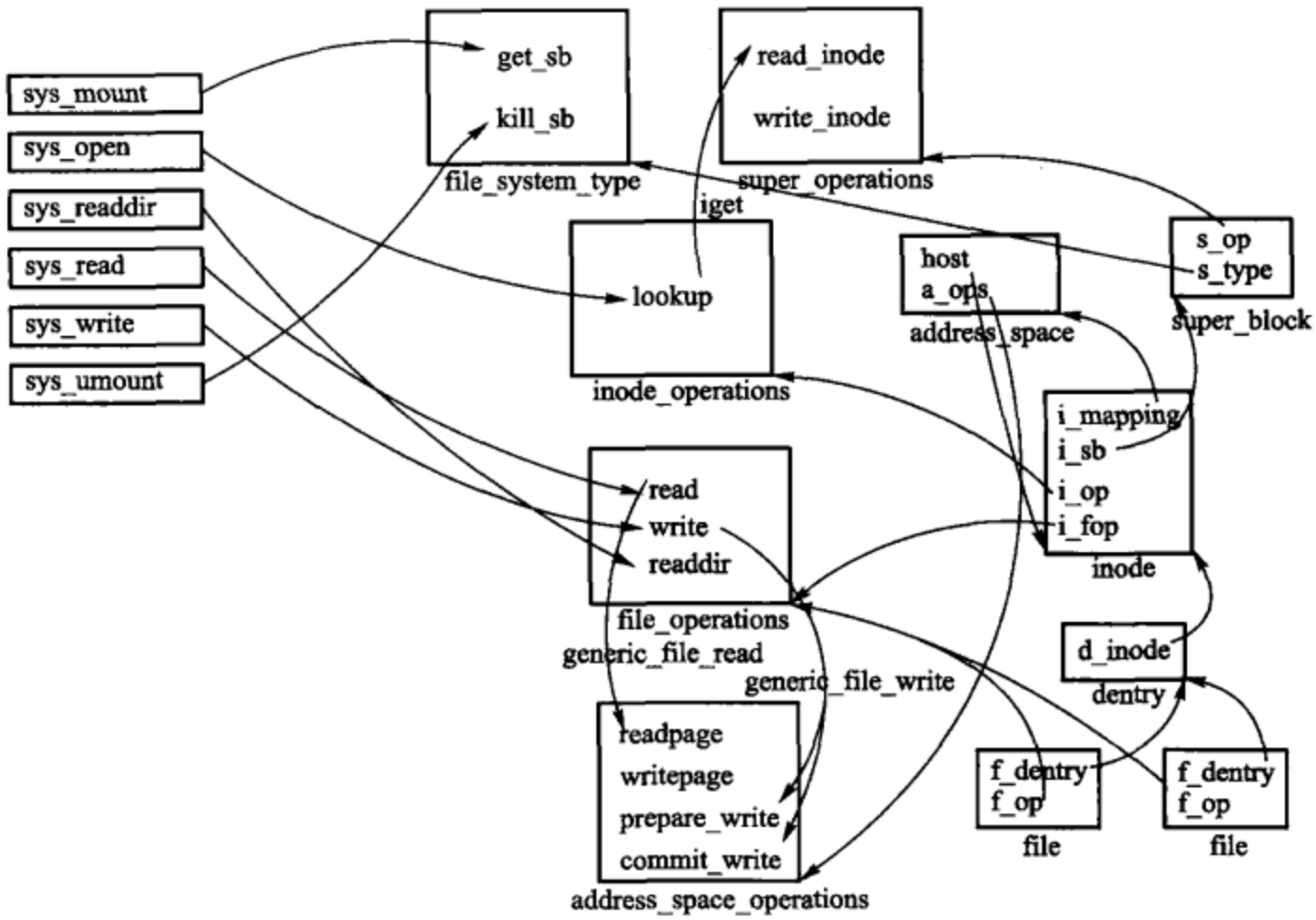


图 17-5 函数和数据对象以及数据对象中操作对象的关系

17.3 实 验 内 容

实验 实现一个虚拟文件系统

1. 实验说明

本虚拟文件系统的实验，需要完成以下 3 个内容：

- 完成 Sparrow 文件系统的编程、调试和挂载。
- 让 Sparrow 文件系统具有文件链接功能。链接功能包括符号链接(Symbolic Links，又称为软链接)和硬链接(Hard Links)。具体做法是：增加链接对应函数，并在一些操作(如取 inode 节点的函数)中予以判断是否是链接。
- 让 Sparrow 文件系统包含相关文件属性。增加文件属性包括 UID(用户 ID)、GID(组 ID)、文件大小、时间戳(Linux 中的时间戳包括 3 类，分别为文件的最后存取时间 atime、文件的最后修改时间 ctime 和索引节点的最后修改时间 mtime)等。因为 Sparrow 文件系统中创建的文件没有附带任何属性信息，因此，尝试为 Sparrow 系统中创建的文件添加属性信息。具体做法是：

参考 setattr( )函数，上述一些操作都是基于 iatt 结构，通过该结构，可以对 uid、gid、mode、timestamps、size 等进行操作和改变。

经过前面的学习，已经了解 VFS 的基本概念，想要深入理解 VFS 的数据结构和操作接口，并且理解内核对 VFS 的支持和一个具体文件系统在内核中的工作流程，最好的方法就是编写和添加一个哪怕最简单的具体文件系统。本节的目的，是通过展示一个最简单的、只有基本功能的具体文件系统，称为“麻雀”文件系统(Sparrow File System)，意指“麻雀虽小，五脏俱全”，来展示 VFS 与具体文件系统连接的各个不可忽视的细节。

相信在理解 Sparrow File System 之后，读者就不难回答以下问题：

- 当加载一个具体文件系统时，内核发生了什么？
- 为了使具体文件系统能被 VFS 支持，具体需要提供什么给内核？
- 应该如何给 inode_operation、dentry_operation 及 file_operation 添加特有的操作函数，来支持具体文件系统独有的对象操作？

Sparrow 文件系统是一个基于 Linux 操作系统平台的、仅仅具有基本功能的文件系统，它能够支持也仅仅支持以下功能：

- 文件系统的初始化和安装、挂载/卸载。
- 文件的创建和删除。
- 目录的创建和删除。
- 文件的编辑和修改。

并且，Sparrow 文件系统是一个驻留在主存中的文件系统，其中的文件只存在于主存中，Sparrow 文件系统被卸载，其中的文件将会丢失。Sparrow 文件使用 Linux 内核模块技术来安装和挂载，所以把 Sparrow 文件系统添加进内核只需要使用 mount 命令即可，不必要重新编译内核，当然，首先需要在内核中加载 Sparrow 文件系统这个内核模块。

Sparrow 文件系统面向的 Linux 内核版本为 Linux-kernel-2.6。Linux-kernel-2.6 的文件系统函数与 v2.4 版相比发生了一些变化，如果以 v2.4 版内核作为实验环境，需要做一些代码修改。

2. 解决方案

前面介绍过，VFS 是 Linux 内核在底层文件系统接口上建立的一个抽象层。通过该抽象层，内核能够方便、简单地同时支持各种类型的文件系统，并使不同的文件系统能够协同工作。从文件系统角度来看，具体文件系统要能够被 Linux 内核所支持，继而被安装、加载并提供给用户使用，它必须“迎合”VFS 所定义的抽象层，即由 VFS 所定义的数据对象和对象操作。

由此可见，想要成功编写和添加具体文件系统，首先要熟悉 VFS 所定义的数据对象和对象操作。继而维护好这些数据对象，并添加自己特有的对象操作。在本小节，将介绍 VFS 定义的各个数据对象和对象操作，当然，VFS 定义的数据结构是非常繁琐和复杂的，这里仅介绍那些最重要的，并且在 Sparrow 文件系统中实际用到的数据结构和操作，以呈现一条清晰的主线。

(1) 超级块(Super Block)

超级块描述一个文件系统的信息。VFS 超级块对象是各个具体文件系统安装时才建立的，

并在这些具体文件系统卸载时被自动删除，可见 VFS 超级块仅存于主存中。它记录文件系统类型、根目录、设备序号等信息。超级块对象的数据结构由 super_block 结构体表示，在文件<Linux/fs.h>中定义，如下所示。

```
struct super_block {
        struct list_head s_list;                /*指向超级块链表的指针*/
        dev_t s_dev;                            /*设备标识符*/
        unsigned long s_blocksize;              /*以字节为单位的块大小*/
        unsigned long s_old_blocksize;          /*以字节为单位的旧的块大小*/
        unsigned char s_blocksize_bits;         /*以比特为单位的块大小*/
        unsigned char s_dirt;                   /*脏标记*/
        unsigned long long s_maxbytes;          /*最大文件大小*/
        struct file_system_type s_type;         /*文件系统类型*/
        struct super_operations s_op;           /*超级块方法*/
        struct dquot_operations *dq_op;         /*磁盘限额方法*/
        struct quotactl_ops *s_qcop;            /*限额控制方法*/
        struct export_operations *s_export_op;  /*导出方法*/
        unsigned long s_flags;                  /*登录标志*/
        unsigned long s_magic;                  /*文件系统的幻数（magic number）*/
        struct dentry *s_root;                  /*目录挂载点*/
        struct rw_semaphore s_umount;           /*卸载信号量*/
        struct semaphore s_lock;                /*超级块信号量*/
        int s_count;                            /*超级块引用计数*/
        int s_syncing;                          /*文件系统同步标志*/
        int s_need_sync_fs;                     /*尚未同步标志*/
        atomic_t s_active;                      /*活动引用计数*/
        void *s_security;                       /*安全模块*/
        struct list_head s_dirty;               /*脏节点链表*/
        struct list_head s_io;                  /*写回链表*/
        struct hlist_head s_anon;               /*匿名目录项*/
        struct list_head s_files;               /*被分配文件链表*/
        struct block_device *s_bdev;            /*相关的块设备*/
        struct list_head s_instances;           /*该类型的文件系统*/
        struct quota_info s_dquot;              /*限额相关选项*/
        char s_id[32];                          /*文本名字*/
        void *s_fs_info;                        /*文本系统特殊信息*/
        struct semaphore s_vfs_rename_sem;      /*重命名信号量*/
};
```

每个不同的具体文件系统可以将独有的信息记录在该结构最后的联合体记录中。与超级块关联的方法就是超级块操作对象(上面代码中的**下划线**部分)，这些操作由 super_operation 结构

体来描述(同样在<Linux/fs.h>中定义)，如下所示。

```
struct super_operations {
        struct inode *(*alloc_inode) (struct super_block *sb);
        void (*destroy_inode) (struct inode *);
        void (*read_inode) (struct inode *);
        void (*dirty_inode) (struct inode *);
        void (*write_inode) (struct inode *, int);
        void (*put_inode) (struct inode *);
        void (*drop_inode) (struct inode *);
        void (*delete_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        void (*write_super) (struct super_block *);
        int (*sync_fs) (struct super_block *, int);
        void (*write_super_lockfs) (struct super_block *);
        void (*unlockfs) (struct super_block *);
        int (*statfs) (struct super_block *, struct statfs *);
        int (*remount_fs) (struct super_block *, int *, char *);
        void (*clear_inode) (struct inode *);
        void (*umount_begin) (struct super_block *);
        int (*show_options) (struct seq_file *, struct vfsmount *);
};
```

结构体中的每一项是一个指向超级块操作的函数指针，而超级块操作函数执行文件系统和 inode 的底层操作。在 Sparrow 文件系统中，仅仅实现上述代码中**具有下划线**的几个函数，其他的函数则默认使用 VFS 本身定义的通用操作。

下面来看 3 个函数指针所指向的函数应该实现的具体功能。

- void (*put_inode) (struct inode *)：在逻辑上释放给定索引节点(delete_inode 则是在物理上释放索引节点)。
- void (*drop_inode) (struct inode *)：在最后一个指向索引节点的引用被释放后，VFS 就会调用该函数来删除这个索引节点。
- int (*statfs) (struct super_block *, struct statfs *)：VFS 通过调用该函数获取文件系统的状态。指定文件系统相关的统计信息将放置在 statfs 结构体中。

(2) 索引节点(Index Node)

索引节点存放通用的文件信息。如果是基于磁盘的文件系统，该对象通常对应于存放在磁盘上的文件控制块(FCB)。索引节点信息中主要包含文件的控制信息，如文件的权限、属性；文件的管理信息，如长度、创建时间、修改时间等；文件的存储信息，如文件存放的磁盘盘块号、文件所在的设备号等。根据索引节点中的信息，文件系统能够在磁盘上直接找到该文件的数据区，并进行数据访问。根据这些介绍，现在可以来看看索引节点对象所包含的主要成员。

索引节点对象由 inode 结构体表示，在文件<Linux/fs.h>中定义，如下所示。

```
truct inode {
        struct hlist_node i_hash;                    /*散列表*/
        struct list_head i_list;                     /*索引节点链表*/
        struct list_head i_dentry;                   /*目录项链表*/
        unsigned long i_ino;                         /*节点号*/
        atomic_t i_count;                            /*引用计数*/
        umode_t i_mode;                              /*访问权限控制*/
        unsigned int i_nlink;                        /*硬连接数*/
        uid_t i_uid;                                 /*用户的 ID 号*/
        gid_t i_gid;                                 /*用户的组 ID 号*/
        kdev_t i_rdev;                               /*实设备标识符*/
        loff_t i_size;                               /*以字节为单位的文件大小*/
        struct timespec i_atime;                     /*最后访问时间*/
        struct timespec i_mtime;                     /*最后修改时间*/
        struct timespec i_ctime;                     /*最后改变时间*/
        unsigned int i_blkbits;                      /*以比特为单位的块大小*/
        unsigned long i_blksize;                     /*以字节为单位的块大小*/
        unsigned long i_version;                     /*版本号*/
        unsigned long i_blocks;                      /*文件块数*/
        unsigned short i_bytes;                      /*使用的字节数*/
        spinlock_t i_lock;                           /*自旋锁*/
        struct rw_semaphore i_alloc_sem;             /*嵌入在 i_sem 内部*/
        struct semaphore i_sem;                      /*索引节点信号量*/
        struct inode_operations *i_op;               /*索引节点操作表*/
        struct file_operations *i_fop;               /*默认的索引节点操作*/
        struct super_block *i_sb;                    /*相关的超级块*/
        struct file_lock *i_flock;                   /*文件锁链表*/
        struct address_space *i_mapping;             /*相关的地址映射*/
        struct address_space i_data;                 /*设备地址映射*/
        struct dquot *i_dquot[MAXQUOTAS];            /*节点的磁盘限额*/
        struct list_head i_devices;                  /*块设备链表*/
        struct pipe_inode_info *i_pipe;              /*管道信息*/
        struct block_device *i_bdev;                 /*块设备驱动*/
        unsigned long i_dnotify_mask;                /*目录通知掩码*/
        struct dnotify_struct *i_dnotify;            /*目录通知*/
        unsigned long i_state;                       /*状态标志*/
        unsigned long dirtied_when;                  /*首次修改时间*/
        unsigned int i_flags;                        /*文件系统标志*/
        unsigned char i_sock;                        /*是个套接字吗？ */
```

```
        atomic_t i_writecount;                  /*写者计数*/
        void *i_security;                       /*安全模块*/
        __u32 i_generation;                     /*索引节点版本号*/
        union {
                void *generic_ip;               /*文件特殊信息*/
        } u;
};
```

在该结构的最后，有一个联合体成员，每一个具体的文件系统能够根据自身需要，将一些特殊信息写入这个联合体。

值得注意的是，在索引节点中并没有包含文件名信息，这是因为在 Linux 中，一些文件系统允许一种“链接”操作，如 Ext2 文件系统。“链接”操作解决文件的静态共享问题，“链接”操作使多个文件能够使用同一处物理存储，即一处物理存储能够有多个文件名。这种方式节省磁盘空间；使多个用户能够方便地共享数据，并且不会造成数据的不一致性。索引节点数据结构中的 i_nlink 成员记录文件的链接数。索引节点实际上记录的是文件的物理存储信息，并不关心该文件具体叫什么名字，因此索引节点没有存储文件名信息。

在 Linux 中，目录是被当成普通文件看待的，因此不但每个文件都必须要有一个索引节点与之对应，每个目录也都有一个索引节点。索引节点包含内核在操作文件或目录时需要的全部信息，文件名可以更改，但索引节点对文件是唯一的，且随文件的存在而存在。对于 UNIX 类文件系统来说，这些信息可以从磁盘索引节点区直接读入 VFS 的索引节点对象中。如果一个文件系统没有索引节点，那么，不管这些相关信息在磁盘上是如何存放的，文件系统都必须提取这些信息，并构造它的索引节点。

当用户通过 VFS 创建、删除文件时，必然涉及对于索引节点的操作。不同文件系统对于索引节点有着不同的操作方法，VFS 通过 inode_operations 结构(在文件<Linux/fs.h>中定义)定义对索引节点的操作方法，如下所示。

```
struct inode_operations {
        int (*create) (struct inode *, struct dentry *,int);
        struct dentry * (*lookup) (struct inode *, struct dentry *);
        int (*link) (struct dentry *, struct inode *, struct dentry *);
        int (*unlink) (struct inode *, struct dentry *);
        int (*symlink) (struct inode *, struct dentry *, const char *);
        int (*mkdir) (struct inode *, struct dentry *, int);
        int (*rmdir) (struct inode *, struct dentry *);
        int (*mknod) (struct inode *, struct dentry *, int, dev_t);
        int (*rename) (struct inode *, struct dentry *,struct inode *, struct dentry *);
        int (*readlink) (struct dentry *, char *, int);
        int (*follow_link) (struct dentry *, struct nameidata *);
        int (*put_link) (struct dentry *, struct nameidata *);
```

```
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (struct vfsmount *, struct dentry *, struct kstat *);
        int (*setxattr) (struct dentry *, const char *,const void *, size_t, int);
        ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
        ssize_t (*listxattr) (struct dentry *, char *, size_t);
        int (*removexattr) (struct dentry *, const char *);
};
```

Sparrow 文件系统实现其中 6 个函数(**有下划线的**部分)，功能分别如下。

- int (*create) (struct inode *, struct dentry *,int)：该函数被函数 create( )和 open( )所调用，为目录项对象创建一个新的索引节点。
- struct dentry * (*lookup) (struct inode *, struct dentry *)：在特定的目录中寻找索引节点，该索引节点要对应目录项中给出的文件名。
- int (*mkdir) (struct inode *, struct dentry *, int)：该函数被函数 mkdir( )所调用，来创建一个新目录。
- int (*rmdir) (struct inode *, struct dentry *)：该函数被函数 rmdir( )所调用，删除 dir 目录中的 dentry 目录项代表的文件。
- int (*mknod) (struct inode *, struct dentry *, int, dev_t)：该函数被函数 mknod( )所调用，创建一个特殊文件。要创建的文件放在 dir 目录中，其目录项为 dentry，关联设备为 rdev。
- int (*getattr) (struct vfsmount *, struct dentry *, struct kstat *)：在通知索引节点需要从磁盘中更新时，VFS 会调用该函数。

(3) 目录项(Directory Entry)

文件系统通过目录将文件组织成树状结构，目录可以看成是一种具有特殊结构的文件，该文件中记录所包含的子目录和文件的信息。文件目录包含许多目录项，每一个目录项包含一个文件及其与索引节点的映射关系。其结构如图 17-6 所示。

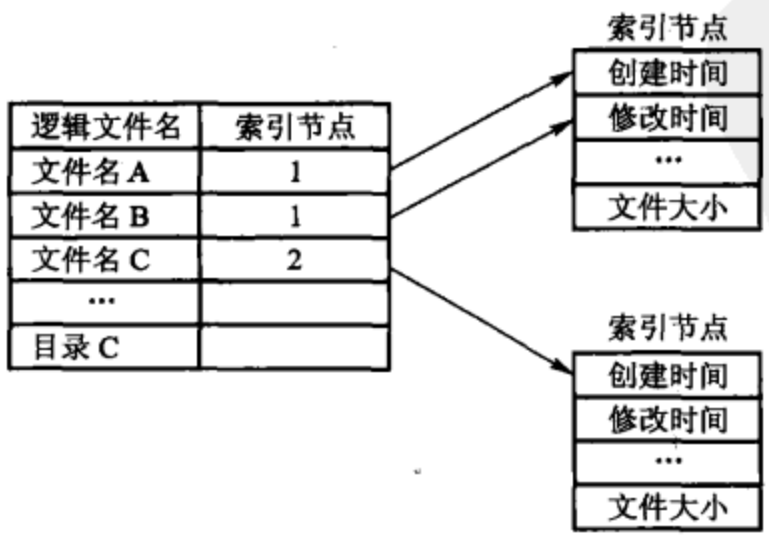


图 17-6 目录结构

在 VFS 对象中，目录项数据结构由 dentry 结构在文件<Linux/dcache.h>中定义，如下所示。

```
struct dentry {
    atomic_t d_count;                       /*使用计数*/
    unsigned long d_vfs_flags;              /*目录项缓存标志*/
    spinlock_t d_lock;                      /*单目录项锁*/
    struct inode *d_inode;                  /*相关的索引节点*/
    struct list_head d_lru;                 /*未用的链表*/
    struct list_head d_child;               /*父目录中目录项对象的链表*/
    struct list_head d_subdirs;             /*子目录*/
    struct list_head d_alias;               /*索引节点的别名链表*/
    unsigned long d_time;                   /*重新生效时间*/
    struct dentry_operations *d_op;         /*目录项操作表*/
    struct super_block *d_sb;               /*文件超级块*/
    unsigned int d_flags;                   /*目录项标志*/
    int d_mounted;                          /*是登录节点的目录项吗？ */
    void *d_fsdata;                         /*文件系统特殊的数据*/
    struct rcu_head d_rcu;                  /*RCU 锁*/
    struct dcookie_struct *d_cookie;        /*cookie*/
    struct dentry *d_parent;                /*父目录的目录项对象*/
    struct qstr d_name;                     /*目录项的名字*/
    struct hlist_node d_hash;               /*散列表*/
    struct hlist_head *d_bucket;            /*散列表头*/
    unsigned char d_iname[DNAME_INLINE_LEN_MIN];    /*短文件名*/
};
```

在该数据结构中，d_inode 指针指向一个索引节点，并用 d_iname 记录文件名。由于“链接”的关系，可能会有多个 dentry 结构指向同一个索引节点。所以在索引节点对象中有一个队列 i_dentry，凡代表同一个文件的所有目录项都通过其 dentry 结构中的 d_alias 域链入相应 inode 结构中的 i_dentry 队列。该结构中还有多个成员将 dentry 结构组织成树状结构或哈希表，这样能够方便的访问文件系统中的 dentry 结构。

试想，如果 VFS 遍历路径名中所有的元素并将它们逐个地解析成目录项对象，将是一件非常低效的工作，所以内核将目录项对象缓存在目录项缓存(dcache)中，并使用散列表和相应的散列函数来快速地将给定路径解析为相关的目录项对象。dentry 结构涉及的操作主要由 dentry_operation 结构描述，同样在文件<Linux/dcache.h>中定义，如下所示。

```
struct dentry_operations {
    int (*d_revalidate) (struct dentry *, int);
    int (*d_hash) (struct dentry *, struct qstr *);
    int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
    int (*d_delete) (struct dentry *);
    void (*d_release) (struct dentry *);
```

```
        void (*d_iput) (struct dentry *, struct inode *);
};
```

Sparrow 文件系统实现如下几个函数：

- int (*d_hash) (struct dentry *, struct qstr *)：为目录项生成散列值，当目录项需要加入到散列表中时，VFS 调用该函数。
- int (*d_compare) (struct dentry *, struct qstr *, struct qstr *)：比较 name1 和 name2 两个文件名。
- int (*d_delete) (struct dentry *)：当目录项对象的 d_count 计数值为 0 时，VFS 调用该函数。

(4) 文件(File)

每个文件都用一个 32 位数字来表示下一个读写的字节位置，通常称它为文件位置或偏移量(offset)，每当打开一个文件时，偏移量被置 0，读写操作便从这里开始，允许通过函数 lseek( ) 对文件位置作随机定位。Linux 建立文件对象(file)来保存打开文件的文件当前位置，file 结构除保存文件当前位置外，还把指向该文件索引节点的目录项指针也放在其中，并形成一个双向链表，称系统打开文件表。操作系统之所以不直接使用 dentry 结构是因为多个进程能够打开同一个文件，因为每一个 file 结构对应一个进程的一次打开过程。file 结构中记录文件访问模式，读写指针等信息。每当打开文件时，就要创建一个 file 结构，该结构在<Linux/fs.h>中定义，如下所示。

```
struct file {
    struct list_head f_list;                /*文件对象链表*/
    struct dentry *f_dentry;                /*相关目录项对象*/
    struct vfsmount *f_vfsmnt;              /*相关的安装文件系统*/
    struct file_operations *f_op;           /*文件操作表*/
    atomic_t f_count;                       /*文件对象的使用计数*/
    unsigned int f_flags;                   /*当打开文件时所指定的标志*/
    mode_t f_mode;                          /*文件的访问模式*/
    loff_t f_pos;                           /*文件当前的位移量（文件指针）*/
    struct fown_struct f_owner;             /*通过信号进行异步 I/O 数据的传送*/
    unsigned int f_uid;                     /*用户的 UID*/
    unsigned int f_gid;                     /*用户的 GID*/
    int f_error;                            /*错误码*/
    struct file_ra_state f_ra;              /*预读状态*/
    unsigned long f_version;                /*版本号*/
    void *f_security;                       /*安全模块*/
    void *private_data;                     /*tty 驱动程序的 hook*/
    struct list_head f_ep_links;            /*事件池列表*/
    spinlock_t f_ep_lock;                   /*事件池锁*/
    struct address_space *f_mapping;        /*页缓存映射*/
};
```

file 结构与 dentry 结构之间的对应关系是多对一。f_dentry 指针指向文件对象所对应的目录项结构。与文件关联的方法就是文件操作对象，这些操作由 file_operation 结构描述，在<Linux/fs.h>中定义，如下所示。

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, char *, size_t, loff_t);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
        ssize_t (*aio_write) (struct kiocb *, const char *, size_t, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int);
        int (*aio_fsync) (struct kiocb *, int);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
        ssize_t (*sendpage) (struct file *, struct page *, int,size_t, loff_t *, int);
        unsigned long (*get_unmapped_area) (struct file *, unsigned long,unsigned long, unsigned long,
        unsigned long);
        int (*check_flags) (int flags);
        int (*dir_notify) (struct file *filp, unsigned long arg);
        int (*flock) (struct file *filp, int cmd, struct file_lock *fl);
};
```

Sparrow 文件系统实现如下几个函数：

- loff_t (*llseek) (struct file *, loff_t, int)：该函数被函数 llseek( )所调用，用于更新偏移量指针。
- ssize_t (*read) (struct file *, char *, size_t, loff_t *)：该函数被函数 read( )所调用，从给定文件的 offset 偏移处读取 count 字节的数据到 buf 中，同时更新文件指针。
- ssize_t (*aio_read) (struct kiocb *, char *, size_t, loff_t)：该函数被函数 aio_read( )所调用，从 iocb 描述的文件里，以同步方式读取 count 字节的数据到 buf 中。
- ssize_t (*write) (struct file *, const char *, size_t, loff_t *)：该函数被函数 write( )所调用，从

给定文件的 buf 中取出 count 字节的数据，写入给定文件的 offset 偏移处，同时更新文件指针。

• ssize_t (*aio_write) (struct kiocb *, const char *, size_t, loff_t)：该函数被函数 aio_write( )所调用，以同步方式从给定的 buf 中取出 count 字节的数据，写入由 iocb 描述的文件里。

• int (*mmap) (struct file *, struct vm_area_struct *)：该函数被函数 mmap( )所调用，将给定的文件映射到指定的地址空间上。

• int (*fsync) (struct file *, struct dentry *, int)：该函数被函数 fsync( )所调用，将给定文件的所有被缓存的数据写回到磁盘。

• ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *)：该函数被函数 sendfile( )所调用，将数据从一个文件复制到另一个文件中。

超级块、索引节点、目录项和文件这 4 种对象的关系如图 17-7 所示。

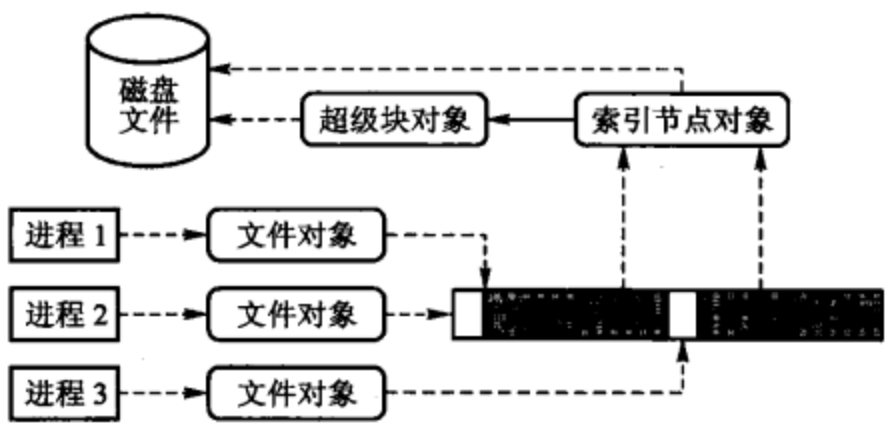


图 17-7 超级块、索引节点、目录项和文件对象关系

(5) 文件系统类型(File System Type)

每个文件系统都有初始化函数，它用于向 VFS 注册，即填写由 file_systems 指向的文件系统注册表数据结构 file_system_type，每个文件系统类型在注册表中有一个登记项，记录该文件系统类型名、文件系统特性、指向对应的 VFS 超级块读取函数的地址以及已注册项的链指针等。

```
struct file_system_type {
        const char *name;               /*文件系统的名称*/
        struct subsystem subsys;        /*sysfs 子系统对象*/
        int fs_flags;                   /*文件系统类型标志*/
        /*以下函数用来从磁盘中读取超级块*/
        struct super_block *(*get_sb) (struct file_system_type *, int, char *, void *);
        /*下面的函数用来终止访问超级块*/
        Void (*kill_sb) (struct super_block *);
        struct module *owner;           /*拥有文件系统的模块*/
        struct file_system_type *next;  /*链表中的下一个文件系统类型*/
        struct list_head fs_supers;     /*超级块对象链表*/
};
```

get_sb 函数指针负责读取文件系统的超级块对象。每个文件系统都有自己的实现函数，如

Ext2 文件系统的具体项是 ext2_get_sb( )。所有的文件系统类型通过 next 成员链接起来。

(6) 地址空间：页高速缓存

页高速缓存是 Linux 内核实现的一种主要磁盘缓存。它主要用来减少对磁盘的 I/O 操作。具体地讲，是通过把磁盘数据缓存到物理主存中，把对磁盘的访问变为对物理主存的访问。

页高速缓存的操作不属于 VFS 的范畴。但是，要想实现一个能实用的文件系统，就必须考虑页高速缓存的问题。Linux 页高速缓存的目标是缓存任何基于页的对象，这包含各种类型的文件。

Linux 页高速缓存使用 address_space 结构体描述页高速缓存中的页面。该结构体在<Linux/fs.h>文件中定义，如下所示。

```
struct address_space {
        struct inode *host;                         /*拥有索引节点*/
        struct radix_tree_root page_tree;           /*包含全部页面的 radix 树*/
        spinlock_t tree_lock;                       /*保护 page_tree 的自旋锁*/
        unsigned int i_mmap_writable;               /*VM_SHARED 计数*/
        struct prio_tree_root i_mmap;               /*私有映射链表*/
        struct list_head i_mmap_nonlinear;          /*VM_NONLINEAR 链表*/
        spinlock_t i_mmap_lock;                     /*保护 i_mmap 的自旋锁*/
        atomic_t truncate_count;                    /*截断计数*/
        unsigned long nrpages;                      /*页总数*/
        pgoff_t writeback_index;                    /*写回的起始偏移*/
        struct address_space_operations *a_ops;     /*操作表*/
        unsigned long flags;                        /*gfp_mask 掩码与错误标志*/
        struct backing_dev_info *backing_dev_info;  /*预读信息*/
        spinlock_t private_lock;                    /*私有锁*/
        struct list_head private_list;              /*私有链表*/
        struct address_space *assoc_mapping;        /*相关的缓冲区*/
};
```

address_space 中的成员 a_ops 指向地址空间对象中的操作函数表，这与 VFS 对象及其操作对象关系类似。操作对象定义在文件<Linux/fs.h>中：

```
struct address_space_operations {
        int (*writepage)(struct page *, struct writeback_control *);
        int (*readpage) (struct file *, struct page *);
        int (*sync_page) (struct page *);
        int (*writepages) (struct address_space *, struct writeback_control *);
        int (*set_page_dirty) (struct page *);
        int (*readpages) (struct file *, struct address_space *, struct list_head *, unsigned);
        int (*prepare_write) (struct file *, struct page *, unsigned, unsigned);
        int (*commit_write) (struct file *, struct page *, unsigned, unsigned);
        sector_t (*bmap)(struct address_space *, sector_t);
        int (*invalidatepage) (struct page *, unsigned long);
```

```
        int (*releasepage) (struct page *, int);
        int (*direct_IO) (int, struct kiocb *, const struct iovec *, loff_t, unsigned long);
};
```

接下来讨论页面的读操作的步骤。首先，一个 address_space 对象和一个偏移量会被传送给 readpage( )内核函数，这两个参数用来在页高速缓存中搜索需要的数据。如果搜索的页面不在 cache 中，那么内核将分配一个新的页面，然后将其加入到页高速缓存中。最后，需要的数据从磁盘读入，再被加入到页面高速缓存，然后返回给用户。

对特定文件的写操作比较复杂。首先，在页高速缓存中搜索需要的页，如果需要的页不在 cache 中，那么在 cache 中新分配一空闲项；下一步，prepare_write( )内核函数被调用，创建一个写请求；接着数据被从用户空间复制到内核缓冲区；最后通过 commit_write( )内核函数将数据写入磁盘。

Sparrow 文件系统实现上面提到的 readpage( )、prepare_write( )和 commit_write( )这 3 个内核函数。

经过上面的分析，了解了要实现一个最简单的 Sparrow 文件系统，必须“填充”的 VFS 的各个基本操作对象中的函数指针。对于在 17.2 节中提到的 4 个基本对象，其所包含的操作对象中所必需实现的函数指针，以及这 4 个基本对象的关系如图 17-8 所示。

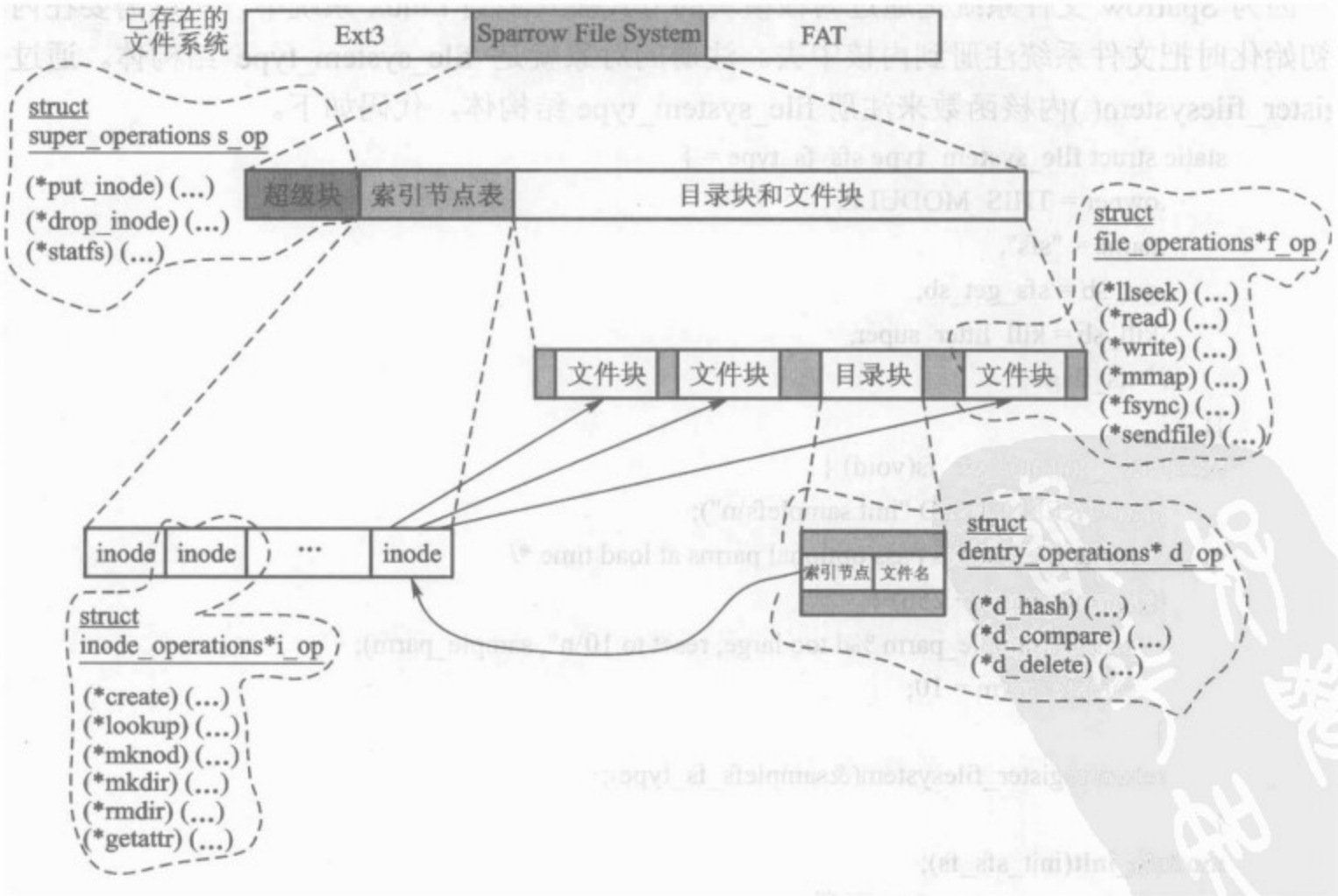


图 17-8 Sparrow 文件系统中 4 个基本对象需要实现的函数

Sparrow 文件系统各个基本对象和需要实现的函数总结如下：

• 超级块对象的 s_op 操作对象中需要实现的函数指针包括(*put_inode)( )、(*drop_inode)( )、(*statfs)( )。

• 索引节点对象的 i_op 操作对象中需要实现的函数指针包括(*create)( )、(*lookup)( )、(*mknod)( )、(*mkdir)( )、(*rmdir)( )和(*getattr)( )。

• 目录项对象的 d_op 操作对象中则需要实现(*d_hash)( )、(*d_compare)( )和(*d_delete)( )函数指针。

• 文件对象的操作对象 f_op 中需要给(*llseek)( )、(*read)( )、(*write)( )、(*mmap)( )、(*fsync)( )和(*sendfile)( )这些函数指针赋值。

动手编写和实现这些函数后，Sparrow 文件系统就初具雏形了！

3. Sparrow 文件系统的实现

在熟悉上述数据对象和其操作对象以后，就可以开始动手实现具体文件系统——Sparrow 文件系统。所要做的，就是实现上述**带有下划线**的内核操作函数的功能，并把所实现的操作函数赋给操作对象中的函数指针，下面来逐步讨论是如何实现这些操作函数的。

(1) init_module( )函数

因为 Sparrow 文件系统是通过内核模块的方式被安装到 Linux 系统中，所以需要在内核模块初始化时把文件系统注册到内核中去。注册的对象就是 file_system_type 结构体。通过调用 register_filesystem( )内核函数来注册 file_system_type 结构体，代码如下。

```
static struct file_system_type sfs_fs_type = {
    .owner = THIS_MODULE,
    .name = "sfs",
    .get_sb = sfs_get_sb,
    .kill_sb = kill_litter_super,
    /* .fs_flags */
};
static int __init init_sfs_fs(void) {
    printk(KERN_INFO "init samplefs\n");
    /* some filesystems pass optional parms at load time */
    if(sample_parm > 256) {
       printk("sample_parm %d too large, reset to 10\n", sample_parm);
       sample_parm = 10;
    }
    return register_filesystem(&samplefs_fs_type);
}
module_init(init_sfs_fs);
```

(2) file_system_type.get_sb( )函数

该函数将会在文件系统被挂载到根目录树的某个路径时被调用。这时需要返回一个超级块

对象。使用系统提供的辅助内核函数 get_sb_nodev( )来分配这个超级块对象，并且通过传递内核函数 sfs_fill_super( )作为填充超级块对象的方法。超级块的 s_op 域由自己定义的 sfs_super_block 结构体来填充。文件系统索引节点的根节点将也在该函数中被分配。而其目录项被设置在超级块对象的 s_root 域。这个目录项对象正是 lookup( )函数所在的入口点。

在索引节点被初始化之后，入口目录项通过调用 d_alloc_root( )内核函数来分配，并且被赋给超级块对象的 s_root 域。

```
static int
sfs_fill_super(struct super_block * sb, void * data, int silent)
{
    struct inode * inode;
    struct sfs_sb_info * sfs_sb;
    sb->s_maxbytes = MAX_LFS_FILESIZE;
    sb->s_blocksize = PAGE_CACHE_SIZE;
    sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
    sb->s_magic = SAMPLEFS_MAGIC;
    sb->s_op = &sfs_super_ops;
    sb->s_time_gran = 1; /*时间粒度为 1ns*/
    sb->s_fs_info = kzalloc(sizeof(struct sfs_sb_info),GFP_KERNEL);
    sfs_sb = SFS_SB(sb);
    if(!sfs_sb) {
        return -ENOMEM;
    }
    inode = sfs_get_inode(sb, S_IFDIR | 0755, 0);
    if(!inode) {
        kfree(sfs_sb);
        return -ENOMEM;
    }
    sb->s_root = d_alloc_root(inode);
    if (!sb->s_root) {
        iput(inode);
        kfree(sfs_sb);
        return -ENOMEM;
    }
    /*以下代码为每个 fs 提供了一个 sb 数据，实际上许多 fs 并不需要*/
    sfs_sb->local_nls = load_nls_default( );
    sfs_parse_mount_options(data, sfs_sb);
    return 0;
}
int sfs_get_sb(struct file_system_type *fs_type, int flags, const char *dev_name, void *data, struct
```

```
        vfsmount *mnt) {
            return get_sb_nodev(fs_type, flags, data, sfs_fill_super, mnt);
        }
```

这里假设 Sparrow 文件系统的安装点在/mnt 目录下。

(3) file_system_type.kill_sb( )函数

该函数在文件系统卸载时被调用。在 Sparrow 中，使用 kill_litter_super( )来实现该函数。

(4) super_operations.read_inode( )函数

索引节点对象由 iget( )函数来分配，并由索引节点号来标识。如果索引节点不在索引节点的 cache 中，内核分配一个新的索引节点并调用超级块对象中的 read_inode( )函数来处理。

(5) super_operations.write_inode( )函数

该函数将会在“脏”索引节点被刷新时调用。

(6) inode_operations.lookup( )函数

该函数在内核解析一个路径时被调用。父索引节点对象操作对象表中的 lookup( )函数被调用去解析子路径。根索引节点的入口目录项对象已经在超级块的 s_root 域中提供。

以一个挂载在/mnt 的文件系统为例。当使用 ls /mnt 命令查看内容时，内核将会创建一个文件对象。这个文件对象位于文件系统的根目录项中。如果使用 ls –l /mnt/hello.txt 命令查看文件时，内核会调用 lookup( )函数从文件系统的根索引节点开始搜索，去设置 hello.txt 文件的索引节点。如果 hello.txt 的索引节点已经存在，该索引节点将会被 d_add( )函数添加到目录项中。否则，返回一些对应的错误提示。

```
static struct
dentry *sfs_lookup(struct inode *dir, struct dentry *dentry, struct nameidata *nd)
{
    struct sfs_sb_info * sfs_sb = SFS_SB(dir->i_sb);
    if (dentry->d_name.len > NAME_MAX)
        return ERR_PTR(-ENAMETOOLONG);
    if(sfs_sb->flags & SFS_MNT_CASE)
        dentry->d_op = &sfs_ci_dentry_ops;
    else
        dentry->d_op = &sfs_dentry_ops;
        d_add(dentry, NULL);
        return NULL;
}
```

(7) file_operations.read( )函数

在一个文件系统中，当内核接收到一个读文件的请求时，就会调用文件操作对象中的 read( )函数指针。为了读取一个文件对象，用户空间的缓存地址、缓存的最大长度和当前缓存的偏移量将会作为参数传递，文件的内容将会被写入缓存，注意这些都是在用户空间中的。在下层实现中，copy_to_user( )函数将会被调用以把数据复制到用户空间的缓存中。

以上操作有两种实现方式。一种方式是直接实现一个将数据写入用户缓存的函数，但是这种实现方式无法利用页缓存的优势，另一种读/写文件的方式是使用主存映射方式。第二种方式是目前标准的读/写方式，该方式能够利用页缓存带来的性能优势，所以在 Sparrow 文件系统中，使用第二种方式。

文件对象操作中的读/写操作与地址空间中页面的读/写操作是紧密联系在一起的。前面已经介绍过，用户空间对象的操作表(地址空间对象的 a_ops 域)提供对地址空间对象的不同的操作函数。其中，readpage( )函数指针将索引节点页面中的内容读入主存。

由于实际读取数据的工作是由地址空间操作对象 readpage( )函数指针实现的，可以使用系统提供的辅助函数 do_sync_read( )来实现 read( )函数指针。这个函数将页面中的数据复制到用户空间的缓存中。如果页面不在页缓存中，该函数将等待直到 readpage( )函数将数据所在的页面装载到缓存中去。

(8) address_space_operations.readpage( )函数

在 Sparrow 文件系统中，使用系统提供的辅助函数 simple_readpage( )实现 readpage( )函数指针。

(9) file_operations.write( )函数

在 Sparrow 文件系统中，使用系统辅助函数 do_syn_write( )来注册文件操作对象中的 write( )函数指针。do_syn_write( )函数首先分配新的页面，然后调用地址空间对象上的 prepare_write( )函数指针，以使缓存前端(head)的对象能够被写入页，并在稍后通过 I/O 操作写入具体设备。do_syn_read( )函数将用户空间的数据复制到页面，并调用 commit_write( )函数指针将这些数据写到地址空间对象上。

使用 write( )函数将用户空间的数据写入缓存和在地址空间的将页面写入设备是紧密联系在一起的。地址空间中的 commit_write( )函数指针通常使用 simple_commit_write( )来实现，它将缓存标记为“脏”的以使稍后刷新时由块设备将数据写入磁盘介质。Sparrow 也是如此，simple_commit_write( )函数将在下面介绍。

(10) address_space_operations.commit_write( )函数

在 Sparrow 文件系统中，使用系统提供的辅助函数 simple_commit_write( )函数来实现地址空间操作对象中的 commit_write 函数指针。该函数指针的功能是将缓存中的数据写入到磁盘介质中。由于 Sparrow 文件系统是一个驻留在主存中的文件系统，不涉及磁盘介质的概念，所以仅仅将数据写入该文件在主存中对应的缓存中，并标记缓存是“脏”的。

(11) cleanup_module( )函数

作为内核模块实现的 Sparrow 文件系统在模块卸载时向内核解除注册。模块的计数会因为文件的函数而递增或递减。当一个模块的计数大于零时，这个模块将不会被删除。内核本身小心地维护着这些技术情况，所以不用去检查一个文件系统是否真在被使用(即不用担心一个正在使用的文件系统被卸载)。

4. Sparrow 文件系统的使用方法

(1) 编译内核模块

在 Linux 2.6 内核上编译代码，将其编译为内核模块。

创建一个 makefile 文件，代码如下。

```
ifneq (${KERNELRELEASE},)
obj-m += sfs.o
else
KERNEL_SOURCE := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

default:
    $(MAKE) -C ${KERNEL_SOURCE} SUBDIRS=$(PWD) modules
clean :
    rm *.o *.ko
endif
```

然后编译：

```
$ make
```

编译的结果是产生 sfs.ko 文件。

(2) 加载内核模块

以 root 身份加载该内核模块：

```
$ insmod sfs.ko
```

加载完成后，可以使用 lsmod 命令查看加载是否成功。

(3) 挂载文件系统

将文件系统挂载到根目录树的/mnt 目录下：

```
$ mount –t sfs any /mnt
```

挂载完成后，可以使用 mount 命令查看是否挂载成功

在/mnt 路径下的文件系统就是 Sparrow 文件系统，可以使用 mkdir、rmdir、touch 等命令创建/删除文件、创建/删除文件夹以及修改文件。

(4) 卸载文件系统

卸载文件系统使用命令：

```
$ umount /mnt
```

(5) 卸载内核模块

卸载 sfs 内核模块使用命令：

```
$ rmmod sfs
```

5. 程序框架

```
/*头文件*/
/*数据结构定义*/
```

```
#define SAMPLEFS_MAGIC 0x73616d70
extern struct inode_operations sfs_dir_inode_ops;
extern struct inode_operations sfs_file_inode_ops;
extern struct file_operations sfs_file_operations;
extern struct address_space_operations sfs_aops;

/*数据结构赋值*/
struct super_operations sfs_super_ops = {
        …
};

struct address_space_operations sfs_aops = {
        …
};

struct file_operations sfs_file_operations = {
        …
};

struct inode_operations sfs_file_inode_ops = {
        …
};

struct inode_operations sfs_dir_inode_ops = {
        …
};

static struct file_system_type sfs_fs_type = {
        …
};

struct dentry_operations sfs_dentry_ops = {
        …
};

struct dentry_operations sfs_ci_dentry_ops = {
        …
};

/*实现主要函数*/
```

```
static void sfs_put_super(struct super_block *sb) {
        …
}

static void sfs_parse_mount_options(char *options, struct sfs_sb_info * sfs_sb) {
        …
}

static int sfs_ci_hash(struct dentry *dentry, struct qstr *q) {
        …
}

static int sfs_ci_compare(struct dentry *dentry, struct qstr *a, struct qstr *b) {
        …
}

static struct backing_dev_info sfs_backing_dev_info = {
        …
};
```

```
struct inode *sfs_get_inode(struct super_block *sb, int mode, dev_t dev) {
      struct inode * inode = new_inode(sb);
        …
}

static struct dentry *sfs_lookup(struct inode *dir, struct dentry *dentry,
        struct nameidata *nd) {
        …
}

static int
sfs_mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t dev) {
        …
}

static int sfs_mkdir(struct inode * dir, struct dentry * dentry, int mode) {
      /*调用 sfs_mknod( )函数*/
}
```

```
static int sfs_create(struct inode *dir, struct dentry *dentry, int mode,
        struct nameidata *nd) {
/*调用 sfs_mknod( )函数 */
}

/*用以支持链接功能*/
static int sfs_symlink(struct inode * dir, struct dentry *dentry,
        const char * symname) {
        …
}

/*填写超级块信息*/
static int sfs_fill_super(struct super_block * sb, void * data, int silent) {
        …
};

struct super_block * sfs_get_sb(struct file_system_type *fs_type,
   int flags, const char *dev_name, void *data) {
   /*调用 get_sb_nodev( )函数*/
}

/*模块初始化函数 */
static int __init init_sfs_fs(void) {
        printk(KERN_INFO "init sfs\n");
        …
        return register_filesystem(&sfs_fs_type);
}

/*模块清理函数*/
static void __exit exit_sfs_fs(void) {
        printk(KERN_INFO "unloading sfs\n");
        …
        unregister_filesystem(&sfs_fs_type);
}

/*注册模块初始化与清理函数*/
module_init(init_sfs_fs)
module_exit(exit_sfs_fs)

MODULE_LICENSE("GPL");
```

第 18 章　proc 文件系统

18.1　实 验 目 的

- 了解 proc 文件系统的作用和原理。
- 掌握 proc 编程，并通过该接口获取进程和内核信息。

18.2　背 景 知 识

18.2.1　proc 文件系统简介

早期 UNIX 系统在设备文件目录/dev 下设置一个/dev/mem 文件，其目的是提供一种便捷的用户和内核之间的通信和交互方式。/dev/mem 以文件系统作为使用界面，通过它可以读/写系统的整个物理主存，而物理主存的地址就用作读写时文件内的位移量，可像普通文件一样，进行读、写等常规文件操作，从而提供在用户空间动态地读写内核数据结构的方法。该方法既可以用于收集状态信息和统计信息，也可以用于程序调试或修改内核数据结构及变量值。在 UNIX 的发展过程中，该功能被进一步加强，在系统根目录下建立/proc 目录，演变成一个特殊的文件系统/proc，该文件系统目录中的特殊文件包含以下内容。

- 资源管理信息：如/proc/slabinfo 是主存管理中关于各个 slab 缓冲块的信息、/proc/swaps 是系统的 swap 设备的信息、/proc/partitions 是磁盘分区的信息。
- 设备相关信息：如/proc/pci 是 PCI 总线上所有设备的列表信息。
- 文件系统信息：如/proc/mounts 是系统中已经安装的文件系统设备的列表，/proc/filesystems 则是已经登记的每种文件系统（类型）的清单信息。
- 中断编号信息：如/proc/interrupts 是关于中断源和它们的中断向量编号的清单信息。
- 模块相关信息：如/proc/modules 是系统中已安装的动态模块的清单，/proc/ksyms 则是内核中供可安装模块动态连接的符号名及其地址的清单信息。
- 系统版本信息：包括号系统版本号，及其他各种统计与状态信息。
- 进程相关信息：每个进程都有一个子目录（以 pid 命名），子目录中包括关于该进程命令行、环境变量、CPU 占用时间、主存映射表、已打开文件的描述符、进程根目录及当前目录、进程的文件链接、及进程状态信息等。

应用程序只要有正确的访问权限，就可使用/proc 文件系统读写进程空间及操作系统整体组织与状态信息。在/proc 文件系统中，大多数文件都是只读的，用户只能从这些文件中读取内核中的信息。但也有一些/proc 文件被设置成可写，用户可以通过写这些/proc 文件将参数传递给内核，结果是直接修改了内核相应变量值。例如，/proc/sys/fs/file_max 指定可以分配的文件句柄数，如果不能打开更多文件，就可以增加该值；又如/proc/sys/kernel/shmall 指定系统可使用的共享主存总量，不够用时应修改该参数。除系统已经提供的子目录和文件，/proc 还留有接口，允许用户在内核中创建新的子目录或文件，从而与用户程序共享信息。例如，可以为函数日志程序（可作为驱动程序或内核模块）在/proc 文件系统中创建新的文件，用来显示函数的使用次数和使用频率等，也可以增加另外的文件，用于设置日志记录规则，如记录 open( )函数的使用情况等。

图 18-1 列出/proc 文件系统中的一些子目录及文件信息。

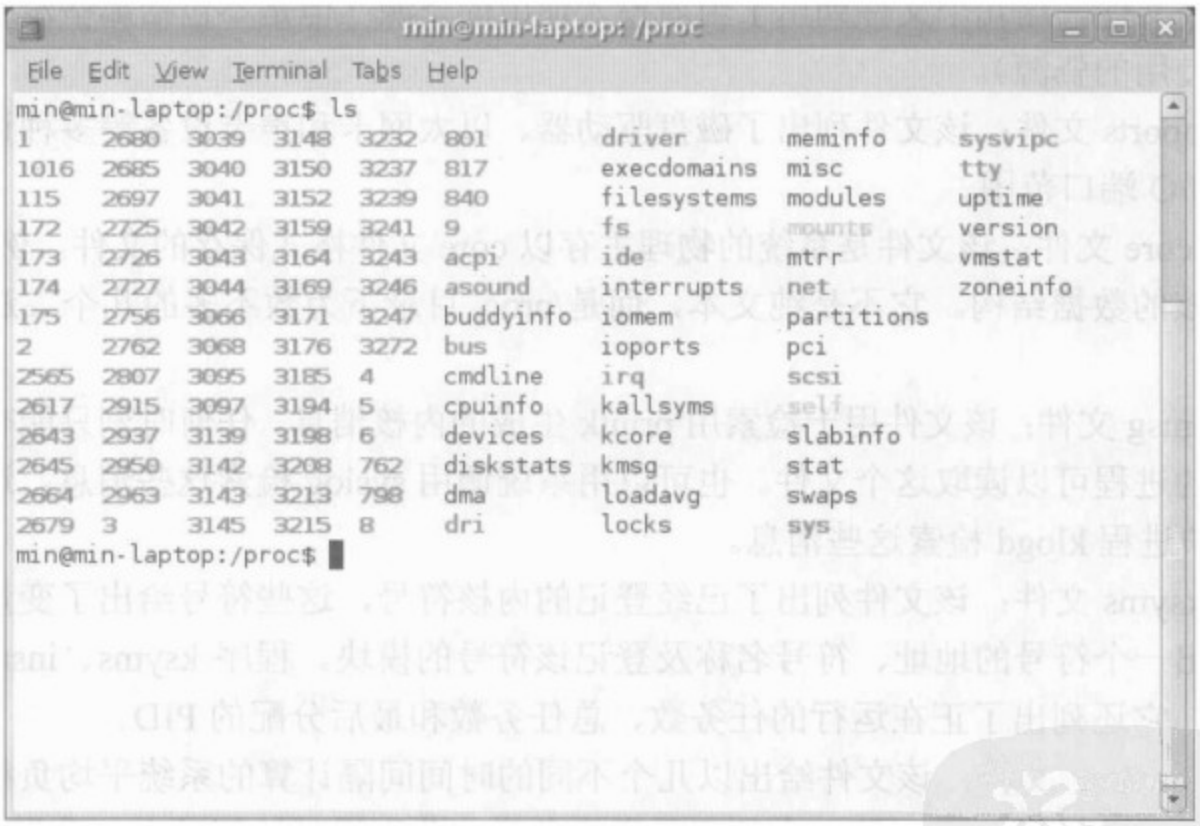


图 18-1 /proc 文件系统

• /proc/cmdline 文件：该文件给出了内核启动的命令行。它和用于进程的 cmdline 项非常相似。

• /proc/cpuinfo 文件：该文件提供了有关系统 CPU 的多种信息。这些信息是从内核里对 CPU 的测试代码中得到的。该文件列出了 CPU 的普通型号（386、486、586、686 等），以及能得到的更多特定信息（制造商，型号和版本）。文件还包含了以 bogomips 表示的 CPU 速度，而且如果检测到 CPU 的多种特性或者 bug，文件还会包含相应的标志。这个文件的格式为：文件由多行构成，每行包括一个域名称，一个冒号和一个值。

• /proc/devices 文件：该文件列出字符和块设备的主设备号，以及分配到这些设备号的设备名称。

• /proc/dma 文件：该文件列出由驱动程序保留的 DMA 通道和保留它们的驱动程序名称。casade 项供用于把次 DMA 控制器从主控制器分出的 DMA 行所使用；这一行不能用于其他用途。

• /proc/filesystems 文件：该文件列出可供使用的文件系统类型，一种类型一行。虽然它们通常是编入内核的文件系统类型，但该文件还可以包含可加载的内核模块加入的其他文件系统类型。

• /proc/interrupts 文件：该文件的每一行都有一个保留的中断。每行中的域有中断号、本行中断的发生次数、可能带有一个加号的域（SA_INTERRUPT 标志设置）以及登记这个中断的驱动程序的名字。可以在安装新硬件前，像查看/proc/dma 和/proc/ioports 一样用 cat 命令手工查看手头的这个文件。这几个文件列出了当前投入使用的资源（但是不包括那些没有加载驱动程序的硬件所使用的资源）。

• /proc/ioports 文件：该文件列出了磁盘驱动器、以太网卡和声卡设备等多种设备驱动程序登记的许多 I/O 端口范围。

• /proc/kcore 文件：该文件是系统的物理主存以 core 文件格式保存的文件。例如，GDB 能用它考察内核的数据结构。它不是纯文本，而是/proc 目录下为数不多的几个二进制格式的项之一。

• /proc/kmsg 文件：该文件用于检索用 printk 生成的内核消息。任何时刻只能有一个具有超级用户权限的进程可以读取这个文件。也可以用系统调用 syslog 检索这些消息。通常使用工具 dmesg 或守护进程 klogd 检索这些消息。

• /proc/ksyms 文件：该文件列出了已经登记的内核符号，这些符号给出了变量或函数的地址。每行给出一个符号的地址、符号名称及登记该符号的模块。程序 ksyms、insmod 和 kmod 使用该文件。它还列出了正在运行的任务数、总任务数和最后分配的 PID。

• /proc/loadavg 文件：该文件给出以几个不同的时间间隔计算的系统平均负载，这就如同 uptime 命令显示的结果一样。前 3 个数字是平均负载，分别通过计算过去 1 分钟、5 分钟、15 分钟里运行队列中的平均任务数得到，随后是正在运行的任务数和总任务数，最后是上次使用的进程号。

• /proc/locks 文件：该文件包含在打开的文件上的加锁信息，文件中的每一行描述了特定文件和文档上的加锁信息以及对文件施加的锁的类型。内核也可在需要时对文件施加强制性锁。

图 18-2 列出/proc 文件系统中的每个进程子目录中包含的主要内容。

假定/proc/pid 表示进程标识 pid 为第 1 级子目录，每一进程下都包含与该进程相关的状态文件。

• /proc/pid/status：相应进程的状态。

• /proc/pid/ctl：相应进程的控制文件。

图 18-2 /proc 文件系统中每个进程子目录包含信息

- /proc/pid/psinfo：相应进程的 ps 信息。
- /proc/pid/as：相应进程的地址空间。
- /proc/pid/map：相应进程的映射信息。
- /proc/pid/object：相应进程的对象信息。
- /proc/pid/sigact：相应进程的信号量操作。
- /proc/pid/sysent：相应进程的系统调用信息。
- /proc/pid/lwp/tid：相应进程的内核线程标示符目录。
- /proc/pid/lwp/tid/lwpstatus：内核线程的状态。
- /proc/pid/lwp/tid/lwpctl：内核线程控制文件。
- /proc/pid/lwp/tid/lwpsinfo：内核线程的 ps 信息。

18.2.2 proc 文件系统数据结构

在/proc 文件系统中，代表各个文件的是 proc_dir_entry 结构，该结构和文件系统中的 dir_entry 类似，管理对用户空间和内核空间的/proc 文件的驱动。proc_dir_entry 是/proc 文件系统中最重要的数据结构，系统初始化期间的主要工作之一就是建立/proc 文件系统树。/proc 文件系统树保存完成读写/proc 文件系统所需要的几乎所有关系、属性和操作函数指针等。由于/proc 文件系统没有外部设备，只存在主存里，因此在读操作时，不像其他文件系统一样从辅存中获取索引节点信息，而是从/proc 树中读取对应索引节点，然后再调用索引节点中登记的函数，动态地从内核读取所需信息。在外部看来，/proc 文件系统就和其他文件系统一样，觉察不出有

任何区别。

系统启动后，创建由 proc_dir_entry 结构形成的文件系统树，每当从用户空间读取/proc 目录下的文件时，内核根据读取的文件映射到对应的函数，动态地获得内核数据。除了提供读出功能外，/proc 文件系统的部分文件还提供写入功能。/proc 文件系统的超级块和普通文件系统的超级块不同，它并不需要从硬件中获得超级块中的数据，而是在内核启动时直接初始化超级块数据，从而完成系统对/proc 文件系统操作函数的初始化过程。

/proc 文件系统的编程接口与 VFS 类似，大部分/proc 文件系统中的函数是 VFS 的函数名加一个“proc_”。例如，在/proc 文件系统中创建目录的函数是 proc_mkdir( )，创建符号链接的函数是 proc_symlink( )，创建设备文件的函数是 proc_mknod( )。

在/proc 文件系统中，proc_dir_entry 数据结构的定义为：

```
struct proc_dir_entry {              /*描述/proc 文件系统的目录结构*/
  unsigned short low_ino;
  unsigned short namelen;
  const char *name;
  mode_t mode;
  nlink_t nlink;
  uid_t uid;
  gid_t gid;
  unsigned long size;
  struct inode_operation *proc_iops;
  struct file_operations *proc_fops;
  get_info_t *get_info;
  struct module *owner;
  struct proc_dir_entry *next, *parent, *subdir;
  void *data;
  read_proc_t *read_proc;
  write_proc_t *write_proc;
  atomic_t count;
  int deleted;
  kdev_t rdev;
};
```

在该结构中，常用到的一些成员包括：

- name：/proc 文件名。
- namelen：/proc 文件名长度。
- mode：文件的类型和权限。
- nlink：读文件的链接数。
- count：用户计数。

- proc_iops：inode 节点操作函数。
- proc_fops：file 文件操作函数。
- read_proc：读操作函数。
- write_proc：写操作函数。
- data：保存与目录相关的数据。
- owner：该文件的所有者模块。

一个这样的数据结构代表一个/proc 文件，很多成员的含义与普通文件的相同。创建新文件时，需要将该结构初始化，再将它传递给内核。当用户读写/proc 文件时，该文件所关联的读/写函数就会被激活，所以 proc_dir_entry 结构中的读/写函数应该预先设置好。

读文件接口的函数原型是：

```
int (*read_proc) (char *page, char **start, off_t off, int count, int *eof, void *data);
```

参数 page 指针是写数据给用户的数据缓冲区；start 指明要把数据写在哪一个位置；offset 指定写入数据相对于缓冲区的偏移量；count 为写入的字节数；eof 是一个返回参数，当文件指针已经指向文件尾时，该标志被置位；data 一般用作保存私有数据。

在读文件接口中有两个偏移量：start 和 offset。两个参数看似重复，但是它们都有各自的用途。/proc 文件只有单个主存页面供用户数据传输。这样就把用户文件的大小限制在 4 KB。start 参数在这里用于实现大数据量文件。如果 read_proc( )函数不对 start 指针进行设置，内核就会假定 offset 参数被忽略，并且数据页包含了返回给用户空间的整个文件。反之，如果需要通过多个片段创建更大的文件，则可以把 start 赋值为 page，这样内核就能知道新数据位于缓冲区的开始位置，并会跳过前 offset 字节的数据。

写文件接口的函数原型是：

```
int (*write_proc) (struct file *file, const char *buffer, unsigned long count, void *data);
```

该函数将从 buffer 开始的缓冲区中的 count 个字节写入 file。由于 buffer 一般是用户空间的指针，指向的是用户空间缓冲区。因此，应该使用 copy_from_user( )将数据复制到内核空间中来。

当定义好 read_proc( )和 write_proc( )函数后，就需要将它们加载到/proc 文件系统中。加载的第 1 步是获得一个 proc_dir_entry 结构，获得 proc_dir_entry 结构的函数是：

```
struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode,
    struct proc_dir_entry *parent );
```

参数 name 是要创建的文件名；mode 是文件的保护掩码(缺省系统范围时可以作为 0 传递)；parent 指出要创建的文件的目录(如果 parent 是 NULL，文件在/proc 根目录下创建)；函数的返回值是要创建的/proc 文件所对应的 proc_dir_entry。例如，如果要在创建/proc/test 这个文件，可以使用：

```
create_proc_entry("test", 0, NULL);
```

当获得 proc_dir_entry 后，第二步就是对该数据结构设置 read_proc( )和 write_proc( )指针。

如果想创建一个只读的 proc 文件，还有一个更加方便的函数：

```
struct proc_dir_entry *create_proc_read_entry(const char *name, mode_t mode,
        struct proc_dir_entry *base, read_proc_t *read_proc, void *data);
```

该函数实际上就是调用 create_proc_entry( )，并将返回结构中的 read_proc 域设置成 read_proc，data 域设置成 data。

/proc 文件系统还提供多个函数方便在/proc 文件系统中创建或者删除文件或目录，主要的函数包括：

- proc_mkdir( )：创建目录。
- remove_proc_entry( )：删除文件或目录。
- proc_symlink( )：创建符号链接。
- proc_mknod( )：创建设备文件。

这几个函数的用法和 VFS 中相应函数的用法类似，可以查询相应手册了解具体细节。

18.3 实 验 内 容

18.3.1 实验 1 向 proc 文件系统中添加可读写文件

18.3.1.1 实验说明

编写一个模块，当模块被加载时，该模块会向/proc 文件系统中添加一个文件。用户可以向该文件中输入一个字符串。当用户读取该文件时，返回用户前一次输入的字符串。

18.3.1.2 解决方案

1. 数据结构

模块需要记录用户输入的字符串，因此需要有一个字符数组记录用户的输入。为了简化编程，将用户输入字符串的最大长度限定在 1 024 B。在这个前提下，只需要定义一个长度为 1 024 B 的字符数组即可。如果要接受任意长度的用户输入，可以通过内核提供的动态主存分配机制实现。

```
#define STRINGLEN     1024
char global_buffer[STRINGLEN];
```

2. 创建和删除文件

加载模块时，需要创建一个可读写的/proc 文件。在本实验中，将在/proc 目录下创建一个 hello 文件。创建/proc 文件有多种方法，例如通过 create_proc_entry( )函数：

```
hello_file = create_proc_entry("hello", 0644, NULL);
```

该函数中，hello 指明文件名；0644 指明该文件是属主可读写，其他用户可读的文件；NULL

表明该文件是创建在/proc 目录下的。

当文件创建成功后，create_proc_entry( )将返回一个 proc_dir_entry 给模块。这时模块需要将 proc_dir_entry 的读写函数设置好。当设置好以后，用户读写相应的/proc 文件便会激活预先设置好的读写函数。

```
hello_file->read_proc = proc_read_hello;
hello_file->write_proc = proc_write_hello;
hello_file->owner = THIS_MODULE;
```

在该段代码中，读函数被设置成 proc_read_hello( )；写函数被设置成 proc_write_hello( )。

卸载模块时，模块需要删除创建的/proc 文件，模块可以通过 remove_proc_entry( )删除 proc 文件：

```
remove_proc_entry("hello",NULL);
```

与 create_proc_entry 类似，hello 指定文件名；NULL 表明要删除的文件是在/proc 目录下的。

3. proc_read_hello( )与 proc_write_hello( )

proc_read_hello( )与 proc_write_hello( )两个函数是用户读写/proc 文件时激活的函数。调用 proc_read_hello( )时，只需要将 global_buffer 中的内容返回给用户即可。由于 global_buffer 的大小只有 1 024 B，程序可以直接通过 sprintf( )将 global_buffer 写入用户的地址空间：

```
len = sprintf (page, "last message: %s\n", global_buffer);
```

调用 proc_write_hello( )时，程序需要检查用户的输入是否过大。如果用户输入的字符数量超过 1 024 B，proc_write_hello( )应该将用户的输入截断，然后才从用户空间中将用户的输入写入 global_buffer。

```
… /*验证输入*/
retval = copy_from_user(global_buffer, buffer, len);/*将用户输入复制到 global_buffer 中*/
```

18.3.1.3 程序框架

```
/*头文件*/
/*数据结构*/

    /*读/proc/hello 时调用的函数*/
int proc_read_hello ( char *page, char **start, off_t off, int count, int *eof, void *data)
{
    /*复制 global_buffer 到用户空间*/
}

    /*写/proc/hello 时调用的函数 */
int proc_write_hello( struct file *file, const char *buffer, unsigned long count,void *data)
{
    /*检查输入*/
```

```
        /*将用户输入复制到内核空间*/
}

        /*模块初始化函数 */
static __init int hello_init( )
{
        /*创建一个 proc_dir_entry*/
        /*设置文件的读写函数 */
}

        /*模块清理函数*/
static __exit void hello_exit( )
{
        /*删除/proc 文件
}

        /*注册模块初始化与清理函数*/
module_init(hello_init);
module_exit(hello_exit);
```

18.3.2 实验 2 通过 proc 文件系统查看进程信息

18.3.2.1 实验说明

编写一个模块，加载模块时，模块会向/proc 文件系统中添加一个文件（/proc/task）。用户可以向该文件中输入需要查看的进程 pid，用户读取该文件时，返回进程控制块中的信息。

18.3.2.2 解决方案

1. proc 文件操作

在本实验中，内核模块需要从用户空间读取需要查询的 pid，并将进程控制块信息写入用户空间。实验中对/proc 文件的操作与实验 1 类似，可以参考实验 1 的代码。

2. 将字符串转换成 pid_t 类型

用户以字符串的形式将 pid 传递给内核模块，需要将该字符串转换成表示 pid 的 pid_t 类型。pid_t 类型在内核中的定义为：

```
typedef int _kernel_pid_t;
typedef _kernel_pid_t pid_t;
```

由此可知，pid_t 类型是用 int 表示。可以编写一个函数将将字符串转换成 int 类型，得到相应的 int 值之后，只需要将该值直接转换成 pid_t 即可。

3. 显示进程信息

可以通过 find_process_by_pid( )查找到相应的进程控制块，该函数原型为：

```
static inline struct task_struct *find_process_by_pid(pid_t pid);
```

通过进程控制块，内核模块能够获得进程的各种信息，如进程所用的环境变量、进程的命令行参数。进程的可执行路径存储在进程描述符的 comm 成员中，可以通过 printk 直接将进程的可执行路径输出。如果需要显示进程的环境变量等信息，可以通过访问进程 mm_struct 结构显示。进程环境变量所占主存空间的起始地址存储在 mm_struct 的 env_start 和 env_end 成员。

18.3.2.3 程序框架

```
/*头文件*/
/*数据结构*/

/*读/proc/task 时调用的函数*/
int proc_read_task ( char *page, char **start, off_t off, int count, int *eof, void *data)
{
…
}

/*写/proc/task 时调用的函数 */
int proc_write_task( struct file *file, const char *buffer, unsigned long count, void *data)
{
    /*将字符串转换成为 pid_t 类型*/
    /*显示命令行信息*/
}

    /*模块初始化函数 */
static_init int task_init( )
{
    …
}

    /*模块清理函数 */
static_exit void task_exit( )
{
    …
}

    /*注册模块初始化与清理函数 */
module_init(task_init);
module_exit(task_exit);
```

第 19 章　设备驱动程序

19.1　实 验 目 的

- 理解 Linux 设备驱动程序的基本原理。
- 掌握设备驱动程序的编写原则和过程。
- 学会编写设备驱动程序。

19.2　背 景 知 识

19.2.1　基础知识

19.2.1.1　设备文件

Linux 函数(系统调用)是应用程序和操作系统内核之间的接口，而设备驱动程序是内核和硬件设备之间的接口，设备驱动程序屏蔽硬件细节，且设备被映射成特殊的文件进行处理。每个设备都对应一个文件名，在内核中也对应一个索引节点，应用程序可以通过设备的文件名来访问硬件设备。此外，设备文件也与普通文件一样，受到文件系统访问权限机制的保护。Linux 为文件和设备提供了一致性的接口，用户操作设备文件与操作普通文件类似。例如，通过 open( ) 函数可打开设备文件，建立起应用程序与目标设备的连接；之后，可以通过 read( )、write( )、ioctl( )等常规文件函数对目标设备进行操作。从应用程序的角度看，设备文件的逻辑空间是一个线性空间(起始地址为 0，每读取一个字节加 1)，从该逻辑空间到具体设备物理空间(如磁盘的磁道、扇区)的映射则由内核提供，并被划分为文件操作和设备驱动两个层次。

Linux 将硬件设备分成两大类：块设备和字符设备。相应地，提供了两个标准接口：块设备文件和字符设备文件。块设备支持面向块的 I/O 操作且数据可以被随机访问，传送任何数据块所需要的时间大致相同，所有 I/O 操作通过在内核空间中的 I/O 缓冲区进行，块设备的典型例子是磁盘、软盘和光盘等；字符设备支持面向字符的 I/O 操作，即逐个字符进行 I/O 操作，不经过系统 I/O 缓冲区，所以，需要管理自己的缓冲区结构。字符设备的数据主要被顺序访问(如声卡)，用于随机访问时，访问数据所需时间很大程度上依赖于数据在设备上的位置(如磁带)。键盘、显示终端、串口、并口、鼠标和网络适配器等均为字符设备。

19.2.1.2 主设备号与次设备号

设备文件通常放置在/dev 目录下，它是存放在文件系统中的实际文件。然而，v2.3.46 内核版本起正式引入设备文件系统 devf，所有设备文件作为一个可挂接的文件系统纳入文件系统的管理范围，而且它是一个 VFS(虚拟文件系统)，仅在系统主存中存在并可被挂接到任何需要的目录下。设备文件的索引节点并不对磁盘上的数据块编址，而是包含硬件设备相关信息的一个表示。每个设备文件除文件名和类型(字符设备或块设备)外，还有两个主要属性：主设备号和次设备号。通常，主设备号指明唯一的设备类型，即标识设备对应的驱动程序类型，它是块设备表或字符设备表中表项的索引。现代 Linux 内核允许多个驱动程序共享主设备号，但是大多数设备仍然按照“一个主设备号对应一个驱动程序”的原则组织；次设备号用于在一组主设备号相同的设备之间唯一标识特定设备，如两个硬件就可用次设备号区分。

表 19-1 给出了某计算机上的设备列表。

表 19-1 某计算机上的设备列表

访问权限	设备类型		主设备号、次设备号	安装日期	安装年份	设备名
brw-rw----	1 root	disk	3, 0	Mar 12	2000	had
brw-rw----	1 root	disk	3, 1	Mar 12	2000	hda1
brw-rw----	1 root	disk	8, 208	May 6	2001	sdn
brw-rw----	1 root	disk	8, 209	May 6	2001	sdn1
crw-------	1 root	root	4, 0	May 7	1999	tty0
crw-rw-rw-	1 root	root	1, 3	Mar 8	1999	null
crw-rw-rw-	1 root	root	1, 5	Mar 8	1999	zero
crw-rw-rw-	1 root	tty	2, 128	Mar 8	2001	ptyx0
crw-rw-rw-	1 root	tty	2, 132	Mar 8	2001	ptyx5
crw-rw-rw-	1 root	tty	3, 0	Mar 8	2000	ttyp0

在内核中，dev_t 类型用来保存设备编号，包括主设备号和次设备号。在 v2.6 内核版本中，dev_t 是一个 32 位数，其中 12 位用来标识主设备号，而其余 20 位用来标识次设备号。使用 dev_t 时，开发者应该采用系统提供的一组宏对设备号进行访问：

```
MAJOR(dev_t dev);                /*从 dev_t 类型中取出主设备号*/
MINOR(dev_t dev);                /*从 dev_t 类型中取出次设备号*/
MKDEV(int major, int minor);     /*将主设备号和次设备号转换成 dev_t 类型*/
```

mknod( )函数可用来创建设备文件，使用该函数需要 4 个参数：设备文件名、设备类型、主设备号及次设备号。用户也可通过 mknod 命令创建设备文件，例如：

```
mknod /dev/fd1 b 2 1
```

该命令在/dev 目录下创建一个名为 fd1 的块设备文件，文件的主设备号为 2，次设备号为 1。

设备文件的主设备号与次设备号放在索引对象的 i_rdev 字段，设备文件的类型存放在索引对象的 i_mode 字段。对于内核而言，设备名是无关紧要的，依靠主设备号和次设备号对设备进行标识。如果两个设备文件的主设备号与次设备号一致，并且类型一致，那么它们所标识的设备是同一个设备。

19.2.1.3 设备文件的 VFS 处理

用户通过相同的一组函数访问设备文件与普通文件。访问普通文件时，文件系统将用户的操作转换成对磁盘分区中数据块的操作；访问设备文件时，文件系统需要将用户的操作转换成对设备的驱动操作。

在 VFS 中，每个文件都有一个索引节点与之对应。在内核的 inode 结构中，有一个名为 i_fop 成员，其类型为 file_operations。file_operations 定义文件的各种操作，用户对文件的操作是通过调用 file_operations 来实现的。为了使用户对设备文件的操作能够转换成对设备的驱动操作，VFS 必须在设备文件打开时，改变其 inode 结构中 i_fop 成员的默认值，将该值替换成与该设备相关的具体函数操作。

当用户准备对设备文件进行访问时，文件系统读取设备文件在磁盘上相应的索引节点，并存入主存 inode 结构中。内核将文件的主设备号与次设备号写入 inode 结构中的 i_rdev 字段，并将 i_fop 字段设置成 def_blk_fops(如果为块设备)或 def_chr_fops(如果为字符设备)。通过这样的设置，用户对设备文件的操作便能转换成对设备的驱动操作。

19.2.2 字符设备

19.2.2.1 数据结构

在字符设备驱动程序中，主要涉及 3 个重要的内核数据结构，分别是 file_operations、file 和 inode。内核通过这 3 个数据结构的关联，将用户对设备文件的操作转换为对驱动程序相关函数的调用，进而实现对设备的驱动操作。

file_operations 是一组函数指针集合，其结构为：

```
struct file_operations {
    ...
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char_user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char_user *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
```

```
        int (*release) (struct inode *, struct file *);
        ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
        …
    };
```

在内核中，通过包含一个指向 file_operations 结构的 f_op 字段，每个打开的设备文件都会与一组文件操作函数相关联，如 open( )、release( )、read( )、write( )、ioctl( )等。file_operations 结构的每个字段都必须指向驱动程序中实现特定操作的具体函数，对于不支持的文件操作，对应字段可以设置为 NULL 值。对于 NULL 指针，内核具体的处理行为是不相同的，例如，如果将 readv 字段(该字段用于读取)设置成 NULL 指针，内核会通过多次调用 read( )来实现函数 readv( )。

file 结构是驱动程序需要使用的第 2 个重要数据结构，file 代表一个打开了的文件。它由内核在使用 open( )函数时建立，并传递给该文件上进行操作的所有函数，直到最后的 close( )函数。当文件的所有操作结束后，内核会释放该数据结构。在 file 结构中，一些比较重要的成员含义如下：

```
struct file{
    …
    mode_t f_mode;          /*文件模式，用于标记文件是否可读或可写*/
    loff_t f_pos;           /*当前的读/写位置*/
    file_operations *f_op;  /*与文件相关操作。内核执行 open( )操作时对该指针赋值*/
    void *private_data;     /*驱动程序可将这个字段用于任何目的，但是要在 release( )方法中，释放
                              该字段占用的主存*/
    …
}
```

内核用 inode 结构在内部标识文件，它和 file 结构不同，后者标识打开的文件描述符。对于单个文件，可能会有许多个表示打开的文件描述符的 file 结构，但它们都指向同一个 inode 结构。对于编写字符驱动程序，inode 结构中有两个重要的字段：

```
struct cdev *i_cdev;
dev_t   i_rdev;
```

i_cdev 结构表示字符设备的内核数据结构，当 inode 指向一个字符设备文件时，该字段包含指向 struct cdev 结构的指针。i_rdev 包含设备编号，内核不推荐开发者直接通过访问 inode 结构的 i_rdev 字段来获得设备的主、次设备号，而提供两个宏供开发者使用：

```
unsigned int imajor(struct inode *inode);    /*获得主设备号*/
unsigned int iminor(struct inode *inode);    /*获得次设备号*/
```

19.2.2.2 申请与释放设备编号

在创建字符设备之前，需要给设备申请设备编号。静态申请设备编号的内核函数为 register_chrdev_region( )，其原型为：

```
int register_chrdev_region(dev_t from, unsigned count, const char *name);
```

其中 from 是要分配的设备编号范围的起始值，count 是所请求的连续设备编号的个数。如果 count 非常大，则所请求的范围可能和下一个主设备号重叠。name 是和该编号范围关联的设备名称。在使用 register_chrdev_region( )时，开发者必须明确知道所需要的设备编号。一部分主设备号已经静态地分配给大部分常见设备，内核源代码的 documentation/devices.txt 文件中有这个列表。对于这些设备，由于设备号已经预先确定，因此使用 register_chrdev_region( )函数没有什么问题。如果对于一些新设备，设备号并没有预先确定，在这种情况下，驱动程序需要对设备号进行动态申请。

动态申请块设备号的内核函数原型为：

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name);
```

其中，dev 用于输入参数，在成功完成调用后将保存已分配范围的第 1 个编号。baseminor 是要使用的被请求第 1 个次设备号。count 和 name 的含义与 register_chrdev_region( )中的含义一样。使用动态申请设备号的缺点在于每次分配的主设备号不能够保持一致，因此用户不能够预先在/dev 目录中用 mknod 命令创建设备文件。用户需要在驱动程序注册后，根据/proc/devices 读取到设备所用到的设备编号，然后创建相应的设备文件。

当设备不再使用这些设备编号时，驱动程序需要将申请的设备编号释放。动态申请和静态申请一样，都采用 unregister_chrdev_region( )释放设备编号，该内核函数的原型为：

```
void unregister_chrdev_region(dev_t from, unsigned count);
```

驱动程序一般通过模块加载，可在模块的初始化函数进行设备号的申请，并在模块的清理函数中，对设备号进行释放。

19.2.2.3 设备注册与注销

内核用 cdev 结构来表示字符设备，在内核调用设备的驱动操作之前，必须分配并注册一个上述结构。分配和初始化 cdev 结构有两种方式，如果开发者打算在运行时获得一个独立的 cdev 结构，可以采用如下代码：

```
struct cdev *my_cdev = cdev_alloc( );
my_cdev->ops = &my_fops;
```

开发者也可以通过内核函数 cdev_init( )来进行 cdev 结构的初始化。在 cdev 结构设置好之后，最后的步骤是通过 cdev_add( )内核函数将 cdev 结构添加到内核中，其原型为：

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count);
```

这里，p 是指向 cdev 结构的指针，dev 是该设备对应的第 1 个设备编号，count 是应该和该设备关联的设备编号的数量。如果 cdev_add 成功返回，驱动程序就随时可被内核调用了，因此，驱动程序在没有完全准备好时，不应该调用 cdev_add( )。

要从系统中移除一个字符设备，执行如下调用：

```
void cdev_del(struct cdev *dev);
```

cdev 结构被传递到 cdev_del( )后，就不应该再访问 cdev 结构。

19.2.3 块设备

19.2.3.1 申请与释放设备编号

块设备驱动程序在使用前，必须向内核注册主设备号，该内核函数原型为：

```
int register_blkdev(unsigned int major, const char *name);
```

其中，major 是需要注册的主设备号，如果 major 为 0，内核将动态地分配一个主设备号，并将该设备号返回；name 为驱动程序在/proc/devices 中显示的名字。在驱动程序卸载时，原先注册的主设备号需要归还给内核，执行该任务的内核函数原型为：

```
int unregister_blkdev(unsigned int major, const char *name);
```

其中参数的含义与 register_blkdev( )一致。

19.2.3.2 注册磁盘

驱动程序通过内核函数 register_blkdev( )注册主设备号后，并不能让系统使用任何磁盘。内核中由 gendisk 结构表示一个独立的磁盘设备，块设备驱动程序在注册主设备号之后，需要设置 gendisk 结构，并将该结构添加到内核中。实际上，内核还用 gendisk 结构表示分区，但是驱动程序开发者并不需要了解这些。gendisk 数据结构如下：

```
struct gendisk {
    int major;                          /*块设备的主设备号*/
    int first_minor;                    /*块设备的第 1 个次设备号*/
    int minors;                         /*块设备包含的次设备号数量*/
    char disk_name[32];                 /*磁盘设备名称*/
    struct block_device_operations *fops;
    struct request_queue *queue;
    void *private_data;                 /*供驱动程序自由使用的字段*/
    sector_t capacity;                  /*磁盘包含的扇区数*/
    int flags;                          /*描述驱动器状态*/
};
```

如果磁盘设备是可被分区的，每个分区都需要分配一个次设备号。minors 成员通常取 16，它使一个磁盘可以包含 15 个分区；如果设备不支持分区，minors 值则设置为 1。fops 是块设备提供给内核的接口，内核通过这些接口完成用户对设备的驱动操作。queue 成员是块设备所用到的请求队列，请求队列的作用将在后面介绍。capacity 是以 512 B 为一个扇区时，该设备所包含的扇区数；驱动程序不能直接设置该值，而要通过 set_capacity( )内核函数来进行设置。

gendisk 是一个动态分配的结构，它需要内核的特殊处理来进行初始化，驱动程序不能自己动态分配该结构，而是必须执行内核函数：

```
struct gendisk *alloc_disk(int minors);
```

参数 minors 是该磁盘使用的次设备号的数量。gendisk 在申请后，其 minors 值是不能够再次被改变的。用户不再需要使用 gendisk 时，必须通过 del_gendisk( )函数将 gendisk 结构删除。驱动程序将 gendisk 结构设置完毕后，需要将 gendisk 注册进内核。gendisk 的注册内核函数是：

```
void add_disk(struct gendisk *gd);
```

一旦调用 add_disk，内核就有可能随时操作 gendisk。因此 gendisk 只有在初始化完毕后才能调用 add_disk 进行注册。

在 gendisk 结构中，fops 是块设备驱动程序提供给内核的接口。fops 的类型为 block_device_operations，该结构主要成员如下：

```
int (*open) (struct inode *, struct file *);
int (*release) (struct inode *, struct file *);
int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
int (*media_changed) (struct gendisk *);
int (*revalidate_disk) (struct gendisk *);
```

open( )、release( )和 ioctl( )的作用与字符设备中对应函数的功能类似。设备被打开或者关闭时，会调用 open( )、release( )函数。media_changed( )内核函数用于检查用户是否更换了驱动器内的介质，如果用户更换过，那么返回一非 0 值。当介质被更换时，内核调用 revalidate_disk( )内核函数作出响应；它告诉驱动程序完成必要的工作，以便使用新介质。

block_device_operations 中并没有负责读和写的操作函数，在块设备中，对设备的读写是通过请求队列来实现的。

19.2.3.3 bio 结构体

在大多数情况下，磁盘控制器采用 DMA 方式进行数据传输。块设备驱动程序只需要向磁盘控制器发送一条指令，它就能够一次性传输主存中的多块数据。待数据传输完毕后，磁盘控制器再发出一个结束中断，通知设备驱动程序。老式磁盘控制器在采用 DMA 方式时，只允许传送主存中连续的数据。新的磁盘控制器支持聚散 I/O，它能够传输主存中不邻接的数据块。

Linux 早期内核版本中，块设备 I/O 通过操作内核 I/O 缓冲区(I/O buffer)进行。自 v2.5 内核版本起，为块设备引入一种新型、灵活的容器 bio 结构体来实现 I/O。该结构体代表正在活动的、以片段(segment)链表形式组织的块 I/O 操作，一个片段是需要传输的一小块连续的主存缓冲区。这样，就不需要保证单个缓冲区一定要连续。通过用片段来描述缓冲区，即使一个缓冲区分散在主存的多个位置上，bio 结构体也能对内核保证 I/O 操作的执行，像这样的向量 I/O 就是所谓的聚散 I/O。bio 结构体定义如下：

```
struct bio {
    sector_t            bi_sector;      /*磁盘上相关扇区*/
    struct bio          *bi_next;       /*请求队列链表*/
    struct block_device *bi_bdev;       /*相关的块设备*/
    unsigned long       bi_flags;       /*状态和命令标志*/
```

```
    unsigned long       hi_rw;          /*读还是写？ */
    unsigned short      bi_vcnt;        /*bio_vec 的个数*/
    unsigned short      bi_idx;         /*bi_vec 数组当前元素的序号*/
    unsigned short      bi_phys_segments;  /*合并后的片段数*/
    unsigned short      bi_hw_segments;    /*DMA 重映射时的片段数*/
    unsigned int        bi_size;        /*总的大小、字节为单位*/
    unsigned int        bi_hw_front_size;  /*第 1 个可合并的段大小*/
    unsigned int        bi_hw_back_size;   /*最后一个可合并的段大小*/
    unsigned int        bi_max_vecs;    /*bio_vecs 数目上限*/
    struct bio_vec      *bi_io_vec;     /*指向 bio_vec 链表*/
    bio_end_lO_t        *bi_end_io;     /*I/O 完成后调用的方法*/
    atomic_t            bi_cnt;         /*使用计数*/
    void                *bi_private;    /*所有者的私有方法*/
    bio_destructor_t    *bi_destructor; /*销毁方法*/
};
```

使用 bio 结构体的目的主要是代表正在现场执行的 I/O 操作，所以该结构体中的主要域都是用来管理相关信息的，几个最重要的成员是 bi_io_vec、bi_vcnt 和 bi_idx，其结构关系如图 19-1 所示。

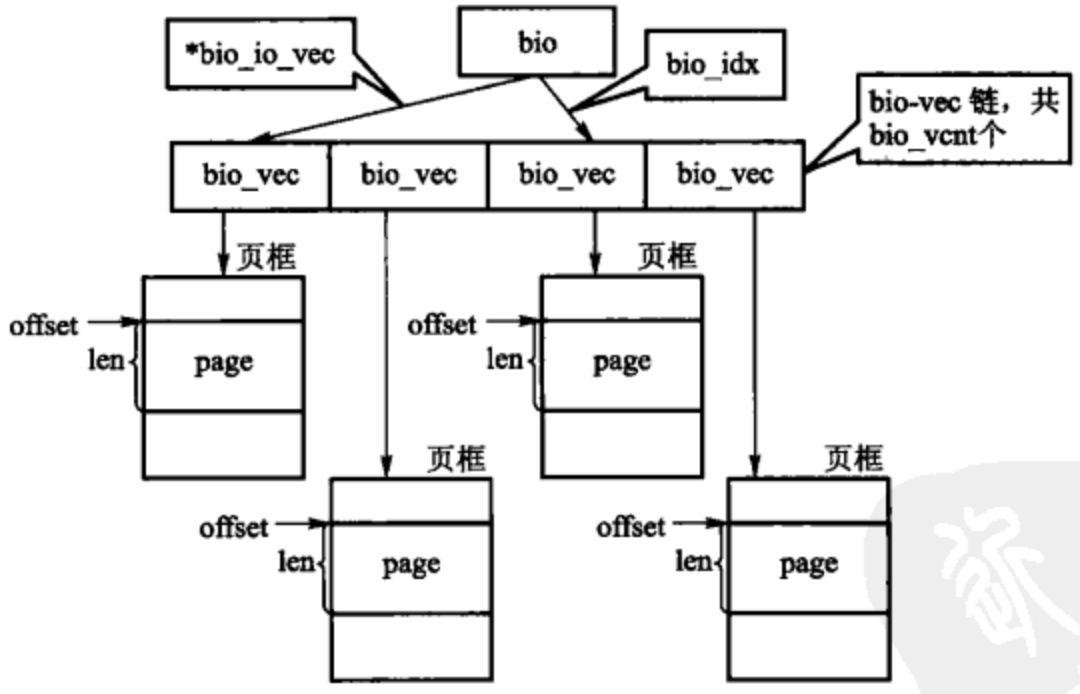


图 19-1 bio、bio_vec 及 page 的关系

bi_io_vec 域指向一个 bio_vec 结构体数组，该结构体链表包含一个特定 I/O 操作所需要使用到的所有片段。每个 bio_vec 结构都是一个形式为＜page，offset，len＞的向量，它描述一个特定片段：片段所在的页框、块在页框中的偏移位置、从给定偏移量开始的块长度。实际上，bio_vec 结构描述 I/O 缓冲区，整个 bio_io_vec 结构体数组表示一个完整的缓冲区。bio_vec 结

构定义为：

```
struct bio_vec{
    struct page     *by_page;     /*指向该缓冲区所驻留的页框*/
    unsigned int    by_len;       /*该缓冲区以字节为单位的大小*/
    unsigned int    bv_offset;    /*缓冲区所驻留的页框中以字节为单位的偏移量*/
};
```

在每个给定的块 I/O 操作中，bi_vcnt 域用来描述 bi_io_vec 所指向的 bio_vec 数组中的向量数目，当块 I/O 操作执行完毕后，bi_idx 域指向数组的当前索引。可见，每次块 I/O 请求都通过一个 bio 结构体表示，每个请求包含一个或多个块，这些块存储在 bio_vec 结构体数组中。这些结构体描述每个片段在页框中的实际位置，并且像向量一样被组织在一起。I/O 操作的第 1 个片段由 bi_io_vec 结构体所指向，其他片段在其后依次放置，共有 bi_vcnt 个片段。当块 I/O 层开始执行请求需要使用各个片段时，bi_idx 域会不断更新，从而总指向当前片段。

bi_idx 域指向数组中的当前 bio_vec 片段，块 I/O 层通过它可以跟踪块 I/O 操作的完成进度，但该域更重要的作用在于分割 bio 结构体，可应用于 RAID 盘。bi_cnt 域记录 bio 结构体的使用计数，如果该域值减为 0，就应该销毁该 bio 结构体，并释放它占用的主存。通过 bio_get( )和 bio_put( )内核函数增加使用计数或减少使用计数，在操作正在活动的 bio 结构体时，一定要先增加它的使用计数，以免在操作过程中该 bio 结构体被释放；相反，在操作完毕后，要减少使用计数。bi_private 域是一个属于所有者的私有域，只有创建了 bio 结构的所有者可以读写该域。

在每次启动一个新的 I/O 操作时，需要通过内核函数 bio_alloc( )申请一个新的 bio 结构。bio 结构通常是通过 slab 分配器分配的，但是当主存不足时，内核会使用一个备用的 bio 小主存池，内核也为 bio_vec 结构提供主存池。结构 biovec_pool 描述 bio_vec 对象池，将若干个 bio_vec 对象组成大小不同的 slab，从而对 bio_vec 对象按不同的数量进行分级管理。结构 biovec_pool 定义为：

```
struct biovec_pool {
    int nr_vecs;
    char *name;
    kmem_cache_t *slab;
    mempool_t *pool;
};
```

对象池数组定义 6 个 biovec_pool 结构实例，即有 6 个不同大小 slab 的 bio_vec 池，例如其大小为 64 或 128 个主存连续页框，从而有利于块的读写操作。块 I/O 的 slab 在初始化时由 init_bio( )创建，具体过程如下：

```
static int _init_bio(void)
  {
    bio_slab = kmem_cache_create("bio", sizeof(struct bio), 0,
          SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL, NULL);
    bio_pool = mempool_create(BIO_POOL_SIZE, mempool_alloc_slab,
```

```
            mempool_free_slab, bio_slab);
        if (!bio_pool)
            panic("bio:can't create mempool\n");
        biovec_init_pools( );
        …
        return 0;
    }
```

操作 bio 结构体的内核函数如下：

bio_alloc( )、bio_get( )和 bio_put( )管理 bio 结构体的分配、引用计数和 bio 结构体的释放。bio_alloc( )调用 bvec_alloc( )给一个非克隆的 bio 结构体分配 bio_vec 链表，它将建立 6 个不同的对象池，bio_alloc( )将从这 6 个 slab 中分配给定大小的 bio_vec 链表。如果 bio 结构体被访问，bio_get( )用来在提交 I/O 时将增加 bio 结构体的引用计数。bio_clone( )用来克隆一个 bio 结构体，该 bio 结构体与原 bio 结构体共享一个 bio_vec 链表。宏 bio_for_each_segment 和 rq_for_each_bio 用来遍历 I/O 请求中的每个 bio 结构体及每个片段。

19.2.3.4 请求队列

块设备将它们挂起的块 I/O 请求保存在请求队列中，该队列由 request_queue 结构体表示，包含一个双向请求链表及相关控制信息。通过内核中像文件系统这样高层的代码将请求加入到队列中。请求队列只要不为空，队列对应的块设备驱动程序就会从队列头获取请求，然后将其送入对应的块设备上去。请求队列表中的每一项都是一个单独的请求，由 request 结构体表示。因为一个请求可能要操作多个连续的磁盘块，所以每个请求可以由多个 bio 结构体组成，虽然磁盘上的块必须连续，但是在主存中这些块并不一定要连续，每个 bio 结构体都可以描述多个片段，而每个请求也可以包含多个 bio 结构体。

块 I/O 层建立请求结构体 request 后，把它放在请求队列中，然后传递给底层驱动程序，触发驱动程序的 request_fn( )内核函数来处理请求，驱动程序利用块 I/O 层的 elv_next_request( )内核函数把下一个请求移出队列。请求结构体 request 定义如下：

```
struct request {
    struct list_head queuelist;              /*请求队列链表*/
    unsigned long flags;                     /*标志*/
    sector_t sector;                         /*将提交的下个扇区*/
    unsigned long nr_sectors;                /*提交请求中的所有扇区数*/
    unsigned int current_nr_sectors;         /*当前 bio 结构体中当前片段将提交的扇区数*/
    sector_t hard_sector;                    /*将完成的下个扇区*/
    unsigned int hard_nr_sectors;            /*将完成的剩余扇区数*/
    unsigned long hard_cur_sectors;          /*当前片段中将完成的剩余扇区数*/
    unsigned long nr_cbio_sectors;           /*当前 bio 结构体将提交的剩余扇区数*/
    struct bio *cbio;                        /*将提交的下个 bio 结构体*/
```

```
    struct bio *bio;                        /*将完成的下个未完成的 bio*/
    struct gendisk *rq_disk;                /*通用磁盘结构*/
    char *buffer;
    int ref_count;                          /*引用计数*/
    request_queue_t *q;                     /*请求队列*/
    struct requeue_list *rl;                /*请求的空闲链表*/
    unsigned short nr_phys_segments;        /*合并物理地址后的片段数*/
    …
};
```

请求队列的结构定义如下：

```
struct request_queue {
    struct list_head queue_head;
    struct request *last_merge;
    elevator_t elevator;
    struct request_list rq;                 /*请求空闲表结构*/
    request_fn_proc *request_fn;            /*指向块设备底层操作函数*/
    merge_request_fn *back_merge_fn;        /*向后合并请求函数*/
    merge_request_fn *front_merge_fn;       /*向前合并请求函数*/
    merge_request_fn *merge_request_fn;     /*合并请求函数*/
    make_request_fn *make_request_fn;       /*生成请求函数*/
    …
    unsigned long nr_request;               /*最大请求数*/
    unsigned short max_sectors;             /*能处理的最大尺寸(扇区为单位) */
    unsigned short max_phys_segments;       /*一个请求中能处理的最大片段数*/
    unsigned short max_hw_segments;;        /*一个请求中能处理的最多 dma 片段数*/
    unsigned short hardsect_size;           /*硬件扇区大小*/
    unsigned int max_segment_size;          /*簇的最大片段尺寸，默认值 64KB*/
    struct blk_queue_tag *queue_tags;       /*队列标志*/
    …
};
```

块设备的读、写操作都不是通过从驱动程序注册时提供的 block_device_operations 结构中获得的，内核通过通用的 generic_file_read( )和 blkdev_file_write( )内核函数完成对设备的读写。当用户对设备发出读操作时，generic_file_read( )将一个读请求发送给设备驱动程序，并保存在该设备的请求队列中。内核对这些请求进行调度，并将请求交给驱动程序执行，写设备的过程也类似。

系统运行的快慢受文件系统访问速度的直接影响，而文件系统的访问行为往往是大量而无序的。在访问块设备时，磁头的移动最耗费时间。无序访问请求会让磁头不断改变位置，从而影响整个系统的性能。为了解决这个弊端，内核引入请求队列，将对设备的读写请求放入请求队列，并通过 I/O 调度程序按照被访问扇区的位置进行优化排序，尽量保证访问时磁头沿直线

移动。

一个请求队列就是一个动态的数据结构，该结构必须由块设备的 I/O 子系统创建，创建和初始化请求队列的内核函数是：

```
request_queue_t *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock);
```

其中，rfn 是驱动程序提供的一个函数指针，该函数是真正用于实现设备的读/写请求的；lock 是驱动程序提供的一个自旋锁，内核通过该自旋锁实现请求队列的同步保护；函数的返回值是一个请求队列的指针。请求队列在初始化后，应该通过 blk_queue_hardsect_size( )设置请求队列所使用的扇区大小。硬件扇区大小作为一个参数放在请求队列中，而不是放在 gendisk 结构中。当请求队列不再使用时，驱动程序需要将请求队列归还给内核，执行该任务的内核函数为：

```
void blk_cleanup_queue(request_queue_t * q);
```

调用该函数之后，不应该再次访问请求队列。请求队列中有一个成员为 queuedata，它是供驱动程序自行使用的，如果驱动程序用该成员指向一段动态分配的主存，那么卸载驱动程序时，要负责将该段主存回收。

请求队列的每一个读写请求在内核中用 request 结构定义，request 结构与驱动程序相关的成员有：

```
struct request {
    ...
    sector_t sector;
    unsigned long nr_sectors;
    char *buffer;
}
```

其中，sector 表示请求的开始扇区的索引号，该扇区号是指 512 B 的扇区编号，如果硬件使用不同大小的扇区，需要对扇区号进行转换；nr_sectors 表示需要传输的扇区数；buffer 是要传输或要接受数据的缓冲区指针，该指针在内核的虚拟地址中，驱动程序可以直接引用它。内核提供 rq_data_dir(struct request *req)宏用于获得请求类型；返回值为 0 表示从设备读取数据，非 0 表示向设备写数据。

当创建请求队列时，需要绑定一个 request( )内核函数，用于处理请求队列中的请求，其原型为：

```
void request(request_queue_t *queue);
```

块设备的读、写操作都是由 request( )完成的，下面对一个简单的 request( )进行说明：

```
void myrequest(request_queue_t *q)
    {
      struct request *req;
      while ((req = elv_next_request(q)) != NULL) {
          printk("sector=%d ,nr_sectors=%d", req->sector, req->nr_sectors);
          ...
```

```
            /*进行数据传输*/
            …
            end_request(req, 1);
        }
    }
```

内核提供内核函数 elv_next_request( )用来获得队列中第 1 个未完成的请求；当没有请求需要处理时，它返回 NULL。elv_next_request( )并不从队列中删除请求，如果不加以干涉而两次调用该内核函数，则两次都返回同一个 request 结构。在传输完数据后，调用 end_request( )通知内核已将该请求成功完成。

19.2.3.5 页高速缓存

页高速缓存是 Linux 内核实现的一种主要磁盘缓存，用来减少对磁盘的 I/O 操作次数，通过把磁盘中的数据缓存到物理主存中，把对磁盘的访问变为对物理主存的访问。它的价值在于：

- 访问磁盘的速度要远远低于访问主存的速度，因此，从主存访问数据比从磁盘访问速度更快。
- 数据一旦被访问，很有可能在短期内再次被访问到。基于程序局部性原理，如果在第 1 次访问数据时缓存它，那就极有可能在短期内再次被页高速缓存命中。

页高速缓存是由主存中的物理页组成的，缓存中每一页都对应着磁盘中的多个块。每当内核开始执行一个页 I/O 操作时，首先会检查需要的数据是否在高速缓存中，如果在，内核就直接使用高速缓存中的数据，从而避免访问磁盘。

也可以通过块 I/O 缓冲区把独立的磁盘块与页高速缓存联系在一起，一个缓冲区就是一个单独物理磁盘块在主存中的表示，缓冲区就是主存到磁盘块的映射描述符，因此通过缓存磁盘块以及缓冲块 I/O 操作，页高速缓存同样也可以减少块 I/O 操作期间的磁盘访问次数，这种缓存经常被称为“缓冲区高速缓存”。v2.2 内核版本中独立设置缓冲区高速缓存和页高速缓存，v2.4 内核版本后，只有唯一的磁盘缓存——页高速缓存，而缓冲区高速缓存成为页高速缓存中的一部分。

看一下页高速缓存封装的操作和数据，在页 I/O 操作中，如执行 read( )和 write( )时，经常用到页高速缓存。每次页 I/O 操作都要处理数据的全部页面，就需要对一个以上的磁盘块进行操作，所以页高速缓存实际缓存的是页面大小的文件块。块 I/O 操作每次操作一个单独的磁盘块，比如读写 inode 就是一个典型的块 I/O 操作。内核提供 bread( )底层内核函数，从磁盘读单个块，通过缓冲，这些磁盘块被映射到主存中相应的页面上，这样，也就被缓存到页高速缓存里，所以页高速缓存能减少大量的磁盘操作。

页缓冲区中的页来自读写普通文件、块设备文件和主存映射文件，在读写 I/O 操作之前，内核会检查数据是否已驻留在页缓冲区，以决定是否访问磁盘。Linux 页高速缓冲由 address_space 结构体描述：

```
struct address_space {
    struct inode_*hast;                          /*属主的索引节点*/
    struct list_head clean_pages;                /*“干净”页面链表*/
    struct list_head dirty_pages;                /*“脏”页面链表*/
    struct list_head locked_pages;               /*锁定页面链表*/
    struct list_head io_pages;                   /*I/O 使用页面链表*/
    struct address_space_operations *a_ops; /*操作函数表*/
    struct list_head i_mmap;                     /*私有映射链表*/
    struct list_head i_mmap_shared;              /*共享映射链表*/
    struct semaphore i_shared_sem;               /*保护链表信号量*/
    unsigned long nrpages;                       /*页面总数*/
    …
    struct list_head private_list; /*私有 address_space 链表
    struct address_space *assoc_mapping;    /*缓冲区*/
};
```

address_space 结构与某些对象相关联，这些对象就是页面内数据的属主，可能是普通文件、目录、块设备文件，也可能是交换区。通常为一个文件(即一个索引节点)，这时 host 域指向该索引节点。除此以外的情况，host 域置为 NULL。

a_ops 指向地址空间对象中的操作函数表，它由 address_space_operations 结构表示：

```
struct address_space_operations {
writepage( );                    /*把页写入磁盘*/
readpage( );                     /*从磁盘读入页*/
sync_page( );                    /*启动页中安排的 I/O 操作，传输数据*/
prepare_write( );                /*准备写操作 */
commit_write( );                 /*完成写操作*/
bmap( );                         /*从文件块索引获得逻辑块号*/
flushpage( );                    /*删除来自磁盘的页*/
releasepage( );                  /*日志文件系统准备释放页*/
direct_io( );                    /*数据页的直接 I/O 传输*/
};
```

19.2.3.6 pdflush 内核线程

由于页高速缓存的缓存作用，写操作实际上会被延迟，当页高速缓存中的数据比磁盘存储的数据更加新时，那么该数据就被称作脏数据，在主存中累积起来的脏页最终必须被写回磁盘。在以下情况发生时，脏页被写回磁盘：

- 当空闲主存低于特定的阈值时，内核必须将脏页写回磁盘，以便释放主存。
- 当脏页在主存中驻留时间超过特定的阈值时，内核必须将超时的脏页写回磁盘，以确保脏页不会无限期地驻留在主存中。

上面两项工作目的不同，在老版本的 Linux 内核中，由两个独立的内核线程 bdflush( )和 kupdated( )分别完成。但是在 v2.6 内核中，由一组内核线程——pdflush( )后台回写例程——统一执行此项工作。首先，pdflush( )内核线程在系统中的空闲主存低于特定的阈值时，将脏页刷新回磁盘。该后台回写例程的目的在于在可用物理主存过低时，释放脏页以重新获得主存。当空闲主存比阈值还低时，内核便会调用内核函数 wakeup_bdflush( )唤醒一个 pdflush( )线程，随后该线程进一步调用内核函数 background_writeout( )开始将脏页写回磁盘。内核函数 background_writeout( )会连续地写出数据，直到满足以下条件：

- 已经有指定的最小数目的脏页被写出到磁盘。
- 空闲主存数已经回升，超过了阈值规定数。

上述条件确保 pdflush( )内核线程的操作可以减轻系统中主存不足的压力，回写操作不会在达到这两个条件前停止，除非 pdflush( )写回了所有脏页。

为了满足第二个目标，pdflush( )内核线程会被周期性唤醒(和空闲主存是否过低无关)，将那些在主存中驻留时间过长的脏页写出，确保主存中不会有长期存在的脏页，以免如果系统发生崩溃，使那些在主存中还没来得及写回磁盘的脏页丢失。在系统启动时，内核初始化一个定时器，让它周期地唤醒 pdflush( )内核线程. 随后使其运行内核函数 wb_kupdate( )，它将把所有驻留时间超过一定时间的页写回。

19.2.3.7 块设备文件的读写操作

在读写块设备时，需要内核多个模块共同协助，需要协助的内核模块的结构如图 19-2 所示，进程通过对文件的操作完成对块设备的操作。

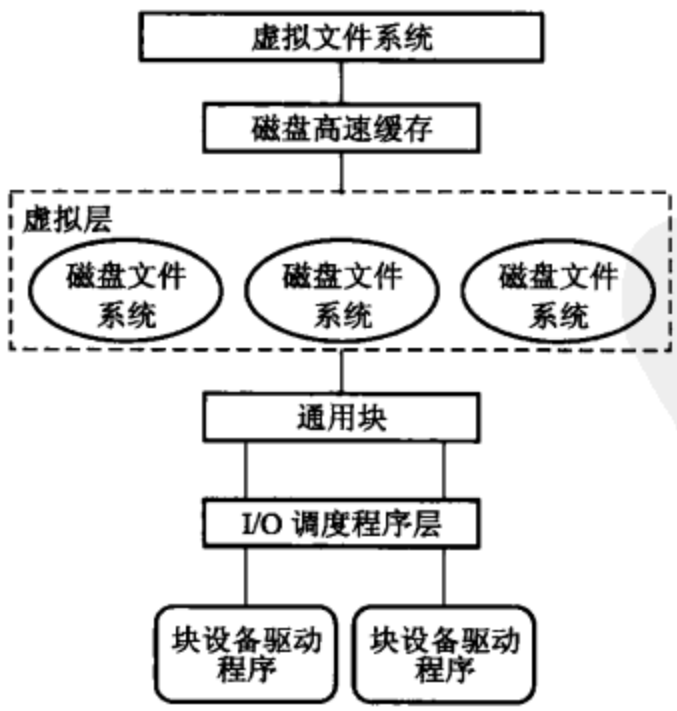


图 19-2 块设备涉及的模块

假设一个进程需要读取一个磁盘文件。其步骤为：

① read( )函数将文件描述符和文件内的偏移量，并传递给 VFS。

② VFS 确定需要获得的数据是否已经缓存在磁盘高速缓存中。如果数据存在，直接将数据回送给用户，VFS 便可返回。如果数据不在缓存中，则继续向下执行。

③ 内核需要通过内核映射层确定数据的物理位置，映射层主要执行下面两个步骤：

a) 文件被看成是由相同大小的块组成。映射层可以通过块大小、文件偏移量确定需要读取的数据相对于文件开始的块号。

b) 内核在步骤 a)中计算出来的块号只是相对于文件起始地址的块号，还需要将该块号转换成相对于磁盘/分区起始地址的物理块号。内核通过读取文件的索引节点，获得相应信息并计算出该块号。

④ 在获得数据在磁盘中的块号后，内核利用通用块层启动 I/O 操作来传送所请求的数据。一般而言，每个 I/O 操作只针对磁盘上一组连续的块。由于请求的数据不一定在邻接的块中，所以通用块层可能启动几次 I/O 操作，每次 I/O 操作通过一个 bio 结构描述。

⑤ 通用块层下的“I/O 调度程序”根据预先定义的内核策略将待处理的 I/O 数据传送请求进行归类。

⑥ 块设备驱动程序向磁盘控制器的硬件接口发送适当命令，从而进行实际的数据传送。

19.2.4 磁盘 I/O 调度程序

磁盘寻址是整个计算机中最慢的操作之一，尽量缩短磁头移动到特定块上某个位置的时间是提高系统性能的关键。为了优化寻址操作，内核会在提交驱动磁盘之前，先执行合并与排序预操作，这种预操作可以极大地提高系统的整体性能，在内核中负责提交 I/O 请求的子系统称为 I/O 调度程序。进程调度程序和 I/O 调度程序都是将一个资源虚化，并分给多个对象使用。对进程调度程序来说，CPU 被虚化并被系统中的可运行进程共享，这种虚化的效果是提供给用户多任务和分时操作系统。而 I/O 调度程序虚化磁盘设备，为多个磁盘读写请求服务，以便降低磁盘寻址时间，确保磁盘性能的最优化。

I/O 调度程序的工作是管理块设备的请求队列，它决定队列中的请求排列顺序以及在什么时刻派发请求到块设备，通过两种方法减少磁盘寻址时间：合并与排序。合并指将两个或多个请求结合成一个新请求。若有访问的磁盘扇区和当前请求访问的磁盘扇区相邻，那么这两个请求就可以合并为一个对单个和多个相邻磁盘扇区操作的新请求。通过合并请求，只需要传递给磁盘一条寻址命令，就可以访问到请求合并前必须多次寻址才能访问完的磁盘区域，因此合并请求显然能减少系统开销和磁盘 I/O 次数。

假设在读请求被提交给请求队列时，队列中并没有其他请求需要操作相邻扇区，此时就无法执行合并操作。但是如果存在一个请求，它要操作的磁盘扇区位置与当前请求的比较接近，那么 I/O 调度程序将整个请求队列按扇区增长方向排序，使所有请求按磁盘上扇区的排列顺序

有序排列。其目的不仅是缩短单独一次请求的寻址时间，更重要的在于，通过保持磁盘头以直线方向移动，能缩短所有请求的磁盘寻址时间。

下面介绍 Linux 的 3 种磁盘调度算法，但本章的实验并不涉及这些内容。有兴趣的用户可在用户空间编程和调试，模拟实现这些磁盘驱动调度算法。

19.2.4.1 Linus 电梯 I/O 调度

在 v2.4 内核中，Linus 电梯调度算法是默认的 I/O 调度算法，它执行合并与排序预处理。当有新的请求加入请求队列时，它首先会检查其他每个挂起的请求是否可以和新请求合并。Linus 电梯调度可以执行向前和向后合并，如果新请求正好连在一个现存的请求前，就是向前合并；相反如果新请求直接连在一个现存的请求后，就是向后合并。实际情况中，向前合并比向后合并要少得多。

如果合并尝试失败，就需要寻找可能的插入点，新请求在队列中的位置必须符合请求以扇区方向次序排序的原则。如果找到，新请求将被插入到该点；如果没有合适的位置，新请求就被加入到队列尾部。另外，如果发现队列中有驻留时间过长的请求，那么新请求也将被加入到队列尾部，即使插入后还要排序。这样做是为了避免由于访问相近磁盘位置的请求太多，从而造成访问磁盘其他位置的请求难以得到执行机会这一问题。不幸的是，这种方法并不很有效，会导致 I/O 请求饥饿现象的发生。

总结一下，当一个请求加入到请求队列中时，可能发生 4 种操作，它们依次是：

① 如果请求队列中已存在一个对相邻磁盘扇区操作的请求，那么新请求将和这个已经存在的请求合并成一个请求。

② 如果请求队列中存在一个驻留时间过长的请求，那么新请求将被插入到队列尾部，以防止其他旧的请求发生饥饿。

③ 如果请求队列中以扇区方向为序存在合适的插入位置，那么新的请求将被插入到该位置，保证队列中的请求是以被访问磁盘物理位置为序进行排列的。

④ 如果请求队列中不存在合适的请求插入位置，请求将被插入到队列尾部。

电梯调度算法有两个问题：一是由于队列动态更新的原因，相距较远的请求可能会延迟相当长的时间，导致饥饿；二是由于写请求通常是异步的，而读请求大部分是同步操作，在写一个大文件时，很可能将一个读请求堵塞很长时间，从而堵塞进程。为此，Linux 2.6 增加两种新磁盘调度算法：时限 I/O 调度算法和预期 I/O 调度算法，尽力确保处理期限到达的请求获得响应。

19.2.4.2 时限 I/O 调度和预期 I/O 调度

为克服这些问题，引入时限调度算法，它使用 3 个队列：读 FIFO 队列、写 FIFO 队列和电梯排序队列。每个新来请求被放置到电梯排序队列中，该队列与前面所述一致，此外，同样的请求还被放置在 FIFO 读队列(如果是读请求)或 FIFO 写队列(如果是写请求)中，这样，读和写

请求队列维护一个按请求发生时间为顺序的请求列表。对每个请求都有一个到期时间，对于读请求默认值为 0.5 s，对于写请求默认值为 5 s。通常，I/O 调度程序从排序队列中分派服务，当一个请求得到满足时，其将从电梯排序队列头部移走，同时也从对应的 FIFO 队列移走。然而，当 FIFO 队列头部的请求超过其到期时间时，调度程序将从该 FIFO 队列中派遣任务，取出到期请求，再加上接下来的队列中的几个请求，当然，任一个请求被服务时，其也从电梯排序队列中移出。所以，时限调度算法能克服“饥饿”和读写不一致的问题。

当存在很多同步读请求时，上述策略可能达不到预期效果。典型地，应用程序会在一个读请求得到满足且数据可用后，才会发出下一个读请求。在接受上次读请求的数据和发出下一次读请求之间有个很小的延迟，利用这个延迟，调度程序可转去服务其他等待的请求。由于局部性原理，相同进程的连续读请求会发生在相邻的磁盘块上，如果调度程序在满足一个读请求后能延迟一小段时间，检查是否有新的附近的读请求发生，则可以提高整个系统的性能，这就是时限调度算法的原理。

预期调度是对时限调度的补充。分派一个读请求时，预期调度程序会将调度程序的执行延迟若干毫秒(取决于配置文件)。在延迟时段中，发出上一条读请求的应用程序有机会发出后继读请求，且该请求发生在相同的磁盘区域。如果是这样，新的请求会立刻获得服务；如果没有新请求发生，则调度程序继续使用时限调度算法。

19.2.4.3 公平排队 I/O 调度

完全公平的排队(Complete Fair Queuing，CFQ)I/O 调度程序是为专有工作负荷设计的，推荐给桌面系统使用，但它对多种工作负荷均能提供良好的性能。

CFQ 调度程序把进入的 I/O 请求放入特定队列中，这种队列是根据引起 I/O 请求的进程来组织的，来自不同进程的 I/O 请求进入不同的 I/O 请求队列。在每个队列中，刚进入的请求与相邻请求合并在一起，并进行插入分类，队列由此按扇区方向排序。CFQ 调度程序的差异在于每个提交 I/O 的进程都有自己的队列。

CFQ 调度程序以时间片轮转调度请求队列，从每个请求队列中选取请求数，默认值为 4，可以进行设置，然后进行下一轮调度。这就在进程级提供了公平性，确保每个进程接收公平的磁盘带宽片断。预定的工作负荷是多媒体播放，这种公平的算法可以保证让音频播放器总能够及时从磁盘再填满它的音频缓冲区，不过，实际上 CFQ 调度程序在很多场合都能很好地工作。

19.3 实 验 内 容

实验 实现一个基于主存的虚拟块设备驱动程序

1. 实验说明

该块设备使用主存作为设备硬件，换句话说，它是一个基于内存的磁盘驱动程序，当该驱

动程序挂载后，用户可以和普通块设备一样在该块设备上创建文件系统，且可通过模块挂载该驱动程序，可以手动设置虚拟块设备的主设备号，并且可以设置块设备的扇区大小和扇区数量。

2. 解决方案

(1) 基本数据结构

为了管理虚拟块设备，首先要为虚拟块设备设计一个管理数据结构。对于一个块设备，基本的成员应该包括：

- struct request_queue *queue：为该设备关联的请求队列指针。
- spinlock_t lock：用于供请求队列使用的自旋锁。
- struct gendisk *gd：块设备在内核中用 gendisk 结构表示，虚拟块设备需要保存该结构的指针。
- u8 *data：数据数组。
- int size：块设备大小(扇区数)。

基于以上定义，虚拟块设备的基本数据结构为：

```
struct vblkdev {
    int size;
    u8 *data;
    spinlock_t lock;
    struct request_queue *queue;
    struct gendisk *gd;
};
```

用户在加载虚拟块设备时，可以定义主设备号，并将扇区大小和扇区数量传递给驱动程序。为了实现该功能，模块中需要定义 3 个模块参数：

- vblkdev_major：设备的主设备号。
- hardsect_size：扇区大小。
- nsectors：扇区数量。

在 12.2.2.1 小节，已经学习过如何定义模块参数，这 3 个参数在模块中的定义如下：

```
static int vblkdev_major = 0;
static int hardsect_size = 512;
static int nsectors = 1024;
module_param(vblkdev_major, int, 0);
module_param(hardsect_size, int, 0);
module_param(nsectors, int, 0);
```

以上代码为模块参数定义了默认值。当用户不传递参数给驱动程序时，默认的主设备号为 0(表明驱动程序将动态获得一个主设备号)；扇区大小为 512 B；扇区数量为 1 024 个。

(2) 设备初始化

设备初始化主要涉及到如下几个任务：

- 申请并注册主设备号。
- 为虚拟块设备管理数据结构申请主存。
- 为虚拟块设备申请主存。
- 初始化请求队列。
- 初始化 gendisk 结构。

在这几个任务中，前 3 个任务比较直观，可以通过以下代码实现：

```
{
    …
    vblkdev_major = register_blkdev(vblkdev_major, "vblkdev");/*申请并注册主设备号*/
    …
    dev = kmalloc(sizeof(struct vblkdev), GFP_KERNEL); /*为管理数据结构申请主存*/
    memset(dev, 0, sizeof(*dev));
    …
    dev->size = nsectors * hardsect_size;
    dev->data = vmalloc(dev->size); /*为虚拟块设备申请主存*/
    …
}
```

接下来需要对请求队列进行初始化，初始化代码如下：

```
{
    …
    spin_lock_init(&dev->lock);
    dev->queue = blk_init_queue(vblkdev_request, &dev->lock);
    blk_queue_hardsect_size(dev->queue, hardsect_size);
    dev->queue->queuedata = dev;
    …
}
```

该段代码对请求队列需要使用的自旋锁进行初始化，并将请求队列的 request( )内核函数绑定为 vblkdev_request( )。值得注意的是，在该段代码最后，请求队列的 queuedata 被设置成虚拟块设备的管理数据结构。这使得将要介绍的 request( )能够很方便地获得虚拟块设备的管理数据结构，并通过该数据结构获得虚拟块设备的数据数组，进而完成数据的读写请求。

初始化的最后一步工作是设置 gendisk 结构，并将该结构添加到主存中。在初始化 gendisk 后，需要对 gendisk 设置容量。因为块设备使用的扇区大小和内核默认的扇区大小可能不一致，因此在设置 gendisk 容量时需要根据内核的扇区大小进行转换。

```
{
    …
    dev->gd = alloc_disk(MINOR_NO);
    dev->gd->major = vblkdev_major;
    dev->gd->first_minor =1;                    /*只有一个分区*/
```

```
        dev->gd->fops = &vblkdev_ops;
        dev->gd->queue = dev->queue;
        snprintf(dev->gd->disk_name, 32, "vblkdev");
        set_capacity(dev->gd, nsectors * (hardsect_size / KERNEL_SECTOR_SIZE));
        add_disk(dev->gd);
        …
    }
```

在 gendisk 的初始化过程中，程序对 gendisk 的 fops 字段进行设置。对于本次实验的虚拟块设备，只需要 fops 中操作设置成空操作即可。

(3) request( )内核函数

在请求队列的初始化过程中，需要绑定 request( )，由该内核函数完成数据的读/写操作。在本虚拟块设备中，request( )的实现为 vblkdev_request( )。vblkdev_request( )的实现大体框架和请求队列小节中实现的 request( )类似。该函数通过将请求队列中的每一个请求取出，并根据请求在虚拟块设备的数据数组中进行读写操作。在取出请求前，程序需要获得虚拟块设备数据数组的位置。数据数组记录在 vblkdev 数据结构中。在请求队列初始化时，程序已经将该结构的指针设置在请求队列的 queuedata 成员中，因此程序能够方便的获得虚拟块设备数据数组的位置。

虚拟块设备的 request( )内核函数只需要支持基本的读/写请求。对于其他请求，只需要将请求抛弃即可。内核提供 blk_fs_request( )内核函数供驱动程序判定是否为读写请求，如果该函数返回 0，则该请求不是一个读写请求。代码框架如下：

```
static void vblkdev_request(request_queue_t *q)
{
    …
    struct request *req;
    struct vblkdev *dev = q->queuedata;
    …
    while ((req = elv_next_request(q)) != NULL) {
        if (! blk_fs_request(req)) {
            printk (KERN_NOTICE "Skip non-fs request\n");
            end_request(req, 0);
            continue;
        }
        /*将数据写入数组*/
        end_request(req,1);
    }
}
```

request 结构中包含用户需要读写的扇区号、扇区数量等信息，根据该结构中的信息便可以将数据写入虚拟块设备的数据数组，或从数据数组中读取数据。读者可以自行完成此段代码。

(4) 使用虚拟块设备

在编写好驱动程序后，首先需要通过 insmod 加载模块。如果用户没有通过模块为设备指定主设备号，那么虚拟块设备将被动态分配一个主设备号。虚拟块设备使用的主设备号可以在模块加载之后通过/proc/devices 文件获得。获得主设备号之后，用户可以通过 mknod 创建设备文件。至此，用户便可以和使用普通块设备一样使用虚拟块设备了。用户可以通过使用 mkfs.ext2 等程序在该设备上创建文件系统，然后可以使用 mount 命令挂载该设备到某个目录，然后像普通磁盘设备一样使用。

3. 程序框架

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/spinlock.h>
#include <linux/genhd.h>
#include <linux/blkdev.h>
#include <linux/errno.h>
#include <linux/bio.h>
#include <linux/hdreg.h>

static int vblkdev_major = 0;                /*主设备号*/
static int hardsect_size = 512;              /*扇区大小*/
static int nsectors = 1024;                  /*扇区数量*/
module_param(vblkdev_major, int, 0);         /*定义模块参数*/
module_param(hardsect_size, int, 0);
module_param(nsectors, int, 0);

#define KERNEL_SECTOR_SIZE 512
/*定义设备结构*/
struct vblkdev {
        int                     size;        /*扇区大小*/
        u8                      *data;
        spinlock_t              lock;
        struct request_queue    *queue;
        struct gendisk          *gd;
};

#define MINOR_NO 1

/*进行数据传送*/
static void vblkdev_transfer(struct vblkdev *dev, unsigned long sector,
```

```
                unsigned long nsect, char *buffer, int write)
{
        /*计算偏移量；*/
        /*通过 memcpy 写入数据；*/
}

        /*传送一个 bio 结构体*/
static int vblkdev_xfer_bio(struct vblkdev *dev, struct bio *bio)
   {
        int i;
        struct bio_vec *bvec;
        sector_t sector = bio->bi_sector;

        /*单独执行每段*/
        bio_for_each_segment(bvec, bio, i)
        {
                char *buffer = __bio_kmap_atomic(bio, i, KM_USER0);
                通过 vblkdev_transfer 写入数据；
                sector += bio_cur_sectors(bio);
                _bio_kunmap_atomic(bio, KM_USER0);
        }
        return 0;
}

/*传送一个请求*/
static int vblkdev_xfer_request(struct vblkdev *dev, struct request *req)
   {
        struct bio *bio;
        int nsect = 0;

        rq_for_each_bio(bio, req) {
        /*传送一个 bio 结构体*/
        }
        return nsect;
}

/*处理请求*/
static void vblkdev_request(request_queue_t *q)
{
        struct request *req;
```

```
        int sectors_xferred;
        struct vblkdev *dev = q->queuedata;

        while ((req = elv_next_request(q)) != NULL)
        {
            /*忽略非读写命令*/
            if (! blk_fs_request(req)) {
                printk (KERN_NOTICE "Skip non-fs request\n");
                end_request(req, 0);
                continue;
            }
            /*通过 vblkdev_xfer_request 传输一个请求*/
        }
}

/*打开设备*/
static int vblkdev_open(struct inode *inode, struct file *filp)
   {
        try_module_get(THIS_MODULE);
        struct vblkdev *dev = inode->i_bdev->bd_disk->private_data;
        filp->private_data = dev;

        spin_lock(&dev->lock);
        check_disk_change(inode->i_bdev);
        spin_unlock(&dev->lock);

        return 0;
}

/*关闭设备*/
static int vblkdev_release(struct inode *inode, struct file *filp)
{
        module_put(THIS_MODULE);
        return 0;
}

/*定义块设备操作函数*/
static struct block_device_operations vblkdev_ops = {
        .owner      = THIS_MODULE,
        .open       = vblkdev_open,
```

```
        .release = vblkdev_release
};

/*初始化虚拟设备*/
static int setup_device(struct vblkdev *dev)
{
        /*分配 dev 结构所需主存*/
        /*初始化请求队列*/
        /*初始化 gendisk*/
}

struct vblkdev *device;

/*模块初始化函数*/
static int __init vblkdev_init(void)
{
        /*注册块设备*/
        return 0;
}

/*模块清理函数*/
static void __exit vblkdev_exit(void)
{
        /*删除设备所占用的内存空间*/
}

module_init(vblkdev_init);
module_exit(vblkdev_exit);
MODULE_LICENSE("Dual BSD/GPL");
```

附　录

附录 A　vi 编辑器

vi(visual interpreter)是 UNIX/Linux 操作系统使用的全屏幕文本编辑器，功能强大、使用方便，是程序员编写程序的得力工具。vi 有 3 种操作状态：命令模式、插入模式和末行模式。

- 命令模式：当执行 vi 后，首先进入命令模式，此时输入的任何字符都被视为命令。命令模式用于控制屏幕光标移动、文本字符/字/行删除、移动复制某区段，以及进入插入模式或进入末行模式。
- 插入模式：在命令模式下输入相应的插入命令进入该模式。只有在插入模式下，才可进行文字数据输入及添加程序代码，按 Esc 键可回到命令模式。
- 末行模式：在命令模式下输入某些特殊字符，如“/”、“？”和“：”，可进入末行命令模式。在该模式下可储存文件或退出编辑器，也可设置编辑环境，如寻找字符串，列出行号等。

vi 的工作模式如下图所示。相关命令见表 A-1～表 A-6。

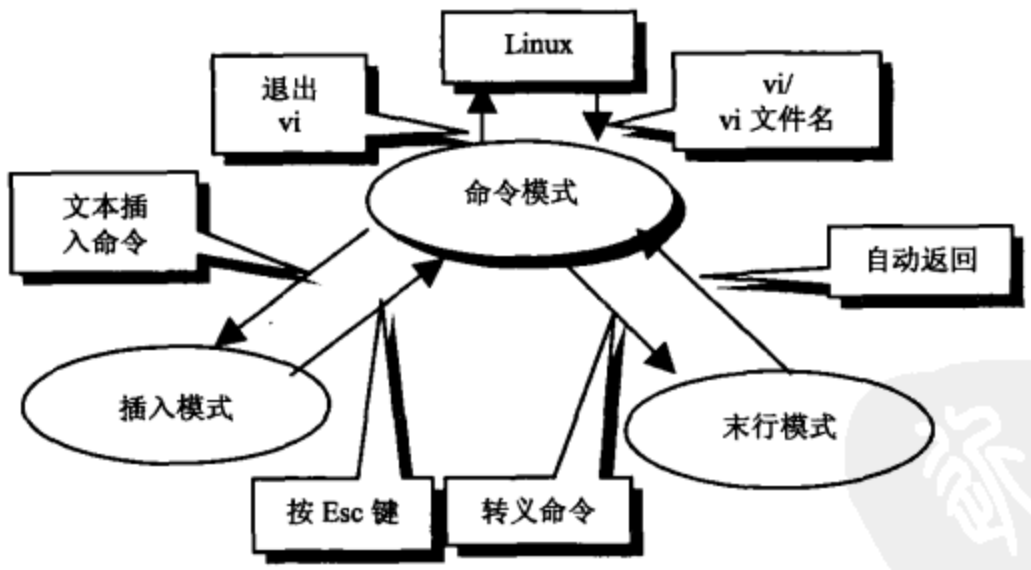


表 A-1　进入、退出 vi 模式和切换命令

命 令 类 型	命 令 形 式	说　　明
进入 vi 命令	$vi 文件名	进入 vi，显示 vi 编辑窗并载入文件，并进入命令模式
退出 vi 命令(退出 vi 时，若在插入模式，先用 Esc 返回命令模式)	:q!	放弃编辑内容，退出 vi
	:wq 或:zz	保存文件，退出 vi
	:w	保存文件，但不退出 vi
	:q	退出 vi，若文件被修改过，要确认是否放弃修改的内容

续表

命令类型	命令形式	说明
进入末行模式(命令模式下，输入特殊字符进入末行模式)	/	进入末行命令模式
	?	进入末行命令模式
进入插入模式(命令模式下，执行下列命令均可进入插入模式)	i、I	插入命令
	a、A	附加命令
	o、O	打开命令
	s、S	替换命令
	c、C	修改命令
	r、R	取代命令

表 A-2 命令模式常用命令——光标移动

命令	说明
h(←)	向左移一个字符
l(→)	向右移一个字符
k(↑)	向上移一个字符
j(↓)	向下移一个字符
0(Home)	光标移至行首
$(End)	光标移至行末
^	光标移至该行的第一个非空白字符处
H	光标移至窗口的第 1 行
M	移至窗口的中间行
L	移至窗口的最后一行
Z	当前行变为窗口第 1 行
nZ	第 *n* 行变为窗口第 1 行
Ctrl+F(PageDown)	向前翻一页
Ctrl+B(PageUp)	向后翻一页
Ctrl+D	向后翻半页
Ctrl+U	向前翻半页

表 A-3 命令模式常用命令——删除和修改

命令	说明
x(delete)	删除光标所在字符
X	删除光标所在位置前面一个字符
nx	删除从光标开始到光标后 *n*–1 个字符
dw	删除光标到下一个单词起始位置
ndw	删除光标起的 *n* 个字

续表

命　令	说　明
dd	删除光标所在行
ndd	删除包括光标所在行的 *n* 行
r	修改光标所在字符，r 后跟为修正字符
R(Insert)	进入替换状态，直到按 Esc 回到命令模式为止
s	删除光标所在字符，并进入输入模式
S	删除光标所在的行，并进入输入模式
u	恢复刚才被修改的文本
U	恢复光标所在行的所有修改
.	重复上一次命令的操作

表 A-4　命令模式常用命令——复制

命　令	说　明
Y	复制当前行至编辑缓冲区
nY	复制当前行开始的 *n* 行至编辑缓冲区
p	将编辑缓冲区的内容粘贴到光标的后面
ap	将编辑缓冲区 a 的内容粘贴到光标的后面
P	将编辑缓冲区的内容粘贴到光标的前面
J	下一行拼接在当前行之后

表 A-5　插入模式常用命令

命　令	说　明
a	从光标所在位置后面开始插入文本
A	从光标所在行尾开始新增文本
i	从光标所在位置前面开始插入文本
I	从光标所在行的第 1 个非空白字符前插入文本
o	在光标所在行下新增一行并进入输入模式
O	在光标所在行上新增一行并进入输入模式
Esc	从插入模式切换为命令模式

表 A-6　末行模式常用命令

命　令	说　明
:w>>文件名	内容写到文件原有内容后
:x	对修改信息存盘，退出 vi
:r 文件名	将文件调入编辑缓冲区
:e!	另行编辑文件并放弃编辑缓冲区内容

续表

命 令	说 明
:e 文件名	编辑名为 filename 的文件
:s/old/new	将当前行中遇到的第 1 个字符串 old 改为 new
:s/old/new/g	将当前行中遇到的所有字符串 old 改为 new
:%s/old/new/g	对所有行的内容完成字符串 old 改为 new
:set nu	显示行号
:set nonu	不显示行号
:set all	显示环境设置
/exp	从光标往前查找字符串 exp
?exp	从光标往后查找字符串 exp
:!cmd	在程序运行中执行 Shell 命令 cmd

附录 B emacs 编辑器

emacs 宏编辑器(macro editor)是自由软件基金会发行的软件产品，由 Richard Stallman 编写。emacs 除编辑功能外，增加了操作系统应支持的用户环境功能，如发送电子邮件、浏览器网页、新闻阅读、日志功能等，另外还包括了 Lisp 语言的解释执行功能。emacs 功能强大，几乎可解决用户与操作系统交互中的所有问题。

emacs(Emacs 22.1.1)用菜单方式完成文档编辑和其他各种功能，包括的主菜单项及完成的功能有以下几种

(1) Buffers

该菜单用于完成文本编辑中缓冲区的管理，具体命令如下。

- Scratch：对缓冲区中保留的内容进行查询。
- Messages：对缓冲区中信息的管理。
- Buffer List：列出缓冲区中的信息列表。
- List All Buffers：对系统中存在的多个缓冲区的管理，如完成缓冲区的剪切、粘贴等操作。

(2) File

该菜单完成对编辑中所用到的文件、目录、Frame、窗口等内容的管理：

- Open File：打开文件。
- Open Directory：打开目录。
- Save As：将缓冲区另存为。
- Revert Buffer：恢复缓冲区。
- New Frame：创建新的 Frame。

• Split Window：分割窗口。

(3) Tools

该菜单包括在 emacs 中完成多种扩充功能的内容，如文件打印、文件比较、文件映射、文件查询路径设置、文件版本控制、发送与接收电子邮件、程序编译、程序调试等。

(4) Edit

该菜单主要完成文档编辑功能。

(5) Options

该菜单主要完成其他杂项的设置与管理，如使用语言设置、文字类型设置输入/输出方法设置、编码风格设置等。

(6) Help

该菜单用于连接联机帮助文档。

emacs 中常用功能键见表 B-1。

表 B-1　emacs 中的常用功能键

功　能　键	说　　明
Ctrl+B	光标向后移动一个字符
Ctrl+N	光标下移一行
Ctrl+F	光标向前移动一个字符
Ctrl+P	光标上移一行
Ctrl+D/Del 键	删除当前字符/删除当前字符串
Ctrl+K	删除当前行
Alt+D	删除当前字符串
Alt+</E	光标移到文件的开始处/结尾处
Ctrl+E	光标移到当前行尾
Alt+>/E	光标移到文件结束处
Ctrl+A	光标移到当前行首
Ctrl+X，Ctrl+F	打开文件
Ctrl+V	屏幕向下滚动一屏
Alt+V	屏幕向上滚动一屏
Ctrl+X, Ctrl+C	退出 emacs
Ctrl+X, Ctrl+S	保存当前文件
Ctrl+X, Ctrl+W	另存文件为
Ctrl+H	显示帮助信息
Ctrl+HT	启动向导程序
Ctrl+_	撤销上次操作

附录 C Linux 常用命令

1. 进程管理命令

命令名	命令功能	选项说明	
		选项	功能
who	查看当前用户信息	–h	打印标题
		–m	仅显示与标准输入有关的主机和用户名
		–q	显示所有注册用户名和用户数
ps	查看进程信息	–e	显示所有进程
		–f	显示进程所有信息
		–h	不显示标题
		–w	显示加宽输出
		–a	显示系统中与 tty 相关的所有进程
		–r	显示正在运行的进程
		–x	显示所有终端上的进程
		–l	长格式显示。若按长格式输出，则显示内容有：PID(进程号)、PRI(进程优先级)、NI(nice 值)、SIZE(虚拟映像的大小)、RSS(驻留空间的大小，显示当前常驻主存程序的 K 字节数)、WCHAN(进程等待的内核事件名)、STAT(进程状态，R 为(可运行态)、S 为(睡眠状态)、D(睡眠态)、T(停止或跟踪态)、Z(僵死状态)、W(进程没有驻留页)、TT(进程的控制 tty 名)、TRS(文本驻留大小)、SWAP(交换设备上的字节数)。
kill	向进程发送信号	–s	发送的信号类型，默认值 15(SIGTERM)，取值范围 1～50
		–p	打印 pid，不送出信号
		–l	列出所有信号名称
bg	进程挂起		
fg	进程恢复		
sleep	进程暂停执行一段时间		
at	在指定时间执行程序		

2. 文件管理命令

命令		命令功能	选项含义	
命令名	命令形式		选项	功能
ls		显示目录	–a	显示指定目录下所有子目录与文件
			–c	按文件的修改时间排序
			–u	按最近一次存取时间排序

续表

命令		命令功能	选项含义	
命令名	命令形式		选项	功　能
ls		显示目录	–d	如果参数是目录，就只显示其名称而不显示其包含的文件
			–i	在输出的第 1 列显示文件的索引节点号
			–R	显示指定目录的各个子目录中的文件
			–F	列出的文件名后加不同符号，以区分文件类型
			–l	以长格式显示文件的详细信。显示的信息依次为：文件类型与权限、链接数、文件属主、文件属组、文件大小、建立和最近修改的时间和文件名
			–L	若指定文件为符号链接文件，则显示链接指向的文件
cp	cp [选项] 源文件/目录 1 目标文件/目录 2	文件或目录复制	–a	复制时保留文件链接和属性，且复制所有子目录及其文件
			–d	复制时保留文件链接
			–f	覆盖已存在的目标文件，且不给出提示
			–I	覆盖已存在的目标文件之前给出提示
			–r	若源文件为目录文件时，将复制该目录下所有子目录和文件。此时目标文件必须为目录名
			–l	不进行复制，创建指向源文件的链接文件
mv		文件或目录移动	–I	提示方式操作
			–f	禁止提示操作
rm	rm [选项] 文件列表	文件或目录删除	–d	删除目录，不论它是否空
			–f	忽略不存在的文件，且不给出提示
			–r	删除参数中列出的全部目录及其子目录。如果没有使用–r 选项，则不删除目录
			–i	进行交互式删除，逐个确认要删除的文件
mkdir		创建目录	–m	数字，对新建目录设置存取权限，权限用八进制数字表示
			–p	可以是一个路径名。此时若路径中的某些目录尚不存在，加此选项后，系统将自动建立尚不存在的目录
rmdir		删除目录	–p	当子目录全被删除后，父目录也被删除
ln	ln[选项]源文件[目标文件]	创建链接	–s	建立符号链接，而不是硬链接
pwd		显示工作目录路径		
cd		改变工作目录		
cat		显示文件	–b	计算非空输出行，开始为 1
			–n	计算输出行，开始为 1
			–s	将相连空行用单个空行代替

续表

命令		命令功能	选项含义	
命令名	命令形式		选项	功能
cat		显示文件	–E	每行末显示$符
			–T	用^I 显示 TAB 符
find		查找文件	–name	查找的文件名，可以使用通配符。find 命令支持多个条件(and、or、not)的组合，–a 表示 and(与)，o 表示 or(或)，！表示 not(非)
			–perm	匹配模式所有模式为指定数字型模式值的文件。如果在模式前是负号(–)，表示采用除这个模式外的所有模式
			–type x	匹配所有类型为 x 的文件。x 为 c(字符文件)、为 b(块文件)、为 d(目录)、为 p(有名管道)、为 I(符号连接)、为 s(套接文件)或为 f(一般文件)
			–links n	匹配所有连接数为 n 的文件
			–size n	匹配所有大小为 n 块的文件
			–user n	匹配所有用户序列号是前面所指定的 n 用户序列号的文件，可以是数字型的值或用户登录名
grep	grep 字符串文件名列表	按模式查找文件	–F	查找模式为字符串
			–v	显示不匹配串的文本行
			–x	显示整行匹配的行
			–c	对匹配的行计数
			–r	查询目录下的所有子目录中的文件
			–l	只显示包含匹配的文件的文件名
			–n	每个匹配行加上行号显示(文件首行为 1)
			–i	不区分大小写的匹配，默认状态是区分大小写
			–e 表达式	用表达式指定模式进行精确匹配
cat	cat[选项]文件名	显示文件内容或连接文件并输出到标准输出设备	–b	从 1 开始对非空输出行编号
			–n	从 1 开始对输出行编号
			–s	多个相邻空行合并成一个空行
			–v	把非显示字符显示出来
more	more [选项]文件名	按页显示	–n	用于建立大小为整数 n 行的窗口
			–c	给文本翻页时通过往最上面清除一行，再在最后写下一行的办法写入。通常，more 清除屏幕，再写每一行
			–d	显示友好信息“Press space to continue，‘q’ quit”代替 more 的默认提示符

续表

命令		命令功能	选项含义	
命令名	命令形式		选项	功　能
more	more[选项]文件名	按页显示	–f	计算逻辑行代替屏幕行，此选项对长行的换行显示不计数
			–l	不处理“^L(换页)”字符
			–s	将多个空行压缩处理为一个空行
			–p	不滚屏，代替它的是清屏并显示文本
			–u	取消加下划线
sort	sort[选项]文件名列表	对文本文件行进行排序	–m	如果文件列表中的文件已经排序，则对这些文件进行合并
			–c	检查给定的文件是否已排序，若没有，则显示一个出错消息，不做排序
			–u	与–c 选项一起用，严格地按顺序检查；与–m 一起用，对排序后的重复行只输出一行
			–o 文件名	将排序输出放到该“文件名”指定的文件中。若该文件不存在，则创建一个新文件
			–d	按字典顺序排序，比较时仅字母、数字有意义
			–f	忽略字母的大小写
			–i	忽略非打印字符
			–M	规定月份的比较次序是“JAN”<“FEB”<…<“DEC”
			–r	按逆序排序。默认排序输出是按升序排序
			–k	指定文本行的排序关键字
			–b	比较关键字时忽略前导的空格或制表符
			–t 字符	将指定的“字符”作为字段间的分隔符
diff	diff[选项]文件/目录 1 文件/目录 2	比较文本，并显示不同	–b	忽略行尾空格
			–c	带上下文的 3 行格式
			–cn	带上下文的 n 行格式
			–r	当给定两个目录时，递归比较找到的子目录
chmod	chmod[选项][操作符][mode]文件名	改变文件或目录访问权限	u	表示用户(user)
			g	表示同组用户(group)
			o	表示其他用户(other)
			a	表示所有用户(all)，系统的默认值
			操作符	+(添加权限)、–(取消权限)、=(赋予权限，并取消其他权限)
			mode	r(可读)、w(可写)、x(可执行)、u(与文件属主有相同权限)、g(与文件属主同组的用户有相同权限)、o(与其他用户有相同权限)

续表

命令		命令功能	选项含义	
命令名	命令形式		选项	功能
chgrp		改变文件或目录所属组		
chown		改变文件或目录所有者和所属组	–R	改变指定目录及其子目录和文件的属主
			–v	显示本命令所做工作

3. 网络管理命令

命令名	功能说明
arp	网址的显示及控制
ftp	文件传输
lftp	文件传输
mail	发送/接收电子邮件
mesg	允许或拒绝其他用户向自己所用的终端发送信息
mutt	电子邮件管理程序
ncftp	文件传输
netstat	显示网络连接、路由表和网络接口信息
pine	收发电子邮件，浏览新闻组
ping	向网络上的主机发送 ICMP 回送请求包
ssh	安全模式下的远程登录
telnet	远程登录
talk	与另一用户对话
traceroute	显示到达某一主机所经由的路径及所使用的时间
wget	从网络上自动下载文件

4. 系统操作命令

命令名	功能说明
alias	设置指令的别名
chkconfig	检查、设置系统的各种服务
clock	调整 RTC 时间
dmesg	显示开机信息
eval	重新运算求出参数的内容
exit	退出目前的 Shell
export	设置或显示环境变量

续表

命　令　名	功　能　说　明
finger	查找并显示用户信息
hostid	显示主机标识
hostname	显示主机名
id	显示用户标识
kill	删除执行中的程序或工作
last	列出目前与过去登录系统的用户相关信息
logout	退出系统
lsmod	显示已加载到系统的模块
modprobe	自动处理可加载的模块
reboot	重启计算机
rhwo	查看系统用户
rlogin	远程登录
rpm	管理 Linux 各项套件的程序
shutdown	关机
top	显示，管理执行中的程序
uname	显示系统信息
useradd/userdel	添加用户/删除用户
userinfo	图形界面的修改工具
usermod	修改用户属性，包括用户的 Shell 类型，用户组等，甚至还能改登录名
w	显示目前注册的用户及用户正运行的命令
whereis	确定一个命令的二进制执行码，源码及帮助所在的位置
whois	查找并显示用户信息

5. 磁盘操作命令

命　令　名	功　能　说　明
df	显示磁盘的相关信息
du	显示目录或文件的大小
e2fsck	检查 ext2/ext3 文件系统的正确性
fdisk	对磁盘进行分区
fsck	检查文件系统并尝试修复错误
losetup	设置循环设备
mformat	对 MS-DOS 文件系统的磁盘进行格式化
mkbootdisk	建立目前系统的启动盘

续表

命 令 名	功 能 说 明
mke2fs	建立 ext2 文件系统
mkisofs	制作 iso 光盘映像文件
mount/umount	挂载文件系统/卸载文件系统
quota	显示磁盘已使用的空间与限制
sync	将主存缓冲区内的数据写入磁盘
tree	以树状图列出目录的内容

6. 系统管理命令

命 令 名	选 项 含 义
write	向用户发送信息
free	查看主存使用情况
df	查看文件空间使用情况
du	显示文件或目录占用文件空间情况
fdformat	格式化磁盘

7. 其他用户命令

命 令 名	选 项 含 义
password	修改用户口令
su	修改用户权限
echo	显示字符串
cal	显示日历
date	显示和设置系统日期和时间
clear	清空屏幕
tar	创建文件备份
gzip	压缩文件
unzip	解压缩文件

8. 帮助命令

命 令 名	选 项 含 义
man	获取相关命令的帮助信息。例如，使用命令“man dir”可以获取如何使用 dir 命令的详细信息
info	获取相关命令的详细使用方法。例如，使用命令“info info”可以获取如何使用 info 命令的详细信息

附录 D　Linux 函数

Linux 中用到的函数可参考表 D-1～表 D-10。

表 D-1　进程控制函数

函　数	功 能 说 明	函　数	功 能 说 明
fork	创建一个新进程	ptrace	追踪进程
clone	创建一个线程	prctl	对进程做特定操作
vfork	创建一个子进程	sched_get_priority_max	取得静态优先级上限
execve	运行可执行文件	sched_get_priority_min	取得静态优先级下限
getpgid	获取指定进程组标识号	sched_ getparam	取得进程调度参数
getpgrp	获取当前进程组标识号	sched_getscheduler	取得指定进程调度策略
setpgid	设置指定进程组标志号	sched_rr_get_interval	取得按 RR 算法调度的实时进程的时间片长度
setpgrp	设置当前进程组标志号	sched_ setparam	设置进程调度参数
getpid	获取进程标识号	sched - setscheduler	设置指定进程的调度策略和参数
getppid	获取父进程标识号	sched_yield	进程主动让出 CPU
getpriority	获取调度优先级	wait	等待子进程终止
setpriority	设置调度优先级	wait3	类似 wait
nice	修改进程静态优先数	waitpid	等待指定子进程终止
sleep	指定进程睡眠时间	wait4	类似 waitpid
pause	挂起进程并等待信号	capget	获取进程能力权限
personality	设置进程运行域	capset	设置进程能力权限
modify_ldt	读写进程本地描述表	getsid	获取会话标识号
exit	终止进程	setsid	设置会话标识号
_exit	立即终止当前进程	getdtablesize	进程打开最大文件数

表 D-2　进程间通信(UNIX IPC)函数

函　数	功 能 说 明	函　数	功 能 说 明
sigaction	设置对指定信号的处理方法	signal	设置信号处理
sigprocmask	根据参数对信号集中的信号执行阻塞/解除阻塞等操作	kill	向进程或进程组发信号
sigpending	为指定的被阻塞信号设置队列	sigvec	为兼容 BSD 而设的信号处理函数
sigsuspend	挂起进程等待特定信号	ssetmask	ANSI C 的信号处理函数
pipe	创建管道		

表 D-3 进程间通信(System V IPC)函数

函 数	功 能 说 明	函 数	功 能 说 明
msgctl	消息控制操作	semop	信号量操作
msgget	获取消息队列	shmctl	控制共享主存
msgsnd	发消息	shmget	获取共享主存
msgrcv	取消息	shmat	连接共享主存
semctl	信号量控制	shmdt	拆卸共享主存
semget	获取信号量	ipc	进程间通信总控调用

表 D-4 存储管理函数

函 数	功 能 说 明	函 数	功 能 说 明
brk	改变数据段空间分配	munmap	去除主存页映射
sbrk	类似 brk	mremap	重新映射虚拟主存地址
mlock	主存页面加锁	msyncm	将映射主存中的数据写回磁盘
munlock	主存页面解锁	mprotect	设置主存映像保护
mlockall	调用进程所有主存页面加锁	getpagesize	获取页面大小
munlockall	调用进程所有主存页面解锁	sync	将主存缓冲区数据写回磁盘
mmap	映射虚拟主存页	cacheflush	将指定缓冲区中内容写回磁盘

表 D-5 文件管理函数(1)

函 数	功 能 说 明	函 数	功 能 说 明
fcntl	文件控制	lseek	移动文件指针
open	打开文件	-llseek	64 位地址空间里移动文件指针
create	创建文件	dup	复制已打开的文件描述字
close	关闭文件	dup2	按指定条件复制文件描述字
read	读文件	flock	文件加/减锁
write	写文件	poll	I/O 多路转换
readv	从文件读入数据到缓冲数组	truncate	截断文件
writev	将缓冲数组的数据写入文件	ftruncate	类似 truncate
pread	对文件随机读	umask	设置文件权限掩码
pwrite	对文件随机写	fsync	文件在主存中的部分写回磁盘

表 D-6 文件管理函数(2)

函 数	功 能 说 明	函 数	功 能 说 明
access	确定文件的可存取性	getdents	读取目录项
chdir	改变当前工作目录	mkdir	创建目录
fchdir	参见 chdir	mknod	创建索引节点

续表

函　数	功 能 说 明	函　数	功 能 说 明
chmod	改变文件方式	rmdir	删除目录
fchmod	参见 chmod	rename	文件改名
chown	改变文件的属主或用户组	link	硬链接
fchown	类似 chown	symlink	符号链接
lchown	类似 chown	unlink	删除链接
chroot	改变根目录	readlink	读符号链接的值
stat	取文件状态信息	mount	安装文件系统
lstat	参见 stat	unmount	卸下文件系统
fstat	参见 stat	ustat	取文件系统信息
statfs	取文件系统信息	utime	改变文件的访问修改时间
fstatfs	参见 statfs	utimes	参见 utime
readdir	读取目录项	quotactl	控制磁盘配额

表 D-7　系统管理函数

函　数	功 能 说 明	函　数	功 能 说 明
ioctl	I/O 控制	alarm	设置进程闹钟
_sysctl	读写系统参数	getitimer	获取计时器值
acct	启用/禁止进程记账	setitimer	设置计时器值
getrlimit	获取系统资源上限	gettimeofday	获取时间和时区
setrlimit	设置系统资源上限	settimeofday	设置时间和时区
getrusage	获取系统资源使用情况	stime	设置系统日期和时间
uselib	选择使用的二进制函数库	time	取得系统时间
ioperm	设置端口 I/O 权限	times	取进程运行时间
iopl	改变进程 I/O 权限级	uname	获取当前系统的名称、版本和主机等信息
outb	低级端口操作	vhangup	挂起当前终端
reboot	重启	nfsservctl	对 NFS 守护进程进行控制
swapon	打开交换文件和设备	vm86	进入模拟 8086 模式
swapoff	关闭交换文件和设备	create_module	创建可装载模块
bdflush	控制 bdflush 守护进程	delete_module	删除可装载的模块
sysfs	取内核支持的文件系统类型	init_module	初始化模块
sysinfo	取得系统信息	query_module	查询模块信息
adjtimex	调整系统时钟		

表 D-8 网络管理函数

函 数	功 能 说 明	函 数	功 能 说 明
getdomainname	取域名	sethostid	置主机标识号
setdomainname	置域名	gethostname	取主机名
gethostid	取主机标识号	sethostname	置主机名

表 D-9 套接字管理函数

函 数	功 能 说 明	函 数	功 能 说 明
socketcall	socket 函数	recvmsg	类似 recv
socket	建立套接字	listen	监听套接字端口
bind	绑定套接字到端口	select	对多路同步 I/O 轮询
connect	连接远程主机	shutdown	关闭套接字上的连接
accept	响应套接字连接请求	getsockname	取得本地套接字名字
send	发送信息	getpeername	获取通信对方套接字名字
sendto	发送 UDP 信息	getsockopt	取端口设置
sendmsg	类似 send	setsockopt	设置端口参数
recv	通过套接字接收信息	sendfile	在文件或端口间传输数据
recvfrom	接收 UDP 信息	socketpair	创建一对已连接的无名套接字

表 D-10 用户管理函数

函 数	功 能 说 明	函 数	功 能 说 明
getuid	获取用户标识号	setreuid	分别设置真实和有效的用户标识号
setuid	设置用户标识号	getresgid	分别获取真实的、有效的和保存过的组标识号
getgid	获取组标识号	setresgid	分别设置真实的、有效的和保存过的组标识号
setgid	设置组标识号	getresuid	分别获取真实的、有效的和保存过的用户标识号
getegid	获取有效组标识号	setresuid	分别设置真实的、有效的和保存过的用户标识号
setegid	设置有效组标识号	setfsgid	设置文件系统检查时使用的组标识号
geteuid	获取有效用户标识号	setfsuid	设置文件系统检查时使用的用户标识号
seteuid	设置有效用户标识号	getgroups	获取后补组标志清单
setregid	分别设置真实和有效的组标识号	setgroups	设置后补组标志清单

附录 E　操作系统实验报告内容

操作系统实验报告宜用电子文档形式提交，实验报告包括以下可选内容。

1．文件 1——实验报告基本信息

- 完成人姓名、学号、组号(大作业分组目的：分工协作、相互研讨)，报告提交日期。
- 实验题目。
- 实验目的。
- 实验内容。
- 设计思路和流程图。
- 主要数据结构及其说明。
- 测试数据的设计及测试结果分析。
- 程序运行时的初值和运行结果。
- 实验体会：实验中遇到的问题及解决过程、实验中产生的错误及原因分析、实验中得到的主要收获及体会、对搞好今后实验提出建设性建议等。

2．文件 2——源程序(附上详细注释，便于批改老师阅读)

3．文件 3——makefile(不包含路径，便于批改老师重新编译，能生成一个可执行二进制文件)

4．文件 4——readme(简单的使用说明或手册)

参考文献

[1] 孙钟秀. 操作系统教程[M]. 4 版. 北京: 高等教育出版社，2008.

[2] 杨宗德. Linux 高级程序设计[M]. 北京: 人民邮电出版社，2008.

[3] 倪继利. Linux 内核分析及编程[M]. 北京: 电子工业出版社，2007.

[4] MATTHEW N, STONES R. Linux 程序设计[M]，陈健，宋健建，译. 北京: 人民邮电出版社，2007.

[5] STALLINGS W. 操作系统——精髓与设计原理[M]. 5 版. 陈渝，译. 北京: 电子工业出版社，2006.

[6] 陈莉君，康华. Linux 操作系统原理与应用[M]. 北京: 清华大学出版社，2006.

[7] BOVET D，CESATI C. 深入理解 LINUX 内核[M]. 3 版. 南京: 东南大学出版社，2006.

[8] LOVE R. Linux 内核设计与实现[M]. 2 版. 陈莉君，译. 北京: 机械工业出版社，2006.

[9] 罗宇. 操作系统课程设计[M]. 北京: 机械工业出版社，2005.

[10] 王红. 操作系统原理及应用（Linux）[M]. 北京: 中国水利水电出版社，2005.

[11] CORBET J. LINUX 设备驱动程序. 巍永明，译. 北京: 中国电力出版社，2005.

[12] 蒋静. 操作系统——原理·技术与编程[M]. 北京: 机械工业出版社，2004.

[13] BECK M. Linux 内核编程指南[M]. 3 版. 张瑜，译. 北京: 清华大学出版社，2004.

[14] BOVET D，CESATI C. 深入理解 Linux 内核[M]. 2 版. 陈莉君，译. 北京: 中国电力出版社，2004.

[15] 骆耀祖. Linux 操作系统分析教程[M]. 北京: 北京交通大学出版社，2004.

[16] MOLAY B. UNIX/Linux 编程实践教程[M]. 杨宗源，黄海涛，译. 北京: 清华大学出版社，2004.

[17] 陈向群. Windows CE.NET 系统分析及实验教程[M]. 北京: 机械工业出版社，2003.

[18] 陈向群. Windows 内核实验教程[M]. 北京: 机械工业出版社，2002.

[19] NUTT G. Linux 操作系统内核实习[M]. 潘登，译. 北京: 机械工业出版社，2004.

[20] 李善平. 边干边学——Linux 内核指导[M]. 浙江: 浙江大学出版社，2002.

[21] NUTT G. Linux 操作系统内核实习[M]. 北京: 机械工业出版社，2002.

[22] 毛德操，胡希明. Linux 内核源代码情景分析[M]. 杭州: 浙江大学出版社，2001.

[23] STEVENS W R. UNIX 环境高级编程[M]. 尤晋元，译. 北京: 机械工业出版社，2000.

[24] FRENCH S. Linux File Systems in 45 Minutes: A Step by Step Introduction to Writing (or Understanding) a Linux File System[CP/OL]. IBM Linux Technology Center. http://svn.samba.org/samba/ftp/cifs-cvs/samplefs.tar.gz.

[25] KIRAN R. Writing a Simple File System[J/OL]. http://www.geocities.com/ravikiran_uvs/articles/rkfs.html.
[26] http://www.kernel.org/.
[27] http://www.gnu.org/.
[28] http://www.ubuntu.org.cn/
[29] http://www.kernel.org/.
[30] http://www.gnu.org/.
[31] http://www.ubuntu.org.cn/.