

本书由北京大学、北京航空航天大学、南京大学共同组织编写

Linux 操作系统 分析与实践

Linux 系列教材编写组 编著



清华大学出版社

序

诞生于 1991 年,今年“寂寞十七岁”的 Linux 一点也不寂寞。从微型嵌入式系统到超级计算机,处处都可看到 Linux 多彩的光环。十七年来一步一个脚印, Linux 正在成为操作系统的另一主流。

Linux 也是开放式创新的好见证。开源社区集结全球精英,为共同的需求一起创新。通过热烈的讨论与相互的脑力激荡,开源项目的开发过程也就是一个宝贵的学习过程。

《Linux 操作系统分析与实践》的出版为莘莘学子及业界人士提供了一本完整的 Linux 教材,理论概念介绍与实验指导紧密结合,结构清晰、内容顺畅易懂。除 Linux 之外,这本教材很有系统地讲述了操作系统的主要模块,因此它也将是一般操作系统课程的好教材;并且更能为未来虚拟化、实时性及安全性系统的教材打下扎实的基础。

英特尔公司多年来不断与高等学校合作,并积极参与 Linux 开源项目的开发,此次能够为长年合作的教授们开发本教材提供支持,英特尔公司深感欣喜,也祝贺他们为中国高等学校计算机专业教学水平的提高做出新的贡献。

英特尔公司副总裁

英特尔亚太研发有限公司总经理

王文汉

2008 年 6 月

前 言

1991年10月,芬兰大学生 Linus Torvalds 在学校的 ftp 上发布了自己所编写的类 UNIX 操作系统——Linux 0.02 版的源代码,并宣布它遵守 GPL 协议,而且符合 UNIX 的操作系统 POSIX 标准,源代码可以在 UNIX 主机上用 gcc 编译生成可执行的二进制代码,也可以在个人计算机上运行。

Linus 在互联网上公布 Linux 0.02 版时,正是互联网和个人电脑兴起和开始高速发展的时代,它吸引了世界各地的程序黑客对 Linux 进行修改和完善,但所有对内核的改动都要通过 Linus。靠 Linus 的把关,Linux 的内核保持着统一,不至于像发行版本那样,种类很多。后来,随着系统在网络上的不断分发、传播,知道的人越来越多,吸引了成千上万的软件高手。到 1994 年发布正式的 1.0 版本时,已经有了相当大的名气,GNU 组织也暂且搁下自己的 Hurd 计划,全力支持 Linux 的发展。今天说的 Linux,事实上只是一个简称,它的正式名称是 GNU/Linux。

十多年过去了,通过全世界众多软件高手,特别是 Linus 本人的不懈努力,使 Linux 能支持多种硬件平台(X86、Alpha、PowerPC、SPARC、ARM),具有源代码公开、功能强大、运行稳定高效、遵守 POSIX 标准、与 UNIX 兼容、免费等优点,终于发展成为主流操作系统的一员。Linux 服务器可与商用 UNIX 相媲美,客户端则向 Windows 发起了挑战,嵌入式 Linux 也成为近几年发展的方向。随着 Linux 的优良性能不断显现,IBM、Intel、Sun 等不少商业科技巨头纷纷涉足 Linux。

“软件是信息产业的灵魂”,操作系统作为软件子系统的核心,是软件的基础运行平台。操作系统本身作为一个大型系统软件,也是自主发展软件产业的关键所在。近十年来,Linux 在促进信息产业发展的同时,也为设计各种新型操作系统提供了难得的案例。

今天,操作系统正向着大型和微型两个不同的方向发展着。大型系统的典型是分布式操作系统和集群操作系统,而微型系统的典型则是嵌入式操作系统。应该说 Linux 为操作系统研究者提供了新理论、新方法、新技术的实践和验证平台,同时反过来促进了操作系统的不断发展。

高等学校是培养计算机科学家和软件工程师的主阵地,操作系统原理是高等学校计算机专业本科生的主干课程,操作系统分析与实践则是计算机专业研究生的核心课程。

Linux 由于源代码公开,特别适合于操作系统课程的教学。而事实上,Linux 本身的产生也正是来源于操作系统课程教学的结果。Linus 在上大学时,操作系统课程中讲述的是 Minix,一种用于教学的小型类 UNIX 操作系统。他觉得 Minix 不太好,决定自己

写一个保护模式下的操作系统。1991年8月, Linus 完成了 Linux 的 0.01 版, 当然它很不完善, 甚至没有软盘驱动程序。

本书第 1 章介绍了操作系统的地位、定义、功能、特性, 以及发展历史、分类、内核结构模型; 第 2 章则以 Intel X86 为例介绍了计算机硬件子系统对操作系统的支持与要求, 重点阐述了 X86 的保护模式、中断机制、Firmware 及 Linux 系统引导; 第 3 章详细介绍了进程管理以及在 Linux 内核里的实现方法; 第 4 章介绍了内存管理的基本原理、技术, 详细介绍了 Linux 内存管理的策略和实现机制; 第 5 章介绍了文件管理的基本概念, 以 Ext2, Proc 为例详细介绍了 Linux 文件系统。第 6 章详细介绍了设备管理的基本原理以及 Linux 设备管理的特点; 第 7 章详细介绍了 Linux 设备驱动程序设计的基础知识; 第 8 章介绍了 Linux 字符设备和块设备驱动程序设计的方法及实例; 第 9 章介绍了 Linux 网络设备驱动程序设计的方法及实例。除此以外, 本书还给出了 Linux 系统管理、内核同步机制、进程间通信、内存管理、Proc 文件创建、设备驱动程序、内核裁减等 7 个实验内容及有关参考代码。综观全书, 处处体现出了对操作系统原理的深入理解, 以及对 Linux 操作系统分析和设计能力的培养。

本书适于作为计算机相关专业方向的高年级学生或研究生的教材, 也适合有志于从事 Linux 操作系统研究和开发的工程师阅读。本书的编写得到了 Intel 公司中国大学合作部门的支持。第 1、2、8、9 章主要由北京大学的陈向群、张齐勋、方跃坚编写, 第 3、4、6 章主要由北京航空航天大学原仓周编写, 第 5、7 章主要由南京大学李敏编写。中国工程院倪光南院士, 北京大学陈钟教授、吴中海教授, 清华大学周悦芝副教授, Intel 公司傅文庆经理, 北京凝思科技公司宫敏总经理与本书作者共同审阅了编写大纲。

由于作者水平有限, 书中难免存在一些缺点和错误, 殷切希望广大读者批评指正。

编著者

2008 年 6 月

目 录

第 1 章 操作系统概述	1
1.1 操作系统在计算机系统中的地位	1
1.2 操作系统的定义、功能和特性	2
1.2.1 操作系统的定义	2
1.2.2 操作系统的特征	3
1.2.3 操作系统的功能	5
1.3 操作系统的发展历史及分类	8
1.3.1 操作系统的发展过程	8
1.3.2 操作系统分类	12
1.4 操作系统的内核体系结构模型	19
1.5 Linux 的特点及发展简史	24
1.5.1 什么是 Linux	24
1.5.2 Linux 的发展历程	24
1.5.3 Linux 的特性	25
1.5.4 Linux 内核的版本演化介绍	27
1.6 Linux 内核的结构及主要组成部分	30
1.6.1 Linux 内核系统体系结构	30
1.6.2 Linux 内核的源码组织结构	31
1.7 本章小结	32
思考题	33
进一步阅读	33
第 2 章 计算机系统硬件基础	34
2.1 计算机硬件的组成结构	34
2.2 处理器的工作模式	36
2.3 80x86 的保护模式	38
2.3.1 80x86 系统寄存器	38
2.3.2 存储管理机制	41
2.3.3 分段机制	41

2.3.4	分页机制	43
2.3.5	保护机制	44
2.4	中断陷入与异常处理	45
2.4.1	中断与异常	45
2.4.2	中断系统	46
2.4.3	中断处理过程	48
2.4.4	时钟中断	48
2.5	任务切换	49
2.6	SMP 及多核技术	51
2.7	Linux 系统引导过程概述	53
2.7.1	BIOS 引导	54
2.7.2	LILO/GRUB 引导	54
2.7.3	Kernel Boot	56
2.7.4	Init 引导	57
2.8	本章小结	57
	思考题	57
	进一步阅读	58
第 3 章	进程管理	59
3.1	进程的原理和实现	59
3.1.1	进程控制块	60
3.1.2	进程状态	62
3.1.3	进程的标识	64
3.1.4	进程描述符在内存里的组织形式	64
3.1.5	进程的睡眠和唤醒	65
3.2	线程的原理与实现	66
3.3	Linux 进程创建和结束	68
3.3.1	进程的创建	68
3.3.2	进程的结束	71
3.4	进程调度	72
3.4.1	进程的切换	73
3.4.2	进程的调度策略	74
3.4.3	进程调度的触发和进程切换的时机	75
3.4.4	运行队列	76

3.4.5	调度域	77
3.4.6	抢占	80
3.4.7	调度的时间复杂度	81
3.4.8	函数 schedule 的实现	81
3.4.9	函数 load_balance 的实现	82
3.5	并发控制原理	83
3.5.1	同步原语简介	84
3.5.2	死锁	87
3.5.3	同步原语	87
3.6	Linux 下的同步机制	88
3.6.1	处理器数据	88
3.6.2	可抢占内核	89
3.6.3	开关中断	89
3.6.4	原子操作	90
3.6.5	栅栏	90
3.6.6	自旋锁和读写锁	91
3.6.7	Fast User Space Mutex	92
3.6.8	信号量	92
3.7	Linux 下进程间通信	95
3.7.1	管道	95
3.7.2	命名管道	97
3.7.3	消息队列	98
3.7.4	共享内存	99
3.7.5	套接口	101
3.8	本章小结	101
	思考题	101
	进一步阅读	102
第 4 章	内存管理	103
4.1	内存管理概述	103
4.1.1	内存管理的目的和功能	103
4.1.2	内存分配的方式	104
4.1.3	重定位	105

4.2	内存管理的基本技术	108
4.2.1	分区法	108
4.2.2	可重定位分区法	110
4.2.3	覆盖技术	112
4.2.4	交换技术	113
4.3	分页技术	113
4.3.1	分页内存管理的基本原理	114
4.3.2	分页技术中的地址映射	115
4.3.3	页面保护方式	118
4.4	分段技术	119
4.4.1	分段技术的基本原理	119
4.4.2	分段的地址转换	120
4.4.3	段的共享和保护	121
4.5	段页式技术	122
4.5.1	段页式技术的基本原理	122
4.5.2	地址转换过程	124
4.6	虚拟存储器技术	124
4.6.1	程序的局部性原理	124
4.6.2	虚拟存储器的概念	125
4.6.3	虚拟存储管理需要考虑的问题	125
4.6.4	虚拟分页技术的基本原理	126
4.6.5	请求页式管理中的置换算法	128
4.6.6	抖动问题	131
4.6.7	性能分析	133
4.7	Linux 的内存管理	134
4.7.1	进程的虚存空间及其划分	134
4.7.2	进程空间的描述	134
4.7.3	物理内存的管理	136
4.7.4	基于 Slab 的缓存管理	139
4.7.5	页面换出或丢弃	142
4.7.6	交换 Cache	143
4.7.7	页面的换入	143
4.8	本章小结	144
	思考题	145

进一步阅读.....	146
第 5 章 文件系统	147
5.1 文件	147
5.1.1 逻辑文件与物理文件.....	147
5.1.2 逻辑文件的组织.....	148
5.1.3 逻辑文件的结构.....	149
5.1.4 逻辑文件的存取.....	150
5.1.5 物理文件的结构.....	150
5.2 目录	154
5.2.1 文件控制块.....	154
5.2.2 目录、文件目录与目录文件	155
5.2.3 文件检索.....	156
5.3 文件存储空间管理	157
5.4 文件共享	159
5.4.1 静态共享.....	159
5.4.2 动态共享.....	161
5.5 Linux 文件系统	162
5.5.1 虚拟文件系统.....	162
5.5.2 文件系统注册与注销,安装与卸载	174
5.5.3 文件系统的缓存机制.....	175
5.5.4 Ext2 文件系统	177
5.5.5 proc 文件系统	179
5.6 本章小结	183
思考题.....	183
进一步阅读.....	184
第 6 章 设备管理	185
6.1 设备管理概述	185
6.1.1 设备的分类.....	185
6.1.2 设备控制器.....	187
6.1.3 设备管理的目标和功能.....	188
6.1.4 I/O 设备的控制方式	190
6.2 I/O 软件	194

6.2.1	I/O 软件的分层结构	194
6.2.2	中断处理程序	195
6.2.3	设备驱动程序	196
6.2.4	设备无关	197
6.2.5	用户空间的 I/O 软件	199
6.2.6	I/O 软件的层次调用关系	200
6.3	缓冲技术	200
6.3.1	缓冲的引入	201
6.3.2	缓冲的种类	201
6.3.3	缓冲池的管理	203
6.4	设备分配	205
6.4.1	与设备分配相关的因素	205
6.4.2	设备分配的原则和方式	206
6.4.3	设备分配技术	206
6.4.4	设备分配算法	207
6.5	I/O 控制	208
6.5.1	I/O 控制的引入	208
6.5.2	I/O 控制的功能	208
6.5.3	I/O 控制的实现	209
6.6	磁盘调度	209
6.6.1	磁盘存取时间	209
6.6.2	磁盘调度算法	210
6.6.3	性能比较	212
6.7	本章小结	213
	思考题	214
	进一步阅读	214
第 7 章	Linux 驱动程序编写基础	215
7.1	内核模块概述	215
7.2	内核模块编程基础	217
7.2.1	内核模块的基本结构	217
7.2.2	编译和加载	218
7.2.3	内核符号表	220
7.2.4	初始化与清理函数	220

7.2.5 模块参数	221
7.3 中断服务例程	223
7.3.1 概述	223
7.3.2 检测中断号	224
7.3.3 安装 ISR	226
7.4 上半部和下半部	229
7.4.1 下半部的实现机制	230
7.4.2 软中断	231
7.4.3 tasklet 机制	232
7.4.4 工作队列	234
7.4.5 选择合适的下半部机制	236
7.5 本章小结	237
思考题	237
进一步阅读	238
第 8 章 字符设备和块设备驱动程序	239
8.1 字符设备和块设备驱动程序设计基础	239
8.1.1 驱动层次结构	240
8.1.2 设备号	240
8.1.3 字符设备驱动程序注册	240
8.1.4 块设备驱动程序注册	242
8.2 字符设备和块设备驱动程序结构	243
8.2.1 字符设备驱动程序结构	243
8.2.2 块设备驱动程序结构	244
8.3 字符设备和块设备驱动程序的内核相关数据结构	245
8.4 开发实例	249
8.4.1 字符设备驱动程序	249
8.4.2 块设备驱动程序	252
8.5 本章小结	255
思考题	255
进一步阅读	256
第 9 章 网络设备驱动程序	257
9.1 网络设备驱动程序设计基础	257

9.2 Linux 网络设备驱动程序的结构	258
9.3 网络设备驱动程序的内核相关数据结构	261
9.3.1 net_device 结构	261
9.3.2 sk_buffer 结构	266
9.4 开发实例	267
9.5 本章小结	273
思考题	273
进一步阅读	274
 实验 1 Linux 系统管理	 275
实验 2 剪裁内核	291
实验 3 内核同步机制	313
实验 4 进程间通信(内存映射、共享内存、信号量、管道和信号等)	317
实验 5 proc 文件系统	325
实验 6 虚拟内存的管理	331
实验 7 字符、块设备和网卡驱动	338
参考文献	343

第1章 操作系统概述

1.1 操作系统在计算机系统中的地位

计算机系统包括硬件(子)系统和软件(子)系统。硬件系统是计算机系统赖以工作的实体。软件系统保证计算机系统按用户指定的要求协调地工作。这两个部分构成了计算机系统的资源。因此,计算机系统的资源包括两大类:硬件资源和软件资源。

中央处理单元(CPU)、内存、外存(如磁盘、磁带等)及各种类型的输入输出设备(如键盘、鼠标、显示器和打印机等)组成了计算机硬件系统;而各种程序和数据则组成了计算机的软件系统,如图 1.1 所示。

计算机系统	软件系统 (程序、数据)	应用软件:文字处理、图形图像处理、科学计算和 MIS 等
		支撑软件:数据库、网络 and 多媒体等
		系统软件:操作系统、编译程序等
	硬件系统	中央处理单元(CPU)、主存储器 外存储器(磁盘、磁带等) 输入输出设备(键盘、鼠标、显示器和打印机等)

图 1.1 计算机系统

一个程序在执行前,必须先要得到计算机系统的内存空间后才能被装入内存。这些程序的执行要依靠中央处理器。程序在执行的过程中需要调用公共服务子程序和共享文件,还要使用各种计算机外部设备,以完成信息的输入和输出。这里的中央处理器、内存空间、公共服务子程序和共享文件以及计算机外部设备等都是计算机系统资源。可见,任何程序的执行都要占用计算机系统的资源。

各个程序对计算机资源的要求和使用是不同的,还可能会发生冲突。比如,正当一个程序在使用打印机时,另外一个程序也要求使用这台打印机。可以想象,如果没有一种方法或规则对程序使用打印机的要求进行管理的话,肯定会引起混乱。对一台打印机的使用需要进行管理,同样,对计算机系统资源的其他资源也需要进行管理。适当地控制和协调计算机资源的分配是必要的。

用户在使用计算机系统时,当然希望计算机能够按照用户的要求工作,控制程序按照用户各自的意愿执行。所以,计算机系统必须有识别用户要求的能力,还必须具有对不同

用户的程序进行不同控制执行的能力。

操作系统属于计算机系统软件(子)系统中的系统软件,是紧密接近硬件的第一层软件,是对硬件功能的首次扩充,其他软件则是建立在操作系统之上的。操作系统对硬件功能进行扩充,各种软件在操作系统的统一管理和支持下运行。

因此,操作系统在计算机系统中占据着一个非常重要的地位,它不仅是硬件与所有其他软件之间的接口,而且任何计算机都必须在其硬件平台上加载相应的操作系统之后,才能构成一个可以协调运转的计算机系统。只有在操作系统的指挥控制下,各种计算机资源才能被分配给用户使用。也只有在操作系统的支撑下,其他系统软件如各类编译系统、程序库、运行支持环境才得以取得运行条件。没有操作系统,任何应用软件都无法运行。

可见,操作系统实际上是一个计算机系统中硬、软件资源的总指挥部。操作系统的性能高低,决定了整个计算机的潜在硬件性能能否发挥出来。操作系统本身的安全可靠程度,决定了整个计算机系统的安全性和可靠性。因此,操作系统是计算机系统中软件技术含量最大、附加值最高的部分,是软件(子)系统的核心,是软件的基础运行平台。

1.2 操作系统的定义、功能和特性

1.2.1 操作系统的定义

操作系统是计算机系统中的一个系统软件,它是这样一些程序模块的集合——它们能有效地组织和管理计算机系统硬件及软件资源,合理地组织计算机工作流程,控制程序的执行,并向用户提供各种服务功能,使得用户能够灵活、方便、有效地使用计算机,并使整个计算机系统能高效地运行。

从操作系统的表现形式上看,它是一个大型软件程序,其中包括大量的为实现操作系统各项功能的软件程序。从设计和结构角度看,这些软件程序通常是模块化的。有的操作系统带有内核,其内核也由各种程序模块组成。有的操作系统本身是层次化的,每一个层次的程序通常也是模块化的。还有的操作系统是客户-服务器结构的,而服务器或客户机内部程序一般也是模块化的。总结起来,操作系统这样一个大型软件程序,是具有各种功能的、大量程序模块的集合。

操作系统的任务之一是组织和管理计算机系统硬件及软件资源。在操作系统内部,为了掌握整个计算机系统的硬件和软件资源,设计了各种不同类型的表格或数据结构,将所有的硬件和软件资源一一加以登记,资源的名称、类型、数量、用途、完好状态及目前使用状态等所有信息,均在有关数据结构中保存,并动态地、实时地不断更新这些数据。记录数据的目的是:让操作系统根据用户程序对各种资源的需求情况,资源的当前分配和使用情况以及有关的资源调度策略,对资源进行有效的组织和管理。

这里,“有效”是指操作用户有不同的要求,系统在管理计算机资源时要考虑到系统运行的效率和资源的利用率。要尽可能提高中央处理器的利用率,让它尽可能少的空转,应该在保证访问效能的前提下尽可能有效利用其他资源,例如减少对内存、硬盘空间的浪费等。

在计算机系统中,不同的用户有不同的要求。系统中的作业(用户提交的程序、数据及对程序执行过程的要求)有大有小。有的作业需要运算时间长,有的作业有大量的信息需要打印输出,有的事件需要立即处理。例如,笔记本式计算机的电池还有 30s 即将没电了,需要立即通知用户,要么给计算机提供新的电源,要么赶紧保存重要的文档。而有的事件推迟一些时间处理,不会引起大的问题,例如,通知用户信箱中刚刚收到一封电子邮件,早 10s 还是晚 10s 发出通知,不会对用户造成多大的影响。操作系统应该根据用户不同的要求,作业的各自特点及紧急程度等,合理地组织计算机工作流程,控制程序的执行。

这里“合理”是指操作系统要“公平”对待不同的用户程序,保证系统不发生“死锁”和“饥饿”的现象。

操作系统另一项重要任务,是向用户提供各种服务功能。一方面,向程序开发和设计人员提供高效的程序设计接口;另一方面,向使用计算机系统的用户提供接口,使得用户能够灵活、方便、有效地使用计算机。

这里“方便”是指操作系统的人机界面要考虑到用户使用界面和程序设计接口两个方面的易用性、易学性和易维护性。

1.2.2 操作系统的特征

作为一种系统软件,操作系统有着与其他软件相比所不同的特征。

1. 并发性

并发性是指在计算机系统中同时存在若干个运行着的程序,从宏观上看,这些程序在同时向前推进。

进一步分析,计算机程序的并发性体现在如下具体两个方面:用户程序与用户程序之间并发执行;用户程序与操作系统程序之间并发执行。

实际上,从微观上看,在单处理器的环境下,这些同时运行着的程序是交替在 CPU 上运行的。在多处理器系统的环境中,多个程序的并发特征,不仅在宏观上是并发的,而且在微观上,即在处理器一级上,程序也是并发执行的。而在分布式系统中,多个计算机的并存,使得程序的并发特征得到了更充分的体现,因为在每个计算机上都可以有程序执行,它们共同构造了程序并发执行的图景。

这里要强调指出的是,不论是哪一种计算环境,并发都是在操作系统的统一指挥下的并发。在两个独立的操作系统控制下的机器,它们的程序也在并行运行,但这种情况不属

于本书所讨论的并发性范畴。

2. 共享性

共享性是指操作系统程序与多个用户程序共用系统中的各种资源。这种共享性是在操作系统控制下实现的。资源的共享性主要针对计算机系统下的如下几项重要资源。

- 中央处理单元。是所有程序都必须使用的最重要的资源,操作系统必须采用恰当的调度策略,对多个并发程序分配处理器资源。
- 内存储器。任何一个程序必须首先调入内存之后,才能执行。管理内存的公共使用是操作系统的重要功能之一。
- 外存储器。主要用来保存各种程序和数据。这些程序和数据一般以文件的形式存放在外存储器上。外存储器有各种类型,如硬盘、软盘、磁带和可读写光盘等,这些外存储器都是为所有的程序和数据共享的,操作系统同样必须对外存储器的共享进行管理,并且保证在外存储器中所有程序和数据的完整性和正确性。
- 外部设备。计算机系统的外部设备是供所有的用户使用的,这些用户包括操作系统、系统用户(如管理员)和普通用户。操作系统必须对这些用户对外部设备的使用要求做出管理,使所有的外部设备能够按照一定的策略被全部用户所共享。

在计算机系统中,对资源的共享一般有两种形式:互斥共享和同时共享。

- 互斥共享。系统中的有些资源,如打印机、磁带机和扫描仪等,虽然可以供多个用户程序同时使用,但是在一段特定的时间内只能由某一个用户程序使用。当这个资源正在被使用的时候,其他请求该资源的程序必须等待,并且在这个资源被使用完了以后,才由操作系统根据一定的策略再选择一个用户程序占有该资源。通常把这样的资源称为临界资源。操作系统维护的许多重要系统数据都是临界资源,它们都要求被互斥共享。
- 同时共享。系统中还有一类资源,它们在同一段时间内可以被多个程序同时访问。需要说明的一点是这种同时访问是指宏观上的同时,微观上这些程序访问这个资源有可能还是交替进行的,而且它们交替访问这个资源的顺序对访问结果没有什么影响。一个典型的可以同时共享的资源就是硬盘,当然,那些可以重入的操作系统代码也是可以同时共享的。

3. 随机性

操作系统的运行是在一种随机的环境下进行的。这种随机环境的含义是,操作系统不能对所运行的程序的行为及硬件设备的情况做出任何事先的假定。显然,一个设备可能任何时候向处理器发出中断请求,而运行着的用户程序要做什么事情也无法事先知道。因而一般来说,操作系统正处于什么样的状态之中是无法确切知道的,这就是随机性

的含义。但是,随机性并不意味着操作系统不能很好地控制资源的使用和程序的运行,随机性突出强调了在进行操作系统的设计与实现时要充分考虑各种各样的可能性。操作系统本身应该稳定、可靠、安全、高效地实现程序并发和资源共享的目的。

1.2.3 操作系统的功能

按照资源管理的观点,操作系统的这些功能主要可以分为进程管理(处理器管理)、存储管理、文件管理和设备管理等。下面分别进行简要的介绍。

1. 进程管理

进程管理的实质是对 CPU 进行管理,所以进程管理往往又被称为处理器管理。CPU 是计算机系统中最宝贵的硬件资源,为了提高 CPU 的利用率,现代操作系统都采用了多道程序技术。如果一个程序因等待某一条件而不能运行下去时,就把处理器占用权转交给另一个可运行程序。或者,当出现了一个比当前运行的程序更重要的可运行程序时,后者应能抢占 CPU。为了描述多道程序的并发执行,就要引入进程的概念。通过对进程的管理,协调多道程序之间的关系,解决有关处理器分配调度策略、分配的实施和回收等问题,以使 CPU 资源得到最充分的利用。

进程管理主要包括进程控制、进程同步、进程间通信和进程调度等几方面的内容,其中进程控制主要处理进程的创建、状态转换、进程撤销以及相关的进程资源的分配与回收等事项;进程同步主要处理进程之间的关系,包括进程的同步和互斥;进程间通信主要处理相互协作进程之间信息的交换问题;而进程调度则是按照一定的算法从就绪队列中挑选一个进程在处理器中真正执行它。下面分别叙述进程管理的主要内容。

1) 进程控制

在多道程序环境下,进程是操作系统进行资源分配的单位。在进程创建时,系统要为进程分配各种资源,例如内存、外设等;在进程退出的时候,系统要从进程中回收被分配给它的资源。进程控制的主要任务就是创建进程、撤销进程以及控制进程运行时候的各种状态转换。

2) 进程同步

多个进程的执行是并发的,它们以异步的方式运行,执行进度也是不可预知的。为了使多个进程可以有条不紊地运行,操作系统要提供进程同步机制,以协调进程的执行。一般有两种协调方式:互斥和同步。互斥指多个进程对临界资源访问时采用互斥的形式;同步则是在相互协作共同完成任务的进程之间,用同步机制协调它们之间的执行顺序。

最简单的实现互斥的方法就是给资源加锁,并提供操纵锁变量的原语。包括加锁和解锁原语。所谓原语就是指具有某种功能、运行时有“原子性”的小段程序。原子性保证

这一段程序要么全部执行,要么全部不起作用,即这一操作不能被进一步分割或者打断,如同人们一度认为原子是物质的最小单位,不可再分割一样。有关进程同步与互斥等内容在后面的章节中将详细讲述。

3) 进程间通信

进程间通信主要发生在相互协作的进程之间。既然需要协作,那么协作的进程之间,就存在着信息或数据的交换。由操作系统提供的进程间通信机制是协作的进程之间相互交换数据和消息的手段。

4) 调度

又称为处理器调度。由于现代操作系统中引入了线程的概念(线程是处理器时间分配的基本单位,它代表一个指令的执行流及执行的上下文环境,一个进程可以包含多个线程),因此,通常有线程调度或进程调度。进程(线程)调度的任务就是从进程(线程)的就绪队列中按照一定的算法挑选出一个来把处理器资源分配给它,并准备好特定的执行上下文让它执行起来。

2. 存储管理

存储管理的任务是管理计算机内存的资源。在当代计算机系统中,计算机内存空间越来越大,256MB的内存空间已是很普通的容量。虽然内存空间可以扩充,但是内存空间再大也跟不上运行程序对内存空间的巨大需求。而且内存空间还受到CPU寻址能力的限制,内存的容量不能无限扩充。

概括地说,存储管理有三个方面的任务。首先,当多个程序共享有限的内存资源时,要考虑如何将有限的内存空间分配给多个程序;其次,存放在内存中的多个程序和数据应该彼此隔离、互不侵扰;第三,解决内存扩充的问题,即将内存和外存结合起来管理,为用户提供一个容量比实际内存大得多的虚拟存储器。下面分别就这三项存储管理功能做进一步的叙述。

1) 内存的分配与回收

操作系统要为每个进程分配内存空间,在分配的过程中还要尽可能提高内存资源的使用效能。对于已经退出运行的进程所占据的内存空间,操作系统还要把它们加以回收,重新使用。人们在使用计算机时经常会发现,原本很大的内存在使用一段时间以后,似乎丢失了一部分空间,仔细检查会看到在空间使用图中,有一大堆空间碎片。这种现象,就是内存的分配与回收算法中的缺陷造成的。可见内存的分配与回收对计算机系统有效运行的影响是很大的。

2) 存储保护

由于内存是供多个程序共享的,从理论上讲,每个程序都应该在分配给自己的内存区中运行。但是,从实际上看,必须考虑某个程序可能发生越界的情况。这是因为,首先程

序自身可能存在一些错误或隐患,造成程序的突然越界运行,不仅自身发生问题,而且影响或破坏了其他用户程序的运行。如果发生故障的程序越界到操作系统运行的内存空间,那么受到危害将更加严重,整个计算机系统有可能崩溃。其次,如果某个程序具有恶意,比如是病毒程序,或者是外界的入侵程序,那么就更有必要防止他们危害其他用户和整个计算机系统了。

在现代的计算机系统中,通常提供有硬件实现的存储保护机制,操作系统则利用这一机制实现进程的地址保护。

3) 内存扩充

物理内存的大小总是有限的,各种程序对运行空间的需求经常超过实际内存的大小。内存扩充功能就是借助于虚拟存储技术在逻辑上增加进程运行空间的大小,这个大小比实际的物理内存要大得多。例如,在 x86 系统中目前常见的台式个人计算机系统中物理内存的大小不超过 1GB,而一个进程的地址空间可有 4GB 大小,各个进程空间总和则可以达到 64TB,远远大于 1GB 的物理内存。

在虚拟存储技术中,操作系统通常将空间划分为 4KB 大小的页面,并且以页面为单位进行存储空间的分配和回收。进程所需的总页面数所对应的空间可以超出实际物理空间。但是操作系统只把正在使用的页面保持在内存中,或把即将使用的页面调入到内存中,而将不用的页面调换到外存上,用户感受不到实际内存对使用空间的限制,仿佛有一虚拟的大容量内存存在。当然,系统要实现这一机制,必须提供请求页面调入的功能和页面置换的功能。

3. 文件管理

在计算机系统中的信息资源(如程序和数据)通常是以文件的形式存放在外存储器(如磁盘、磁带)上的,需要时再把它们装入内存。操作系统一般都提供很强的文件系统。文件管理的任务是有效地支持文件的存储、检索和修改等操作,解决文件的共享、保密和保护问题,以使用户方便、安全地访问文件。

1) 文件存储空间的管理

任何文件存放在外存储器上,都需要以某种形式占据外存储器的空间。文件系统的很重要的功能就是为每个文件分配一定的外存空间,并且尽可能提高外存空间的利用率和文件访问的效能。文件系统设置专门的数据结构记录文件存储空间的使用情况。为了提高空间利用率,存储空间的分配通常采用离散分配方式,以 512B 或几千字节大小的块为基本单位进行分配。

2) 目录管理

目录管理的主要任务就是给出组织文件的方法,它为每个文件建立目录项,并对众多的目录项加以有效的组织,以便为用户提供方便的按名存取。

3) 文件操作和目录操作

4) 文件系统的安全性

安全性包括文件的读写权限管理及存取控制,用以防止未经核准的用户存取文件,防止越权访问文件,防止使用不正确的方式访问文件。

4. 设备管理

在现代计算机系统中,通常存在着大量的外部设备,从键盘、鼠标、显示器到打印机、数字音响设备和 DVD 等。这些外部设备的种类繁多,功能差异很大。如果要求每个用户程序亲自对连接到计算机系统各种不同的外部设备进行控制操作,实际是行不通的。

操作系统应该向用户提供设备管理。这里的设备管理是指计算机系统中除了 CPU 和内存以外的所有输入、输出设备的管理。由操作系统的设备管理功能负责外部设备的分配、启动和故障处理,用户不必详细了解设备及接口的技术细节,就可以方便地通过操作系统提供的设备管理手段,对设备进行操作。

在操作系统中,为了提高设备的使用效率和整个系统的运行速度,需要采用一系列的技术,包括中断技术、通道技术、虚拟设备技术和缓冲技术等,尽可能地发挥设备和主机的并行工作能力。此外,设备管理应为用户提供一个良好的人机界面,使用户不必涉及具体设备的物理特性即可方便灵活地使用这些设备。

除此之外,操作系统还要具备中断处理、错误处理等功能。操作系统的各功能之间并非是完全独立的,它们之间存在着相互依赖的关系。

1.3 操作系统的发展历史及分类

1.3.1 操作系统的发展过程

操作系统如同任何其他事物一样,也有它的诞生、成长和发展的过程。为了更清楚地把握操作系统的实质,了解操作系统的发展是很有必要的。操作系统的许多基本概念,都是在操作系统的发展过程中出现并逐步得到发展和成熟的。了解操作系统发展的历史,有助于更深刻地认识操作系统基本概念的内在含义。下面介绍操作系统的发展过程。

1. 手工操作

由于第二次世界大战对武器装备设计的需要,美国、英国和德国等国家,陆续开始了使用真空管建造数字电子计算机。在这个阶段,通过在一些插板上的硬连线来控制计算机的基本功能,程序设计全部采用机器语言,没有程序设计语言(甚至没有汇编语言),更

谈不上操作系统。那时实际上所有的题目都是数值计算问题。到了 20 世纪 50 年代早期,出现了穿孔卡片,可以将程序写在卡片上。

在一个程序员上机期间,整台计算机连同附属设备全被其占用。程序员兼职操作员,效率低下。其特点是手工操作、独占方式。后来人们开发了汇编语言及相应的汇编程序,以及其他一些控制外设的程序等。

2. 监控程序(早期批处理)

20 世纪 50 年代晶体管的发明极大地改变了整个状况。计算机比较可靠,厂商可以成批地生产并销售计算机给用户,用户可以指望计算机长时间运行,完成一些有用的工作。这一时期出现了 FORTRAN、ALGOL 及 COBOL 等高级语言。此时要运行一个作业(Job,即一个或一组程序),程序员首先将程序写在纸上(用高级语言或汇编语言),然后穿孔成卡片。再将卡片盒带到输入室,交给操作员(如图 1.2 所示)。计算机运行完当前任务后,其计算结果从打印机上输出,操作员到打印机上取下运算结果并送到输出室,程序员稍后就可取到结果。然后,操作员从已经送到输入室的卡片盒中读入另一个任务。如果需要 FORTRAN 编译器,操作员还要从文件柜把它取来读入计算机。

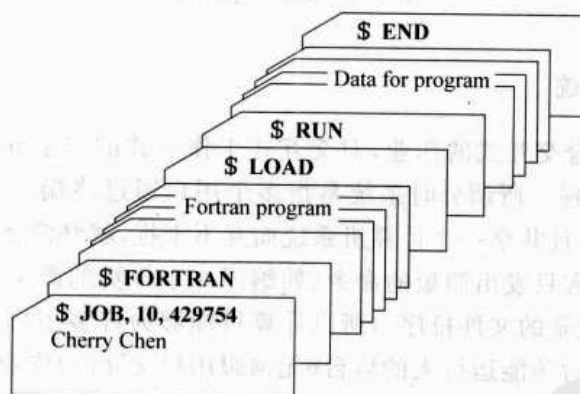


图 1.2 作业卡片示意图

由于处理器速度提高,导致手工操作设备输入输出信息与计算机计算速度不匹配。因此,人们设计了监督程序(或管理程序),来实现作业的自动转换处理。操作员将作业“成批”地输入到计算机中,由监督程序识别一个作业,进行处理后再取下一个作业。这种自动定序的处理方式称为“批处理(Batch Processing)”方式。而且,由于是串行执行作业,因此称为单道批处理。

3. 多道批处理

进入第三代计算机以后,硬件有了很大发展,通道使得输入输出操作与 CPU 操作并

行处理成为可能。与此同时软件系统也随之相应变化,实现了在硬件提供并行处理之上的多道程序设计。

所谓多道是指它允许多个程序同时存在于主存之中,由中央处理器以切换方式为之服务,使得多个程序可以同时执行,如图 1.3 所示。计算机资源不再是“串行”地被一个个用户独占,而可以同时为几个用户共享,从而极大地提高了系统在单位时间内处理作业的能力。这时管理程序已迅速地发展成为一个重要的软件分支——操作系统。

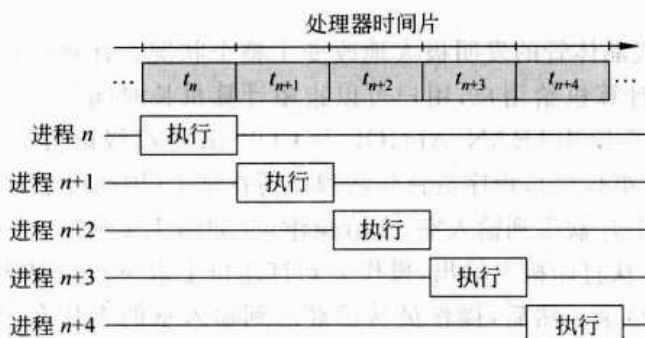


图 1.3 多道程序处理

4. 分时与实时系统

批处理系统不适合交互式的作业,对交互式工作方式的需求导致了分时系统(Time Sharing System)的出现。所谓分时系统是指多个用户通过终端设备与计算机交互作用来运行自己的作业,并且共享一个计算机系统而互不干扰,就好像自己有一台计算机。由于调试程序的用户常常只发出简短的命令(如编译一个 5 页的源文件),而很少有长的费时命令(如上百万条记录的文件排序),所以计算机能够为许多用户提供交互式快速的服务,同时在 CPU 空闲时还能运行大的后台(无须跟用户交互的)作业。

5. UNIX 通用操作系统

20 世纪 60 年代末,贝尔实验室的 Ken Thompson 和 Dennis M. Ritchie 设计了 UNIX 操作系统。它是现代操作系统的代表,显示出了强大的生命力。

UNIX 是用 C 语言编写的,因此它是可移植的,UNIX 是世界上唯一能在笔记本计算机、PC、工作站直至巨型机上运行的操作系统。UNIX 系统源代码非常有效,系统容易适应特殊的需求。UNIX 是一个良好的、通用的、多用户、多任务、分时操作系统。UNIX 有树形文件系统和一定的安全机制;有简洁的字符流文件和文件保护机制;UNIX 有功能强大的 shell,它既是命令解释器,又是编程用高级语言,还可用于扩充系统的功能。

UNIX 不但在理论界有着广泛而深入的影响,而且因为 UNIX 出色的设计思想与实现技术,它在产业界同样掀起了一场革命,许多重要的软件公司相继推出了自己的 UNIX 版本,这进一步推动了操作系统的发展。

6. 个人计算机操作系统

20 世纪 70 年代末期,出现了面向个人计算机的操作系统,例如微软公司的 MS DOS 操作系统。MS DOS 属于单用户单任务操作系统。

1984 年,出现了具有交互式图形功能的苹果操作系统。1992 年微软推出了有交互式图形功能的操作系统 Windows 3.1。Windows 95 在 1995 年 8 月正式登台亮相,从此,Windows 成为个人计算机平台的主流操作系统。

1991 年,Linus 在 Internet 上公布了 Linux 操作系统,于是,Linux 从最开始的一个人的产品逐渐演变变成了一个成功的操作系统。Linux 遵从国际 UNIX 标准 POSIX,继承了 UNIX 的全部优点,而且还开放全部源码,并能在网上自由下载。Linux 极其健壮,世界上很多 Linux 连续不停机运行一年以上而不曾崩溃过。在 UNIX 操作系统家族中,Linux 是具有自由版权的 UNIX 类操作系统中一个较突出的代表。

7. 当代操作系统的两大发展方向——宏观应用与微观应用

在当代,从规模上看,操作系统向着大型和微型两个不同的方向发展着。大型系统的典型是分布式操作系统和集群操作系统,而微型系统的典型则是嵌入式操作系统。

分布式操作系统和集群操作系统是适应计算平台向异构、网络化演变而出现的。分布式系统是由多个连接的处理资源组成的计算系统,它们在整个系统的控制下可合作执行一个共同任务,最少依赖于集中的程序、数据或硬件。这些资源可以是物理上相邻的,也可以是在地理上分散的。

集群操作系统适用于由多台计算机构成的机群。较典型的有用商业化的微型计算机、Linux 和以太网设备等产品构造的机群,如 Beowulf 系统,现在已有分布在全世界各地的约近百个大学和研究机构在使用它。

在当代的电子、电器和智能机械设备上,除了传统的工业控制、航空航天和武器制导等领域外,嵌入式操作系统正在向各类家电、智能电器渗透。VxWorks、pSOS、eCOS、QNX、RTEMS 及嵌入式 Linux,如 RTLinux、KURT 等是较为突出的嵌入式操作系统。图 1.4 是 VxWorks 系统在美国火星探测项目中的一个应用。



图 1.4 登陆火星飞行器“极地探路者”,采用了 VxWorks 嵌入式操作系统

1.3.2 操作系统分类

本节将从另外一种角度,根据操作系统在用户界面的使用环境和功能特征的不同,对操作系统进行分类。分类的目的是为了便于读者更好地理解和掌握操作系统的本质和特征。按照用户界面的使用环境和功能特征的不同,一般可以把操作系统分为三种基本类型,即批处理系统、分时系统和实时系统。随着计算机体系结构的发展,又出现了许多类型的操作系统,它们是人操作系统、网络操作系统、分布式操作系统和嵌入式操作系统。本节将对这些不同类型的操作系统进行简要的介绍和分析。

1. 批处理操作系统

从操作系统的发展历史中可知,批处理操作系统是一种早期的操作系统。在本小节中,首先概要地叙述批处理系统的基本工作方式,然后分析它的特点、种类和设计思想,介绍作业控制说明书、一般指令、特权指令和 SPOOLing,最后分析批处理系统对操作系统发展的贡献。

1) 基本工作方式

批处理操作系统的基本工作方式是:用户将作业交给系统操作员,系统操作员在收到作业后,并不立即将作业输入计算机,而是在收到一定数量的用户作业之后,组成一批作业,再把这批作业输入到计算机中。这批作业可在系统中形成一个连续的、自动转接的作业流。系统操作员然后启动操作系统,系统自动、依次执行每个作业。最后由操作员将执行完毕的作业结果交给用户。

2) 特点与分类

批处理操作系统的特点是成批处理。在批处理系统中,用户自己不能干预自己作业的运行。显然,在用户提交的作业投入运行之后,如果发现错误则无法及时改正,需要重新提交一份经过改正的作业,再次排队运行。这种处理方式延长了软件开发时间,所以批处理更为适用于成熟的程序。

批处理操作系统追求的目标是系统资源利用率高,作业吞吐率高。这里所谓作业吞吐率是指单位时间内计算机系统处理作业的个数。批处理操作系统的优点是作业流程自动化较高,资源利用率较高,作业吞吐量大,从而提高了整个系统效率。其缺点是用户不能直接与计算机无交互,不适合调试程序。

依据系统的复杂程度和出现时间的先后,可以把批处理操作系统分类为简单批处理系统和多道批处理系统。

3) 设计思想

简单批处理系统是在操作系统发展的早期出现的,因此它有时又被称为早期批处理

系统。在操作系统发展的早期,简单批处理系统被称为监控程序。其设计思想是:在监控程序启动之前,操作员有选择地把若干作业合并成一批作业,将这批作业安装在输入设备上。然后启动监控程序,监控程序将自动控制这批作业的执行。系统的运行过程是:先把第一个作业调入主存,运行之;等这一个作业运行结束之后,再把下一个作业调入主存并运行之;如此往复,直到这一批所有的作业都处理完毕。全部作业运行结束之后,操作员把运行的结果一起交给用户。可以看到,在这种系统工作方式下,作业的运行以及作业之间的衔接都由监控程序自动控制,从而有效地提高了作业运行的效率。

4) 作业控制说明书

下面,具体地分析批处理系统的工作原理。在简单批处理系统中,作业的运行步骤是由作业控制说明书来传递给监控程序的。作业控制说明书是由作业控制语言编写的一段程序,它通常存放在被处理作业的前面。在运行过程中,监控程序读入并解释作业前面的这段作业控制说明书中的语句,以控制各个作业步的执行。

这里举一个作业说明书的例子。某用户的作业包括 A、B 两个程序段,A 用汇编语言编写,B 用 FORTRAN 编写。用户要求把两段程序分别汇编、编译之后链接成一个程序并运行它。这个作业说明书可以采取如下编写形式:

```
.STEP1  ASM    A
.STEP2  FTN    B
.STEP3  LINK   A,B,C
.STEP4  RUN    C
```

在这段作业说明书中,.STEP n 指定了第 n 步作业,ASM 表示进行汇编,FTN 表示进行 FORTRAN 编译,LINK 代表进行链接装配,而 RUN 表示执行。A、B、C 分别代表各个不同的程序段。

作业运行后,监控程序逐条解释每一行语句。

① 第一个作业步:指示监控程序调入汇编程序源程序 A 并得到相应的目标代码。汇编结束之后,控制权转回监控程序。

② 第二个作业步:指示监控程序把 FORTRAN 的编译程序调入主存,启动编译程序编译源程序 B,从而得到程序 B 的目标代码。

③ 第三个作业步:监控程序调入链接装配程序,将前面两个作业步得到的目标代码装配成可执行的程序 C。

④ 最后一步:监控程序启动执行程序 C。

5) 一般指令和特权指令

在简单批处理系统的发展阶段,为了防止由于用户的错误而导致整个系统发生不可预料的后果,有必要对中央处理器的运行划分出不同的状态。这就为中央处理器引入了运行模式(mode)的概念。

运行模式通常分为用户模式和特权模式。在操作系统领域中,人们往往把为用户服务的用户模式称为目态,而把为系统专用的特权模式称为管态。相对应地,机器指令被划分为一般指令和特权指令。特权指令包括输入输出指令、停机指令等,只有监控程序才能执行特权指令。用户程序只能执行一般指令。一旦用户程序需要执行特权指令,处理器会通过特殊的机制将控制权移交给监控程序。

有了这样的区分之后,由于输入输出指令属于特权指令,所以用户程序将不能直接使用输入输出指令,即不能直接指挥外部设备,它们必须向操作系统请求这些功能,这个请求通过所谓系统调用或者叫做广义指令完成。监控程序实现这些系统调用的功能。这些系统调用一般都有对应的功能代码,供调用时使用。

6) 系统调用的过程

下面来考察系统调用的处理过程。

① 当系统调用发生时,处理器通过一种特殊的机制,通常是中断或者异常处理,把控制流程转移到监控程序内的一些特定的位置。同时,处理器模式转变成特权模式。

② 由监控程序执行被请求的功能代码。这个功能代码代表着对一段标准程序段的执行,用以完成所请求的功能。

③ 处理结束之后,监控程序恢复系统调用之前的现场;把运行模式从特权模式恢复成为用户方式;最后将控制权转移回原来的用户程序。

7) SPOOLing 技术

简单批处理系统的监控程序犹如一个系统操作员,它负责批处理作业的输入输出,自动根据作业控制说明书以串行方式运行各个作业,并且提供一些最基本的系统功能。但是,它并不具有并发能力,即不能实现多个程序的同时运行。真正引入并发机制的是多道批处理系统。

为了提高硬件资源的利用率,人们在监控程序中间引入了缓冲技术和多道程序设计概念,批处理系统发展为更加高级的多道批处理系统。在多道批处理系统中,关键技术就是多道程序运行、假脱机技术等。

在简单批处理系统中,作业是串行的,作业的执行速度受到各种慢速设备的制约,系统有很多时候(尤其在操纵慢速外设时)只能忙等待,处理器利用率难以提高。为了解决这个问题,出现了脱机输入输出技术。为主机配备相对高速的磁带设备,主机的所有输入输出操作在磁带机上完成,另外配备若干卫星机,它们负责将用户作业从卡片传输到磁带上,执行时,由操作员负责把成卷记录了若干用户作业的磁带装到主机上去处理。这种技术通过使输入输出与计算在不同的设备上并行操作,从而有效地提高了处理器的利用率。不过这种脱机输入输出技术并没有从根本上解决输入输出缓慢的问题,于是出现了假脱机技术,借助硬件通道技术,实现了输入输出操作和处理器动作的自动并行处理。通道是指专门用来控制输入输出的硬件设备,可以看作是专门的 I/O 处理机。通道基本上是自

主控制外设的,可以与 CPU 并行工作。

假脱机(Simultaneous Peripheral Operating On-Line, SPOOLing)技术的全称是“同时的外部设备联机操作”。这种技术的基本思想是用磁盘设备作为主机的直接输入输出设备,主机直接从磁盘上选取作业运行,作业的执行结果也存在磁盘上。相应地,通道则负责将用户作业从卡片机上动态写入磁盘,而这一操作与主机并行。类似的操作也用于打印输出用户作业运行结果。SPOOLing 技术如图 1.5 所示。

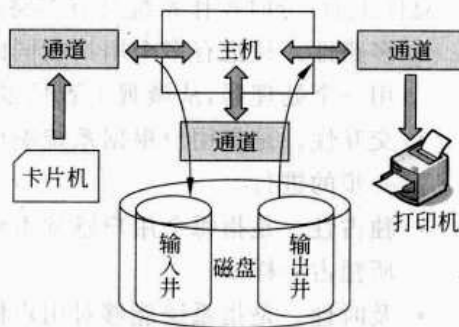


图 1.5 SPOOLing 技术示意图

这里需要指出,通道直接受主机控制,它们之间通过中断相互通信。假脱机技术为实现多道批处理系统中的多道程序设计思想提供了重要的基础。

多道程序设计的基本思想是在内存中同时保持多个作业,主机可以以交替的方式同时处理多个作业。一般来说任何一道作业的运行总是交替使用处理器和外设资源,而不同的作业一般也不会相同时段内要求使用同一台外设或者处理器,如果通过合理的调度,让多个作业交替地同时使用不同的资源,那么就可以大大提高各种外部设备的利用率。这一简单思想通过多道批处理系统得以具体的实现。

2. 分时系统

从操作系统的发展历史上看,分时操作系统出现在批处理操作系统之后。它是为了弥补批处理方式不能向用户提供交互式快速服务的缺点而发展起来的。

1) 基本工作方式

在分时系统中,一台计算机主机连接了若干个终端,每个终端可由一个用户使用。用户通过终端交互式地向系统提出命令请求,系统接受用户的命令之后,采用时间片轮转方式处理服务请求,并通过交互方式在终端上向用户显示结果。用户根据系统送回的处理结果发出下一道交互命令。

2) 设计思想

分时操作系统将 CPU 的时间划分成若干个小片段,称为时间片。操作系统以时间片为单位,轮流为每个终端用户服务。由于每个时间片极短,而计算机处理速度远比人的反应速度快得多,所以尽管所有的用户在轮流使用一个处理器,但这些时间片段轮回的速度远远比用户敲击键盘的速度快,通常处理器总是在一个极短的时间片内先于人的反应而处理完毕了用户发出的交互请求,并在终端屏幕上发出了相应的回应,所以每个用户并不感到有别的用户存在而影响了其对计算机的使用效率,用户似乎感觉到系统是被他独

占的。

3) 特点

总体上看,分时操作系统具有多路性、交互性、“独占”性和及时性的特点。

- 多路性。是指有多个用户在同时使用一台计算机。从宏观上看是多个人同时使用一个处理器,从微观上看是多个人在不同时刻轮流使用一个处理器。
- 交互性。是指用户根据系统响应的结果提出下一个请求。用户直接干预操作每一步的进行。
- 独占性。是指每个用户感觉不到计算机为其他人服务,就好像整个系统为他个人所独占一样。
- 及时性。是指系统能够对用户提出的请求及时给予响应。

分时操作系统追求的目标是及时响应用户输入的交互命令。用来衡量系统及时响应的指标是响应时间,即系统对一个输入的反应时间。在一个交互系统中,通常把用户从终端发出命令到系统给予回答所经历的时间,定义为响应时间。显然,响应时间越短越好。

一般通用操作系统结合了分时系统与批处理系统两种系统的特点。典型的通用操作系统是 UNIX 操作系统。在通用操作系统中,对于分时与批处理的处理原则是:分时优先,批处理在后。

在通用操作系统中,一般把系统的分时和批处理运行状态,称为前台和后台。把处理终端用户的、交互式的、比较小的、但需要及时处理的作业,即分时作业称为前台作业;而把那些需要运行时间较长、要调用其他外部设备的、比较大型、但无须终端用户干预的,且不需要立即处理的作业,即可以批量处理的作业,称为后台作业。换句话说,在系统的“前台”响应需频繁交互的作业,如大量终端的处理要求;而“后台”则处理对时间性要求不强的作业。

3. 实时操作系统

实时操作系统(Real Time Operating System, RTOS)是指,使计算机能在规定的时间内,及时响应外部事件的请求,同时完成对该事件的处理,并能够控制所有实时设备和实时任务协调一致地工作的操作系统。实时操作系统主要目标是:在严格时间范围内,对外部请求做出反应,系统具有高度可靠性。

实时操作系统主要有两类:第一类是硬实时系统。硬实时系统对关键外部事件的响应和处理时间有着极严格的要求,系统必须满足这种严格的时间要求,否则会产生严重的不良后果。火箭和导弹控制、机器人控制、核反应堆控制等是硬实时系统应用的典型领域。飞行控制系统就是硬实时的,它对飞行数据和外部数据采集与处理时机的把握要非常严格,否则就有可能造成飞机碰撞这样的灾难后果。这种时间精确度的要求通常会在微秒以下。第二类是软实时系统。软实时系统对事件的响应和处理时间有一定的时间范

围要求,不能满足相关的要求会影响系统的服务质量,但是通常不会引发灾难性的后果。如视频(信息)处理、电信(自动交换机)、银行和飞机订票等领域的信息处理。在一个网络视频服务中心中,如果不能将视频信息连续定期地(如每一秒钟每个用户 25 帧图像)传送给多个用户,用户将会感觉到影像的停滞、跳动或变形。这种后果,虽然通常不会引发致命或灾难性的后果,但是也影响了系统服务的质量。所以良好的定时特性也是软实时系统的要求。

实时系统为了能够实现硬实时或软实时的要求,除了具有多道程序系统的基本能力外,还需要有以下几方面的能力。

1) 实时时钟管理

实时系统的主要设计目标是对实时任务能够进行实时处理。实时任务根据时间要求可以分为两类:第一类是定时任务,它依据用户的定时启动并按照严格的时间间隔重复运行;第二类是延时任务,它非周期地运行,允许被延后执行,但是往往有一个严格的时间界限。依据任务功能的不同,还可以分为主动式任务和从动式任务,前者依据时间间隔主动运行,多用于实时监控;后者的运行依赖于外部事件的发生,当外部事件出现时(例如某个中断),这种实时任务应尽可能快地进行处理,并且尽量保证不丢失事件。绝大多数实时任务均与时间相关,良好的实时时钟管理能力就成为实时系统的一个关键能力。

2) 过载防护

实时系统中的实时任务往往与环境的依赖关系很大。实时任务的启动时间和数量具有很大的随机性,突发的大量实时任务极有可能超出系统的处理能力,从而发生过载。实时系统在出现过载现象时,要有能力在大量突发的实时任务中,迅速分析判断并找出最重要的实时任务,然后通过抛弃或者延后次要任务以保证最重要任务成功的执行。

3) 高可靠性

高可靠性是实时系统的设计目标之一。由于实时系统往往用在一些关键应用上,如航空控制、工业机器人等领域。实时操作系统的任何故障,都有可能对整个应用系统带来极大的危害。所以实时操作系统需要有很强的健壮性和坚固性。当然高可靠性的要求不仅仅是对软件系统的要求,也是对硬件系统的要求。

4. 个人计算机操作系统

个人计算机操作系统(Personal Computer Operating System)是一种单用户多任务的操作系统。个人计算机操作系统主要供个人使用,功能强,价格便宜,在几乎任何地方都可安装使用。它能满足一般人操作、学习、游戏等方面的需求。个人计算机操作系统的主要特点是:计算机在某一时间内为单个用户服务;采用图形界面人机交互的工作方式,界面友好;使用方便,用户无需具备专门知识,也能熟练地操纵系统。

5. 网络操作系统

为计算机网络配置的操作系统称为网络操作系统(Network Operating System)。网络操作系统是基于计算机网络的,在各种计算机操作系统之上按网络体系结构协议标准设计开发的软件,它包括网络管理、通信、安全、资源共享和各种网络应用。

网络操作系统把计算机网络中的各个计算机有机地连接起来,其目标是相互通信及资源共享。通过网络操作系统,用户可以使用网络中其他计算机的资源,实现计算机间的相互信息交换,从而扩大计算机的应用范围。

计算机网络有不同的模式。在集中式模式中,运算处理在主计算机里发生,终端仅作为输入输出设备使用,通过连接两台或更多主机的方式构成计算机网络。在分布式模式中,每台计算机都有运算处理能力,多台计算机通过网络交换数据并共享资源和服务。在分布式模式中,网络环境中的计算机不仅能共享数据、资源及服务,还能够共享运算处理能力。

6. 分布式操作系统

将大量的计算机通过网络连接在一起,可以获得极高的运算能力及广泛的数据共享。这样一种系统称作为分布式系统(Distributed System)。为分布式系统配置的操作系统称为分布式操作系统(Distributed Operating System)。

分布式操作系统是网络操作系统的更高级形式,分布式操作系统除了保持了网络操作系统的各种功能之外,还具备如下的特征。

(1) 分布式操作系统是一个统一的操作系统。系统中的所有主机使用的是同一个操作系统。

(2) 实现资源的深度共享。在网络操作系统中,由于各个主机使用不同的操作系统,不能随意地将一个计算任务从一台主机迁移到另一台主机执行。而在分布式系统中,通过统一的操作系统的调度,在某台主机上的一个计算任务可以迁移到另一台主机上执行,真正实现了处理机资源的共享。

(3) 透明性。即在用户眼里整个分布式系统像是一台具有强大功能的计算机系统。用户并不知道该分布式系统运行在多少台计算机上,各个主机地理位置上的差异对用户来讲是透明的,分布式操作系统屏蔽了这种差异。相应地,在网络操作系统中,用户能够清晰地感觉到本地主机和非本地主机之间的区别。

(4) 自治性。即处于分布式系统中的各个主机都处于平等的地位,各个主机之间没有主从关系。一个主机的失效一般不会影响整个分布式系统。

分布式系统把系统中的所有计算机构成一个完整的、功能更加强大的计算机系统。分布式操作系统可以使系统中的计算机相互协作,共同完成一个大型计算任务,即把一个

计算任务分解成若干可以并行执行的子任务,让每个子任务分别在不同的计算机上执行。可见分布式操作系统可以充分利用各种资源,从而使计算机系统处理能力增强,速度更快,可靠性更高。

分布式系统的优点在于它的分布式,分布式系统可以以较低的成本获得较高的运算性能。分布式系统的另一个优势是它的可靠性。由于有多个计算机系统,当一个计算机系统发生故障时,整个分布式系统仍旧能够工作。对于高可靠的环境,如核电站等,分布式系统是有其用武之地的。

集群(cluster)是分布式系统的一种,一个集群通常由一群处理器密集构成,集群操作系统专门服务于这样的集群。在集群操作系统的指挥下,可以用低成本的微型计算机和以太网设备等产品,构造出性能相当于超级计算机运算性能的集群。

网络操作系统与分布式操作系统在概念上的主要不同之处在于:网络操作系统可以构架于不同的操作系统之上,也就是说它可以在不同的本机操作系统上通过网络协议实现网络资源的统一配置。在网络操作系统中并不要求对网络资源透明的访问,即需要显式的指明资源位置与类型,对本地资源和异地资源访问区别对待。分布式操作系统强调单一操作系统对整个分布式系统的管理、调度。在这种操作系统中,网络的概念在应用层被淡化了,所有资源,无论是本地的还是异地的,都用同一的方式管理与访问,用户不必关心资源所在的真正物理位置,或者是如何存储的。

1.4 操作系统的内核体系结构模型

操作系统是一种大型软件,为了研制操作系统,必须研究分析它的结构,也就是要考虑如何把一个大型软件划分成若干较小的模块,以及这些模块之间怎样接口。操作系统是一个具有并发特性的大型程序,模块间的接口是相当复杂的,信息交换也是十分频繁的,因而对结构的研究就显得更加重要了。而任何一个软件开发出来投入运行之后,就进入了系统维护阶段。只有易读、易懂,维护人员才能真正了解操作系统的结构和工作原理,从而做好系统维护工作。

所谓操作系统的结构,是指操作系统各部分程序的存在方式及其相互关系。若操作系统的各部分程序以程序模块方式存在,相互之间通过调用建立起关系,那么这种操作系统具有模块接口结构。若各部分程序以进程的方式存在,相互之间通过通信建立起关系,那么这种操作系统具有进程结构。若操作系统能按照诸模块的调用顺序或主进程的信息发送顺序把模块进程分层,各层之间只能单向依赖,这样就分别产生了模块层次结构的操作系统或进程层次结构的操作系统。

在早期的操作系统设计中,由于计算机结构比较简单,系统规模也较小,逻辑关系较简单,设计者往往只注重功能设计和效率,而忽视了结构的设计。但随着计算机结构的复

杂化,应用范围的不断扩大,使用要求的不断提高,人们不仅要求系统有较强的功能,而且要求系统有较强的可适应性和可靠性。比如,在容错的概念出现之后,人们希望设计一种在部分硬、软件失效的情况下仍能正常工作的系统,从事这种容错操作系统的设计,更加需要对操作系统的结构有深入的研究和认识。

可见,无论设计哪一种操作系统,都需要对操作系统的结构和结构设计方法进行研究。近年来,对操作系统的结构和结构设计方法的研究,已成为软件领域的一个重要研究方面。

常见的操作系统体系结构有整体式结构、层次式结构和微内核(客户/服务器)结构等。本节将对这些常见的操作系统体系结构做简要的介绍。

1. 整体式结构

这是早期操作系统设计中所采用的方法,即首先确定操作系统的总体功能,然后将总功能分解为若干个子功能,实现每个子功能的程序称为模块。再按照功能将上述每个大模块分解为若干个较小的模块,如此下去,直至每个模块仅包含单一功能或紧密联系的小功能为止,即分解为最基本的模块为止。最后通过接口将所有模块连接起来形成一个整体。把这种操作系统的结构称之为模块组合结构。它的主要优点是:结构紧密,接口简单直接,系统效率较高。此时,操作系统是一个有多种功能的系统程序,可以看成是一个整体模块,也可看成是由若干个模块按一定的结构方式组成的。

模块组合法(或称无序模块法、模块接口法等)中,系统中的模块不是根据程序和数据本身的特性而是根据它们完成的功能来划分的,数据基本上作为全程量使用。在系统内部,不同模块的程序之间可以不加控制地互相调用和转移,信息的传递方式也可以根据需要随意约定,因而可能造成模块间的循环调用。

模块组合法的缺点有以下三点。

(1) 模块间转接随便,各模块互相牵连,独立性差,系统结构不清晰。

(2) 数据基本上作为全程量处理,系统内所有模块的任一程序均可对其进行存取和修改,从而造成了各模块间有着更为隐蔽的关系。要更换一个模块或修改一个模块都比较困难,因为要弄清各模块间的接口,按当初设计时随意约定的格式来给信息,这是一件相当复杂的事。

(3) 由于模块组合结构常以大型表格为中心,为保证数据完整性,往往采用全局关中断办法,从而限制了系统的并发性。系统中实际存在的并发生也未能抽象出明确的概念,缺乏规格的描述方法。所以,这种结构的可适应性比较差。

模块组合法的关键在于“接口”,因为把各基本模块之间的有机联结都推到“接口”上去了,所以无需太多的结构设计工作,很快就可进入编码阶段,而且模块之间转接的灵活性使得系统具有效率高的优点。但是,由于各基本模块之间可以任意相互调用,各模块之

间相互依赖,甚至可能构成循环,形成一个复杂的网络。这种网络实际上是一种相当复杂的有向图,它使得难于对结构做出综合性的观察,也难于对系统做局部性修改,因而可靠性、易读性和适应性都很难得到保证。随着系统规模的不断增大,采用这种结构构造的系统的复杂性迅速增长,以致人们难以驾驭,这就促使人们去寻求新的结构概念和新的结构设计方法。

2. 层次式结构

从对整体式结构缺点的分析可以看到,要克服整体式结构的缺点,就必须减少各模块之间毫无规则地互相调用、相互依赖的关系,特别是循环调用现象。层次式结构正是从这点出发,力求使模块间调用的无序性变为有序性。因此所谓层次式结构就是把操作系统的所有功能模块,按功能流图的调用次序,分别将这些模块排列成若干层,各层之间的模块只能是单向依赖或单向调用(如只允许上层或外层模块调用下层或内层模块)关系。这样不但操作系统的结构清晰,而且不构成循环。

在一个层次结构的操作系统中,若不仅各层之间是单向调用的,而且每一层中的同层模块之间不存在互相调用的关系,则称这种层次结构关系为全序的层次关系。

但是在实际的大型操作系统中,要建成一个全序的层次结构关系是十分困难的,往往无法完全避免循环现象,此时应使系统中的循环现象尽量减少。

层次结构法的优点是:它既具有模块接口法的优点——把复杂的整体问题分解成若干个比较简单的相对独立的成分,即把整体问题局部化,使得一个复杂的操作系统分解成许多功能单一的模块。同时它又具有模块接口法不具有的优点,即各模块之间的组织结构和依赖关系清晰明了。这不但增加了系统的可读性和可适应性,而且还使我们对操作系统的每一步扩展都建立在可靠的基础上。

因为层次结构是单向依赖的,上一层各模块所提供的功能(以及资源)是建立在下一层的基础上的。或者说上一层功能是下一层的扩充和延续。最内层是硬件基础——裸机,裸机的外层是操作系统的最下面(或内层)的第一层。按照分层虚拟机的观点,每加上一层软件就构成了一个比原来机器功能更强的虚拟机,也就是说进行了一次功能扩充。而操作系统的第一层是在裸机基础上进行的第一次扩充后形成的虚拟机,以后每增加一层软件就是在原虚拟机上的又一次扩充,又成为一个新的虚拟机。因此只要下层的各模块设计是正确的,就为上层功能模块的设计提供了可靠基础,从而增加了系统的可靠性。

层次式结构的优点还在于很容易对操作系统增加或替换掉一层而不影响其他层次。不难理解,采用层次结构法设计的操作系统具有易于调试、修改、扩充、维护、保证正确性等优点,因而被广泛地采用。

层次式结构的操作系统的各功能模块应放在哪一层,系统一共应有多少层,这是一个很自然地会提出的问题。但对这些问题通常并没有一成不变的规律可循,必须要依据总

体功能设计和结构设计中的功能流图和数据流图进行分层,大致的分层原则如下。

(1) 为了增加操作系统的可适应性,并且方便于将操作系统移植到其他机器上,必须把与机器特点紧密相关的软件,如中断处理、输入输出管理等放在紧靠硬件的最低层。这样经过这一层软件扩充后的虚拟机,硬件的特性就被隐藏起来了,方便了操作系统的移植。为了便于修改移植,它把与硬件有关和与硬件无关的模块截然分开,而把与硬件有关的部分如 BIOS(管理输入输出设备)或 HAL(硬件抽象层)放在最内层。所以当硬件环境改变时只需要修改这一层模块就可以了。

(2) 对于一个计算机系统来说,往往具有多种操作方式(例如既可在前台处理分时

用户界面
应用程序
作业管理
文件管理
输入输出管理
进程间通信
存储管理
处理器管理
硬件

作业,又可在后台以批处理方式运行作业,也可进行实时控制),为了便于操作系统从一种操作方式转变到另一种操作方式,通常把三种操作方式共同要使用的基本部分放在内层,而把随三种操作方式而改变的部分放在外层:如批作业调度程序和联机作业调度程序,键盘命令解释程序和作业控制语言解释程序等,这样操作方式改变时仅需改变外层,内层部分保持不变。

(3) 当前操作系统的设计都是基于进程的概念,进程是操作系统的基本成分。为了给进程的活动提供必要的环境和条件,因此必须要有一部分软件——系统调用的各功能,来为进程提供服务,通常这些功能模块(各系统调用功能)构成操作系统内核,放在系统的内层。内中又分为多个层次,通常将各层均要调用的那些功能放在更内层,如图 1.6 所示。

图 1.6 一种操作系统的层次式结构示意图

在操作系统的层次式结构研究和应用中,还演化出几种不同的层次式结构,如进程分层结构、层次管程结构、虚拟机结构等,限于篇幅,就不在这里介绍了。

3. 微内核(客户/服务器)结构

当前计算机技术发展的突出特点是要求广泛的信息和资源的共享。所以,分布式处理合乎客观实际和新的应用需要。

在这样的应用环境下,操作系统的体系结构也在发展和变化。采用客户/服务器结构的操作系统就适宜于应用在网络环境下分布式处理的计算环境中。由于这种体系结构所具有的一些特征,它又被称为微内核体系结构。

采用客户/服务器结构模式的典型操作系统有卡内基·梅隆大学研制的 Mach 操作系统和 Windows NT 的早期版本。它们的共同特点如下。

(1) 运行在核心态的内核。内核提供所有操作系统基本都具有的操作,如线程调度、虚拟存储、消息传递、设备驱动以及内核的原语操作集和中断处理等。这些部分通常采用层次结构并构成了基本操作系统。因为这时的内核只提供了一个很小的功能集合,所以通常又称为微内核。

(2) 运行在用户态并以客户/服务器方式运行的进程层。除内核部分外,操作系统所有的其他部分被分成若干个相对独立的进程,每一个进程实现一组服务,称为服务进程(用户应用程序对应的进程,虽然也以客户/服务器方式活动于该层,但不作为操作系统的功能构成成分看待)。这些服务进程可以提供各种系统功能、文件系统服务以及网络服务等。服务进程的任务是检查是否有客户机提出要求服务的请求,并在满足客户机进程的请求后将结果返回。而客户机可以是一个应用程序,也可以是另一个服务进程。客户机进程与服务器进程之间的通信是采用发送消息进行的,这是因为每个进程属于不同的虚拟地址空间,它们之间不能直接通信,必须通过内核进行,而内核则是被映射到每个进程的虚拟地址空间内的,它可以操纵所有进程。客户机进程发出消息,内核将消息传给服务进程。服务进程执行相应的操作,其结果又通过内核用发消息方式返回给客户机进程,这就是客户/服务器的运行模式。

这种模式的优点在于,它将操作系统分成若干个小的并且自包含的分支(服务进程),每个分支运行在独立的用户进程中,相互之间通过规范一致的方式接收发送消息而联系起来。操作系统在内核中建立起了最小的机制,而把策略留给用户空间中的服务进程,这带来了很大的灵活性,直接的好处如下。

(1) 可靠。由于每个分支是独立的自包含的(分支之间耦合最为松散),所以即使某个服务器失败或产生问题,也不会引起系统其他服务器和系统其他组成部分的损坏或崩溃。

(2) 灵活。便于操作系统增加新的服务功能,因为它们是自包含的,且接口规范。同时修改一个服务器的代码不会影响系统其他部分,可维护性好。

(3) 适宜于分布式处理的计算环境。因为不同的服务可以运行在不同的处理器或计算机上,从而使操作系统自然地具有分布式处理的能力。

当然这种体系结构也有它的缺陷,主要是对于效率的考虑。因为所有的用户进程只能通过微内核相互通信,微内核本身就成为系统的瓶颈。在一个通信很频繁的系统中,微内核往往不能提供很好的效率。例如高性能的图形用户界面系统中经常有大量的数据在不同的进程中来回复制,那么把图形引擎作为一个运行在用户态的服务进程对一个有着高性能图形需求的系统来说将是不明智的选择。

1.5 Linux 的特点及发展简史

1.5.1 什么是 Linux

Linux 是由 Linus Torvalds 和许多自由软件开发者,通过 Internet 共同努力而形成的一种能运行于多种硬件平台(Intel、Alpha、PowerPC 和 Sun SPARC)之上,源代码公开,功能强大,运行稳定高效,遵守 POSIX 标准,与 UNIX 兼容的免费操作系统。

1.5.2 Linux 的发展历程

Linux 的历史可以上溯到 20 世纪 60 年代末,美国贝尔实验室发明了 UNIX,一个多用户多任务的操作系统。那个年代,计算机程序的源码都是公开的,尽管它们很多时候只能被专业前沿的程序员们看懂。到了 20 世纪 70 年代,操作系统开始商业化,出于利益最大化等的考虑,源代码开始向用户封闭,这给许多传统的程序员黑客带来了不便,他们能够发现程序中的漏洞并有解决的能力,但是提供程序的商业公司却拒绝让用户直接修改。对传统黑客来说,这对计算机软件的发展是一个相当大的限制。

1984 年,黑客 Richard M Stallman 成立了自由软件基金会(Free Software Foundation,FSF)和开源项目计划 GNU,并提出了著名的开源协议标准 GPL。GNU 的意思就是 GNU's Not UNIX,以一个递归式的定义描述了自己和 UNIX 有关又不同于 UNIX 的特点。其计划是开发出一套完整的免费、公开源代码的 UNIX 操作系统和及其应用软件。

到 20 世纪 80 年代末,GNU 计划的很多工作已经完工,包括了 C 语言的编译器 GCC,文本编辑软件 emacs 等,但是操作系统计划 Hurd 却迟迟没有推出。这时芬兰大学生 Linus Tovalds 于 1991 年 10 月在校的 ftp 上发布了自己所编写的类 UNIX 操作系统——Linux 0.02 版的源代码,并宣布它遵守 GPL 协议,而且符合 UNIX 的操作系统 POSIX 标准,源代码可以在 UNIX 主机上用 gcc 编译生成可执行的二进制代码,可以在个人计算机平台(Intel 80386)上运行。这些纯粹是源于他的业余爱好。Linus 在上大学时,操作系统课程中讲述的是 Minix,一种用于教学的小型类 UNIX 操作系统。他觉得 Minix 不太好,决定自己写一个保护模式下的操作系统。1991 年 8 月,Linus 完成了 Linux 的 0.01 版,当然它很不完善,甚至没有软盘驱动程序。

当 Linus 最初在互联网上公布时,正是互联网和个人电脑兴起和开始高速发展的时代,它吸引了世界各地的黑客对 Linux 进行修改和完善,但所有对内核的改动都要通过 Linus。靠 Linus 的把关,Linux 的内核保持着统一,不至于像发行版本那样,种类很多。后来,随着系统在网络上的不断分发、传播,知道的人越来越多,吸引了成千上万的软件高

手。到 1994 年发布正式的 1.0 版本时,已经有了相当大的名气,GNU 组织也暂且搁下自己的 Hurd 计划,全力支持 Linux 的发展。我们今天说的 Linux,事实上只是一个简称,它的正式名称是 GNU/Linux。

十多年过去了,通过全世界许多软件高手,特别是 Linus 本人的不懈努力,Linux 终于发展成为主流操作系统的一员。它在服务器端可与商用 UNIX 相媲美,在客户端则向 Windows 发起了挑战。最近几年,随着 Linux 的优良性能不断显现,不少商业科技巨头纷纷涉足 Linux。最近,IBM 就准备联合三家日本电子业巨头向 Linux 投巨资来开发商用 Linux 系统。

1.5.3 Linux 的特性

Linux 内核具有以下特性。

1. 多用户

多用户是指系统资源可以被不同用户各自拥有并使用,即每个用户对自己的资源(如文件、设备)有特定的权限,互不影响。在 Linux 系统中不但可以拥有许多用户账号,而且可以让多个用户在同一时间登录进入系统并在系统上同时工作。每个用户都能拥有按自己意愿定制的环境:存放自己文件的主目录、自己的桌面界面(如自己安排的图标、菜单和应用等)。

2. 多任务

多任务是现代计算机的最主要的一个特点。它是指计算机同时执行多个程序,而且各个程序的运行互相独立。Linux 系统调度每一个进程,平等地访问微处理器。由于 CPU 的处理速度非常快,所以启动的应用程序看起来好像在并行运行。事实上,从处理器执行一个应用程序中的一组指令到 Linux 调度微处理器再次运行这个程序之间只有很短的时间延迟,用户是感觉不出来的。在 Linux 中,可以同时运行多个程序。这些程序不仅包括各个在线的用户启动的许多用户程序,还包括 Linux 本身在后台运行的程序。这些在后台运行的系统进程使得 Linux 系统作为服务器成为可能。多任务可以说是现代操作系统的基本特点。

3. 多平台

Linux 所支持的平台种类是操作系统历史上最多的,如 Sun SPARC、SGI MIPS、Apple Mac、PowerPC、Alpha、HP-PA、IntelX86 及 PDA 装置等,可以说无所不在。Linux 内核的高度可移植性由此可见一斑。这主要得益于 Linux 的大部分代码是用 C 语言写的,而 C 语言具有很好的可移植性。这也是 Linux 发展如此之快的一个原因。在 Linux

出现之前,对于 Intel CPU 系列的 PC 机用户来说,UNIX 是可望不可即的。Linux 对于 Intel CPU 的强力支持推动它正快步迈入主流操作系统世界。

4. 良好的用户界面

Linux 向用户提供了两种界面:用户界面和系统调用。Linux 的传统用户界面是基于文本的命令行界面,即 shell,它既可以联机使用,又可存在文件上脱机使用。可编程 Shell 是指将多条命令组合在一起,形成一个 Shell 程序,这个程序可以单独运行,也可以与其他程序同时运行。shell 有很强的程序设计能力,用户可方便地用它编制程序,从而为用户扩充系统功能提供了更高级的手段。同时 Linux 也传承了 UNIX 强大的图形用户界面——X-Window。X-Window 是支持图形化应用程序正常工作的框架。它与 GUI 应用进行通信,并在 X 服务器进程(管理用户的屏幕、鼠标、键盘的进程)上进行显示。

系统调用给用户编程时使用的界面。用户可以在编程时直接使用系统提供的系统调用命令。系统通过这个界面为用户程序提供低级、高效率的服务。

5. 设备独立性

设备独立性是指操作系统把所有外部设备统一当作成文件来看待,只要安装它们的驱动程序,任何用户都可以像使用文件一样,操纵、使用这些设备,而不必知道它们的具体存在形式。

具有设备独立性的操作系统,通过把每一个外围设备看作一个独立文件来简化增加新设备的工作。当需要增加新设备时,系统管理员就在内核中增加必要的连接。这种连接(也称作设备驱动程序)保证每次调用设备提供服务时,内核以相同的方式来使用它们。设备独立性的关键在于内核的适应能力。其他操作系统只允许一定数量或一定种类的外部设备连接。而具备设备独立性的操作系统能够容纳任意种类及任意数量的设备,因为每一个设备都是通过其与内核的专用连接独立进行访问。

Linux 是具有设备独立性的操作系统,它的内核具有高度适应能力,随着更多的程序员加入 Linux 编程,会有更多硬件设备加入到各种 Linux 内核和发行版本中。另外,由于用户可以免费得到 Linux 的内核源代码,因此,用户可以修改内核源代码,以便适应新增的外部设备。

6. 强大的通信和联网功能

完善的内置网络是 Linux 的一大特点。Linux 在通信和网络功能方面具有明显的优势。其他操作系统通常不包含如此紧密地和内核结合在一起的连接网络的能力,也没有内置这些联网特性的灵活性。而 Linux 为用户提供了完善的、强大的网络功能。

(1) 支持 Internet。Linux 免费提供了大量支持 Internet 的软件,Internet 是在

UNIX 领域中建立并繁荣起来的,在这方面使用 Linux 是相当方便的,用户能用 Linux 与世界上的其他人通过 Internet 网络进行通信。

(2) 文件传输。用户可以通过一些 Linux 命令完成内部信息或文件的传输。

(3) 远程访问。Linux 不仅允许进行文件的传输,还为系统管理员和技术人员提供了访问其他系统的应用软件。通过这种远程访问的功能,一位技术人员能够有效地为多个系统服务,即使那些系统位于相距很远的地方。

Linux 支持种类繁多的网卡、调制解调器和串行设备等连接设备。除了对局域网各种协议(如 Ethernet、Token Ring 等)的支持外,Linux 还内建了对绝大部分流行的上层网络协议(如 Internet 中大行其道的 TCP/IP、Novell 的 IPX 及分组交换网的 X.25)的支持。

Linux 还为局域网用户甚至于整个 Internet 提供了许多强大的网络服务。Linux 中有各种软件包,可用来搭建文件/打印服务器、Web 服务器、FTP 服务器、Mail 服务器、News 服务器或工作组服务器等。

7. 可靠的系统安全

Linux 采取了许多安全技术措施,包括对读、写进行权限控制、带保护的子系统、审计跟踪和核心授权等,这为网络多用户环境中的用户提供了必要的安全保障。

8. 开放性

开放性是指系统遵循世界标准规范,特别是遵循开放系统互连(Open System Interconnection, OSI)国际标准。凡遵循国际标准所开发的硬件和软件,都能彼此兼容,可方便地实现互连。

9. 应用程序支持

由于与 POSIX 标准及几个不同的应用程序设计接口(Application Program Interface, API)兼容,因此 Linux 可以使用的免费软件或共享软件的范围很广。绝大多数 GNU 软件都可以运行在 Linux 上。经过短短十多年的发展,不仅许多 UNIX 下的软件都已移植到了 Linux 下,而且人们还为 Linux 开发了不少软件。

1.5.4 Linux 内核的版本演化介绍

Linux 内核是不断演化的。Linux 内核版本分为两种:稳定版和开发版。其内核版本命名规则为:主版本号.次版本号.修订版本号。如果次版本号是偶数,说明该内核是稳定版本;否则就是开发版本。例如,2.6.* 是稳定版本,而 2.5.* 就是开发版本。下面给出 2.4 版和 2.6 版所增加的新特性和新功能。

1. Linux 2.4 版本的主要新特性

(1) 对于编译生成的内核镜像,旧版本内核仅仅支持 zImage/Image,从 Linux 2.4 版本开始支持 bzImage。

(2) 从 Linux 2.4 版本开始,才真正对 SMP 有良好的支持。

(3) Linux 2.4 版本修正了一些 VFS 层(虚拟文件系统)的问题并对文件缓冲区做了修改,将复杂的双缓冲区改为简单有效的单缓冲系统。

(4) Linux 2.4 增加了 dvd 驱动器使用的 udf 文件系统和早期 irix 版本的 efs 文件系统;加入了对嵌入式设备使用的 jffs 和 ramfs 文件系统;第一次对最新的 nfs 协议版本 nfsv3 支持。

(5) Linux 2.4 版本引入了设备文件系统(DevFS),它主要是为了有效的管理/dev 目录而开发的。但是由于种种原因在以后的版本中没有继续使用。

(6) Linux 2.4 版本从文件 api 中把块设备的 api 独立出来;在主流内核中加入 lvm(逻辑卷管理)子系统;raid 子系统对单处理器系统 and 对称多处理系统都作了优化。

(7) Linux 2.4 支持的 ide 控制器数目从 4 个增加到 10 个。

(8) Linux 2.4 版本一个全新的特性是实现了 raw I/O 设备,能够用于数据要求严格的情况下。

(9) Linux 2.4 版本改写了很多设备驱动程序,如帧缓冲、网络硬件和并行端口子系统等。

(10) Linux 2.4 新加入内核级 Web 守护程序——khttpd。

(11) Linux 2.4 加入了对三种新体系的支持: ia64 (itanium)、s/390 和 superH (Windows CE 的硬件)。

(12) Linux 2.4 还包括了一套更丰富的文档,包括首次加入的 DocBook 文档格式,它类似于 HTML,并被 GNOME 和其他 GNU 项目采用。

2. Linux 2.6 版本的主要新特性

(1) 中断。现在的中断处理函数的返回代码是一个 irqreturn_t 类型,可以用来捕获假中断。

(2) 调度器。Linux 2.6 版本的 Linux 内核使用了新的 $O(1)$ 调度算法,而 Linux 2.4 版本的调度器的复杂度为 $O(n)$ 。

(3) 内核抢占。抢占式内核使得用户程序在执行系统调用期间可以被抢占,从而使新被唤醒的高优先级进程能够运行。Linux 2.6 是抢占式内核。

(4) 工作队列接口。用于取代任务队列接口。每个工作队列有一个专门的线程,工作队列中任务在进程上下文运行,从而可以休眠。而 Linux 2.4 版本的任务队列多是不

可以休眠的。

(5) 线程模型。在 Linux 2.6 版本中出现的是 NPTL(Native POSIX Threading Library),而 Linux 2.4 版本的是 Linux Threads。

(6) Linux 2.6 内核现在可以处理任意数目的线程(Linux 2.4 内核中每个处理器最多支持 8192 个线程),PID 的最大数量为 10 亿(Linux 2.4 中为 32000)。引入了 TLS(Thread Local Storage)系统调用。

(7) 引入了退出系统调用的一个变种 `exit_group()`,这个系统调用终止整个进程和它的线程。

(8) 修改了 `proc` 文件系统,不再报告所有的线程而只是报告原始的线程,避免了 `/proc` 报告速度的下降。

(9) 支持更大数量的 RAM,在 32 位系统上在分页模式下最高可达 64GB。这归功于 PAE(Physical Address Extensions,物理地址扩展)。

(10) 提供了对非一致内核访问(Non-Uniform Memory Access,NUMA)系统的支持。

(11) 内存池。Linux 2.6 中加入了内存池技术。

(12) 反向映射。从虚拟内存的角度来看,新内核融合了反向映射(reverse mapping, RMAP)技术,将显著改善虚拟内存在一定程度负载下的性能。

(13) 文件系统。Linux 2.6 所允许的可寻址文件系统大小最大为 16TB,对 EXT2、EXT3 和 ReiserFS 等传统 Linux 文件系统做了改进,支持 IBM 的 JFS(Journaling File System)和 SGI 的 XFS。支持以读/写模式安装 NTFS 文件系统。

(14) 引入了新的文件系统变化通知机制 `inotify`(如文件增加、删除等事件可以立刻让用户态得知)。

(15) 支持更多的设备。在 Linux 2.6 版本之前,内核的某些限制会约束大型系统,如每条链 256 个设备。而 Linux 2.6 内核不但可以支持更多类型的设备(4095 中主要设备类型),而且支持更多同类型的设备(每一类型可以有超过 100 万个子设备)。

(16) 引入了 `sys` 文件系统(挂载在 `/sys` 目录下),它给出了系统的设备模型的用户空间描述。

(17) 统一设备模型是创建 Linux 2.6 内核最重要的变化之一,其目的是更好地控制和管理设备。这个设备模型通过维持大量的数据结构囊括了几乎所有的设备结构和系统。它促进了模块接口的标准化。

(18) 内核模块。新的内核模编译后产生 `.ko`(kernel object)内核目标文件,而不是之前的 `.o` 模块目标文件。在 Linux 2.6 中,模块不再需要对引用计数进行加或减,这些工作将在模块代码外部进行。

(19) 用于 x86 体系结构的 HZ 值增加到 1000。引入一个叫做 `jiffies_64` 的瞬间计算

器,以避免由于 HZ 值的变化而引起瞬间变量的迅速溢出。

(20) 引入了一个叫做 seqlock() 的新类型的锁,用于锁定小段的经常被访问的数据(不是指针)。引入了新的锁机制——RCU(Read-Copy Update,读-副本修改),适用于读者多写者少的情况。

1.6 Linux 内核的结构及主要组成部分

1.6.1 Linux 内核系统体系结构

Linux 内核主要由 5 个子系统组成: 进程调度、内存管理、虚拟文件系统、网络接口和进程间通信。

(1) 进程调度(SCHED)。控制进程对 CPU 的访问。当需要选择下一个进程运行时,由调度程序选择最值得运行的进程。可运行进程实际上是指那些仅等待 CPU 资源的进程,如果某个进程在等待其他资源,则该进程是不可运行进程。Linux 使用了比较简单的基于优先级的进程调度算法选择新的进程。

(2) 内存管理(MM)。允许多个进程安全的共享主内存区域。Linux 的内存管理支持虚拟内存,即在计算机中运行的程序,其代码、数据、堆栈的总量可以超过实际内存的大小,操作系统只是把当前使用的程序块保留在内存中,其余的程序块则保留在磁盘中。必要时,操作系统负责在磁盘和内存间交换程序块。内存管理从逻辑上分为硬件无关部分和硬件有关部分。硬件无关部分提供了进程的映射和逻辑内存的对换;硬件相关的部分为内存管理硬件提供了虚拟接口。

(3) 虚拟文件系统(Virtual File System,VFS)。隐藏了各种硬件的具体细节,为所有的设备提供了统一的接口,VFS 提供了多达数十种不同的文件系统。虚拟文件系统可以分为逻辑文件系统和设备驱动程序。逻辑文件系统指 Linux 所支持的文件系统,如 EXT2,FAT 等,设备驱动程序指为每一种硬件控制器所编写的设备驱动程序模块。

(4) 网络接口(NET)。提供了对各种网络标准的存取和各种网络硬件的支持。网络接口可分为网络协议和网络驱动程序。网络协议部分负责实现每一种可能的网络传输协议。网络设备驱动程序负责与硬件设备通信,每一种可能的硬件设备都有相应的设备驱动程序。

(5) 进程间通信(IPC)。支持进程间各种通信机制。

它们之间的关系如图 1.7 所示。

处于中心位置的进程调度,所有其他的子系统都依赖它,因为每个子系统都需要挂起或恢复进程。一般情况下,当一个进程等待硬件操作完成时,它被挂起;当操作真正完成时,进程被恢复执行。例如,当一个进程通过网络发送一条消息时,网络接口需要挂起发

送进程,直到硬件成功地完成消息的发送,当消息被成功地发送出去以后,网络接口给进程返回一个代码,表示操作的成功或失败。其他子系统以相似的理由依赖于进程调度。

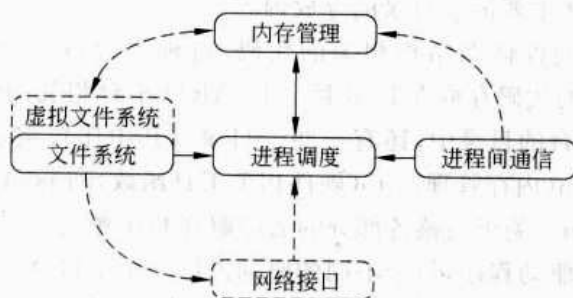


图 1.7 Linux 内核 5 个子系统的关系

各个子系统之间的依赖关系如下。

- 进程调度与内存管理之间的关系。这两个子系统互相依赖。在多道程序环境下,程序要运行必须为之创建进程,而创建进程的第一件事情,就是将程序和数据装入内存。
- 进程间通信与内存管理的关系。进程间通信子系统要依赖内存管理支持共享内存通信机制,这种机制允许两个进程除了拥有自己的私有空间,还可以存取共同的内存区域。
- 虚拟文件系统与网络接口之间的关系。虚拟文件系统利用网络接口支持网络文件系统(Network File System,NFS),也利用内存管理支持 RAMDISK 设备。
- 内存管理与虚拟文件系统之间的关系。内存管理利用虚拟文件系统支持交换,交换进程(swapd)定期由调度程序调度,这也是内存管理依赖于进程调度的唯一原因。当一个进程存取的内存映射被换出时,内存管理向文件系统发出请求,同时,挂起当前正在运行的进程。

除了这些依赖关系外,内核中的所有子系统还要依赖于一些共同的资源。这些资源包括所有子系统都用到的过程。例如,分配和释放内存空间的过程,打印警告或错误信息的过程,还有系统的调试例程等。

1.6.2 Linux 内核的源码组织结构

Linux 的内核源代码的组织采用目录树的结构,通常每个有源码文件的目录下都有 Makefile 和 Kconfig(Linux 2.6 内核,Linux 2.4 用 config.in)两个文件,用于组织内核的配置和编译。顶层目录下的 Makefile 是整个内核配置和编译的总体控制文件,负责启动配置工具、组织子目录编译。另外,还可设置目标 CPU 的体系结构和交叉编译命令等。

如果是初次接触 Linux 内核源代码,可通过顶层目录下的 readme 文件了解相关的内

核介绍和编译命令。但需注意该文件主要是针对 i386 平台的,对别的体系结构不见得完全适用。

下面简要介绍一些主要的子目录的存放内容。

- arch。包含与硬件体系结构相关的代码,每种平台占一个相应的目录。例如,32 位 PC 相关的代码存放在 i386 目录下,ARM 平台相关的代码存放在 arm 目录下。在每个平台的目录中,还有一些子目录。其中比较重要的包括 kernel(内核核心部分)、mm(内存管理)、lib(硬件相关工具函数)和 boot(引导程序)等。
- Documentation。关于内核各部分的通用解释和注释。
- drivers。设备驱动程序,每个不同的驱动占用一个子目录。
- fs。各种支持的文件系统,如 EXT、FAT 和 NTFS 等。
- include。内核头文件。其中,include/asm-*/中是体系结构相关的头文件(*表示体系结构的名称,如 include/asm-arm)。include/linux 中是内核基本的头文件。
- init。内核 C 语言部分的初始化代码(注意不是系统引导代码)。
- ipc。进程间通信的代码。
- kernel。内核的最核心部分,包括进程调度、定时器等,和平台相关的一部分代码放在 arch/*/kernel 目录下。
- lib。各种库文件代码。
- mm。内存管理代码,和平台相关的一部分代码放在 arch/*/mm 目录下。
- net。网络相关代码,实现了各种常见的网络协议。
- crypto。常用加密和散列算法(如 AES、SHA 等),还有一些压缩和 CRC 校验算法。
- scripts。用于配置内核文件的脚本文件。
- sound。常用音频设备的驱动程序等。
- usr。实现了一个 cpio。cpio 可以从 cpio 或 tar 格式的归档包中存入和读取文件,归档包是一种包含其他文件和有关信息的文件。

Linux 的内核源代码将与体系结构相关的部分统一组织到一个目录下,极大的方便了内核的移植。

1.7 本章小结

本章介绍了操作系统基本概念,主要有操作系统的定义、操作系统的基本特征。操作系统是计算机系统中的一个系统软件,它由一些程序模块组成,这些模块完成组织和管理计算机系统中的硬件及软件资源,合理地组织计算机工作流程,控制程序的执行,并向用

户提供各种服务的功能,使用户能够灵活、方便、有效地使用计算机,从而使整个计算机系统能高效地运行。介绍操作系统在计算机系统内的特殊地位和起到的作用,它具有并发、共享和随机三个主要特征。

从管理计算机系统软硬件资源的角度,操作系统应该具有的主要功能,包括进程(线程)管理、内存管理、文件管理和设备管理。

本章还介绍了操作系统的发展历史,以及操作系统的分类。一般操作系统可以分为批处理操作系统、分时操作系统、实时操作系统、个人计算机操作系统、网络操作系统、分布式操作系统和嵌入式操作系统等若干类型。

本章的最后介绍了 Linux 操作系统的起源、发展历史;介绍了 Linux 操作系统的特性。并给出 Linux 内核结构以及 Linux 内核源代码结构。

思考题

1. 计算机硬件系统主要由运算器、主存储器、控制器、输入输出设备、辅助存储器及将这些部件连接起来的系统总线等功能部件组成。请举出一个包含了上述主要硬件部件的、一个尽可能最小配置的、实用的单板计算机硬件系统(或者单片计算机,或者片上系统(System on Chip))。例如,辅助存储器可以不配备,也可以只提供输入输出的接口,不必直接配备有输入输出设备。根据这样一个最小配置的、实用的计算机硬件系统所能从事的应用环境,提出一个操作系统的功能配置要求。比如,某个小计算机硬件系统可能对从事某一类实时工业控制合适,另一类小计算机硬件系统可能适合用于掌上计算机。显然这两种应用对操作系统的功能要求是不一样的。

2. 了解 Linux 最新版本的情况,总结出最新版本提供的新功能。

进一步阅读

<http://www.cs.helsinki.fi/u/torvalds/>

<http://wiki.ccw.com.cn/Linux>

<http://www.linux.org/>

<http://www.chinaunix.net/>

第2章 计算机系统硬件基础

计算机系统是由硬件系统与软件系统组成的,它们之间相互依存,互相配合完成具体的应用。图 2.1 为计算机系统的结构图。

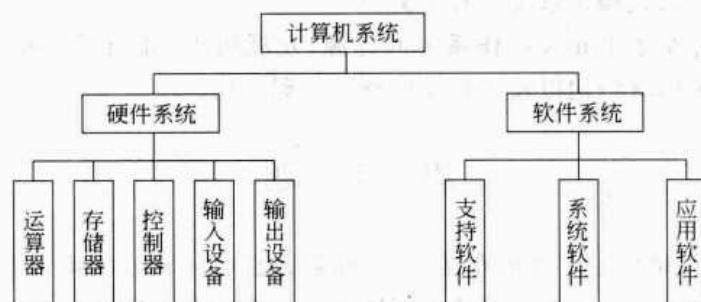


图 2.1 计算机系统结构图

硬件系统包括运算器、控制器、存储器、输入设备和输出设备。软件系统包括系统软件、支持软件与应用软件。Linux 操作系统就是一种系统软件。在深入了解 Linux 操作系统之前,先了解一下它的硬件运行环境。本章主要讨论计算机硬件的组成与工作原理以及硬件对操作系统的支持与需求。

2.1 计算机硬件的组成结构

现在我们使用的计算机,主要采用存储程序和程序控制的设计思想,它是由世界著名的美籍匈牙利数学家冯·诺依曼提出的。因此冯·诺依曼被称为“计算机之父”。

冯·诺依曼计算机由运算器、控制器、存储器、输入设备和输出设备 5 部分组成,如图 2.2 所示。图中的实线箭头代表程序或数据,虚线箭头代表控制信号。程序和数据可以从输入设备读入存储器,再经存储器送到输出设备输出。存储器中的数据可以送到运算器或输出设备,但是存储器中的指令只能送到控制器。运算器可以从存储器和输入设备接收数据,也可以将数据送到存储器或输出设备。控制器可以向每一个设备发送控制信号(如图中的虚线箭头)。只有控制器才能向输入输出设备或存储器传送地址(地址传送路线在图中没有标出)。

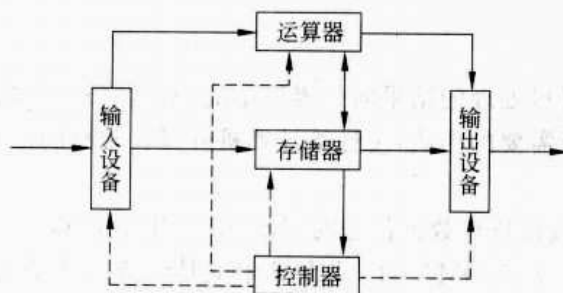


图 2.2 冯·诺依曼计算机组成

1. 运算器

运算器是进行算术运算和逻辑运算的部件。算术运算是指各种数值运算(如加、减、乘、除等),逻辑运算则是指因果关系判断等非数值运算(如与、或、非等)。运算器的核心部件是加法器和若干个高速寄存器。前者用于运算,后者用于存放参加运算的各种数据及运算结果。

2. 存储器

存储器是用来存放程序 and 数据的,是一种具有记忆功能的部件。存储器分为主存储器和辅助存储器两类。主存储器也称为主存或者内存,主要是存放正在运行的程序和数据。辅助存储器指的是磁盘、光盘或磁带等外部存储设备,用于长期保存程序和数据。这里讨论的存储器一般指的是主存储器。

3. 控制器

控制器是分析和执行指令的部件,负责指挥和控制计算机的所有部件协调工作,是整个计算机的控制中枢。控制器从存储器中获取程序指令,经过译码产生机器指令,根据相应的控制指令控制系统中其他部件的动作,使计算机按照程序所规定的任务来完成相应的工作。

运算器和控制器是计算机的核心部件,通常将这两个部件集成在一块芯片上,称为中央处理器。

4. 输入设备

输入设备是计算机用来接收外界信息的装置。其功能就是将外界的信息转换为计算机所能识别的数据形式并传递给计算机内部。常见的输入设备有键盘、鼠标、扫描仪和数码相机等。

5. 输出设备

输出设备是将计算机处理的结果向外界展示的一种设备。其功能是将计算机内的数据或程序转换为人们所需要的形式,传递给计算机外界。常见的输出设备有显示器、打印机和绘图仪等。

CPU 与输入输出设备进行数据传送时,经常会使用 DMA,DMA 是一种完全由硬件完成 I/O 操作的方式。关于 DMA 的工作原理和具体实现方式请读者参考第 6 章。

6. 总线

总线(也叫系统总线)包括地址总线、数据总线和控制总线。CPU 通过系统总线与系统其他部件进行数据通信。地址总线是 CPU 输出地址信息所使用的总线。数据总线是用来实现 CPU、主存及 I/O 接口电路之间的数据交换。控制总线是 CPU 与其他部件之间传送控制信号的总线。

2.2 处理器的工作模式

早期的计算机如 IBM-PC 采用的处理器芯片是 8086/8088,这是一种 16 位或准 16 位的微处理器。它所支持的操作系统是单用户、单任务的系统,最大寻址空间为 1MB(2^{20})单元。但随着计算机技术的发展,CPU 的数据带宽和寻址能力迅速增加,如 80386/Pentium 系列的 CPU,其内部运算为 32 位,寻址能力为 4GB(2^{32})。并且集成了存储管理部件以及保护机构,可以支持多用户、多任务的操作系统。为了兼容 8086 处理器,自 80286 以后,处理器引入了处理器的工作模式的概念。这样可以较好地解决处理器兼容性问题。常见的处理器工作模式有实模式(real mode)、保护模式(protected mode)和虚拟 8086 模式(virtual 8086 mode)。

1. 实模式

实模式是 80286 以上的微处理器所采用的 8086 的工作模式。也就是说在实模式下,处理器的存储器寻址方式和 8086 是一样的,由段寄存器的内容乘以 16 作为段基址,再加上段内偏移形成最终的物理地址,此时 32 位地址线只使用了低 20 位。在实模式下,80386 处理器不能对内存进行分页管理,所以指令寻址的地址就是内存中实际的物理地址。实模式下的中断处理方式和 8086 处理器相同,也用中断向量表来定位中断服务程序地址。在中断向量表中,每 4 个字节组成一个中断向量,包括两个字节的段地址和两个字节的偏移地址。

DOS 操作系统就是运行在实模式下。计算机启动时,系统自动进入实模式,即使希

望系统运行在保护模式,处理器也需要经过实模式跳转到保护模式。在实模式下首先进行一些初始化的工作。

2. 保护模式

保护模式是 80386 以上处理器所使用的主要工作模式。保护模式是支持多任务的工作模式,它提供了内存分段分页管理和特权级保护等功能。80386 在实模式下可以访问 4GB 的物理存储空间。在这种工作模式下操作系统可以引入虚拟存储的机制,来扩充编程者所使用的地址空间。Windows 和 Linux 等操作系统都需要在保护模式下运行。

3. 虚拟 8086 模式

虚拟 8086 模式是一种既有保护功能又能执行 8086 代码的工作模式,又称“V86 模式”,是一种动态工作模式。可以通过指令与保护模式进行切换。它是一种特殊的保护模式,它同样支持任务切换、内存分页管理和特权级保护等功能。只是它的内存寻址方式和 8086 相同,寻址范围为 1MB 的空间。

三种工作模式的转换如图 2.3 所示。

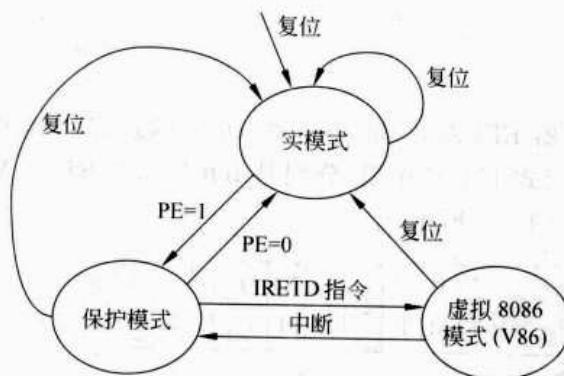


图 2.3 三种工作模式转换图

在实模式下可以通过指令切换到保护模式,图 2.3 中 PE 是保护模式允许位,是 80x86 控制寄存器 CR0(见 2.3.1 节)的一位。通过复位可以将机器从保护模式返回到实模式。保护模式和 V86 模式之间可以通过指令互相切换,但不能从实模式直接进入 V86 模式。

实模式和虚拟 8086 模式是为了和 8086 处理器兼容而设置的。在实模式下,80386 处理器就相当于一个快速的 8086 处理器。保护模式是 80386 处理器的主要工作模式。下面以 80x86 为例详细介绍保护模式的主要特性。

2.3 80x86 的保护模式

80386 有三种工作模式：实模式、保护模式和虚拟 8086 模式。尽管实模式下 80386 的功能要大大超过其先前的处理器(8086/8088、80186、80286)，但只有在保护方式下，80386 才能真正发挥更大的功能。本节介绍 80x86 保护模式下的相关特性。

在保护模式下，32 位地址线有效，可寻址范围为 4GB 的地址空间。存储器的分段机制和分页机制为存储器的共享及保护提供了支持，也为系统的虚拟存储管理功能提供了支持。硬件支持多任务，能够快速的切换任务以及对任务进行保护。4 个特权级以及完善的特权级检查机制保证了代码和数据的安全。

2.3.1 80x86 系统寄存器

为了协助处理器控制系统操作，80x86 提供了一个标志寄存器 EFLAGS 和几个控制寄存器和系统地址寄存器。标志寄存器用于指示微处理器的状态并控制它的操作。80x86 控制寄存器和系统地址寄存器用于控制系统工作方式，控制分段管理机制及分页管理机制的实现。

1. 标志寄存器

80386 的标志寄存器 EFLAGS 和 8086 的 16 位标志寄存器相比，增加了 4 个控制位，这 4 个控制位在实模式下不起作用，分别是 IOPL、NT、RF 和 VM。80386 标志寄存器的各个标志位分布如图 2.4 所示。

	V	R		N	IO	O	D	I	T	S	Z		A		P		C
	M	F		T	PL	F	F	F	F	F	F		F		F		F

图 2.4 80386 标志寄存器 EFLAGS

- CF 位。进位标志。
- PF 位。奇偶标志。
- AF 位。辅助进位。
- ZF 位。0 标志。
- SF 位。符号标志。
- TF 位。陷阱标志。
- IF 位。中段允许标志。
- DF 位。方向标志。
- OF 位。溢出标志。

- IOPL 位。I/O 特权级,指定了 I/O 指令的特权级。如果当前的特权级别在数值上小于或等于 IOPL,那么 I/O 指令可执行。否则,将发生一个保护性异常。
- NT 位。嵌套任务标志。控制着被中断任务和调用任务之间的连接关系。NT=0,用堆栈中保存的值恢复 EFLAGS、CS 和 EIP 从而实现中断返回;NT=1,则通过任务切换实现中断返回。
- RF 位。恢复标志。
- VM 位。虚拟 8086 模式。VM=0,处理器工作在保护模式下;VM=1,工作在 V8086 模式下。

2. 控制寄存器

80386 有 4 个 32 位的控制寄存器,分别为 CR0、CR1、CR2 和 CR3。CR1 被保留,在 80386 中不能使用 CR1,否则会引起无效指令操作异常。CR0 包含指示处理器工作方式的控制位,启用和禁止分页管理机制的控制位,以及控制浮点协处理器操作的控制位。CR2 及 CR3 由分页管理机制使用。CR0 中的第 5~30 位及 CR3 中的第 0~11 位是保留位,这些位在 80386 处理器中必须为 0,如图 2.5 所示。

Bit	31	30-12	11-5	4	3	2	1	0
CR0	PG	0000000000000000		ET	TS	EM	MP	PE
CR1	保留							
CR2	页故障线性地址							
CR3	页目录表物理地址			000000000000				

图 2.5 控制寄存器

CR0 含有系统的控制标志。其中包含如下标志位。

- PE: 保护模式允许位。如果设置了该位,处理器在保护模式下运行。机器复位后 PE=0。
- MP: 数学运算协处理器存在标志。该位为 1 时表示系统中存在一个数学运算协处理器。
- EM: 仿真控制位。表明是否需要仿真协处理器的功能。如果系统中存在一个软件模拟器执行数学运算,则此位应该置 1。这里应注意,在同一时刻只能使 MP 和 EM 中的某一位置 1。
- TS: 任务切换位。每当任务切换时处理器就会设置该位为 1。
- ET: 扩展类型位。该位指出了系统中所含有的协处理器类型。该位为 1 时表示系统有 80387 协处理器存在,该位为 0 时表示系统有 80287 协处理器存在。

- PG: 分页操作位。该位指示是否使用页表将线性地址转换成物理地址。表 2.1 为 PG 和 PE 位与处理器工作模式关系表。

表 2.1 PG 和 PE 位与处理器工作模式关系表

PG 位	PE 位	处理器工作方式	PG 位	PE 位	处理器工作方式
0	0	实模式	1	0	无效
0	1	保护模式,禁用分页机制	1	1	保护方式,启用分页机制

CR1 保留不用,CR2 和 CR3 用于分页机制。

CR2 用于发生页异常时报告出错信息。当发生页异常时,处理器把引起页异常的线性地址保存在 CR2 中。操作系统中的页异常处理程序可以检查 CR2 的内容,从而查出线性地址空间中的哪一页引起本次异常。

CR3 用于保存页目录表的起始物理地址。由于目录是页对齐的,所以仅高 20 位有效,低 12 位保留未用。

Pentium 处理器又增加了一个控制寄存器 CR4,并对 CR0 的相应位进行重新定义,请读者阅读相关资料。

3. 系统地址寄存器

全局描述符表 GDT、局部描述符表 LDT 和中断描述符表 IDT 等都是保护方式下非常重要的特殊段,它们包含了段机制所使用的重要数据。为了方便快速地定位这些段,处理器采用一些特殊的寄存器保存这些段的基地址和段界限。把这些特殊的寄存器称为系统地址寄存器。

- 全局描述符表寄存器 GDTR。用于存放全局描述符表 GDT 的 32 位线性基地址和 16 位表长度值。指令 LGDT 和 SGDT 分别用于加载和保存 GDTR 寄存器的内容。机器加电时基地址默认为 0,表长度值为 0xFFFF。
- 局部描述符表寄存器 LDTR。用于存放局部描述符表 LDT 的 32 位线性基地址、16 位表长度值和描述符属性值。
- 中断描述符表寄存器 IDTR。与全局描述符表寄存器 GDTR 类似,用于存放中断描述符表 IDT 的 32 位线性基地址和 16 位表长度值。指令 LIDT 和 SIDT 分别用于加载和保存 IDTR 寄存器的内容。机器加电时基地址默认为 0,表长度值为 0xFFFF。

另外一个重要的系统寄存器为任务状态段寄存器 TR。TR 用于存放当前任务 TSS 段的 16 位段选择符、32 位基地址、16 位段长度以及描述符的属性值。当执行任务切换时,处理器会将新任务的 TSS 的段选择符和段描述符自动加载到 TR 中。

2.3.2 存储管理机制

保护和地址变换是操作系统内存管理的两个关键部分。保护是为了防止一个任务访问另外一个任务或操作系统内核的内存区域。地址变换是指将操作系统中使用的逻辑地址转换为硬件识别的物理地址。计算机中的物理内存是字节的线性数组,每个字节具有唯一的一个物理地址。操作系统在给任务分配内存时的地址是由两部分组成的逻辑地址。逻辑地址不能直接映射到物理地址,必须通过一种地址转换机制来将逻辑地址转化为物理地址。每个任务的这种转化关系是不一样的,因此地址转换机制也为任务之间提供了保护。

80x86 使用分段和分页两种机制来实现地址转换(如图 2.6 所示)。分段机制是将逻辑地址转换为线性地址的过程。分页机制是将线性地址转换为物理地址的过程。

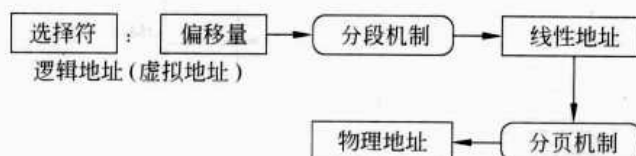


图 2.6 线性地址到物理地址的转换

在存储管理部分经常见到逻辑地址、虚拟地址、线性地址和物理地址这几个名词,下面简单的介绍这几个地址。

- 逻辑地址。当程序访问存储器时,如果不考虑操作方式以及物理硬件细节时,它所使用的地址就称为逻辑地址。每个逻辑地址由一个段(segment)和偏移量(offset)组成。
- 虚拟地址。在保护模式下,程序访问存储器所使用的逻辑地址称为虚拟地址。
- 线性地址。线性地址是经过分段机制将虚拟地址转换而成的,是线性地址空间的一个绝对地址。线性地址空间是一段连续的、不分段的、0~4GB 范围的地址空间。线性地址经过分页部件处理后可以转换成物理地址。在 80386 下线性地址是 32 位无符号整数,可以表示 4GB 的地址空间。
- 物理地址。是指物理内存空间上的一个实际地址。物理地址是逻辑地址转换的最终结果。

2.3.3 分段机制

分段机制是指由段描述符选择符从“段描述符表”中得到段基址,再与 32 位的偏移量相加形成一个中间地址(称为“线性地址”)的机制。

段描述符表中主要包括以下几项。

- 段基地址(Base address)。指定段在线性地址空间中的起始地址。
- 段界限(Limit)。段可以使用的最大偏移量,也是段的长度。
- 段属性(Attributes)。段的访问权限,如该段是否可读写及段的特权级等。

每一个段描述符项由 8 个字节组成,其中基地址 32 位,界限 20 位,属性 12 位。

段基址与段界限定义了段所能映射到线性地址空间的区域范围,如图 2.7 所示。段内 0 到 Limit 的地址范围对应着线性地址空间的 Base 到 Base+Limit 的区域。

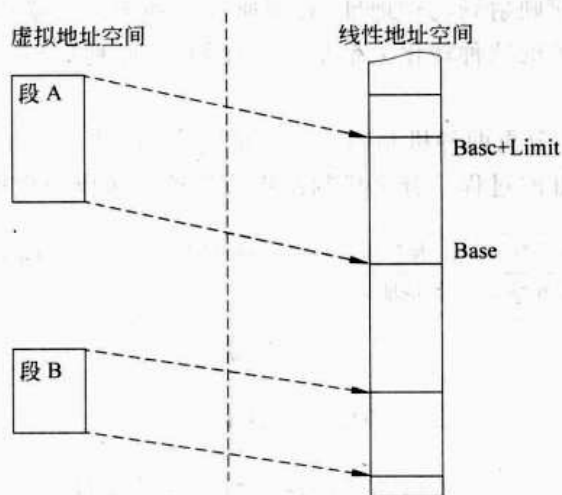


图 2.7 虚拟空间的段映射到线性地址空间

在保护模式下段寄存器中存放的就是段描述符表中的索引,因为这样可以知道段描述符在段描述符表中的位置,实际上也就是存放了虚拟地址“选择符:偏移量”中的选择符。选择符中存在三个域。第 0~1 位(RPL)表示请求特权级。第 2 位(TI)为选择域,决定从哪个描述符表中选择相应的段描述符。TI=0 表示描述符在 GDT 中, TI=1 表示描述符在 LDT 中。第 3~15 位为索引域,给出了描述符在 GDT 或 LDT 表中的索引项号。选择符的结构如图 2.8 所示。

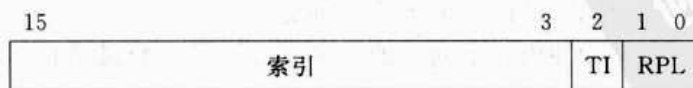


图 2.8 选择符结构

在保护模式下存在三种类型的描述符表:全局描述符表(Global Descriptor Table, GDT)、局部描述符表(Local Descriptor Table, LDT)和中断描述符表(Interrupt Descriptor Table, IDT)。并且有专门的寄存器用来存储描述符表的地址及描述符表的长度界限。图 2.9 是全局描述符表的结构。

地址转换过程如图 2.10 所示,根据 TI 标志位选择 GDT 或 LDT,根据索引值找到相应的段描述符,使用段基址与偏移量的和来计算线性地址空间的线性地址。

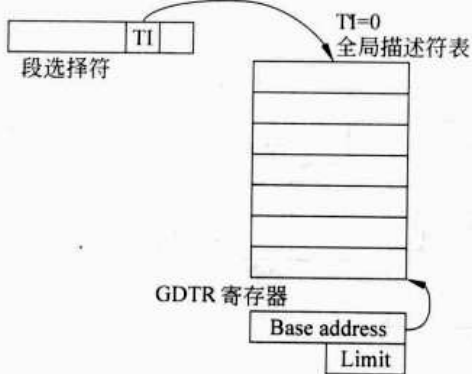


图 2.9 全局描述符表

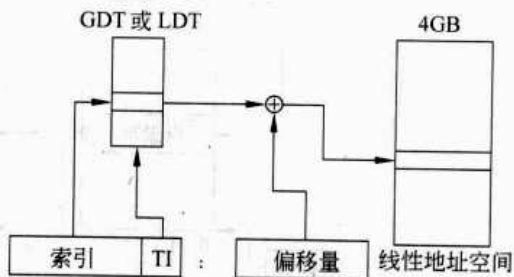


图 2.10 逻辑地址到线性地址的变换

Linux 支持了非常多的硬件平台,由于大部分硬件平台不支持分段机制,只有分页机制。因此 Linux 为了使其具有更好的移植性而没有使用分段机制。但是在 IA32 中不能禁止分段机制,因此在 Linux 下就将段基址定义为 0,段的界限定义为 4GB。

2.3.4 分页机制

分页机制是在分段机制之后进行的,是 80x86 内存管理的第二部分。分页机制是可选的,可通过设置 CR0 的最高位来允许和禁止分页机制。分页机制主要是将分段机制计算的线性地址转换为内存的物理地址。

80x86 通常使用 4KB 作为一个固定大小的页面(在新的 x86 体系中引入了 2MB 或者 4MB 的大页支持)。在线性地址空间和物理地址空间分别划分固定大小的区域,并建立对应关系,如图 2.11 所示。将线性地址空间划分为固定大小的片,称为页(page)。将物理地址空间也划分为同样大小的块,称为页面(page frame)。

线性地址空间的页与物理地址空间的页面的映射关系是由页表(page table)来描述的。4GB 的线性地址空间被划分为 2^{20} (1M) 个 4KB 大小的页。页面在物理地址空间的基地址一定是 4096 的倍数,低 12 位一定是 0。因此在页表项中可以用高 20 位存放页面的基地址,低 12 位用来存放页面的属性。页表中总共有 2^{20} (1M) 个页表项,每个页表项占用 32 位(4B)。因此需

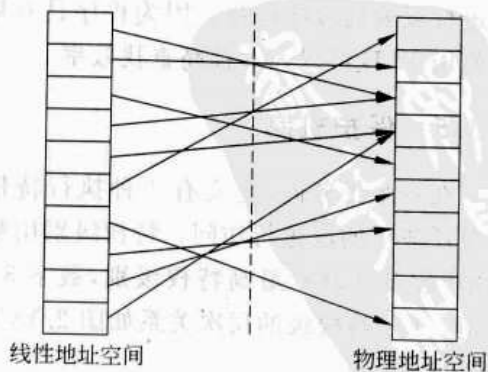


图 2.11 线性地址空间页与物理地址空间页面的对应关系图

要 4MB 的内存空间来存放页表,并且要求是连续的空间,显然这是很难满足的。因此采用两级页表来解决这个问题,同时采用两级页表也加快了页表的查找速度。

两级页表的结构如图 2.12 所示。可以通过线性地址很方便地寻找到物理地址。

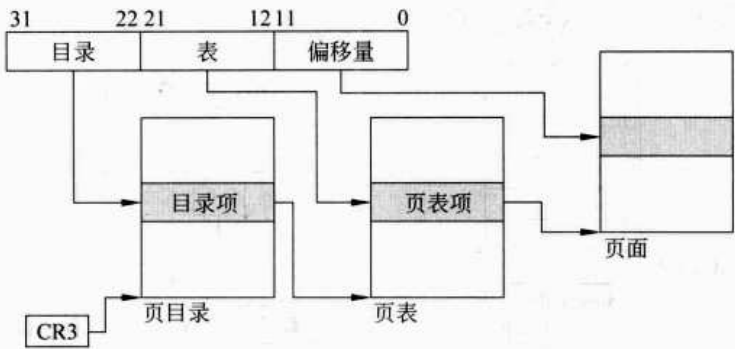


图 2.12 x86 两级页表结构

查找过程如下。

- (1) 从 CR3 中取出页目录的地址。
- (2) 根据线性地址的高 10 位,索引页目录项,找到第二级页表的地址。
- (3) 根据线性地址的中间 10 位,索引页表,找到页面的基地址。
- (4) 根据线性地址的低 12 位的偏移量,最终寻找到物理地址。

每次将 32 位线性地址转换为物理地址都需要进行两级页表的查找操作。另外新的 x86 处理器支持 PAE 扩展,如果打开 PAE 扩展则硬件将使用三级页表,在 PAE 扩展被打开以后,物理寻址范围将达到 64GB,Linux 对此已经有编译选项支持。为了更好地提高查找效率,x86 处理器提供一个 TLB(Translation Lookaside Buffer)来加快线性地址的转换。这样程序在使用一个线性地址时,可以先到 TLB 中查找转换地址,如果不存在则再进行页表的查找过程。因为程序具有局部性特性,因此使用 TLB 可以大大提高查找效率。

2.3.5 保护机制

在一个任务内,定义有 4 种执行特权级别,用于限制对任务中的段进行访问。特权级别用数字 0、1、2 和 3 表示,数字 0 表示最高特权级别,数字 3 表示最低特权级别。4 种特权级的层次关系如图 2.13 所示(图中的数字为特权级)。

在任何时候,一个任务总是在 4 个特权级之一下运



图 2.13 处理器特权级示意图

行,任务在特定时刻的特权级称为当前特权级(Current Privilege Level,CPL),即当前运行程序的特权级。每当一个程序试图访问一个段时,就把 CPL 与要访问的段的特权级进行比较,来决定是否允许这一访问。对于给定 CPL 执行的程序,允许访问同一级别或更低级别的数据段。

2.4 中断陷入与异常处理

2.4.1 中断与异常

在程序运行过程中,系统外部、内部或程序本身出现紧急事件,处理器必须立即中止正在运行的程序,调用相应的事件处理程序来处理发生的事件,再恢复原来的程序运行。这个过程就是中断。其中相应的事件处理程序就是后面将要提到的中断处理程序。

中断使得操作系统可以捕获用户程序发出的系统功能调用,及时处理设备的中断请求,防止用户程序中的破坏性活动等。

8086/8088 把中断分为内部中断和外部中断两大类。为了支持多任务和虚拟存储器等功能,80386 把外部中断称为“中断”,把内部中断称为“异常”。80386 最多处理 256 种中断或异常。

图 2.14 是中断和异常的分类。

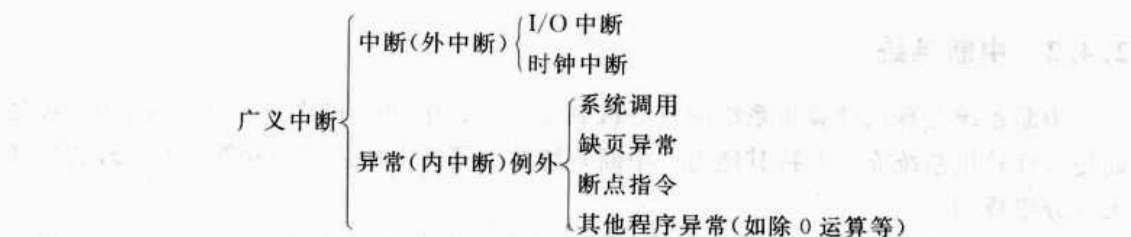


图 2.14 中断分类

图 2.14 中的中断分类方法是对广义中断进行简单的划分,在新的 x86 体系中还定义了不可屏蔽中断,读者可阅读相关手册进一步了解此内容。

中断与异常的区别为中断与正在执行的指令无关,而异常与正在执行的指令有关;中断是由硬件信号引发的,分为可屏蔽和不可屏蔽中断,异常是由指令执行引发的,比如除零异常。80x86 处理器发布了大约 20 种不同的异常,对于某些异常,CPU 会在执行异常处理程序之前产生硬件出错码,并压入内核态堆栈。

80386 最多处理 256 种中断或异常。为了使处理器容易识别每个中断源,所以给每个中断一个编号,也就是标识号,也称为向量号,从 0~255。因此这个 8 位的无符号整数就叫做中断向量。Linux 对这 256 个中断向量的分配如表 2.2 所示。

表 2.2 中断向量分配表

向量范围	用途
0~19(0x0~0x13)	非屏蔽中断和异常
20~31(0x14~0x1f)	Intel 预留
32~127(0x20~0x7f)	外部中断 (IRQs)
128(0x80)	系统调用异常
129~238(0x81~0xee)	外部中断
239(0xef)	本地 APIC 时钟中断
240~250(0xf0~0xfa)	Linux 预留
251~255(0xfb~0xff)	处理器间中断 (interprocessor interrupts, IPI)

0x80 号中断是一个系统调用异常。因此操作系统在发生系统调用时实际上是通过调用 `int 0x80` 指令来完成的。指令 `int 0x80` 会跳转到系统调用在内核中定义的入口地址。这个地址是在内核启动过程中,由内核初始化程序来指定的,这个地址被定义为 `system_call`。在内核中还存在一张系统调用表 `sys_call_table`,每个系统调用都是通过查找这张系统调用表来查找相应系统调用的偏移,最终定位到真正的内核调用地址。

为了提高系统性能,新型的 CPU 开始支持新的系统调用指令 `sysenter/sysexit`。

2.4.2 中断系统

中断系统是现代计算机系统的核心机制之一。是由硬件和软件相互配合、相互渗透而使得计算机系统充分发挥其能力。中断系统是由硬件中断装置和软件中断处理程序两大部分组成的。

Intel x86 计算机使用两片中断控制器 8259A 来接收外部中断源。每个 8259A 可以管理 8 个中断源,两片级联总共可以管理 15 个外部中断源。两片 8259A 级联的结构图如图 2.15 所示。

中断控制器负责将硬件的中断信号转换为中断向量,并引发 CPU 中断。与中断控制器相连的每条线叫做中断线。要使用中断线,需要进行中断线的申请,因此称为 IRQ (Interrupt Requirement)。并不是每个设备都可以向中断线上发送中断信号,只有对某一条中断线拥有了控制权后,才可以向这条中断线发送中断信号。由于中断线总共只有 15 根,而外部设备是多种多样的,因此中断线的资源是有限的,只有当设备需要中断时才能申请占有一根中断线。当然也可以采用共享的方式来申请中断线。

在实地址模式下,中断向量存储在内存的起始部分,从 0 开始形成一个中断向量表。中断向量表中每一项由 4 个字节组成,包括两个字节的段地址和两个字节的偏移量。这

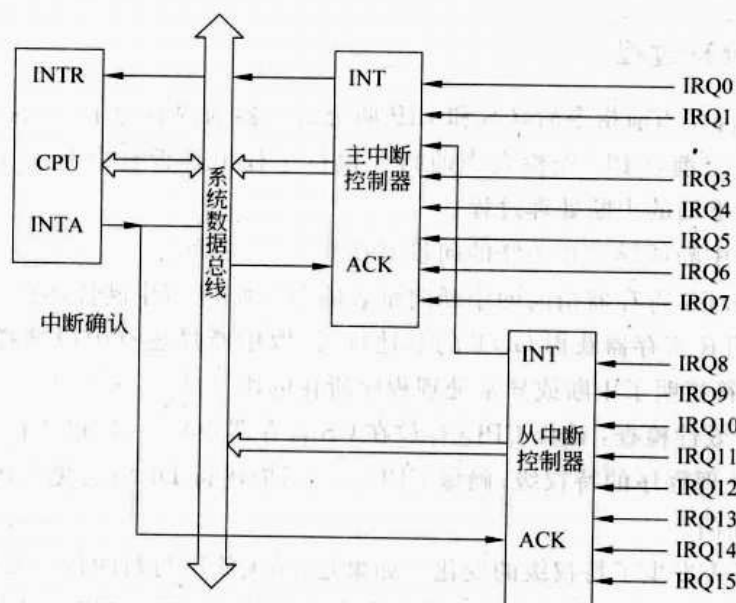


图 2.15 级联的 8259A 的结构图

样构成的是一个地址,也就是相应的中断处理程序的首地址。但是在保护模式下需要对中断处理机制重新设置,此时中断向量表的每一表项由 8 个字节组成,每一个表项叫做一个门描述符(gate descriptor)。此时中断向量表叫做门描述符表。门描述符的一般格式如图 2.16 所示。

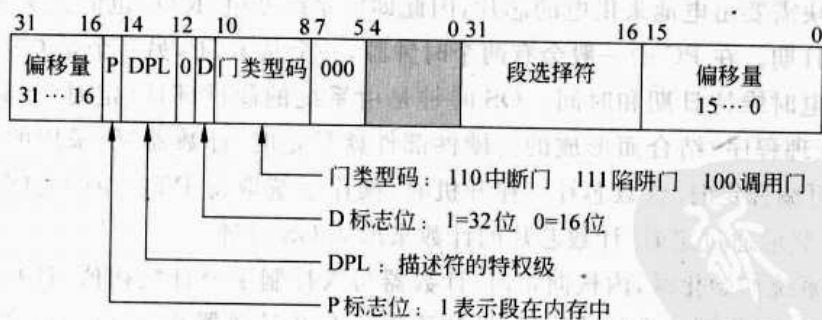


图 2.16 门描述符一般格式

在保护模式下,中段描述符表的位置可以放在内存中的任意位置。CPU 提供了一个寄存器 IDTR 来存放中断描述符表在内存的起始地址。IDTR 是一个 48 位寄存器,低 16 位是中断描述符表的大小,高 32 位是中断描述符表在内存中的基地址。

2.4.3 中断处理过程

当 CPU 执行完当前指令后,CS 和 EIP 两个寄存器的内容为下一条指令的地址。在执行下一条指令之前,CPU 先检查当前指令执行过程中是否有中断或异常发生。如果有,CPU 则执行下面的中断处理过程。

- (1) 确定与中断或异常相关联的向量 i (在 $0 \sim 255$ 之间)。
- (2) 读取 IDTR 寄存器指向的中断向量表的第 i 项,获得中断描述符。
- (3) 从 GDTR 寄存器获得 GDT 的基地址,读取中断描述符中段选择符所标识的段描述符,该描述符指明了中断或异常处理程序所在的段基址。
- (4) 进行有效性检查:确保 CPL(存放在 CS 寄存器中) $\geq$ 段描述符 DPL;当前特权级不高于中断处理程序的特权级;确保 CPL $\geq$ 门描述符 DPL;避免应用程序访问特殊的陷阱门或中断门。
- (5) 检查是否发生了特权级的变化。如果是,必须使用与新的特权级相关的栈。
- (6) 硬件压栈,保存上下文。如果异常产生了硬件出错码,则将它保存在栈中。
- (7) 通过中断描述符中的段内偏移,找到中断或异常处理程序入口,执行其第一条指令。

2.4.4 时钟中断

在计算机中都包含了一个叫做实时时钟(Real Time Clock,RTC)的时钟芯片,RTC 是主板上一块需要由电池来供电的芯片,因此即使系统断电,RTC 也能正常工作,维持系统的时间和日期。在 PC 中一般会有两个时钟源,一个是 RTC,另一个是 OS 时钟。RTC 是在系统断电时维持日期和时间。OS 时钟是由系统的硬件部件(定时/计数器)和软件(时钟中断处理程序)结合而形成的。硬件部件就是定时/计数器,所采用的芯片一般是 8253/8254 可编程定时/计数芯片。在开机时,操作系统取得 RTC 中的时间数据来初始化 OS 时钟,然后通过定时/计数芯片的计数来形成 OS 时钟。

在操作系统初始化时,内核向定时/计数器写入控制字及计数初值,计数器便从计数初值开始计数,每收到一次脉冲信号,计数器就减 1,当计数器减至 0 时,就会输出一个脉冲。随后便从计数初值开始重复另一次计数过程,周期性产生输出脉冲。

定时/计时器对从晶振输入脉冲进行计数(分频),产生三个输出脉冲 Out0、Out1 和 Out2,三个输出脉冲分别输出到中断控制器、DRAM 和扬声器。定时/计数器产生的输出脉冲接到中断控制器上,定期产生中断信号。这个中断就是时钟中断。时钟中断的产生过程如图 2.17 所示。从本质上说,时钟中断只是一个周期性的信号,完全是硬件行为,该信号触发 CPU 去执行一个中断服务程序。

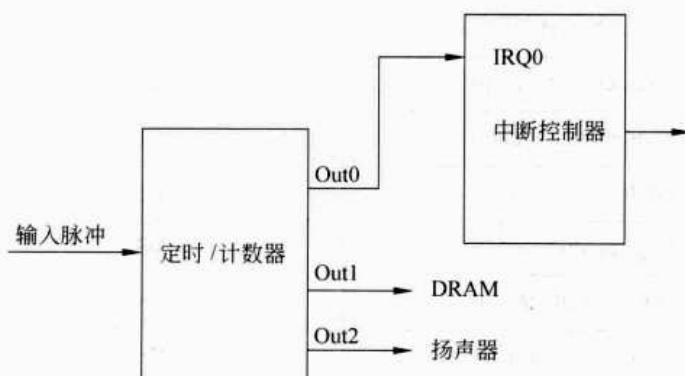


图 2.17 8253 和 8259A 的物理连接方式

时钟中断对于操作系统来说是一个非常重要的中断,因为整个操作系统的活动都受到它的激励。可以说,时钟中断是整个操作系统的脉搏。

2.5 任务切换

80x86 支持多任务操作,这主要是通过一种任务切换机制来实现的。这种机制需要硬件及软件互相配合来完成,其中一个重要的部分是任务状态段 TSS,每个任务有一个任务状态段 TSS,用于保存任务的有关信息。任务是处理器可以分配、调度和挂起的一个工作单元。

任务状态段 TSS 的基本格式如图 2.18 所示。TSS 中包括保存 EAX、ECX、EDX、EBX、ESP、EBP、ESI 和 EDI 寄存器的内容,以及保存 ES、CS、SS、DS、FS 和 GS 段寄存器的内容,还包括存放任务的 LDT 段的选择符,CR3 内容,特权级 0、1 和 2 的堆栈指针字段,调试陷阱 T 标志字段等。EFLAGS 字段用来保存切换之前的 EFLAGS 值。EIP 字段保存切换之前的 EIP 寄存器的内容,返回链接字段保存前一个任务 TSS 段选择符。

任务状态段是保存一个任务重要信息的特殊段,是使用段描述符来定义的。图 2.19 是 TSS 描述符的格式。其中字段 TYPE 的 B 用来标明任务是否处于忙状态,忙状态的任务是指正在运行的任务或被挂起的任务。

在任务切换过程中会使用任务状态段寄存器 TR。TR 包括 16 位当前任务的任务状态段描述符的选择子,TR 的不可见的高速缓冲寄存器部分包含当前任务状态段的整个描述符。

TSS 在任务切换过程中起着非常重要作用,通过它实现任务的挂起和恢复。在任务切换过程中,首先处理器各寄存器的当前值被自动保存到 TR 所指定的 TSS 中,然后下一任务的 TSS 的选择子被装入 TR,最后从 TR 所指定的 TSS 中取出各寄存器的值送到处理器的各寄存器中。通过在 TSS 中保存任务现场各寄存器状态的完整映象,来实现任务的切换。

位图偏移量	0000000000000000	T	64H
0000000000000000	LDT		60H
0000000000000000	CS		5CH
0000000000000000	FS		58H
0000000000000000	DS		54H
0000000000000000	SS		50H
0000000000000000	CS		4CH
0000000000000000	ES		48H
EDI			44H
ESI			40H
EBP			3CH
ESP			38H
EBX			34H
EDX			30H
ECX			2CH
EAX			28H
EFLACS			24H
EIP			20H
CR3			1CH
0000000000000000	SS2		18H
ESP2			14H
0000000000000000	SS1		10H
ESP1			0CH
0000000000000000	SS0		8
ESP0			4
0000000000000000	返回链接		0
Bit31~Bit16	Bit15~Bit1	Bit0	Offset

图 2.18 任务状态段 TSS

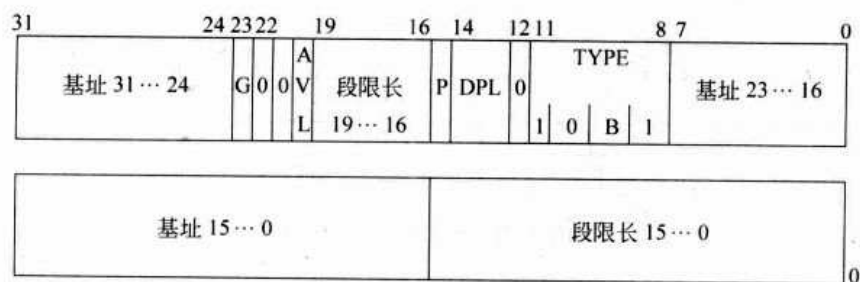


图 2.19 TSS 段描述符格式

80x86 对任务状态段及任务切换过程也提供了保护机制。从较高特权级的任务可以顺利地切换到较低特权级的任务。从较低特权级的任务切换到高特权级的任务就需要通过任务门来实现。图 2.20 给出了任务门描述符的一般格式。

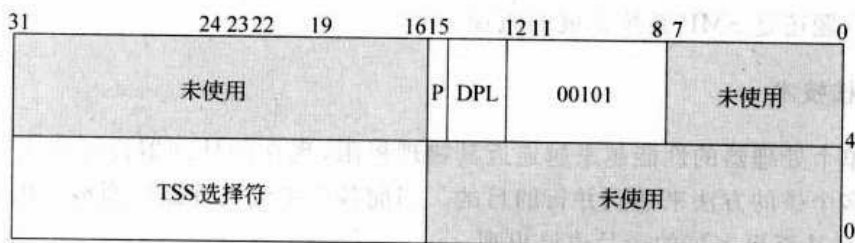


图 2.20 任务门描述符格式

任务门描述符中的 TSS 选择符字段指向 GDT 中的一个 TSS 段描述符。任务门描述符中的 DPL 用于在任务切换过程中对 TSS 段的访问控制,如果程序的 CPL 及指向任务门的门选择符的 RPL 值小于或等于任务门的描述符中的 DPL,则程序可以通过任务门调用或跳转到一个任务。

在高版本的 Linux 内核中,每个 CPU 仅使用了一个 TSS 来完成任务切换,为了简化设计,使用软件的方法来切换任务,也就是将 CR3 的值设成不同的值来切换,在第 3 章中会有详细的介绍。

2.6 SMP 及多核技术

1. SMP 体系

SMP(Symmetrical Multi-Processing, 对称多处理)技术是指在一台计算机上存在多个处理器,每个处理器都含有它自己的控制单元、算术逻辑单元和寄存器;每个处理器共享主存和 I/O 设备。图 2.21 是 SMP 的体系结构图。

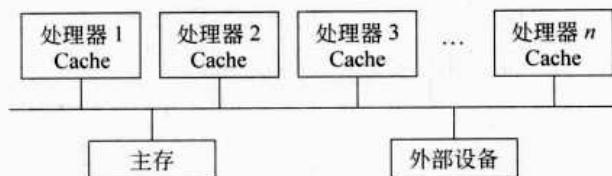


图 2.21 SMP 体系结构图

在 SMP 架构中,所有处理器在运行时都是对称的,没有主次之分。多个处理器运行操作系统的单一复本,并共享内存和计算机的其他资源。当一个任务可分成几个独立部分时,它们可以并行的在多个处理器上运行,这样比单处理机器有更好的性能。用户如果希望提高系统运算能力,可以通过增加处理器的个数来达到目的。

SMP 技术有效地解决了负载均衡的问题,但关于内存争用、内存同步和内存总线带宽争用等问题还是 SMP 系统扩展的瓶颈。

2. 多核技术

随着单个处理器的性能越来越逼近其物理极限,现在的处理器设计方向渐渐采用单片上集成多个核的方法来达到并行的目的。当前各个主流处理器厂商都推出了自己的多核产品,8 核甚至更多核的产品也已出现。

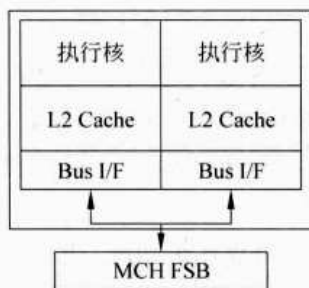


图 2.22 多核处理器结构

多核技术是指在单个芯片上实现两个或多个“执行核”。目前,两大台式机处理器的制造厂商 Intel 和 AMD 都推出了自己的双核处理器。多核处理器从根本上说是多个物理处理器协同工作,但是也存在着并行程度的问题。由于不同厂家提出的处理器体系结构的方案不同,从而导致了多核处理器的并行度也存在着不同。图 2.22 是 Intel 公司的某款早期多核产品的结构图。可以看出 L2 Cache 不是共享的,每个核分别存在一个 L2 Cache。两个核共享内存控制器的前端总线(MCH FSB)。

多核处理器的出现面临的最大问题就是软件支持问题。多核处理器与多处理器还存在很多不同,不能将以前在多处理器上的并行计算技术直接应用到多核上。多核处理器在性能和吞吐量上存在很大优势。与多处理器相比,多核处理器核之间的通信延迟会有量级的减少,核之间的带宽也有量级的增加。多核处理器的高速 Cache 层次比较复杂,其中有每个核私有的 L1 Cache,有多个核共享或私有的 L2 Cache,甚至更多的核共享的 L3 Cache。这就导致不同的核访问不同位置 Cache 的速度和延迟不同。因此使得一个 CPU 里的各个核很好的工作变得异常困难。多核还存在一个资源冲突的问题,多核同时运行

期间很多资源都是共享的,比如高速缓存、存储体和 BUS 等,这就需要改进操作系统和用户调度,来加以缓解。

多处理器及多核技术使得计算机系统在性能得到提高,但同时系统所付出的代价是操作系统内核复杂度和协同开销的增加。

Linux 刚出现时是单 CPU 的操作系统,在每一个特定的时刻只有一个进程在运行。而基于 SMP 结构以及多核结构的系统,在同一时刻可以运行多个进程,是真正的并行处理。因此对操作系统提出了新的要求。

在对称多处理器 SMP 系统中存在着多个 CPU,每个 CPU 处于同等地位,每个 CPU 都有自己的 APIC(高级可编程中断控制器)和 I/O APIC(处理 I/O 设备引起的中断)。CPU 之间通过彼此发送中断来完成之间的通信,并且通过给中断附加动作,不同的 CPU 可以在某种程度上彼此控制。这些需要软件来协助实现。

为了使每个处理器能同时执行系统的内核代码,内核必须是可重入的。进程可能在多个处理器上并发执行,并争抢资源,因此在设计操作系统时必须考虑冲突问题。

SMP 及多核结构都是共享主存的,因此存储器管理模块必须可以处理在单处理器机器上发现的所有问题。另外,还存在一致性问题,包括高速缓存一致性和存储一致性。高速缓存一致性指私有高速缓存中共享数据的副本数据和共享存储器中共享数据之间的一致性,存储一致性是指多个处理器程序的执行次序与共享存储器中共享数据存取次序之间的一致性。高速缓存与内存的数据一致性问题几乎都是由硬件完成的,但页面映射目录和页面映射表的缓冲存储(TLB)的一致性,要通过软件来辅助实现。

2.7 Linux 系统引导过程概述

本节主要分析从机器上电到 Linux 系统被引导起来的整个过程。在用户打开 PC 的电源后,首先是 BIOS 开机自检,然后按照 BIOS 中设置的启动设备(通常是硬盘)来启动引导程序,启动设备上安装的引导程序 LILO 或 GRUB 开始引导 Linux。Linux 首先进行内核的引导,此过程初始化相关的硬件设置,为系统运行做准备。接下来执行 init 程序,init 启动 mingetty 后,打开终端供用户登录系统,然后用户登录成功后进入了 Shell,这样便完成了从开机到登录的整个启动过程,如图 2.23 所示。

在系统引导的过程中,与系统引导相关的代码有 bootsect. S、setup. S、head. S 及 init 目录下的 main. c 文件等。读者在阅读这部分代码时需要熟悉相关汇编语言知识。下面简单介绍一下这些引导过程所完成的工作。

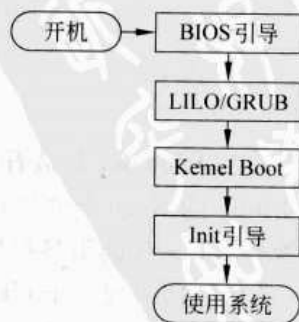


图 2.23 Linux 引导过程

2.7.1 BIOS 引导

当 PC 加电启动后, 80x86 结构的 CPU 将自动进入实模式(前面已经介绍), 此时 CS:IP 的值为 FFFF0H, 因此从地址 0XFFFF0 开始自动执行程序代码, FFFF0H 处的指令一般是一个 JMP 指令, 跳转地址通常是 BIOS 的入口地址。此时 BIOS 中的程序将开始运行, 执行一些系统检测, 如内存测试等, 并对本地设备进行枚举和初始化。接下来搜索 BIOS 中设置的处于活动状态并且可以引导的启动设备。通常 Linux 都是从硬盘引导的, 硬盘的引导扇区在每个分区里都存在, 但是主引导扇区却在硬盘的第一个物理扇区。它由两个部分组成, 即主引导记录(Master Boot Record, MBR)和硬盘分区表(DPT)。在硬盘的主引导扇区中包含了主引导记录(MBR)的引导加载程序。此时, BIOS 将控制权交给 MBR。

2.7.2 LILO/GRUB 引导

主引导扇区是一个 512B 大小的扇区, 包括 MBR 程序代码, 分区表 DPT 以及结束标志, 如图 2.24 所示。在总共 512B 的主引导扇区里 MBR 程序代码占 446B(偏移 0~1BDH), DPT 占 64B(偏移 1BEH~1FDH), 最后两个字节 AA55(偏移 1FEH~1FFH)是分区的结束标志。

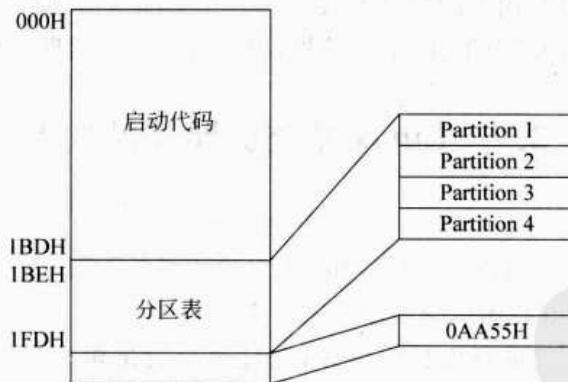


图 2.24 主引导扇区结构

LILO 和 GRUB 被看作是操作系统的 Boot Loader。它的责任是在启动计算机时装入 Linux 的 Kernel, 并将计算机的控制权交给 Kernel, 然后 Kernel 再进一步初始化剩余的操作。Linux 的引导程序有很多种, 但是最常见是 LILO 和 GRUB。

LILO 的出现早于 GRUB, 在 GRUB 出现之前, LILO 可谓 Linux 引导程序的最佳选择。LILO 是一个灵活多用的引导程序。它不依赖于某一特定文件系统, 能够从软盘和硬盘引导 Linux 内核映像, 它也可以引导其他的操作系统。

GRUB 可以引导的操作系统有 Linux、OS/2、Windows 系列、BeOS、Solaris、FreeBSD 和 NetBSD 等。它的优势在于支持大硬盘、支持开机画面、支持菜单式选择,使用非常方便。较新发行的各版本 Linux 大多采用 GRUB 作为默认的引导程序。但是目前仍然有很多人喜欢使用 LILO。

LILO 和 GRUB 可以存放在任何一个分区的引导扇区中,MBR 引导程序会寻找到引导扇区中的 LILO 或 GRUB 引导程序。并将控制权交给 LILO 或 GRUB 引导程序。

实际上 Linux 自己有一个被称为自举引导的程序 (bootsect-loader), 对应源程序为 arch/i386/boot/bootsect.S, 它为实模式的汇编程序。但是由于大家都需要安装多个操作系统, 因此目前使用最多还是 LILO 和 GRUB。为了方便分析启动程序的工作过程, 下面以 arch/i386/boot/bootsect.S 源代码为例进行简单介绍。

```
SETUPSECTS = 4                //安装时的缺省值
BOOTSEG     = 0x07C0          //引导时的原始地址
INITSEG     = DEF_INITSEG     //移动引导程序
SETUPSEG    = DEF_SETUPSEG    //从这里开始安装
SYSSEG      = DEF_SYSSEG      //系统装载
SYSSIZE     = DEF_SYSSIZE     //系统大小
```

在 include/asm-i386/boot.h 中一些宏定义如下。

```
#define DEF_INITSEG    0x9000
#define DEF_SYSSEG     0x1000
#define DEF_SETUPSEG   0x9020
#define DEF_SYSSIZE    0x7F00
```

bootsect.S 在内核编译的时候生成 bootsect 模块, 在整个内核镜像二进制文件的最前面。当计算机启动的时候先进入实模式, 首先执行位于 0xffff0 这个地址的程序代码, 这个地址实际上是映射到计算机 ROM BIOS 地址, 也就是说首先执行 BIOS 的代码, 执行某些检测, 在地址 0x0000 位置设置中断向量表, 并把第一个扇区读入到内存绝对地址 0x7c00 这个位置。bootsect 的代码就位于第一扇区, 所以跳到这个位置执行。bootsect 将自己移动到 0x90000, 并把 setup 模块搬运到 0x90200 这个位置。bootsect 执行完后, 执行 setup 模块。在 setup 执行过程中, 完成必要的设置, 并且把 system 模块移动到 0x10000 这个位置。当 system 前面的 head 部分执行完毕后, 然后把 system 搬运到 0x0000 的地址, 并跳转到那里执行。当 setup 模块将 system 搬运到 0x0000 的地址以后, system 模块中代码的地址就等于实际的物理地址, 这样便于对内核代码的操作。之所以不在开始时直接把 system 模块搬运到 0x0000 的位置是因为, 当 setup 模块没有执行完毕的时候, 0x0000 这个位置还存放着 BIOS 的信息和一些中断向量表。如果开始就搬运就会把这些内容覆盖掉而出错。

setup.S 的主要功能就是将系统参数(包括内存、磁盘等,由 BIOS 返回)复制到 0x90000~0x901FF 内存中。这个位置原本是 bootsect.S 被存放的位置,这时它将被系统参数覆盖。以后这些参数将由保护模式下的代码来读取。除此之外,setup.S 还将 video.S 中的代码包含进来,检测和设置显示器和显示模式。最后,setup.S 将系统转换到保护模式,并跳转到 0x100000 的内核引导代码,Bootloader 过程结束。

2.7.3 Kernel Boot

head.S 含有 32 位的 Startup 代码。Startup 的入口点在绝对地址的 0x100000 处。当 Bootloader 跳转到 0x100000 时,此处代码开始执行。这里主要是完成一些初始化的工作,包括清除 BSS、复制启动参数、初始化寄存器、初始化 GDT 和 IDT 等。然后跳转到 start_kernel()去执行。

start_kernel 中调用了一系列初始化函数,以完成 kernel 本身的设置。start_kernel 是 init/main.c 中的 asmlinkage 函数,主要完成一些相关的初始化工作:设置与体系结构相关的环境(setup_arch()),进程调度器初始化(sched_init()),控制台初始化(console_init()),系统 IRQ 初始化(init_IRQ()),内存初始化(mem_init())等。这些初始化工作都是非常重要的,也是必需的,建议读者在学习后面章节时,对相关的初始化函数进行简单的分析。start_kernel()结束后,基本 Kernel 运行环境已经建立起来了。在 start_kernel()函数的最后调用了 rest_init()函数。

rest_init()函数代码如下(在/init/main.c 中)。

```
416 static void noinline rest_init(void)
417     __releases(kernel_lock)
418 {
419     kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND);
420     numa_default_policy();
421     unlock_kernel();
422
423
424
425
426
427     preempt_enable_no_resched();
428     schedule();
429     preempt_disable();
430
431
432     cpu_idle();
433 }
```

在这段代码入口处,可以看到程序建立了 init 内核线程。最后调用了 `cpu_idle()` 函数。可以这样理解: `start_kernel()` 最后 clone 出一个新的进程(也就是 init 进程),然后原来的进程就去执行 `cpu_idle()` 函数了(也就是变成 idle 进程),当发生一次进程调度后, init 进程被调度运行。

2.7.4 Init 引导

核心线程 `init()` 主要是进行一些外设初始化的工作,包括 SMP 的初始化,以及调用 `do_basic_setup()` 完成外设及其驱动程序的加载和初始化。当 `do_basic_setup()` 函数返回 `init()`, `init()` 又打开了 `/dev/console` 设备,重定向输入输出文件到控制台。最后通过调用 `kernel_execve` 加载执行 init 程序。

2.8 本章小结

本章主要介绍计算机系统的硬件基础知识,介绍了硬件系统的基本组成。计算机硬件主要包括运算器、控制器、存储器、输入设备和输出设备。处理器存在三种工作模式(实模式、保护模式和虚拟 8086 模式)。保护模式是 80386 以上处理器所使用的主要工作模式,是支持多任务的工作模式,它提供了内存分段分页管理和特权级保护等功能。80386 在实模式下可以访问 4GB 的物理存储空间,段的最大长度在启动分页功能时是 4GB,不启动分页功能时是 1MB。在这种工作模式下操作系统可以引入虚拟存储的机制,来扩充编程者所使用的地址空间。计算机硬件提供了任务间的保护及同一任务内的保护机制。中断机制也是计算机系统的核心机制,中断系统是由硬件中断装置和软件中断处理程序两部分组成的。时钟中断对于操作系统来说是一个非常重要的中断,整个操作系统的活动都受到它的激励,可以说,时钟中断是整个操作系统的脉搏。本章还介绍了 Firmware 及系统引导的原理、SMP 技术及多核技术。最后对 Linux 系统引导过程进行了简单介绍。

思考题

1. 在 32 位系统下,使用什么方法可以使系统支持 4 GB 以上物理内存?
2. 64 位 x86 处理器下虚拟存储管理的工作机制与 32 位有哪些不同?
3. 在内核引导过程中,会初始化 RAM 磁盘(`initrd`)。这里 `initrd` 的主要作用是什么? `initrd` 是否是必需的? 为什么?

进一步阅读

1. Intel 64 及 IA-32 构架软件开发者手册:

<http://www.intel.com/products/processor/manuals/index.htm> 下载 PDF 文档

2. Intel 多核方面的更多资料, 请参考以下网址内容:

<http://www.intel.com/multi-core/>

第3章 进程管理

进程是多任务并发的基本概念,是现代操作系统最重要的思想之一。本章详细介绍进程的原理及其在 Linux 内核里的实现,包括基本数据结构,进程的状态、创建和结束。

在并行处理中,进程调度是影响系统性能的最重要因素之一,本章将详细介绍进程调度的基本原理和实现机制。

运行在一个计算机里的多个进程之间很少是完全独立的,它们要共享资源,分工协作,相互通信,所以进程的并发控制与同步机制是操作系统不可或缺的。本章也将详细介绍进程的各种同步原语、通信机制及其在 Linux 内核里的实现。

3.1 进程的原理和实现

早期的计算机采用单任务处理方式,操作员向计算机提交作业,计算机串行执行作业。在单道程序工作环境中,可以把一个“程序”理解为“一个在时间上按次序串行的操作序列”。

一切串行执行的程序都具有下列特性。

(1) 顺序性。

(2) 资源独占。

(3) 结果的无关性。

串行执行作业缺点是计算机的资源不能够得到充分的利用。比如一个作业需要做磁盘输入输出,由于磁盘比较慢,CPU 只能不断去查询是否磁盘已经完成输入输出,这样就浪费了 CPU 的资源。

由于作业具有一定的独立性,相互之间的依赖性较小,一定程度上可以并行处理,所以就有多任务处理方式。当一个任务需要做磁盘输入输出时,CPU 向磁盘提交输入输出请求后,就立即调度运行其他任务。当磁盘输入输出完成后,CPU 再恢复运行前一个任务。这样,CPU 资源得到更充分的利用。在计算机价格昂贵的时候,多任务处理方式大大提高了计算机的利用率和性价比。

无论是操作系统自身的程序还是用户程序,通常总是存在一些相对独立、但又能并发执行的程序段。由于这些程序段可以被多个用户作业调用,因此可在同一时间间隔内发生。这样一来,某个程序段可能对应多个“计算”,于是程序与“计算”已不具有——对应关系了。这些“并发程序”就构成了一个“并发环境”。在多道程序环境下,程序不能独立运

行。作为资源分配和独立运行的基本单位是进程。操作系统所有的特征都是基于进程而体现的。

进程是操作系统中的可执行程序的一次执行。例如,如果 100 个用户在同一台机器同时启动命令 `awk`,则有 100 个 `awk` 进程同时运行。进程有自己的生命周期,包括创建,完成一定任务后就被撤销。

进程与程序的区别和相互关系如下。

(1) 动态性和静态性。

(2) 从结构上看每个进程的实体都是由程序段和相应的数据段两部分构成的,这一特征与程序的含义相近。

(3) 一个进程可以涉及到一个或几个程序的执行;反之一个程序可以对应多个进程,即同一程序段可在不同数据集合上运行,可构成不同的进程。

(4) 并发性。

(5) 进程具有创建其他进程的功能。

(6) 操作系统中的每一个程序都是在一个进程现场中运行的。

在 Linux 中,用户进程是用 `fork` 的系统调用创建的。一个进程必须有系统资源,比如内存和 CPU。在多任务系统中,操作系统分配并调度这些资源,一方面,进程无须关心资源管理的细节,使进程集中在自己的事务处理上,另一方面,多个进程共享系统资源,使资源利用趋于最大化。实际上,操作系统是把进程看作资源分配的一个个体,大多数资源是基于进程分配的。其中,在大多数机器上,内存是采用虚拟内存的方式组织的。每一个进程有一个自己的进程地址空间。

每一个进程只能有一个父进程,但是一个父进程可以有多个子进程。当进程创建时,操作系统会给予进程创建新的地址空间,并把父进程的地址空间的映射复制到子进程的地址空间去。父子进程共享只读数据和代码段,但是堆栈和堆是分离的。

3.1.1 进程控制块

进程描述程序的一次执行,它存在于内存中。系统的资源分配、调度是以进程为单位的。进程的资源、运行状态等由内核管理,在内核中需要一个数据结构来描述。通常这种数据结构称为进程控制块(Process Control Block,PCB)。一个进程可以说是由进程控制块、代码和数据组成(数据和代码是由程序提供)。

进程控制块描述的内容非常多,所有和进程相关的信息都由它记录。不同的操作系统中进程控制块的内容也不相同,但总的来说主要有如下几类。

1. 进程标识符

进程标识符用于标识一个进程。标识符包括进程的 `id`、进程的名字、进程的家族关系

和拥有此进程的用户标识等。

2. 进程上下文环境

进程的上下文环境主要是指此进程运行时 CPU 各寄存器的内容,包括指令计数器、程序状态字和堆栈指针等。每个进程运行时 CPU 寄存器的内容都不同,一个进程被调度出去时需要保护它的上下文环境,当它被调度时需要恢复到之前保存的内容。

3. 进程调度信息

进程调度相关的信息包括进程的状态、进程的调度策略、进程的优先级、进程的运行睡眠时间、进程的阻塞原因和进程的队列指针等。当进程处于不同的运行状态时会被放到不同的队列中。当进程处于就绪状态时,调度器会根据进程的调度信息在必要时调度进程执行。一个系统中通常同时有多个就绪进程,哪个进程被调度就由调度器根据进程的调度信息选择一个调度执行。

4. 进程控制信息

进程的控制信息包括:进程的代码、数据、堆栈的起始地址;进程的资源控制,比如进程的内存描述符、文件描述符、信号描述符和 IPC 描述符等。进程使用的所有系统资源都会在 PCB 中描述。

进程创建时内核分配 PCB 块,当进程请求资源时内核会将相应的资源描述信息加到进程的 PCB 中,进程退出时内核会释放 PCB 块。通常来说进程退出时应该释放它申请的资源,如文件描述符等。为了防止进程遗忘某些资源而导致资源泄漏,内核通常会根据 PCB 中的信息回收进程使用的所有资源。

5. 进程控制块的组织方式

前面介绍了进程在它的生命周期中会处于不同的状态。当进程处于不同状态时,内核会以不同的方式组织它们。常用的方式如下。

(1) 链接方式。链接方式把进程按照不同的状态分别放到不同的队列。比如,系统中所有就绪的进程可以放到一个队列中,进程调度算法从中选取进程执行。再比如,系统可以分配很多个阻塞队列,每个队列对应一种阻塞原因,因为某种原因被阻塞的进程被加入到相应的队列,当进程的阻塞原因解除时内核可以通过相应的阻塞队列找到进程。

(2) 索引方式。索引方式应用像 hash 表等方式建立相应的索引表,处于不同状态的进程被放到相应的索引表中。

一个实用的内核通常会根据需要混合使用这些方式。不管使用哪种方式,都是把 PCB 放到不同的队列或表中,PCB 就代表了进程。

在 Linux 内核里,进程控制块定义为 task_struct。图 3.1 描述了 task_struct 的主要成员。

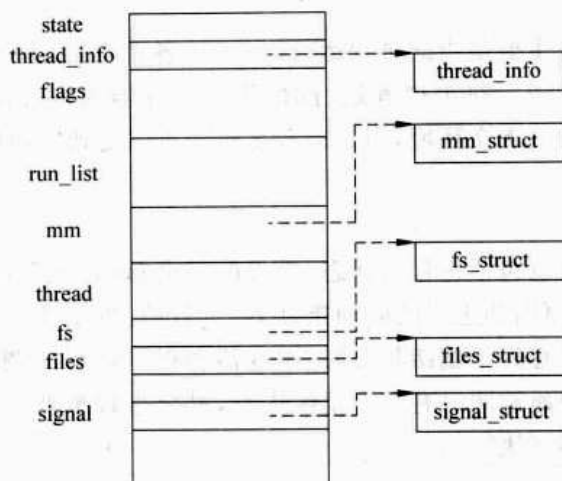


图 3.1 task_struct 的主要成员

3.1.2 进程状态

因为进程是程序的一次执行,是操作系统分配系统资源的基本单位,所以进程有状态,用于表示资源获取到或获取不到的情况。传统的操作系统把进程状态分为 3 种。

(1) 运行状态。进程正在处理机上运行的状态,该进程已获得必要的资源,也获得了处理机,用户程序正在处理机上运行。

(2) 阻塞状态。进程等待某种事件完成(例如,等待输入输出操作的完成)而暂时不能运行的状态,处于该状态的进程不能参加竞争处理机,此时,即使分配给它处理机,它也不能运行。为了方便处理,现在的操作系统一般把阻塞状态细分为更多的子状态。

(3) 就绪状态。该进程运行所需的一切条件都得到满足,但因处理机资源个数少于进程个数,所以该进程不能运行,而必须等待分配处理机资源,一旦获得处理机就立即投入运行。

这些状态是互斥的并可以做转化,也就是说,一个进程在一个特定的时间只能处于一种状态,当条件满足时,转化到另一状态。

- (1) 就绪状态变化到运行状态。
- (2) 运行状态变化到就绪状态。
- (3) 运行状态变化到阻塞状态。
- (4) 阻塞状态变化到就绪状态。

Linux 内核用 `task_struct->state` 表示进程的状态。

- **TASK_RUNNING**。TASK_RUNNING 表示进程处于运行状态。传统的 UNIX 系统定义了进程 READY 和 RUNNING 状态,这里 Linux 内核定义的 TASK_RUNNING 大致相当于 UNIX 的 READY 状态,也就是说,TASK_RUNNING 表示阻碍进程运行的条件消失了,进程可以调度到 CPU 上运行了,并不表示该进程正在 CPU 上运行,因为处于 TASK_RUNNING 状态的进程数量可能超过机器内的 CPU 数量。为了方便内核调度,系统把处于 TASK_RUNNING 状态的进程连接在一块。Linux 2.6 内核做了优化,每一个逻辑 CPU 单独维护一个进程运行队列。与传统的操作系统定义的进程状态相对照,Linux 内核定义的 TASK_RUNNING 是运行状态和就绪状态的合集。
- **TASK_INTERRUPTIBLE**。当进程运行过程中可能会遇到资源不满足的情况,或者需要等待某件事情的发生,这时候进程就要进入睡眠状态,等待条件满足后再恢复运行。典型的例子是进程提交了一个读硬盘的请求,如果缓存内没有要找的数据,内核就启动硬盘驱动程序去读对应的数据块。在数据到位之前,进程需要进入睡眠状态,出让 CPU 给其他进程。Linux 内核把睡眠状态分为 TASK_INTERRUPTIBLE 和 TASK_UNINTERRUPTIBLE 两种子状态。所谓的可中断和不可中断是指的是否被信号中断。信号是 Linux 内核提供的一种通信机制。当进程处于 TASK_INTERRUPTIBLE 状态时,如果内核或者其他进程向该进程发送了一个信号,该进程就会被唤醒,以及时处理收到的信号。当然,如果没有收到信号,但是进程的运行条件满足了,例如在前面的读硬盘的例子中,数据到位了,这时候进程就会被唤醒。当进程需要睡眠时,它的进程描述符就会从 CPU 的运行队列中摘除,然后加入到睡眠队列。依据进程所等待的条件不同,睡眠队列有很多种,例如前面的读硬盘例子中,每一个读硬盘的请求有一个队列,该进程就挂在该队列上,当然在该例中,一般这个队列只包含一个进程。当硬盘读操作完成后通过硬件中断通知驱动程序,驱动程序会唤醒挂在读硬盘请求队列的进程。
- **TASK_UNINTERRUPTIBLE**。当进程处于 TASK_UNINTERRUPTIBLE 状态时,信号不可以唤醒该进程。这种状态表示进程等待的条件必须满足,否则进程无法前进,就连回退取消正在进行的操作也不可以。此时如果想在用户空间通过命令 `kill` 来杀死该进程,该进程也不会立即死掉,因为 `kill` 命令就是向进程发送 SIGTERM 信号来杀死进程的,而处于 TASK_UNINTERRUPTIBLE 状态的进程不会被唤醒去处理信号。
- **TASK_STOPPED**。TASK_STOPPED 是用于作业控制的。当进程收到信号 SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU 时,进入 TASK_STOPPED 状态。
- **TASK_TRACED**。一般操作系统提供调试的能力,常用的调试工具有 gdb、

strace 等。当 gdb 调试一个程序时, gdb 本身就是一个进程, 被调试的程序是另外一个进程。为了设置断点和查看数据, gdb 需要访问被调试进程的地址空间。为了支持这些调试工具, 内核提供 ptrace 的系统调用。通过 ptrace, gdb 就可以读写被调试进程的地址空间。被调试进程可能正在运行, 随时可能更改自己的数据, 甚至是代码。那么什么时候读写被调试进程的地址空间是安全的呢? TASK_TRACED 状态就是为此而引入的。如果一个进程需要访问另一个进程的地址空间, 它通过 ptrace 首先使被访问的进程转换到 TASK_TRACED 状态, 然后访问数据, 完成后, 再让该进程恢复运行。

- EXIT_ZOMBIE。EXIT_ZOMBIE 状态又被称为僵尸状态。当一个进程死亡后, 有时父进程需要取得子进程的信息, 比如退出状态, 所以并不立即回收子进程的描述符。父进程可以调用 wait 系列函数取得子进程的信息, 同时回收子进程描述符。如果父进程始终不调用 wait 系列函数取得子进程的信息, 则又分为两种情况: 一是只要父进程活着, 则子进程描述符不回收; 二是父进程死了, 则其所有子进程的父进程改成内核进程 init, init 进程会回收所有已死的子进程。
- EXIT_DEAD。如果一个进程被 ptrace 了, 这个进程的父进程会改成 ptrace 它的进程, 比如 gdb, 而该进程的原父进程的地址信息保存在 task_struct->real_parent。这样一来, 当进程死亡转到 EXIT_ZOMBIE 状态时, gdb 和该进程的原父进程都可能争抢着回收该进程的描述符, 可能出现竞态。EXIT_DEAD 就是为了避免这种现象, 通过一个原子操作设置该进程状态为 EXIT_DEAD, 保证了只有一个父进程回收该进程的描述符。

3.1.3 进程的标识

在用户空间, 每一个进程用一个数字唯一标识, 称做进程 ID。

Linux 内核也采用 ID 标识进程。但是有一个例外, 每一个逻辑 CPU 有一个 IDLE 进程, 所有的 IDLE 进程的 ID 都是 1。另外为了使用的方便, 在内核里, 一般进程用 task_struct 的地址唯一标识。内核定义了一个宏——current, 指向当前进程的 task_struct。

3.1.4 进程描述符在内存里的组织形式

图 3.2 描述了进程描述符在内存里的组织形式。在 i386 机器架构下, 内核为每一个进程分配两个连续的物理页面, 每个物理页面是 4096B, 总共 8192B。这两个连续的物理页面必须在物理地址空间里按照 8192B 对齐。进程的内存堆栈从这两个物理页面的高位地址向下长。在两个物理页面的底部则是数据结构 thread_info。thread_info 包括一些最常使用的进程信息, 以及指向 task_struct 的指针。因为 task_struct 比较大, 为了减少堆栈覆盖掉 task_struct 的可能性, i386 内核把 task_struct 单独分配。

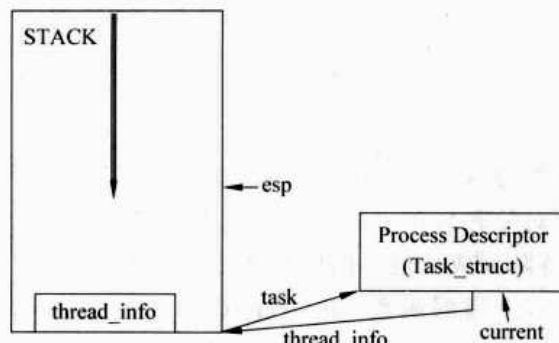


图 3.2 进程描述符在内存里的组织形式

因为两个物理页面按照 8192B 对齐,所以 `current` 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_struct` 得到 `task_struct` 的地址。

为了加快进程描述符的查找,内核在进程描述符设置了很多指针,如 `parent`、`sibling` 和 `children` 等。`parent` 指向父进程,`sibling` 指向兄弟进程,`children` 指向第一个子进程。所有的子进程用 `sibling` 的双向链表链接在一起。如果想遍历一个进程的所有子进程,可以从这个进程的 `task_struct->children` 出发,按照 `sibling` 绕双向链表一圈即可。

3.1.5 进程的睡眠和唤醒

进程处于堵塞状态时一般称为进程睡眠。反之,从堵塞状态转换到运行状态称为唤醒。操作系统需要提供数据结构支持进程的睡眠和唤醒,如记录进程的相关睡眠信息,在唤醒时通过查询这些信息可以根据需要唤醒特定的进程。

Linux 内核使用等待队列实现进程睡眠的基本数据结构。一个等待队列包含一个等待队列头和一到多个等待队列成员。等待队列头的定义如下。

```
struct __wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};

typedef struct __wait_queue_head wait_queue_head_t;
```

等待队列成员的定义如下。

```
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;

```

等待队列成员挂在等待队列头后面。一般,等待队列头作为其他数据结构的一个成员,这样和具体的事件/对象联系在一块。而等待队列成员经常是局部变量,用在进程睡眠的函数内。当然,等待队列成员也可以以其他方式存在。

Linux 内核提供了多个睡眠函数,如 `sleep_on`、`interruptible_sleep_on`、`prepare_to_wait`、`prepare_to_wait_exclusive`、`finish_wait`、`wait_event` 和 `wait_event_interruptible`。它们适用于不同的场合。一般用 `wait_event` 和 `wait_event_interruptible` 可以保证更好的操作原子性。

Linux 内核同时提供了多个唤醒函数,如 `wake_up`、`wake_up_nr`、`wake_up_all`、`wake_up_interruptible`、`wake_up_interruptible_nr`、`wake_up_interruptible_all`、`wake_up_interruptible_sync` 和 `wake_up_locked`。

3.2 线程的原理与实现

前面介绍了进程的概念。进程是程序的一次执行,同时也是资源分配的基本单位。为了提高执行效率,现代操作系统引入了线程的概念。线程是比进程更小的、能独立运行的基本单位。线程比进程能更好地提高程序的并行执行,现代的主流操作系统如 Linux、Windows 都引入了线程的概念。

1. 线程的基本概念

引入线程主要是为了提高系统的执行效率。线程和进程的主要区别和联系如下。

(1) 进程是资源分配的基本单位。所有和进程相关的资源如内存、文件描述符等是以进程为单位分配的。一个进程中的所有线程共享该进程的所有资源。

(2) 线程是进程中的最小调度单位。

(3) 一个线程属于一个进程。一个进程至少有一个线程。

2. 为什么引入线程

了解了线程和进程的区别,那么引入线程的好处就容易理解了。

(1) 线程的创建开销小。线程使用进程的资源,因此创建线程重新申请的资源大大减少,自然创建速度快。

(2) 线程间的同步开销小。进程之间有相互隔离的地址空间,进程的同步必须 OS

的介入。线程之间共享地址空间,可以实现很多轻量级的同步原语,因此同步开销小。

(3) 线程更易于调度。一个进程的线程之间共享资源,因此在它们之间进行切换时需要保存的现场更少,更易于切换。

(4) 线程易于实现并发。线程间共享资源,多个线程执行程序的不同部分时易于访问共享的资源。程序也可使用多个进程并发执行,但由于进程间有独立的地址空间,共享资源难于实现。

3. 线程的实现方法

线程的实现是指操作系统在一个进程中创建多个线程的方法。线程的实现在各个操作系统中采用了不同的方法。一般来说有以下两种方法。

1) 用户级线程

用户级线程是指线程由应用程序建立,所有线程相关的内容都由应用程序管理和调度,操作系统并不知道线程的存在,只对进程进行管理。早期的 Linux 线程就属于此类。

其优点如下。

(1) 实现简单。用户级线程和内核无关,不需要修改内核,因此便于实现。

(2) 便于移植。用户级线程的创建、调度和管理等算法和内核无关,很容易在多个操作系统间移植。

(3) 开销小。因为不需要内核的干预,线程的创建、退出等开销小。

其缺点如下。

(1) 多个线程不能同时运行。在这种实现中内核的调度单位是进程,一个进程只会在一个 CPU 上运行,因此这样的线程无法同时在多个 CPU 上运行,减少了程序的并发。同时,如果一个线程的系统请求在内核阻塞,则整个进程被阻塞,其他线程也不能被调度。

(2) 因为没有内核干预,有些功能难于实现。例如做线程调度时需要知道每个线程的运行时间,没有内核干预很难准确的得到这样的信息。

2) 内核级线程

内核级线程中所有线程的创建、调度和管理都由内核完成。Windows NT 属于此类。

内核线程的优点如下。

(1) 一个进程的多个线程可以同时执行。在内核级线程的实现方法中,内核的调度单位通常为线程。内核可以将多个线程同时调度到多个 CPU 上执行。

(2) 一个线程的阻塞不会影响其他线程。线程的阻塞只会将本线程挂起,其他线程不会受影响。

内核线程的缺点是实现复杂,内核需要专门的数据结构、模块来描述线程。另外,内核线程的创建和管理等需要内核的干预,因此和用户线程相比要相对开销稍大。

当然在一个实现内核级线程的操作系统中,应用程序还是可以使用用户级线程,因此

有些操作系统也使用混合的方式。如何选择要看具体的需求。

4. Linux 的线程模型

Linux 2.6 内核使用的是内核级线程的实现方式。Linux 内核实现了轻量级进程的概念,轻量级进程实质就是线程。为了实现内核级线程,大多数操作系统单独定义一套描述线程的数据结构,采用独立的模块来管理线程,这些增加了内核的复杂性。在 Linux 内核中,进程和线程不需要区分,内核使用统一的方式对它们进行创建、管理和调度。同时, Linux 的轻量级进程又保留了线程的特性,如没有独立的地址空间、线程之间共享资源等。Linux 以一种简单的方式实现了内核级线程。

3.3 Linux 进程创建和结束

3.3.1 进程的创建

在 Linux 中,进程的创建是由系统调用 `fork` 和 `vfork` 完成的,它们生成一个子进程并且子进程是父进程的复制品。

1. `fork` 的实现

`fork` 系统调用对应的 kernel 函数是 `sys_fork`,此函数简单的调用 kernel 函数 `do_fork`。一个简化版的 `do_fork` 执行如下。

(1) `alloc_pid()`。给新的进程分配一个新的 pid,即进程号。

(2) `copy_process()`。此函数会做 `fork` 的大部分事情,它主要完成将父进程的运行环境复制到新的子进程,比如信号处理、文件描述符和进程的代码数据等。下面会详细介绍此函数。

(3) `wake_up_new_task()`。计算此进程的优先级和其他调度参数,将新的进程加入到进程调度队列并设此进程为可被调度的,以后这个进程可以被进程调度模块调度执行。

`fork` 系统调用返回到父进程时会返回子进程的进程号,在子进程中这个调用会返回 0,读者可能会觉得奇怪为什么一个函数会有两个不同的返回值。介绍完 `copy_process()` 之后这个问题就清楚了。

简化的 `copy_process()` 流程如下。

(1) `dup_task_struct()`。分配一个新的进程控制块(`struct task_struct`),包括新进程在 kernel 中的堆栈。新的进程控制块将复制父进程的进程控制块,但是因为每个进程都有一个 kernel 堆栈,新进程的堆栈将被设置成新分配的堆栈。

(2) 初始化一些新进程的统计信息,如此进程的运行时间(`task_struct. utime, task_`

struct, stime)。

(3) copy_semundo()。复制父进程的 semaphore undo_list 到子进程。进程使用的所有 IPC semaphore 都会按照被调用的顺序记录在 task 的 sysvsem. undo_list 成员中。如果 fork 设置了 CLONE_SYSVSEM 标志,子进程会继承父进程的 undo_list。

(4) copy_files()、copy_fs()。复制父进程文件系统相关的环境到子进程。copy_files()将父进程所有打开的文件描述符复制到子进程中,因此子进程可以使用相同的文件描述符来使用父进程打开的文件。copy_fs()复制父进程的当前目录、根目录等文件系统相关的信息到子进程。

(5) copy_sighand()、copy_signal()。复制父进程信号处理相关的环境到子进程。copy_sighand()复制父进程注册的所有信号处理函数到子进程。copy_signal()复制其他信号相关内容到子进程,如未决信号集合。

(6) copy_mm()。复制父进程内存管理相关的环境到子进程,包括页表、地址空间和代码数据等。Linux 使用了写时复制的技术来减少复制,后面会详细介绍此技术。

(7) copy_thread()。设置子进程的执行环境,如子进程运行时各 CPU 寄存器的值、子进程的 kernel 栈的起始地址。在 Linux i386 中系统调用的返回值存储在寄存器 EAX 中,copy_thread()会将子进程的 EAX 设置为 0,因此 fork()系统调用返回到子进程后会返回 0。而对于父进程,它的 EAX 寄存器被置为子进程的进程号。

(8) sched_fork()。设置子进程进程调度相关的参数,即子进程的运行 CPU、初始时间片长度和静态优先级等。子进程的初始时间片的长度被设置为父进程的一半。

(9) 将子进程加入到全局的进程队列中。

(10) 设置子进程的进程组 ID 和会话 ID 等,这些都是从父进程继承而来。

简单来说 copy_process()就是将父进程的运行环境复制到子进程并对某些子进程特定的环境做相应的调整。

2. fork

fork 系统调用和 fork 类似,但是 vfork 的子进程必须马上调一个 exec 系统调用来运行一个可执行文件。执行 fork 的父进程会一直等待,直到子进程执行 exec 系统调用。fork 在 Linux 中也是由 do_fork 实现,和 fork 系统调用相比只是调用时参数不同,多了一个 CLONE_VFORK 标志。父进程在上面介绍的 do_fork()步骤(3)后面会检查 CLONE_VFORK 标志,如果是一个 vfork,do_fork()会等待 task_struct. vfork_done(它其实也是一种 mutex 的变体),从而让父进程睡眠。子进程执行 exec 系统调用时会唤醒父进程。

3. 线程的创建

Linux kernel 实际上没有线程的概念。所有的线程都被看作是一个进程,或者说轻量级进程,也就是线程和轻量级进程是等价的。轻量级进程没有独立的地址空间,一个进程中的所有轻量级进程会共享资源,如地址空间、信号处理等。进程中的轻量级进程是平等的,没有主从之分。轻量级进程也有一个 `task_struct`。某些仅仅针对进程的机制比如进程信号处理等, Linux 则是使用进程中的任意轻量级进程来处理。可以说 Linux 以一种统一的方式来处理进程和线程,只有在必要的时候对进程做特殊处理。

在 Linux 中,创建一个线程实际上也是执行上述步骤(`do_fork()`),但是创建线程会使用一些特殊的标志,如 `CLONE_THREAD`。有了这样的标志, `copy_mm()`、`copy_signal()` 等会让新的进程和以前的进程(注意是轻量级进程之间,不是父进程和子进程)共享一些环境,如地址空间、信号处理和文件描述符等。举个例子, `copy_mm()` 会检查 `CLONE_VM` 标志,创建线程时会设置此标志。如果设了此标志,则不会创建新的 `mm_struct`(进程的内存管理控制块),新创建出来的进程会使用以前进程的 `mm_struct`,也就是说以前进程的 `mm_struct` 的引用记数会增加。

一个有趣的问题是如果一个多线程的进程进行 `fork()`,新的进程会有几个线程。答案是只有一个线程,因为 `fork()` 永远只会创建一个轻量级进程。

4. 写时复制(COW)

前面的 `copy_mm()` 中提到了写时复制,在进程创建时现代操作系统都使用了这项技术, Linux 也不例外。 `fork()` 出来的子进程和父进程有相同的代码和数据。如果不使用写时复制,这些代码和数据会在子进程和父进程中各有一份。为了避免重复的代码和数据, `copy_mm()` 复制页表时会做一些处理以便父子进程的页表会指向相同数据并且页表会被设置成只读,但是数据本身并不会被复制一份。当进程对数据进行写操作时会产生一个异常,因为此数据是只读的。这个异常由内存管理中的缺页处理程序处理,它会复制一份新的数据给当前进程。因为代码是不可被更改的,此项技术对父子进程共享代码尤其有用。

5. fork 的标志

前面提到了 `do_fork` 会接受不同的标志来区分 `fork`、`vfork` 等。下面是一些比较典型的标志。

- `CLONE_VM`。表示新的进程和父进程共享内存管理环境,创建线程会设置此标志。
- `CLONE_FS`。表示新的进程和父进程共享文件管理环境,创建线程会设置此

标志。

- CLONE_FILES。表示新的进程和父进程共享打开的文件,创建线程会设置此标志。
- CLONE_SIGHAND。表示新的进程和父进程共享信号管理环境,创建线程会设置此标志。
- CLONE_VFORK。vfork 系统调用设置此标志。
- CLONE_THREAD。创建线程设置此标志。

6. 进程 0

进程 0,也就是 swapper 进程,是所有进程的祖先。这个进程的创建有点特殊,它不是从某个进程 fork 出来。其进程控制块是静态分配的,也就是说在 kernel 编译时就已经存在了。在 Linux 中,进程 0 的进程控制块是由宏 INIT_TASK 初始化的。

3.3.2 进程的结束

1. 进程结束

应用程序使用系统调用 `exit()` 来结束一个进程,此系统调用接受一个退出原因代码,父进程可以使用 `wait()` 系统调用来获取此代码,从而知道子进程的退出原因。对应到 kernel,此系统调用 `sys_exit_group()`。它的基本流程如下。

(1) 将信号 SIGKILL 加入到其他线程的信号队列中,并唤醒这些线程。这些线程被唤醒后都会调用 `do_group_exit()`(由信号处理执行路径调用,调用函数是 `get_signal_to_deliver()`),在这种情况下这些线程开始执行 `do_exit()`。

(2) 此线程执行 `do_exit()` 来退出。

实际上 `sys_exit_group()` 就是唤醒一个进程中所有的线程,让这些线程执行 `do_exit()`。`do_exit()` 完成线程退出的任务,其主要的功能是将线程占用的系统资源释放,`do_exit()` 的基本流程如下。

(1) `exit_mm()`。将进程内存管理相关的资源释放。这包括释放进程的地址空间、释放将虚拟地址转换成物理地址的页表,释放进程的代码数据页等。由于进程的页表被释放,此进程将暂时借用进程 0 的页表。

(2) `exit_sem()`。将进程 IPC semaphore 相关的资源释放。前面介绍过进程使用的所有 IPC semaphore 都会按照被调用的顺序记录在 task 的 `sysvsem.undo_list` 成员中。退出进程时此进程占用的所有 semaphore 都会被反序释放。进程退出时有可能没有释放 semaphore,如果 kernel 不强制释放它们,其他进程永远都没法得到这些 semaphore,`exit_sem()` 就是要避免此情况发生。

(3) `__exit_files()`, `__exit_fs()`。将进程文件管理相关的资源释放。例如,将进程所有打开的文件关闭,将描述进程根目录和当前目录的资源释放。

(4) `exit_thread()`。主要目的是释放平台相关的一些资源。在 i386 中它主要是释放 io 位图。

(5) `exit_notify()`。在 Linux 中进程退出时要将其退出原因告诉父进程,父进程调用 `wait()` 系统调用后会在一个等待队列 (`task_struct`, `signal->wait_chldexit`) 上睡眠。`exit_notify()` 设置退出原因(由调用提供)并从等待队列上唤醒父进程,这样父进程可以得到此退出原因。`exit_notify()` 还要处理一些进程间关系,如此进程有子进程则所有子进程会变成孤儿进程,这样的子进程需要一个新的父进程,通常是进程 0。最后此函数还会调用 `release_task()` 来释放进程的 kernel 堆栈,将此进程未使用的时间片还给父进程等。

(6) `schedule()`。调用进程调度器,因为此进程已退出,切换到其他进程。以后这个进程不会再被调度执行,因此 `do_exit()` 函数是永远不会返回的。

如果某些资源是多个线程共享的,则上面的某些步骤如 `__exit_files()` 实际上只是更新了资源的引用计数,最后一个退出的线程才会真正将共享的资源释放。这点也适用于下面介绍的线程结束。

2. 线程结束

线程的结束和上面进程结束的流程类似。线程结束对应的 kernel 函数是 `sys_exit()`, 它直接调用 `do_exit()`, 唯一的差别是线程结束时不会将其他线程结束。

3. task_struct 的释放

细心的读者会发现上面的 `do_exit()` 实际上并没有将进程的 `task_struct` 释放。是什么原因呢? 进程调度模块进行调度时需要知道被调度出去和被调度进来的进程的进程描述符,如果 `do_exit()` 释放进程描述符则最后一步 `schedule()` 无法做调度。另外 `do_exit()` 是在进程自身上下文环境执行的,如果进程 `task_struct` 被释放了这种上下文环境也不存在了,进程不可能在上下文环境被释放的情况下执行。Linux 的解决方式是在 `do_exit()` 中给进程设置一个标志 (`TASK_DEAD`), 在进行进程切换时(将设置了此标志的进程切换出去时)检查此标志,相应的将此进程对应的进程描述符释放(对应的 kernel 函数是 `finish_task_switch()`)。

3.4 进程调度

一个计算机系统可以同时运行很多进程,如何调度进程是影响系统性能和进程响应时间的重要因素。在多处理器的机器里,进程的调度表现的更为重要。

传统的操作系统在进程调度上比较简单,主要采用优先级的方法,给不同的进程赋予不同的优先级。优先级高的进程优先得到调度运行,优先级低的进程推迟运行。进程调度必须防止进程饿死,即低优先级的进程始终得不到运行。

本节集中介绍 Linux 内核实现的进程调度。

3.4.1 进程的切换

进程的切换是操作系统调度的一个基本功能,以此实现多个进程共享 CPU。另外,当进程需要睡眠时,也必须切换进程。进程的切换主要包含几个内容。

1) 硬件上下文的切换

硬件上下文主要是寄存器,包括控制寄存器和应用或通用寄存器。例如 i386 中,CR3 保存页目录表的起始物理地址。当一个进程切换到另一个进程时,如果二者不共享地址空间,CR3 必须切换,否则后一个进程就访问前一个进程的数据了。通用寄存器如 EAX、EBX、ECX、EDX、ESP 和 EBP 等。它们的内容是具体进程相关的,需要切换。切换主要是通过保存和恢复实现的,就是保存前一个进程的寄存器到里,或到进程描述符索引到的地方,以及从保存后一个进程的寄存器的内存里恢复到寄存器里。

2) 任务状态段的切换

任务状态段是 i386 硬件提供的一种数据结构。其最初的想法是每一个进程有一个任务状态段。为了简化设计,Linux 内核实现了每一个逻辑 CPU 一个任务状态段。内核的堆栈地址是保存在任务状态段里,当从用户空间向内核切换时,硬件自动从任务状态段中取出内核堆栈地址并初始化堆栈指针寄存器。另一方面,不同的进程使用不同内核堆栈,所以当进程切换时,必须切换任务状态段中的堆栈指针。

I/O 位图是 i386 提供的另一个硬件数据结构,如果前后两个进程的 I/O 位图不同,也需要切换。

3) 内核堆栈

进程的切换一定是发生在内核状态下。如前所述,当进程切换时,必须切换内核堆栈指针寄存器。这时候的切换,是用软件的方法实现的。Linux 内核定义了一个宏,switch_to,实现内核堆栈指针寄存器的切换。其主要方法是把当前的堆栈指针寄存器的内容保存到 task_struct->thread.esp,然后把下一个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。

4) 浮点数寄存器

浮点数寄存器的特点是数量多,进程的每一次运行并不都访问浮点数寄存器。i386 提供了一个优化的能力。控制寄存器 CR0 有一个位叫做 TS。当 TS 等于 1 时,如果访问浮点数寄存器,CPU 会触发一个特殊的设备不可用的故障中断。利用这一特殊能力,内核实现了恢复浮点数寄存器的功能。当进程切换时,如果当前进程使用过浮点数寄存器,

则立即保存浮点数寄存器,但是不立即恢复下一个进程的浮点数寄存器,只是设置 CR0 的 TS 位为 1。当下一个进程访问浮点数寄存器时,内核再在处理故障中断时恢复浮点数寄存器,这样一来如果下一个进程没有在其时间片内使用浮点数寄存器,系统就节省了恢复浮点数寄存器的时间。

内核在三种情况下会保存浮点数寄存器。一是任务切换;二是处理信号,因为信号处理是异步事件,信号处理函数可能也会使用浮点数寄存器;三是内核自己需要使用浮点数寄存器。

3.4.2 进程的调度策略

通用操作系统可以用于处理各种不同的任务。这些任务,或者说应用程序可能差别很大。有的要求进程快速的响应,有的要求好的吞吐量。当操作系统同时处理这些任务时,必须统一考虑,避免进程饿死,协调好高优先级和低优先级的任务。这些要求存在一定的矛盾性,更加大了系统调度的难度。

Linux 的进程调度采用了策略(policy)来区分进程。Linux 定义了几种进程:实时进程、交互式进程和批处理进程。在 `task_struct` 中定义了 `policy` 字段,指定该进程的调度策略。Policy 可以等于 4 个值。

- `SCHED_NORMAL`。基于时间片机制。进程具体可以表现为交互式进程或批处理进程。
- `SCHED_FIFO`。先进先出的实时进程。
- `SCHED_RR`。循环执行的实时进程。
- `SCHED_BATCH`。批处理进程。

既然 Linux 定义了批处理进程策略 `SCHED_BATCH`,为什么还要定义 `SCHED_NORMAL`? 或者为什么不直接定义交互式进程策略? 原因是如果一个进程是交互式进程或批处理进程,则程序员在编写程序时一般不去设置进程的调度策略,而子进程继承父进程的调度策略,所以系统初始化时设置多数进程的调度策略为 `SCHED_NORMAL`,至于是交互式进程还是批处理进程,由操作系统动态监测并动态修改进程的优先级。另外一个原因是同一个进程在不同的执行时间可能表现为不同性质的进程。例如当进程刚启动时,它可能要求一些输入做一些配置,表现为交互式进程的特征,当输入配置完成后,它开始批处理,又变成了批处理进程。为了支持这些多样性和多变性,操作系统必须能够动态监测进程的调度策略。

调度策略只是比较大概的进程分类,实际调度时必须映射到调度的优先级。Linux 定义了 140 个优先级。

- 进程的优先级。0~139。
- 实时进程的优先级。0~99。

- 交互式进程和批处理进程优先级。100~139。
- 静态优先级。100~139。
- Nice。-20~19。
- 用户优先级。0~39。

0 是最高优先级,139 最低优先级。由于实时进程的优先级总是比交互式进程和批处理进程优先级高,所以总是实时进程优先运行。只有当没有处于运行状态的实时进程时,交互式进程和批处理进程才可以得到调度。

nice 和用户优先级是为了兼容 POSIX 标准设立的,在内核里实际使用时映射到交互式进程和批处理进程优先级。

时间片是进程实际被分配到的运行时间,以 tick 作为单位。tick 是从何而来? 操作系统有多个时钟处理机制,其中一个最重要的是用于 jiffies 计数的时钟,老的机器一般采用芯片上的可编程间隔时钟 8254(PIT),新的机器可以使用高精度时间时钟 HPET,另外还有其他的一些时间时钟。这些时钟都是外部时钟,设置成每秒触发时钟中断 HZ 次,每一次内核在变量 jiffies 上加 1。HZ 是内核定义的一个宏,实际是一个整数。时钟中断每发生一次算一个 tick。每当时钟中断发生时,内核就在当前进程的时间片里减 1,如果时间片变成了 1,内核就重新调度进程。当然,实际实现要复杂多了。在单 CPU 机器里,进程调度完全依赖于外部时钟,如 PIT/HPET 的时钟中断。在 i386 的多 CPU 机器里,CPU 内部的可编程中断控制器提供一个每个 CPU 自己的时钟中断源。在这种情况下,外部时钟仅用于驱动 jiffies,进程调度则由 CPU 自己的局部时钟中断源驱动。

实时进程包括 FIFO 和 RR 两种。FIFO 就是先进先出。当一个 FIFO 进程运行时,如果没有比它的优先级更高的进程可以运行,即使该 FIFO 进程的时间片运行完了,它也不会出让 CPU,而是再获得一个新的时间片,继续运行。从某种意义上讲,时间片对于 FIFO 进程没有约束力。RR 是循环调度运行的进程。每一个 RR 进程有一个时间片,当它的时间片用完后就需要向同一优先级的其他 RR 进程出让 CPU,所以称作循环调度运行。无论 FIFO 进程还是 RR 进程,内核不会动态调整它们的优先级。如果需要调整它们的优先级,则进程自己需要调用特定的系统调用改变优先级。

与实时进程不同,交互式进程和批处理进程的优先级是动态变化的。内核用一个公式计算出进程新的优先级。这个公式的输入参数是进程的静态优先级和最近的过去进程睡眠的时间。如果最近的过去进程睡眠的时间越长,表示进程越像交互式进程,应该获得更高的优先级。函数 `recalc_task_prio` 计算进程新的优先级。

3.4.3 进程调度的触发和进程切换的时机

进程调度的核心任务由函数 `schedule` 处理,主要是选择并切换到下一个合适的进程。但是进程调度的触发则由多种情况。前面提到进程调度主要由时钟中断触发。在内

核里,函数 `scheduler_tick` 处理由时钟中断触发的调度。但是这里的调度只是给当前进程打标记 `TIF_NEED_RESCHED`。`TIF_NEED_RESCHED` 是 `task_struct->thread_info.flags` 中的一个标志位。当在合适的时候,当前进程自己检查自己的 `TIF_NEED_RESCHED` 标志位,如果 `TIF_NEED_RESCHED` 被设置了,则调用 `schedule`。那么什么是合适的时候呢?为什么不立即在 `scheduler_tick` 里调用 `schedule` 去立即选择并切换到下一个合适的进程?原因是当进程在内核里运行的时候,可能申请了很多共享资源,如自旋锁 `spinlock`,如果此时立即让该进程出让 CPU,如果下一个进程也需要相同的共享资源,则会导致系统的死锁。所以一定要当进程进入合适的时候,也就是已经释放了共享资源后并处于安全的时候,才可以调用函数 `schedule` 做真正的进程调度。这里的合适的时候一般是当进程要退出内核,回到用户空间时,因为此时进程必须已经释放了可能导致内核死锁的资源。如果内核支持先占式调度,在释放自旋锁和退出中断处理时,也可能引发调用函数 `schedule`。

进程自己可以主动调用函数 `schedule` 选择并切换到下一个合适的进程。如一些内核线程处理一些比较费时的任务时,中间会调用函数 `schedule`,使其他进程有机会运行。

除了函数 `scheduler_tick` 可以触发调度,当唤醒一个进程时也可能触发调度。函数 `try_to_wake_up` 用于处理唤醒。如如果被唤醒的进程的优先级比当前进程的优先级高,而且被唤醒的进程需要在当前 CPU 上运行,则 `try_to_wake_up` 会设置当前进程的 `TIF_NEED_RESCHED` 标志位。另外,创建新的进程和 `exec` 时,都可能引发进程的调度。

3.4.4 运行队列

运行队列(runqueue)是内核支持调度的重要的数据结构。下面是 runqueue 的一个简化定义。

```
struct runqueue {
    spinlock_t lock;
    unsigned long nr_running;
#ifdef CONFIG_SMP
    unsigned long cpu_load[3];
#endif
    unsigned long long nr_switches;
    unsigned long nr_uninterruptible;

    unsigned long expired_timestamp;
    unsigned long long timestamp_last_tick;
    task_t * curr, * idle;
    struct mm_struct * prev_mm;
```

```

prio_array_t* active, * expired, arrays[2];
int best_expired_prio;
atomic_t nr_iowait;

#ifdef CONFIG_SMP
    struct sched_domain* sd;

    //For active balancing
    int active_balance;
    int push_cpu;

    task_t* migration_thread;
    struct list_head migration_queue;
    int cpu;
#endif

#ifdef CONFIG_SCHEDSTATS
    :
#endif
};

```

在 runqueue 的定义里, active、expired 和 arrays 是比较重要的成员。字段 arrays 是一个有两个成员的数组, 每一个成员内含 140 个双向链表, 分别表示 0~139 种优先级。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 的两个成员。进程调度时从指针 active 指向的 arrays 的成员里选择下一个要运行的进程。当交互式进程和批处理进程的时间片用完后, 就会被移到指针 expired 指向的 arrays 的成员的链表里。

每一个逻辑 CPU 有一个运行队列, 这样 CPU 之间的竞争就会比较少。这个特性加上 140 个双向链表设计是实现线性时间复杂度调度算法的关键。

所有的处于 TASK_RUNNING 状态的进程的描述符必须链接到一个运行队列里, 否则进程就会不被调度而永远睡眠过去。

3.4.5 调度域

由于每一个逻辑 CPU 有一个运行队列, 一般调度时只是从 CPU 自己的运行队列选择下一个要运行的进程, 那么必然容易出现一个运行队列有很多进程而另一个运行队列进程很少甚至没有进程的情况, 这样 CPU 的资源就会浪费, 进程也不能得到及时的响应。为了减少这种情况出现, 操作系统必须做到运行队列里进程数量的相对平衡。当一

个 CPU 的运行队列比较短时,内核就要向其他 CPU 的运行队列查询,如果其他 CPU 的运行队列比自己的运行队列长,内核会把一些进程从其他 CPU 的运行队列里摘除,然后加入到自己的运行队列。这叫做 CPU 负载平衡。那么当一个 CPU 的运行队列比较短时,内核应该首先查看哪一个 CPU 的运行队列呢?这涉及到计算机的系统结构。

早期的计算机只有一个 CPU,随后出现了 SMP,也就是对称多处理机。对称多处理机的所有 CPU 对内存的访问速度是一样的,所以称作对称。对称多处理机的每一个 CPU 有自己的独立 cache、寄存器和运算单元。

cache 是现代计算机不可或缺的重要部件,是提高计算机性能关键部位之一。访问一级 cache 的时间可能只需一个 CPU 周期,但是访问内存的时间很容易就达到几十个 CPU 周期。cache 的设计原理是基于访问的局部性原理。所谓局部性原理包含时间局部性和空间局部性。时间局部性指 CPU 刚访问过的数据,CPU 在不久的将来还会访问到。空间局部性指 CPU 访问了某一数据后,CPU 很可能会访问这一数据空间上相邻的数据。cache 存放的是内存的映像,或者被 CPU 修改过的数据,以后刷新到内存。如果对某一数据读或写,发现访问的数据在 cache 存在时,称作 cache 命中,反之称作 cache 不命中。好的计算机 cache 命中率可以达到 99% 以上。cache 存放的内存映像是以缓存线(cache line)为单位的,一般一个 cache 包含 64B 或 128B。为了提高 cache 的响应速度,cache 分级,最靠近运算单元的称作一级 cache,其次称作二级 cache,余者以此类推。有些级别的 cache 又分割为两块,即指令 cache 和数据 cache,因为指令和数据的访问模型是不一样的。

作为对称多处理机的一种形式,Intel 公司推出了多线程 CPU (hyperthreading),得到了广泛的应用。多线程 CPU 的特点是一个物理的 CPU 可以包括多个逻辑 CPU。这些逻辑 CPU 各自有自己的通用寄存器,但是逻辑 CPU 共享 cache 和运算单元。由于现代 CPU 通常包括多个运算单元,因此多线程 CPU 实质上可以同时运行。另外多线程 CPU 的一个设计原因是如果 cache 里没有要访问的数据时,CPU 必须访问内存,这样就有一个比较大延迟,利用这个延迟,多线程 CPU 硬件就会调度另一个 CPU 线程运行。和单 CPU 相比,多线程 CPU 实际上大约提高系统性能 30%。

多线程 CPU 之后,又有了多核 CPU。Intel 公司已经推出了双核和 4 核 CPU。多核 CPU 就是一个物理的 CPU 包含了两个或多个 CPU 核心。每一个核心有自己独立的寄存器和运算单元。同一个物理 CPU 不同核心可以共享最后一级 cache,也可能不共享最后一级 cache。和多线程 CPU 相比,多核 CPU 真正实现了一个物理 CPU 有多个真正的 CPU,所以理论上系统性能应提高至 N 倍。N 是核心的个数。同一个物理 CPU 不同核心共享最后一级 cache 的好处是一个核心访问过的数据,另一个核心可以直接从 cache 里访问到,而不用访问内存。比如 Intel 公司推出的 Woodcrest 系列 CPU 核心共享最后一级 cache。

无论是多线程 CPU 还是多核 CPU, 每一个逻辑 CPU 访问内存的速度是一样的。所以它们都是对称多处理机。对称多处理机的一个特点是 CPU 共享连接 CPU 和内存的前端总线。当 CPU 的数目比较大时, 这种架构的缺点就暴露出来。因为共享前端总线, 当一个 CPU 访问内存时, 其他 CPU 不得使用总线, 必须等待前一个 CPU 使用完总线后, 其他 CPU 才可以重新竞争总线。随着 CPU 的数目变大, 越来越多的 CPU 会处于等待状态, 系统性能受到严重损害。为了克服这个缺点, 不对称 NUMA (Non-Uniform Memory Access) 架构计算机出现了。NUMA 机器内部分为节点 (node), 每个节点有自己的一组 CPU 和内存。每个节点自己就像一个小的 SMP。节点之间有专门的器件相连, 一个节点的 CPU 可以访问另一个节点的内存, 只是访问的速度远远落后于访问本节点的内存的速度。根据 cache 是否与内存保持一致, NUMA 又分为 cache 一致 NUMA 和 cache 不一致 NUMA。目前 cache 一致 NUMA 应用较多, 后者更像集群 (cluster), 不在本次讨论的范围。

在做 CPU 负载平衡时, 主要的出发点是当进程被移动到另一个 CPU 的运行队列后, 该进程的运行效率不能降低。这里影响进程的运行效率的主要因素就是 cache, 或者说访问数据的速度。如果一个进程在一个 CPU 上运行时修改了大量数据, 并且这些数据在 cache 里, 当这个进程被移动到另一个 CPU 的运行队列恢复运行后, 必然触发大量的 cache 不命中事件, 进程运行效率剧烈降低。根据访问 cache 和内存的速度, 内核把逻辑 CPU 划分成组。首先是分成几个很小的组, 然后几个小组构成大组, 几个大组又可以构成更大的组, 最后一个最大的组包含所有的逻辑 CPU。这样一个逻辑 CPU 同时属于一个小组, 一个中组, 一个大组, 直至最后一个最大的组。这种分组的设计思想称作调度域 (schedule domain)。

具体实现上, 一个运行队列有一个调度域链表。每个调度域包含几个调度组。调度组用双向链表连接在一块。CPU 成员越少的域, 离运行队列越近。

调度组的简化定义如下。

```
struct sched_group {
    struct sched_group * next;           //Must be a circular list
    cpumask_t cpumask;
    unsigned long cpu_power;
};
```

调度域的简化定义如下。

```
struct sched_domain {
    struct sched_domain * parent;
    struct sched_group * groups;
    cpumask_t span;                     //span of all CPUs in this domain
};
```



```
//其他的有关调度域的字
```

```
:
```

```
};
```

`sched_group` 的成员 `cpumask` 表示这个组的所有 CPU。`sched_domain` 的成员 `span` 表示这个域的所有 CPU。

当前内核实现了以下几种调度域。

- `cpu_domains`。多线程 CPU 的调度域,是一个最小的域,一般包含两个调度组,每一个组包含一个逻辑 CPU,即一个 CPU 核心的两个线程 CPU。因为它们共享 cache,所以把一个进程从一个线程的运行队列移动到另一个线程的运行队列,不会浪费时间。
- `core_domains`。CPU 核心的调度域。它的每一个调度组包含在一个核心里的所有的线程 CPU。一个物理 CPU 有几个核心,这个调度域就有几个调度组。
- `phys_domains`。物理 CPU 的调度域。其实质是 NUMA 机器的一个节点,或 SMP 的整台机器的 CPU。
- `node_domains`。把 NUMA 机器的几个结点合成一个调度域,它的每一个调度组包含在一个节点内的所有逻辑 CPU。一个节点是一个调度组。
- `allnodes_domains`。最大的调度域,包含所有的节点。

函数 `arch_init_sched_domains` 会初始化调度域。

在做 CPU 负载平衡时,在一个调度域里,内核首先调用函数 `find_busiest_group` 找最忙的调度组,然后再调用函数 `find_busiest_queue` 在最忙的调度组里找最忙的逻辑 CPU,然后调用函数 `move_tasks` 从这个最忙的逻辑 CPU 的运行队列里把某些进程移动到本 CPU 的运行队列。

触发 CPU 负载平衡主要是由函数 `scheduler_tick`,每一个 tick,在每个 CPU 上 `scheduler_tick` 都会调用 `rebalance_tick`。除了 `scheduler_tick`,其他触发 CPU 负载平衡的时机有当 CPU 处于空闲时,当唤醒一个进程时,当生成一个新的进程时,当进程执行 EXEC 时。

3.4.6 抢占

抢占(`preempt`)是指当一个低优先级的进程在运行时如果一个高优先级的进程就绪,高优先级的进程可以抢占 CPU,马上开始执行。在 Linux 2.6 之前,只有当进程处于用户态执行时才可被抢占,Linux 2.6 kernel 实现了在 kernel 态的进程抢占。Linux 的抢占是有限的抢占,即不是所有时刻进程都可被高优先级进程抢占。Linux 中进程执行时可以将某些代码禁止抢占,当进程执行这些代码时该进程不能被抢占,一旦进程执行完这样的代码,该进程就可以被抢占。前面讲的是进程在 kernel 态时的抢占情况,当进程处于

用户态时总是可以被抢占的。Linux 提供了两个抢占相关的函数：`preempt_disable()`和 `preempt_enable()`，前者禁止抢占，后者打开抢占。注意，有些函数会间接使用这样的函数，比如 `spin_lock()`，这样的函数执行时也会禁止抢占。

3.4.7 调度的时间复杂度

评价调度的算法好坏是根据它的时间复杂度。调度算法的时间复杂度和很多因素相关，比如进程的数目、CPU 的数目和实时性的要求等。当然，对于一个非实时操作系统它的调度算法的时间复杂度是不可能通过一个公式计算出来的，我们只能做个大致的评估。通常情况下起决定作用的是进程的数目。Linux 2.6 实现的调度算法是 $O(1)$ 的算法，即调度所用的时间和系统的进程数目无关。Linux 的调度算法采取了如下的措施来保证调度算法是 $O(1)$ 的。

(1) 进程按照优先级插入到调度队列，即前面 `runqueue` 结构里面的 `prio_array_t`。每个优先级有个队列，同时每个优先级对应一个位图的一个比特。调度算法选择进程时只需选择位图第一个置为 1 的比特，再根据这个比特在位图的位数得到优先级队列。需要调度的进程就是优先级队列的第一个进程。可以看出这个搜索算法和进程数目是无关的。

(2) 每个 CPU 都有自己的调度队列，即上面所讲的调度队列都是由对应的 CPU 来访问。如果所有 CPU 共享调度队列调度的时间复杂度就会和 CPU 数目相关，因此不是 $O(1)$ 的算法。

(3) 系统有专门的迁移进程保证 CPU 之间的负载平衡。因为调度队列是每个 CPU 都有一个，必然存在某些 CPU 上的进程数目(负载)要小于其他 CPU。Linux 调度算法不直接处理负载平衡，否则调度的时间复杂度就不是 $O(1)$ 。Linux 采取的策略是每个 CPU 上都有一个专门的内核线程来处理负载平衡。

3.4.8 函数 `schedule` 的实现

函数 `schedule` 负责选择下一个要运行的进程并切换到该进程。

简化的 `schedule` 执行流程如下。

(1) 检查当前进程是否处于原子状态。这里的原子状态表示当前进程不可以被调度出去。每一个线程的 `thread_info` 中有一个整数 `preempt_count`，`preempt_count` 被分切为抢占记数部、软中断部、硬中断部和是否已经抢占(`PREEMPT_ACTIVE`)标志位。如果内核编译时打开了 `CONFIG_PREEMPT`，当除了 `PREEMPT_ACTIVE` 标志位之外的其他位不为零时，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，后面会用这些数据调整进程的 `sleep_avg`、`timestamp` 和 `last_run` 等。

(3) 从运行队列中删除进程。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 0), 内核调用函数 deactivate_task 从运行队列中删除进程。一个例外是如果进程处于 TASK_INTERRUPTIBLE 并且有信号等待处理, 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。如果只剩下 IDLE 进程, 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, 则把 active 和 expire 交换。

(6) 从运行队列根据优先级选取下一个要运行的任务, 并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) 切换任务。如果前一个进程不是新选出的进程, 则切换任务。三个函数 prepare_task_switch、context_switch 和 finish_task_switch 负责切换。context_switch 做了大部分操作, 下面详细介绍 context_switch。

- 如果下一个进程有自己的 task_struct->mm, 调用 switch_mm 切换虚拟内存空间。在 i386 机器上, 主要是把下一个进程的页表起始物理地址写入寄存器 CR3。当然, 如果下一个进程和前一个进程共享同一个虚拟内存空间, 则不切换 mm。
- 调用 switch_to。switch_to 是用嵌入式汇编语言实现的, 用于切换内核堆栈 ESP, 执行指针 EIP, 并调用 _switch_to。_switch_to 检查如果前一个进程使用了浮点数, 则保存浮点数, 之后修改 TSS 中的 ESP0, 保存 fs 和 gs, 并根据需要重新设置 fs、gs、iopl、调试寄存器和 I/O 位图。

(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。

3.4.9 函数 load_balance 的实现

前面关于多处理机负载平衡讲述了负载平衡的多种时机。其中内核每一个 tick 都调用 scheduler_tick, scheduler_tick 又调用 rebalance_tick, 而 rebalance_tick 调用 load_balance。函数 load_balance 实际实现一个 CPU 的运行队列在一个调度域内的负载平衡。下面是 load_balance 实现的流程。

(1) 调用 find_busiest_group, 获得在一个调度域内的最忙的组。Imbalance 是函数 find_busiest_group 的一个参数, 负责传回需要做负载平衡的权重。随后被移动的进程的权重必须小于 imbalance。负载平衡的权重是如何规定并计算的呢? 每一个进程根据其静态优先级或实时优先级得到一个负载权重, 一个运行队列上的所有的进程的负载权重之和, 就是该运行队列的原始权重。运行队列的原始权重只是反映当前的权重, 如何反映系统在过去一段时间里的权重呢? 每个运行队列有一个数组记录三个权重, 通过把当前的原始权重与老的权重按不同比例相加, 得到这三个权重。

(2) 调用 `find_businest_queue`, 获得在一个调度域内的最忙的运行队列, 基本方法就是通过比较组内所有的运行队列的原始权重。

(3) 调用 `move_tasks`, 根据进程的权重和优先级, 把进程从最忙的运行队列移动到当前队列。被移动的进程必须不是热缓存, 即进程在原 CPU 上运行的时间不能太少。

(4) 如果移动的进程数目不足, 则启动内核线程 `migration`, 由 `migration` 负责做进一步的负载平衡。

(5) 重新计算负载平衡的时间间隔。

3.5 并发控制原理

由于多任务系统的特性, 操作系统需要在某种程度上保证多个任务并发运行时的正确性。

同时随着硬件技术的发展, 对称多处理器系统 (SMP) 及多核处理器变得越来越普遍, 对操作系统的同步控制提出了更高的要求。

当多任务的操作系统运行在单核处理器上时, 实际上任何时间只能有一个单一的进程/线程在运行。系统快速地在进/线程间切换, 造成并发的假象。

早期的多任务操作系统只运行在一个 CPU, 只在用户态可抢占, 这就意味着当进程运行在内核态时除了进程主动放弃 CPU 资源, 系统不可能被切换到别的进程, 这避免了许多问题。但在这种系统上, 操作系统仍需关心以下的同步问题。

1. 阻塞操作

当系统需要访问某些资源时, 可能需要阻塞进程等待资源状态, 在这种情况下, 某些资源可能需要加锁保护, 以防止阻塞进程被唤醒以前, 资源的状态被破坏。例如进程在阻塞等待磁盘 I/O 完成时, 可能需要锁住某些 I/O 缓冲块, 防止被其他进程破坏。

2. 中断

单核上运行的不可抢占内核仍然有可能被外部设备的中断打断。如果内核被打断时正在操作内核数据结构, 同时中断处理程序也要访问同一个数据结构, 就可能发生不一致的状态。内核处理这种竞态条件的办法是在访问这些数据结构的时候屏蔽中断。注意: 中断屏蔽时间过长将会降低系统的可用性, 而且通常在中断被屏蔽时, 进程通常不可以主动放弃 CPU 进入阻塞状态。

很多多任务操作系统可以在许多内核态被抢占, 这个特性可以使操作系统具有更好的反应速度以满足实时性的要求。但操作系统也需要更复杂的同步控制机制。通常的做

法是在内核里设置一些可抢占点,把一些关键的程序片段标志成不可抢占。在这些程序片段里可以安全的操作内核数据结构而不必担心会被切换到其他进程。

随着多处理器及多核结构的出现,操作系统作出改进以使多个进程可以同时在不同处理器/核上运行。早期的 SMP 操作系统不允许进程在内核态同时运行在不同处理器/核上,每次进程进入内核态的时候需要获得一个全局的大锁,在返回用户态的时刻释放这个锁。这样的程序仍然可以获得一定程度在多处理器上的并发性。以后的实现在内核里逐渐增加了用于多处理器同步的同步原语,并且理顺了内核里共享数据结构的处理逻辑,使得内核态也可以在最大程度上达到并行。

简单地说,对同步性的要求如下:单核上运行的不可抢占内核<单核上运行的可抢占内核<多核或 SMP 下的不可抢占内核<多核或 SMP 下的可抢占内核。也就是说,如果一个操作系统在多核或 SMP 下是并发安全的,则它在单处理器系统中必然也是并发安全的,反之则不然。

为了支持多处理器同步操作,处理器必须提供类似 test-and-set 之类最基本的原子操作。

操作系统需要在这些基本的原子操作上构建其他更复杂的同步原语。

3.5.1 同步原语简介

1. 原子操作

内核数据结构经常需要一些简单的更新和改变操作,例如对内核里一些计数器的++、--、赋值。

这样简单的对内存操作在多处理器上也不一定是原子的,在一些处理器结构中需要对齐操作的内存地址并且锁总线来保证操作的原子性,一旦这样的操作满足了处理器的原子性要求,这样的操作就可以成为一种最轻便的同步办法,适用于一些简单数据类型的同步。

例如,对于以下操作:

```
Lock()
++count;
Unlock();
```

如果++count 可以使用 atomic_add(count)来实现,则系统可以更高效的同步。但要注意在很多体系结构下,原子操作(atomic operation)需要锁总线和刷洗处理器的流水线,所以原子操作相对于对应的普通操作通常开销还是要更大一些。

2. 自旋锁

如果某个自旋锁(spinlock)已经被获得,其他试图获得自旋锁的将通过自旋忙等(spin)来等待资源解锁。

实现如下。

```
void spin_lock(spinlock_t *s) {
    while(test_and_set(s)!=0)
        /* empty loop */;
}
```

```
void spin_unlock(spinlock_t *s) {
    *s=0;
}
```

spinlock 只在多处理器/多核环境下有效,在单处理器环境下没有必要使用 spinlock。

事实上,由于 test_and_set(s)在很多体系结构下需要锁住总线,所以出于性能考虑,操作系统中的 spin_lock 往往被实现为

```
void spin_lock(spinlock_t *s) {
    while(test_and_set(s)!=0)
        while(*s!=0)
            /* empty loop */;
}
```

spinlock 是一种轻量级的同步原语,它的获得或者等待只需要很少的指令并不需要发生上下文切换。它的缺点是当发生自旋的时候会占用处理器资源,所以操作系统内核会很频繁地使用它来同步一些基本的数据操作,并且会利用它来保护更上层的同步原语。

当一个线程获得了 spinlock 以后,它不可以阻塞或者睡眠,原因是如果一个线程在持有锁的情况下阻塞,其他线程如果在同一个 CPU 上试图获得这个锁的时候将发生自旋,已经阻塞的线程将永远没有机会醒来,从而发生死锁。

3. 信号量

生产者-消费者问题(也称之为有界缓冲区问题)是计算机科学中关于多进程同步的经典问题。此问题描述了两个分别为生产者和消费者的进程共享了一块固定大小的公共缓冲区。其中,生产者的工作是产生一条数据信息并把它放入到缓冲区中,如此不断地循环。与此同时,消费者会消耗这些数据信息(每次一条)并把它从缓冲区中移出。在整个过程需要满足以下两个条件。

(1) 当缓冲区被填满时,生产者必须停止生产新的信息。

(2) 当缓冲区的内容为空时,消费者必须停止使用缓冲区中的信息。

解决这个问题的办法是当缓冲区被填满时,生产者主动进入睡眠,当下次消费者从缓冲区中移走一条信息以后,再让它唤醒生产者,这样生产者就可以继续向缓冲区中加入新的信息。同样,当消费者发现缓冲区为空时也主动进入睡眠状态,当下次生产者向缓冲区中加入了新的信息以后紧跟着唤醒消费者,从而消费者得以继续使用缓冲区中的内容。

解决生产者-消费者问题的一个主要途径是使用信号量(semaphore)。信号量是 Edsger Dijkstra 于 1965 年最早提出和使用的。信号量创建出一个新的变量类型 semaphore,它记录了缓冲区中元素的个数(当信号量为 0 时表示缓冲区中没有元素),并且规定了对于这个变量的操作必须使用特殊的原子操作来完成即 P 操作和 V 操作。V 是荷兰语中“增加”的首字母,V 操作就表示了对信号量的数值进行增加。相应的,P 操作^①表示对信号量的数值进行减少。信号量的数值(即缓冲区中元素的个数)是不可能为负数的,所以当信号量为 0 时,P 操作就会自动进入睡眠状态,直到随后的 V 操作将它唤醒为止。这相当于在缓冲区为空时,消费者睡眠等待新的信息元素的加入。和有界缓冲区中的生产者不同的是,V 操作在任何时候都不会进入到睡眠状态。其原因是信号量的数值可以表示为足够大,相当于一个无界缓冲区。在 Linux 内核中 P、V 操作对应为 DOWN 和 UP 操作。

```
void down(struct semaphore * sem)
{
    atomic_dec(sem->count);           //信号量里的记数减一
    while(sem->count < 0)              //阻塞等待信号量
        sleep();
}

void up(struct semaphore * sem)
{
    atomic_inc(sem->count);           //信号量里的记数加一
    wake_up a thread,                //唤醒等待在信号量上的一个线程
}
```

当某种资源需要互斥被保护的时候,可以使用一个记数初始为 1 的信号量,在使用资源的前后加上 down 和 up。信号量的优点在于它在获得不到资源的时候会主动放出处理器资源。但这也意味着信号量本身是个更重的同步原语,在资源发生互斥的时候,需要上下文切换。同时信号量本身一般不能被直接映射到处理器支持的原子操作上,通常

^① 在荷兰语中“减少”这个词也是由 V 开头的,为了避免混淆,Dijkstra 创造了一个新的合成词 prolaag 表示“尝试并减少”,P 即为 prolaag 的首字母。

需要更低层的原语(如 spinlock、atomic op)来支持。

4. CPU 独有数据

某些数据如果被分离成每 CPU 独有的数据(percpu data),可以取消锁和其他同步原语。

5. 读写锁

某些同步操作具有比较明显的特征就是对所保护的数据结构读操作的频率远远大于写操作,所以操作系统往往会为某种同步原语提供读写锁的版本。这种锁在被保护的资源同时拥有多个读者时不会发生阻塞或者自旋,只有当有进程试图写所被保护的资源的时候,才会发生。对于具有读密集特性的数据结构,把普通锁改成读写锁可以减少互斥发生的频率。

6. 尝试加锁

有时候系统需要尝试加锁(try_lock)的机制,在试图操作关键数据结构的时候先测试是否能够获得保护这个数据结构的原语,如果不能获得,系统将转到其他执行路径。同样操作系统往往为每种同步锁机制提供了尝试加锁的机制。

3.5.2 死锁

如果进程 1 已经获得 A 锁,想获得 B 锁,同时进程 2 已经获得 B 锁,想获得 A 锁,死锁将发生,进程 1 和 2 将永远没有机会获得两个锁。这种情况通常被称为 ABBA 死锁。避免发生 ABBA 死锁的简单办法是按照一定顺序来获得多个锁。以上的例子,如果规定进程要同时获得 A 锁和 B 锁,必须先获得 A 锁再获得 B 锁,这样可以避免死锁的发生。但在有些比较复杂的代码路径里,很难把代码完全组织成按顺序获得锁,这时候可以利用某些规则(例如锁的内存地址)对锁进行排序,然后使用例如尝试加锁的机制来使代码按照顺序来获得锁。

还有一种死锁的情况是某个进程在已经获得某个锁的情况下重复去获得同一个锁。这在逻辑某些非常复杂的代码内可能发生。一种避免这类死锁的方法是用 t 尝试加锁。如果不能获得锁,则转去执行别的代码。

3.5.3 同步原语

同步原语一般是用来保护数据和资源的,所以在设计同步原语的时候一般把它附在被保护的数据结构上。不同的同步原语有不同的特性。

如果这个功能可以使用原子操作来实现,那么应该优先使用原子操作;占用时间不长

的一些数据结构操作,例如链表插入,可以用自旋锁或者读写锁来保护;占时比较长的,或者需要阻塞等待外部资源的,可以使用信号量;还要注意对有些可能被中断处理程序修改的数据结构的操作需要关中断来保护。

在使用同步原语的时候,需要选择合适的保护粒度(granularity),保护的粒度越大,系统的可并行程度就越低。但另一方面,保护的粒度越细,系统加减锁就越频繁,开销就会变大,同时内存开销也会变大。系统的复杂度和出现死锁代码的危险也会加大。因此,清理代码逻辑来合理调整调锁的粒度,是每个操作系统需要做的事情。

在一个多 CPU/多核的系统中,同步原语被大量使用。使用不当的同步原语将严重影响性能,通常通过分析发生竞争的情景可以对大系统下系统调优有很大的帮助。

一般的多 CPU/多核系统中同步原语对系统性能可能发生的严重影响有以下几种。

(1) 同步原语本身没有发生竞争(contention),但同步原语的使用造成了 cache line 乒乓(bouncing),这时候应该合理安排同步原语在内存中的分布,去除 cache line 乒乓现象。cache line 乒乓效应是指因某 CPU 修改一个存于某 cache line 中的数据,其他 CPU 因此标记此 cache line 为 invalid,而其他 CPU 如果要使用在此 cache line 中的数据时,需要重新读进来,但很快又因为某 CPU 的修改而标为 invalid,称为乒乓(bouncing)效应。

(2) 自旋锁和读写锁一类不放弃 CPU 资源的锁发生了过多的竞争。这种情况下, CPU 使用率会大量上升,但系统的吞吐量反而下降,严重情况下系统甚至会失去响应。

(3) 信号量之类发生竞争以后会主动放弃 CPU 资源的锁发生了竞争,这种情况下系统吞吐量不能上升,但 CPU 总是有很多空闲。

以上情况都应当设法找到竞争最严重的锁,并想办法绕开。

3.6 Linux 下的同步机制

最早的 Linux 是单处理器的内核,当 Linux 刚开始支持多处理器/SMP 的时候,系统在内核态需要获得一把大锁(BKL Big Kernel Lock),随着内核的不断优化,越来越多的地方开始不需要用 BKL,在最新的内核里,只有很少的地方需要用 BKL 来保护了。随着内核的发展,一些复杂而精巧的同步机制被逐渐引入了内核,内核获得了越来越好的可扩展性。

3.6.1 处理器数据

处理器数据(percpu data)严格地说不能被称为同步机制,但 percpu data 的使用是为了解决一些本来需要同步机制来处理的问题,从而提高系统性能。Linux 里典型使用 percpu 数据来处理的数据有伙伴内存分配器(buddy system)里维护的每个 CPU 可分配页的表(pcp list),还有 slab 分配器里的每个 CPU 的可用和已被分配列表,一些内存和中断的统计信息。

根据把原先需要加锁保护的数据结构分离到每个 CPU 数据的思想,每个 CPU 可以在大多数情况全速的处理本 CPU 的数据,不必考虑同步,从而提高系统的可扩展性 (scalability)。

3.6.2 可抢占内核

Linux 2.6 的内核支持抢占,内核利用进程控制结构 (task struct) 里的一个记数 preempt_count 来控制某个进程序在内核态是否可以被抢占。内核可以调用 preempt_disable、preempt_enable 来控制进程能否被抢占。

在内核获得自旋锁、读写锁和关中断时,抢占是被关掉的。

在处理器数据和抢占同时存在的多处理器内核里,有一种比较有趣的情景。

假设有以下程序。

```
DEFINE_PER_CPU(unsigned long, data);
int cpu=smp_processor_id();           //获得当前 cpu 的逻辑 ID
unsigned long mydata=per_cpu(data, cpu); //用逻辑 ID 号取得 cpu 数据
```

这段程序在可抢占的内核里有个问题,假设在取得当前 CPU 的逻辑 ID 和用这个逻辑 ID 去得到 CPU 数据之间发生了抢占,进程在回来的时候已经不在原来的 CPU 上运行,而是被调度到了其他 CPU 上。这样获得的数据就会出错。所以内核在取得 CPU 的逻辑 ID 以前通常要关抢占,然后在使用完 ID 以后再打开。

```
int cpu=get_cpu();
unsigned long mydata=per_cpu(data, cpu);
put_cpu();

get_cpu, put_cpu 是宏,在可抢占内核上:

#define get_cpu()    ({preempt_disable();smp_processor_id();})
#define put_cpu()    preempt_enable();
```

3.6.3 开关中断

内核用 local_irq_disable() 和 local_irq_enable() 来开关中断。local_irq_disable() 不同于 disable_irq(), disable_irq() 通常是对中断控制器 (如 IOAPIC) 编程来组织外部中断的发生。而 local_irq_disable() 通常是通过设置 CPU 寄存器来阻塞中断,当 local_irq_enable 的时候被阻塞中断会重新到达,并不会丢失。

内核用关中断来保护可能被中断处理程序修改的数据。local_irq_save 和 local_irq_restore 和起到类似作用,所不同的是 local_irq_save 和 local_irq_restore 会保存和恢复当前处理器中断的开关状态。

在使用开关中断的时候要注意以下几点。

(1) 长时间的关中断会影响系统的反应性,要尽可能避免,只保护应该保护的数据。

(2) 中断关闭以后不可发生调度。由于关中断默认禁止了抢占,所以在中断关闭以后进程不会被被动地调度出去。但主动地调用 `schedule` 也被禁止,因为 Linux 的调度本身是依赖于中断的,在处理器关中断的情况下调度出去将很有可能使系统没有机会发生下一次调度。

与开关中断类似,内核也提供开关软中断的方法——`local_bh_disable()`,`local_bh_enable()`用来保护有可能被软中断(`softirq`)处理的数据。由于软中断在网络协议栈比较常用,所以在网络协议的代码里可以比较多地看到开关软中断的代码。

3.6.4 原子操作

Linux 内核定义了 `atomic_t` 类型的数据结构,并提供了一系列的原子操作的原语,不同的体系结构利用它们的处理器特性来实现这些原语。比较常用的原子操作原语有 `atomic_read`、`atomic_set`、`atomic_add`、`atomic_sub`、`atomic_add_and_test` 和 `atomic_sub_and_test` 等。内核利用它们来处理简单数据或者利用它们来构造更复杂的同步原语。

注意原子操作虽然非常轻量,但它们通常不是完全没有开销的(for free)。只有在必要的时候才应该使用。

3.6.5 栅栏

Linux 提供一些栅栏(barriers)宏来确保读写内存的顺序。栅栏的语义是阻拦它前面或者后面的读和写内存操作,确保它们在栅栏之前完成或者之后发生。最简单的栅栏宏 `barrier()` 只是个编译器的提示,告诉编译器(gcc)不要在优化的时候调整栅栏前后读写内存的次序。由于很多现代处理器是乱序执行(out-of-order)的处理器,指令的顺序不一定是最后执行的顺序。内核通过一些相关的栅栏宏来解决一些对外设操作和 SMP 情况下乱序的问题。

- `wmb()`。任何在 `wmb` 以前的写操作都必须比 `wmb` 以后的写操作先可见(visible)。
- `rmb()`。任何在 `rmb` 以前的读操作都必须比 `rmb` 以后的读操作先读到数据。
- `mb()`。等于 `wmb()+rmb()`。

在单处理器上,通常对同一内存地址的读写顺序是可以由一些内部硬件机制来保证的。

考虑以下在链表中插入一项的代码。

```
x->next=y-> next;  
y->next=x;
```

如果这两条语句都被映射成了原子操作(这个前提很重要,否则以后的描述将不正确),在单处理器机器上,不牵涉设备 I/O 的时候,处理器通常将保证对 `x->next` 的赋值将比对 `y->next` 的赋值更先可见。这样无论在什么时候代码开始遍历这个链表的时候,要么看到的是 `x` 还没被插入的状态,要么是 `x` 已经插入的状态,所以不会出错。但在一个乱序的多处理器/核的机器上,这个顺序不能得到保证。假设处理器 1 执行以上代码,在处理器 2 看来有可能 `y->next=x` 先可见,而这时 `x->next=y->next` 还不可见,是个无效的地址。这时候处理器 2 遍历该链表将会出错。在内核里可以通过在两条语句之间插入 `wmb()` 来保证一致性。

```
x->next=y->next;
wmb();
y->next=x;
```

这个开锁要比加锁保护这段代码要小得多。

3.6.6 自旋锁和读写锁

自旋锁在 Linux 里被广泛使用,不同的体系实现了它们自己自旋锁和读写锁。虽然不同体系结构的汇编代码差别很大,但基本的代码结构都类似于以下代码。

```
void spinlock_init(spinlock_t *s)
{
    s->lock=0; //初始化为 0
}
void spin_lock(spinlock_t *s)
{
    while(atomic_dec_and_test(&s->lock)!=0)
        while (s->lock!=0)
            /* empty loop */;
}
void spin_unlock(spinlock_t *s)
{
    s->lock=0;
}
int spin_trylock(spinlock_t *s)
{
    return (atomic_dec_and_test(&s->lock)>0); //返回 1 表示失败,0 表示成功
}
```

同样通常读写锁的实现基本类似自旋锁。一般是当 `write_lock` 和 `write_unlock` 的时候原子性地加减或者 `test_and_set` 一个大数,而 `read_lock` 和 `read_unlock` 的时候原子性地加减或者 `test_and_set 1` 来实现 `rwlock` 的功能。

对应可抢占的内核,如果调用 `spinlock` 时的上下文是可抢占,在自旋锁自旋还没有获得锁的时候,进程是可抢占的,一旦进程获得了 `spinlock`,将不可抢占。

由于 `spinlock_t` 和 `rwlock_t` 的长度通常远小于一条 `cache line`,对 `spinlock` 和 `read_lock` 的修改,除了对本 CPU 的运行效率有影响以外,对其他 CPU 也有可能造成 `cache line` 乒乓(bouncing)效应,需要在使用的時候注意。

`spinlock` 和开关中断一起使用的函数如下。

`spinlock_irqsave()`: 关中断,保存中断开关状态,获得自旋锁。

`spinunlock_irqrestore()`: 开中断,恢复中断开关状态,释放自旋锁。

3.6.7 Fast User Space Mutex

Fast User Space Mutex(Futex)是 Linux 2.6 内核提供的原语,它通过一块可以被几个进/线程共享的内存来进行同步。同步通常通过在用户空间操作共享内存来实现,在没有发生竞争时,同步不用进入内核态。只有发生了竞争以后,同步的进/线程才需要进入内核,内核通过维护 hashed 等待队列来维护发生竞争的进/线程序的等待,唤醒操作。新的 glibc 利用 `futex` 来实现用户态的线程锁 `posix_mutex_lock`。

3.6.8 信号量

Linux 内核支持 AT&T UNIX System V(系统五)形式的信号量。它的实现在内核代码中的 `ipc` 目录下。系统五与 POSIX 规范中的信号量不但提供了标准的 UP 和 DOWN 操作,它还提供了一些更高级的功能。例如,非阻塞的信号量操作;信号量自动撤销;受时间限制的信号量操作。

信号量实现的前提条件是原子操作。由于在多处理器的情况下,处理器间的竞争条件是软件无法避免的,所以一般的处理器体系结构中都提供了原子操作的指令,例如 x86 架构中的 `XCHG` 指令和在某些特定的指令前加 `LOCK` 前缀等。有了这些最基本的原子操作,内核就可以用它们来保护临界区间的数据,从而实现信号量。由于信号量需要维护一个计数器来记录信号量的数值以及一些其他需要保护的数据,所以必须使用原子操作来保护它们。在 Linux 内核中,信号量关键数据的保护是借助自旋锁(`spinlock`)来完成的。有了这个原子同步机制,系统五信号量的实现就变得非常简单了,内核只需要有保护的访问和增减信号量的数值,并在尝试减少信号量数值时判断原来的信号量是否为 0,若大于 0 则直接减少信号量的数值;否则,需要把调用该操作的进程放入到睡眠状态。在实现 UP 操作时,若信号量的原始数值为 0,则需要尝试唤醒等待队列中所有阻塞在该信号

量上的所有进程。

Linux 内核不但提供系统五形式的信号量给用户使用,内核自己在某些情况下也需要使用信号量。在内核版本 2.6.16 以后,增加了一种新的数据结构互斥锁(mutex)。所谓互斥锁其实是一个信号量的特例,即信号量的数值仅为 0 或 1 的信号量。因为在内核中绝大多数信号量仅仅是作为一种可以睡眠的锁来使用的,互斥锁的引入在既简化了信号量实现的同时又满足了绝大多数内核用户的需求,使得整个系统更加简单快捷。下面介绍一下互斥锁在内核中的实现。

在 Linux 2.6 内核中,互斥锁定义为如下的结构类型。

```
struct mutex {
    atomic_t count;
    spinlock_t wait_lock;
    struct list_head wait_list;
}
```

在这个结构中,count 是一个原子变量,它用来记录锁的状态。其中,1 代表解锁状态;0 代表加锁状态;负数代表加锁状态并且系统中存在一个或多个正在等待该所锁的用户。wait_list 是一个等待队列,当有调用者的加锁请求无法得到满足时,调用者所对应的进程会被加入到这个等待队列中以便在稍后当锁被其他用户释放时得以被唤醒。wait_lock 是用来保护对这个队列操作的原子性的自旋锁。

内核中提供了如下对互斥锁的原子操作。

1) mutex_init()

初始化 struct mutex 结构。count 初始值为 1,表示解锁状态。

2) mutex_lock()

在大多数情况下,内核的代码尝试请求锁的操作是成功的,其原因是内核代码在编写时就会考虑到尽量避免过多的锁竞争冲突,否则这些锁竞争区域将会成为影响系统整体性能的瓶颈。这是操作系统在设计上所不希望的,所以即使存在这样的区域也早已随着许多年来对系统的优化过程被取代掉了。正因如此,在取得锁的过程中才会出现一条快速路径(__mutex_fastpath_lock())。取锁的快速路径发生在当 mutex->count 由 1 变为 0 的时刻,即在尝试获取锁之前,这把锁正处于打开状态。在快速路径下,取锁操作只需要将 mutex->count 原子减 1 即可,大大地减少了获取锁的操作步骤。若快速路径的条件得不到满足,即此锁先前已被其他进程获取,则取锁操作进入到慢速路径(__mutex_lock_slowpath())。在慢速路径中,首先将调用此操作的进程挂入到 mutex->wait_list 队列中以便将来可以被唤醒。随后,进入到一个无限循环中,这个循环只有在调用进程得到锁(或可中断睡眠的进程收到信号)的时候才会退出。在每次循环的开始,mutex->count 的值都会与 -1 进行原子交换操作。在这里把 mutex->count 的值与 -1 交换的

目的有以下几点：首先，`mutex->count` 的值为 -1 表示锁已被占用，这样其他进程就不会在这个时刻得到锁；其次不与 0 交换的原因是在这里我们已经可以确定至少有一个进程需要获得这把锁（调用进程本身），如果设置成 0 的话，`mutex_lock()` 就进入了快速路径，因而就不会唤醒其他等待锁的进程了。接下来，若交换后所得到的原来 `mutex->count` 的值为 1 的话，说明锁在此时已经被持有者释放掉，那么循环将退出，表示已经成功取得了锁。反之，如果交换后得到的值不是 1，说明锁在此刻依然被其他进程占有，因而调用进程无法获得该锁，需要把当前进程放入到睡眠状态并主动让出 CPU。让出 CPU 的方法比较简单，即首先通过 `__set_task_state()` 把进程的状态 (`task->state`) 设置为 `TASK_INTERRUPTIBLE` 或 `TASK_UNINTERRUPTIBLE`，然后调用 `schedule()` 主动要求 CPU 进行一次调度，这样其他处于就绪状态的进程就有可能被调度并最终执行。

3) `mutex_unlock()`

同样，解锁的过程也存在快速路径。解锁的快速路径 (`__mutex_unlock_fastpath()`) 出现在当 `mutex->count` 从 0 变化到 1 的时刻，即解锁时没有正在等待获得该锁的用户。在快速路径中，解锁操作不需要做例如唤醒进程等的复杂处理，只需要把 `mutex->count` 的值加 1 即可（增加后 `mutex->count` 变为 1）。在具体的实现中内核使用了 `atomic_inc_return()` 函数，从而保证了这个操作的原子性。若快速路径的条件得不到满足（即 `mutex->count` 的值原来不为 0），则解锁操作进入到慢速路径 (`_mutex_unlock_slowpath()`)。在慢速路径中，首先使用 `atomic_set()` 原子操作将 `mutex->count` 的值设置为 1。（在这里，互斥锁其实已经被解开，如果这时候正好有新的获取锁的请求，它是可以被得到满足的）接下来，解锁操作还需要从 `mutex->wait_list` 队列里把所有那些等待获取该锁的进程逐个唤醒，注意，这里是所有，而不仅仅是其中的某一个。原因是唤醒操作需要考虑到机会的均衡，即所有等待获得互斥锁的进程都具有相同的获取锁的机会。而具体哪个进程真正的获得锁是由进程调度器来决定的。具体地讲，当 `mutex_unlock()` 释放锁以后，所有等待获得互斥锁的进程都由睡眠状态进入到了就绪状态。这样，在随后的进程调度中，第一个被调度运行的进程就获得了这把锁（参见 `mutex_lock()` 中的无限循环部分代码，通过把 `mutex->count` 的值与 -1 原子交换后发现其原始值为 1 的那个进程），而其他的进程在被调度到以后仍旧会发现这把锁处于加锁状态（检测 `mutex->count` 发现为 -1），故而依次重新进入睡眠状态。

由此可见，在不发生碰撞的情况下，互斥锁的获得和释放就简化成为对一个原子变量的增减操作，快速路径全部得以应用。这样的优化方法在 Linux 内核中是比较常见的，既对频繁使用的关键路径进行了性能优化又全面地考虑到所有的可能执行条件，这样的做法既提高了系统的性能又保证了程序的正确性。

3.7 Linux 下进程间通信

在 Linux 内核中,进程各自的用户地址空间是相互独立的,当进程与进程之间需要协同工作和交互信息时,内核必须提供一种高效的方法来帮助完成(当然,进程也可以通过文件来传递信息,但由于涉及到磁盘操作所以效率不高),这就是进程之间的相互通信技术——IPC(Inter Process Communication)。Linux 内核支持 POSIX 规范中标准的 IPC 和很多 UNIX 中广泛使用的 IPC 机制。例如,UNIX 中最常见的管道、命名管道、系统五形式的信号量、消息队列和共享内存、POSIX 消息队列以及伯克利套接口等。另外,信号作为一种 UNIX 中常用的异步处理机制,也可被用来作为一种进程间的通信手段。Linux 内核支持 POSIX 规范的信号及信号处理,并在应用中广泛使用。

3.7.1 管道

管道是一种最早的 UNIX IPC 形式,几乎所有的 UNIX 版本都支持这种通信机制。

管道的使用比较简单,但同时它的局限性也是所有 IPC 机制中最大的。例如:

(1) 管道是半双工的,即一条管道中数据只能单一方向流动。

(2) 管道两端的进程必须是父子关系或具有相同的祖先。

管道的最常用的使用模式是:一个进程首先通过系统调用 `pipe(int fd[2])` 创建出一条管道,其中参数 `fd[2]` 是一对文件描述符。当系统调用成功以后(`pipe` 返回值为 0),`fd[0]` 被赋值为管道的读端,`fd[1]` 被赋值为管道的写端。接下来,进程调用 `fork()` 系统调用创建一个新的子进程,由于子进程会继承所有父进程打开的文件描述符,因此 `fd[0]`, `fd[1]` 在子进程中也可以同样使用(如图 3.3 所示)。此时父子进程都同时可以对管道进行读写操作。但是这样的操作会带来问题,例如,如果子进程向管道内写入数据,父子进程都有可能从管道中读出数据(具体哪个进程最终读到数据由读操作的时间和进程调度器来决定)。这是我们所不希望发生的,原因是当我们让子进程进行写操作时一般都是希望父进程能够读到数据,以达到数据通信的目的(子进程内部的数据传递可直接使用内存操作来完成)。所以在这种情况下,我们关闭父进程的写文件描述符,同理也关闭子进程的读文件描述符,如图 3.4 所示。从图 3.4 中,可以看到在父进程到子进程之间形成了一条通道,但这条通道是单方向的,即只有子进程向父进程发送信息,而不能反方向的发送。如果程序需要使用全双工的管道,则还需要再创建一条管道,并把父进程的 `fd[0]` 与子进程的 `fd[1]` 关闭。这样就又创建出来一条从父进程通往子进程的管道。

在 Linux 内核中,管道是借助于管道文件系统(`pipefs`)来实现的。在文件系统的 `inode` 结构中(`struct inode`),有一个指向 `struct pipe_inode_info` 类型的指针 `i_pipe`,这样就把文件系统于管道联系了起来。

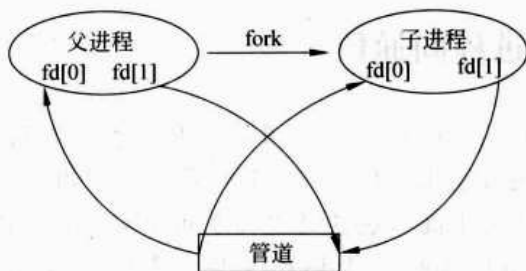


图 3.3 fork 之后的父进程与子进程之间的管道

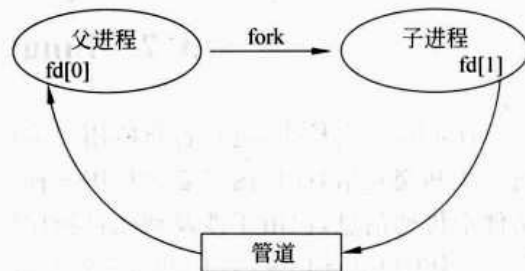


图 3.4 父子进程关闭了相应的文件描述符后的管道

```
struct pipe_inode_info {
    wait_queue_head_t wait;
    unsigned int nrbufs, curbuf;
    struct page * tmp_page;
    unsigned int readers;
    unsigned int writers;
    unsigned int waiting_writers;
    unsigned int r_counter;
    unsigned int w_counter;
    struct fasync_struct * fasync_readers;
    struct fasync_struct * fasync_writers;
    struct inode * inode;
    struct pipe_buffer bufs[PIPE_BUFFERS];
};
```

struct pipe_inode_info 类型是内核中用来表示管道和存放管道信息的数据结构。其中, struct pipe_buffer 类型的数组 bufs[PIPE_BUFFERS](其中 PIPE_BUFFERS 的大小一般定义为 16)用来存放管道的内容,每一个 pipe_buffer 可以存放最大为一个页面的数据(PAGE_SIZE 在 x86 架构下通常情况下为 4k)。nrbufs 表示当前 bufs[] 数组中所使用到的缓冲区个数。curbuf 为起始缓冲区索引。readers 和 writers 表示读写者的个数。

当用户调用 pipe 系统调用时,内核首先在 pipefs 文件系统中创建一个 inode(使用 get_pipe_inode())分配一个新的 struct pipe_inode_info 结构,并将其地址存入 inode->i_pipe 以便将来使用。接着使用 get_empty_filp()分配一个 struct file 结构对其进程初始化,并把 f->f_op 设置为 write_pipe_fops。这个新分配出来的 struct file 结构就是管道的写端。有了管道的写端以后,管道的读端就可以共享其 inode 和 dentry,而只需把读端的 f->op 设置为 read_pipe_fops。最后分配两个文件描述符 fdr、fdw(使用 get_unused_fd()),把它们分别与先前创建的用于读写的 struct file 结构相关连起来(使用 fd_install

()),并把这两个描述符赋值给用户的 fd[0]、fd[1],这样管道就建立成功了。

接下来当用户需要读写管道时,管道的读写描述符就会根据 struct file 的 f->op 指针找到相应的操作函数,具体为 pipe_read()与 pipe_write()。以 pipe_write()为例,首先通过 inode->i_pipe 找到管道所对应的 struct pipe_inode_info 结构,然后检查 pipe->readers 计数。若为 0,则表示管道的读端已经被关闭,这时便使用 send_sig()向当前进程发送一个 SIGPIPE 信号,并且 write 操作返回 -EPIPE;若大于 0,表示读端有进程需要读取信息,pipe_write()会将用户的内容写入到 pipe->bufs 缓冲区中,并且唤醒 pipe->wait 等待队列中的进程(对于那些使用同步的阻塞读取方法的进程),并且向 pipe->fasync_readers 队列的进程发送 SIGIO 信号(对于那些使用异步读取方法的进程)。最后,若需要写入的数据实在太太(超过 16 个页面)那么这些请求没有办法被立刻满足,这时需要等待管道的读端进程取走部分数据后才能继续写入。

3.7.2 命名管道

为了解决管道所存在的只能由相关进程使用、半双工等缺陷,UNIX 操作系统中提出了命名管道(first in,first out,FIFO)的概念。FIFO 是一种特殊文件(类型为 S_IFIFO),它在文件系统中真实的存在。两个或多个进程可以同时打开并读写 FIFO 文件以达到进程通信的目的。因为所有具有 FIFO 文件访问权限的进程都可以对其进行操作,因此并不要求 FIFO 两端通信的进程具有任何联系。此外,进程操作 FIFO 文件的方法与操作普通文件的方法完全相同,那么如果进程选择以读写方式(O_RDWR)打开 FIFO 文件,那么对 FIFO 文件描述符就既可以进行读操作又可以进行写操作(而不是像管道那样只能对 fd[0]进行读操作,对 fd[1]进行写操作),因而 FIFO 是一种双向的进程间通信机制。

命名管道(FIFO)的实现与管道类似,都是通过 pipefs 文件系统来完成的。不同之处在于管道是通过 pipe 系统调用创建的,所以当管道的文件描述符关闭或相关的进程退出后管道也就不复存在了;而 FIFO 是通过 mknod 系统调用创建的,因而在虚拟文件系统中真实的存在了一个 FIFO 文件。当一个用户进程打开一个 FIFO 文件时,文件系统会检查到这是一个特殊文件,这时就将其对应 inode 的文件操作(inode->f_ops)设置为 def_fifo_fops,即操作 FIFO 相对应的方法。其中 fifo_open()用于 FIFO 的打开操作,其实现与管道的创建类似,也是依赖于 pipefs 在 inode->i_pipe 指针找到管道数据结构并分配相应的管道缓冲区。此后对于内核来说 FIFO 和管道的处理就几乎一样了(除了对于文件描述符所对应的读写能力以外,FIFO 的文件描述符可同时被读写,而管道却不可以)。可见,所谓命名管道就是利用文件系统(FIFO 文件的 inode)建立起与管道缓冲区的映射,这样就避免了管道对于父子进程或相关进程的限制。

3.7.3 消息队列

前面我们讲到的管道和 FIFO 提供了进程间通信的机制。但它们的局限性是仅仅提供了基本的通信方法,用户如果要真正使用它们来传送消息,还需要自己定义消息的格式、大小并保证消息的完整性。例如,用户需要自己定义消息的起始标志与结束标志(或消息长度)以便区分不同的消息,用户读取消息的函数还需要保证在一条消息的所有内容都接收完毕以后才将其传递给用户处理等。另外,管道和 FIFO 只能顺序读写,这并不能满足某些用户的按消息的类型操作(消息优先级)的需求。最后,管道要求通信的双方都要同时连接在管道的两端(否则会接到 SIGPIPE 错误信号),这也给消息的传递带来了限制。为了方便以上的应用以及增加消息的分类机制,在 UNIX 系统五中最先引入了消息队列。消息队列可以看作是消息的链表,用来提供进程间简短信息的传递。

Linux 内核实现了系统五和 POSIX 规范两种形式的消息队列。

1. Sys V 的消息队列

消息队列在内核中创建了消息的链表,每个消息队列有一个唯一的队列标识(msgid),应用程序可使用 msgget 系统调用来打开或创建(类似于 open 的 O_CREAT 标志)一个消息队列并得到相应的队列标识。然后发消息送方就可以根据这个标识使用 msgsnd 系统调用发送消息;接收方也可以根据这个标识使用 msgrcv 系统调用来接收消息。消息队列提供的一个重要特征就是消息的分类。在发送消息的时候,可以为每一个消息指定一个消息类型,消息类型是一个大于零的整数。在接收消息的时候,msgrcv 系统调用需要用户指定一个所希望接收到消息的类型(msgtyp)。当 msgtyp 为 0 时,表示用户希望接收到队列中的第一个消息;当 msgtyp 大于 0 时,表示用户希望接收到队列中第一个类型为 msgtyp 的消息(当用户指定 MSG_EXCEPT 标记时,表示用户希望接收到队列中第一个类型不为 msgtyp 的消息);如果 msgtyp 小于 0,那么表示用户希望接收到队列中第一个类型最低并小于等于 msgtyp 绝对值的消息。可见,系统五消息队列消息的分类机制是在接收端实现的。

2. POSIX 消息队列

POSIX 标准定义了另外一种形式的消息队列。与系统五的消息队列相比,POSIX 消息队列增加了如下一些新的特性。

- (1) 简化了的基于文件的接口。
- (2) 增强的消息优先级支持。
- (3) 异步消息机制,并可指定通过信号处理函数或创建新的进程来处理消息。
- (4) 可设置超时的可阻塞的消息发送和接收方法。

应用程序在使用 POSIX 消息队列时,首先调用 `mq_open()` 系统调用来打开一个队列。`mq_open()` 的第一个参数是队列的名称,它是一个类似于文件绝对路径名的字符串,它的作用是为了唯一的标识这个消息队列(回忆系统五的消息队列是由 IPC key 来标识一个消息队列的,而用户通常使用 `ftok()` 函数把一个文件路径名转化为一个 IPC key)。

`mq_open()` 的第二个参数是一个类似于 `open()` 系统调用所使用的标志位,如可使用 `O_CREAT` 来创建一个新的队列等。在打开了一个 POSIX 消息队列以后,应用程序就可以使用 `mq_send()` 和 `mq_receive()` 系统调用来发送和接收消息。它们相应的可设置处理超时的版本为 `mq_timedsend()` 和 `mq_timedreceive()`,用户可以为这些操作设定一个超时时间,若在此限定时间间隔内请求的操作无法完成,则调用进程会从睡眠中退出而不是一直阻塞在等待队列中直到操作被最终完成。最后,在使用完毕 POSIX 消息队列后,用户需要调用 `mq_close()` 系统调用来关闭打开的 POSIX 消息队列。

Linux 2.6 内核对于 POSIX 消息队列的实现是通过 `mqueue` 文件系统来完成的。借助于虚拟文件系统,当 `sys_mq_open()` 处理一个名字为 `name` 的 POSIX 消息队列的打开请求时,它首先把 `name` 作为文件路径名在 `mqueue` 文件系统中找到对应的 `dentry`,若 `dentry` 存在则返回对应的文件描述符;否则根据参数标志位决定是否要在 `mqueue` 文件系统中创建一个新的文件 `inode`。在内核中,POSIX 消息队列的描述符就是用文件描述符来代替的。这样,当某些系统调用如 `sys_mq_send()` 使用队列描述符作为参数时,就可以直接使用虚拟文件系统 `fs` 中的 `fget()` 操作得到此队列在 `mqueue` 文件系统中的 `struct file` 和 `struct inode`,从而取得对应 POSIX 消息队列的相关内容。

通常 `mqueue` 文件系统是不需要被主动安装(`mount`)到系统目录树中的,但如果安装了以后,用户就可以使用一些常用的文件系统操作来操作 POSIX 消息队列。例如,使用 `open()`、`read()`、`write()`、`poll()` 和 `select()` 等对 POSIX 消息队列在 `mqueue` 文件系统中对应的文件进行操作。这是使用文件系统来实现 POSIX 消息队列的另一个优点。

3.7.4 共享内存

系统五的共享内存进程间通信机制通过将同一块物理内存分别映射到不同进程的虚拟地址空间来实现数据的交换。由于数据不需要在进程和进程之间复制,因此在理论上这是一种最快速的 IPC 机制。首先,用户在使用共享内存时需要用 `shmget` 系统调用创建一个新的或取得一个已经建立的共享内存标识 `shmid`(shared memory ID);然后就可以使用 `shmat` 系统调用把由 `shmid` 标识的共享内存片段映射到调用进程的虚拟地址空间中,接下来就可以像操作普通虚拟内存那样操作这块共享内存进程操作。最后在使用完毕后进程还需要调用 `shmdt` 系统调用释放共享内存的映射。若用户进程不主动调用 `shmdt`,系统也会在进程退出时释放进程所有共享内存的映射。注意,这里释放的是共享内存的映射而不是共享内存本身。若需要释放共享内存本身,用户需要 `shmctl` 系统调用

(加命令选项 IPC_RMID)来完成。共享内存虽然有使用简便、传输速度快的优点,但对于给定存储区间的内容却没有任何保护(由于共享内存的访问是与普通内存操作相同的,所以内核无法进行特殊的保护),这就要求对这些数据内容保护的任务必须由调用的用户进程来承担。通常,信号量作为一种常见的同步机制被用来完成对共享内存读写的保护。

Linux 2.6 内核的共享内存是通过 shm 文件系统实现的。如图 3.5 所示,两个进程的 struct vm_area_struct 结构的 vm_file 指针同时指向 shm 文件系统下的 struct file,再由 dentry、inode 找到 address_space 对应的 page cache。这样,就把不同进程的不同虚拟地址空间映射到了同一个(或几个)物理页面当中,从而达到了共享内存的目的。在 Linux 内核实现中,共享内存的内容在必要的时候是可以被放入到交换分区的,这样可以避免大量不常用的共享内存长期占据在系统的物理内存中,从而最大限度的节省了系统的可用内存资源。另外,与普通的内存分配一样,共享内存的分配也是在使用时按需分配的,即用户调用 shmat()时,内核调用 shm 文件系统的 mmap()方法在调用进程的虚拟地址空间分配了相应的内存区域,但并没有建立起页表也没有立即分配具体的物理内存,而直到调用进程第一次真正的使用这块虚拟内存时(处理器会产生缺页异常),内核才会真正的为调用进程分配物理内存并建立起页表。

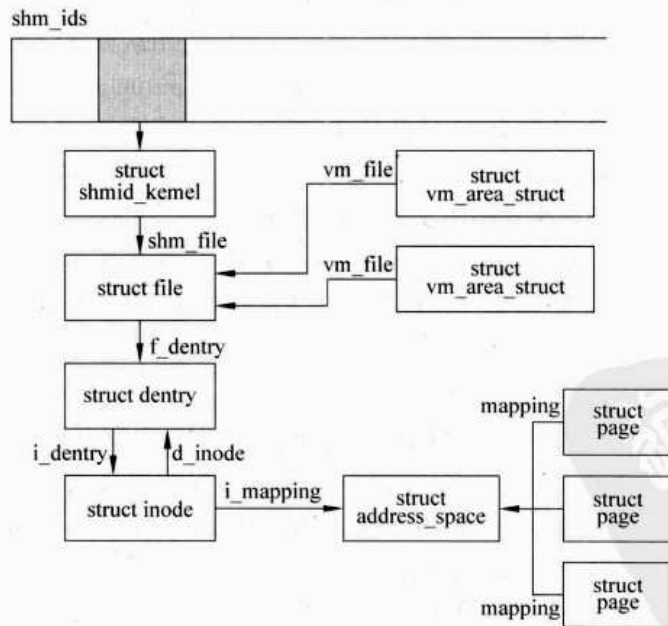


图 3.5 共享内存 IPC 的数据结构

3.7.5 套接口

套结构(socket)最初由 BSD 提出的。套接口不但支持在同一台主机上的进程间通信,而且还支持不同主机上的进程间通信。套接口根据其使用域(domain)可以分为网络套接口和 UNIX 域套接口(UNIX domain socket)。其中网络套接口可用于不同主机上的进程间通信,而 UNIX 域套接口只能用于同一主机上的进程间通信。UNIX 域套接口在通信进程的两端建立了全双工的链路,并且在操作上与网络系统调用兼容,因此也是一种经常使用的本地进程间通信的方式。UNIX 域套接口的操作和实现都与网络套接口相似,不同之处在于 UNIX 域套接口使用一个套接口类型的特殊文件(S_IFSOCK)来作为通信链路的标识(这一点类似于 FIFO),而网络套接口通信链路的寻找需要由网络协议栈来帮助完成。在这里就不具体介绍 UNIX 域套接口了,有兴趣的读者可以参考内核网络部分的实现。

3.8 本章小结

作为系统运行的基本单位,进程在操作系统中几乎无处不在。Linux 内核使用进程描述符描述了进程使用的相关资源,通过设定不同的状态,很好地支持了进程的生命周期。线程是进程的一种自然延伸,提高了系统的运行效率。

在并行机环境里,好的进程调度机制是系统性能优化的基石。Linux 内核对单处理器机器、对称多处理机器和非对称多处理机器都做了精心设计。

锁是进程同步的基本机制。Linux 实现了花样繁多的锁,分别处理不同情形下的同步和互斥。

为了更好的分工协作,操作系统提供了进程通信机制,包括管道、消息队列、信号量和共享内存等。

思 考 题

1. 一个进程可以修改另一个进程的私有数据吗?如果不是,请说明原因;如果是,请举出实际例子。
2. 为了支持多任务操作系统,CPU 至少需要提供什么功能?
3. 在进行同步操作时,怎样在 spinlock 和 semaphore 之间取舍?
4. 有些进程进入内核睡眠后,在用户空间无法用 kill 命令杀死。为什么?
5. 内核是如何帮助 gdb 实现调试功能的?
6. 比较下面代码的输出,解释原因。

```
int main()
{
    printf("hello world");
    fork();
    return 0;
}
```

```
int main()
{
    printf("hello world\n");
    fork();
    return 0;
}
```

进一步阅读

Understanding Linux kernel, 3rd Edition, Daniel P. Bovet, Marco Cesati,
O'Reilly, 2005

UNIX Internals: The New Frontiers by Uresh Vahalia
内核源代码包中的 Documentation 目录。

第4章 内存管理

现代计算机系统的存储通常由内存和外存组成。计算机系统运行时,CPU 直接存取内存中的数据和指令,而外存中的数据则需要首先通过 I/O 设备读取到内存,然后才能由 CPU 使用。内存的访问速度快,但容量小,价格相对较贵,系统断电后数据丢失;外存访问速度慢,但容量大,价格相对便宜,而且断电后数据不会丢失。

尽管内存硬件的成本急速下降,但内存资源仍然是计算机系统中关键且紧张的资源。因此,内存管理是否高效在很大程度上决定整个系统的性能,是现代操作系统中十分重要的功能。而对于外存,因为一般是以文件或设备的形式去使用,所以放在文件系统和设备管理章节中讨论。

4.1 内存管理概述

4.1.1 内存管理的目的和功能

内存管理的主要目的有两个:一是为用户使用存储器提供方便,二是充分发挥内存的利用率。

为用户使用存储器提供方便包含如下两个含义。

(1) 为每个用户程序都提供独立的运行空间,即用户在编程时只需要在各自的逻辑空间内编译生成可执行程序,而不必关心他的程序对物理内存空间的使用情况。

(2) 为用户提供足够大的存储空间,使得用户程序在通常情况下不必考虑实际内存容量的限制。

为了充分发挥内存的利用率,既要为每个用户程序提供足够大的内存空间,使它们得以有效地运行,又要不浪费内存空间,在公平合理的前提下,让尽可能多的用户程序进入内存运行。

为了达到上述目的,内存管理一般需要具有以下几项功能。

- 内存的分配与回收。内存管理记录每个内存单元的使用状态,为运行进程的程序段和数据段等需求分配内存空间,并在不需要时回收它们。
- 地址转换。当程序装入物理内存执行时,如果程序中编译时生成的地址(逻辑地址)与其装入内存的实际地址(物理地址)不一致,必须要把程序中的逻辑地址转换成实际物理内存地址,才能使程序正确执行。这种转换过程称为地址转换,也

称为地址重定位。目前的计算机中,通常由专门的硬件——内存管理单元(MMU,Memory Management Unit)来完成这一功能。

- 内存扩充。用户通常希望自己的程序不会受到物理内存容量的限制。因此,当内存容量不足时,操作系统需要采取某种措施,在不改变实际内存容量的前提下,借助于大容量的外存解决内存不够用的问题。通常采用的手段包括虚拟存储、覆盖和交换等技术。
- 内存的共享与保护。内存共享是指若干个进程能够共同访问内存中某一段内存空间,这不仅能提高内存空间的利用率,而且可实现数据的共享。内存保护的职责就是防止内存中各程序在执行中互相干扰,并保证进程对内存中的信息正确访问。当多道程序共享内存空间时,需要对该空间中的信息进行保护,以保证信息的一致性。例如,多个进程共享内存中某段数据,有的只能读取,而有的不仅可以读取,还可以对它进行写操作。这就需要操作系统能够阻止只有读取权限的进程的写操作。

不同的操作系统的内存管理功能的强弱也大不相同。一般来说,内存管理功能越强,用户越满意,但它对应的管理软件也越复杂,所需硬件支持越多;相反,功能差一些的系统,相应的软硬件开销也较少。因此,到底采用什么样的内存管理方案,取决于机器的结构和服务的对象。

4.1.2 内存分配的方式

在多道程序设计中,内存分配要解决的问题是:什么时候、以什么方式为所运行的程序分配内存空间。通常来说,这一过程对用户来说应该尽可能做到“透明”。内存分配的方式通常可分为静态分配和动态分配两种。

1. 静态分配

所谓静态分配,是指操作系统在装载用户程序时,一次性地为其分配所需的内存空间。如果没有足够多的内存空间,则不能装入该程序。另外,一旦装入内存后,在程序退出系统之前,它将一直占据着分配给它的全部内存空间。在程序运行的过程中,程序的相关代码和数据不能在内存中移动,也不能再申请新的内存空间。

显然,静态内存分配方式使得内存管理变得非常简单,但同时也失去了灵活性,不仅不能有效地利用内存空间,而且不支持动态数据和程序的加载,不能满足动态地增长内存空间的需求。

2. 动态分配

所谓动态分配,是指操作系统在装载用户程序时,只为其分配必需的基本内存空间,而在程序的运行过程中,允许其申请分配新的内存空间,或加载相关的代码和数据。因

此,动态分配是一种更加有效地使用内存的方法。它支持不可预测的分配与释放内存区域的请求,具有更大的灵活性。

动态内存分配方式支持动态数据结构与动态程序结构(如递归数据、递归程序),不必让整个进程占据一个连续的存储区域,也不必一次全部装入内存。现代操作系统通常采用这种内存分配方式。

4.1.3 重定位

在讨论重定位前,首先介绍两个概念:逻辑地址空间(简称地址空间)和物理内存空间(简称内存空间)。

1. 逻辑地址空间

在用汇编语言或高级语言编写的程序中,是通过符号名来访问子程序 and 数据的,通常把程序中由程序员建立的符号名的集合叫做“名字空间”。汇编语言源程序经过汇编,或者高级语言程序经过编译,得到的目标程序是以 0 作为参考地址的模块。然后多个目标模块由连接程序连接成一个具有统一地址的装配模块,以便最后装入内存中执行。目标模块中的地址称为逻辑地址(或相对地址),而把逻辑地址的集合叫做“逻辑地址空间”或简单地叫做“地址空间”。

装配模块虽然具有统一的地址空间,但编译程序对一个源程序编译时,通常总是以 0 作为起始地址,其他地址都是以此顺序安排下来的,即是浮动的,如图 4.1 所示。显然,地址空间中的所有地址都是相对于起始地址的,因此称它们为相对地址。所以,逻辑地址又叫“以零为起始的相对地址”。

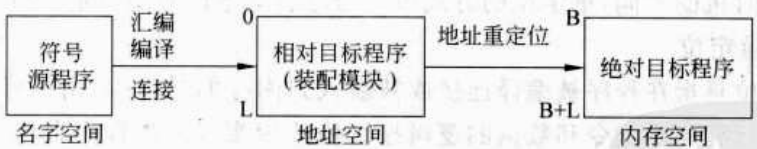


图 4.1 程序的名字空间、地址空间和内存空间

2. 物理内存空间

所谓物理内存空间,是指内存中一系列物理单元的集合,又称物理空间或内存空间。这些单元的编号,称物理地址或绝对地址。物理地址是从地址 0 开始统一编号的,因此,物理内存是一维的线性空间。物理空间的大小,是由内存的容量决定的,如图 4.1 所示。总之,地址空间是逻辑地址的集合,物理空间是物理地址的集合。

3. 重定位的时机

将逻辑地址空间重定位到物理地址空间的时机有以下三种。

(1) 程序编译连接时。

如果在编译时就可以知道程序运行时将处于内存的什么位置,那么就可以产生绝对地址。例如,如果预先知道一个用户进程的驻留地点将从 B 开始,那么产生的编译代码将从这个地址开始并依次向后延伸。

在程序编译时确定的绝对地址,如果日后程序在物理内存中的开始地点有所变动,那么就必须重新编译。MS-DOS 的 .COM 格式的可执行程序就是在编译时确定的绝对地址。

(2) 程序装入内存时。

如果在编译时不能知道进程将处于内存的什么位置,那么编译程序就必须产生可重定位代码。在这种情况下,最终的地址重定位推后到程序装入内存时。如果开始地址有所变动,那么只需要利用变动后的地址重新装入程序代码即可。

(3) 程序执行时。

如果在程序编译时和程序装入时都不进行地址重定位,而将其推后到 CPU 执行程序时,那么,在进程执行期间就可以把它各个段的内容在内存中灵活地放置或移动,这显然为多道程序系统中的内存使用提供了极大的方便。

在这种情况下,当 CPU 用到一个逻辑地址时,需要快速地将其转换为对应的物理地址,因此需要特殊的硬件支持(如 i386 的 MMU)。大多数现代操作系统采用这种方法。

4. 重定位的方式

按重定位时机的不同,重定位的方式可分为静态重定位或动态重定位两种。

(1) 静态重定位。

静态重定位是指在程序被编译连接或被装入到内存的过程中,由编译连接程序或装入程序将可执行程序中指令和数据的逻辑地址修改为实际的物理内存地址。显然,程序编译连接或程序装入内存时的重定位都属于这种情况。

静态重定位是由专门设计的重定位装入程序或重定位连接程序完成的。对每个程序来说,这种地址变换仅在程序连接或装入时一次完成,而在程序运行期间则不再进行重定位,也不允许对程序位置做任何移动。图 4.2 展示了程序 A 在静态重定位前后代码中有关地址的变换情况。

从图 4.2 中可以看出,经过静态重定位,程序中 100 号单元(逻辑地址)中的指令 LOAD 1500 放到了内存 5100 号单元(物理地址)中,该指令中的相对地址 500 相应地变换成 5500。在程序运行时,CPU 从绝对地址 5500 号单元中取出数据 12 345,装入寄存器 1 中。如果现在将程序 A 装入到内存单元 9000~9700 的空间中,那么,经静态重定位,原

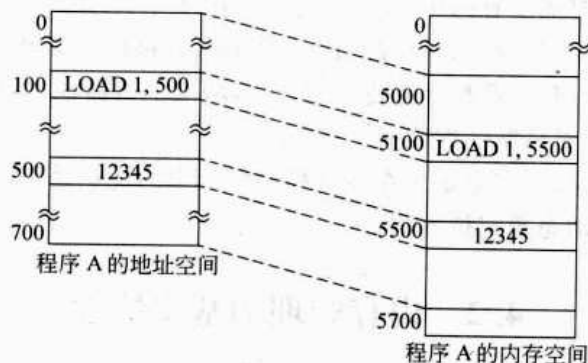


图 4.2 静态重定位示意图

来 100 号单元中的指令 `LOAD 1, 500` 放到内存 5000 号单元, 该指令中的相对地址 500 将相应地变成 5500。依次类推, 程序中所有与地址有关的量都要作相应地变换。

静态重定位的优点是无须增加专门的硬件地址转换机构, 便于实现程序的静态连接。在早期计算机系统中大多采用这种方案。

静态重定位的主要缺点有两个: 一是程序的内存空间只能是连续的一片区域, 而且在重定位之后就不能移动, 这不利于内存空间的有效使用; 二是各个用户进程很难共享内存中的同一程序的副本。

(2) 动态重定位。

动态重定位是在程序执行期间进行的重定位, 通常是通过硬件地址变换机构实现的。在程序执行时, 每当 CPU 访问内存中的指令或数据之前, 由硬件地址变换机构将逻辑地址转换为物理地址。动态重定位的过程如图 4.3 所示。

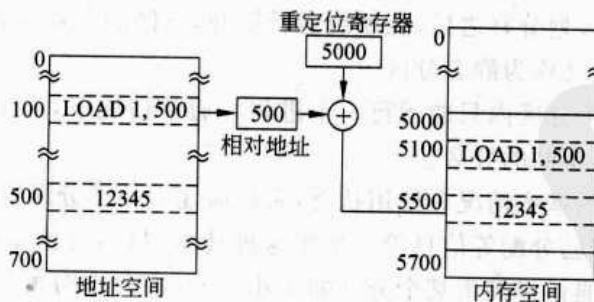


图 4.3 动态重定位示意图

硬件地址变换机构采用一个重定位寄存器, 放置当前执行程序在内存空间中的起始地址。如果用 (BR) 表示重定位寄存器中的内容, 用 $addr$ 表示操作对象的逻辑地址, 则操作对象的物理地址就是 $(BR) + addr$ 的值。

动态重定位的主要优点有两个：一是程序占用的内存空间动态可变，也不必连续存放在一起；二是比较容易实现几个进程对同一程序副本的共享使用。

动态重定位的缺点是：需要附加硬件的支持，增加了机器成本，而且实现内存管理的软件算法也比较复杂，对系统的性能有一定影响。

同静态重定位相比，动态重定位在多道程序系统中的优点是明显的。因此，现在的计算机系统中都采用动态重定位方法。

4.2 内存管理的基本技术

内存管理的基本功能是为待运行的程序分配内存，并在使用结束后回收这些内存。本节首先介绍一些基本的内存管理技术，包括分区法、可重定位分区法、覆盖和交换等技术。

4.2.1 分区法

分区法是一种最简单的内存管理方式。在这种方式下，除操作系统占用内存的某个固定分区（通常是低地址部分）外，其余的内存由各应用程序共享使用。分配的方式是将内存划分为若干块（称为分区），用于存储各程序的代码和数据。

按分区划分方式，分区法可分为两种：固定分区法和动态分区法。

1. 固定分区法

固定式分区法是指在程序装入内存之前，就将全部内存划分成若干固定大小的分区（每个分区大小固定，但不同分区大小可以不同）。这一划分可由操作人员完成，也可由操作系统完成。一旦分区划分好之后，在系统运行期间，分区的个数和各个分区的大小不再改变。因而，这种分区又称为静态分区。

固定分区法在每个分区内只能运行一个进程。通常情况下，为了适应进程大小的差异，将内存划分成大小不同的分区。

为了记录各分区的基本情况和使用状态，系统应建立一个分区说明表，其中包括分区大小、起始地址和是否已分配等信息等。采用这种技术，虽然可使多个作业共驻内存，但一个作业的大小不可能正好等于某个分区的大小，于是在每个分配的分区中很可能有部分内存被浪费。

固定式分区实现技术简单，所需操作系统软件和处理开销都小，但其内存利用率不高，仅适用于作业的大小和数量都事先比较清楚的系统中。

固定分区法曾用于早期的 IBM 大型机操作系统 OS/MFT (Multiprogramming with a Fixed number of Tasks)。现在几乎没有哪一种操作系统单独采用这种内存管理模式。

2. 动态分区法

动态分区法是指在装入作业时根据其实际内存容量的需求,动态地为作业划分一块连续的内存空间(分区)。也就是说,分区的大小是在作业装入到内存时确定的,其大小恰好等于作业的大小。IBM 的 OS/360 MVT(具有可变任务数的多道程序设计)操作系统采用这种技术。

在动态分区法中,系统同样需要设置一种数据结构来记录各分区的基本信息和使用情况,包括各个分区的大小、位置和对各个分区的存取限制等。常用的数据结构有空闲分区表和空闲分区链两种。

(1) 空闲分区表。

图 4.4 给出了一个空闲分区表的例子。内存中每一个未被使用的分区对应该表的一个表项,一个表项中包括分区序号、分区大小、分区的起始地址及该分区的使用状态。

序号	分区大小/KB	分区始址/KB	状态
1	64	44	可用
2	24	132	可用
3	40	210	可用
4	30	270	可用
5	...	...	...

图 4.4 空闲分区表

(2) 空闲分区链。

空闲分区链是使用链指针把所有的空闲分区链接成一个链表。为了实现对空闲分区的分配和链接,在每个分区的起始部分设置状态位、分区大小和链接各个分区的前向指针,由状态位指示该分区是否已经分配;同时,在分区尾部设置一个后向指针,用来链接后面的一个分区;分区的中间部分是用来存入作业的空闲内存空间,当该分区分配出去后,状态位就由 0 置为 1,如图 4.5 所示。

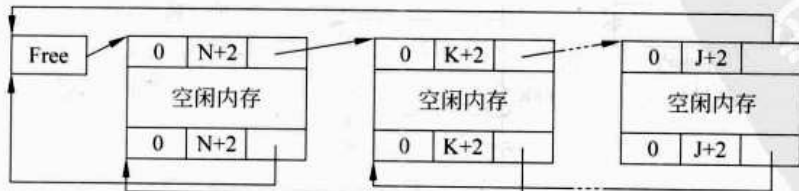


图 4.5 空闲分区链

采用动态分区法,当系统初启时,内存中除了常驻的操作系统外,其余的是一个完整的大空闲区。当作业申请装入内存时,系统就从空闲表(链)中查找一个足以容纳这个作业的空闲区。如果大小恰好一样,则把该分区分配给这个作业使用,并相应的修改空闲分区表(链)中的状态位。如果这个空闲区比作业需要的空间大,则将该分区划分为两部分:一部分给作业使用;而另一部分设置为较小的空闲区加入到空闲分区表(链)中。当一个作业完成后,其所释放的分区将与其相邻的空闲分区合并起来,组成一个更大的分区。

4.2.2 可重定位分区法

虽然动态分区法比固定分区法的内存利用率高,但动态分区法有零头(碎片)的问题。碎片是不连续的小块空闲区,可能所有碎片的总和超过某作业的容量要求,但由于不连续也无法分配。解决碎片问题的方法是拼接,即当出现上述这种无法分配情况时,向一个方向(例如向低地址端)移动已分配的作业,使那些零散的小空闲区在另一方向连成一片,这就是“紧缩”技术。

在紧缩过程中,因为内存中已经存在的作业需要“移动”,因而其中所有关于地址的项均需得到相应的修改,也就是需要进行地址重定位。因此,将这种分区方法称为可重定位分区法。

动态重定位技术的系统开销很大,通常使用硬件实现。硬件支持包括一对寄存器,其中一个用于存放用户程序在内存中的起始地址,即基址寄存器;另一个用于表示用户程序的逻辑地址的最大范围,即限长寄存器。图 4.6 是一个动态重定位的例子。从图中可以看出,当作业装入内存后,其内存空间中的内容与逻辑空间中的内容是相同的,也就是说,

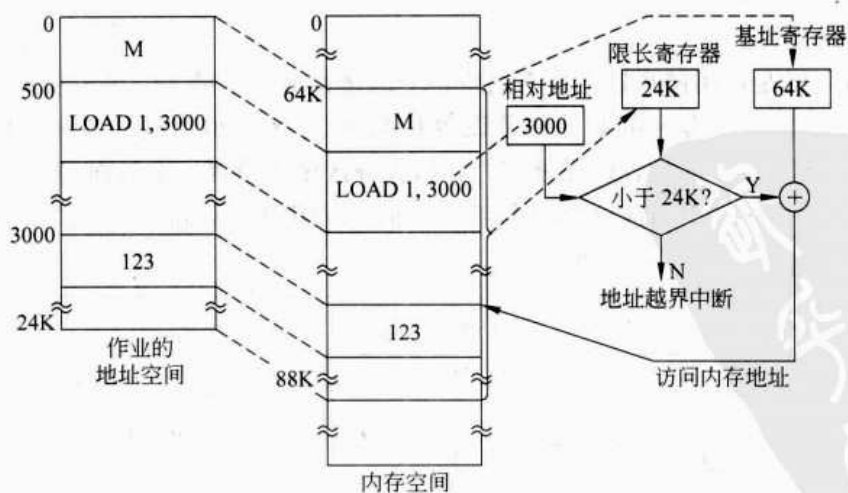


图 4.6 动态重定位的实现过程

装入程序在装入过程中不需要进行任何地址相关的修改和变换。当该作业被调度到 CPU 上运行时,操作系统就将该作业在内存的起始地址 64KB 装入基址寄存器。当执行指令 LOAD 13000 时,硬件首先会将相对地址 3000 与限长寄存器中的长度 24KB 进行比较,如果超长则激发相应的异常,并进行处理。如果在限定范围内,则将两个寄存器中的内容自动相加,并正确的将内存地址中的内容 123 加载到 1 号寄存器中。

可重定位分区法与动态分区法的分配算法基本上相同,差别仅在于:在这种分配算法中,增加了“紧缩”功能,通常是在找不到足够大的空闲分区来满足用户需求时,才进行紧缩处理。常用的分区分配算法主要包括以下几种。

(1) 最佳适应算法。

在这种算法中,将空闲块按其规模从小到大递增的顺序组成链。用户提出申请时,分配程序按链查找到适合用户需求的块,必定是最接近用户申请量的块(相对后面的块而言),因而称为最佳适应算法。

该算法将按用户申请量进行分配,并把残余部分留在链中,但需要重新修改分区的起始地址和大小,并排序其位置。回收时,要按回收块(或者合并后的块)大小,依次与空闲链中各块比较,直到找到一个适合的位置并放入。

这种算法的优点是便于释放内存时进行合并,且为大作业预留了大块的空闲块。缺点是会出现很小的空闲块,分配和回收时都需要进行排序,影响效率。

(2) 最坏适应算法。

在这种算法中,空闲块按其大小递减的顺序组成链,即大块在先,小块在后。分配时总是先分配最大的空闲块,可使剩余空间不会太小。但是,由于最大的空闲块总是首先被分配出去,当有大的程序时,其存储空间的申请往往得不到满足。当回收时也因需要按大小寻找回收块在链中的适合位置,因而也相当费时。

最坏适应算法对较小的程序有利。

(3) 首次适应算法。

由以上讨论可见,最佳适应与最坏适应算法各有利弊。首次适应算法是对它们进行折衷考虑后设计出来的。这里,每个空闲块按其在存储空间中地址递增的顺序链在一起,即每个后继空闲块的起始地址总比前边的大。

在为程序分配存储区域时,从这个空闲块链的始端开始查找,找到第一个足以满足请求的空闲块时,就停止查找,而不管它究竟有多大。和上述算法一样,这个被选中的空闲区被分成两部分。一部分与请求的大小相等,分配给程序;剩下另一部分留在空闲块链中。显然,这个算法倾向于优先利用存储空间中低地址部分的空闲块。它的主要优点是算法简单,查找速度快。此外,留在高地址部分的大空闲块被划分的机会较少,因而当大程序到来时也比较容易得到满足。

当然,这种算法常会利用一个大的空闲块来响应小程序的请求,从而留下一些较小的

空闲块。由于所有的请求都要从空闲块链的起始端开始查找,而这些小的甚至无用的空闲块又恰恰集中在链的前部,相应地一些大的空闲块则多在链的尾部才能发现,因此这种情况将导致合适空闲块的查找速度降低。

(4) 下次适应算法。

这种算法是首次适应算法的变形。在为作业分配内存空间时,是从上次所分配的空闲区的下一个空闲区开始查找,直到找到第一个能满足要求的空闲区,从中划出一块与请求的大小相等的一块存储区分配给作业。在该算法中应采取循环查找方式,即最后一个空闲区的大小仍不能满足要求时,应再从第一个空闲区开始查找,所以这种算法又称为循环适应算法。

该算法能使内存中的空闲块分布得更均匀,减少查找空闲块的开销。在实现时设置一个指针,用于指示下一次搜索的起始位置。它的缺点是同样无法为大作业预留大的空闲块。

4.2.3 覆盖技术

在早期的操作系统中,为了允许一个作业能够使用大于它所能获得的内存容量,可以使用覆盖技术。覆盖技术的思想是把大的程序划分为一系列的覆盖(程序段),每个覆盖是一个相对独立的程序单位,把程序执行时不需要同时装入内存的覆盖构成一组,称之为覆盖段。这个覆盖段中的覆盖都分配到同一个内存区域,这个内存分配区域称为覆盖区,它与覆盖段一一对应。并且,为了使覆盖区能被相应的覆盖段中的每个覆盖在不同时刻共享,其大小就由其中最大的一个覆盖来确定。

覆盖技术无须操作系统的专门支持,但要求程序员必须把一个程序划分成不同的程序段,并规定好它们的执行和覆盖顺序,操作系统根据程序员提供的覆盖结构来完成程序段之间的覆盖。因此要求程序员明确地描述作业中各个程序模块间的调用关系,这无疑给程序员提出了很高的要求。因为如果程序很大(小程序无需使用覆盖),完全理解程序将会非常困难。因此,覆盖技术的使用通常限于微型计算机和其他物理内存有限而且缺乏支持更高级技术硬件的系统中。有些微型计算机编译程序为程序员提供了对覆盖技术的支持。

图 4.7 是关于覆盖技术的一个例子。从图中可以看出,程序段 B 不会调用 C,程序段 C 也不会调用 B,因此,B 和 C 不需要同时驻留内存它们可以共享同一内存区;同理,程序段 D、E 和 F 也可共享同一内存区。整个程序段被分为两部分:一部分是常驻内存的,称为根程序,它与所有的被调用程序段有关,因而不能被覆盖,图中的程序段 A 是根程序;另一部分是覆盖部分,被分为两个覆盖区,一个覆盖区由程序段 B 和 C 共享,其大小为 50KB,另一个覆盖区由 D、E 和 F 共享,大小为 40KB。

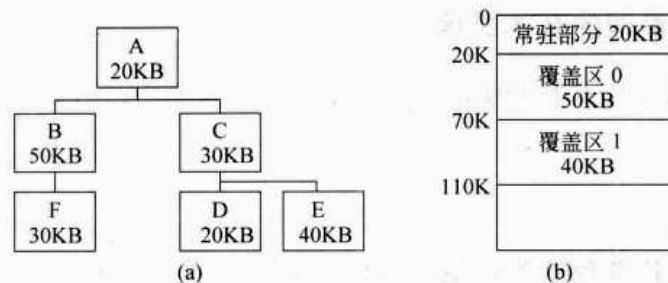


图 4.7 覆盖技术的例子

4.2.4 交换技术

所谓交换,就是允许把内存中的一个作业的一部分或全部从内存转移到外存中去,以便腾出必要的内存空间;当需要用到换出到外存上的作业内容时,再将其从外存读入到内存中,并将控制权转给它。因此,交换技术实际上是利用外存作为内存缓存的一种内存扩充技术。

交换技术由换出和换入两个过程组成。换出服务程序根据系统资源包括内存的使用情况来控制各作业的调入调出。当前运行的作业在用完时间片或因 I/O 请求被阻塞时就可以换到外存上,而把外存上准备运行的作业调入。这种对象可使系统资源的利用更为充分有效。交换技术一般都有动态重定位机构的支持,因而一个作业换入内存时不一定要装入到它被换出前所占据的内存区域。

与覆盖技术相比,交换技术的特点是交换过程由操作系统完成,对用户透明,但需要较多的软件支持。早期的分时系统(如 MIT 的 CTSS)采用这种技术。

4.3 分页技术

无论是分区技术还是交换技术,均要求把一个作业放在一片连续的内存区域中,从而造成内存中的碎片问题,作业的大小也受到了限制,而且分区式管理也不利于作业程序段和数据的共享。解决这种问题的方法有两种:一种是前面讲的紧缩法,即通过移动作业,使空闲块变成连续的较大的块,从而得以利用,但这需要较大的 CPU 开销;另一种是分页技术,它允许程序的内存空间不一定连续,这样就可以把一个程序分散地放置在各个空闲的物理块中。因为在程序运行时无须移动内存中的已有信息,所以分页技术既解决了(作业)外部碎片的问题,(相对紧缩技术)又提高内存的管理效率。

4.3.1 分页内存管理的基本原理

与前面介绍的几种技术不同的是,分页技术不仅将物理内存空间分块,同时也将程序的逻辑地址空间分块。

1. 页和页面

在分页技术中,各进程的逻辑地址空间被划分成若干个长度相等的块,称为页(page)。页长的划分和内存外存之间数据传输速度及内存大小等有关。一般每个页长大约为1~4KB,经过页划分之后,进程的逻辑地址变为页号 p 与页内地址 w 所组成,页号和页内地址都从0开始编号。例如,一个页长为4KB,拥有 2^{20} 页(32位)的逻辑空间地址结构如图4.8所示。

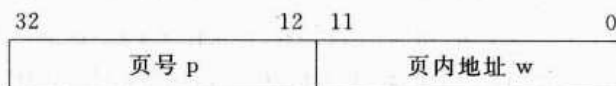


图 4.8 页技术的 32 位地址结构

除了把进程的逻辑地址空间划分为等大小的页之外,分页技术还把内存空间也按页的大小划分为等大小的块,称为页面(page frame)或物理页。在权限许可的情况下,这些页面可以被系统中的任一作业所共享。

在这种架构下,用户进程在内存空间内除了在每个页面内的地址连续之外,每个页面之间不再连续。这样做有两大好处:第一,实现了内存中碎片的减少,因为任一碎片都会小于一个页面;第二,实现了由连续存储到非连续存储的飞跃,为在内存中局部地、动态地存储那些反复执行或即将执行的程序和数据段打下了基础。

2. 页表

那么,如何由页式逻辑地址变换为内存页面的物理地址呢?分页技术把逻辑页号与内存页面物理页号建立一一对应页表(如图4.9所示),并用相应的硬件地址变换机构,来解决逻辑地址到物理地址的变换问题。

在进程开始执行之前,把该进程的程序段和数据全部装入内存的各个页面中,并通过页表(page mapping table)和硬件地址变换机构实现逻辑地址到物理地址的地址映射。

页表在内存中占有一块区域,其大小由进程的长度决定(有些操作系统中采用固定大小的页表)。例如,对于一个每页长4KB,大小为20KB的进程来说,如果一个内存单元存放一个页表项,则只要分配给该页表5个存储单元即可。显然,分页技术中的每个进程至少拥有一个页表,在切换进程时需要切换页表。

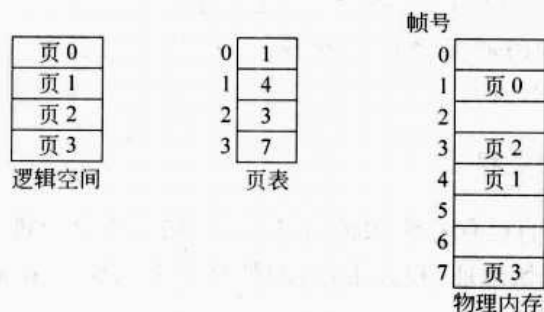


图 4.9 分页技术中页表的作用

3. 空闲物理内存页面管理

空闲物理内存页面可采用位示图和空闲页面链两种管理方法。

位示图是在内存中划分一块区域,每个单元的每个比特代表一个页面。如果该页面已被分配,则对应比特位置 1,否则置 0,如图 4.10 所示。

32	31	30	29	28	...	4	3	2	0
0	1	1	1	1	...	1	1	0	1
0	0	0	1	1	...	0	0	1	1
0	0	1	1	1	...	0	0	0	0

图 4.10 物理页面位示图

位示图要占据一部分内存容量。例如,一个划分为 1024 个页面的内存,如果内存单元长 32 比特,则位示图要占据 $1024/32=32$ 个内存单元。

在空闲页面链中,队首页面的第一个单元和第二个单元分别放入空闲页面总数与指向下一个空闲页面的指针。其他页面的第一个单元中则分别放入指向下一个页面的指针。空闲页面链的方法由于使用了空闲页面本身的单元存放指针,因此不占据额外的内存空间。

4.3.2 分页技术中的地址映射

在采用分页技术的系统中,程序中给出的地址是逻辑页号和页内偏移,如何将这种逻辑地址在运行时快速映射到正确的物理地址是决定系统性能的一个关键问题。这是因为

页表是存储在内存中的,CPU 访问内存的速度相对较慢,因此,地址映射的速度是分页技术中设计地址映射机构时必须考虑的问题之一(目前大多数系统中这一映射由硬件 MMU 实现,无需 CPU 处理)。

1. 基本的地址映射过程

每个操作系统都有自己保存页表的方式,大多数是为每个进程分配一个页表,并在 PCB 中存放该页表的起始地址(和大小,有的处理器上的操作系统使用的页表都是等大小的,所以不必存放页表大小)。

在分页技术的地址映射过程中,需要一组硬件(包括一些寄存器)的支持。在开始地址映射时,首先需要能够找到该进程的页表。通常有利用一个硬件寄存器来存储正在运行进程的页表起始地址(和页表长度),称为页表寄存器。在调度一个新的进程运行时,操作系统把调度到要执行的进程页表始址(和长度)设置到控制寄存器中。

图 4.11 所示的是一个从逻辑地址映射到物理地址的转换过程。在这个例子中,进程有 3 个逻辑页,页号分别是 0,1,2,其对应的物理页面号分别为 2,3,8。设每个页面长度为 1KB,指令 LOAD 1,2500 的逻辑地址为 100。

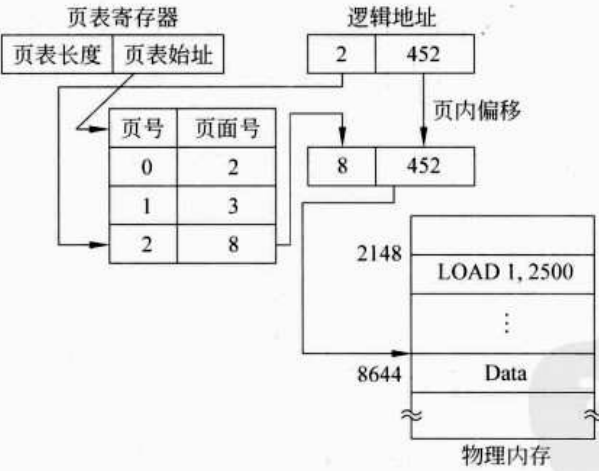


图 4.11 分页技术的地址映射过程

由页表寄存器中存放的页表始址,找到页表所在位置。由指令的逻辑地址 100 可知,指令 LOAD 1,2500 在第 0 页的第 100 单元之中。由于第 0 个逻辑页与第 2 个物理页面相对应,因此,该指令在物理内存中的地址为 $2048 + 100 = 2148$ 。

当 CPU 执行到第 2148 单元的指令时,CPU 要从逻辑地址 2500 中取数据放入 1 号寄存器中。为了找出 2500 对应的实际物理地址,地址映射机构首先将 2500 转换为页号

与页内相对地址组成的地址形式。即 $p=2, w=452$ 。

由页表可知第 2 个逻辑页所对应的物理页面号等于 8。最后,将页面号 8 与页内相对地址 $w=452$ 相连,得到待访问的物理内存地址 $8 \times 1024 + 452 = 8644$ 。其映射过程由图 4.11 所示。

上述地址映射过程通常全部由硬件地址映射机构 (Memory Management Unit, MMU) 自动完成, CPU 不参与地址映射, 这样可降低 CPU 负载, 提高系统性能。

2. 基于快表的地址映射

由于页表是驻留在内存中的, 而取数据或指令又必须经过页表变换才能得到实际物理地址。因此, CPU 从内存中取一次数据或指令至少要访问内存两次以上 (至少一次访问页表, 另一次是取数据或指令), 这比通常执行指令的速度慢了一倍以上。

为了提高查找页表的速度, 有两种办法: 一种最直观的办法就是把页表放在寄存器中而不是内存中, 但由于寄存器价格太贵, 这样做是不可取的; 另一种办法是在地址映射机构中加入一个高速、小容量的联想存储器, 构成一张快表 (translation lookaside buffer, TLB)。快表相当于页表的专用高速缓存, 其中缓存 (正在运行) 进程最近用到的几个页号和页面号的对应关系, 所以快表的每项包括页号 (称为键值) 和页面号 (称为值) 两部分。

当需要进行页号映射时, MMU 首先拿逻辑页号到快表中查找是否有对应的项, 如果有, 则直接返回其对应的页面号, 并利用页面号最终形成物理地址。如果在快表中没有查到对应项, 则还需要到内存中查找页表, 当然, 从页表中查到的新的页号和页面号的对应关系也会缓存到快表中。如果快表中没有空闲项, 则需要采用一定的置换策略, 选定并用新项替换其中一个已经存在的项。置换的策略有多种, 可以采用后面将要介绍的页面置换算法的任一种。图 4.12 是基于快表的地址映射过程。

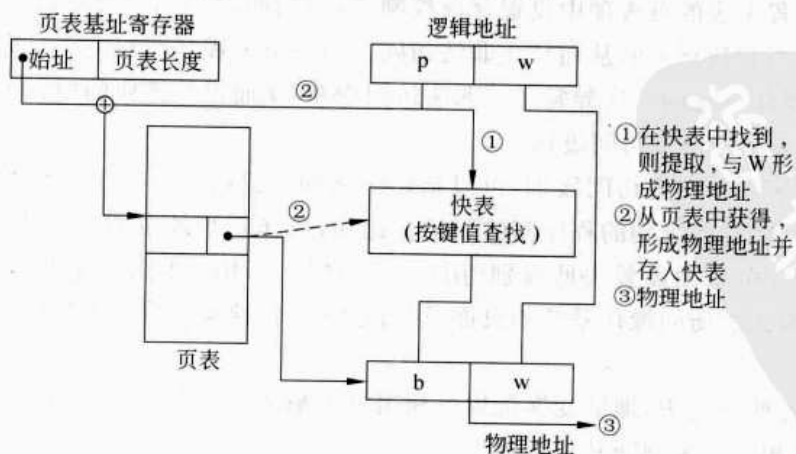


图 4.12 基于快表的地址映射过程

有些快表在每项中还设置了地址空间标识符(Address-Space Identifier, ASID),每个进程对应唯一的 ASID,用来保护该进程的地址空间。当利用快表将页号映射到内存块号时,需要保证当前运行进程的 ASID 与该页面的 ASID 相匹配。如果不匹配,则表示快表未命中。因此,利用 ASID 可允许快表中同时包含若干不同进程的映射表。如果快表不支持 ASID,那么每当进程切换时,就需要相应的刷新快表,以保证运行进程取得正确的地址转换信息。

因为寄存器式的快表的成本很高,所以项数一般很少,通常只有 8~32 项。在快表中成功查找到指定页号的次数占总查找次数的百分比称为命中率。随着快表项数的增加,命中率也会提高。当然命中率还与程序的结构和快表的置换算法有关。

分页技术解决了分区管理时的碎片问题。但是,由于静态分页技术要求进程或作业在执行前全部装入内存,如果可用页面数小于用户要求时,该作业或进程只好等待。而且,作业或进程的大小仍受内存可用页面数的限制。这些问题将在虚拟(请求)分页技术中解决。

4.3.3 页面保护方式

在分页系统中,多个进程所用的内存页面呈交错分布状态。为了防止不同进程间的非法访问以及本进程对自己地址空间中的数据的误操作,需要对进程的页面提供相应的保护措施。分页系统中提供的内存保护方式有以下三种形式。

(1) 利用页表本身进行越界保护。

每个进程有自己的页表,页表的基址信息存放在该项进程的 PCB 中。每个进程运行时使用其自身的页表,这样通过页表的地址映射就只能访问到进程自己的存储空间。

(2) 在页表项中设置存取控制位。

通过在进程页表的页表项中设置存取控制位,可指明对应内存页中的内容允许执行何种操作(读、写和执行等),从而禁止非法访问。当一个进程试图写一个只允许读的内存页时,会引起操作系统的一次异常——非法访问中断,从而保护该块的内容不被破坏。

(3) 在页表项中设置访问级别。

通过在页表项中设置访问级别,可以指定当访问本页时,CPU 所需要的最低运行级别,从而实现不同运行级别的程序间的保护。比如,操作系统的页面设置为高级别(内核态),而用户程序的页面设置为低级别(用户态),这样,当用户程序运行时,由于 CPU 处于用户态,从而无法访问操作系统的页面,这就达到了保护操作系统不被用户程序影响的目的。

另外,在有些系统中,地址变换机构可利用页表基址寄存器中的页表长度值和所要访问的逻辑地址相比较来判断其是否越界。

4.4 分段技术

在分区法和分页技术中,提供给用户程序的逻辑地址空间都是一维的线性空间,这些方式的优点是内存的物理组织基本相同,但与用户所写程序的逻辑结构却相差甚远。

通常,用户程序是由若干程序模块和数据模块组成,各自命名,并且大小、功能都有差异。在前面介绍的各种技术中,都要求在对源程序进行编译、链接时,把源程序中的主程序、子程序、数据区等按线性空间的一维地址顺序排列起来。这使得不同作业或进程之间共享公用子程序和数据(因地址重定位的问题)变得非常困难。如果系统不能把用户给定的程序名和数据块名与这些被共享程序和数据在某个进程中的逻辑页对应起来,则不可能共享这些存放在物理页面中的程序和数据。

在分页技术中,一个页面中可能装有两个不同子程序段的指令代码,因此,通过页面共享来达到共享一个逻辑上完整的子程序或数据块是不可能的。另外,在分区法中只能采用静态链接,这无疑加重了系统内存空间的开销。

对于上述问题,采用分段技术能够很好地解决。

4.4.1 分段技术的基本原理

1. 分段的基本思想

分段技术是基于为用户提供一个方便灵活的程序设计环境而提出来的,其基本思想是:把程序按内容或过程(函数)关系分成若干个段,每个段有自己的名字(和唯一的编号),起始地址为0,长度由段的实际内容长度确定(如图4.13所示)。程序中的地址结构表示为段名字(或段号)和段内偏移(如图4.14所示)。



图 4.13 分段技术的逻辑地址空间



图 4.14 分段技术中的地址结构

在程序运行时,操作系统以段为单位来分配内存,每段单独占用一块连续的内存分区。各分区的大小由段的大小决定。这有点类似于动态分区分配方式,不同的是在分段

式内存管理系统中,一个作业或进程可以有多个段,这些段可以离散地放入内存的不同分区中,这些内存分区也不必连续。

与分页系统类似,为了找到每个逻辑段在内存中的对应位置,系统为每个作业或进程建立一个段映射表,简称“段表”。每个段在段表中占有一项,段表项中包含段号、段长度和段起始地址(又称“基址”)等。段表按段号从小到大排列,一个作业或进程的所有段都在其段表中登记。

通常,段表放在内存中,为了方便地找到正在运行的作业或进程的段表,系统还要设置一个段表寄存器。它有两个部分:一部分指出段表在内存的起始地址;另一部分指出段表的长度,表明该段表中共有多少项,即当前作业或进程一共有多少段。

2. 分页与分段的区别

分页和分段技术有很多相似之处,例如,二者在内存中都不是整体连续的,都需要通过地址映射机构将逻辑地址映射到物理内存中。但是,二者在概念上完全不同,主要有以下4点。

(1) 页是信息的物理单位,用户透明,分页是为了实现离散分配,以减少内存的零头,提高内存利用率,便于系统管理;而段是信息的逻辑单位,每一段在逻辑上都是相对完整的一组信息,如一个函数、过程和数组等,用户知道自己的程序多少段,每个段的作用是什么,分段的目的是为了更好地了解用户需要。

(2) 分页式内存管理的作业地址空间是一维的,地址编号从0开始,顺次递增一直排到末尾;而分段式内存管理的作业的地址空间是二维的,要标识一个地址,除给出段内地址外,还必须给出段号。

(3) 页的长度由系统硬件平台确定,是等长的;而段的长度是由具有相对完整意义的程序信息长度确定,是变长的。

(4) 分页系统很难实现过程和数据的分离,因此,无法对它们加以区别和保护,也不便于在用户间共享过程(或函数);分段系统却可以很容易实现这些功能。

4.4.2 分段的地址转换

分段地址的转换过程与分页过程基本相同,其过程如图4.15所示。

程序的逻辑地址由段号 s 和段内偏移地址 d 构成。MMU根据段表寄存器中的段表起始地址和程序中的逻辑地址中的段号 s ,计算出段号 s 所对应的段表项的入口地址。从段表项中可以取出该段的长度值 m ,如果逻辑地址中的偏移值 d 大于 m ,则系统将发生越界中断,进而终止程序的执行。如果小于 m 则表示地址合法,从而将该段在内存中的起始地址从段表项中取出,与段内偏移 d 相加后得到所要访问的内存地址。

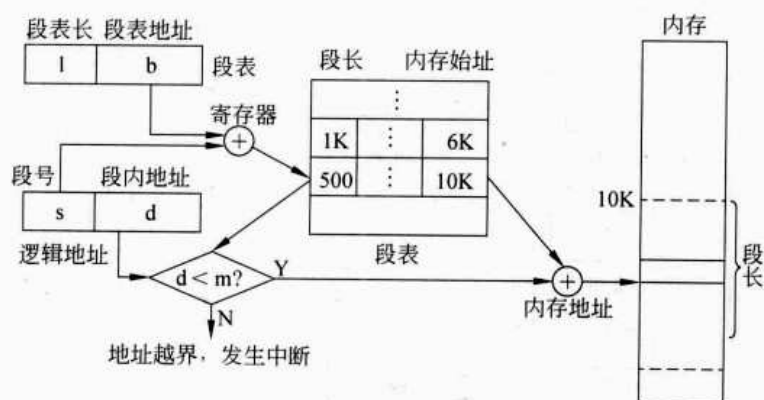


图 4.15 分段的地址转换过程

4.4.3 段的共享和保护

段式存储管理可以方便地实现内存信息共享和进行有效地内存保护。这是因为段是按逻辑意义来划分的,可以按段名(或段号)访问。

1. 段的共享

在多道环境下,常常有许多程序段和数据段是被多个用户程序所共享使用的。特别是随着窗口系统、支持工具等快速发展,系统内需要共享的程序和数据段的个数和体积都在急剧增加,有时这些共享段的大小甚至超过用户程序自有长度的许多倍。

这种情况下,如果每个用户进程或作业都在内存保留它们共享程序和数据的一个副本,那就会极大地浪费内存空间。最好的办法是内存中只保留一个副本,供多个用户共享使用。图 4.16 给出了一个段式系统中共享的例子。

如果用户进程或作业需要共享内存中的某段程序或数据,只要用户使用相同的段名,在新的段表中填入已存在于内存之中的段的起始地址,并置以适当的读写控制权,就可做到共享一个逻辑上完整的内存段信息。

另外,在多道环境下,由于进程的并发执行,一段程序为多个进程共享时,有可能出现多次同时重复执行该段程序的情况(即某个进程在未执行完该段程序之前,其他并发进程又已开始执行该段程序)。这就要求它在执行过程中,该段程序的指令和全局数据不能被修改,即可重入。

2. 段的保护

分段管理的另一个突出优点是便于实现各段的保护。因为各段是有特定意义的逻辑信息单位,即使在进程运行过程中也不失去这些性质。因此段中所有内容可用相同的方式使用。例如,一个程序某些段只含指令,就可以设置为只读和可执行,数据段则可读可

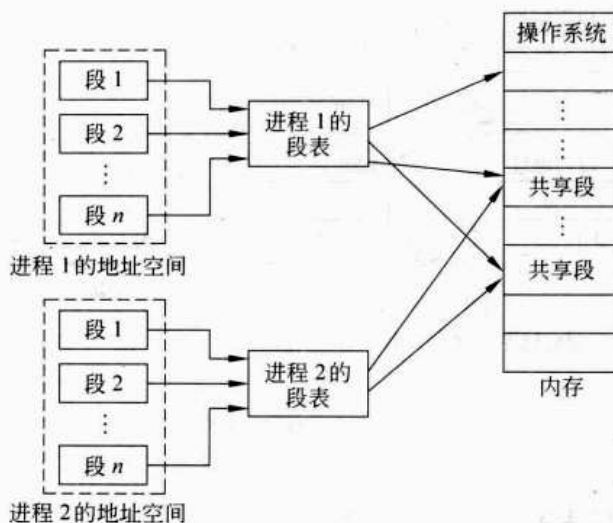


图 4.16 进程间的共享段

写,但不可执行。负责地址转换的硬件会对这些权限进行检验,防止对信息的非法存取。

除存取控制外,分段的另外一种保护就是地址越界保护法。地址越界保护利用段表中的段长项与逻辑地址中的段内相对地址比较进行的。若段内相对地址大于段长,系统就会产生保护中断(在允许段动态增长的系统中,段内相对地址大于段长是允许的,为此段表中设置相应的增补位以指示该段是否允许该段动态增长)。

4.5 段页式技术

分页技术有效地克服了碎片问题,提高了内存的利用率,而分段技术则为用户提供了一个二维的逻辑地址空间,反映了程序的逻辑结构,有利于段的动态增长以及共享和内存保护等。从内存管理的目的来讲,主要是方便用户的程序设计和提高内存的利用率。因此,把分段技术和分页技术有机地结合起来,“各取所长”,正是段页式技术被提出的原因。

段页式技术的基本思想是在分段技术的基础上,再对各个段组成的一维线性空间进行分页后,以页为单位为其分配物理内存。因此,段页式技术的开销较大,不过,随着硬件技术的快速发展,这一技术已经被广泛采用。

4.5.1 段页式技术的基本原理

1. 逻辑地址的构成

在段页式系统,一个作业或进程仍然拥有一个自己的二维地址空间,这与段式管理时

相同。首先,一个进程中所包含的具有独立逻辑功能的程序或数据仍被划分为段,并有各自的段号 s : 这反映和继承了段式管理的特征。其次,对于段 s 中的程序或数据,则按照一定的大小将其划分为不同的页。和页式系统一样,最后不足一页的部分仍占一页。这反映了段页式技术中的页式特征。从而,段页式技术中的进程的逻辑地址空间中的逻辑地址结构由三部分组成: 即段号 s 、页号 p 和页内相对地址 d , 如图 4.17 所示。

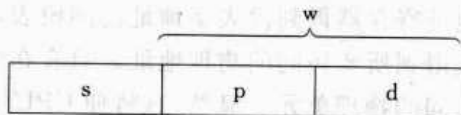


图 4.17 段页式技术的地址结构

对于这个由三部分组成的逻辑地址来说,程序员可见的仍是段号 s 和段内相对地址 w 。 p 和 d 是由地址变换机构把 w 的高几位解释成页号 p , 并把其余的低位解释为页内地址 d 而得到的。

由于逻辑空间的最小单位是页而不是段,从而内存可用区也就被划分成为大小相等的页面,且每段所拥有的程序和数据在内存中可以页为单位(不一定连续地)分开存放。分段的大小也不再受内存可用区的限制。

2. 段表和页表

为了实现段页式技术,系统必须为每个作业或进程建立一张段表。另外,由于一个段又被划分成了若干页,每个段又必须建立一张页表(也可以多个段甚至进程的所有段共享一个页表),把段中的逻辑页号对应到物理内存中的实际页面号。另外,由于在段页式技术中,页表不再是属于进程而是属于某个段,因此,段表中应有表项指出该段所对应页表的页表始址和页表长度(如果是进程所有段共享一个页表,则页表始址和页表长度可存放在 PCB 中)。段页式技术中段表、页表以及内存的关系如图 4.18 所示。图中各表中各项的含意与其在单独的分段和分页中的含义相同。

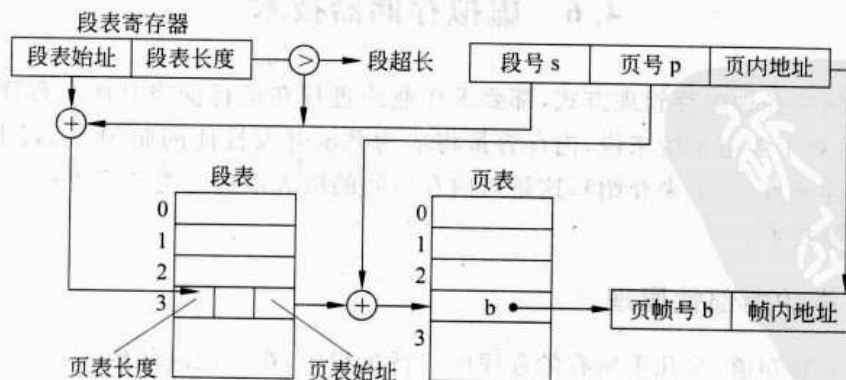


图 4.18 段页式技术的地址转换过程

4.5.2 地址转换过程

在使用段页式内存管理的系统中,进程的段表和页表通常都存放在内存中。因此,要对内存中指令或数据进行一次存取的话,至少需要访问三次以上的内存。第一次是由段表地址寄存器得到段表始地址访问段表,由此取出对应段的页表地址。第二次则是访问页表得到所要访问的物理地址。只有在访问了段表和页表之后,第三次才能访问真正需要访问的物理单元。显然,这将使 CPU 的执行指令速度大大降低。

为了提高地址转换速度,设置快速联想寄存器就显得比段式管理或分页技术时更加需要。在快速联想寄存器中,存放当前最常用的段号 s 、页号 p 及其所对应的内存页面和一些权限控制属性。当要访问内存空间某一单元时,可在通过段表、页表进行内存地址查找的同时,根据快速联想寄存器查找其段号和页号。如果所要访问的段或页在快速联想寄存器中,则系统不再访问内存中的段表、页表而直接把快速联想寄存器中的值与页内相对地址 d 拼接起来得到物理地址。经验表明,一个在快速联想寄存器中装有 $1/10$ 左右的段号、页号及页面的段页式技术系统,可以通过快速联想寄存器找到 90% 以上的所要访问的内存地址。

段页式技术的地址转换过程如图 4.18 所示。

段页式技术中的存储保护和共享与分段和分页技术中的方法类似,在此不再赘述。

从总体上看,因为段页式技术是段式管理和分页技术方案结合而成的,所以具有它们二者的优点。但反过来说,由于管理软件的增加,复杂性和开销也就随之增加了。另外,需要的硬件及段表和页表所占用的内存也有所增加。更重要的是,如果不采用联想寄存器的方式提高 CPU 访问内存的速度,将会使得系统的性能大大下降。

4.6 虚拟存储器技术

前面介绍的各种内存管理方式,都要求作业或进程在运行前将其所有程序和数据全部装入内存,对于多道系统来说,内存容量将成为系统并发性能的瓶颈。而实际上,按照程序的局部性原理(接下来介绍),这是对内存空间的极大浪费。为了解决这一问题,引入了虚拟存储器技术。

4.6.1 程序的局部性原理

由模拟实验知道,在几乎所有的程序的执行过程中,在一段时间内,CPU 总是集中地访问程序中的某一个部分而不是随机地对程序所有部分平均地访问,这种现象称为程序的局部性原理。局部性原理主要体现在以下几方面。

- 时间局部性。由于程序中存在迭代循环、临时变量和子程序调用,所以 CPU 最

近访问的单元会在不久的将来再次或循环多次被访问到。

- 空间局部性。对某一地址空间的表或数据堆栈进行集中访问。
- 顺序局部性。通常情况下, CPU 跟踪程序的执行是按照指令在主存中的连续地址进行的。只有在遇到转移指令时才会跳转到一个不连续的地址空间, 但跳转到新的地址空间后, 指令又是被顺序执行的。

根据局部性原理, 任一时刻对于任意一个进程只在内存中存放它的一部分信息块是可行的, 这样就可以更好地支持多道程序设计。这也是虚拟存储器技术的基础。

4.6.2 虚拟存储器的概念

所谓虚拟存储器, 并不是以物理形式存在的存储器, 而是用户能作为可编址内存对待的虚拟存储空间, 它使用户程序的逻辑空间与物理的内存空间分离, 是操作系统为用户程序提供的一个比真实内存空间大得多的地址空间。

实现虚拟存储技术的物质基础是二级存储器结构和动态地址转换机构。经过操作系统的管理, 将内存和外存有机地联系在一起, 在用户面前呈现一个足以满足编程需要的特大内存空间。

虚拟存储器实质上把用户地址空间和实际物理内存空间分离, 当作两个不同的概念进行管理, 动态地址转换机构在程序运行时负责把逻辑(虚拟)地址转换成物理地址, 以实现动态定位。

采用虚拟存储技术的系统中, 程序开始运行时, 只将程序的一部分装入内存, 而其余的部分只有当程序运行过程中使用到时再装入内存; 当没有足够的内存空间时, 系统的内存管理程序会按一定的策略自动地选择一些内存空间, 将其内容交换到外存空间, 并释放这些内存空间由其他进程使用。这样做的结果是程序的运行不会受到影响(至少用户感觉不到)。

需要注意的是, 尽管虚拟存储为用户提供了特大的地址空间, 但其容量不是无限大的, 它主要受限于系统的以下两方面特性: 一是指令中表示地址的字长, 也就是地址线的位数; 另一个是外存的容量, 当内存不足时, 虚拟存储技术会利用外存来保存内存中的中间数据, 所以由于外存容量的限制, 从而可能造成虚拟空间的限制。

虚拟存储技术主要分为虚拟页式和虚拟段式两种, 也可将两者结合起来构成虚拟段页式存储器。

4.6.3 虚拟存储管理需要考虑的问题

分页技术、分段技术以及段页式技术都能提供虚拟存储管理的基础机制: 将内存和外存统一管理, 内存中只存放那些经常反复被调用和访问的程序段和数据, 而进程或作业的其他部分则保存于外存中, 待需要时再调入内存。

虚拟存储的这一机制的性能瓶颈在于：它需要在内存和外存之间交换信息。这种信息的交换涉及到一系列处理程序（如设备管理、中断处理等），因此需要消耗较多的时间。如果这种交换非常频繁，势必会造成大量的 CPU 开销，从而降低系统性能。

当一个程序只有局部装入时，对内存容量有一个临界值要求。当内存分配小于这个临界值时，内存和外存之间的交换频率将会急剧增加，而内存分配大于这个临界值时，再

增加内存分配也不能显著减少交换次数。每个程序要求的这个内存临界值被称为工作集（如图 4.19 所示）。

因此，在采用虚拟存储时，需要主要考虑以下几方面的问题。

（1）调入策略。是指如何确定什么时候把需要的信息从外存调入内存。请求调入算法是仅当需要时才调入；先行调入算法中，系统试图预测程序的需求，在实际使用之前，就把适当的信息调入内存。

（2）分配策略。是指如何确定把调入的信息放置在内存的何处，即给它分配哪块空闲内存。

（3）置换策略。又称淘汰策略，它是指当内存可用空间不足时，如何确定将哪些已经占用的页交换到外存空间，并重新分配给其他进程使用这些物理页。

4.6.4 虚拟分页技术的基本原理

虚拟分页技术是在传统的分页技术的基础上发展起来的。它分为请求页式管理和预调入页式管理两种。

请求页式管理和预调入页式管理在作业或进程开始执行之前，都不把作业或进程的程序段和数据段一次性地全部装入内存，而只装入被认为是经常反复执行和调用的部分，其他部分则在执行过程中动态装入。

请求页式管理与预调入页式管理的主要区别在它们的调入方式上。请求页式管理的调入方式是：当需要执行某条指令而又发现它不在内存时，或当执行某条指令需要访问其他的数据或指令时，这些指令和数据不在内存中，从而发生缺页中断，系统将外存中相应的页面调入内存。预调入方式是：系统对那些在外存中的页进行调入顺序计算，估计出这些页中指令和数据的执行和被访问的顺序，并按此顺序将它们顺次调入和调出内存。

除了在调入方式上请求页式管理和预调入管理有些区别之外，其他方面这两种方式基本相同。因此，下面主要介绍请求页式管理。

请求页式管理的地址变换过程与传统的分页技术相同，也是通过页表查出相应的页

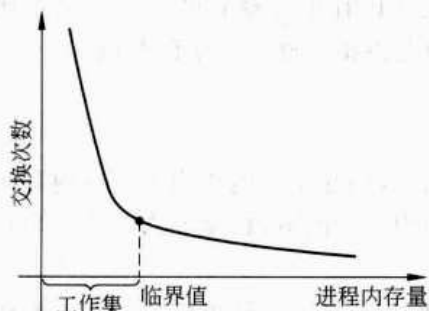


图 4.19 内存量和交换次数的关系

面号之后,由页面号与页内相对地址相加而得到实际物理地址。

但是,由于请求页式管理只让进程或作业的部分程序和数据驻留在内存中,所以在执行过程中,不可避免地会出现某些虚页不在内存中的问题。因此,实现一个请求页式的虚拟存储系统,需要解决下面三个具体问题。

(1) 系统如何获知进程当前所需页面不在内存中。

(2) 当发现缺页时,如何把所缺页面调入内存。

(3) 当内存中没有空闲页面时,为了接受一个新页,需要把老的淘汰出去,这又根据什么策略选择淘汰页面。

第一个问题可以用扩充页表项的方法解决。即与每个虚页号相对应,除了页面号之外,再增设该页是否在内存的“中断位”以及该页在外存中的副本起始地址。扩充后的页表如图 4.20 所示。

页号	页面号	中断位	外存始址	改变位

图 4.20 扩充后的页表

关于虚页不在内存时的处理,涉及到如下两个问题。

(1) 采用何种方式把所缺的页调入内存。

(2) 如果内存中没有空闲页面时,把调进来的页放在什么地方。也就是说,采用什么样的策略来淘汰已占据内存的页。还有,如果在内存中的某一页被淘汰,且该页曾因程序的执行而被修改,则显然该页是应该重新写到外存上加以保存的。而那些未被访问修改的页,因为外存已保留有相同的副本,写回外存是没有必要的。因此,在页表中还应增加一项以记录该页是否曾被改变。增加改变位后的页表项如图 4.20 所示。

当内存中没有空闲页面时,有关缺页的调入和存放问题,实际上是一个内存页面置换算法的问题。选择什么样的置换算法,将直接影响到内存利用率和系统效率。事实上,如果置换算法选择不当,有可能产生刚被调出内存的页又要马上被调回内存,调回内存不久又马上被调出内存,如此反复的局面。这使得整个系统的页面调度非常频繁,以致大部分时间都花费在主存和辅存之间的来回调入调出上。这种现象被称为抖动(thrashing)。有关抖动现象,将在后面讨论。

虚拟分页技术的流程图如图 4.21 所示。从图中可以看出,有关地址变换部分是由硬件自动完成的。当硬件变换机构发现所要求的页不在内存时,产生缺页中断信号,由中断

处理程序做出相应的处理。中断处理程序是由软件实现的。除了在没有空闲页面时要按照置换算法选择出被淘汰页面之外,还要从外存读入所需要的虚页。这个过程不仅要启动相应的外存设备,并且涉及到文件系统。因此,请求页式管理是一个十分复杂的处理过程,内存利用率的提高是以牺牲系统开销的代价换来的。

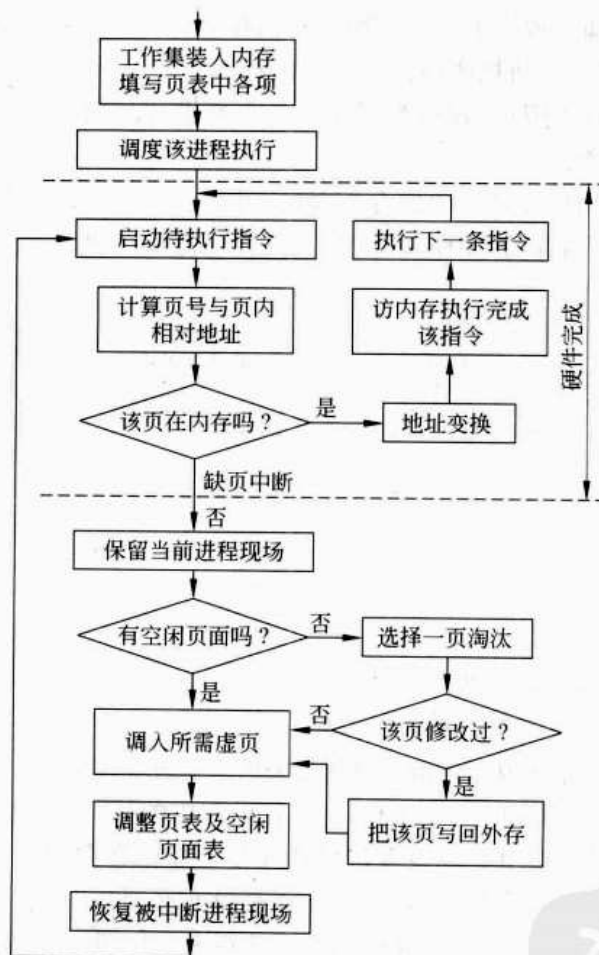


图 4.21 请求页式管理流程图

4.6.5 请求页式管理中的置换算法

置换算法在内存中没有空闲页面时被调用。它的目的是选出一个被淘汰的页面。如果内存中有足够的空闲页面存放所调入的页,则不必使用置换算法。把内存和外存统一管理的真正目的是把那些被访问概率非常高的页存放在内存中。因此,置换算法应该置换那些被访问概率最低的页,将它们移出内存。

下面介绍几种比较常用的置换算法。

1. 随机淘汰算法

在系统设计人员认为无法确定哪些页被访问的概率较低时,随机地选择某个用户的页面并将其换出将是一种明智的作法。

2. 轮转法和先进先出算法

轮转法又称 Round Robin(RR)。轮转法循环换出内存可用区内一个可以被换出的页,无论该页是刚被换进或已换进内存很长时间。FIFO(先进先出)算法认为先调入内存的页不再被访问的可能性要比其他页大,因而选择最先调入内存的页换出。

实现 FIFO 算法需要把各个已分配页面按分配时间顺序链接起来,组成 FIFO 队列,并设置一个置换指针指向 FIFO 队列的队首页面。这样,当要进行置换时,只需把置换指针所指的 FIFO 队列前头的页面顺次换出,而把换入的页面链接在 FIFO 队尾即可。

由实验和测试发现 FIFO 算法和 RR 算法的内存利用率不高。这是因为,这两种算法都是基于一个假设:即 CPU 是按线性顺序访问地址空间的。事实上,许多时候并非如此,例如执行循环语句时。因此,那些在内存中停留时间最长的页往往也是经常被访问的页。尽管这些页变“老”了,但它们被访问的概率仍然很高。

FIFO 算法的另一个缺点是它有一种反常现象。一般来说,对于任一作业或进程,如果给它分配的内存页面数越接近于它所要求的页面数,则发生缺页的次数会越少。在极限情况下,这个推论是成立的。因为如果给一个进程分配了它所要求的全部页面,则不会发生缺页现象。但是,使用 FIFO 算法时,在未给进程或作业分配它所要求的页面数时,有时会出现分配的页面数增多,缺页次数反而增加的奇怪现象。这种现象称为 Belady 现象,如图 4.22 所示。

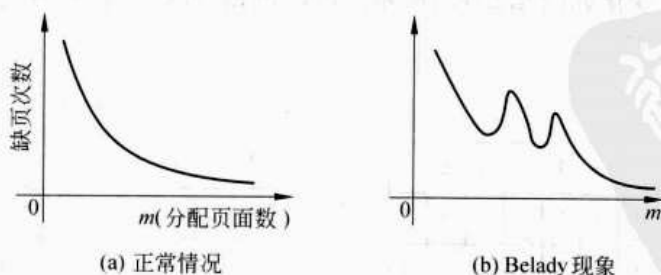


图 4.22 Belady 现象

下面的例子可以用来说明 FIFO 算法的正常换页情况和 Belady 现象。设进程 P 共有 8 页,且已在内存中分配有 3 个页面,程序访问内存的顺序(访问串)为 7,0,1,2,0,3,

0,4,2,3,0,3,2,1,2,0,1。这里,这些自然数代表进程 P 所建的程序和数据的页号。内存中有关进程 P 所建的程序和数据的各页面变化情况如图 4.23(a)所示。

由图可以看出,进程在一个执行过程中,实际上发生了 12 次缺页。如果设缺页率为缺页次数与访问串的访问次数之比,则该例中的缺页率为 $12/17=70.5\%$ 。

如果给进程 P 分配 4 个页面,则在其执行过程中内存页面的变化情况如图 4.23(b)所示。进程 P 在拥有 4 个内存页面时,共发生 9 次缺页,其缺页率为 $9/17=52.9\%$ 。

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0
	0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1
		1	1	1	1	0	0	0	3	3	3	3	3	2	2	2

(a)

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1
7	7	7	7	7	3	3	3	3	3	3	3	3	3	2	2	2
	0	0	0	0	0	0	4	4	4	4	4	4	4	4	4	4
		1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
			2	2	2	2	2	2	2	2	2	2	1	1	1	1

(b)

图 4.23 Belady 现象举例 1

以上是使用 FIFO 算法正常换页时的例子。下面来看另外一种访问串时的情况。设进程 P 可分为 5 页,访问串为 1,2,3,4,1,2,5,1,2,3,4,5。当进程 P 分得 3 个页面时,执行过程中内存页面变化如图 4.24(a)所示。由图可知,进程 P 在执行过程中共缺页 9 次,其缺页率为 $9/12=75\%$ 。

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	4	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4

(a)

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3

(b)

图 4.24 Belady 现象举例 2

但是,如果为进程 P 分配 4 个内存页面,是否缺页率会变小呢? 进程 P 分得 4 个页面时,执行过程中内存页面的变化情况如图 4.24(b)所示。

由图 4.24(b)可知,当进程 P 分得 4 个页面时,在执行过程中的缺页次数为 10 次。即缺页率 $=10/12=83.3\%$ 。

先进先出算法产生 Belady 现象的原因在于它根本没有考虑程序执行的动态特征。

3. 最近最少使用页面置换算法

该算法的基本思想是:当需要淘汰某一页时,选择离当前时间最近的一段时间内最少使用过的页面先淘汰。理由是:如果某页被访问了,则它可能马上还要被访问。或者反过来说,如果某页很长时间未被访问,则它在最近一段时间也不会被访问。

要完全实现 LRU(least recent used)算法是十分困难的。因为要找出最近最少使用的页面的话,就必须对每一个页面都设置有关的访问记录项,而且每一次访问都必须更新这些记录。这显然要花费巨大的系统开销。因此,在实际系统中往往使用 LRU 的近似算法。

比较常用的近似算法如下。

(1) 最不经常使用页面淘汰算法 LFU(least frequently used)。该算法在需要淘汰某一页时,首先淘汰到当前时间为止,被访问次数最少的那一页。这只要在页表中给每一页增设一个访问计数器即可实现。每当该页被访问时,访问计数器加 1,而发生一次缺页中断时,则淘汰计数值最小的那一页,并将所有的计数器清 0。

(2) 最近没有使用页面淘汰算法 NUR。该算法在需要淘汰某一页面时,从那些最近一个时期内未被访问的页面中任选一页淘汰。这种算法只要在页表中增设一个访问位即可实现。当某页被访问时,访问位置 1。否则,访问位置 0。系统周期性地对所有页面的访问位清零。当需淘汰一页时,从那些访问位为 0 的页中选一页进行淘汰。

4. 理想型淘汰算法 OPT(optimal replacement algorithm)

该算法淘汰在访问串中将来再也不出现的或是在离当前最远的位置上出现的页面。这样,淘汰掉该页面将不会造成因需要访问该页又立即把它调入的现象。遗憾的是,这种算法无法实现,因为它要求必须预先知道每一个进程的访问串。

4.6.6 抖动问题

在虚拟分页系统中,如果一个进程处于内存的页面太少,很快会发生缺页。此时,内存空闲页面不足,就需要换出某些页。如果置换的算法和流程设计得不好,可能会换出马上就要用到的页,这就造成刚换出的页马上又需要被调入。如此反复,使得整个系统的

页面替换非常频繁,以致大部分 CPU 的时间都用在页面的换入和换出上。这种局面称为系统“抖动”(thrashing)。

要防止抖动发生或者限制抖动的影响,有多种方法,可根据产生抖动的原因选用。

1. 采用局部策略

如果一个进程出现抖动,它不能从另外的进程那里获取内存块,这样就不会引发其他进程的抖动,使抖动局限于一个小范围内。但是,这种方法并未消除抖动,而且在多个进程发生抖动的情况下,等待磁盘 I/O 的进程增多,使得平均缺页处理时间加长。

2. 利用工作集策略防止抖动

工作集模型是 Dennin 于 1968 年提出的研究和描述程序的局部性问题的理论。所谓工作集,就是一个进程在某一小段时间内所访问页的集合。为使程序有效地运行,其工作集必须放在内存中,否则,由于程序频繁地从外存请求页面,而出现“抖动”。

工作集存储管理策略力求把活跃程序的工作集保存于内存中,在多道程序运行环境下,当要增加一道新程序时,其关键是决定内存中是否有足够的可利用空间,以提供给新程序的页面工作集。这种决定常常是采用探索方式须做出的,特别是在初始化新进程的情况下,因为系统事先并不知道给定进程的工作集应是多长。

一个进程(总页数为 N)在时刻 t 的工作集可形式化地定义为

$$w(t, h) = \{ \text{页 } i \mid \text{页 } i \in N \text{ 且页 } i \text{ 在 } t \text{ 时刻前的一段时间 } h \text{ 内被访问} \}$$

换句话说,工作集是最近被访问过的页的集合,“最近”是集合参数之一 h 。根据局部

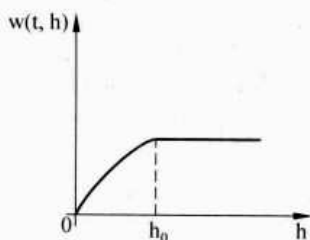


图 4.25 工作集的大小 $w(t, h)$ 与时间间隔 h 的关系

性原理,可以期望工作集成员的改变在时间上是缓慢的。Dennin 给出了工作集大小 $w(t, h)$ 随 h 变化的关系,如图 4.25 所示。

随着 h 的增加,在工作集可能出现的缺页的机会就减少。这样,就给出一个适当的 h 值(例如 h_0),使得即使再增加 h 值,也不会明显地增加工作集尺寸。

就调入和淘汰策略而言,工作集的价值意味着“对于一个进程,在任一时刻,仅当其全部工作集都在内存中时,它才能运行下去”。

由于程序的执行是动态的、不可预测的,所以工作集也是变化的、瞬态的。进程下一个工作集可以完全不同于它的前一个工作集。所以,使用工作集存储管理策略是很困难的。但是,这一理论使人们认识到,只有在具备足够容量内存的情况下,才能有效地实现多道程序的并发运行,它也可以提醒人们注意防止抖动现象的发生。

3. 挂起某些进程

当出现 CPU 利用率很低,而磁盘 I/O 非常频繁的情况时,可能因为并发程度太高而造成抖动。为此,可以挂起一个或几个进程,腾出内存空间供抖动进程使用,从而消除抖动现象。选择被挂起进程的策略有多种,如低优先级进程、缺页进程、最近激活的进程、内存驻留最少的进程和内存驻留最多的进程等。

4. 采用缺页频度法

抖动发生时缺页率必然很高,通过控制缺页率就可预防抖动。如果缺页率太高,表明进程需要更多的内存;如是缺页率太低,表明进程可能占用了太多的内存块。

可以设定一个缺页率的范围(上限和下限),当系统的实际缺页率超过上限时,就增加进程的内存分配,如果系统的实际缺页率低于下限时,可适当减少进程的内存分配。这样通过测量和控制缺页率,就可避免抖动。

4.6.7 性能分析

页式虚拟存储管理解决了内存实际容量的限制,能使更多的作业或进程同时多道运行,从而提高了系统的效率。但缺页中断处理要付出相当的代价,不仅由于页面的调入、调出要增加 I/O 系统的负担,而且影响系统的效率。因此,应该尽可能地减少缺页中断的次数。这需要从程序的质量、页面的大小、内存的容量,以及淘汰算法等几个方面去考虑。

虚拟存储器的效率与程序的局部性程度密切相关。在程序设计时,可借助于循环、经常用到的变量和子程序等程序结构来提高程序的时间局部性;也可以尽量采用顺序的指令串、线性的数据结构(例如,数组或常用的变量彼此相近置放)来提高程序的空间局部性。

其次,页面的大小也是影响虚拟存储系统性能的因素之一。页面大的好处是:页表占用的内存空间少,查表速度快;缺页中断的次数也就相对少些。但是,这样在页面调度时,一次换页的时间较长,页内零头空间浪费的可能性较大。页面小的利弊则正好相反。一般说来,页面的大小应根据实际情况来确定,它和计算机的性能以及用户的要求都有关系。目前常见的页面大小有 4KB,8KB,4MB 等。

从原理上说,在虚拟存储系统中,只要为每一个作业分到一块内存空间能够运行就成了,因此尽量可以增加同时运行的作业的数目;而实际上,如果一个作业或进程运行时,占有内存的页数小于其工作集,则会造成过多的缺页中断,增加 I/O 次数,降低 CPU 利用率。因此,对每个程序来说,要使其有效地工作,必须在内存中放置一定的页面数,具体数量可参考工作集理论。

4.7 Linux 的内存管理

Linux 的内存管理子系统是采用请求调页式的虚拟存储器技术实现的。Linux 内核根据内存的当前使用情况动态换进换出进程页,通过外存上的交换空间存放换出页。内存和外存之间的相互交换信息以页为单位进行的,这样可使进程的地址转换具有更大的灵活性,并充分利用了内存空间。

4.7.1 进程的虚存空间及其划分

内存的管理方法与计算机硬件结构有密切的关系,不同的硬件结构导致了不同的虚拟空间管理方法。在 32 位硬件平台上,Linux 的逻辑地址为 32 位,因此,每个进程的虚拟地址空间为 4GB,如图 4.26 所示。

在 4GB 的空间中,操作系统占用了高端的 1GB,而低端的 3GB 则留给用户程序使用。

4.7.2 进程空间的描述

1. 关键数据结构

对于每个进程,Linux 内核使用一个 `mm_struct` 结构体描述进程使用的地址空间的各种信息(如图 4.27 所示)。在每个进程的 PCB(`task_struct` 结构体)中包含一个 `mm_struct` 结构的指针(`mm` 域)指向

本进程的 `mm_struct`。每个进程与用户空间相关的各种信息都存放在 `mm_struct` 结构体中,其中包括本进程的页目录表的地址,本进程的用户区的组成情况等重要信息。

Linux 的每个进程的 4GB 的虚存空间是通过页目录表来描述的,也就是说,在创建每个进程(只有内核空间的内核线程除外)时,Linux 都会为该进程创建一个固定大小的页目录表,实际上进程 4GB 的虚拟存储空间就是由这个页目录表描述的,所以每个进程都有相同大小的虚拟存储空间。

每个进程的页目录表中对应于 3GB 以上虚存空间的页目录项都映射到操作系统启动时初始化好的操作系统所使用的物理页面的页表,也就是操作系统的页表,这样,每个进程的 3GB 以上的虚存空间自然就映射到了装载操作系统的物理内存空间。

每个进程的用户区是由一组 `vm_area_struct` 结构体组成的链表来描述的。用户区的每个段(如代码段、数据段和栈等)都由一个 `vm_area_struct` 结构体描述,其中包括了本段



图 4.26 Linux 进程的虚存空间及其组成(32 位平台)

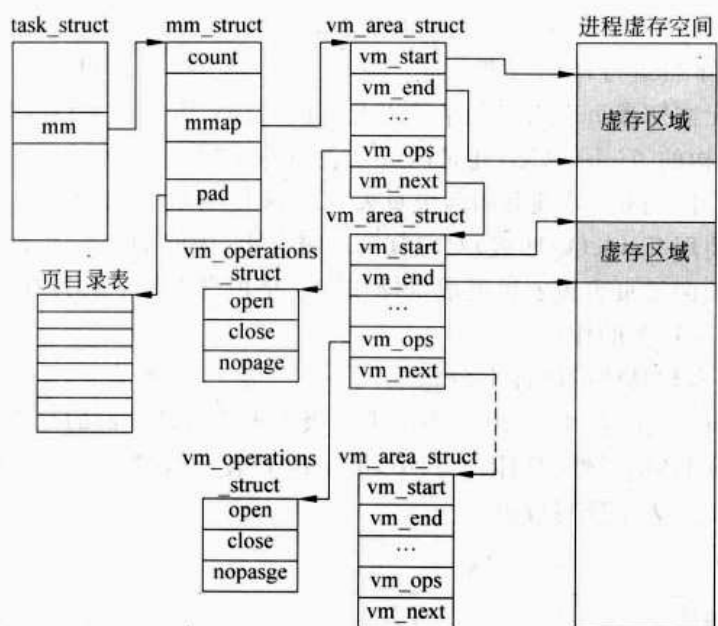


图 4.27 Linux 进程虚存空间的描述

的起始虚拟地址和结束虚拟地址,也包括了当发生缺页中断时如何(通过函数 `nopage`)找到本段在外存上的相应的内容(即后面讨论的内存映射)。

2. Linux 的分页模型

在 Linux 2.6 内核中, Linux 采用了 4 级页表结构(如图 4.28 所示)。第一级页表的起始地址保存在 PCB 中(在 i386 平台上,进程运行时第一级页表的起始地址加载到 CR3

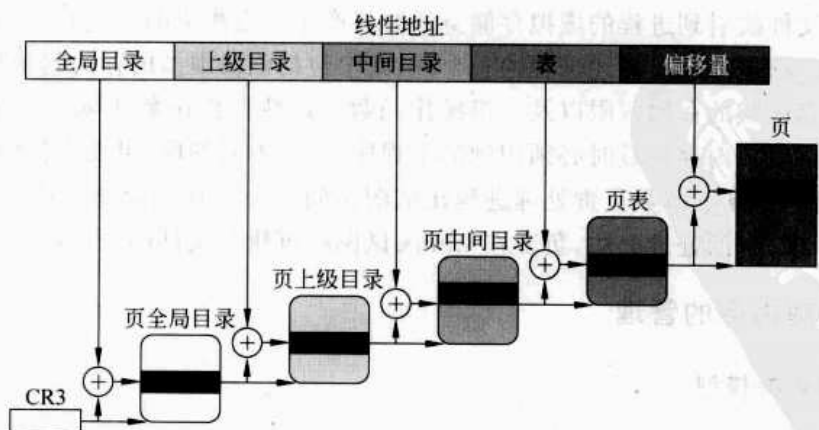


图 4.28 Linux 的 4 级页表结构

寄存器),页表中的每个页表项保存了下级页表的页面号(即起始地址)。

为了实现跨平台运行,Linux 提供了一系列转换宏使得内核可以访问特定进程的页表。这样,内核无需知道页表入口的结构以及它们的排列方式。采用这种策略后,无论在具有三级页表结构的 Alpha AXP 还是两级页表的 i386 处理器中,Linux 总是使用相同的页表操纵代码。对于 i386 处理器的两级页表,Linux 使用 Global Directory(第一级)对应其第一级页表,使用 Table(第四级)对应其第二级页表,中间的两级页表在使用的过程中同样有用(中间的两级页表被宏扩展成空操作),只是相当于每个表只有一个页表项,也就是实际上只起了个过渡的作用。

页表项的内容与具体的硬件平台相关,图 4.29 是 i386 平台上的页表结构。从图可知,一个页表项有 32 位,其中高 20 位表示下一级页表或物理页面的页面号,低 12 位表示该页表项对应页的各项属性和权限。其中,第 0 位 P 表示该逻辑页在内存中是否有对应的映射页面,第 1 位表示读写权限。

31	12	11								0
页框地址 位 31..12 (PAGE FRAME ADDRESS)										
可用 (AVAIL)										
0 0										
D										
A										
0 0										
U/S										
R/W										
P										

图 4.29 i386 平台的页表结构

3. 内存映射

当可执行文件需要创建为进程准备运行时,可执行文件的内容仅仅映射到了对应进程的虚拟地址空间中,而并没有调入物理内存。可执行文件使用的共享库也同样如此。当程序开始运行并使用到这部分时,Linux 才通过缺页中断把它们从磁盘上调入内存。这种将文件连接到进程虚拟地址空间的过程称为内存映射。

可执行文件映射到进程的虚拟存储空间时将产生一组相应的 `vm_area_struct` 数据结构。一个 `vm_area_struct` 描述进程的一个段,其中包括该段虚拟内存的起始与结束位置,进程对此内存区域的存取权限以及一组操作函数。这些操作函数实际上都是函数指针,是内核在访问虚拟内存区域时必须用到的子程序,对于不同的段,可能会指向不同的处理函数。其中的 `nopage` 函数负责处理进程在试图访问一个不在当前物理内存中的虚拟页(通过缺页中断)时的处理办法,负责在 Linux 试图将可执行文件的页面调入内存时使用。

4.7.3 物理内存的管理

1. 物理内存模型

为了适应 NUMA(non-uniform memory access)硬件结构(在第 3 章的 3.4.5 节有介绍)的发展,支持更多内存,提高内存管理的性能,并建立一种独立于硬件架构的内存描述

方法, Linux 建立了图 4.30 所示的物理内存模型。在这个模型中, 对于一个 CPU 来说, 使用性能相同的一段内存空间被称为一个节点(node)。也就是说, 一个节点内的内存对于一个 CPU 来说, 其访问性能是相同的, 但对不同的 CPU, 访问性能则不一定相同。对于 PC 来说, 因为内存对每个 CPU 的使用特性都是相同的, 所以 PC 上的所有内存都属于同一个节点。

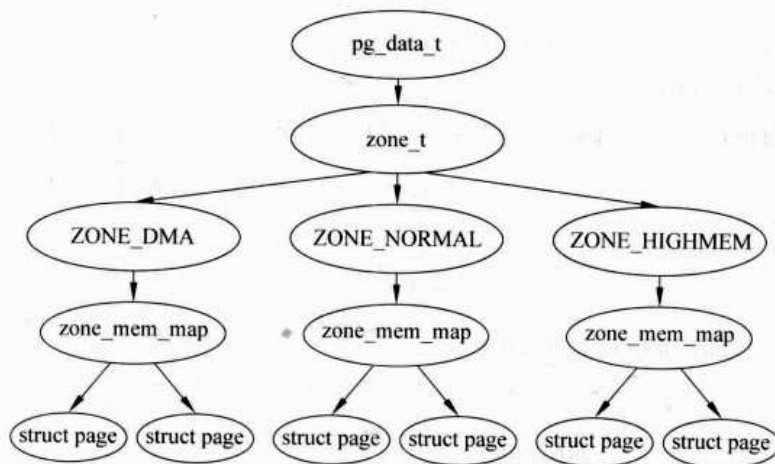


图 4.30 Linux 的物理内存模型

对于每个节点中的内存, 根据其适用场合的不同, 可被进一步划分成一些区域(zone)。i386 平台上的区域主要包含以下三种类型。

- ZONE_DMA(0~16MB)。该区域包含 ISA/PCI 设备需要的低端物理内存区域中的内存范围。
- ZONE_NORMAL(16~896MB)。在该区域内, 内核可以直接方便地实现虚拟地址到物理地址的映射。因为内核管理内存时需要频繁地在虚拟地址和物理地址之间进行转换, 所以内核主要使用这一区域。当然, 内核不用时, 用户程序同样可以使用。
- ZONE_HIGHMEM(896MB 以上)。该区域内的内存通常由用户程序使用, 内核使用时的性能要比 ZONE_NORMAL 低而且使用过程复杂。

在内核中, 节点使用 struct pglist_data 结构描述, 区域使用 struct zone_struct 结构描述, 每一个物理页面使用一个 struct page 结构描述, 所有这些 page 结构体都保存在全局结构数组 struct mem_map 中。

page 结构体中与内存管理相关的重要域如下。

- _count。该页面的当前被引用的次数。

- `_mapcount`。指向本页面的页表项的数目。
- `flags`。类型为 `unsigned long`。每个 `bit` 表示该页面的一种使用标志(由定义的宏表示)。如 `PG_locked` 表示本页面由于某种原因(如 I/O)被加锁, `PG_dirty` 表示本页面被修改过, `PG_referenced` 表示本页面被访问过等。

2. 物理页面的分配

Linux 使用 `free_area` 数组来管理空闲页面(如图 4.31 所示), 每个区域(zone)都有一套这样的管理结构。`free_area` 中的每个元素都包含两个重要指针: 一个指向一个位图区域; 而另一个则指向一个由 `page` 结构体(页面信息)作为元素的双向链表。

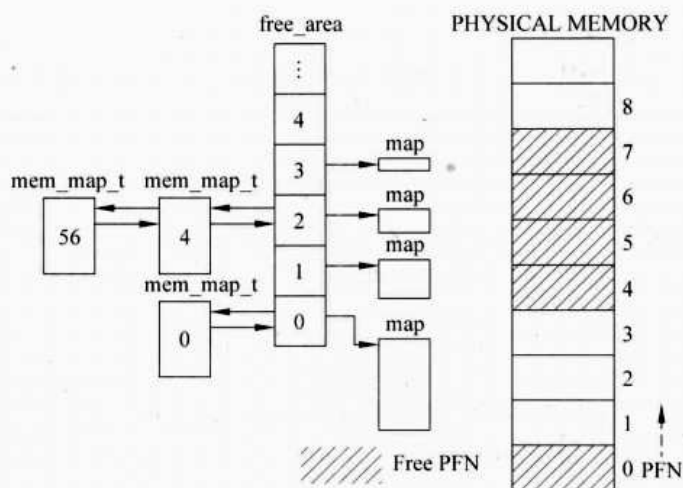


图 4.31 Linux 空闲物理内存管理结构

Linux 使用 Buddy(伙伴)算法来管理页面的分配和回收。页面管理程序以 2 的幂次方作为单位来管理内存块的, 如 1 个、2 个、4 个等连续页面组成的内存块。在实际分配时, 如果申请的内存块的大小不是 2 的整数幂个页面, 页面管理程序会选择一个略大于但最接近申请块大小的连续块, 多出来的页则作为空块继续等待分配。例如, 如果申请 3 个页, 内核如果找到一个连续 4 页的块, 则在分配完前 3 页后, 将第 4 页继续作为一个单页的空块进行管理。

`free_area` 数组中第一个元素描述本区域内所有的 1 个空闲页面, 对应的位图每个位表示一个页面, 如第 0 位表示第 0 个页面的使用状态。第二个元素描述本区域内的所有两个连续的空闲页面组成的块, 对应的位图每个位表示两个页面, 如第 0 位表示第 0 和 1 页面的使用状态, 只要有任一页面被使用就设置为 1, 表示已使用, 依次类推。list 域表示一个队列头, 它包含指向 `mem_map` 数组中 `page` 数据结构的指针。所有的空闲页面都在

此队列中。map 域是指向某个特定页面尺寸的位图的指针。当页面的第 N 块空闲时,位图的第 N 位被置零。

在图 4.31 中,单个的空闲物理页面有一个,其页面号是 0;4 个页面连续的内存块有 2 个,第一个从页面号 4(4、5、6、7)开始,而后一个从页面号 56(56、57、58、59)开始。

在需要分配空闲页时,分配算法首先搜寻满足请求大小的页面块。它从 free_area 数据结构的 list 域着手沿链表来搜索空闲页面。如果没有请求大小的空闲页面,则搜索略大于且最接近于请求大小的内存块。这个过程一直将持续到 free_area 被搜索完或找到满足要求的内存块为止。如果找到的页面块大于请求的块则对其进行分割以使其大小与请求块匹配。空闲块则被连入相应的队列并修改相关的位图。比如,如果需要分配三个连续的页面,则选择一个 4 页的连续块,分配前三块,而第 4 块的块号则挂入一个空闲页的链表,并相应地修改相关的位图(free_area 第 1、2、3 个元素对应的位图都需要修改)。由于块大小都是按 2 的幂次方管理的,所以分割过程十分简单。

在图 4.31 中,当系统中有大小为两个页面块的需求发出时,第一个 4 页面大小的内存块(从页面号 4 开始)将分成两个两页面大小的块。前一个,从页面号 4 开始的,将分配出去返回给请求者,而后一个,从页面号 6 开始,将被添加到 free_area 数组中表示两个页面大小的空闲块的元素 1 中。

3. 页面回收

将大的页面块打碎进行分配将会不断地增加系统中零碎空闲页面块的数目。页面回收时需要解决的关键问题就是如何快速地确定回收的页面能否与已经存在的相邻空闲页面组成更大的空闲页块。free_area 数组中的每个元素指向的位图正是用于这一目的。

当页面块被释放时,代码将通过位图检查是否有相同大小的相邻或者 buddy 内存块存在。如果有,则将它们结合起来形成一个大小为原来两倍的新空闲块。每次结合完之后,代码还要通过更上一位的位图检查是否可以继续合并成更大的页面。

在图 4.31 中,如果释放页面号 1,首先设置 free_area 的第 0 个元素的位置的第 1 位为 0,然后通过检查其相邻的位,发现页面号 0(第 0 位)也是空闲的,于是就将表示两个页面的位图(free_area 第 1 个元素的位图)的对应位置为 0,表示空闲,然后继续检查该位图中表示 2 和 3 号页面的位,结果发现不能再合并,于是,将页面号 1 和空闲页面号 0 结合作为大小为两个页面的空闲块排入 free_area 的第一个元素指向的链表中,并将页面 0 从表示一个页面块的链表中摘除。

4.7.4 基于 Slab 的缓存管理

Buddy 算法采用页面作为分配内存的基本单位,虽然有利于解决外部碎片问题,但却只适合于对大块内存的请求。而对于需要小内存区的请求,例如只请求几十个或几百个字节的内存区,如果也按页面为单位分配的话,对于内存资源显然将是一个极大的浪费。

1. SLAB 分配器的设计思想

从 2.2 版本开始, Linux 内核对内存区的管理采用 Slab 分配器算法。Slab 分配器于 1994 年开始在 Solaris 操作系统上使用, 它比传统的其他分配器有更好的性能和内存利用率。其设计目标基于三个方面。

(1) 对象复用。

Slab 分配器把它管理的内存区看作是各种各样的临时对象, 如 `page`、`vm_area_struct` 和 `inode` 结构等。而分配器操作一个对象的一般顺序是: 为对象分配内存; 构造(初始化)对象; 使用对象; 析构(回收)对象; 释放内存。

内核在为用户进程服务的过程中, 同一类型的数据结构经常被反复请求和释放。如果将先前使用过的某数据结构的内存区保留给下一个申请本数据结构的请求使用, 将大大提高系统效率。例如, 只要内核创建一个新进程, 它就很可能要用到多个 `vm_area_struct`。当该进程结束后, 包含这些结构的内存区还可以被同一类型的对象重新使用。为了避免对同一种对象的重复初始化, Slab 分配器并不丢弃已分配的对象, 而是把它保存在缓存中。当以后有新的请求时, 就可以直接从缓存获取, 而不用重新分配。

另外, 对象复用还包括另一类情况: 就是按照请求对象发生的频率来分别对待。对于频繁地对大小基本固定的内存区的请求, 可以通过创建一组专用的对象来高效地处理, 以避免产生碎片。而对于很少用到的内存区请求, 可以使用传统的 2 的幂次方的分配模式来处理, 这样产生的碎片数量可以限制在一个允许的范围内。

(2) 硬件 cache 利用率。

传统的内存分配器在使用处理器的硬件 cache 时, 都是将所请求的内存大小取整为 2 的整次幂, 其分配地址都是对于相应尺寸而对齐。但是大部分内核对象都是在对象开始的部分存放经常访问的域, 这些域所占请求内存的面积很小(大约是 10% 左右), 结果造成硬件 cache 经常访问这部分内存区, 而其余部分利用率很低。这是由于内存地址内容的不平衡性造成硬件 cache 不能有效地利用, 导致了内存访问效率的下降。

Slab 分配器由于使用数据结构的起始地址不是 2 的幂次方, 因此有利于处理器硬件 cache 做高速缓存处理, 提高内存利用率。

2. Slab 分配器的描述

Slab 分配器有一组缓存, 每个缓存保存同一种对象类型, 例如 `inode`、`mm_struct` 等。通常, 内核从它们各自的缓存中分配和释放对象。用户可以在运行时读取 `/proc/slabinfo` 文件查看 Linux 所使用的 Slab 分配器高速缓存的情况。

包含 cache 的工作内存区被划分为多个 Slab, 每个 Slab 由一个或几个连续的页面组

成。这些页面中既包含已分配的缓存对象,也包含空闲的对象,如图 4.32 所示。描述高速缓存的数据结构是 struct kmem_cache_s 结构体(在 mrn/slab.c 文件中定义)。描述高速缓存中的每个 Slab 所使用的数据结构是 struct kmem_slab_t。

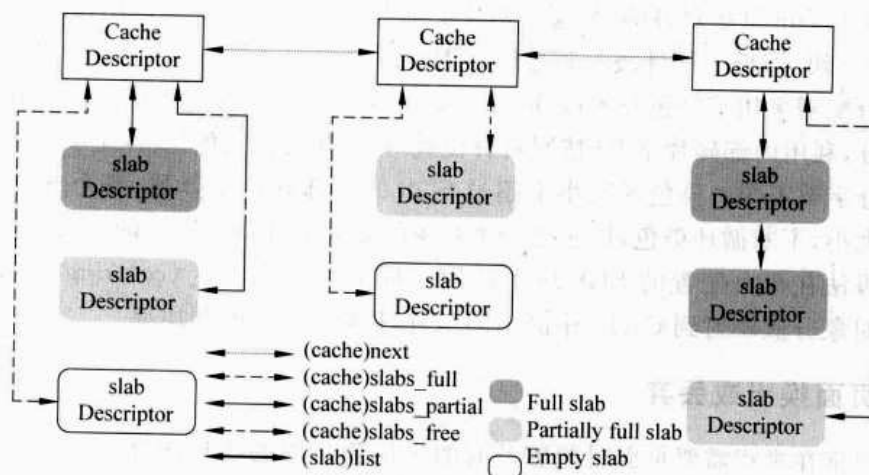


图 4.32 Slab 分配器的描述

从图 4.32 中可以看出,缓存将所有的 Slab 组成一个有序的双向链表,所有对象中,正在使用的 Slab 排在前,有部分活动的 Slab 居中,完全空闲的排在最后。缓存中的 c_freep 指向第一个有空闲块的 Slab 指针。有用户分配请求时,就从该 Slab 上分配对象,直至所有部分活动的 Slab 用完后,才使用空闲的 Slab。

Linux 内核把 Slab 分为通用和专用两种类型。通用 Slab 主要服务于 Slab 分配器自己使用,而专用 Slab 则供内核的其余部分使用。通用 Slab 是在系统初始化期间调用 kmem_cache_init() 和 kmem_cache_sizes_init() 来创建的。专用 Slab 是由 kmem_cache_create() 函数创建的。它先创建一个描述 Slab 的 struct kmem_cache_s 结构,然后把它插到第一个(由变量 cache_cache 所指向的)通用 Slab 中。

Slab 分配器涉及的其他主要操作函数如下。

- int kmem_cache_destroy (kmem_cache_t * cachep)。缓存的销毁。
- int kmem_cache_shrink(kmem_cache_t * cachep)。缓存的收缩。
- void * kmem_cache_alloc (kmem_cache_t * cachep, int flags)。对象的分配。
- void kmem_cache_free (kmem_cache_t * cachep, void * objp)。对象的释放。
- void cache_reap(void * unused)。缓存中的对象析构,并回收对象所占内存。
- void * kmalloc(size_t, int flags)。申请连续内存块。
- void kfree(const void * objp)。释放连续内存块。

3. Slab 分配器的染色技术

传统的 2 次幂表分配器总把对象统一地分配到内存块中,这样由于每个对象的起始地址与硬件 Cache 总是对齐的,而每个对象的前几个元素又最常被使用,可能映射到硬件 Cache 的同一块,降低了硬件缓存和总线利用率。

Slab 分配器采用了染色技术(coloring)解决这一问题。具体实现方法为在每个 Slab 的起始部分,利用内部碎片字节,按照对齐位置对 Slab 进行染色(实际上就是在页的起始处空出部分字节不用),染色区大小不超过 Slab 的内部碎片字节数,每个 Slab 利用不用的染色区大小,不断循环染色,以避免每个对象的起始地址都是以 2 的幂次方开始。

Slab 设法在每次配置的 Slab 块上使用不同大小的染色区(coloring area),以便将 Slab 中的对象分散映射到 Cache 中的不同位置,提高 cache 的利用率。

4.7.5 页面换出或丢弃

为了保证在进程需要页面时能够及时的申请到足够的物理内存,Linux 利用内核线程 kswapd 来定期的检查系统内的空闲物理内存总量。

kswapd 在内核启动时创建,被内核交换定时器周期性的调用。当定时器到时后,kswapd 将检查系统中的空闲页面数 `nr_free_pages` 是否太少。它使用两个全局变量 `zone->pages_high` 和 `zone->pages_low` 来判断是否应该换出或丢弃一些页面。当系统中的空闲页面数大于 `zone->pages_high` 时,kswapd 不做任何工作,它将睡眠并等待下一次定时器到时。当系统中的空闲页面数在 `zone->pages_high` (甚至 `zone->pages_low`) 以下时,kswapd 将通过换出或丢弃页面来腾出内存。如果系统中空闲页面数低于 `zone->pages_low`,kswapd 不仅要换出或丢弃部分页面,还要将睡眠的时间缩减为平时的一半,而在空闲页面数大于 `zone->pages_low` 时,睡眠时间将恢复。

kswapd 通过以下三个途径来减少系统中使用的物理页面的个数。

- 减少缓存与页面 cache 的大小。
- 将系统 V 类型的共享内存页面交换出去。
- 换出或者丢弃属于进程用户空间的页面。

kswapd()通过函数嵌套调用 `try_to_free_pages()` 函数进行页面回收的操作。`try_to_free_pages()` 根据内存的使用情况通过调用 `shrink_caches()` 函数完成页面的换出或丢弃操作。`shrink_caches()` 通过调用函数 `shrink_zone()` 回收各个 zone 的相关页面。首先使用时钟算法对存放内存映射文件的页面缓存(Page Cache)和块缓存(Buffer Cache)使用的页面进行扫描。如果在这些缓冲区中释放的页面数量没有达到要求,继续使用时钟算法置换进程共享内存占用的部分页面。如果找到一个替换对象,要将所有共享此页的进程相对应的 PTE 改为无效,同时共享计数也要减到 0。这时就可将该共享页写入交换

空间。

如果这种方法还未释放足够的物理页面的话,再通过扫描各用户进程的当前位于内存的页面。如果一个页面已被加锁,或标记为保留,或是用于 DMA 操作,那么该页是不能被换出的。在换出这样的页面时,如果是一个“脏”页,就要将其回写到交换空间。否则可直接将其丢弃,不需要写回交换空间。同时,对换出页面的 PTE 有效位要清零。

无论使用哪种方式,被释放的物理页面将被加入到 free_area 数组所标识的空闲页面链表中。当 kswapd 释放了足够的空闲页面后,kswapd 将再次睡眠到下次定时器到时。当 kswapd 试图释放物理页面时,它将记录上次使用的最后一种方法(页面 cache、共享内存还是进程用户空间),下一次运行时它会首先运行上次最后成功的方法。

4.7.6 交换 Cache

当将页面交换到交换空间时,Linux 总是避免页面写,除非必须这样做。当页面已经被交换出内存但是当有进程再次访问时又要将它重新调入内存。只要页面在内存中没有被写过,则交换空间中的复制是有效的。

Linux 使用交换缓存来跟踪这些页面。这个交换缓存是一个页表入口链表,每个对应于系统中的物理页面。这是一个对应于换出页面的页表入口并且表示页面放置在哪个交换空间中以及在交换空间中的位置。如果交换缓存入口为非 0 值,则表示在交换空间中的这一页没有被修改。如果此页被修改(或者写入),则其入口从交换缓存中删除。

当 Linux 需要将一个物理页面交换到交换空间时,它将检查交换缓存,如果对应此页面存在有效入口,则不必将这个页面写到交换空间中。这是因为自从上次从交换空间中将其读出来,内存中的这个页面还没有被修改。

交换缓存中的入口是已换出页面的页表入口。它们虽被标记为无效但是为 Linux 提供了页面在哪个交换空间中以及文件中的位置等信息。

4.7.7 页面的换入

保存在交换空间中的页可能被再次使用到,例如,当应用程序向包含在已交换出物理页面上的虚拟内存区域写入时。由于已被交换出去,此时表示此页面的页表入口被标记成无效。处理器不能处理这种虚拟地址到物理地址的转换,所以它将控制传递给操作系统,同时通知操作系统页面错误的地址与原因。这些信息的格式以及处理器如何将控制传递给操作系统与具体硬件有关。

处理器相关页面错误处理代码将定位表示包含出错虚拟地址对应的虚拟内存区域的 vm_area_struct 数据结构。它通过在此进程的 vm_area_struct 中查找包含出错虚拟地址的位置直到找到为止。这些代码与系统性能关系重大,进程的各 vm_area_struct 数据结

构特意组织成红黑树,以使查找操作时间更少。

执行完这些处理器相关操作并且找到出错虚拟地址的有效内存区域后,页面错处理过程其余部分和前面类似。

通用页面错处理代码为出错虚拟地址寻找页表入口。如果找到的页表入口是一个已换出页面,Linux 必须将其换入物理内存。已换出页面的页表入口的格式与处理器类型有关,但是在所有的处理器平台上,这些页面被标记成无效,并把定位此页面的必要信息放入页表入口中。Linux 利用这些信息以便将页面换入物理内存。

此时,Linux 知道出错虚拟内存地址,并且拥有一个包含页面位置信息的页表入口。通过 `do_swap_page()` 函数,系统分配物理页面并将已换出的页面读入。关于页面在交换空间中位置信息,可从页表项中取出。

如果引起页面错误的访问不是写操作,则页面被保留在交换缓存中,并且它的页表入口不再标记为可写。如果页面随后被写入,则将产生另一个页面错误,这时页面被标记为“脏”,同时其入口从交换缓存中删除。如果页面没有被写,并且被要求重新换出,Linux 可以免除这次写,因为页面已经存在于交换空间中。

如果引起页面从交换空间中读出的操作是写操作,这个页面将被从交换缓存中删除并且其页表入口被标记成“脏”且可写。

4.8 本章小结

在计算机存储体系中,内存位于处理器(高速缓存)和外存之间的特殊位置,是程序和数据得以处理的必经之道,因此,其管理效率是整个系统性能的关键所在。

内存管理的主要任务是为系统中运行的操作系统、进程或作业提供合适的存放空间,实现相互之间的共享和保护,并通过各种技术扩充内存容量,使得内存中能够同时存放更多的进程,提高内存的利用率和系统的并发性。

程序编译时生成的逻辑地址,在运行时需要能够正确的映射到放置程序内容的物理地址,也就是地址重定位。静态重定位是指在程序运行之前就完成了地址的映射,是由专门设计的重定位装配程序来完成的;而动态重定位是指在目标程序运行的过程中再实现地址的映射,通常需要硬件地址转换机构的支持。

分区法是连续的内存管理技术,即一个作业或进程放在一个连续的地址空间。按照分区的划分方式,可分为分区大小固定的固定分区法和分区大小可变的动态分区法。如果允许在程序执行的过程中从一个分区移动到另一个分区,则需要可重定位分区法。在移动过程中,专门的移动程序需要完成作业或进程的地址重定位。分区法采用的都是静态重定位方式。分区法可以采用覆盖和交换技术达到扩充内存的目的。覆盖技术是让不同时调用的子程序模块共享同一内存区,而交换技术则是将暂时不用的信息交换到外存,

需要时再换入。由于各作业和进程连续存储,因此分区法使得作业或进程间的程序和数据的共享变得非常困难。分区法采用的是越界保护。

分页、分段和段页式技术则是非连续的内存管理技术,允许作业或进程分散在不连续的内存块,都需要采用动态重定位技术。为了达到扩充内存的目的,分页、分段和段页式技术可以采用虚拟存储技术。虚拟存储技术利用程序的局部性原理,通过请求调入和交换功能,对内外存进行统一管理,使外存和内存通过作业或进程的虚拟空间有机地结合起来。同分区法一样,分页技术实现共享较难,而分段和段页式因为与程序的逻辑信息内容相符,因此,实现共享比较容易。分页、分段和段页式技术采用的保护技术是越界保护和存取权控制保护相结合的方式。

在通用操作系统中,目前广泛采用的是基于页式和段页式的虚拟存储技术。Linux操作系统采用的是页式虚拟存储技术。

思 考 题

1. 虚拟存储技术的实现需要硬件机制的支持,i386 架构的处理器设置了哪些用于支持虚拟存储技术的相关硬件机制,它们的作用分别是什么?

2. 应用网关在目前的信息安全领域有着广泛的应用前景。其主要作用是对通过网络到达的信息内容进行分析,来过滤和阻止或者发现一些特定条件的信息。要做到这一点,需要对到达网关的所有数据进行解包组合后才能进行。因此,需要在作为网关的计算机中缓存大量的数据。同时,因为 Linux 的网络协议栈在内核中实现,为了保证效率,应用网关软件通常也在内核中实现。在 32 位平台上,Linux 进程的内核空间只有 1GB,除内核本身使用外,剩余的空间对于网络流量较大的应用网关对缓存空间的需求来说,就显得非常狭小。请根据虚拟存储空间的相关原理,试想一下,可通过哪些途径来“扩大”32 位平台上进程的内核空间(如 4GB),以满足应用网关的空间需求。

3. 首先试描述 Linux 内核是如何利用 page 结构体来管理物理内存页的。在此基础上,假设一台 32 位的 i386 服务器(CPU 支持 PAE 功能)拥有 8GB 的物理内存,那么运行于其上的 Linux 内核需要多大的空间来存储 page 结构体呢? 通过以上情况,分析 Linux 这种物理内存页描述机制的优缺点。

4. 分段、分页和段页式是非常重要的虚拟存储技术,分别试述其工作原理和优缺点。Linux 实际上只采用了分页技术,而将需要由分段技术解决的问题留给了编译(连接)器。毕竟程序是一次编译(连接),多次运行的。试述这种策略的优缺点。

5. Buddy 算法的工作过程如何? 试分析其优缺点。为什么 Linux 内核要同时在其管理数据结构中同时采用链表和位示图?

6. SLAB 可以看成是内核中一些专用数据结构的缓存机制,其染色技术的作用是什么?

么？试思考内存管理与 cache 的关系，操作系统需要管理 cache 吗？为什么？Linux 进程的 PCB(task_struct)并没有采用 SLAB，这样做的理由有哪些？

进一步阅读

关于 Linux 内核空间的扩展问题，可以参阅 *Understanding the Linux Kernel, 3rd Edition* 的第 8 章。关于 Linux 所使用的虚拟存储技术，可参阅《Linux 源代码情景分析》的第 2 章。关于 SLAB 机制和 cache 的关系，可参阅 *Understanding the Linux Kernel, 3rd Edition* 的第 8 章。



第5章 文件系统

在计算机的使用过程中,用户需要存储大量的数据。在早期的计算机系统中,用户自行根据需要对信息的存储进行管理。用户直接对辅存进行操作,进行信息的存储。随着多道程序、分时系统的出现,多个用户需要同时共享辅存,多个用户需要对辅存同时进行操作。用户间的冲突无法避免,自行管理辅存的复杂性也开始增加。

文件系统的出现是为了解决多用户存储、管理信息时出现的问题。文件系统是操作系统中负责存取和管理信息的模块,它用统一的方法管理用户和系统信息的存储、检索、更新、共享和保护,并为用户提供一套完整的文件使用和操作方法。从用户的角度看,文件系统提供了抽象的逻辑文件的管理与使用。逻辑文件是由文件名标识的一组信息的集合。用户通过文件系统检索到逻辑文件后,可以根据逻辑文件的文件结构进行信息的访问与存储。用户所有的操作都是基于逻辑文件的,文件系统最终需要将用户对逻辑文件的操作转换成对物理文件的操作。物理文件可以是存储设备上的存储区域,如磁盘上的一组磁盘块,即用户对逻辑文件的操作将被文件系统转换成对磁盘块的操作。在现代操作系统中,物理文件的范围得到了扩大,物理文件也可以是一个设备、管道或套接字,文件系统将用户对文件的操作转换成用户对设备的操作、用户间的通信操作和网络操作。

文件系统的功能是在逻辑文件与物理文件、逻辑地址与物理地址、逻辑结构与物理结构之间实现转换,并做到存取速度快,存储空间利用率高,数据可共享,安全可靠性好。操作系统必须考虑文件目录的建立和维护、存储空间的分配和回收、数据的保密和保护、监督和控制用户存取和修改文件的权限、处理在不同存储介质上信息的表示方式、编址方法、存储次序及怎样检索用户信息等问题。

5.1 文 件

5.1.1 逻辑文件与物理文件

在文件系统中,用户所面对的文件是逻辑文件。逻辑文件是由文件名标识的一组信息的集合。文件名是一个字符串,字符串的格式与长度在各个操作系统都有不同的规定。

用户在逻辑文件上对信息进行存储、操作。在文件系统中,逻辑文件并不真实存在;每一个逻辑文件都需要有一个实现,称为物理文件。用户对逻辑文件的所有操作都将被文件系统转换成对物理文件的操作。在早期的文件系统中,物理文件是真实的信息载体,

即是存储设备上的存储区域。用户对逻辑文件的操作被转换成对存储区域的操作。随着文件系统的发展,物理文件的范围得到扩大。物理文件可以是一个设备,用户对逻辑文件的操作被转换成用户对设备的操作。物理文件还可以是管道、套接字等虚拟的设备。物理文件甚至还可以是另外一个逻辑文件,用户对一个逻辑文件的操作可以被转换成对另一逻辑文件的操作。逻辑文件与物理文件的分层关系如图 5.1 所示。



图 5.1 逻辑文件与物理文件

逻辑文件与物理文件的分层结构增加了文件系统的灵活性。用户在存储数据的时候,只需要关心逻辑文件的访问方式、结构;无需考虑信息在物理文件上的存储结构。分层结构使得物理文件与用户完全隔离,文件系统可以在更改物理文件结构时不影响用户的使用。当文件存放位置作了改变,甚至更换了文件的存储设备,对使用者也没有丝毫影响。在逻辑文件、物理文件的两层结构中,文件系统可以在两层之间插入各种安全、保密和保护措施,可以防止对文件信息的有意或无意的破坏或窃用。此外,文件系统还可以在其中加入备份功能,当文件使用过程中出现硬件设备故障时,系统可组织重执或恢复,对于因硬件失效而可能造成的信息破坏,可组织转储以提高文件的可靠性。文件系统还可将多个逻辑文件映射到同一物理文件,方便用户共享,节省文件存储空间,提高了文件空间的利用率。

5.1.2 逻辑文件的组织

用户在使用计算机的过程中将会用到多个逻辑文件。文件系统需要将逻辑文件进行组织,方便用户检索。

最简单的组织形式是一级目录结构。文件系统将所有的逻辑文件都放置在同一层目录中,缺点是文件重名难以解决。比较好的组织方法是将逻辑文件组织成多级的层次结构,形成一个树形结构。在树形结构中,每一个非叶结点称为目录,叶结点称为逻辑文件。每一个目录下都可以包含多个子目录和多个逻辑文件。树型多级目录结构有许多优点:较好地反映了现实世界中具有层次关系的数据集合和系统内部文件的分支结构;不同逻辑文件可以重名,只要它们位于不同目录中;易于规定不同层次或子目录中逻辑文件的不同存取权限,便于文件的保护;易于系统的维护和查找。图 5.2 给出了 Linux 的目录层次结构。

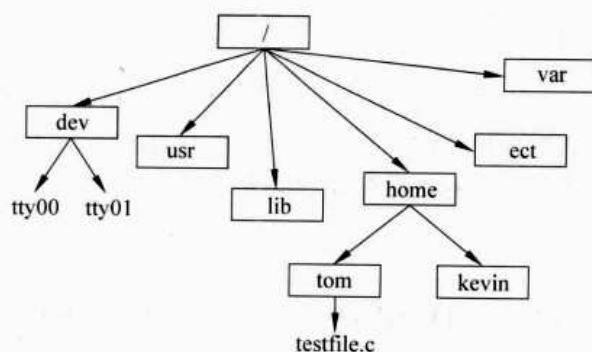


图 5.2 Linux 目录层次结构

在树形结构中,用户对逻辑文件的访问需要包含逻辑文件所在的路径名。路径是指树型结构中的某个目录层次到某个文件的一条道路。此路径的主要构成是目录名,中间一般用“/”或“\”分割。用户对逻辑文件的访问要给出文件所在相对路径或绝对路径。绝对路径是指从根目录开始的路径名。相对路径和工作目录一起使用。用户可以指定一个目录作为当前的工作目录。这个时候,所有的路径名如果不是从根目录开始,则都是相对于工作目录的。例如,某进程的工作目录是/home,当该进程用相对路径 tom/testfile. c 访问一个文件,则该文件的绝对路径为/home/tom/testfile. c。

5.1.3 逻辑文件的结构

用户需要根据逻辑文件的结构进行信息的管理和存储。逻辑文件的结构是独立于物理环境加以构造的,文件系统提供若干操作以使用户构造逻辑文件,这样,用户不必顾及物理文件的构造,而只需了解逻辑文件的结构,利用文件名和有关操作就能存储、检索和处理信息。

逻辑文件的结构分两种形式:记录式文件和流式文件。

(1) 记录式文件:是一种有结构的文件,它包含若干逻辑记录,逻辑记录是文件中按信息在逻辑上的独立含意划分的一个信息单位,记录在文件中的排列按出现的次序编号,记录 0、记录 1……

(2) 流式文件:指文件内的数据不再组成记录,只是一串信息的集合,又称字节流文件。流式文件中的每个字节都有一个索引,第一个字节的索引为 0,第二个字节的索引为 1,依此类推,打开文件的进程使用文件读写位置来访问文件中的特定字节,当文件打开时,文件读写位置指向首字节,每 k 个字节读或写操作完成则将文件读写位置增加 k。大多数现代操作系统如 Linux 对用户只提供流式文件。

5.1.4 逻辑文件的存取

用户在检索到逻辑文件之后,需要访问逻辑文件中的信息。存取方法是文件系统为应用程序提供的访问文件的方法和手段。由于物理存储设备有不同的特性,因此对于逻辑文件有着不同的存取方法。主要的存取方法有顺序存取、直接存取和索引存取。

1. 顺序存取

采用顺序存取方法时,系统设置读和写两个位置指针,指向要读出或写入的字节位置或记录位置,存取操作都是在上次的基础上进行。读操作总是读出位置指针指向的若干字节或下一个记录,写操作将若干字节或一个记录写到写指针指向的位置,根据读出或写入的字节个数或记录号,系统自动修改相应指针的位置。顺序存取主要用于磁带文件,但也适用于磁盘上的顺序文件。

2. 直接存取

很多应用场合要求快速地以任意次序直接读写某个记录。为了实现直接存取,一个文件可以看作由顺序编号的物理块组成的,这些块常常划成等长,作为定位和存取的一个最小单位,如一块为 1024B、4096B。用户可以请求读块 22、然后,写块 48,再读块 9 等,直接存取文件对读或写块的次序没有限制。用户提供给操作系统的是相对块号,它是相对于文件开始位置的一个位移量,而绝对块号则由系统换算得到。对于记录式文件,要为每个文件记录指定关键字,可通过关键字映射直接检索和存取文件信息。

3. 索引存取

第三种类型的存取是基于索引文件的索引存取方法。由于文件中的记录不按它在文件中的位置,而按它的记录名或记录键来编址,所以,用户提供给操作系统记录名或记录键后,先搜索记录名或记录键,再查找到所需记录。采用记录键时应按某种顺序存放,例如,按字母先后次序来排序,对于这种文件,除可采用按键存取外,也可以采用顺序存取或直接存取的方法。信息块的地址都可以通过查找记录键而换算出。实际的系统中,大都采用多级索引,以加速记录查找过程。

5.1.5 物理文件的结构

逻辑文件的结构与存储方法通常需要考虑其所对应的物理文件的特性与结构。物理文件可以是物理存储设备上的一片存储空间、外部设备、套接字和管道等。如果物理文件是物理存储设备上的一片存储空间,文件系统往往根据存储设备类型、存取要求、记录使用频率和存储空间容量等因素提供若干种存储结构,把逻辑文件以不同方式保存到物理

存储设备的介质上去。物理文件的存储结构涉及块的划分、记录的排列、索引的组织 and 信息的搜索等许多问题。

有两类方法可用来构造物理文件的结构。第一类称计算法,实现原理是设计一个映射算法,通常用线性算法、杂凑法等,通过对记录键的计算转换成对应的物理地址,从而找到所需记录。直接寻址文件、计算寻址文件和顺序文件均属此类。计算法的存取效率高,又不必增加存储空间存放附加控制信息,能把分布范围较广的键均匀地映射到一个存储区域中。第二类称指针法,这类方法设置专门指针,指明相应记录的物理地址或表达各记录之间的关联。索引文件、索引顺序文件、连接文件和倒排文件等均属此类。使用指针的优点是可将文件信息的逻辑次序与在存储介质上的物理块排列次序完全分开,便于随机存取,便于更新,能加快存取速度。但使用指针要耗用较多存储空间,大型文件的索引查找要耗用较多处理机时间,所以,究竟用哪种文件存储结构,必须根据应用目标、响应时间和存储空间等多种因素进行权衡折中。下面介绍常用的几种物理文件的结构和组织方法。

1. 顺序文件

将一个文件中逻辑上连续的信息存放到存储介质的依次相邻的物理块上形成顺序结构,叫顺序文件,又称连续文件。显然,这是一种逻辑记录顺序和物理块顺序完全一致的文件,通常,记录按出现的次序被顺序读出或修改。文件控制块中要保存的磁盘定位信息很简单,它由第一个物理块地址和文件信息块的总块数组成。为了改善顺序文件的处理效率,用户常常对顺序文件中的记录按某一个或几个数据项的值从小(大)到大(小)重新排列,经排列处理后,记录有某种确定的顺序,成为有序的顺序文件,能较好地适应批处理等顺序应用。

顺序文件的基本优点是:顺序存取记录时速度较快。所以,批处理文件,系统文件用得很多。采用磁带存放顺序文件时,总可以保持快速存取的优点;若以磁盘作存储介质时,顺序文件的逻辑记录也被存于邻接的物理块中,因而,顺序的磁盘文件也能像磁带文件一样进行严格的顺序处理。然而,由于多个程序访问时,在同一时间段内另外的应用程序可能驱动磁头移向了其他文件,因而,会影响访问下一个要处理的连续记录和块的速度。顺序文件的主要缺点是:建立文件前需要能预先确定文件长度,以便分配存储空间;修改、插入和增加文件记录有困难;对可变长记录的处理很困难;磁盘作连续分配,会造成空闲块的浪费。

2. 连接文件

连接结构的特点是使用连接字(又称指针)来表示文件中各个记录之间的关系。如图 5.3 所示,文件信息存放在辅存的若干个物理块中,第一块文件信息的物理地址由文

件目录给出,而每一块的连接字指出了文件的下一个物理块位置,通常,连接字内容为 0 时,表示文件至本块结束。这种文件叫连接文件,又称串联文件。

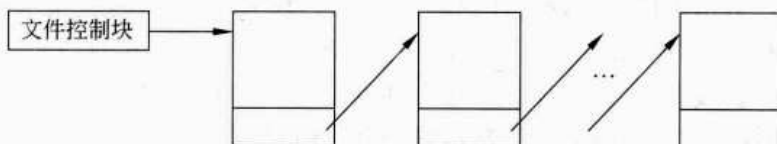


图 5.3 连接文件结构示意图

使用指针可以将文件的逻辑记录顺序与它所在存储空间的物理块顺序完全独立开来,即存放信息的物理块不必连续,而借助于指针表达记录之间的逻辑关系;连接结构必须将连接字与数据信息混合存放,破坏了物理块的完整性;存取信息须通过缓冲区,待获得连接字后,才能找到下一物理块的地址,因而,仅适宜于顺序存取。连接结构恰好克服了顺序结构不适宜于增、删、改的缺点。

3. 直接文件

使用连接文件很容易把数据记录组织起来,但是查找某个记录需遍历链接结构,效率很低,另一种技术,称为散列法或杂凑(hash)法可以解决效率问题。在直接存取存储设备上,利用 hash 法把记录的关键字与其地址之间建立某种对应关系,以便实现快速存取的文件叫直接文件或散列文件。采用 hash 技术需要建立 hash 表,这是一个指针数组,数组通过索引访问,索引是与数据记录有关的关键字,而记录在介质上的位置是通过对记录的键施加变换来获得相应地址。这种存储结构用在不能采用顺序组织方法,次序较乱,又需在极短时间主存取场合。

计算寻址结构中的难点是“冲突”问题。一般说来,地址的总数和可能选择的关键字之间不存在一一对应关系。因此,不同的关键字可能变换出相同的地址来,这就叫“冲突”。一种散列算法是否成功的重要标志是将不同键映射成相同地址的几率有多大,几率越小冲突就越小,则此散列算法的性能也越好。解决冲突会增加额外代价,因而,“冲突”是计算寻址结构性能变坏的主要因素。解决冲突的办法叫溢出处理技术,这是设计散列文件需要考虑的主要内容。常用的溢出处理技术有顺序探查法、两次散列法、拉链法和独立溢出区法等。

4. 索引文件

索引结构是实现非连续存储的另一种方法,适用于数据记录保存在随机存取存储设备上的文件。如图 5.4(a)和图 5.4(b)所示,系统为每个文件建立索引表(index table),可有不同索引形式,一种是它记录了组成指定文件的磁盘块号,所以,这种索引表只是

盘块号的序列,适用于流式文件。另一种是它的索引表项包含记录键及其磁盘块号,适用于记录式文件。

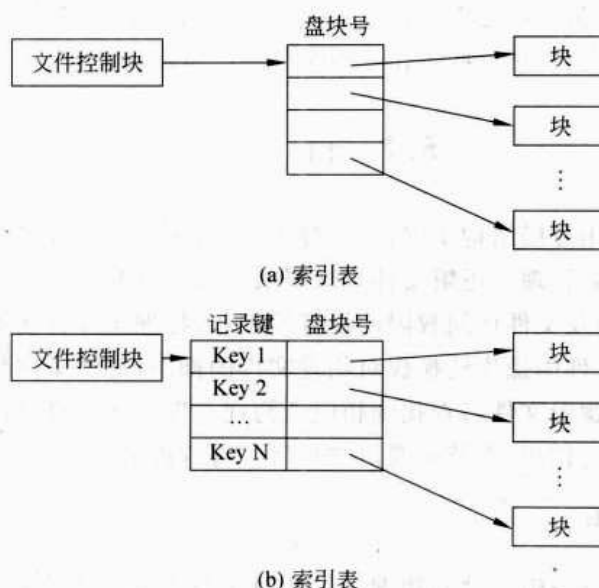


图 5.4 两种索引文件结构示意图

索引结构是连接结构的一种扩展,除具备连接文件的优点外,记录可以散列存储,具有直接读写任意记录的能力,便于信息的增、删、改。缺点是增加了索引表的空间开销和查找时间,大型文件的索引表的信息量甚至可能远远超过文件记录本身的信息量。

索引文件中的索引项可分为两类:一类稠密索引、对每个数据记录,在索引表里都有一个索引项。因而,索引表本身很大,但不要求数据记录排序,通过对索引的依次查找可确定记录地址或记录是否存在。另一种称稀疏索引,它对每一组数据记录保存一个索引项。因而,索引表本身较小,但数据记录必须按某种方式排序,在查找时要花费代价,因为,找到索引之后,只判定了记录所在的组,而该记录是否存在?是组内哪一个记录?还要进一步搜索。

索引顺序文件是顺序文件的扩展,其中各记录本身在介质上也顺序排列,例如,按字母次序排列,由于有随机访问的关键字,具有直接处理和修改记录的能力。索引顺序文件能像顺序文件一样进行快速顺序处理,既允许按物理存放次序(记录出现的次序);也允许按逻辑顺序(记录键决定的次序)进行处理。

有时记录数目很多,索引表要占用许多物理块,在查找某键对应的索引项时,可能依次需交换很多块,若索引表占用 n 块,则平均要交换 $(n+1)/2$ 次,才能找到所需记录的物理地址,当 n 很大时,这是很费时间的操作。提高查找速度的一种办法是做一个索引的索引,叫二级索引。二级索引表的表项列出一级索引表每一块最后一个索引项的记录键及

该索引表的盘块号,也就是说,若干个记录的索引本身也是一种记录。查找时先查看二级索引表找到某键所在的索引表盘块号,再搜索一级索引表按键找出数据记录。当记录数目十分大时,索引的索引也可能占用许许多多块,那样,可以做三级索引,有的计算机系统还建立更多层次的索引,当然这些工作都由文件系统自动完成。

5.2 目 录

文件系统通常采用分层结构实现,大致分成三层:逻辑文件管理、物理文件管理和逻辑文件与物理文件映射管理。逻辑文件管理层实现逻辑文件的结构与组织,为用户提供各种文件系统调用,以及文件访问权限的设置等工作;物理文件管理主要负责物理文件的操作,将用户对逻辑文件的操作转换成对物理文件的操作;逻辑文件与物理文件映射管理负责根据用户提供的逻辑文件名查找到相应的物理文件。文件目录在逻辑文件管理和映射管理中都起着重要的作用,本节主要对文件目录进行讲解。

5.2.1 文件控制块

文件系统中存在着逻辑文件和物理文件。对于逻辑文件和物理文件,文件系统都需要对每一个文件记录相应的管理信息,这样的数据结构被称为文件控制块(File Control Block,FCB)。

在物理文件控制块中,系统记录的主要信息如下。

- 文件控制信息。如用户名、文件主存取权限、授权者存取权限、文件口令、文件属性,是读写文件、执行文件,还是只读文件等。
- 文件使用信息。已打开文件的进程数,文件被修改的情况,共享信息说明,文件最大长度和当前大小等。
- 文件管理信息。如文件建立日期、文件最近修改日期、文件最近访问日期、文件保留期限和记账信息等。
- 物理文件的类型。如是设备文件、普通文件和套接字文件等。
- 物理文件的结构信息。如文件所在设备名,文件物理结构类型,记录存放在辅存的盘块号或文件信息首块盘块号,也可指出文件索引的所在位置等。

在逻辑文件控制块中,系统可以记录的主要信息如下。

- 逻辑文件的标识符。逻辑文件的文件名。
- 逻辑文件的结构信息。逻辑文件的结构,如记录类型、记录个数、记录长度和成组因子数等。

与物理文件类似,逻辑文件也可以包含文件的控制信息、使用信息和管理信息。逻辑文件控制块和物理文件控制块的内容有着许多重复的地方,并且一个逻辑文件一定对应

一个物理文件。因此在一般文件系统的设计中,逻辑文件控制块的信息被简化成只包含逻辑文件名信息,而其他的所有信息都通过读取对应的物理文件控制块中的信息获得。如用户向文件系统请求读取一个逻辑文件的创建日期,文件系统读取该逻辑文件所对应的物理文件的创建日期后返回给用户。在这样简化后,通常在文件系统中所指的文件控制块是指物理文件控制块。

5.2.2 目录、文件目录与目录文件

逻辑文件的组织结构是一个树状结构。文件系统用一定的数据结构记录下这样的结构。这样的数据结构是文件目录。文件系统还用文件目录记录逻辑文件到物理文件的映射。

在逻辑文件的组织中,每一个目录都可以包含子目录和逻辑文件。目录可以看成是一种具有特殊结构的逻辑文件,该逻辑文件中记录了所包含的子目录和逻辑文件的信息。记录这些信息的数据结构就是文件目录。每一个目录都对应着一个文件目录。文件目录包含许多目录项,该目录包含的每一个逻辑文件和目录各对应一个目录项。每一个目录项包含了一个逻辑文件名和一个物理文件控制块。一个目录项指明了一个逻辑文件到物理文件的映射。

目录是一种特殊的逻辑文件,这种逻辑文件的结构就是文件目录。每一个逻辑文件都要有一个物理文件与之对应,目录对应的物理文件就是目录文件。目录文件在文件系统存储和普通的物理文件相同。与普通文件不同之处在于,目录文件永远不会空,因为它至少包含两个目录项:当前目录项“.”和父目录项“..”。

图 5.5 是一个文件目录的示意图。将物理文件控制块放入文件目录项中,会给文件系统造成以下两个缺点。

逻辑文件名	文件控制块
文件 A	文件控制块 A
文件 B	文件控制块 B
目录 C	文件控制块 C
...	...

图 5.5 文件目录结构示意图

(1) 浪费内存。文件系统通过检索目录中的内容对逻辑文件进行查找。在检索的过程中,文件系统需要将目录项读入内存,查找逻辑文件对应的物理文件控制块。在这一过程中,读入所有的物理文件控制块是没有必要的,对内存造成不必要的浪费。

(2) 不方便文件共享。不同的逻辑文件可以指向同一个物理文件。一个物理文件只能有一个物理文件控制块。如果将物理文件控制块放入目录项,当不同逻辑文件指向同一个物理文件时,该物理文件控制块会存在两个文件目录项中。保持物理文件控制块的同步会增加文件系统的负担,影响文件系统的效率。

为了解决这些问题。UNIX/Linux 引入了一种新的方法。在 UNIX/Linux 中,文件控制块被称为 inode。该方法将所有 inode 集中存放在磁盘中的一块区域中。在文件目

录项中只保存文件名和 inode 的索引,即目录项只保存文件名和 inode 指针。采用这种方法,当文件系统读入目录项的时候,只需要读入 inode 索引,不需要读入所有的文件控制块,节约了内存。如果两个逻辑文件需要映射到同一个物理文件,只需要将 inode 索引置成相同值即可,如图 5.6 所示。文件系统还可以将最近访问过的 inode 放入主存的缓冲区。当系统要访问 inode 时,系统可以根据 inode 索引号先在缓冲区中查找,减少系统访问磁盘的次数,提高系统的效率。

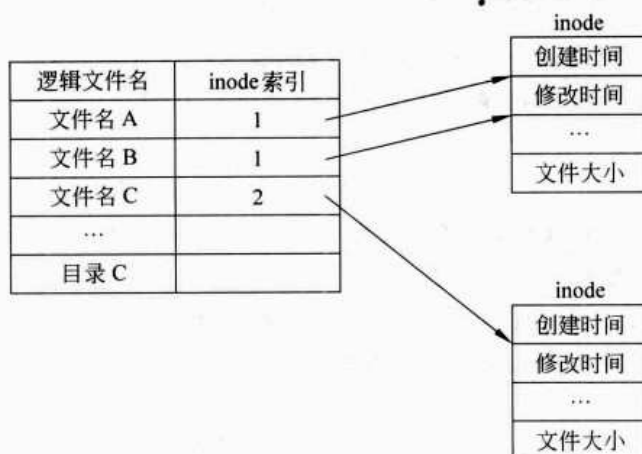


图 5.6 采用 inode 索引的文件目录示意图

目录、文件目录与目录文件三个概念容易混淆,在最后进行一下小结。

- 目录。在逻辑文件的树状组织结构中,非叶结点被称为目录,包含着子目录和逻辑文件。目录是一个具有特殊结构的逻辑文件。该逻辑文件中记录着包含的子目录与逻辑文件的信息。
- 文件目录。文件目录是一种数据结构。目录是一种逻辑文件,该逻辑文件的结构为文件目录。
- 目录文件。目录文件是一种存储在磁盘上的物理文件,一个文件目录对应一个目录文件。

5.2.3 文件检索

文件目录使得用户可以“按名存取”。当用户访问逻辑文件时,系统根据用户提供的文件路径名,从根目录或当前工作目录开始,逐级查找路径名中的各子目录名,用它们作为索引,逐层搜索各级目录文件,最后找到匹配的文件目录项。文件系统根据文件控制块中的物理文件信息对物理文件进行访问。

下面是目录查找的一个例子。假设用户进程想要打开文件/home/tom/myfile.c,文件

系统开始搜索,首先,遇到根目录“/”,根目录通常放在磁盘的固定盘块中,根据主存活动 inode 表中根目录的活动 inode,把它作为当前工作索引节点并将其第一个物理块读入主存缓冲区,接着读入路径的第一个分量字符串 home。文件系统对根目录内容进行搜索,若找不到依次读入第二、第三个物理块……进行比较,直到找着 home 的 inode 号;检查主存活动 inode 表,若找不到 home 的 inode,为它分配一个活动 inode,由于每个 inode 位于磁盘上分配好了的固定位置,可从磁盘上 home 的 inode 中装入它的内容。否则直接查找 home 主存活动 inode,通过属性查明 home 为子目录,经核符合访问权限,把它作为当前的工作索引节点,读入路径的第二个分量字符串 tom。类似地读入子目录 tom 目录文件的物理块能找到 myfile.c 的 inode 号。文件系统为该文件在主存 inode 表中分配一个主存活动 inode,从磁盘的 myfile.c 的 inode 中装入内容。中间任何一步出错便返回错误信息,由于路径名分析完毕,这时修改填写 inode 的有关信息,打开文件操作到此结束。

类似于文件操作,文件系统也提供了一系列处理文件目录的系统调用和操作命令,有的操作系统甚至允许应用程序像对文件一样打开、读写和操作目录。与文件目录有关的操作有创建目录、删除目录、列目录、改变目录、移动目录、重命名目录、改变保护、目录链接和查找目录等。

5.3 文件存储空间管理

在磁盘等大容量辅存空间中,用户作业运行期间常常要建立、扩充和删除文件,文件系统应能自动管理和控制辅存文件空间,创建和扩充文件时,决定分配哪些磁盘块是很重要的,这会影响今后磁盘访问次数,删除文件或缩短长度时,需回收磁盘块并把盘块移入空闲队列中。

辅存文件空间的有效分配和释放是文件系统应解决的一个重要问题。最初,整个存储空间可连续分配给文件使用,随着用户文件不断建立和撤销,存储空间中会出现“碎片”。系统应定时或根据命令要求集中碎片,在收集过程中往往要对文件重新组织,让其存放到连续存储区中。辅存文件空间分配常采用以下两种办法。

- 连续分配。文件被存放在辅存空间连续存储区(连续的物理块号)中,在建立文件时,用户必须给出文件大小,然后,查找到能满足的连续存储区供使用;否则文件不能建立,用户进程必须等待。连续分配的优点是顺序访问时通常无需移动磁头,查找速度快,管理简单,但为了获得足够大的连续存储区,需定时进行“碎片”收集。因而,不适宜于文件频繁进行动态增长和收缩的情况,用户事先不知道文件长度也无法进行分配。
- 非连续分配。一种方法是以块(扇区)为单位,按文件动态要求分配给它若干扇区,这些扇区不一定要连续,属于同一文件的扇区按文件记录的逻辑次序用链指

针连接或用位示图指示。另一种方法是以簇为单位,簇是若干个连续扇区组成的分配单位;实质上是连续分配和非连续分配的结合。各个簇可以用链指针、索引表和位示图来管理。非连续分配的优点是辅存空间管理效率高,便于文件动态增长和收缩,访问文件执行速度快,特别是以簇为单位的分配方法已被广泛使用。下面介绍常用的几种具体文件辅存空间管理方法。

(1) 位示图。

磁盘空间通常使用固定大小的块,可方便地用位示图管理。用若干字节构成一张位示图,其中每一字位对应一个物理块,字位的次序与块的相对次序一致,字位为 1 表示相应块已占用,字位为 0 表示该块空闲。主要优点是:每个盘块仅需 1 个附加位,如盘块长 1KB,位示图开销仅占 0.012%;可把位示图全部或大部分保存在主存,再配合现代机器都具有的位操作指令,实现高速物理块分配和回收。

(2) 空闲区表。

这种分配方法常常用于连续文件,将空闲存储块的位置及其连续空闲的块数构成一张表。分配时,系统依次扫描空闲区表,寻找合适的空闲块并修改登记项。删除文件释放空闲区时,把空闲区位置及连续的空闲区长度填入空闲区表,出现邻接的空闲区时,还需执行合并操作并修改登记项。空闲区表的搜索算法有优先适应、最佳适应和最坏适应算法等。

(3) 空闲块链。

把所有空闲块连接在一起,系统保持一个指针指向第一个空闲块,每一空闲块中包含指向下一空闲块的指针。申请一块时,从链头取一块并修改系统指针;删除时释放占用块,使其成为空闲块并将它挂到空闲链上。这种方法效率低,每申请一块都要读出空闲块并取得指针,申请多块时要多次读盘,但便于文件动态增长和收缩。

(4) 成组空闲块链。

图 5.7 给出了 UNIX/Linux 采用的空闲盘块成组连接法。存储空间分成 512B 一块。为方便讨论,假定文件卷启用时共有可用文件 338 块,编号为 12~349。每 100 块划

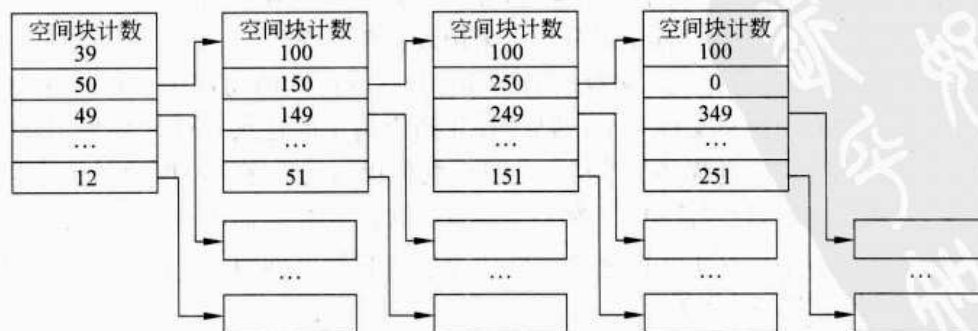


图 5.7 UNIX/Linux 系统的空闲块成组连接示意图

分一组,每组第一块登记下一组空闲盘块号和空闲总数。其中,50#~12#一组中,50#物理块中登记了下一组100个空闲盘块号150#~51#。同样下一组的第一块150#中登记了再下一组100个空闲盘块号250#~151#。注意,最后一组中,即250#块中第1项是0,作为结束标志,表明系统空闲盘块链已经结束。操作系统启动时,将磁盘专用块复制到主存系统工作区中,多数情况访问主存可完成申请或释放空闲盘块操作。

分配空闲盘块时,总是先把专用块中空闲块计数减1,并以它为指针找到专用块的相应表项,该内容就是要分配的空闲块号。当空闲块计数减1后等于0时,专用块中仅剩1个盘块号,此时要取出该表项中的盘块号*i*,再把此盘块中保存的下一组空闲盘块链接信息经缓冲区复制到主存专用块中,然后,再把当前盘块*i*分配出去。

释放存储块时,将块号记在由专用块所指示的表项中,然后空闲块计数加1。若发现此表已满(达100),则应把整个表的内容经缓冲区复制到下面要释放的盘块中,然后将释放块,块号写入专用块中第一个位置,置空闲块计数为1。

搜索到全0块时,系统应向用户发出警告,表明空闲块已经用完。需要注意,开始时空闲块是按顺序排列的,但只要符合分组及组间连接原则,空闲块可按任意次序排列。事实上,经过若干次分配,释放操作后,空闲块物理块号必定不能按序排列了。

5.4 文件共享

文件共享是指不同进程共同使用同一文件。文件共享不仅是不同进程完成共同任务所必需的,而且还可以节省存储空间,减少同步错误。文件共享可以为静态共享或动态共享。在UNIX/Linux系统中,允许多个用户静态共享,或动态共享同一个文件,当一个文件被多个进程动态共享使用时,每个进程可以拥有各自的读写指针,也可以共用同一读写指针。

5.4.1 静态共享

文件的静态共享指的是将多个逻辑文件映射到同一个物理文件。当一个用户对其中的一个逻辑文件进行操作时,其他逻辑文件也被修改。由于这种共享关系不管进程是否正在工作,其文件的共享关系都是存在的,因此称它为静态共享。

文件静态共享主要有两种方式:硬链接法和符号链接法。

1. 硬链接法

文件目录项记录了逻辑文件到物理文件的映射关系。硬链接法是通过修改文件目录项把不同的逻辑文件映射到同一物理文件来实现文件的共享。在硬链接法中,需要共享的不同逻辑文件在文件目录项中具有相同的inode索引,通过此方法映射到同一个物理

文件。

在 UNIX/Linux 中,文件硬链接的系统调用形式为

```
link (oldnamep, newnamep);
```

其中,oldnamep 和 newnamep 分别为指向已存在文件名字符串和文件别名字符串的指针。这一系统调用的执行步骤如下。

(1) 检索目录,找到 oldnamep 所指向文件的 inode。

(2) 再次检索目录,找到 newnamep 所指文件的父目录文件,并把已存在文件的 inode 编号与别名构成一个目录项,记入到该目录中去。

(3) 把已存在文件 inode 的连接计数 i_nlink 加 1。

从上述过程可知,所谓链接,实际上是共享已存在文件的 inode。

解除硬链接的调用形式为 unlink (namep)。实际上,UNIX/Linux 中解除链接与文件删除执行的是同一段系统调用代码。删除文件是从文件属主角度讲的,而解除文件链接是从共享文件的其他用户角度讲的,不论删除还是解除链接,都要删去对应目录项,把 i_nlink 减 1。不过,只有当 i_nlink 减为 0 时,才是真正意义上的删除文件。

在实现静态链接时,为了反映共享同一文件的用户数,在每个索引节点中设立了一个变量 i_nlink。当文件第一次创建时,i_nlink 等于 1,以后,每次链接时就把 i_nlink 加 1。当用户删除文件或解除链接时,就把该计数值减 1,直到发现结果为 0 时,才释放文件的物理存储空间,从而真正删除这个文件。

2. 符号链接法

在 UNIX/Linux 中,每个分区有自己的文件目录树,当有多个文件系统时,可通过安装(mount)的办法整合成一棵更大的文件目录树。在每一个分区中,系统中每个物理文件有一个 inode,但是处于两个不同的磁盘或分区的物理文件可以含有相同的 inode 号。也就是说,整合的目录树中,inode 号并不唯一地标识一个文件。因而,无法做到从不同的文件系统生成指向同一个文件的链接,对于像 link 之类的系统调用,只能拒绝创建跨越文件系统的链接。

另外一种解决静态共享的方法是符号链接法。符号链接法中,文件系统中存在一种称为链接文件的物理文件。该种文件的存储方法和普通物理文件一致,但是链接文件中的内容是一个包含完整路径的逻辑文件名。当文件系统访问该链接文件时,文件系统会读入链接文件中包含的逻辑文件名,并再去访问该逻辑文件名所映射的物理文件。例如,假设用户 A 创建了一个逻辑文件/home/A/filea.c,如果用户 B 想用/home/B/filea.c 这个逻辑文件名访问 A 用户创建的文件。如果采用符号连接法,用户 B 创建一个名字/home/B/filea.c 的链接文件,该链接文件中的内容为/home/A/filea.c。当用户 B 访问

/home/B/filea.c 时,文件系统会自动访问/home/A/filea.c,实现文件共享。

硬链接是一种用 inode 引用文件的方法,而符号链接是一种用逻辑文件名引用文件的方法。采用符号链接的主要优点是:能用于链接计算机系统中不同文件系统中的文件,也可以用于链接目录;进一步可以链接计算机网络中不同机器上的文件,此时,仅需提供文件所在机器地址和该机器中文件的路径。

符号链接的主要缺点是:链接文件需要耗费额外的存储空间,多耗费一次路径查询的开销,并可能产生“死链”。符号链接只记录了共享的逻辑文件名,并没有类似硬链接记录了共享同一个文件的用户数。这样可能引起符号链接指向一个无效的逻辑文件。在本小节上文的例子中,如果用户 A 删除了/home/A/filea.c,用户 B 访问/home/B/filea.c 时就会出现错误,形成了一个“死链”。还有一种“死链”的情况是两个符号链接文件相互指向对方。继续刚才的例子,如果用户 A 在删除了/home/A/filea.c 后又创建了一个指向/home/B/filea.c 的符号链接/home/A/filea.c,这样会导致两个文件相互链接,最终会引起操作系统的错误。

硬链接和符号链接各有利弊。符号链接的灵活性大,但是效率与稳定性不如硬链接。用户应该根据自己的需要选择合适的静态文件共享方法。

5.4.2 动态共享

文件的动态共享是指系统中不同用户的进程或同一用户的不同进程并发地访问同一个文件,这种共享关系只有当进程存在时才可能出现,一旦进程消亡,其共享关系也就自动消失。文件的动态共享有两种不同的需求。在第一种情况中,两个进程希望能够协同完成同一任务,若使用同一读/写位移,那么,一个进程改变它时,另一个进程能够感觉到它的变化。这样,使父、子进程很容易同步地对文件进行操作。在另外的情况中,两个进程虽然访问同一个文件,但是进程希望能独立地读、写这个文件。这两种情况的区别就是是否通过文件的读/写指针。

在 UNIX/Linux 系统中,文件被打开以后,对应的 inode 就在活动 inode 表中,活动 inode 表是整个系统公用的。为了使用户掌握当前使用文件的情况,系统在每个进程的进程控制块中设立了用户打开文件表,并通过它与各自打开文件的活动 inode 相联系。如果文件的读/写指针被放入活动 inode 中,两个进程共享文件的同时也就共享了读/写指针。如果将文件的读/写指针放入进程的打开文件表中,两个进程共享文件的时候就不能共享读/写指针。放入 inode 表或者用户的打开文件表中都不能同时解决两种需求。为了同时解决用户的两种需求,UNIX/Linux 系统引入了全局的系统打开文件表。

系统打开文件表是介于用户打开文件表和活动 inode 表的中间一层。原先的用户打开文件表→活动 inode 表的关联变成了用户打开文件表→系统打开文件表→活动 inode 表的三层关联。文件的读写指针被放入系统打开文件表的表项中,用户打开文件表中的表项变成了指向系统打开文件表表项的指针。

如果两个进程共享文件时希望共享读/写指针。两进程的用户打开文件表中的表项只要指向同一个系统打开文件表项即可。因为该表项中含有读/写指针,进程同时读写文件的时候就能共享读/写指针。

如果两个进程共享文件时不希望共享读/写指针。两进程的用户打开文件表中的表项需要指向不同的系统打开文件表中的表项,并且这两个系统打开文件表的表项要指向同一个 inode。因为系统打开文件表的表项指向同一个 inode,因为两进程共享了同一个文件;因为两进程没有共享同一个系统打开文件表中的表项,因此两个进程可以有不同的读/写指针。

用户打开文件表、系统打开文件表和活动 inode 表三者的关系如图 5.8 所示。在图中,进程 A、B、C 共享同一个文件,但是 A 与 B 共享读/写指针,A 与 C 不共享读/写指针。

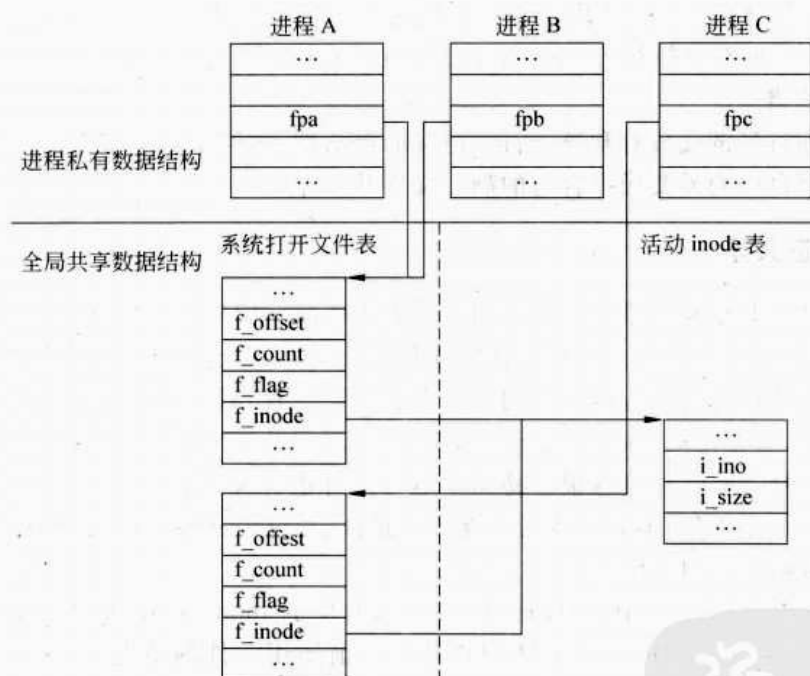


图 5.8 两种动态共享示意图

5.5 Linux 文件系统

5.5.1 虚拟文件系统

Linux 最早采用 Minix 文件系统,20 世纪 90 年代初推出了专门为自己设计的优秀、高效文件系统 Ext (Extended File System),目前常用的是 Ext2、Ext3。为了解决开放

性,Ext 的设计有一个重大的改进,引入了虚拟文件系统(Virtual File System),它为用户提供一个统一、抽象的文件系统界面,以支持多种具体的文件系统。

虚拟文件系统是 Linux 内核的一个子系统,提供了一个通用文件系统模型,该模型概括了所能见到的文件系统常用功能和行为,并为应用程序提供一致性的文件系统接口。VFS 可以分为图 5.9 所示的三层。

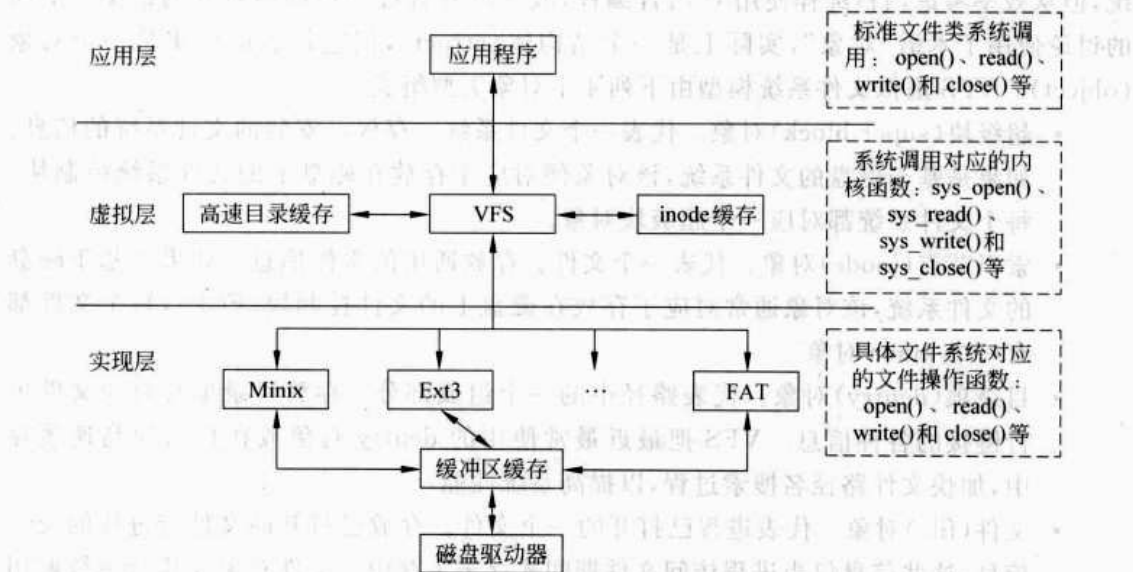


图 5.9 VFS 和具体文件系统的关系

(1) 应用层。VFS 模型源于 UNIX 文件系统,使得用户可以直接使用标准 UNIX 文件系统调用来操作文件,无需考虑具体文件系统特性和物理存储介质,通过 VFS 访问文件系统,才使得不同文件系统之间的协作性和通用性成为可能。

(2) 虚拟层。在对所有具体文件系统的共同特性进行抽象的基础上,形成一个与具体文件系统实现无关的虚拟层,并在此层次上定义与用户的一致性接口。

(3) 实现层。该层使用类似开关表技术进行具体文件系统转接,实现各种具体文件系统的细节,每一个都是自包含的,包含文件系统实现的各种设施,如超级块、节点区、数据区以及各种数据结构和文件类的操作函数。

VFS 抽象层之所以能衔接各种不同的文件系统,是因为它定义了所有文件系统都支持的基本抽象接口和数据结构,同时具体的文件系统也将自己的诸如“文件如何打开”、“目录如何定义”等概念在形式上与 VFS 的定义保持一致。对于像 FAT 和 NTFS 这类非 UNIX 风格文件系统,必须经过封装,提供符合 VFS 概念的接口。比如,一个文件系统不支持 inode 概念,它也必须在主存中装配 inode 结构体,就像它本身包含 inode 一样。

这些装配和转换需要在使用现场引入特别处理,使得非 UNIX 文件系统能够兼容 UNIX 文件系统的使用规则和满足 VFS 的接口需求,这样一来,非 UNIX 文件系统便可与 VFS 一道工作,只是性能上会有少许影响。

Linux 虚拟文件系统 VFS 采用了面向对象设计思想,文件系统中定义的 VFS 相当于面向对象系统中的抽象基类,从它出发可以派生出不同的子类,以支持多种具体文件系统,但从效率考虑内核纯粹使用 C 语言编程,故并没有直接利用面向对象的语义。下面的讨论使用了术语“对象”,实际上是一个结构体(struct),但它代表的确实是一个对象(object)。VFS 虚拟文件系统模型由下列 4 个对象类型组成。

- 超级块(super block)对象。代表一个文件系统。存放已安装的文件系统的信息。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(FCB),每个文件都有一个 inode 对象。
- 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 dentry 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。文件对象在执行系统调用 open()时创建,执行系统调用 close()时撤销。

每个主要对象都包含一个操作对象,它描述了内核针对主要对象可以使用的方法,这些操作对象如下。

- super_operation 对象。其中包括内核针对特定文件系统所能调用的方法。
- inode_operation 对象。其中包括内核针对特定文件所能调用的方法。
- dentry_operation 对象。其中包括内核针对特定目录所能调用的方法。
- file_operation 对象。其中包括进程针对已打开文件所能调用的方法。

操作对象作为一个指针结构体被实现,该结构体包含指向操作其父对象的函数指针,其中的许多方法可继承使用 VFS 的通用函数,若通用函数不能满足需要,就必须使用具体文件系统的独有方法填充这些函数指针,使其指向文件系统实例。

1. 超级块对象

超级块描述一个文件系统的信息。对于每个具体的文件系统来说,都有各自的超级块,如 Ext2 超级块,并被存放在磁盘特定扇区上。当内核对一个具体文件系统初始化注册时,调用文件系统提供的函数为其分配一个 VFS 超级块,并从磁盘读取具体文

件系统超级块中的信息填充进来。也就是说,VFS 超级块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。超级块的数据结构如下。

```
struct super_block {
    struct list_head s_list;           //把所有超级块双向链接起来
    kdev_t s_dev;                      // 文件系统所在设备标识符
    unsigned long s_blocksize;         //以字节为单位的盘块大小
    unsigned char s_blocksize_bits;    /*以 2 的幂次表示的盘块大小,如块为 4KB,值为 12*/
    unsigned long long s_maxbytes;     //文件大小上限
    unsigned char s_dirt;              //修改(脏)标志
    ...
    struct file_system_type * s_type;  // 指向注册表 file_system_type 结构的指针
    struct super_operations * s_op;    // 指向超级块操作函数集的指针
    struct dentry * s_root;            //安装目录的目录项对象
    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
    struct list_head s_dirty;         //脏节点 inode 链表
    struct list_head s_io;            //回写链表
    char s_id[32];                    //文本名字
    ...
    void * s_fs_info;                 //各个具体文件系统私有的数据结构
};
```

每个被安装的文件系统都有一个 super_block 结构,以环形双向链表把它们链接在一起,指向该链表第一个元素和最后一个元素的指针存放在该超级块的成员 s_list 域中。

该结构中的成员 s_fs_info 是实现支持多种文件系统的关键,它指向 Linux 文件系统所支持的各种具体文件系统超级块的特有信息。例如当系统上安装 Ext2 文件系统时,s_fs_info 将指向一个 ext2_sb_info 结构。ext2_sb_info 结构将包含 Ext2 文件系统超级块中特有的信息。

与超级块关联的方法就是超级块操作对象,这些操作由 super_operations 结构来描述。

```
struct super_operations {
    struct inode * (* alloc_inode) (struct super_block * sb); //分配一个 inode
    void (* write_super) (struct super_block *, int);        //把超级块写回磁盘
    void (* put_super) (struct super_block *);               //释放超级块对象
    void (* read_inode) (struct inode *);                     //读取指定 inode
```

```

void (*dirty_inode) (struct inode*);           //将 inode 标记为"脏"
void (*write_inode) (struct inode*,int);       //将给定 inode 写回磁盘
void (*put_inode) (struct inode*);            //逻辑上释放 inode
void (*delete_inode) (struct inode*);         //物理上释放 inode
...
};

```

结构体中的每一项是一个指向超级块操作函数的指针,而超级块操作函数执行文件系统和 inode 的底层操作。

2. 索引节点对象

inode 对象内包含了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 inode 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 inode 直接读入 VFS 的 inode 对象中。如果一个文件系统没有 inode,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 inode。可以把具体文件系统存放在磁盘上的 inode 称为静态节点,它的内容被读入主存 VFS 的 inode 才能工作,后者也称为动态节点,其数据结构的主要域定义如下。

```

struct inode {
    struct hlist_node i_hash;           //散列值相同的 inode 链表
    struct list_head i_list;           //指向 inode 链表的指针
    struct list_head i_dentry;         //同属一个 inode 的 dentry 链表
    ...

    unsigned long i_ino;               //inode 号
    dev_t i_rdev;                     //所在设备的设备号
    umode_t i_mode;                   //文件类型及存取权限
    nlink_t i_nlink;                 //连接到该 inode 的硬连接数
    uid_t i_uid;                     //文件拥有者的用户 ID
    gid_t i_gid;                     //用户所在组的 ID
    loff_t i_size;                   //以字节为单位的文件大小
    ...

    struct rw_semaphore i_alloc_sem;   //inode 信号量
    struct inode_operations* i_op;     //指向 inode 操作函数的指针
    struct super_block* i_sb;         //指向该文件系统超级块的指针
    atomic_t i_count;                 //当前使用该 inode 的引用计数,0 表示空闲
    atomic_t i_writecount;            //写者计数
    struct file_operations* i_fop;     //指向文件操作函数的指针
}

```

```

...
time_t i_atime;           //最近访问时间
time_t i_mtime;          //最近修改时间
time_t i_ctime;          //文件创建时间
union {
    struct pipe_inode_info *i_pipe;    //管道信息
    struct block_device *i_bdev;       //块设备信息
    struct cdev *i_cdev;               //字符设备信息
};
struct address_space *i_mapping;       //文件所有页的集合
unsigned long i_state;                 //inode 状态标志
unsigned int i_flags;                  //文件系统标志
...
union {
    void *generic_ip;                 //各文件系统 inode 的私有数据
} u;
};

```

一个 inode 代表文件系统中的文件，它也可以是设备或管道这类特殊文件，故 inode 中会包含特殊的项。与 inode 关联的方法就是 inode 操作对象，这些操作由 inode_operations 结构来描述。

```

struct inode_operations {
    int (* create)(struct inode*, struct dentry*, int, struct nameidata*);
    struct dentry* (* lookup)(struct inode*, struct dentry*, struct nameidata*);
    int (* link)(struct dentry*, 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*);
    ...
};

```

其中，主要函数功能如下。

- create()。创建一个新的 inode。
- lookup()。查找一个 inode 所在的目录。
- link() 和 unlink()。创建和删除一个硬连接。
- symlink()。为符号链接创建一个 inode。
- mkdir() 和 rmdir()。为目录项创建和删除一个 inode。

3. 目录项对象

VFS 把每个目录看作一个文件,如在路径/tmp/test 中,tmp 和 test 都是文件,tmp 是目录文件,而 test 是普通文件,路径中的每个组成部分都由一个 inode 对象表示。为了方便查找,VFS 引入目录项的概念,每个 dentry 代表路径中的一个部分,如根目录/、tmp 和 test 都是目录项对象。前两个是目录,后一个是普通文件。于是在路径查找中,VFS 为根目录/、tmp 和 test 分别创建了三个目录项对象。所以,每一个文件除了有一个 inode 数据结构外,还有一个 dentry 数据结构与之关联,dentry 结构中的 d_inode 指针指向相应的 inode 结构,引入 dentry 的主要目的是对目录进行缓存,加快对文件的快速定位,改进文件系统效率。dentry 结构代表逻辑意义上的文件,描述文件的逻辑属性,它在磁盘上并没有对应的映像;而 inode 结构代表物理意义上的文件,记录文件的物理属性,它在磁盘上有对应的映像。

dentry 数据结构的主要域定义如下。

```
struct dentry {
    atomic_t d_count;           //目录项引用计数
    unsigned int d_flags;       //目录项状态标志
    struct inode* d_inode;      //dentry 所属的 inode
    struct dentry* d_parent;    //父目录的目录项对象
    struct hlist_node d_hash;   //目录项形成的哈希表
    struct list_head d_lru;     //未用的 LRU 双向链表
    struct list_head d_child;   //父目录的子目录项形成双向链表
    struct list_head d_subdirs; //该目录项的子目录的双向链表
    struct list_head d_alias;   //inode 别名的链表
    int d_mounted;             //是否为安装点目录项
    struct qstr d_name;         //目录项名字,用于快速查找
    unsigned long d_time;       //重新生效时间
    struct dentry_operations* d_op; //操作目录项的函数
    struct super_block* d_sb;    //指向文件的超级块
    struct hlist_node d_hash;   //哈希表
    void* d_fsdata;            //文件系统特殊数据
    unsigned char d_iname[DNAME_INLINE_LEN]; //文件名前 15 个字符
    ...
};
```

除了某些特殊的文件系统,如 RAMFS,一个有效的 dentry 结构必定对应一个 inode 结构,这是因为目录项要么代表一个目录,要么代表一个文件,目录实际上也是文件。所

以,只要 dentry 结构有效,则其指针 d_inode 必定指向一个 inode 结构。反之不然,一个 inode 可能对应多个 dentry 结构,也就是说,一个文件可以有多个文件名或路径名,这是因为一个已经建立的文件可以被链接(link)到其他文件名。所以在 inode 结构中有一个队列 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。

在内核中有一个散列表 dentry_hashtable,是一个 hlist_head 的指针数组,一旦在主存中建立一个目录节点的 dentry 结构,就通过它的 d_hash 域链入散列表中的某个队列中。内核中还有一个队列 dentry_unused,凡是已经没有用户使用的 dentry 结构就通过其 d_lru 域链入空闲队列。dentry 结构中除 d_alias、d_hash 和 d_lru 三个队列外,还有 d_subdirs 队列,当该目录节点有父目录时,则其 dentry 结构就通过 d_child 链入其父节点的 d_subdirs 队列中,同时,又通过 d_parent 指向其父目录的 dentry 结构,而它自己各个子目录的 dentry 结构则挂在其 d_subdirs 域指向的队列中。

可见一个文件系统中所有目录项结构,或组织成一个散列表,或组织成一棵树,或组织成一个链表,这将为文件访问和文件路径搜索奠定良好基础。与目录项关联的方法就是目录项操作对象,这些操作由 dentry_operations 结构来描述。

```
struct dentry_operation {
    int (*d_revalidate)(struct dentry*, struct nameidata*);
    int (*d_hash)(struct dentry*, struct qstr*);
    int (*d_compare)(struct dentry*, struct qstr*, struct qstr*);
    int (*d_delete)(struct dentry*);
    int (*d_release)(struct dentry*);
    int (*d_iput)(struct dentry*, struct inode*);
    ...
};
```

其中,主要函数功能如下:d_revalidate()判定目录项是否有效;d_hash()生成一个散列值;d_compare()比较两个文件名;d_delete()删除 d_count 为 0 的目录项对象;d_release()释放一个目录项对象;d_iput()丢弃目录项对应的 inode。

4. 与进程相关的文件结构

(1) 系统打开文件表——file 结构。

每个文件都用一个 64 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。允许通过系统调用 lseek 对文件位置作随机定位。Linux 建立了文件对象(file)来保存打开文件的文件位置,file 结构除保存文件当前位置外,还把指向该文件 inode 的指针也放在其中,

并形成一个双向链表,称系统打开文件表。每当打开文件时,就要创建一个 file 结构。该结构的域定义如下。

```
struct file{
    struct dentry* f_dentry;           //指向相关目录项的指针
    struct vfsmount* f_vfsmnt;        //指向 VFS 安装点的指针
    struct file_operations* f_op;      //指向文件操作函数的指针
    struct file_ra_state f_ra;         //文件预读取状态
    mode_t f_mode;                     //文件访问模式
    loff_t f_pos;                       //文件当前偏移量
    unsigned int f_flags;               //打开文件时指定的标识
    atomic_t f_count;                  //使用该文件的进程数
    unsigned int f_uid;                //使用者的用户标识
    unsigned int f_gid;                //使用者的用户组标识
    void* private_data;                //私有信息,可供驱动程序使用
    ...
};
```

文件对象的创建和释放是通过 Slab 分配器实现的,文件对象所用的 Slab 分配器的缓存名称为 filp,缓存描述符存储在 filp_cache 变量中。系统中对于文件对象的数量是有限制的,系统中能够同时打开的文件对象数量定义在宏 NR_FILE 中。

与文件关联的方法就是文件操作对象,这些操作由 file_operations 结构来描述。

```
struct file_operations {
    loff_t (*llseek) (struct file*, loff_t, int);
    ssize_t (*read) (struct file*, char __user*, size_t, loff_t*);
    ssize_t (*aio_read) (struct kiocb*, char __user*, size_t, loff_t);
    ssize_t (*write) (struct file*, const char __user*, size_t, loff_t*);
    ssize_t (*aio_write) (struct kiocb*, const char __user*, size_t, loff_t);
    int (*mmap) (struct file*, struct vm_area_struct*);
    int (*open) (struct inode*, struct file*);
    int (*ioctl) (struct inode*, struct file*, unsigned int, unsigned long);
    int (*flush) (struct file*, fl_owner_t id);
    int (*release) (struct inode*, struct file*);
    int (*fsync) (struct file*, struct dentry*, int datasync);
    ...
};
```

其中,主要函数功能如下:llseek()修改文件指针;read()从文件的偏移处读出若干字节;write()向文件指定偏移处写入若干字节;aio_read()以异步方式从文件的偏移处读出若

干字节; `aio_write()` 以异步方式向文件指定偏移处写入若干字节; `mmap()` 文件到主存的映射; `open()` 打开一个文件; `ioctl()` 向硬设备发送命令; `flush()` 关闭文件时刷新文件; `release()` 释放文件对象; `fsync()` 将文件在缓冲的数据写回磁盘。

(2) 用户打开文件表——`files_struct` 结构。

文件描述符 `fd` 用来描述打开的文件, 每个进程用一个 `files_struct` 结构来记录文件描述符的使用情况, 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。此结构定义如下:

```
struct files_struct{
    atomic_t count;                //共享该表的进程数
    struct fdtable * fdt;          //文件描述符表指针, 指向 fdtab
    struct fdtable fdtab;          //文件描述符表
    spinlock_t file_lock __cacheline_aligned_in_smp; //保护该结构体的锁
    int next_fd;                   //下一个空闲 fd
    struct embedded_fd_set close_on_exec_init; //执行 exec() 函数时需关闭的文件描述符初值集
    struct embedded_fd_set open_fds_init;      //文件描述符初值集
    struct file * fd_array[NR_OPEN_DEFAULT];  //指向文件对象的初始化指针数组
};
```

该结构包含了 `fdtable` 结构, 其中包含了文件描述符数组的信息。

```
struct fdtable {
    unsigned int max_fds;          //进程当前具有的最大文件数
    int max_fdset;                 //当前文件描述符的最大数
    struct file ** fd;             //指向文件对象(系统打开文件表项)的指针数组
    fd_set * close_on_exec;        //指向执行 exec() 函数时需关闭的文件描述符
    fd_set * open_fds;             //指向打开文件的描述符的指针
    ...
};
```

`fd` 是指向文件对象的指针数组的指针, 数组的长度存放在 `max_fdset` 域中。通常, `fd` 域指向 `files_struct` 结构的 `fd_array` 域, 该域包含 `NR_OPEN_DEFAULT` (默认值为 32) 个文件对象指针。如果进程打开的文件数目大于 `NR_OPEN_DEFAULT`, 内核分配一个新的文件指针数组, 并将其地址存放在 `fd` 域中, 同时更新 `max_fds` 域的值。

对于在 `fd` 数组中有入口地址的每个文件来说, 数组的下标就是文件描述符。通常, 数组的第一个元素(索引为 0)、第二个元素(索引为 1)和第三个元素(索引为 2)分别表示标准输入文件、标准输出文件和标准错误文件, 且这三个文件通常从父进程处继承而来。通过适当的系统调用, 两个文件描述符可以指向同一个打开的文件, 亦即数组的两个元素

可以指向同一个文件对象。

注意区分系统打开文件表和用户打开文件表,前者是由 file 对象组成的链表,它的内核态,由内核控制。而后者就是文件描述符表,在用户进程空间,是进程的私有数据。因而,应用程序只能使用文件描述符作为参数的系统调用才能访问系统打开文件表。

(3) 根目录和当前工作目录——fs_struct 结构。

当程序通过文件名访问一个文件时,内核首先通过遍历文件系统树找到相对应的目录,然后在该目录中找到该文件名和存放它的 inode 号。有了 inode 号,也就确定了文件在磁盘上的物理位置。为了进行文件查找,每个进程都有一个当前工作目录和当前工作目录所在文件系统的根目录。这是进程状态的一部分,以使用户既可以使用相对路径名也可以使用绝对路径名来访问所要的文件。

对一个文件进行访问时,搜索的起点要视路径名是相对路径名还是绝对路径名而定。如果是相对路径名,则从当前目录开始搜索;如果是绝对路径名,则从根目录开始搜索。因为相同的文件名允许出现在不同的目录中,说明只有文件名还不能唯一地确定一个文件。因此绝对路径名才可以唯一地指定一个文件,绝对路径名要求给出在目录树中从根目录访问文件的路径上的所有节点。

许多情况下可以不给出全路径名。在系统正常运行后的任意时刻,每个用户都有一个正在使用的当前工作目录。如果路径名不以“/”开头,则路径被认为是以当前工作目录开头,这就是相对路径名。用户登录时与 shell 相关联的当前工作目录成为用户的主目录。fs_struct 结构的定义为

```
struct fs_struct {
    atomic_t count;           //共享 fs_struct 结构的进程数
    rwlock_t lock;           //保护此结构体的锁
    int umask;                 //默认的文件访问权限掩码
    struct dentry* root;      //根目录的目录项对象
    struct dentry* pwd;       //当前工作目录的目录项对象
    struct dentry* altroot;   //可替换的根目录的目录项对象
    struct vfsmount* rootmnt; //根目录的安装点对象
    struct vfsmount* pwdmnt;  //当前工作目录的安装点对象
    struct vfsmount* altrootmnt; //可替换的根目录的安装点对象
};
```

fs_struct 中的 dentry 结构是对一个目录项的描述,root、pwd 和 altroot 三个指针都指向这个结构,在程序实际运行时,这三个目录不一定都在同一个文件系统中。例如,进程的根目录通常安装于“/”节点上的 Ext2 文件系统,而当前工作目录可能是安装于 /msdos 目录下的 DOS 文件系统。因而,fs_struct 结构中的 rootmnt、pwdmnt 及 altroot 就是对上述三个目录的安装点的描述,安装点的数据结构类型为 vfsmount。

5. Linux 文件系统的逻辑结构

Linux 中的每个进程都有两个数据结构来描述进程与文件的相关信息,一个进程所处的位置是由 `fs_struct` 结构来描述,它包含了两个指向 VFS dentry 的指针,分别指向根目录节点(`root`)和当前目录节点(`pwd`);而进程打开的文件是由 `files_struct` 结构来描述,最多能同时打开的文件为 `OPEN_MAX`(默认值为 256)个,由 `fd[0]~fd[255]` 所表示的指针指向对应的 `file` 结构。每当打开一个文件时,就从 `files_struct` 结构中找一个空闲的文件描述符,使它指向打开文件的描述结构 `file`,对文件的操作通过 `file` 结构中定义的文件操作函数和 VFS inode 的信息来完成。整个系统打开的文件则由 `file` 结构来描述。图 5.10 给出了这些数据结构之间的关系。

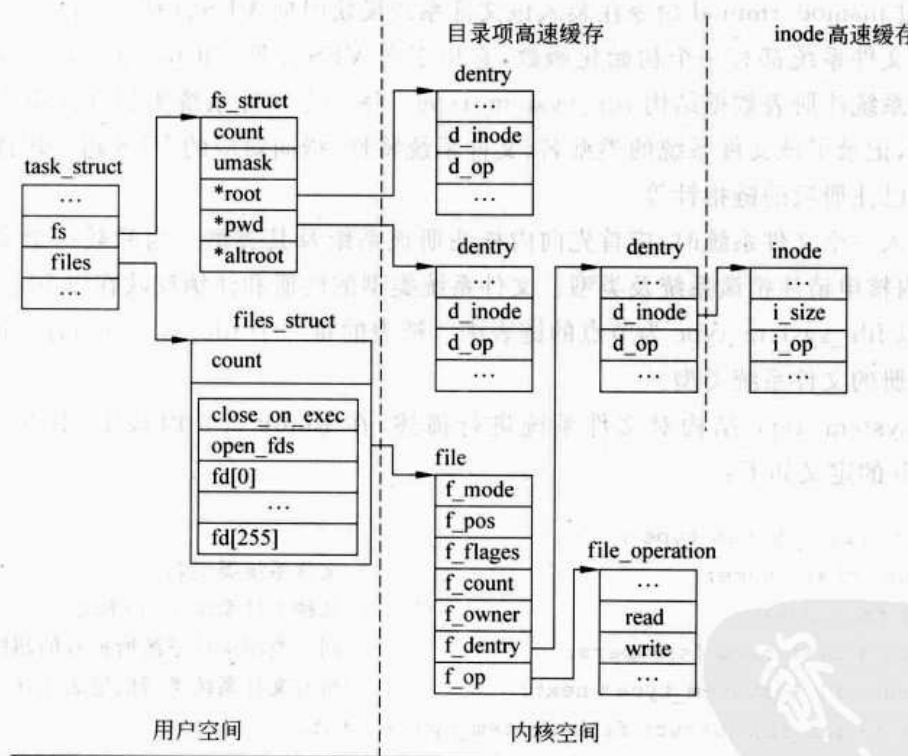


图 5.10 Linux 文件系统逻辑结构

有了 `fs_struct`、`file` 和 `files_struct` 结构,可以通过两种途径共享文件,一是多个进程共享同一个 `file` 结构,`file` 唯一对应一个文件;二是多个进程的 `file` 结构共享一个 `inode`,`inode` 唯一对应一个文件。前者在父子进程间发生,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,都有项指向同一个 `file` 结

构。后者是通过文件的链接机制实现,两个独立进程打开同一个文件也属于这种共享,系统中专门设计了文件锁,解决两个进程同时读写文件时的互斥问题。

5.5.2 文件系统注册与注销,安装与卸载

1. 文件系统的注册与注销

同其他操作系统一样,Linux 支持多个物理硬盘,每个物理磁盘可以划分为一个或多个磁盘分区,在每个磁盘分区上都可以建立文件系统。VFS 以链表形式管理已注册的文件系统。向系统注册文件系统类型有两种途径,一是在编译操作系统内核时确定了可以支持哪些文件系统,在文件系统被引导时,在 VFS 中进行注册;二是文件系统当作可装载模块,通过 `insmod/rmmod` 命令在装入该文件系统模块时向 VFS 注册/注销。

每个文件系统都有一个初始化函数,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`。每一个文件系统类型在注册表中有一个登记项,记录了该文件系统的类型名、文件系统特性、指向对应的 VFS 超级块读取函数的地址及已注册项的链指针等。

当装入一个文件系统时,应首先向内核注册该系统及其类型。当卸载一个文件系统时,应向内核申请注销该系统及类型。文件系统类型的注册和注销反映在以 `file_systems` 为链头,以 `file_system_type` 为节点的链表中。链表的每一个 `file_system_type` 节点描述一个已注册的文件系统类型。

`file_system_type` 结构对文件系统进行了描述,在 Linux 2.6 内核中,其在 `include/linux/fs.h` 的定义如下:

```
struct file_system_type {
    const char * name;           //文件系统类型名
    int fs_flags;                //几种文件系统 flag 标志
    struct list_head fs_supers;  //同一类型文件系统所拥有的超级块链表
    struct file_system_type * next; //所有文件系统类型的链表指针
    int (*get_sb) (struct file_system_type *, int,
                  const char *, void *, struct vfsmount *);
    void (*kill_sb) (struct super_block *);
    ...
};
```

`get_sb` 和 `kill_sb` 分别是在挂载(`mount`)和卸载(`unmount`)文件系统实例时会被调用的方法。VFS 通过 `get_sb()` 函数读取超级块,通过 `kill_sb()` 函数删除超级块。函数 `register_filesystem()` 用于注册文件系统类型,函数 `unregister_filesystem()` 用于注销一个

文件系统类型。

2. 文件系统的安装与卸载

Linux 不通过设备标识来访问某个文件系统,而是通过命令把它安装到文件系统树形目录结构的某个目录节点,安装后该文件系统的所有文件和子目录就是该目录节点的文件和子目录,直到用命令显式地卸载该文件系统。安装 Linux 时,硬盘的一个分区已经安装了 Ext2,它是作为根文件系统在启动时自动安装的。如果超级用户欲安装另外的文件系统,则需要指定文件系统类型名、文件系统所在的物理设备名及在已有文件系统安装点等信息,再用 mount 命令进行安装。

安装一个文件系统时,内核首先检查参数的合法性,VFS 通过查找由 file_systems 指向的注册表,寻找匹配的 file_system_type。当找到匹配项时,就可获得读取文件系统超级块函数的地址;接着查找作为新文件系统安装点的 VFS inode;VFS 安装程序必须分配一个 VFS 超级块,然后读入安装文件系统的超级块,并进行填充;再申请一个 vfsmount 数据结构(其中包含了文件系统所在的块设备的标识、安装点及指向 VFS 超级块的指针等),使它的指针指向所分配的 VFS 超级块。当文件系统安装以后,它的根 inode 便常驻在 inode 高速缓存中。

使用 umount 卸装某个文件系统时,必须首先检查文件系统是否是可卸载的,当它正在被使用时,该文件系统不能卸载;如果文件系统中的文件或目录正在使用,则 VFS inode 缓存中可能包含对应的 VFS inode;内核根据文件系统所在设备的标识,检查在缓存中是否有来自该文件系统的 inode,如果有且使用计数大于 0,则该文件系统不能卸载;否则查看对应的 VFS 超级块的标志,如果为“脏”,必须把 VFS 超级块写回磁盘。上述过程结束后,释放对应的 VFS 超级块,从而卸载该文件系统。

5.5.3 文件系统的缓存机制

Linux 支持多种类型的文件系统,又能保持很高的性能,其中多种高速缓存起到了关键作用。

1. VFS inode 缓存

VFS 提供缓存,把当前使用的 inode 保存起来,同时为了从中快速找到所需 inode,还采用散列技术。其基本思想是:根据文件系统所在的超级块和 inode 号计算出每个已分配的 inode 的散列值,凡具有相同散列值的 inode 被链入同一个队列中,系统建立一张散列表,其中每项包含一个指向 VFS inode 散列队列的头指针。

当 VFS 按需要计算出一个散列值时,就把它作为访问散列表的索引,从散列表得到指向相应的 inode 队列的指针。如果所指队列中包含想要查找的 inode,只需将该 inode

访问计数加 1,表明又有一个进程使用该 inode。否则,必须找出一个空闲的 VFS 的 inode,且从具体文件系统中读取该 inode,把新的 inode 加入对应的散列队列中。

2. VFS 目录高速缓存

通过路径名查找该目录文件的 inode 是使用频率极高的操作,为提高此类操作的效率,系统维护了表达路径与 inode 对应关系的 VFS 目录缓存,其中存放被访问过的目录,这样当同一目录被再次访问时就可以快速获得。目录缓存也采用散列表方法管理,表的每项是一个指针,指向具有相同散列值的目录缓存队列,而散列值利用目录的父节点目录名计算出来。由于高速缓存容量有限,在使用过程中,VFS 便采用 LRU 算法替换缓存中的目录项。其方法是:VFS 维护两级 LRU 链表,当第一次查找一个目录项时,该目录项就被放入目录缓存中,同时进入第二级 LRU 链表的末尾;如果此时缓存已满,将替换 LRU 链表中的队首目录项,并把它放进目录缓存中。以后该目录项再次被存取时,它将被提升并链入到第一级 LRU 链表的末尾。类似地,如果此时缓存已满,则替换该 LRU 链表的队首目录项。这样越是最近使用的目录项越靠近链尾,只有最近不常用的目录项才逐步移到链表头部,最终被替换掉。

3. 页高速缓冲区

Linux 维护一组页缓冲区,它独立于任何类型的文件系统,被所有的物理设备所共享。数据一经使用,就会在页缓冲区中留下备份,并且尽量让它停留较长时间,下次再访问该数据时,不必从物理设备读出,减少了内外交换的次数。

页缓冲区由主存物理页框组成,缓冲区中的一页对应磁盘上的多个块。例如,块大小为 1KB,则一页对应 4 块,所以,页缓冲区实际缓存的是以页为单位的文件块。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中,以决定是否访问磁盘。Linux 页面缓冲由 address_space 结构体描述。

```
struct address_space {
    struct inode * host;                //属主的 inode
    struct radix_tree_root page_tree;   //页面的集合
    struct address_space_operations * a_ops; //操作函数表
    struct prio_tree_root i_mmap;       //私有或共享映射表
    spinlock_t i_mmap_sem;              //保护映射表的锁
    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 inode),这时,host 域指向该 inode,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
    releasepage();         //日志文件系统准备释放页
    direct_io();           //数据页的直接 I/O 传输
    ...
};
```

5.5.4 Ext2 文件系统

Ext2(second extended file system)文件系统支持标准 UNIX 文件类型,包括普通文件、目录文件、特殊文件和符号链接文件。Ext2 文件系统可以管理特大磁盘分区,文件系统最大可达 4TB。此外,它还支持长文件名、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,但不同 Ext2 文件中块数据的块大小可以不同,其物理结构如图 5.11 所示。Ext2 所占用的磁盘除引导块 (boot block) 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、inode 位示图、inode 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编号,由块设备驱动程序将逻辑块号转换成块设备的

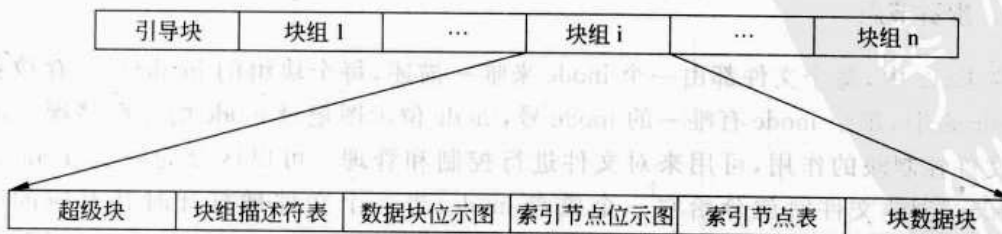


图 5.11 Ext2 文件系统结构

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。

采用块组划分的目的,一是提高文件系统可靠性,因为每个块组中都有管理信息的副本,当文件系统崩溃时就可很容易地恢复它;二是提高文件系统的性能,由于块组内数据块靠近其 inode,文件 inode 靠近其目录 inode,从而将磁头定位时间减到最少,加快磁盘访问速度。

1. 超级块

Ext2 的超级块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,一般来说只有块 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(一般为 1~4KB)、空闲块数量、inode 数量、空闲 inode 数量、第一个 inode 号、第一个数据块位置、每个块组中的块数、每个块组的 inode 数,以及安装时间、最后一次写时间、安装信息和文件系统状态信息等内容。

2. 块组描述符

每个块组都有一个块组描述符,记录了该块组的以下信息。

- 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。
- inode 位示图。表示 inode 位示图占用的块号,此位示图反映块组中 inode 的分配情况,在创建或删除文件时需使用 inode 位示图。
- inode 表。块组中 inode 占用的数据块数,系统中的每个文件对应一个 inode,每个 inode 都由一个数据结构来描述。
- 空闲块数、空闲 inode 数和已用数目。

一个文件系统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。

3. 索引节点

在 Ext2 中,每个文件都由一个 inode 来唯一描述,每个块组的 inode 集中存放在一个 inode 表中,每个 inode 有唯一的 inode 号,inode 位示图记录 inode 的分配情况。inode 起着文件控制块的作用,可用来对文件进行控制和管理。可以区分为磁盘 inode 和主存 inode。创建文件时就分给它一个磁盘 inode,当一个文件被打开时其对应的磁盘 inode 复制到主存 inode 中,当文件关闭时其主存 inode 的内容写回磁盘 inode,并释放该主存 inode。

4. 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,访问的速度大幅下降。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。

- 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那个数据块,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,而且首先考虑 8 个一簇的连续的块。
- 预分配策略。如果 Ext2 引入了预分配机制(设 EXT2PREALLOCATE 参数),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的话,也被保留下来。当文件关闭时仍保留的数据块给予释放,这样保证了尽可能多的数据块被集中成一簇。

5.5.5 proc 文件系统

早期的 UNIX 系统在设备文件目录/dev 下设置了一个特殊的设备文件,称为/dev/mem。通过/dev/mem 可以读写系统的物理内存,提供了一个在内核外部动态读写内核数据结构的机会。这个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,还可以动态地修改内核的数据结构,改变内核的运行状态。在 UNIX 的发展过程中,人们进一步加强了该功能。在系统根目录下建立了 proc 目录,每当一个进程创建时,该目录下便会创建一个以该进程 PID 命名的子目录。通过这些目录,用户可以修改进程空间中的各种信息。

经过多年的发展,/proc 形成了一个特殊的文件系统。该文件系统的类型即为 proc。内核通过 proc 文件系统将一些信息输出给用户。/proc 目录下每个文件的读写都可以绑定到一个内核函数上。当文件被读写时,被绑定的相应内核函数就会被触发,由这些函数处理文件的输出与输入。经过多年的发展,proc 文件系统中所涵盖的内容已几乎涉及系统的所有方面。用户通过 proc 文件系统可以了解进程的地址空间信息、系统的硬件信息、系统的中断信息和系统的 I/O 信息等。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。但是也有一些 proc 文件被设置成可写,用户就可以通过写这些 proc 文件将参数传递给内核。

在 proc 文件系统中,代表各个文件节点的是 proc_dir_entry 结构。该结构和文件系统中的 dir_entry 类似,它管理着从操作系统的用户空间到内核空间对文件读写的驱动。

proc_dir_entry 是 proc 文件系统中最重要的数据结构。在系统初始化时,主要的工作就是建立 proc 文件系统树。proc 文件系统树保存了完成读写 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,只存在内存里,因此在读操作时,不像其他文件系统那样从外存中读取 inode 信息,而是从 proc 树中读取 inode 信息,然后再调用 inode 中登记的函数,动态地从内核读取所需要的信息。在外部看来,proc 文件系统就和其他文件系统一样,觉察不出区别。

系统启动之后,创建了由 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。除了提供读的功能,proc 文件系统的部分文件还提供写的功能。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。

1. proc 文件系统初始化

遵循 VFS 的规范,proc 文件系统在使用之前必须进行注册。Linux 内核在启动过程中,会调用 proc_root_init 进行 proc 文件系统的初始化工作。函数 proc_root_init 完成 proc 文件系统的注册,并初始化 /proc 下的目录。

文件系统的注册函数为 register_filesystem(),该函数的参数类型为 file_system_type。file_system_type 对文件系统类型进行描述。在 linux/fs/proc/root.c 文件中,proc_fs_type 对 proc 文件系统进行了描述。

```
static struct file_system_type proc_fs_type={
    .name="proc",
    .get_sb=proc_get_sb,
    .kill_sb=kill_anon_super,
};
```

在该变量中,name 指明了文件系统的名称为 proc;用于读取文件系统超级块的函数为 proc_get_sb。

在 proc_root_init 函数中,内核做的主要工作如下。

- (1) 调用 register_filesystem 函数将 proc 文件类型加入文件类型的单向链表中。
- (2) 调用 kern_mount 函数初始化超级块,并创建 proc 文件系统的 vfsmount 结构,将其加入已装载文件系统的链表中。
- (3) 创建 /proc 目录下的各种子目录,建立 proc 文件系统树。
- (4) 如果在上述过程中出现错误,内核会将文件系统卸载。

2. 初始化超级块

一般文件系统的超级块信息是存储在辅存中。当文件系统被挂载时,内核通过文件

系统提供的读取超级块函数从辅存中获得超级块信息。proc 文件系统是一个虚拟的文件系统,它并不需要从辅存中获得超级块信息。proc 文件系统使用固定的数据将超级块初始化。

在 proc 文件系统的定义中,读取超级块信息的函数为 `proc_get_sb`。在 `proc_root_init` 函数中,`kern_mount` 函数会进行文件系统的挂载。该函数调用 `proc_get_sb` 进行超级块的初始化。函数 `proc_get_sb` 通过 `proc_fill_super` 将固定信息填充超级块的数据结构。该函数进行了如下 4 个步骤。

(1) 首先将文件系统的基本信息写入超级块数据结构。函数将 `s_blocksize` 设置为 1024,`s_blocksize_bit` 设置为 10,并设置 proc 文件系统的魔数。函数将超级块的函数集设置为 `proc_sops`,设置 proc 文件系统中的文件最大字节数为 `MAX_NON_LFS`。

(2) 使用 `proc_get_inode()` 函数建立根结点 `root_inode`。

(3) 修正 `root_inode` 的链接数。函数将遍历进程链表,对于其中每一个进程,都使 `i_nlink++`,这是因为在 `/proc` 目录中,每一个进程都有一个目录。换句话说,存在一个进程,就必然在 `proc` 目录中对应一个子目录,因此在建立 `proc` 的根节点时,要根据进程数修改它的 `i_nlink`。

(4) 根据刚刚建立的 `root_inode`,为超级块的 `s_root` 建立 `root dentry`。`proc_fill_super` 返回时已经向超级块写入了必要的数据,为 `proc` 文件系统建立了根节点,并在目录高速缓存中建立相应的 `dentry` 结构。

3. 对 proc 文件的管理

在 `proc` 的文件系统中,所有属于 `proc` 文件系统的文件都有一个 `proc_dir_entry` 与之对应。当 VFS 需要操作 `proc` 文件时,`proc` 文件系统需要将该文件的 `proc_dir_entry` 与 `inode` 连接。

```
struct proc_dir_entry {
    unsigned short low_ino;           //inode 节点中成员 i_ino 的低 16 位
    unsigned short namelen;          //名字的长度
    const char * name;                //名字
    mode_t mode;                      //属性
    nlink_t nlink;                    //目录下子目录的数目
    uid_t uid;                        //用户号
    gid_t gid;                        //组号
    unsigned long size;               //大小
    struct inode_operation * proc_iops; //节点操作函数
    struct file_operations * proc_fops; //文件操作函数
    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;           //计数器;
int deleted;              //删除标志,表示一个结构实例可以被再使用
kdev_t rdev;              //主设备号与次设备号
};

```

(1) 建立 proc 文件。

内核通过函数 `create_proc_entry()` 创建并注册 proc 文件。该函数做的主要工作如下：

- ① 首先做一些必要的检查,例如要确保它的父节点必须存在等。
- ② 创建一个 `proc_dir_entry` 结构,并且为该文件的名称也分配空间。
- ③ 根据该文件的类型,设置适当的模式和链接数。
- ④ 调用 `proc_register(parent, ent)` 函数,将这个结构链接到 proc 文件树中。

`proc_register` 函数主要完成以下三部分的工作：

- ① 使用 `make_inode_number()` 函数动态创建一个节点号,并且设置 `low_ino`。
- ② 将这个 `proc_dir_entry` 结构链接到它的父节点上。
- ③ 根据文件类型的不同,设置不同的(索引节点和文件)默认操作函数集。

经过以上步骤,一个 proc 文件就注册成功了。用户可以开始通过在 `/proc` 下创建的文件与内核交互。

(2) 删除 proc 文件。

内核通过函数 `remove_proc_entry()` 删除 proc 文件。该函数做的主要工作如下。

- ① 在参数 `parent` 的所有子节点中查找指定的名字,如果找到匹配的节点,就将该结构从树结构中去掉。
- ② 如果删除的 `proc_dir_entry` 是目录结构,减少其父节点的链接数。
- ③ 调用 `clear_bit()` 函数,清除该节点号。
- ④ 将该结构的链接数置 0,并调用 `atomic_read(&de->count)` 来检查它的引用计数。如果是 0,使用函数 `free_proc_entry` 释放该节点;否则就将它的删除标记位置 1,在以后适当的机会中,再将其释放。

(3) 其他管理函数。

内核还提供了其他的一些函数,方便管理和使用 proc 文件系统。读者通过阅读代码对这些函数进行分析。主要的函数如下。

- `struct proc_dir_entry * proc_mkdir(const char * name, struct proc_dir_entry * parent)`。用来在 `proc` 文件系统中注册一个子目录。
- `struct proc_dir_entry * proc_mknod(const char * name, mode_t mode, struct proc_dir_entry * parent, kdev_t rdev)`。用来在 `proc` 文件系统中建立一个设备文件。
- `struct proc_dir_entry * proc_symlink(const char * name, struct proc_dir_entry * parent, const char * dest)`。用于在 `proc` 文件系统中创建一个链接文件。

5.6 本章小结

文件系统是操作系统中负责存取和管理信息的模块,它用统一的方式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。文件系统把磁盘的硬件特性和用户隔离开来,为用户提供“按名存取”的功能,提供了一种简单方便、高度抽象的磁盘使用方法。

文件是由文件名字标识的一组信息的集合。一个文件必须从逻辑结构和物理结构两个侧面来观察它,文件从逻辑结构方面分为流式文件和记录式文件;从物理结构方面分为顺序文件、连接文件、索引文件和直接文件。

文件目录是实现按名存取的主要数据结构,文件系统的基本功能之一就是负责文件目录的建立、维护和检索,要求编排的目录便于查找、防止冲突,目录的检索方便迅速。实际的操作系统常使用树型目录结构,目录可以按不同方法组织,有的把文件名、文件属性和磁盘地址放在一起存放组成文件控制块;有的仅存放文件名字和 `inode` 号,其他信息放到文件 `inode` 中。

文件共享是指不同用户(进程)共同使用同一个文件,文件共享有时不仅为不同用户完成共同的任务所必须,而且还可以节省大量的辅存空间,减少由于文件复制而增加的访问辅存次数。文件共享分静态共享和动态共享。

为了兼容多种文件系统,Linux 设计了虚拟文件系统。它是使一个操作系统中同时支持多种具体文件系统而采用的一种通用文件系统模型。

思 考 题

1. 为了快速访问,又易于更新,当数据为以下形式时,你选用何种文件组织方式。
(1)不经常更新,经常随机访问;(2)经常更新,经常按一定顺序访问;(3)经常更新,经常随机访问。
2. 一些系统允许用户同时访问文件的一个备份来实现共享,另一些系统为每个用户提供一共享文件备份,试讨论各自的优缺点。

3. 在 Linux 文件系统中,为什么要把文件属性数据从目录项中分离出来?

进一步阅读

1. sysfs 是一个特殊的文件系统,类似于 proc 文件系统。sysfs 不仅和 proc 一样允许用户空间访问内核的数据,而且它以更结构化的方式向用户提供内核数据信息。

参考资料:

sysfs 实现:倪继利.《Linux 内核分析及编程》7.5 节:电子工业出版社,2005.9

2. 在 Linux2.6 版本的内核中,有两种用于管理设备的文件系统:devfs 和 udev。这两种文件系统是不存在于磁盘中的文件系统,用于对设备进行管理。采用设备文件系统之前,系统需要在/dev 中预先创建许多无用的设备文件;采用 devfs 或 udev 之后,当新硬件添加时,设备文件系统将在/dev 下自动添加新的节点,方便设备的管理。devfs 工作在核心态,是内核的一部分;而 udev 完全在用户态工作,利用设备加入或移除时内核所发送的 hotplug 事件来工作。

参考资料:

devfs 实现:毛德操,胡希明.《Linux 内核源代码情景分析》8.11 节:浙江大学出版社 2001.9

udev 介绍: <http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html>

devfs 与 udev 比较: http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev_vs_devfs

第6章 设备管理

计算机系统的外部设备不仅种类繁多,各自的特性和操作方式也有很大差别,而且外部设备及其连接方式还在不断地发展中。因此,管理和控制所有外部设备是操作系统的重要功能之一。

操作系统设备管理的目标就是要为设备的使用提供简单方便的统一接口,支持连接的可扩充特性,并且优化 I/O 操作,实现最大的并发性。为了实现这一目标,操作系统需要在计算机进行输入输出操作时,管理和控制 I/O 设备,并提供适当的 I/O 操作支持,如驱动程序接口等。操作系统的这部分功能代码称为设备管理子系统。

在操作系统的设计和实现中,设备管理子系统是相当庞大复杂的功能模块。

6.1 设备管理概述

6.1.1 设备的分类

为了便于设备的管理和使用,简化设备管理程序,通常从不同的角度出发,将设备进行不同的分类。这是因为,设备管理程序是和硬件打交道的,不同的设备硬件需要对应于不同的管理程序。但是,对于同类设备来说,由于设备的硬件特性十分相似,从而可以利用相同的管理程序或只需做很少的修改即可。最常见的有以下两种分类方法。

1. 按设备的信息组织方式分类

为了方便内核对设备的管理,通常根据设备的控制复杂性及每次的数据传输单位等特性,将设备分为三大类:字符设备、块设备和网络设备。Linux 系统采用这种分类方法。

这种方法的优点是可将控制不同 I/O 设备的驱动程序与操作系统的其他部分隔离开来,为系统中增加和撤销设备提供便利。例如在管理块设备时,文件系统仅仅控制抽象的块设备,而将与设备细节相关的部分留给低层的设备驱动程序去处理。

(1) 块设备。

块设备是以块为单位进行数据传输的设备,块设备中的数据也是以块为单位进行组织和管理的。块的大小通常是 2^n 个字节。通常,块设备是可任意访问的,即可以按块号对块设备进行随机访问。为了匹配 CPU 与块设备之间传输速度的差异,块设备在进行

I/O 时通常都需要利用缓冲技术,本章后面将会对其进行详细介绍。

常用的外部存储设备基本上都是块设备,如硬盘、软盘及 CD-ROM 等。此外,也可将 RAM 模拟为 RAM 盘,作为块设备来使用。

(2) 字符设备。

字符设备是以字符为单位(早期一个字符可能为 5、7、8 位,目前通常都是 8 位,即一个字节)进行数据传输的设备,其速度较慢,主要用于与用户打交道。通常,字符设备的数据传输只能顺序进行,而不能进行随机定位。另外,在 I/O 过程中,一般不采用缓冲技术。

常见的字符终端、打印机、扫描仪、传感器和鼠标等都属于字符设备。

(3) 网络设备。

网络设备是一类有别于字符设备和块设备的设备。从传输速度上看,它通常快于字符设备而慢于块设备。从传输单位上看,它以字节为单位。但由于网络的分层组织结构,因此从不同的层次上看,它既具有字符设备的特点,又具有块设备的特点。Linux 中将这类设备单独分类管理,不像字符设备和块设备那样通过设备文件访问。

但是,并非所有的设备都适用这种分类方法。例如,时钟和显示器,既不符合块设备,也不符合字符设备的特性。

2. 按设备的共享性分类

按设备的共享性,从资源分配的角度可以分为独占设备、共享设备和虚拟设备。

(1) 独占设备。

独占设备是指在同一时刻只能被一个作业或进程使用的设备。这类设备一旦分配给某个作业后,就完全由该作业独占使用,直到作业主动释放后才能分配给新的作业使用。多数低速设备,如打印机、绘图仪等,都属于独占设备。如果占用设备的作业不能充分、连续地利用设备,则独占设备的利用率会较低甚至很低。

(2) 共享设备。

共享设备是指允许若干个作业分时(并发)共享使用的设备,如磁盘等大多数直接存取设备等都属于共享设备。对于这些设备,几个进程或作业可分时(交替)地从其中读写信息。显然,当一个作业不能充分、连续地利用设备时,共享可提高设备的利用率。

为了实现设备的共享,Linux 内核通过分层抽象,将设备的管理细节屏蔽起来。

(3) 虚拟设备。

为了提高设备的利用率,操作系统利用虚拟(SPOOLing)技术(在 6.2.5 节有进一步介绍),把物理上独占的设备转变成逻辑上共享的设备,即虚拟设备。在采用虚拟设备的情况下,作业都是通过虚拟设备使用物理上的独占设备。作业在使用虚拟设备时,表面上是并发的,但实际的物理设备还是由虚拟设备以独占方式使用。

虚拟设备出现后,用户所用的(虚拟)设备与物理设备之间不再有严格的一一对应关系。为了避免引起混乱,操作系统在为用户的逻辑设备分配相应的物理设备时,应建立二者的对应关系映射表,以便管理。

6.1.2 设备控制器

I/O 设备通常包含一个机械部件和一个电子部件。为了达到设计的模块性和通用性,一般将两者分开。机械部件就是设备本身。电子部件(通常是一块可以插入主板扩展槽的印刷电路板)称为设备控制器或适配器。它可以管理端口、总线或设备,实现设备(机械部分)与主机之间的数据传输。现在的设备控制器已经可以控制多台同类型的设备。例如,IDE 硬盘相当于机械部件,而 IDE 接口则是电子部件。典型的设备控制器是各种 PCI 或 PCI EXPRESS 卡(适配器),IDE 接口一般就是一种 PCI 设备。

控制器卡上一般都有两个方向的接口:一个用于连接主机的总线,而另一个则用于连接设备。为了达到通用性,这些接口通常都符合特定的标准,如 ANSI、IEEE、ISO 或企业自定标准等。

图 6.1 描述的是一个单总线模型的 I/O 组织结构。从图中可以看出,设备总是通过控制器与设备进行通信。因此,操作系统实际上总是通过设备控制器来对设备进行控制和操作。

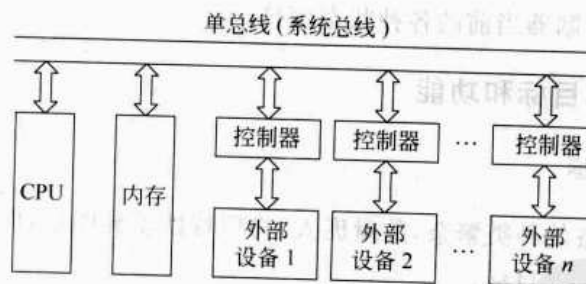


图 6.1 连接 CPU、存储器、控制器和设备的单总线模型

设备控制器是一个可编址的器件。当它控制一个设备时,它应用一个设备地址,而当它控制多个设备时,则应具有多个设备地址(每个地址对应一台设备)。设备控制器与设备之间的接口中包含三类信号:数据信号、控制信号和状态信号。设备控制器根据 CPU 提供的设备地址去选择一个设备接口。

每个控制器都有一些用来与 CPU 通信的寄存器(称为 I/O 端口),用于数据、控制命令和状态等信息的存储。为了寻址这些 I/O 端口,需要提供相应的寻址机制。在某些计算机上,这些寄存器占用内存地址空间的一部分,这种方案称作内存映像 I/O。而在其他

计算机上, I/O 端口则使用专用的 I/O 地址(独立于内存单独编址)。PC 混合采用了以上两种方案: 显存采用了内存映像 I/O, 而其他 I/O 端口则是独立于内存编址。现代计算机设备更多的是采用内存映像 I/O。

操作系统只需通过向 I/O 端口写入相应的参数对控制器进行初始化, 然后就可以去做其他工作, 控制器将自行控制相应设备完成指定的工作。

除 I/O 端口外, 许多控制器还使用中断来通知 CPU 它们已做好准备。硬件的中断请求通过信号线或者以类似于数据的方式(MSI/MSI-X)向 CPU 发送。

设备控制器通常应具有如下功能。

(1) 接收主机的控制命令。发出控制命令后, CPU 转去执行其他工作。而控制器在收到控制命令后, 独立控制设备完成相应的动作。当命令完成后, 控制器发出一个中断, 使 CPU 执行相应的中断处理程序。

(2) 地址识别。能够根据 CPU 发来的地址信息, 识别出其对应的设备。

(3) 实现数据的缓冲。设备的种类不同, 缓冲区的大小差别也很大。例如, 显卡的显存通常有几兆, 而串口则只有几个字节。

(4) 实现主机和控制器、控制器和设备之间的数据交换和格式转换。例如, 磁盘控制器把来自磁盘驱动器的串行位流进行组装, 存入控制器内部的缓冲区, 形成以字节为单位的块, 然后进行错误校验, 若无误, 则把该块复制到内存中。

(5) 提供设备和控制器当前的各种状态信息。

6.1.3 设备管理的目标和功能

1. 设备管理的目标

计算机系统中设备的种类繁多, 数量庞大, 物理特性差异明显, 而且变化众多, 因此设备管理通常需要达到以下目标。

(1) 使设备独立于用户程序。

在多道系统中, 为了防止个别用户程序对设备的滥用和破坏, 通常由操作系统实现设备的控制和管理的工作, 而用户程序不能直接对物理设备进行操作。当用户程序需要使用设备时, 需要向操作系统提出请求, 由操作系统根据设备的具体使用情况分配合适的设备给其使用。

为了做到这一点, 通常将物理设备抽象为逻辑设备, 并为逻辑设备提供统一的命名机制。例如, 在 Linux 中, 大多数设备都可以文件名的形式使用。这样, 当用户使用设备时, 实际上就是使用统一命名的逻辑设备, 而非物理设备。由操作系统负责把逻辑设备对应到相应的物理设备, 从而使用物理设备独立于用户程序, 因此称为设备的独立性。

(2) 向用户提供设备的统一方便的使用接口。

由于设备的物理特性复杂,且相互差异很大。如果由用户程序来直接控制设备的 I/O 操作,不仅程序设备任务量大,而且极易出错。因此,操作系统应屏蔽设备的物理特性和管理细节,对各种不同的设备向用户提供统一方便的使用接口。例如,无论是磁盘,还是串口,只要是读操作,用户都可以通过系统调用 read 完成。

(3) 尽量提高设备的使用效率。

在多道系统中,设备管理子系统必须和进程管理子系统各级组织有效地配合,从而尽量提高设备和设备之间、CPU 和设备之间,以及进程和进程之间的并行操作程度,以使操作系统获得最佳效率。

2. 设备管理的功能

为了完成上述主要任务,设备管理子系统一般要提供下述功能。

(1) 监视设备状态。

操作系统需要对系统中存在的各种设备控制器和通道的运行状态进行记录和监视,这是设备管理和控制的基础。

(2) 进行设备分配。

在多道系统中,通常不可避免地会形成多个进程对设备的竞争。设备管理的一个重要任务,就是按照设备类型和相应的分配算法把设备分配给请求该设备的进程,并把未分配到所请求设备的进程放入等待队列。在进程使用完设备后,还需要回收设备并分配给其他等待进程使用。完成这项功能的程序称为设备分配程序,具体实现有多种形式。

(3) 实现设备和 CPU 等之间的并行操作。

这部分工作通常由设备驱动程序完成,当然也需要有相应的硬件支持。除了装有控制状态寄存器、数据缓冲寄存器等的控制器之外,对应于不同的输入输出控制方式,还需要有 DMA、通道等硬件。在具有通道(或 DMA)的系统中,在设备分配程序根据进程要求分配了设备、控制器和通道(或 DMA)等硬件之后,通道(或 DMA)将自动完成设备和内存之间的数据传送工作,从而完成并行操作的任务。在没有通道(或 DMA)的系统里,则由设备驱动程序利用中断技术来完成上述并行操作。现代设备接口一般采用 PCI 或 PCI EXPRESS,而符合 PCI 和 PCI EXPRESS 标准的设备自己具有 DMA 能力。

(4) 进行缓冲区管理。

一般来说,CPU 的执行速度和访问内存速度都比较高,而外部设备的数据流通速度则低得多(例如键盘),为了减少外部设备和内存与 CPU 之间数据速度不匹配的问题,大多数 I/O 操作都设有缓冲区(器)来暂放数据。设备管理子系统负责进行缓冲区分配、释放及有关的管理工作。

6.1.4 I/O 设备的控制方式

I/O 控制方式是指设备与内存或 CPU 之间的数据传送方式。随着计算机技术的发展, I/O 控制方式也在不断地发展, 主要目标就是降低 I/O 的系统开销, 提高传输速度, 并使 CPU 和外设更高效的并行工作。

根据数据传送过程中 CPU 干预程序的不同, I/O 控制方式主要有以下 4 种。

- (1) 程序直接控制方式(也称轮询方式)。
- (2) 中断控制方式。
- (3) DMA 方式。
- (4) 通道方式。

1. 程序直接控制方式

程序直接控制方式就是由用户进程来直接控制内存或 CPU 与外设之间的信息传送。这种方式的控制者是用户进程。当用户进程需要数据时, 它通过 CPU 发出启动设备准备数据的启动命令, 然后用户进程进入测试等待状态。在等待时间内, CPU 不断地用测试指令检查描述外设的工作状态的控制状态寄存器。而外设只有将数据传送的准备工作做好之后, 才将该寄存器置为“准备完毕”状态。从而, 当 CPU 检测到控制状态寄存器为“准备完毕”状态之后, 设备开始往内存或 CPU 传送数据。反之, 当用户进程需要向设备输出数据时, 也必须同样发送启动命令来启动设备, 并等待设备准备好之后才能输出数据。

程序直接控制方式虽然控制简单, 不需要特别的硬件支持, 但是, 明显地存在下述缺点。

(1) CPU 和外设只能串行工作。由于 CPU 的处理速度要大大高于外设的数据传送和处理速度, 所以, CPU 的大量时间都处于等待和空闲状态。这使得 CPU 的利用率大大降低。

(2) CPU 在一段时间内只能和一台外设交换数据信息, 从而不能实现设备之间的并行工作。

因此, 程序直接控制方式只适用于那些 CPU 执行速度较慢且外设较少的系统。

在 Linux 系统中, 打印机的驱动程序(drivers/char/lp.c)采用 `lp_wait_ready(int minor, int nonblock)` 函数查看并口状态, 在并口没有到达就绪状态前, 函数一直等待。其中, `minor` 表示次设备号, `nonblock` 表示是否采用非阻塞方式。

2. 中断方式

中断方式通过硬件中断来减少 CPU 等待时间, 以提高系统的并行工作程度。在需

要时,设备控制器可以向 CPU 发出硬件中断信号。发送的方法有多种,例如 PCI 设备通过 INTx 触发中断,中断信号会发送到中断控制器,中断控制器再以数据的形式发送到 CPU。MSI/MSI-X 是另外一种中断发送方式,设备直接向 CPU 以数据的形式发送中断,CPU 会把该数据解释成中断。

在中断控制方式中,当进程需要数据时,首先通过 CPU 发出指令启动外设准备数据。之后,该进程放弃处理机,调度程序调度其他就绪进程运行。当输入完成时,I/O 控制器向 CPU 发出中断信号。CPU 在接收到中断信号之后,运行预先设计好的中断处理程序对数据传送工作进行相应的处理。

在中断方式中,当 CPU 发出启动设备和允许中断指令之后,并不像程序直接控制方式那样循环测试状态控制寄存器的状态是否已处于“准备就绪”状态,而是被调度程序分配给其他进程执行。当设备将数据送入缓冲寄存器并发出中断信号之后,CPU 接收中断信号进行中断处理。显然,此时即使 CPU 处在别的进程的上下文中,也可以发出启动其他不同设备的启动指令,或者指示设备进行下一个设备请求的处理,从而做到设备与设备间的并行操作及设备与 CPU 间的并行操作。

尽管中断方式 CPU 的利用率大大提高,并且能支持多道程序和设备的并行操作,但仍然存在一些不足。

(1) 由于 I/O 控制器的数据缓冲寄存器装满数据之后将会发生中断,而且数据缓冲寄存器通常较小,因此,在一次数据传送过程中,发生中断次数较多,这仍将耗去大量的 CPU 处理时间。

(2) 现代计算机系统通常配置有各种各样的外设。如果这些设备均通过中断处理方式进行并行操作,则由于中断次数的急剧增加而造成 CPU 无法响应中断和出现数据丢失现象。

(3) 在中断控制方式中,总是假定外设的速度非常低,而 CPU 处理速度非常高。如果外设的速度也非常高,则可能造成数据缓冲寄存器的数据由于 CPU 来不及取走而丢失。

下面介绍的 DMA 方式和通道方式则不会造成上述问题。

3. DMA 方式

DMA(direct memory access)方式的基本思想是借助于 DMA 控制器在外设和内存之间开辟直接的数据交换通路,从而在无需 CPU 控制的情况下实现内存和设备之间的成批数据交换。

在 DMA 控制器中,除控制状态寄存器和数据缓冲寄存器之外,还包括传送字节计数器、内存地址寄存器等。它通过窃取或挪用 CPU 的一个工作周期把数据缓冲寄存器中的数据直接送到内存地址寄存器所指向的内存区域。批量数据(数据块)的传送由计数器

逐个计数,并由内存地址寄存器确定内存地址(如图 6.2 所示)。除了在数据块传送开始时需要 CPU 的启动指令和在整个数据块传送结束时需发中断通知 CPU 进行中断处理之外,不再像中断控制方式那样需要 CPU 的频繁干涉。

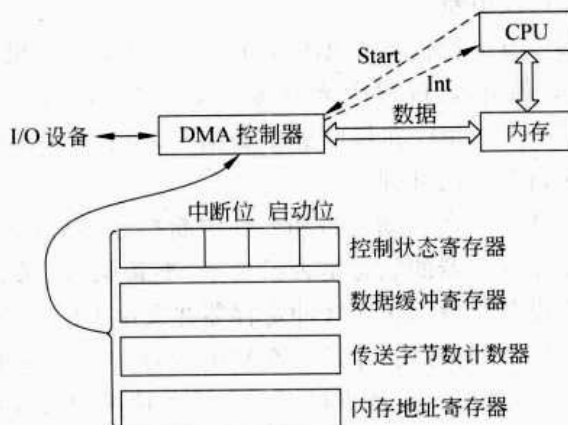


图 6.2 DMA 控制器的组成及作用

DMA 方式的数据输入处理过程如下。

(1) 当进程要求设备输入数据时,CPU 把准备存放输入数据的内存地址及要传送的字节数分别送入 DMA 控制器中的内存地址寄存器和传送字节计数器。另外,还把控制状态寄存器中的中断允许位和启动位置 1,从而启动设备开始进行数据输入。

(2) 发出数据要求的进程进入等待状态,进程调度程序调度其他进程到 CPU 上运行。

(3) 输入设备不断地挪用(CPU 对)总线的工作周期,将数据缓冲寄存器中的数据源源不断地写入内存,直到所要求的字节全部传送完毕。

(4) DMA 控制器在传送字节数完成时通过中断请求线发出中断信号,CPU 在接收到中断信号后转中断处理程序进行善后处理。

(5) 中断处理结束时,CPU 返回被中断进程处执行,或被调度到新的进程上下文中执行。

DMA 方式与中断方式的一个主要区别是:中断方式是在数据缓冲寄存器装满之后发出中断请求,要求 CPU 进行中断处理;而 DMA 方式则是在所要求转送的数据块全部传送结束时,才要求 CPU 进行中断处理,这大大减少了 CPU 进行中断处理的次数。另一个主要区别是:中断方式的数据传送是在中断处理时由 CPU 控制完成的;而 DMA 方式是在 DMA 控制器的控制下不经过 CPU 控制完成的,这就排除了因并行操作设备过多时,CPU 来不及处理或因速度不匹配而造成数据丢失等现象。

不过,DMA 方式仍存在着如下一些局限性。

(1) DMA 方式对外设的管理和某些操作仍由 CPU 控制。系统所配置的外设种类越来越多,数量也越来越大,因而对外设管理的控制也就越来越复杂。

(2) 多个 DMA 控制器的同时使用显然会引起对总线访问的冲突并使得控制过程进一步复杂化。多个 DMA 控制器的同时使用也是不经济的。

PCI 设备由于自己具有 DMA 能力,无需计算机另外提供 DMA 控制器,所以以上局限已经不再那么明显。

在计算机系统中,除了设置 DMA 器件之外,还设置专门的硬件装置——通道。

4. 通道控制方式

通道(channel)是一个独立于 CPU 的专用于 I/O 控制的处理机(硬件),它控制设备与内存直接进行数据交换。它有自己的通道指令,这些通道指令由 CPU 启动,并在操作结束时向 CPU 发中断信号。

在通道控制方式中,I/O 控制器中没有传送字节计数器和内存地址寄存器,但多了通道设备控制器和指令执行机构。在通道方式下,CPU 只需发出启动指令,指出通道相应的操作和 I/O 设备,该指令就可启动通道并使该通道从内存中调出相应的通道指令执行。

通道指令一般包含有被交换数据在内存中应占据的位置、传送方向、数据块长度,以及被控制的 I/O 设备的地址信息、特征信息等。在通道中没有存储器时通道指令存放在内存中。

通道控制方式的数据输入处理过程如下。

(1) 当进程要求设备输入数据时,CPU 发启动指令指明 I/O 操作、设备号 and 对应通道。

(2) 对应通道接收到 CPU 发来的启动指令后,把存放在内存中的通道指令程序读出,设置对应设备的 I/O 控制器中的控制状态寄存器。

(3) 设备根据通道指令的要求,把数据送往内存中指定区域。

(4) 若数据传送结束,I/O 控制器通过中断请求线发中断信号请求 CPU 做中断处理。

一个通道可以以分时方式同时执行几个通道指令程序。按照信息交换方式的不同,通道可分为三种类型:字节多路通道、数组多路通道和选择通道(如图 6.3 所示)。

字节多路通道以字节为单位传送数据,它主要用来连接大量的低速设备,如终端、打印机等。数组多路通道以块为单位传送数据,它具有传送速率高和能分时操作不同的设备等优点。数组多路通道主要用来连接中速块设备,如磁带机等。

数组多路通道和字节多路通道都可以分时执行不同的通道指令程序。但是,选择通道一次只能执行一个通道指令程序。所以,选择通道一次只能控制一台设备进行 I/O 操

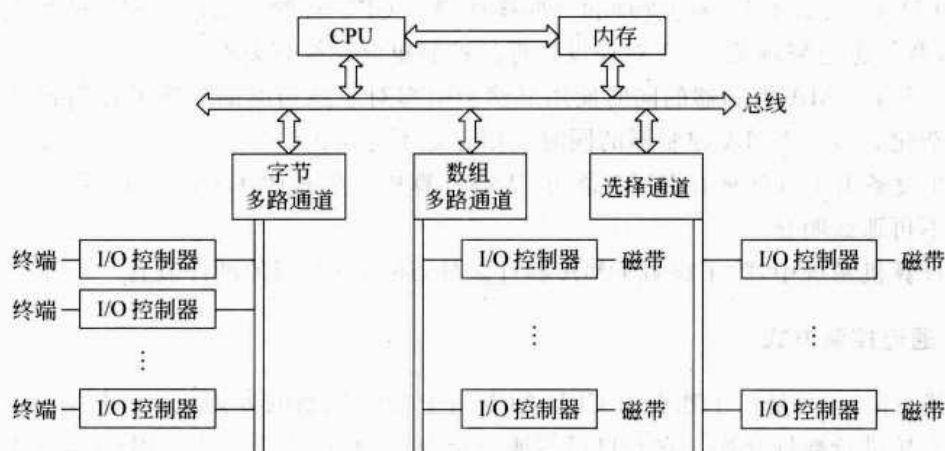


图 6.3 通道方式的数据传送结构

作。不过,选择通道具有传送速度高的特点,因而它被用来连接高速外部设备,并以块为单位成批传送数据。受选择通道控制的外设有磁盘机等。

与 DMA 方式类似,通道控制方式也是一种以内存为中心,实现设备和内存直接交换数据的控制方式。与 DMA 方式不同的是:在 DMA 方式中,数据的传送方向、存放数据的内存始址及传送的数据块长度等都由 CPU 控制;而在通道方式中,这些都由通道来进行控制。另外,与 DMA 方式时每台设备至少一个 DMA 控制器相比,通道控制方式可以做到一个通道控制多台设备与内存进行数据交换,从而进一步减轻了 CPU 的工作负担,并提高了系统的并行工作程度。

6.2 I/O 软件

6.2.1 I/O 软件的分层结构

虽然需要控制和管理的 I/O 设备种类繁多、性能差异很大,但是从用户的使用角度来考虑,还是希望系统能够为他们提供一种通用一致的设备管理方案。因此, I/O 软件的总体设计目标如下。

(1) 设备独立性。操作系统能够屏蔽 I/O 设备细节及其相互之间的差异,向用户提供一个友好、通用、一致并且规范的使用接口。也就是应用程序应该独立于具体使用的物理设备。

(2) 提高系统整体效率。由于 CPU 的运行速度比 I/O 设备的工作速度高几个数量级,因此需要采用直接或间接的手段(如缓冲技术)来加快 I/O 操作的速度,以提高整体性能。

为了满足上述两个目标,通常采用分层的方法来实现 I/O 管理。一般将通用的 I/O 软件自底向上分成以下 4 个层次(如图 6.4 所示)。

- (1) 与硬件相关的中断处理程序(硬件与软件接口)。
- (2) 设备驱动程序或叫设备 I/O 层。
- (3) 设备无关软件,是指与设备硬件操作无关的 I/O 管理软件,也叫逻辑 I/O 层。
- (4) 用户空间的 I/O 软件。

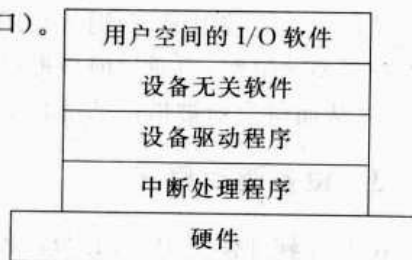


图 6.4 I/O 软件的分层模型

在上述分层结构中,除了用户空间的 I/O 软件在运行时位于进程的用户空间外,其他三层均位于内核空间。用户空间的 I/O 软件以系统调用的方式使用下层相关功能。

下面具体讨论各个层次应该完成的具体功能。

6.2.2 中断处理程序

尽管程序直接控制 I/O 也有偶尔适用的时候,但绝大多数情况下,中断是 I/O 控制中不可缺少的手段(其他三种控制方式都需要)。中断处理程序位于 I/O 软件的最底层,当输入就绪、输出完成或者出错时,设备控制器就向 CPU 发出中断信号。CPU 接到中断请求后,如果中断优先级高于运行程序的优先级,则响应中断,然后把控制权交给中断处理程序。中断处理程序分析中断原因,依据中断原因调用相应的处理程序。

不同的 I/O 设备,其中断处理程序的内容和处理步骤会因硬件平台的不同而有所不同,通常包括以下主要步骤。

- (1) 保存没有被中断硬件自动保存的其他寄存器(压栈)。
- (2) 为中断服务程序设置运行上下文,包括 TLB、MMU 和页表等。
- (3) 为中断服务程序设置堆栈。
- (4) 应答中断控制器,使其放开中断,可以接受新的中断。
- (5) 将寄存器从它们被保存的地方(可能是某个堆栈)复制到进程表中。
- (6) 运行中断服务程序,从发出中断的设备控制器的寄存器中提取信息。
- (7) 选择下一次运行哪个进程,如果中断导致某个被阻塞的高优先级进程变为就绪,则可能选择它现在就运行。
- (8) 为下一次要运行的进程设置 MMU 上下文,也许还需要设置某个 TLB。
- (9) 将保存的新进程的寄存器值装入寄存器。
- (10) 开始运行新进程。

由此可见,中断处理需要花费相当多的 CPU 指令,特别是寄存器、MMU、TLB 和页表等的保存和恢复。

为了加快中断处理的速度,在 Linux 系统中,中断处理程序会借用中断发生时正在运行的进程的页表、MMU 和内核栈等上下文。采用这种方法的限制是中断处理程序在运行的过程中不能被阻塞,否则就会阻塞被借用的进程。解决的办法是:如果一个中断处理程序需要被阻塞,则通过内核提供的函数将其提升为内核线程,使其具备被调度(阻塞)的条件,从而将当前被借用的进程解放出来。

6.2.3 设备驱动程序

在 I/O 软件模型中,所有与设备相关的代码都放在驱动程序或中断处理程序中。由于驱动程序与设备硬件密切相关,所以通常每种硬件设备需要一种驱动程序。

设备驱动程序通常必须是内核的一部分,也就是运行在内核态,因此要求其必须安全可靠。当然,让驱动程序运行在用户态并通过系统调用来控制设备,从理论上可使操作系统更加安全,但目前通用的操作系统都没有这样做。

设备驱动程序可静态连接到内核二进制代码中,但更多情况是当系统运行时根据对设备的具体需求,(在 Linux 系统中以模块的形式)动态加载其相应的驱动程序,以减少不必要的内存开销。

1. 设备驱动程序的功能

设备驱动程序是控制设备动作(如打开、关闭、读和写等)的核心模块,用来控制设备的运行及其与计算机之间的数据传输。一般来说,设备驱动程序具有以下几项功能。

(1) 接受来自上层的逻辑 I/O 层的抽象读写请求,将其转换为具体读写请求并将该请求排在请求队列的队尾,同时还要检查 I/O 请求的合法性。

(2) 取出请求队列中队首请求,将相应的设备分配给它。

(3) 检查设备的工作状态,向设备控制器发送命令和设置参数,启动该设备工作,完成指定的 I/O 操作。

(4) 执行确定的缓冲区策略。

(5) 及时响应控制器或通道发来的中断请求,并调用相应的中断处理程序进行处理。

2. 设备驱动程序的处理过程

这里以块设备为例说明设备驱动程序的处理过程。

假设上层的逻辑 I/O 层请求读取块设备的第 n 块。如果请求到来时驱动程序空闲,则驱动程序立即执行该请求,因为设备的速度较 CPU 慢许多,所以这时进程会在所请求的数据块缓存上阻塞,并调度新的进程运行。但如果驱动程序正在处理另一个请求,则将该请求挂在一个请求队列中,相应的请求进程同样也阻塞于所请求的数据块,调度程序调度别的进程运行。

当完成一个请求的处理时,设备控制器会向系统发出一个中断信号。来自设备的中断包括数据传输完成的结束中断、传输错误中断和设备故障中断。结束中断的处理是把设备控制器和通道的控制块中的状态位设置成“空闲”,然后查看请求队列是否为空。如果为空,则设备驱动程序返回;如果不为空,则处理请求队列中下一个请求。若是传输错误中断,则向系统报告错误或者相应进程重复执行处理。对于故障中断,则向系统报告故障,由系统做进一步的处理。

3. 逻辑 I/O 与设备驱动程序接口

为了能够被上层的逻辑 I/O 层以与设备无关的方式使用,各种设备驱动程序需要向逻辑 I/O 层提供统一的使用接口。在 Linux 系统中,设备访问采用与文件访问完全相同的使用接口,也就是 VFS 对文件的使用接口。设备驱动程序本质上就是要实现这些接口。

尽管访问设备和访问文件对用户来说是一样的,没有什么区别,但是在内核里面设备文件和普通文件在树目录中的索引节点是有区别的。只有这样,逻辑 I/O 层才会从该索引节点中的文件类型域中确认出是硬件设备还是普通文件,并确定要访问相应的设备驱动程序还是文件。

Linux 对设备及其驱动程序的注册和管理将在第 7 章中进行介绍。

6.2.4 设备无关

设备无关软件层统称 I/O 子系统。该层处理的是与用户 I/O 操作请求有关,但与具体 I/O 设备的驱动程序无关的操作。或者说,逻辑 I/O 层实现启动所有 I/O 设备驱动程序之前都必须执行的操作。通过逻辑 I/O 层向用户层提供通用一致的接口,如打开、读、写和关闭等系统调用,给用户使用。所以,为了实现设备独立性的设计目标,在本层需要解决以下几个问题。

- (1) 设备命名方法和使用接口。
- (2) 访问设备的权限和设备保护。
- (3) 为设备分配缓冲区。
- (4) 处理独占设备。
- (5) 错误报告。

1. 设备命名方法和使用接口

如何给文件和 I/O 设备命名是 I/O 子系统要解决的首要问题。在建立了统一的命名规范后,逻辑 I/O 层负责把设备的符号名映射到它所对应的设备驱动程序上。显然,设备的命名简单易用为好。

在 Linux 内核中,使用设备类型和(主次)设备号来标识设备。设备类型指字符设备还是块设备。设备号包括主设备号和次设备号两部分。主设备号用于表示设备的种类,通常主设备号是与驱动程序一一对应的。次设备号用于区分同一种设备中的各个具体设备。例如,系统中的块设备 IDE 硬盘的主设备号是 3,而多个 IDE 硬盘及其各个分区分别赋予次设备号 1、2、3 等。

Linux 系统中字符设备和块设备的主设备号独立编号,也就是说,两者的主设备号可以相同,但分别表示一种字符设备和一种块设备。

2. 访问设备的权限和设备保护

与设备命名机制密切相关的是设备保护。为了保护设备安全,并保证设备能够正确地使用,在目前的绝大多数系统中,完全禁止用户进程对 I/O 设备的直接访问。用户进程只能通过系统调用,间接地利用操作系统完成对设备的使用。

在 Linux 系统中,针对某一设备的访问权限同普通文件的访问权限一样,将所有的用户分为三类:文件主(u)、组用户(g)和其他用户(o)权限。文件的创建者是默认的文件主,文件主所属的组中的用户是默认的组用户,除此之外的用户为其他用户。每类用户对文件或设备的访问方式又可分为三种:读(r)、写(w)和可执行(x),其中 x 表示的可执行权限对设备来说没有意义。文件主、组用户及每类用户的操作权限都可以使用命令进行修改。系统资源(如设备)的文件主通常都是超级用户。

当一个用户进程对设备进行操作时,Linux 要求进程所属的有效用户对该设备需要具备相应的操作权限,否则操作将被拒绝。

3. 为设备分配和释放缓冲区

不同的磁盘驱动器可能采用不同的扇区尺寸,即各种磁盘的物理块大小可能不同。逻辑 I/O 层的一个任务就是向高层的用户进程屏蔽这些物理细节,并提供统一的使用接口。为了给较高层软件提供统一尺寸的数据块,逻辑 I/O 层负责将设备驱动程序传来的物理扇区数据块合并成一个由系统定义的逻辑块大小。这样高层软件只与逻辑设备打交道,而不必了解底层硬件物理扇区的尺寸。同样,一些字符设备每次传送的字符数也不一样,这些硬件差别都由逻辑 I/O 层向上层屏蔽起来。

在向一个新文件写入数据时,内核要为该文件分配新的磁盘块,在空闲块表或位图上查找一个空闲块是与设备无关的,因此可以放在逻辑 I/O 层处理。

4. 独占设备的分配

有些设备,如磁盘驱动器,不能共享使用而只能独占。这就需要在用户请求使用这些设备时,由逻辑 I/O 层负责对此请求进行检查,如果所请求的设备当前没有被使用,则接

受这个请求,否则请求将被拒绝。一个简单的处理这些请求的方法是要求进程直接通过 open 系统调用打开设备的特殊文件来提出请求。当设备不可用时,该调用失败。Linux 系统采用这种方法。

5. 错误处理

系统对设备发生错误的处理由两部分来完成。因设备错误大部分是由设备硬件造成的,所以这样的错误由设备驱动程序来处理,例如第一次读数据失败后,驱动程序可以再重试一次,或者有些错误可以忽略。如果发生的错误不能忽略,重试又失败,设备驱动程序才将错误报告给逻辑 I/O。由逻辑 I/O 决定将错误进一步报告给用户还是报告给内核。例如,如果在读一个用户文件时出现错误,则将错误信息报告给调用者。而如果是在读一些关键系统数据结构时出错,如磁盘的空闲块位图,则逻辑 I/O 只能打印一些信息并出错返回。

6.2.5 用户空间的 I/O 软件

尽管大部分的 I/O 软件都包含在内核中,但也有一小部分 I/O 软件是以位于用户空间的库(函数)的形式存在的。通常使用系统调用时需要用到汇编语言(以实现陷入),为了使用的方便,就将这些代码以库函数的形式提供。例如,在 C 语言中,有如下语句:

```
count=write(fd, buffer, nbytes);
```

其中的 write 就是一个库函数。在该函数中,根据其三个参数会发起相应的 write 系统调用,以实现向文件 fd 的写操作。有些时候,经常将上例中的 write 函数就称为系统调用,其实是不准确的,因为这些函数(再如 read 函数)实际上只是在内部发起了相应的系统调用。

但也有一些库函数,确实自己完成了一些非常实际的 I/O 操作。例如,C 语言中的 printf 函数,其输入为一个格式字符串,其中可能带有一些变量,它随后调用 write 系统调用,输出格式化后的一个 ASCII 码串。与此类似的还有 scanf、gets 等。标准的 I/O 库包含了许多涉及 I/O 的过程,它们都是作为用户程序的一部分运行在用户空间的。

Linux 设备管理的特点是将物理设备看成文件(即设备文件),采用文件系统(VFS)的接口和系统调用来管理和控制设备。从设备向内存输入数据相当于读文件,而把内存中的数据写向设备则看成是写文件。

对于 Linux 系统中的用户程序来说,使用设备实际上就是使用设备文件。Linux 的设备文件名一般由两部分组成:第一部分 2~3 个字符,表示设备的种类,如串口设备是 cu、并口设备是 lp、IDE 普通硬盘是 hd、软盘是 fp 等;第二部分通常是字母或数字,用于区分同种设备中的单个设备,如 hda、hdb 和 hdc 分别表示第一块、第二块和第三块 IDE

硬盘。而 hda1、hda2 则表示第一块硬盘中的第一、第二个磁盘分区。

设备文件通常都直接存放在 /dev 目录中,如 /dev/hda2、/dev/lp0 等。

Linux 使用虚拟文件系统 VFS 作为统一的操作接口来处理文件和设备。与普通的目录和文件一样,每个设备也使用一个 VFS inode 来描述,其中包含设备类型(块设备或字符设备)的主、次设备号。

对设备的操作也是通过对文件操作的 file_operations 结构体来调用驱动程序的设备服务子程序。在打开一个设备准备使用时,设备管理子系统根据设备的主设备号从其管理的设备驱动程序列表中选择正确的驱动程序映射到文件操作的 file_operations 结构体。这样,当使用该文件的某个操作系统时,实际上就是使用的设备驱动程序中的操作函数。

例如,当进程要求从某个设备上输入数据时,由该设备的 file_operations 结构体得到其设备驱动程序的操作函数入口,然后调用其中的 read() 函数来控制设备完成数据输入操作。

同样,使用 file_operations 中的 open()、close() 和 write() 函数分别映射到设备驱动程序中的相应函数,完成对设备的启动、停止设备运行,向设备输出数据的操作。

需要说明的是,设备文件只对应字符设备和块设备,而网络设备并不与设备文件相对应。例如,系统中的第一块网卡只有逻辑设备名 eth0。因此,不能使用 Linux 的文件系统接口来管理和控制网卡设备,而是使用套接口(socket)以文件 I/O 的形式来对网络进行访问。

6.2.6 I/O 软件的层次调用关系

在需要 I/O 时,用户进程首先利用用户空间的 I/O 软件通过内核所提供的系统调用接口与设备无关软件交互,向设备无关软件提出 I/O 请求。设备无关软件接受用户的请求之后,通过它与设备驱动程序的接口调用设备驱动程序中对应的函数,完成向 I/O 设备布置任务的操作。当 I/O 设备完成相应的 I/O 操作后,通过中断激活对应的设备驱动程序,而设备驱动程序根据设备完成任务的情况向上层的设备无关软件层作出回应,设备无关软件层则根据驱动程序的报告情况决定如何向用户进程答复,如报告写操作完成,或读入的数据复制到用户空间,并唤醒进程。

6.3 缓冲技术

为了改善 CPU 与外设之间速度不匹配的矛盾,协调逻辑记录大小与物理记录大小不一致的问题,提高 CPU 和 I/O 设备的并行性,并减少 I/O 对 CPU 的中断次数和放宽对 CPU 中断响应时间的要求,目前的操作系统普遍采用了缓冲技术。

6.3.1 缓冲的引入

在操作系统中,引入缓冲的主要原因,可归结为以下几点。

(1) 改善 CPU 与 I/O 设备间速度不匹配的矛盾。

例如,当计算进程阵发性地把大批量数据输出到打印机上打印时,由于 CPU 输出数据的速度大大高于打印机的打印速度,因此,CPU 只好停下来等待。反之,在计算进程进行计算时,打印机又因无数据输出而空闲着。

如果设置一个缓冲区,程序输出的数据先送到缓冲区暂存,然后由打印机慢慢地输出。这时,CPU 不必等待,可以继续执行程序。实现了 CPU 与 I/O 设备之间的并行工作。事实上,凡在数据的到达速率与其离去速率不同的地方,都可设置缓冲,以缓和它们之间速度不匹配的矛盾。

(2) 降低对 CPU 的中断频率,放宽对中断响应时间的限制。

如果 I/O 操作每传送一个字节就要产生一次中断,那么设置了 n 个字节的缓冲区后,则可以等到缓冲区装满 n 个字节才产生一次中断,这样中断次数就减少到 $1/n$,而且中断响应的时间也可以相应的放宽。

(3) 提高 CPU 和 I/O 设备之间的并行性。

缓冲的引入可显著提高 CPU 和设备的并行操作程度,避免 CPU 与 I/O 设备间频繁的“互相等待”,提高系统的吞吐量和设备的利用率。

根据 I/O 控制方式,缓冲的实现方法有两种,一种是采用专用硬件缓冲区,例如 I/O 控制器中的数据缓冲寄存器。另一种方法是在内存中划出一个具有 n 个单元的专用缓冲区,以便存放输入输出的数据。内存缓冲区又称软件缓冲。

6.3.2 缓冲的种类

根据系统设置的缓冲区的个数,可把缓冲技术分为单缓冲、双缓冲和循环缓冲(如图 6.5 所示),以及缓冲池几种。

1. 单缓冲

单缓冲是在设备和处理机之间设置一个缓冲区。设备和处理机交换数据时,先把被交换数据写入缓冲区,然后需要数据的设备或处理机从缓冲区取走数据。单缓冲是 OS 提供的一种最简单的缓冲。

由于缓冲区属于临界资源,即不允许多个进程同时对一个缓冲区操作,因此,设备和设备之间不能通过单缓冲达到并行操作。

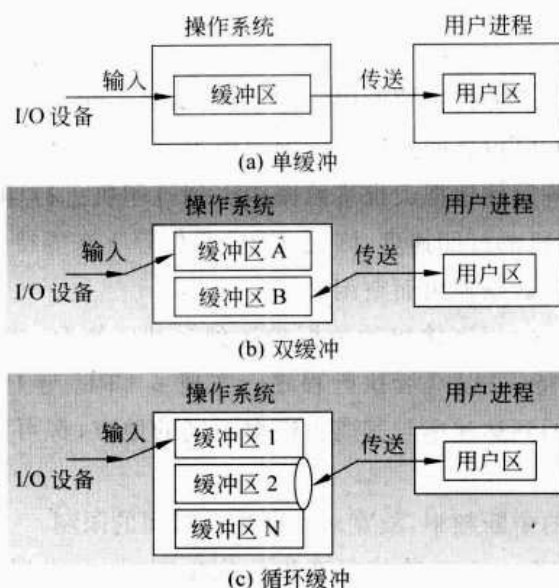


图 6.5 缓冲技术

2. 双缓冲

解决两台外设、打印机和终端之间并行操作问题的办法是设置双缓冲。有了两个缓冲区之后,CPU 可把输出到打印机的数据放入其中一个缓冲区,让打印机慢慢打印。然后,它又可以从另一个为终端设置的缓冲区中读取所需要的输入数据。

显然,双缓冲只是一种说明设备和设备、CPU 和设备并行操作的简单模型,并不能用于实际系统中的并行操作。这是因为当生产者-消费者的速度大致相匹配时,采用双缓冲基本可行。但若是两者的速度相差太大时,则速度较快的一方将需要等待。而在实际计算机系统中,外设间的速度通常相差甚远。而且,当计算机外设较多时,双缓冲也很难匹配设备和处理机的处理速度。因此,现代计算机系统中一般使用循环缓冲或缓冲池结构。

3. 循环缓冲

循环缓冲是把多个缓冲区连接起来组成两部分,一部分专门用于输入,另一部分专门用于输出的缓冲结构。一般将循环缓冲组织成循环缓冲形式。对于用作输入的循环缓冲,通常提供给输入进程和计算进程使用,输入进程不断向空缓冲区输入数据,计算进程则从中提取数据用于计算。

循环缓冲的缓冲区是系统的公共资源,可供各个进程共享,并由系统统一分配和管理。

在 Linux 系统中,无论是块设备还是字符设备,都使用循环缓冲技术。尤其是块设备,它作为文件系统的物质基础,用完整的多重缓冲技术提高了文件系统效率。

4. 缓冲池

循环缓冲区仅适用于某个特定的 I/O 进程和计算进程,因而属于专用缓冲。当系统较大时,将会有许多这样的循环缓冲,这不仅需要消耗大量的内存空间,而且其利用率也不高。

为了提高缓冲区的利用率,目前广泛流行的方式是采用公用缓冲池,池中的缓冲区可供多个进程共享。

缓冲池则是把多个缓冲区连接起来统一管理,既可用于输入又可用于输出的缓冲结构。

显然,无论是循环缓冲,还是缓冲池,由于缓冲区是临界资源,在使用缓冲区时都有一个申请、释放和互斥的问题。下面以缓冲池为例,介绍缓冲的管理。

6.3.3 缓冲池的管理

1. 缓冲池的结构

缓冲池由多个缓冲区组成。而一个缓冲区由两部分组成:一部分是用来标识该缓冲区和用于管理的缓冲首部,另一部分是用于存放数据的缓冲体。这两部分有一一对应的映射关系。对缓冲池的管理是通过每一个缓冲区的缓冲首部进行操作实现的。缓冲首部如图 6.6 所示,包括设备号、设备上的数据块号(块设备时)、互斥标识位,以及缓冲队列连接指针和缓冲区号等。

系统把各缓冲区按其使用状况连成三种队列。

(1) 空白缓冲队列 em , 其队首指针为 $F(em)$, 队尾指针为 $L(em)$ 。

(2) 装满输入数据的输入缓冲队列 in , 其队首指针为 $F(in)$, 队尾指针为 $L(in)$ 。

(3) 装满输出数据的输出缓冲队列 out , 其队首指针为 $F(out)$, 队尾指针为 $L(out)$ 。其队列构成如图 6.7 所示。

除了三种缓冲队列之外,系统(或用户进程)从这三种队列中申请和取出缓冲区,并用得到的缓冲区进行存数、取数操作,在存数、取数操作结束后,再将缓冲区放入相应的队列。这些缓冲区被称为工作缓冲区。在缓冲池中,有如下 4 种工作缓冲区。

(1) 用于收容设备输入数据的收容输入缓冲区 hin 。

(2) 用于提取设备输入数据的提取输入缓冲区 sin 。

设备号
数据块号
缓冲区号
互斥标识位
连接指针

图 6.6 缓冲首部

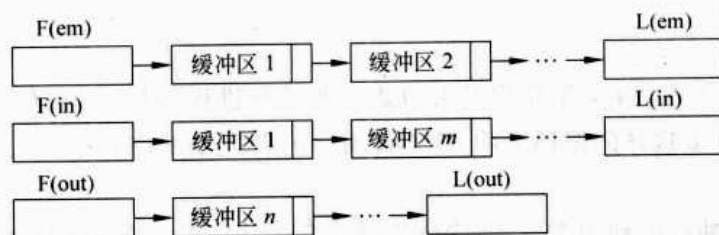


图 6.7 缓冲区队列

(3) 用于收容 CPU 输出数据的收容输出缓冲区 hout。

(4) 用于提取 CPU 输出数据的提取输出缓冲区 sout。

缓冲池的工作缓冲区如图 6.8 所示。

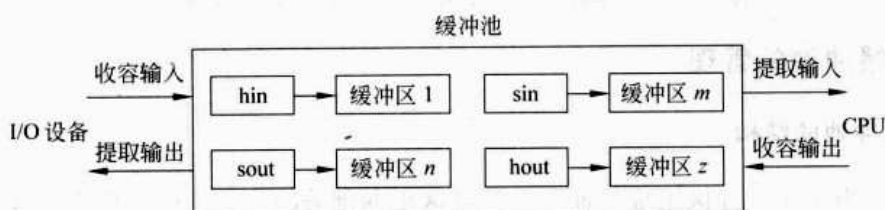


图 6.8 缓冲池的工作缓冲区

2. 缓冲池管理

对缓冲池的管理由如下几个操作组成。

(1) 从三种缓冲区队列中按一定的选取规则取出一个缓冲区的过程 `take_buf(type)`。

(2) 把缓冲区按一定的选取规则插入相应的缓冲区队列的过程 `add_buf(type, number)`。

(3) 供进程申请缓冲区用的过程 `get_buf(type, number)`。

(4) 供进程将缓冲区放入相应缓冲区队列的过程 `put_buf(type, work_buf)`。

其中,参数 `type` 表示缓冲队列类型,`number` 为缓冲区号,而 `work_buf` 则表示工作缓冲区类型。

使用这几个操作,缓冲池的工作过程可描述如下。

首先输入进程调用 `get_buf(em, number)` 过程从空白缓冲区队列中取出一个缓冲号为 `number` 的空白缓冲区,将其作为收容输入缓冲区 `hin`,当 `hin` 中装满了由输入设备输入的数据之后,系统调用过程 `put_buf(in, hin)` 将该缓冲区插入输入缓冲区队列 `in` 中。

另外,当进程需要输出数据时,输出进程经过缓冲管理程序调用过程 `get_buf(em, number)` 从空白缓冲区队列中取出一个空白缓冲区 `number` 作为收容输出缓冲区 `hout`,待 `hout` 中装满输出数据之后,系统再调用过程 `put_buf(out, hout)` 将该缓冲区插入输出缓冲区队列 `out`。

对缓冲区的输入数据和输出数据的提取也是由过程 `get_buf` 和 `put_buf` 实现的。`get_buf(out, number)` 从输出缓冲队列中取出装满输出数据的缓冲区 `number`,将其作为 `sout`。当 `sout` 中数据输出完毕时,系统调用过程 `put_buf(em, sout)` 将该缓冲区插入空白缓冲队列。而 `get_buf(in, number)` 则从输入缓冲队列中取出一个装满输入数据的缓冲区 `number` 作为输入缓冲区 `sin`,当 CPU 从中提取完所需数据之后,系统调用过程 `put_buf(em, sin)` 将该缓冲区释放和插入空白缓冲队列 `em` 中。

显然,对于各缓冲队列中缓冲区的排列及每次取出和插入缓冲队列区的顺序都应有一定的规则。最简单的方法是 FIFO,即先来先出的排列方法。采用 FIFO 方法,过程 `put_buf` 每次把缓冲区插入相应缓冲队列的队尾,而过程 `get_buf` 则取出相应缓冲队列的第一个缓冲区,从而 `get_buf` 中的第二个参数 `number` 可以省略。而且,采用 FIFO 方法也省略了对缓冲队列的搜索时间。

过程 `add_buf(type, number)` 和 `take_buf(type, number)` 分别用来把缓冲区 `number` 插入 `type` 队列和从 `type` 队列中取出缓冲区 `number`。它们分别被过程 `get_buf` 和 `put_buf` 调用,其中, `take_buf` 返回所取缓冲区 `number` 的指针,而 `add_buf` 则将给定缓冲区 `number` 的指针链入队列。

6.4 设备分配

6.4.1 与设备分配相关的因素

进程需要申请使用资源时,由于设备、控制器和通道等资源的有限性,不是每一个进程随时随地都能得到这些资源。为了使系统能够有条不紊地工作,系统必须具有合理的设备分配原则,这主要与下列 4 个因素相关。

1. I/O 设备的固有属性

某些设备要求人工干预,例如,把一盘磁带放到磁带机中。这种工作很费时间,如果这类设备由一个进程独占,待其使用完成后再由另一进程使用,则可节省人工干预的时间。而对于高速磁盘来说,则完全可由多个进程共享使用。

2. 系统所采用的分配算法

分配算法确定哪些进程可以得到设备,因此需要兼顾公平性,同时需要兼顾系统和设

备的效率和利用率。

3. 设备分配时应防止死锁

死锁直接导致系统内的部分进程永远无法得到正确的结果,而且使进程占用的资源无法被释放。将独占的设备(虚拟)共享化,使其通过 SPOOLing 技术做到共享,是解决死锁的一种有效方法。另外,通过要求进程按一定的顺序申请其所使用的资源,也可达到防止死锁的目的。

4. 使用户程序与实际使用的物理设备无关

为了提高系统的适应性和扩展性,用户程序中使用的设备都是逻辑设备,由系统根据用户的请求和资源使用情况,分配具体的物理设备。这可大大减少用户程序对设备的滥用或破坏。

6.4.2 设备分配的原则和方式

设备分配的原则是根据设备特性、用户要求和系统配置情况决定的。设备分配的总原则是既要充分发挥设备的使用效率,尽可能地让设备忙,又要避免由于不合理的分配方法造成进程死锁。另外还要做到把用户程序和具体物理设备隔离开来,即用户程序面对的是逻辑设备,而分配程序将在系统把逻辑设备转换成物理设备之后,再根据要求的物理设备号进行分配。

设备的分配方式有两种,即静态分配和动态分配。静态分配方式是在用户作业开始执行之前,由系统一次分配该作业所要求的全部设备、控制器和通道。一旦分配之后,这些设备、控制器和通道就一直为该作业所占用,直到该作业被撤销。静态分配方式不会出现死锁,但设备的使用效率低。因此,静态分配方式并不符合设备分配的总原则。

动态分配在进程执行过程中根据执行需要进行。当进程需要设备时,通过系统调用命令向系统提出设备请求,由系统按照事先规定的策略给进程分配所需要的设备、I/O 控制器和通道,一旦用完之后,便立即释放。动态分配方式有利于提高设备的利用率,但如果分配算法使用不当,则有可能造成进程死锁。

6.4.3 设备分配技术

在本章的开始部分,已经介绍了与设备分配相关的设备分类方式,即独占设备、共享设备和虚拟设备。对于这三种设备,都有其各自的设备分配技术。

1. 独占分配技术

独占分配技术是把独占设备固定地分配给一个进程,直至该进程完成 I/O 操作并且释放它为止。在该进程占用设备期间,即使闲置不用,也不能分配给其他进程使用。这是由设备的物理性质决定的。这样做的好处是使用起来方便,而且可避免死锁的发生。但采用这种分配方式,会降低设备的利用率。

例如,对行式打印机,如果在进程执行前就固定地分配好,那么在进程运行的过程中,即使没有打印输出,即打印机处于空闲状态,其他的进程也无法正常使用。

2. 共享分配技术

共享分配技术适用于高速、大容量的直接存取存储设备,如磁盘机和可读写 CD-ROM 等可共享设备。各个进程分时地使用这些设备,系统保证其彼此之间互不干扰,这使共享设备的利用率显著提高。

对于独占设备,也可采用这种共享分配的形式。例如打印机,在分配给一个进程打印完一段内容后,可暂时释放并分配给另外的进程使用。管理程序可以在打印内容前加入打印进程的进程名、文件名和用户名等信息,使用户可以很方便地识别出输出信息属于哪个用户。这样可大大提高设备的利用率。

3. 虚拟分配技术

虚拟分配技术利用共享设备去实现独占设备的功能,从而使独占设备从使用者的角度成为可共享的、快速的 I/O 设备。实现虚拟分配最成功的技术就是前面介绍的 SPOOLing 技术。

6.4.4 设备分配算法

与进程调度相似,动态设备分配也是基于一定的分配算法的。常用的分配算法有先请求先分配、优先级高者先分配算法等。

(1) 先请求先分配。

当有多个进程对某一设备提出 I/O 请求时,或者是在同一设备上进行多次 I/O 操作时,系统按提出 I/O 请求的先后顺序,将进程发出的 I/O 请求命令排成队列,其队首指向被请求设备。当该设备空闲时,系统从该设备的请求队列的队首取下一个 I/O 请求消息,将设备分配给发出这个请求消息的进程。

(2) 优先级高者先分配。

优先级指的是发出 I/O 请求命令的进程的优先级。这种策略和进程调度的优先级法是一致的,即进程的优先级高,它的 I/O 请求也优先予以满足。对于相同优先级的进

程来说,则按先请求先分配策略分配。因此,优先级高者先分配策略把请求某设备的 I/O 请求命令按进程的优先级组成队列,从而保证在该设备空闲时,系统能从 I/O 请求队列的队首提取下一个具有最高优先级进程发来的 I/O 请求命令,并将设备分配给发出该命令的进程。

6.5 I/O 控制

6.5.1 I/O 控制的引入

前面各节在描述了 I/O 数据传送控制方式的基础上,讨论了中断、缓冲技术及进行 I/O 数据传送所必需的设备分配策略与算法。那么,系统在何时分配设备,在何时申请缓冲,由哪个进程进行中断响应呢? 另外,尽管 CPU 向设备或通道发出了启动指令,设备的启动及 I/O 控制器中有关寄存器的值由谁来设置呢?

从用户进程的输入输出请求开始,给用户进程分配设备和启动有关设备进行 I/O 操作,在 I/O 操作完成之后响应中断,直到进行善后处理为止的整个系统控制过程,称为 I/O 控制。

6.5.2 I/O 控制的功能

I/O 请求处理是用户进程和设备管理程序接口的一部分,它把用户进程的 I/O 请求变换为设备管理程序所能接受的信息。一般来说,用户的 I/O 请求包括所申请进行 I/O 操作的逻辑设备名、要求的操作、传送数据的长度和起始地址等。I/O 请求处理模块对用户的 I/O 请求进行处理。它首先将 I/O 请求中的逻辑设备名转换为对应的物理设备名,然后检查 I/O 请求命令中是否有参数错误。在 I/O 请求命令参数正确时,它把该命令插入指向相应设备的 I/O 请求队列,然后启动设备分配程序。在有通道的系统中,I/O 请求处理模块还将按 I/O 请求命令的要求编制出通道程序。

在设备分配程序为 I/O 请求分配了相应的设备、控制器和通道之后,I/O 控制模块还将启动缓冲管理模块为此次 I/O 传送申请必要的缓冲区,以保证 I/O 传送的顺利完成。缓冲区的申请也可在设备分配之前进行。例如 Linux 系统首先请求缓冲区,然后把 I/O 请求命令写到缓冲区中并将该缓冲区挂到设备的 I/O 请求队列上。

另外,在数据传送结束后,外设发出中断请求,I/O 控制过程将调用中断处理程序做出中断响应。对于不同的中断,其处理不同。例如,处理结束中断时,要释放相应的设备和控制器,并唤醒正在等待该操作完成的进程。另外,还要检查是否还有等待该设备的 I/O 请求命令。如有,则要通知 I/O 控制过程进行下一个 I/O 传送。

6.5.3 I/O 控制的实现

I/O 控制过程在系统中可以有三种方式实现。

(1) 作为请求 I/O 操作的进程的一部分实现。这种情况下,请求 I/O 操作的进程应具有良好的实时性。系统应根据中断信号的内容准确地调度到请求所对应 I/O 操作的进程,并使其占据处理机。因为在大多数情况下,当一个进程发出 I/O 请求命令之后,都将被阻塞。

(2) 作为当前进程的一部分实现。这种情况下,不要求系统具有很高的实时性。但由于当前进程与完成的 I/O 操作无关,所以当前进程不能接受 I/O 请求命令的启动 I/O 操作。不过,当前进程可以在接收到中断信号后,将中断信号转交给 I/O 控制模块处理。因此,如果让请求 I/O 操作的进程调用 I/O 操作控制部分 I/O 请求处理、为设备分配缓冲区分配等,而让当前进程负责调用中断处理部分也是一种可行的 I/O 控制方案。

(3) I/O 控制由专门的系统进程——I/O 进程完成。在用户进程发出 I/O 请求命令后,系统调度 I/O 进程执行,控制 I/O 操作。同样,在外设发出中断请求之后,I/O 进程也调度执行以响应中断。

I/O 进程也可分为三种方式实现。

- ① 每类(个)设备设一个专门的 I/O 进程,该进程只能在系统状态下执行。
- ② 整个系统设一 I/O 进程,全面负责系统的数据传送工作。
- ③ 每类(个)设备设一个专门的 I/O 进程,但该进程可在用户态也可在系统态下执行。

Linux 的进程控制采用的是第二种方案,即作为当前进程的一部分实现,其本质实际上是由中断驱动的,只是借用了当前进程的上下文。

6.6 磁盘调度

6.6.1 磁盘存取时间

几乎所有的计算机都用磁盘存储信息,这是由于磁盘相对于内存有以下三个主要优点:容量很大;每位的存储价格非常低;当关掉电源后,存储的信息不丢失。但是近 30 年来,处理器和主存速度的提高远远大于磁盘。因此,磁盘的性能变得至关重要,很多人都在积极探索提高磁盘性能的方法,改进调度策略,降低查找时间。

存取磁盘中信息的时间一般分为如下三部分。

(1) 系统要把磁头移到相应的磁道或柱面上,这个时间叫做寻道时间。

(2) 一旦磁头到达指定磁道,必须等待所需要的扇区转到读写头下,这个延迟时间叫做旋转延迟时间。

(3) 信息实际在磁盘和内存之间进行传输也要花费时间,这部分时间叫做传输时间。

一次磁盘服务的总时间就是这三者之和。对于大多数磁盘来说,寻道时间远远大于旋转延迟时间与传输时间之和。所以,减少平均寻道时间可以显著地改善系统性能。

6.6.2 磁盘调度算法

1. 先来先服务法(First Come First Serve, FCFS)

顾名思义,就是按各进程对磁盘提出请求的时间对等待队列进行排序,并按此次序给予服务的一种调度算法。这种调度算法对各进程是公平的,它不管进程的优先级多高,只

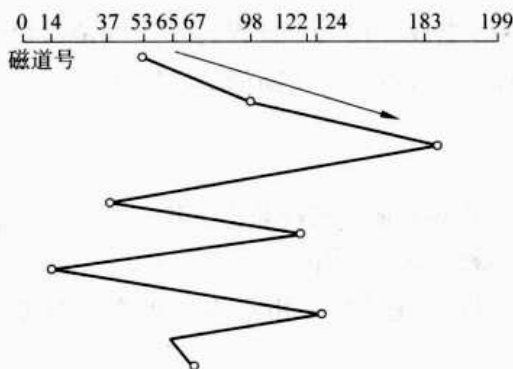


图 6.9 先来先服务调度算法示例

要是新来的访问请求,都被排在队尾。这种调度算法是最简单的,也是最容易实现的。但是它并没有提供最佳的服务(平均来说)。例如,有一个请求磁盘队列,要访问的磁道分别是 98, 183, 37, 122, 14, 124, 65, 67。设磁盘有 200 个磁道,磁头最初在 53 道上。这样磁头总共要移动 640 个磁道,这个调度如图 6.9 所示。

当用户提出的访问请求比较均匀地遍布整个盘面,而不具备某种集中倾向时(通常是这样的),FCFS 无法对访问进行优化。在对盘的访问请求较多的情况下,这种调度算法的平均寻道距离较大,响应时间较长并且会降低设备服务的吞吐量,但是各进程得到的服务的响应时间变化幅度较小。FCFS 适用于访问请求不是很多的情况。

在使用硬件 RAID 时,一般使用 FCFS 调度算法,因为操作系统对磁盘逻辑块的位置一般不会直接一一映射到磁盘物理块上,RAID 控制器需要对操作系统的请求重新排队。

2. 最短寻道时间优先法(shortest seek time first, SSTF)

这是将请求队列中柱面号最接近于磁头当前所在柱面的访问要求,作为一个调度对象的调度算法。

如果对上述请求队列使用 SSTF 算法,那么调度序列依次是 65,67,37,14,98,122,124,183。这个调度如图 6.10 所示。

采用这种调度算法,磁头总共移动了 236 个磁道,大概是先来先服务算法的 1/3,很明显这种调度算法可以得到比较好的吞吐量和较短的响应时间(与 FCFS 相比),它改善了磁盘服务。但是 SSTF 也存在缺点:对用户请求的响应机会不是均等的,对中间磁道的访问请求将得到较好的服务,对内外两侧磁道的服务随偏离中心磁道的距离增加而越来越差,因而导致响应时间的变化幅度很大。甚至它可能导致某些请求长期得不到响应(即出现“饥饿”问题)。在实际系

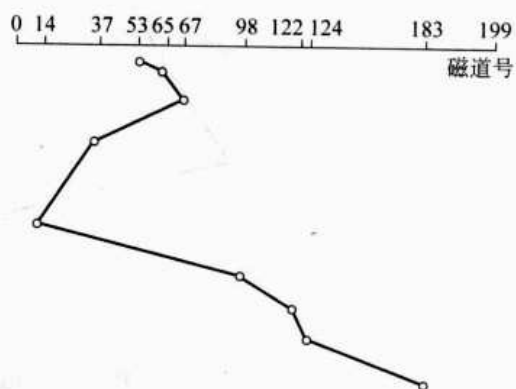


图 6.10 最短寻道时间优先调度算法示例

统中,请求可在任何时候到达。假设队列中有两个请求,14 和 186 道。当系统正在为 14 道服务时,一个靠近它的请求到来,那么这个靠近 14 道的请求将早于 186 道请求得到服务,如果这种彼此接近的请求流接连不断地到达,那么 186 道请求将无限期地等下去。

另外,SSTF 算法并不是最优的。例如,对于上述例子,若把磁头从 53 道移到 37 道(尽管它不是最靠近的),然后移到 14 道,接下去依次是 65,67,98,122,124,183 道,则磁头总移动量减小到 208 个磁道。

3. 扫描法(SCAN)

SCAN 算法也叫电梯算法(elevator algorithm),是将请求队列中,磁臂前进方向上最接近磁头当前所在柱面的访问要求作为下一个调度对象,若前进方向上没有请求,那么磁臂就改变移动方向。也就是说,如果磁臂目前向内移动,那么下一个调度对象应该是在磁头当前位置以内的柱面上诸访问请求中的最近者,这样依次地进行调度,直到没有更内侧的调度请求,磁臂才改变移动方向,转而向外移动,并依次服务于此方向上的访问请求。如果仍以上面请求序列为例,并且开始时磁头向内移动,那么磁头首先从 53 道移到 37 道,然后依次是 14,65,67,98,122,124,183 道,磁头总移动量为 208 个磁道。调度过程如图 6.11 所示。

SCAN 调度算法在一定程度上克服了 SSTF 算法的集中于中间磁道和响应时间变化较大的缺点。

4. 循环扫描法(C-SCAN)

C-SCAN 算法是 SCAN 算法的变种,可使等待时间变得更均匀。它的调度方法是:

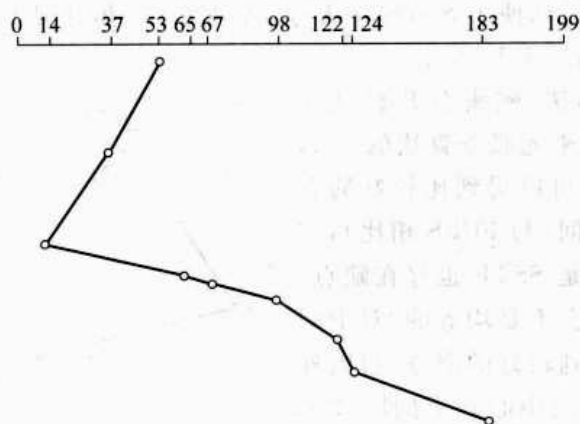


图 6.11 电梯调度算法示例

总是按同一方向移动磁臂,处理完最高(低)编号柱面上的请求后,磁臂立即返回请求的最低(高)编号柱面,然后继续按该方向处理请求。注意磁臂在返回过程中,不进行任何服务。也就是说循环扫描是单向反复地扫描,这是 C-SCAN 算法和 SCAN 算法的不同之处。

同样以前面访问序列为例,并假设调度方向是从内向外,那么调度序列依次为 65, 67, 98, 122, 124, 183, 14, 37 道。

根据模拟研究的结果,在磁盘访问负荷较小的情况下,SCAN 算法是最好的;而在中等以上负荷的情况下,C-SCAN 算法效果最佳。

6.6.3 性能比较

磁盘的调度策略很多,各有利弊。如何选择相应调度策略与磁盘的使用环境因素有关。表 6.1 比较了上述 4 种策略的性能。

表 6.1 磁盘调度策略的性能比较

FIFO 从磁道 53 开始		SSTF 从磁道 53 开始		SCAN 从磁道 53 开始按磁道数减少方向		C-SCAN 从磁道 53 开始按磁道数增加方向	
下次访问 的磁道	访问的 磁道数	下次访问 的磁道	访问的 磁道数	下次访问 的磁道	访问的 磁道数	下次访问 的磁道	访问的 磁道数
98	45	65	12	37	16	65	12
183	85	67	2	14	23	67	2
37	146	37	30	65	49	98	31
122	85	14	23	67	2	122	24

续表

FIFO 从磁道 53 开始		SSTF 从磁道 53 开始		SCAN 从磁道 53 开始按磁道数减少方向		C-SCAN 从磁道 53 开始按磁道数增加方向	
下次访问 的磁道	访问的 磁道数	下次访问 的磁道	访问的 磁道数	下次访问 的磁道	访问的 磁道数	下次访问 的磁道	访问的 磁道数
14	108	98	84	98	31	124	2
124	110	122	24	122	24	183	59
65	59	124	2	124	2	14	169
67	2	183	59	183	59	37	23
平均寻道数	80	平均寻道数	29.5	平均寻道数	26	平均寻道数	40.25

磁盘调度是影响系统性能的重要因素, Linux 内核实现了多种 I/O 调度器, 例如 deadline、noop、as 和 cfq。noop 就是先来先服务法。deadline 和 as 类似于扫描法。cfq (Completely Fair Queue, 完全公平队列) 表示在进程之间要保证 I/O 的公平性, 它给每一个进程一个 I/O 优先类和优先级, 优先类和优先级高的进程提交的 I/O 请求优先被调度。

6.7 本章小结

现代计算机系统外设的多样性和复杂性, 决定了分层的软件结构是设备管理的最佳选择。采用分层结构, 可以屏蔽硬件细节, 为用户提供统一、方便、易用的设备使用接口。

为了达到上述目的, 操作系统通常会按使用方式和技术特点对设备进行分类, 以便实现管理和服务。按设备的使用特性, 可分为存储设备、I/O 设备、终端设备及脱机设备等; 从资源的管理角度对设备分类, 可以分为独占设备、共享设备和虚拟设备; 而从内核管理的角度, 则分为字符设备、块设备和网络设备三大类。

设备管理的主要任务之一是控制设备与内存或 CPU 之间的数据传送, 外设和内存之间的常用数据传送控制方式有程序直接控制方式、中断控制方式、DMA 方式和通道方式 4 种。前两种方式仅适用于外设数量很少, 且传输数据量较小的系统; 而后两种方式则支持外设与内存之间的直接数据传输, 因此可大大减轻 CPU 的负担。

I/O 软件通常分为 4 层结构: 最上层是用户程序, 包括 I/O 库等; 第二层是与设备硬件操作无关的 I/O 管理软件, 或叫逻辑 I/O 层, 通常称这部分为操作系统的设备管理子系统; 第三层是设备驱动程序或叫设备 I/O 层, 所有与设备细节相关的代码都放在驱动程序中; 第四层是与硬件相关的中断处理程序, 用于响应设备对系统的请求。

在设备管理子系统中, 缓冲技术主要用于匹配外设和 CPU 的处理速度, 因此, 对于

保证系统性能非常重要。缓冲技术可分为单缓冲、双缓冲和循环缓冲,以及缓冲池几种。

I/O 进程控制介绍了整个 I/O 过程的三种控制方式:基于请求 I/O 的进程、基于当前进程及基于专门的 I/O 进程。Linux 采用第二种方式。

磁盘担负着繁重的输入输出任务,通过选用合适的磁盘调度算法,可在很大程度上提高系统的性能。磁盘调度的对象是对磁盘的输入输出请求,常用的调度算法包括先来先服务法、最短寻道时间优先法、扫描(电梯调度)法和循环扫描法等多种。

思考题

1. Linux 的设备文件的文件名是如何构成的? 设备文件在磁盘上的文件系统中是如何保存的? 采用什么数据结构? 在磁盘上是否有文件实体?
2. Linux 的设备文件是如何对应到实际的物理设备的? 如何根据设备文件找到其对应的设备驱动程序?
3. Linux 如何进行设备的注册? 设备管理程序如何区分字符设备和块设备? 时钟和显示器属于哪类设备? Linux 如何管理它们? 为什么?
4. 中断处理程序和驱动程序的关系是什么? 在 Linux 系统中,中断处理程序从外设获取的数据,如何能够正确交付给对应的设备驱动程序?
5. 以键盘为例,分析字符设备的打开和读取设备时所执行的主要操作,进而思考在 Linux 系统中什么样的非同类设备才能共享一个中断号,键盘的中断号可否与其他设备共享。
6. 通过 Linux 内核对块设备请求的管理数据结构,思考为什么会出现多个进程并发地对磁盘进行读写操作的情况。如果多个进程在同时请求一个相同的磁盘块,系统会从磁盘读入内存几次? 如果其中的一个进程进行了修改,其他进程能够马上看到修改的结果吗?

进一步阅读

关于字符设备和块设备的管理可进一步阅读本书的第 7 章。关于设备驱动程序和中断处理程序的关系可进一步阅读 *Linux Device Drivers, 3rd Edition* 一书的第 10 章。关于多进程在磁盘块请求时的行为可进一步阅读 *Understanding the Linux Kernel, 3rd Edition* 一书的第 14 章。

第7章 Linux 驱动程序编写基础

在 Linux 系统中,内核为不同类型的设备定义好了不同的接口。在实现驱动程序的过程中需要实现这些不同的接口。除了实现这些接口外,驱动程序开发者还需要考虑另外一些基础问题:驱动程序如何加载到内核?是否允许用户卸载驱动程序?驱动程序如何响应设备的中断?怎样提高系统的中断响应速度?本章着重介绍解决这些问题的一些基础知识。

7.1 内核模块概述

Linux 的内核模块机制提供给内核很强的灵活性。用户通过加载内核模块可以方便地给内核添加功能;用户也可以通过内核模块将内核不需要用的功能卸载。虽然驱动程序也可以静态编译进内核,但是由于外设众多,内核不可能包含所有设备的驱动。用户通过内核模块机制可以把需要用到的驱动程序动态地加入内核。在学习内核模块编程之前,本节将简单介绍内核模块。

现代操作系统的内核主要可以分为单内核和微内核两种。单内核操作系统内部可划分成层次或模块,但运行时是一个大的二进制映像,模块间的交互通过直接调用其他模块中的函数来实现。单内核操作系统模块之间的交互效率较高,但是灵活性较差。单内核操作系统常常需要将不常用的模块也编译进内核,给系统内存造成浪费。对于微内核操作系统,操作系统内核中仅包含资源控制与通信功能,其他功能都在内核外,由用户态下运行的服务进程实现,通过微内核提供的消息传递机制完成进程之间的通信。微内核操作系统有较高的灵活性,能够动态地给内核添加或删除功能,但是消息传递机制降低了系统的性能。

Linux 操作系统是一种单内核结构的操作系统,但是 Linux 在单内核操作系统的基础上引入了模块机制,克服了单内核的缺点,一定程度上解决了单内核功能的适应性、灵活性和可伸缩性问题。

Linux 提供的模块机制的完整叫法是动态可加载内核模块 (Loadable Kernel Module, LKM),一般简称为内核模块或模块。内核模块是一个目标文件,其代码可以在运行时链接到内核或从内核中解除链接。提供了内核模块机制的 Linux,可以在不必对内核重新编译链接的条件下,将内核模块动态地插入运行中的内核,成为内核的一部分,增加内核的功能。对于内核不再使用的内核模块,内核也可以在适当的时候将其解除链

接以减少内核对内存的占用。采用内核模块的主要优点如下。

(1) 模块化方法。因为任何模块都可以在运行时被链接或解除链接,因此,系统程序设计员必须提出明确定义的软件接口以访问由模块处理的数据结构。这使得开发新模块变得容易。

(2) 平台无关性。即使模块依赖于某些特殊的硬件特点,但它不依赖于某个固定的硬件平台。例如,符合 SCSI 标准的磁盘驱动程序模块,在 IBM 兼容 PC 与惠普的 Alpha 机上都能很好地工作。

(3) 节省内存使用。当需要模块功能时,把它链接到正在运行的内核中;否则,将该模块解除链接。这种机制也可以对用户透明,因为链接和解除链接可以由内核自动完成。

(4) 无性能损失。模块的目标代码一旦被链接到内核,其作用与静态链接的内核目标代码完全等价。因此,当模块的函数被调用时,无需显式的消息传递。

在 Linux 中,驱动程序必须在内核态下运行,因此驱动程序必须成为内核的一部分。Linux 并不把驱动程序都预先编译进内核,大部分驱动程序都以内核模块的形式加载进内核。

内核在管理内核模块时,主要有两个任务:一是符号表管理;二是维护内核模块的引用计数。内核模块被加载后,通常需要使用原内核导出的全局符号。内核模块需要知道这些符号在内核中的地址。所有的内核模块在加载后都在同一地址空间中,内核模块之间也会相互引用各自导出的符号,因此在内核模块之间会产生依赖性。如果一个模块需要用到另一个模块导出的符号,而另一模块没有被载入内核的话,该内核模块的加载就会出错。同样,对于一个内核模块,如果有其他内核模块引用它导出的符号,内核也不应该允许该内核模块被卸载。为了维护内核模块之间的依赖性,Linux 采用引用计数的方法管理内核间的依赖。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 时才允许被卸载。

内核提供了 `init_module()` 和 `delete_module()` 两个系统调用实现内核模块的动态安装和卸载。被安装的内核模块是在内核空间,能够访问并修改内核空间中的各种数据,为了防止一般用户加载恶意的内核模块,Linux 系统限制只有包含 `CAP_SYS_MODULE` 权限的用户才能对内核模块进行安装和卸载。Linux 系统也提供了 `insmod`、`rmmod` 和 `modprobe` 三个命令行工具供用户在命令行下对内核模块进行操作。`rmmod` 是卸载模块的操作;`insmod` 和 `modprobe` 都是安装内核模块的命令,它们的不同之处在于:`modprobe` 会检查内核模块之间的依赖性,在安装一个内核模块时,`modprobe` 会把该内核模块所依赖的其他内核模块一并载入内核。`modprobe` 通过 `insmod` 命令来实现模块的加载,`modprobe` 本身只是检查内核模块之间的依赖关系。`modprobe` 检查模块之间的依赖关系是通过 `depmod` 这个外部命令实现的。`depmod` 命令会查找系统中的所有内核模块,并将所有内核模块的依赖关系写入一个名为 `modules.dep` 的文件。这样,`modprobe` 就可以对

该文件中所存放的信息系统调用所产生的链接模块表进行比较,来获得模块间的依赖关系。这几个命令最终还是使用系统调用完成内核模块的操作。

7.2 内核模块编程基础

7.2.1 内核模块的基本结构

内核模块是在内核空间运行,内核模块编程就是在内核空间的编程。内核模块可以引用内核空间导出的全局符号,因此内核模块编程与内核的版本密切相关。下面以 Linux 2.6.x 为基础讨论内核模块编程的基础知识。现在以一个简单的 hello world 模块对如何编写内核模块加以说明。

```
1. #include<linux/init.h>
2. #include<linux/module.h>
3. MODULE_LICENSE("GPL");
4.
5. static int hello_init(void)
6. {
7.     printk(KERN_ALERT "Hello, world\n");
8.     return 0;
9. }
10.
11. static void hello_exit(void)
12. {
13.     printk(KERN_ALERT "Goodbye, cruel world\n");
14. }
15.
16. module_init(hello_init);
17. module_exit(hello_exit);
```

上面的代码是一个内核模块的典型结构,该模块在被载入内核时会向系统的日志中写入 Hello, world; 当该模块被卸载时,该内核模块也会向系统日志写入一条 Goodbye, cruel world 的消息。一个典型的内核模块应该包含如下几部分。

(1) 头文件声明。1、2 行是模块头文件的声明,头文件 module.h 和 init.h 是必须的。module.h 包含了大量加载模块需要的函数和符号的定义;init.h 中包含了模块初始化和清理函数的定义。如果模块在加载时允许用户传递参数,模块还应该包含 moduleparam.h 这个头文件。

(2) 模块许可声明。Linux 内核从 2.4.10 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,内核会显示一个 kernel tainted (内核被污染)的警告。从 linux/module.h 文件中可以看到,被内核接受的有意义的许可证有 GPL、GPL v2、GPL and additional rights、Dual BSD/GPL、Dual MPL/GPL、Dual MIT/GPL 和 Proprietary。

(3) 初始化、清理函数声明。在 Linux 2.6 内核中,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 16、17 行声明了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。需要注意的是,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。

一个基本的模块只要包含以上三个基本部分就能够正常地工作了。Linux 内核还提供了其他的一些函数与宏,以帮助开发者开发更加强大的内核模块,这些将在以下几节进行讲解。

7.2.2 编译和加载

Linux 内核从 2.4 到 2.6,内核模块的编译过程发生了比较大的变化。Linux 2.4 内核中,模块的编译只需要内核源代码的头文件;需要在包含 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 Linux 2.6 内核中,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;编译过程首先会到内核源码目录下,读取顶层的 makefile 文件,然后返回模块源代码所在的目录继续编译。

对于 Linux 2.6 内核,编译内核模块的 makefile 非常简单。对于本节前面展示的 hello world 例子,编译该模块的 makefile 只需要以下这一行就可以了。

```
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 应该写成如下的形式:

```
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,用户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 Linux 2.6 的内核源代码所在的目录在 /usr/src/linux-2.6,用户在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 还提供了另外一种形式,用户可以在 makefile 中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。对于本章所示的 hello world 例子,在 makefile 中指定内核源代码的方式为:

```
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD:= $(shell pwd)
default:
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
```

KERNELDIR 指定了内核的源代码目录。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,用户用超级用户的身份就可以将内核模块加载到内核中,内核提供了 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 和 modprobe 等实用程序。

1. insmod 命令

调用 insmod 命令,把需要载入的模块以目标代码的形式加载进内核中。在加载时,insmod 自动调用 init_module() 函数运行。只有超级用户才能使用这个命令,该命令的格式为:

```
#insmod [path] modulename
```

2. rmmod 命令

调用 rmmod 命令,将已经载入内核的模块从内核中卸载。该命令的格式为:

```
#rmmod modulename
```

3. lsmod 命令

显示已经载入的模块信息。该命令会显示被载入模块的模块名、大小和引用计数信息。该命令的格式为:

```
#lsmod
```

对于本节中的 hello 模块,超级用户可以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,用户可以用 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, cruel world.

7.2.3 内核符号表

内核符号表是用来存放所有模块可以访问的符号及相应地址的表。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。模块根据系统符号表从内核空间获取符号的地址,从而确保在内核空间中正确地运行。在 Linux 2.6 内核中,用户可以从 `/proc/kallsyms` 中以文本的方式读取内核符号表。

在 Linux 2.4 内核中,默认情况时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。在 Linux 2.6 内核中,默认情况下,这些符号是会被输出到内核符号表中的。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)
```

上面的任何一个宏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: `EXPORT_SYMBOL_GPL` 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。为了使模块便于维护,模块中不需要输出到内核空间且不需被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。

7.2.4 初始化与清理函数

在 Linux 2.6 内核中,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,使得它们不会在特定文件之外可见。声明中的 `__init` 告诉内核:该函数只是在初始化使用,模块加载者在模块加载后会丢弃这个初始化函数,内核可以把该函数占用的内存释放掉。`__init` 用于声明只在初始化使用的函数, `__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,释放该数据占用的内存。

在模块初始化过程中,程序需要密切注意初始化过程中可能发生的各种错误。初始化过程中的错误主要发生在资源的申请过程中,例如分配内存失败、打开设备失败等。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。初始化过

程发生的错误有不同的严重性。对于一些不严重的错误,模块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了。如果模块不能继续运行,它需要把已经申请的资源都及时释放。模块与内核处于同一空间,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。大部分的模块都需要设置一个清理函数,这个函数在模块被卸载时被调用。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。清理函数的函数定义为:

```
static void __exit exit_func (void)
{
    //清理代码
}
```

```
module_exit(exit_func);
```

清理函数没有返回值,因此它被声明为 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。该修饰符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,该函数除了在模块被卸载时调用外,不能够在任何其他的地方被调用。

7.2.5 模块参数

在用户使用外设的时候,可能希望在不同情况下让外设在不同的状态下工作。如果驱动程序是以模块的方式被加载,用户就需要与内核模块发生一些简单的交互。在 Linux 中,用户和内核模块能够有多种交互方式,例如通过 proc 文件系统、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,比较好的方法是采用内核模块参数的方法。在 Linux 中,内核允许用户在加载模块时将一些参数传递给模块,用户可以通过加载时的不同参数和模块发生一些简单的交互。参数的传递有多种方式,用户可以在 insmod、modprobe 命令中直接指定参数,例如:

```
#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是一样的,每次都在命令行中输入参数是比较繁琐的。用户可以在/etc/modprobe.conf 这个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` 是对数组型的参数进行定义。在这两个宏中, `name` 指明了参数的名称。 `name` 是参数在模块中的变量名, 用户在传递参数时采用 `name=value` 的形式传递参数给模块。其他三个参数的含义如下。

(1) `type` 指明参数的类型, 模块直接支持的参数类型如下。

- `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` 和 `ushort`。基本的整型值。以 `u` 开头的是无符号值。

以上的类型可以在 `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, 模块代码也允许用户使用自定义类型, 读者可以查看 `moduleparam.h` 了解更详细的信息。

(2) `perm` 为参数在 `sysfs` 文件系统中所对应的文件节点的属性。在 Linux 2.6 内核中, 系统使用了 `sysfs` 文件系统, `sysfs` 文件系统可以动态、实时, 有组织层次地反映当前系统中的硬件、驱动等情况。模块被加载后, 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, 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` 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` 为一个指针。该指针所指的变量保存了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, 该指针可以设置为 `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。设定好 `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。

有一点需要注意的是, 所有的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, 参数会使用预先设定好的默认值。

现在以本章开始的 `hello world` 模块(称为 `helloworld`)为例, 说明模块参数的使用方法。模块将使用两个参数: 一个是名为 `howmany` 的整型值; 一个称为 `whom` 的字符串。当模块加载时, 将欢迎 `whom` 不止一次, 而是 `howmany` 次。用户可以采用以下的命令行加载:

```
#insmod hellop howmany=10 whom="Mom"
```

当模块以这样的方式加载后,模块会显示 hello, Mom10 次。为了使用 howmany 和 whom 这两个参数,在模块的源代码中,应该插入如下的代码:

```
static char * whom="world";  
static int howmany=1;  
module_param(howmany, int, S_IRUGO);  
module_param(whom, charp, S_IRUGO);
```

module_param 定义了两个参数,一个是整型的 howmany,该参数在 sysfs 中的权限是只读的;另外一个参数是字符串类型的 whom,该参数在 sysfs 中也是只读的。

7.3 中断服务例程

外设的操作时间一般较长,当驱动程序向外设发出操作指令后,驱动程序需要等待外设完成操作。驱动程序可以采用两种方式等待外设完成操作。一种是查询方式,驱动程序在提交命令之后就开始查询设备的状态寄存器,当状态寄存器表明操作完成时,驱动程序才继续后续的处理。另一种方式是利用中断,驱动程序提交命令之后立即进入休眠状态。设备结束操作之后,产生中断信号,操作系统则根据设备的中断信号负责唤醒驱动程序。比较这两种方式,查询方式白白浪费了大量的处理器时间,中断方式是多任务操作系统中有效利用处理器的方式。

在 Linux 中,多个设备可以共享同一个中断。对于一个中断,内核维护了一个中断服务例程(Interrupt Service Routine,ISR)的链表。当中断到来时,该中断的每一个中断服务例程都会被依次调用。内核允许设备驱动将自己的中断服务例程插入相应中断的队列。本节将介绍在设备驱动中如何注册和释放中断服务例程。

7.3.1 概述

外部设备通过中断线将产生的中断通知处理器,在外设采用中断模式开始工作前,外设必须先申请一个中断号。由于受中断向量数量较少的硬件限制,不能给每一个外设分配一根单独的中断号。为了解决这个问题,一些外设必须共享同一个中断号。共享同一个中断号的多个外设发出的中断信号是相同的。在 i386 中,内核在中断向量表中注册中断处理程序(interrupt handler),一个中断信号对应一个中断处理程序。当多个外设共享一个中断线时,它们便共享了同一个中断处理程序。当共享中断号的任何一个外设产生中断,处理器会转跳到同一个中断处理程序。为了使每个设备能够有独立的中断处理程序,Linux 引入了中断服务例程的概念。中断服务例程与中断处理程序是两个不同的

概念。

(1) 中断处理程序。中断处理程序相当于某个中断的总处理程序,它并不服务于任何特定的设备,它是为特定的中断向量服务的。当发生中断后,中断处理程序的主要工作是做一些通用的现场保护,并把该中断包含的所有 ISR 依次运行一遍。中断向量与中断处理程序的映射在中断向量表中注册好后,在内核运行的过程中一般是不会被修改的。

(2) 中断服务例程。ISR 是为某一个具体设备服务的程序。虽然多个外设能够共享同一个 ISR,但是在一般情况下,每个外设都有各自的 ISR 与之对应。共享同一个中断的外设所对应的 ISR 被放置在同一个链表中。当一个中断发生时,中断处理程序必须把属于该中断的所有 ISR 都运行一遍。

外设、中断处理程序和中断服务例程三者的对应关系,如图 7.1 所示。当中断发生时,被执行的应该是发生中断的设备所对应的 ISR。由于中断处理程序并不知道中断是由哪一个设备发出的,因此属于该中断的 ISR 会被全部运行一次。在这种情况下,为了使整个中断处理过程的时间尽量缩短,每一个 ISR 被运行时,ISR 先要检测该中断是不是由 ISR 所对应的设备发出的。如果不是,ISR 应该立即退出。

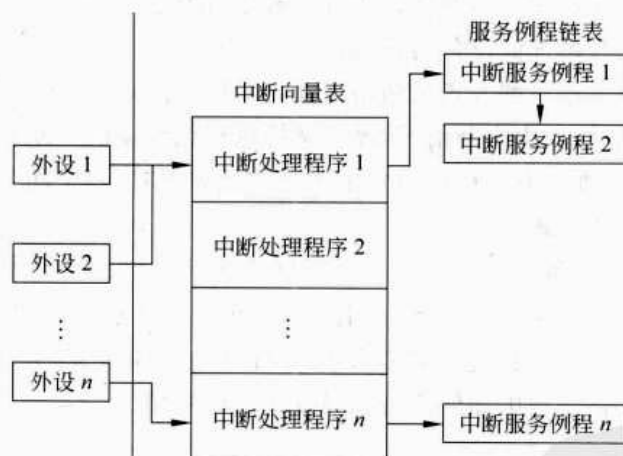


图 7.1 外设、中断处理程序和中断服务例程的关系

7.3.2 检测中断号

驱动程序初始化的过程中,如果需要添加 ISR,驱动程序需要知道设备使用哪一个中断通道号(IRQ)。对于 IRQ 的确定,主要有以下几种方法。

(1) 固定通道号。对于一些特定的设备,例如软盘驱动器,其 IRQ 固定为 6。

(2) 设备指定通道号。一些设备能够自己指定它要使用的 IRQ,驱动程序只需要从设备的一个 I/O 端口或者 PCI 配置空间中读取一个状态字节就能知道设备使用哪一

个 IRQ。

(3) 自动检测通道号。驱动程序能够通过程序自动探测一些设备的 IRQ。

(4) 用户指定通道号。对于驱动程序不能自动探测 IRQ 的设备,驱动程序可以要求用户指定 IRQ,用户将 IRQ 作为参数传递给驱动程序。

采用固定通道号的设备最为简单,这些设备很少会改变默认的 IRQ。在这种情况下,驱动程序在默认情况下采用默认值即可。驱动程序可以留出接口,当设备改变 IRQ 时,用户可以将新的 IRQ 作为参数传递给驱动程序。

对于能够自动指定 IRQ 的设备,驱动程序探测 IRQ 就变得非常简单。大部分的 PCI 设备都支持这样的特性。驱动程序可以方便地获得设备配置头信息中的 IRQ 信息。

对于一些老式的设备,如 ISA 设备,驱动程序需要自动检测 IRQ。对于自动检测 IRQ,总体的一个思路是:驱动程序让设备产生中断,驱动程序观测哪个 IRQ 被激活。驱动程序除了可以自己检测 IRQ,也可以使用 Linux 自身提供的一些自动检测 IRQ 的函数。

自动检测 IRQ

Linux 内核自身提供的检测机制也是比较容易使用的,内核在 `<linux/interrupt.h>` 中声明了两个用于自动检测 IRQ 的函数:

```
unsigned long probe_irq_on(void);  
int probe_irq_off(unsigned long);
```

这两个函数在探测 IRQ 时需要一起使用。probe_irq_on 将返回一个未使用的中断位掩码。驱动程序必须保留返回的位掩码,并且在后面传递给 probe_irq_off。在调用这个函数之后,驱动程序应该至少让设备产生一次中断。当设备产生一个中断后,驱动程序应该调用 probe_irq_off 函数,把之前 probe_irq_on 的返回值作为参数传递给该函数。probe_irq_off 将返回在 probe_on 之后外设产生的中断号。如果没有中断发生,probe_irq_off 返回 0。如果多于一个中断发生,probe_irq_off 返回一个负值。用户在 probe_irq_on 之后应该将设备的中断启用,在调用 probe_irq_off 之后应该禁止设备的中断。这两个函数使用的一般结构如下。

- (1) 调用 probe_irq_on() 函数,保存返回值。
- (2) 打开设备的中断功能,并让设备发出一个中断。
- (3) 等待一段时间让中断发生。
- (4) 将 probe_irq_on 的返回值作为参数传递给 probe_irq_off,获得设备使用的 IRQ。
- (5) 关闭设备的中断功能。应该在用户真正使用设备时才把设备的中断功能再次打开。

以下的代码是用这两个函数检测 IRQ 的一个例子:

```

int count=0;                //尝试探测的次数
do
{
    unsigned long mask;
    mask=probe_irq_on();

    //这里为打开设备中和让设备发生中断的代码

    udelay(5);              //延时等待设备发生中断
    irq=probe_irq_off(mask);

    if(irq==0) {             //返回值为 0,表明没有中断发生
        printk(KERN_INFO "no irq reported by probe\n");
        irq=-1;
    }

    //如果返回值为负值,继续尝试,直到达到最大尝试次数
} while(irq<0 && count++<5);

```

因为设备的速度慢于处理器,所以代码中需要用 `udelay` 等待一段时间让设备发生中断。对于不同的设备,需要等待的时间是不一样的。处理器有可能需要等待比较长的时间,因此 IRQ 一般只探测一次。IRQ 的探测可以在用户请求打开设备时探测,也可以在用户加载驱动程序时就开始探测。考虑到探测 IRQ 所耗费的时间,在用户打开设备时才探测 IRQ 是比较好的方法。

驱动程序也可以不采用内核提供的自动检测机制来检测 IRQ,这需要驱动程序在检测前注册 ISR,其实现过程在下一节进行介绍。

7.3.3 安装 ISR

如果设备需要用到中断,驱动程序要为设备安装相应的 ISR。当驱动程序获得设备使用的 IRQ 后,驱动程序就能利用内核提供的函数,安装或卸载 ISR 了。安装和卸载 ISR 的两个函数是在 `<linux/interrupt.h>` 中声明的。

```

int request_irq(unsigned int irq,
                irqreturn_t (* handler)(int, void*, struct pt_regs*),
                unsigned long flags,
                const char * dev_name,
                void * dev_id);

void free_irq(unsigned int irq, void * dev_id);

```

request_irq 用于安装 ISR,而 free_irq 用于卸载一个 ISR。request_irq 的几个参数的含义为:

- unsigned int irq。请求的中断号。
- irqreturn_t(* handler)。指向 ISR 的函数指针。
- unsigned long flags。一个与中断管理相关的选项。
- const char * dev_name。设备名,这个设备名会在 /proc/interrupts 中显示。
- void * dev_id。如果多个设备共享 IRQ,dev_id 是作为 ISR 的一个标识。该标识会在释放 ISR 时被用到。如果 IRQ 没有被共享,dev_id 可以设置为 NULL。

flags 是安装 ISR 时的一些选项,flags 中可以设置的位如下。

- SA_INTERRUPT。表明该 ISR 必须以禁止中断的方式执行。
- SA_SHIRQ。表明多个设备共享同一个 IRQ。
- SA_SAMPLE_RANDOM。表明设备的中断产生是随机的,可以帮助一些程序产生比较安全的密钥。如果该设备的中断产生是可预测的,这个标志不应该被设置。

驱动程序调用 request_irq 后,可能会获得不同的返回值。不同的返回值指示了安装 ISR 遇到的不同情况。

(1) 返回 0 表示 ISR 安装成功。

(2) 返回-INVAL 表示 irq>NR_IRQS 或者 handler==NULL,NR_IRQS 是在 <asm/irq.h>中定义的一个独立于平台的宏,指明系统中可用的中断线程数量。

(3) 返回-EBUSY 表明中断已经被占用并且不允许被共享。

与探测 IRQ 类似,安装 ISR 最好的时机并不是在驱动程序初始化的时候。因为系统中可供使用的 IRQ 是有限的,如果一个设备在还没有使用的时候就把 ISR 安装上,那么其他不允许共享该 IRQ 的设备就完全没有运行的机会了。所以安装 ISR 应该在设备第一次被打开的时候。

如果为了最大效率地使用 IRQ,驱动程序还应该在用户不使用该设备时,自动将 ISR 卸载下来,让其他设备有机会独享 IRQ。这样要求驱动程序在每次设备被打开、关闭时要对设备的使用次数作记录,当发现设备不再有用户使用时,驱动程序就将 ISR 卸载下来。free_irq 是用于卸载 ISR 的函数,其第一个参数指明了 IRQ;第二个参数指明了 ISR 的标识,这个标识在多设备共享 IRQ 时需要使用。

手动检测 IRQ

在了解了如何注册、卸载 ISR 后,就可以自动为设备检测 IRQ 了。实现原理和内核提供的检测 IRQ 的方法类似,主要步骤如下。

(1) 列举所有设备可能使用到的 IRQ。

(2) 为每一个 IRQ 都注册一个 ISR。

- (3) 让设备发出一个中断。
- (4) 设备发出中断后,监测哪些 ISR 被激活。
- (5) 如果只有一个 ISR 被激活,则找到设备使用的 IRQ,记录 IRQ 并转跳到第(9)步。
- (6) 如果没有任何一个 ISR 被激活,说明没有找到设备使用的 IRQ,转跳到第(8)步。
- (7) 如果多个 ISR 被激活,表明检测失败。
- (8) 增加失败次数,如果到达最大尝试次数,转跳到第(9)步;否则转跳到第(3)步。
- (9) 将安装的 ISR 都卸载。

以下代码是该步骤在单处理器上的一个实现(在多核和 SMP 系统中还要考虑同步),该代码通过测试所有“可能的”中断并且查看发生的事情来探测中断。trials 数组列出要尝试的中断,以 0 作为结尾标志;tried 数组用来跟踪哪些 IRQ 上的 ISR 被注册成功。

```
int trials[]={3, 5, 7, 9, 0};
int tried[]={0, 0, 0, 0, 0};
int i, count=0;

//注册 ISR
for (i=0; trials[i]; i++)
    tried[i]=request_irq(trials[i], probing,
                        SA_INTERRUPT, "probe", NULL);

do
{
    irq=0;                //表示没有找到 IRQ
                        //让设备发出中断
    udelay(5);            //等待设备发出中断

    if (irq==0) {          //没有找到 IRQ
        printk(KERN_INFO "no irq reported by probe\n");
    }
    //irq<0 表明多个 ISR 被激活,监测失败.在没有到达最大失败次数前继续尝试检测
} while (irq<=0 && count++<5);

//卸载 ISR
for (i=0; trials[i]; i++)
    if (tried[i]==0)
        free_irq(trials[i], NULL);
```

```

if (irq<0)
    printk("probe failed %i times, giving up\n", count);

//ISR 代码
irqreturn_t probing(int tirq, void* dev_id, struct pt_regs* regs)
{
    if(irq==0) irq=tirq;           //第一次被激活,将 irq 标记为触发的 IRQ
    if(irq!=tirq) irq=-tirq;       //被多次激活时,将 irq 标记成负值
    return IRQ_HANDLED;
}

```

驱动程序可能事先不知道要尝试测试哪些 IRQ。在这种情况下,驱动程序需要探测所有空闲的 IRQ。驱动程序需要从 IRQ 0 探测到 IRQ NR_IRQS-1。

7.4 上半部和下半部

中断服务例程一般都是在中断请求关闭的条件下执行,这样能够避免因中断嵌套带来的中断控制的复杂化。当中断发生时,中断服务例程被串行化执行。如果每一个中断服务例程都占用比较长的处理器时间,那么在中断服务例程执行的过程中,很可能发生其他的中断。由于中断请求被关闭,这些中断就不能及时得到处理器的响应,也使得内核的响应速度大大降低。为了提高内核处理中断的吞吐量,提高内核的响应速度,每一个中断服务例程都应该尽快地释放处理器,把能够推后的工作尽量往后推。例如,假设一个数据块已经传到了网络设备,当中断控制器接收到这个中断请求信号时,驱动程序应该只是简单地标志数据已经到来,然后让处理器恢复到它以前运行的状态,其余的数据传输将稍后进行。因此内核把中断服务例程分成两个部分:上半部(top half)和下半部(bottom half)。上半部分会立即被内核执行,而下半部分会被推迟执行。下半部的执行并不需要指明一个确切时间,只要把这些任务推迟,让它们在系统不太繁忙并且中断恢复后执行就可以了。

在响应中断时,并不存在严格明确的规定来说明到底什么任务应该在哪个部分中完成。驱动程序开发者可以根据自己的经验判断哪些应该被放入下半部。不同的分割方法可能会导致性能上的差异。开发者要尽可能地减少上半部的执行时间。对于在上半部和下半部之间划分工作,尽管不存在某种严格的规则,但还是有一些提示可供借鉴。

- (1) 如果一个任务对时间非常敏感,将其放在上半部中执行。
- (2) 如果一个任务和硬件相关,将其放在上半部中执行。
- (3) 如果一个任务要保证不被其他中断(特别是相同的中断)打断,将其放在上半部

中执行。

(4) 其他所有任务,考虑放置在下半部执行。

在 Linux2.6 中,上半部是在中断服务例程中被执行的,下半部有多种实现方法。本节将对这些机制进行讲解,并分析适合各种机制使用的场景。

7.4.1 下半部的实现机制

在 Linux 不同版本的内核中有着不同的下半部机制,有些机制在最新的内核中已经被废弃了。为了让读者对下半部实现机制的演变有比较完整的了解,本小节将把曾经出现过的一些常用的下半部的实现机制做一个简单的介绍。

Linux 内核最早提供的任务延迟执行机制称 BH。在内核中维护着一个 BH 的函数指针数组,采用数组索引来访问,最多能有 32 个不同的下半部分处理函数。BH 机制的主要缺陷如下。

(1) 下半部的个数限制在 32 个,而且每个 BH 上只能挂接一个函数,在现在的内核中,32 个 BH 是不够用的。

(2) 每个 BH 在全局范围内进行同步,不允许任何两个 BH 同时执行。

由于 BH 的缺陷,使得 Linux 在 2.5 版本时将 BH 机制丢弃了。

下一个出现的下半部机制是任务队列(task queue),它是内核各种任务延迟执行的方法。任务队列提供了对任务队列中任务排队及调度执行的通用机制。任务队列是一种通用的机制,可以在中断处理之外使用。所谓任务队列就是以双向列表形式连接起来的任务链表,每一个链表成员都描述一个可执行的内核任务。内核任务就是一个或几个普通的内核函数。当驱动程序或内核的其他部分要将任务排队进行处理时,可将任务添加到相应的任务队列。任务队列是基于 BH 的,要运行任务队列只需要通知内核进行 BH 处理即可。任务队列的缺点在于不够灵活,没法替代整个 BH 接口。任务队列对于一些性能要求较高的子系统,像网络部分,也不能胜任。

在 Linux 2.3 版本中,内核引入了软中断(softirq)和 tasklet 两种下半部机制。软中断是一组静态定义的下半部接口,可以在所有处理器上同时执行——即使两个类型相同也可以。软中断保留给对时间要求严格的下半部分使用,如网络和 SCSI。每个处理器有一个称为 ksoftirqd 的辅助处理软中断的内核线程,例如,进行大流量网络通信期间,软中断被触发的频率相当高,内核便在特殊时机唤醒 ksoftirqd。

tasklet 是基于软中断来实现的,但比软中断接口简单,同步要求较低;软中断保留给执行频率及时间要求高的下半部使用(如网络和 SCSI),多数情况下都可使用 tasklet。引入 tasklet 是因为 BH 是全局串行处理的,不能适应 SMP 环境,充分提高多个 CPU 的利用率。不同的 tasklet 可以同时运行于不同的 CPU 上,相同的 tasklet 不会同时在不同的 CPU 上运行。在这个条件下,tasklet 就不需要是可重入的。tasklet 能被动态创建,使用

方便且效率也高,较好的支持 SMP 结构,因此,在新版 Linux 中,tasklet 是推荐的异步任务延迟执行机制。

工作队列(work queue)接口是在 2.5 的开发过程中引入的,用于取代任务队列接口(用于调度内核任务)。工作队列是另外一种任务延迟执行机制,它与其他机制的工作原理都不相同,它把一个任务延迟,并交给内核线程去执行,而且该任务总是会在进程上下文中执行。这样,通过工作队列执行的代码能占尽进程上下文的所有优势,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,可选择工作队列,否则就要使用 tasklet 或 softirq。

tasklet 和工作队列是常用的实现下半部的机制,接下来的几个小节将介绍这些机制的使用方法。

7.4.2 软中断

软中断是只能静态定义的下半部机制。同一种软中断可以同时运行在多个 CPU 上。因此软中断执行的函数必须是要可重入的,当软中断在访问临界区时需要使用同步机制,例如自旋锁(spin lock)。由于软中断处理的时间快,在一些对时间要求非常严格的任务中,需要使用软中断。在目前的内核中,只有网络和 SCSI 两个子系统直接使用软中断。在内核中,内核时钟和 tasklet 也是建立在软中断的基础上的。对于一般的任务,采用 tasklet 和工作队列就能够胜任。

软中断在内核中用 softirq_action 表示,该结构表示如下:

```
struct softirq_action{
    void(* action)(struct softirq_action*);    //软中断处理函数
    void* data;                                //传给软中断处理函数的参数
};
```

在<kernel/softirq.c>中,定义了一个长度为 32 的软中断的数组:

```
static struct softirq_action softirq_vec[32];
```

数组中的每一项对应一个软中断。系统中最多可以有 32 个软中断,内核目前只预定义了 6 个软中断,它们的作用如表 7.1 所示。

软中断的类型在<linux/interrupt.h>中用一个枚举类型定义,内核在访问软中断时,采用这些枚举值作为数组的下标进行访问。

当软中断被执行之前,软中断首先要被标记成可执行。内核用一个 32 位的整型用户标记软中断的状态,一位对应一个软中断。如果软中断被标记成可执行,当内核开始执行软中断时,该软中断就会被执行。内核在三个点会进行软中断的执行:从中断返回时、ksoftirqd 被执行时及内核代码显示执行软中断。

表 7.1 软中断类型

软中断类型	优先级(数组中的下标)	说 明
HI_SOFTIRQ	0	处理高优先级的 tasklet
TIMER_SOFTIRQ	1	时钟的下半部
NET_TX_SOFTIRQ	2	用于发送网络数据包
NET_RX_SOFTIRQ	3	用于接收网络数据包
SCSI_SOFTIRQ	4	用户处理 SCSI 命令
TASKLET_SOFTIRQ	5	处理普通 tasklets

在大部分情况下,开发者没有必要使用软中断。如果需要使用软中断,开发者需要定义新的软中断、注册软中断处理函数、触发软中断。

软中断不能动态的创建。要使用软中断,开发者只能静态声明软中断。如果需要添加一个新的软中断,开发者可以在<linux/interrupt.h>文件中软中断的枚举类型中添加一项。内核采用该枚举值作为访问软中断的序号。在软中断的执行过程中,软中断是从枚举值为0的软中断开始执行。因此,枚举值越小的软中断具有越高的优先级。在添加软中断时,需要考虑不同枚举值可能给软中断执行带来的影响。

在运行时,程序可以通过 open_softirq()函数注册软中断的处理函数。open_softirq接受三个参数,软中断的序号、软中断处理函数及传递给软中断处理函数的参数。例如,在网络系统中,注册软中断处理函数的方法类似于:

```
open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
```

当软中断处理程序被注册后,该软中断就可以被触发运行了。触发软中断的函数为 raise_softirq(),该函数接受要被激活的软中断序号作为参数。当调用 raise_softirq()函数,内核调用 do_softirq()函数后,软中断处理函数就会被运行。例如,当网络系统调用后,该软中断处理函数 net_tx_action 就会在内核下次执行软中断时被执行。

```
raise_softirq(NET_TX_SOFTIRQ);
```

7.4.3 tasklet 机制

tasklet 是一种软中断机制,是下半部处理的首选机制;tasklet 是在中断的上下文中执行,所有的 tasklet 代码必须是原子的。tasklet 是不能睡眠的,所以不能在 tasklet 中使用信号量或者其他可能产生阻塞的函数。tasklet 是一个特殊的函数,会被调度来运行。每一个 tasklet 只会被运行一次,不同的 tasklet 可以与 SMP 系统上的其他 tasklet 并行

运行。如果驱动程序使用多个 tasklet, 由于多个 tasklet 有可能同时执行, tasklet 之间需要采用一些同步机制进行保护。

tasklet 所对应的数据结构为 tasklet_struct, 其定义为:

```
struct tasklet_struct {
    struct tasklet_struct * next;           //指向下一个 tasklet
    unsigned long state;                   //tasklet 的状态
    atomic_t count;                        //引用计数, 通常用 1 表示禁止
    void (* func)(unsigned long);          //处理函数指针
    unsigned long data;                    //处理函数的参数
};
```

tasklet 的状态中, 定义了两个位的含义: TASKLET_STATE_SCHED 表示该 tasklet 已被调度, 正准备投入运行; TASKLET_STATE_RUN 表示该 tasklet 正在运行。count 是 tasklet 的引用计数器, 如果它不为 0, 则 tasklet 被禁止; 仅当 count 为 0 时, tasklet 才能被激活并执行。func 指定了 tasklet 在执行时所运行的函数, 该函数可以把 data 作为参数传递进去。

用户可以静态地创建、也可以动态地创建 tasklet。如果静态地创建一个 tasklet, 可以使用下面的两个宏:

```
DECLARE_TASKLET(name, function, data)
DECLARE_TASKLET_DISABLED(name, function, data)
```

在这两个宏中, name 是静态定义的 tasklet 变量名; function 是 tasklet 执行时所运行的函数; data 是一个 tasklet 执行时传递给 function 的参数。两个宏的不同之处在于 DECLARE_TASKLET_DISABLED 声明出来的 tasklet 的引用计数被置为 0, 表明这个 tasklet 被禁止, 只有激活后该 tasklet 才能被调度执行。

tasklet_schedule() 函数用于把一个 tasklet 进行调度, 使得该 tasklet 能够在适当的时候被执行。在 tasklet 被调度后, 只要有机会它就会尽可能早的运行。如果一个 tasklet 在运行之前又再次被调度, 该 tasklet 还是只会运行一次。内核还提供了多个用于控制 tasklet 的函数。

- tasklet_hi_schedule()。调度一个 tasklet, 但是该 tasklet 比用 tasklet_schedule() 函数调度的 tasklet 具有更高的优先级。
- tasklet_disable()。禁止 tasklet, 如果该 tasklet 当前正在执行, 函数会等到它执行完毕再返回。
- tasklet_disable_nosync()。禁止 tasklet, 如果该 tasklet 已经被运行, 该函数不会等待它运行结束。
- tasklet_enable()。激活 tasklet。

- `tasklet_kill()`。从队列中删除一个 `tasklet`。如果该 `tasklet` 正在执行,函数也会等待该 `tasklet` 执行完毕后,才将它删除。

7.4.4 工作队列

工作队列和 `tasklet` 这两种下半部机制的主要区别在于: `tasklet` 在软中断上下文中运行,所有的 `tasklet` 代码必须是原子的;工作队列函数在一个内核线程上下文运行,并且可以在延迟一段确定的时间后才执行;它有更多的灵活性,工作队列函数可以使用信号量等能够睡眠的函数。

驱动程序在使用工作队列时的主要步骤如下。

- (1) 如果驱动程序不使用默认的工作队列,驱动程序可以创建一个新的工作队列。
- (2) 驱动程序需要将工作延迟时,动态或静态地创建工作队列任务。
- (3) 将工作队列任务插入工作队列。

1. 创建工作队列

当驱动程序需要使用工作队列时,并不一定需要创建一个新的工作队列。驱动程序可以使用内核提供的默认工作队列。由于这个工作队列由很多驱动程序共享,任务可能会需要比较长的时间才能开始执行。为了解决这一问题,工作函数中的延迟应该保持最小。默认队列对所有驱动程序来说都是可用的,但是只有经过 GPL 许可的驱动程序才可以创建新的工作队列。

每一个工作队列的任务都是运行在一个内核线程的上下文,因此每个工作队列都需要有一个专门的工作线程。工作队列任务可以在该线程的上下文中运行。驱动程序可以创建并使用它们自己的工作队列,工作队列为 `struct workqueue_struct` 类型,在 `<linux/workqueue.h>` 中定义。一个工作队列必须明确地在使用前创建,使用下列两个函数之一:

```
struct workqueue_struct * create_workqueue(const char * name);  
struct workqueue_struct * create_singlethread_workqueue(const char * name);
```

这两个函数的区别在于用 `create_workqueue` 创建的工作队列在每一个处理器上都会有一个工作线程,而用 `create_singlethread_workqueue` 创建的工作队列只含有一个工作线程。

2. 创建工作队列任务

当工作队列创建好后,驱动程序可以创建工作队列任务并将它放入工作队列中运行。工作队列任务需要封装为一个叫做 `work_struct` 的结构体。工作队列任务可以静态地创

建,也可以动态地创建。

静态初始化一个工作队列任务时可以用宏:

```
DECLARE_WORK(name, void(* function)(void*), void* data);
```

在这里 name 是工作队列任务的变量名, function 是当任务被调度时调用的函数, data 是传递给该函数的参数。

动态初始化一个工作队列时用到的宏为:

```
INIT_WORK(struct work_struct* work, void(* function)(void*), void* data);  
PREPARE_WORK(struct work_struct* work, void(* function)(void*), void* data);
```

这两个宏在运行时创建一个新的工作队列任务。PREPARE_WORK 与 INIT_WORK 的区别在于: PREPARE_WORK 仅仅设置了 work_struct 中的 func 和 data 成员; INIT_WORK 还初始化了用于链接到工作队列的链表指针,并对用于延迟执行的定时器进行了初始化。

3. 将任务插入工作队列任务

当工作任务创建好后,用以下的函数插入工作队列:

```
int queue_work(struct workqueue_struct* queue, struct work_struct* work);  
int queue_delayed_work(struct workqueue_struct* queue, struct work_struct*  
work, unsigned long delay);
```

其中 queue 是指向工作队列的指针, work 是指向工作任务的指针, delay 指明了工作任务运行的延迟时间。

用 queue_work() 插入的工作任务是在稍后一个不确定的时候后运行; 用 queue_delayed_work() 插入工作任务时需要指定延迟时间, 该工作任务保证至少在经过给定的时间后才会被内核执行。

如果驱动程序使用的是默认的工作队列, 可以使用下面的两个函数:

```
int schedule_work(struct work_struct* work);  
int schedule_delayed_work(struct work_struct* work, unsigned long delay);
```

因为使用默认工作队列, 这两个函数不需要指定工作队列指针, 其他参数用法与 queue_work 系列函数一致。

4. 从工作队列中删除任务

如果想取消一个工作任务的执行, 可以采用下面函数:

```
int cancel_delayed_work(struct work_struct * work);
```

如果当一个取消操作的调用返回时,任务正在执行中,那么这个任务将继续执行下去,但不会再加入到队列中。清空工作队列中的所有任务可以使用:

```
void flush_workqueue(struct workqueue_struct * queue);
```

当用完一个工作队列,如果驱动程序觉得不再需要使用该队列,可以使用以下函数销毁工作队列:

```
void destroy_workqueue(struct workqueue_struct * queue);
```

最后,当模块被卸载时应该去调用一个 `flush_scheduled_work()` 函数,这个函数会使等待队列中所有的任务都被执行。

7.4.5 选择合适的下半部机制

在 Linux2.6 中,可以使用的下半部机制有软中断、tasklet 和工作队列。tasklet 基于软中断实现,所以两者很相近。工作队列机制与它们完全不同,它靠内核线程实现。驱动程序开发者根据需求选择不同的下半部机制。

从设计的角度考虑,软中断提供的执行序列化的保障最少。这就要求软中断处理函数必须格外小心地采取一些步骤确保共享数据的安全,两个甚至更多相同类别的软中断有可能在不同的处理器上同时执行。对于时间要求严格和执行频率很高的应用来说,它执行的也最快。由于两个同种类型的 tasklet 不能同时执行,如果选择 tasklet,开发者需要考虑的同步比较少。如果下半部中可能用到睡眠等阻塞操作,就只能选择采用工作队列。因为工作队列牵扯到内核线程甚至是上下文切换,因此它造成的开销最大。如果对效率要求比较严格,不适合采用工作队列。表 7.2 对这三种机制进行了比较。

表 7.2 三种下半部机制对比

下半部	上下文	同步保障
软中断	中断	没有
tasklet	中断	同类型不能同时执行
工作队列	内核线程	没有(和线程一样被调度)

在一般的驱动编写中,如果下半部有休眠的需要,工作队列是唯一的选择;否则最好用 tasklet;要是性能的要求较高,就应该使用软中断。

7.5 本章小结

Linux 操作系统是一种单内核结构的操作系统, Linux 在此结构上引入了内核模块机制。内核模块机制扩展了单内核操作系统的功能, 解决了单内核操作系统的适应性、灵活性和可伸缩性问题。在 Linux 系统中, 大部分驱动程序都以内核模块的形式存在于系统中。内核可以在需要时动态地载入驱动程序, 也可以在驱动程序不再使用时将驱动程序从内核中移出。为了实现内核模块的动态加载特性, 内核需要对内核模块的符号表与内核模块引用计数进行管理。这也需要驱动程序开发者在驱动程序的编写时进行配合。

一个基本的内核模块主要包含头文件声明、许可证声明和初始化、清理函数声明三个部分。Linux 允许内核模块在运行时引用内核导出的符号, 也允许内核模块使用其他模块导出的符号。内核模块在导出符号供其他模块使用时, 需要使用内核提供的宏将需要使用的符号导出。如果模块允许用户传递参数给它, 这些参数的名称与类型也需要在源代码中事先指定。内核模块在编写好后, 可以采用多种 makefile 对内核模块进行编译。开发者可以根据需要灵活选择。

在驱动程序开发中, 设备与驱动程序的交互通常采用中断的方式。由于中断数量有限, 多个设备常需要共享同一个中断。在 Linux 中, 内核对每一个中断维护了一个中断服务例程链表。驱动程序在探测到设备所使用的中断号后, 需要将相应的中断服务例程插入相应的中断服务例程链表中。当中断服务例程被插入后, 驱动程序便可以开始响应中断了。为了提高中断的利用率, 一般在设备真正被用户打开时, 驱动才将进行中断服务例程的安装; 在用户关闭设备后, 驱动可以将中断服务例程卸载。

为了加快内核响应中断的速度, 驱动程序开发者可以将中断服务例程分成两个部分: 上半部和下半部。上半部会被内核立即执行掉, 而下半部分会被推迟执行。Linux 提供的下半部的实现机制主要有软中断、tasklet 和工作队列。软中断只能被静态定义, 执行速度快, 一般用于对时间要求非常严格的场合; tasklet 是下半部处理的首选机制, 能够动态被创建, 但它在执行过程中不能睡眠; 工作队列在一个内核线程的上下文中执行, 在工作队列中的任务是能够睡眠的, 并且工作队列中的任务还可以在指定延时之后才被执行。

思 考 题

1. 用户态和内核态进行参数传递有哪些方法?
2. 各种下半部机制在实现上有何联系?
3. 如果在中断服务例程中需要使用信号量, 采用什么样的下半部机制比较好?

进一步阅读

本章主要介绍了进行驱动程序编程的基础知识。这些机制在 Linux 中的实现原理可参见以下资料。

1. 内核模块实现机制：(美)博韦(Bovet, D. P.), 西斯特(Cesati, M.) 著.《深入理解 Linux 内核(第三版)》附录二
2. 软中断与 Bottom Half 实现机制：毛德操, 胡希明.《Linux 内核源代码情景分析》3.5 节
3. tasklet 与工作队列实现机制：《深入理解 Linux 内核(第三版)》4.7 节、4.8 节



第8章 字符设备和块设备驱动程序

外部设备是计算机系统的重要组成部分,设备管理是操作系统的基本功能之一。Linux 操作系统的一个重要特点就是将所有的设备都当作文件进行处理。这类文件称为设备文件,用户可以对这些文件进行操作,就像操作普通文件一样,大大方便了系统对设备的管理。

Linux 系统的设备一般分为三类:字符设备、块设备和网络设备。本章主要介绍字符设备和块设备驱动程序的设计方法。

8.1 字符设备和块设备驱动程序设计基础

Linux 系统中字符设备与块设备是硬件设备的两种标准类型。字符设备是指设备发送和接收数据是以字符为单位进行的,例如键盘、串口、虚拟控制台。块设备是指设备发送和接收数据以整个数据块(如 512B)为单位进行,例如 IDE 硬盘、SCSI 硬盘及光驱等。

操作系统对字符设备和块设备的处理方式是不同的。一般来说,块设备比字符设备复杂得多,本书后面会有相关内容进行讲解。

设备驱动程序是操作系统最基本的组成部分,是 Linux 内核的一部分;在 Linux 内核源码中约占三分之二。前面章节已经提到过,Linux 内核采用了可加载内核模块(Loadable Kernel Modules, LKM)设计,一般情况下内核是支持动态加载模块的,可以将驱动程序设计为一个模块,在需要时动态加载到内核中。

Linux 操作系统将硬件设备看作是一个特殊的文件来操作,这些文件称作设备文件,一般位于/dev 目录下。可以通过 mknod 命令来创建一个设备文件,利用标准的文件操作函数对设备文件进行打开、关闭、读取和写入等操作

在/dev 目录下输入命令 #ls-l,可以看到如下一些信息:

```
brw-rw---- 1 root disk 8, 1 2007-01-14 20:26 sda1
brw-rw---- 1 root disk 8, 2 2006-06-14 20:26 sda2
brw-rw---- 1 root disk 8, 3 2007-03-21 12:30 sda3
crw-rw-rw- 1 root root 5, 0 2007-04-08 13:43 tty
```

上面这些文件都是设备文件,每一个文件代表一个设备。每一行的第一个字母代表设备类型,b 表示是块设备文件,c 表示是字符设备文件。日期前的两个数字分别代表设备的主设备号及从设备号。

8.1.1 驱动层次结构

Linux 设备驱动程序是内核的一部分,图 8.1 是设备驱动程序在系统中的层次结构图。用户通过操作设备文件可以对硬件设备进行访问。Linux 设备驱动程序包括设备服务子程序和中断处理程序两部分:设备服务子程序包含了设备操作相关的程序代码;中断处理程序负责处理设备通过中断方式向设备驱动程序发出的输入/输出请求。

8.1.2 设备号

Linux 系统使用设备号来标识设备文件。设备号是一个数字,它是设备的标志。设备号分为主设备号和从设备号。

主设备号表明某一类设备,是同一类设备的标识,对应着一个驱动程序。每个在内核中的设备驱动程序都有一个唯一的主设备号,包括编译进内核的和动态加载的驱动程序。内核根据主设备号将设备与相应的驱动程序关联起来。

从设备号是在驱动程序中来指示某个物理设备实例的,从设备号使得不同的设备实例可以使用同一个驱动程序。

在 Linux 2.6 的内核中,主、从设备号定义为一个 `dev_t` 类型的 32 位数(`dev_t` 类型定义在 `/include/linux/types.h` 文件中),其中前 12 位表示主设备号,后 20 位表示从设备号。另外,在 `include/linux/kdev_t.h` 中定义了如下几个宏,来操作主、从设备号。

- `int MAJOR(dev_t dev)`。从 32 位 `dev_t` 类型数据中获得主设备号。
- `int MINOR(dev_t dev)`。从 32 位 `dev_t` 类型数据中获得从设备号。
- `dev_t MKDEV(unsigned int major, unsigned int minor)`。将主设备号和从设备号转换为 `dev_t` 类型数据。

8.1.3 字符设备驱动程序注册

在 Linux 操作系统中,字符设备是最简单的一类设备,在内核中存在一个数组 `chrdevs[]` 来保存所有字符设备驱动程序的信息,如图 8.2 所示。这个数组的数据结构(定义在 `fs/char_dev.c` 文件中)如下:

```
static struct char_device_struct {  
    struct char_device_struct * next;  
    unsigned int major;  
    unsigned int baseminor;
```

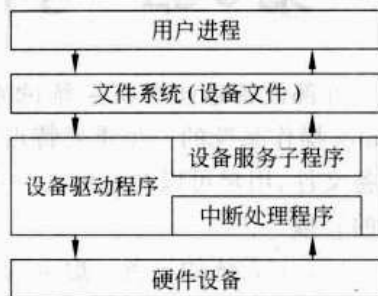


图 8.1 设备驱动程序在 Linux 系统中的层次结构图

```

int minorct;
char name[64];
struct file_operations * fops;
struct cdev * cdev;
} * chrdevs[CHRDEV_MAJOR_HASH_SIZE];

```

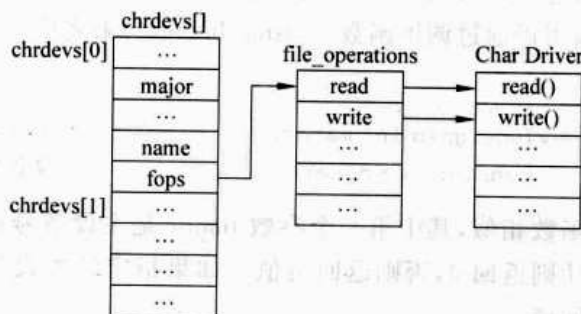


图 8.2 Linux 的字符设备注册表

数组的每一项代表一个字符设备驱动程序。字符设备驱动程序的注册也就是将新的字符设备驱动程序插入到字符设备驱动程序数组中。

通过调用 `register_chrdev()` 向系统注册字符型设备驱动程序。`register_chrdev()` 的原型如下：

```

int register_chrdev(unsigned int major,           //主设备号
                    const char * name,           //设备名称
                    struct file_operations * ops); //指向设备操作函数指针

```

其中 `major` 是设备驱动程序向系统申请的主设备号。如果 `major` 为 0, 则系统为该驱动程序动态分配一个空闲的主设备号。`name` 是设备名称, `ops` 是指向设备操作函数结构的指针。

`register_chrdev()` 函数有以下几个返回值。

- 0。表示注册成功。
- -EINVAL。表示申请的主设备号非法。
- -EBUSY。表示所申请的主设备号正被其他设备驱动程序使用。
- 正数。如果动态分配主设备号成功, 则该函数将返回所分配的主设备号。

通过 `/proc/devices` 文件可以查看到注册成功的设备驱动程序名。

字符设备驱动程序注册成功后, 还必须在文件系统中使用 `mknod` 命令为其创建一个设备文件, 每个设备文件代表一个具体的设备实例。创建设备文件时需要使用设备的主设备号和从设备号作为参数。

注销字符设备驱动程序的函数是 `unregister_chrdev()`, 原型如下:

```
void unregister_chrdev(unsigned int major, const char * name);
```

8.1.4 块设备驱动程序注册

和字符设备驱动程序类似, 内核中块设备驱动程序也需要一个主设备号来标识。对于块设备驱动程序的注册可以通过调用函数 `register_blkdev()` 来完成。 `register_blkdev()` 的原型如下:

```
int register_blkdev(unsigned int major,          //主设备号
                    const char * name);          //设备名
```

与字符设备注册函数相似, 其中第一个参数 `major` 是主设备号; 第二个参数 `name` 是设备名。函数调用成功则返回 0, 否则返回负值。如果指定的主设备号为 0, 则该函数将分配的设备号作为返回值。

`register_blkdev` 函数将设备号与驱动程序建立起关联。对块设备操作时还需要用到另外一个结构 `gendisk`。 `gendisk` (在 `<linux/genhd.h>` 中声明) 的数据结构表示了一个独立的磁盘。因此还需要使用这个数据结构向内核注册磁盘。 `gendisk` 结构定义如下:

```
struct gendisk {
    int major;                //主设备号
    int first_minor;          //第一个从设备号, 一般为 0
    int minors;               //从设备数
    char disk_name[32];       //主设备名
    struct hd_struct ** part;
    struct block_device_operations fops; //块设备的操作函数结构指针
    struct request_queue queue; //内核管理块设备 I/O 请求的队列
    void private_data;
    sector_t capacity;
    int flags;                //驱动程序状态标志
    ...
};
```

首先使用 `alloc_disk()` 函数来分配 `gendisk` 结构。 `alloc_disk()` 函数的原型如下:

```
struct gendisk * alloc_disk(int minors);
```

分配 `gendisk` 结构后, 驱动程序调用 `add_disk()` 函数将自己的 `gendisk` 添加到系统的设备列表中。此时磁盘设备被激活, 可以使用相关的操作函数对磁盘进行操作。

卸载磁盘可以调用函数 `del_gendisk()` 来完成。

注销块设备驱动程序的函数是 `unregister_blkdev()`, 原型如下:

```
int unregister_blkdev(unsigned int major, const char * name);
```

8.2 字符设备和块设备驱动程序结构

驱动程序对用户隐藏了设备的具体细节,对不同的硬件设备提供了相同的操作接口。设备驱动程序一般可以分为初始化程序、I/O 请求服务程序和中断服务程序三个主要部分。

初始化程序负责检测硬件设备是否存在和是否能够正常工作。如果该设备一切正常,则对这个设备进行初始化。这部分驱动程序仅在初始化时调用一次。

I/O 请求服务程序是在系统执行时,由用户进程调用的。当用户进程调用此部分程序时,系统由用户态切换到内核态,完成相关的操作。

中断服务程序是当硬件发生中断时系统调用的一段内核服务程序。在 Linux 系统中,中断服务程序是由系统接收硬件中断后由系统调用的。

8.2.1 字符设备驱动程序结构

前面已经学习了如何注册字符设备驱动程序。这里介绍字符设备驱动程序的 I/O 请求处理函数。

在系统内部,I/O 设备的存取通过一组固定的入口点来进行,字符设备驱动程序一般提供以下几个入口点。

(1) open。打开设备时进行的操作。如果设备是独占的,则需要设置一些标志量来表明设备状态。

(2) read。通过设备读数据。对于有缓冲区的设备,一般是从缓冲区里读数据。对字符设备文件进行读操作将调用 read 程序。

(3) write。向设备写数据。对于有缓冲区的设备,一般是把数据写入缓冲区里。对字符设备文件进行写操作将调用 write 程序。

(4) ioctl。执行读、写之外的操作。

(5) release。释放一个设备。如果设备是独占设备,须标记设备状态。

在 Linux 系统中,设备的操作演变成对文件的操作。因此字符设备驱动程序中一个重要的工作就是填充 fops 指针对应的文件操作函数。每个打开的设备文件均与一组函数关联。

当一个设备文件被进程打开时,open()函数被调用,该函数的实现比较简单。

```
static int mydev_open(struct inode * inode, struct file * filp) {  
    ...  
    return 0;  
}
```

```
}
```

release()函数的定义和 open()函数相似,在文件被 close 时调用。

```
static int mydev_release(struct inode * inode, struct file * filp) {  
    ...  
    return 0;  
}
```

对于 read()和 write()函数的实现稍微复杂一些。read()函数从设备中读取数据,实际上是在内核中读取,传送给用户空间。write()函数是从用户空间的缓冲区 buf 复制到硬件或内核的缓冲区中。

ioctl()函数主要是用来获取或改变正在运行的设备的参数,原型如下:

```
static int mydev_ioctl(struct inode * inode, struct file * filp, unsigned long int  
cmd, unsigned long arg);
```

参数 cmd 是驱动程序要执行的命令的特殊代码,参数 arg 是为特定的 cmd 提供参数。

8.2.2 块设备驱动程序结构

字符设备使用 file_operations 结构指定相应的操作接口,块设备使用 block_device_operations 结构来指定相应的操作接口。该结构中同样存在 open、release 及 ioctl 等操作函数,但是没有负责读和写的操作函数。块设备 I/O 均需要高效地传输数据,因此由内核提供的 request 机制统一来完成。

request 请求函数是块设备驱动程序的核心。块设备的读、写操作全部是由 request()函数来完成的。每个块设备有一个请求队列和一个请求队列的自旋锁,请求函数捆绑在请求队列上。请求队列只要不为空,相应的驱动程序就会从队列头获取请求,将其送到相应的块设备上。

初始化一个请求队列的方法如下:

```
request_queue_t * blk_init_queue(request_fn_proc *, spinlock_t *);
```

该函数返回指向成功初始化后的请求队列指针,此时设备驱动程序提供的请求函数 request_fn_proc 已经与请求队列绑定。成功调用后构造出一个 request 结构的实例,以后的 I/O 处理函数都使用该结构进行工作。

elv_next_request 函数用来获取队列中第一个未完成的请求,当没有请求要处理时返回 NULL,但不删除队列中的 request。函数定义如下:

```
struct request * elv_next_request(struct request_queue * q);
```


一个块设备请求队列中包含一些实际不需要向磁盘读写数据的请求。例如请求一些设备的控制和诊断的信息。判断一个请求是否是文件数据请求,可以使用 `blk_fs_request()` 函数,返回非 0 表示是一个文件数据请求;返回 0 表示不是一个文件数据请求。函数 `end_request` 可以用来确认请求是否被处理。

```
void end_request(struct request * req, int uptodate);  
uptodate 为 0 表示请求被忽略;uptodate 为 1 表示请求被处理。
```

`request` 机制可以采用电梯算法处理并发的请求,对于处理大量连续读写的数据访问是非常高效的。在 `request` 机制中还使用了 `bio` 机制。`bio` 机制在系统中能够表示包括多个物理页的 I/O 操作。这可以大大提高某些块设备的性能。`Request` 机制使用 `bio` 机制来表示相应的 I/O 操作。

8.3 字符设备和块设备驱动程序的内核相关数据结构

通过前面的介绍,基本了解了字符设备和块设备驱动程序的基本框架。在驱动程序的相关操作中使用到了一些重要的内核数据结构,如 `file_operations`、`file` 和 `inode` 等。下面介绍这些重要的数据结构。

`file_operations` 结构是在内核中定义的。指向 `file_operations` 的指针常用 `fops` 表示。在 Linux 系统中,设备驱动程序所提供的 I/O 操作的入口点是由一个文件操作结构来向系统进行说明。`file_operations` 结构定义于 `linux/fs.h` 文件中,随着内核版本的不断变化,`file_operations` 结构也越来越大,不同版本的内核会稍有不同。Linux 2.6.18 版本的 `file_operations` 结构体定义如下(`include/linux/fs.h`)。

```
struct file_operations {  
    struct module * owner;  
    loff_t (* llseek)(struct file *, loff_t, int);  
    ssize_t (* read)(struct file *, char __user *, size_t, loff_t *);  
    ssize_t (* aio_read)(struct kiocb *, char __user *, size_t, loff_t);  
    ssize_t (* write)(struct file *, const char __user *, size_t, loff_t *);  
    ssize_t (* aio_write)(struct kiocb *, 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);  
    long (* unlocked_ioctl)(struct file *, unsigned int, unsigned long);  
    long (* compat_ioctl)(struct file *, unsigned int, unsigned long);  
    int (* mmap)(struct file *, struct vm_area_struct *);  
    int (* open)(struct inode *, struct file *);
```

```

int(* flush)(struct file*, fl_owner_t id);
int(* release)(struct inode*, struct file*);
int(* fsync)(struct file*, struct dentry*, int datasync);
int(* aio_fsync)(struct kiocb*, int datasync);
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);
int(* dir_notify)(struct file* filp, unsigned long arg);
int(* flock)(struct file*, int, struct file_lock*);
ssize_t(* splice_write)(struct pipe_inode_info*, struct file*,
    loff_t*, size_t, unsigned int);
ssize_t(* splice_read)(struct file*, loff_t*, struct pipe_inode_
    info*, size_t, unsigned int);
};

```

每个打开的设备文件均与一组函数关联。file_operations 结构中的成员大部分是函数指针。对于设备中不支持的操作,该结构中对对应指针可置为 NULL。下面是一个设备驱动程序的 file_operations 定义:

```

struct file_operations my_fops={
    .open=my_open,
    .read=my_read,
    .release=my_release,
};

```

每个进程对设备的操作,都会根据设备的设备号,转换成对 file_operations 结构的访问。

块设备使用与 file_operations 相似的结构,块设备里定义了 block_device_operations,结构如下:

```

struct block_device_operations {
    int(* open)(struct inode*, struct file*);
    int(* release)(struct inode*, struct file*);
};

```

```

int(* ioctl)(struct inode*, struct file*, unsigned, unsigned long);
long(* unlocked_ioctl)(struct file*, unsigned, unsigned long);
long(* compat_ioctl)(struct file*, unsigned, unsigned long);
int(* direct_access)(struct block_device*, sector_t, unsigned long*);
int(* media_changed)(struct gendisk*);
int(* revalidate_disk)(struct gendisk*);
int(* getgeo)(struct block_device*, struct hd_geometry*);
struct module* owner;
};

```

在块设备中也同样存在打开、关闭设备等操作,也同样包括设备控制接口,但是不存在读写函数接口。前面已经介绍过,块设备使用 request 机制来进行数据的传输。

file 结构代表一个打开的文件,每执行一次 open 就建立一个 file 结构,直到所有打开的文件都关闭。file 结构定义如下:

```

struct file {
    union {
        struct list_head    fu_list;
        struct rcu_head      fu_rcuhead;
    } f_u;
    struct dentry            *f_dentry;
    struct vfsmount          *f_vfsmnt;
    const struct file_operations *f_op;
    atomic_t                 f_count;
    unsigned int              f_flags;
    mode_t                   f_mode;
    loff_t                   f_pos;
    struct fown_struct        f_owner;
    unsigned int              f_uid, f_gid;
    struct file_ra_state      f_ra;
    unsigned long             f_version;
    void                      *f_security;
    void                      *private_data;
    struct address_space      *f_mapping;
};

```

inode 结构在内核中包含了关于文件的大量内部信息,多个文件指针可以同时指向单一的一个 inode。对于设备驱动主要关注的字段如下:

```

struct inode {
...

```

```

...
    dev_t          i_rdev;

    struct block_device * i_bdev;
    struct cdev        * i_cdev;
};

```

当从块设备输入输出数据时, bio 结构被传递给代码。bio 结构包含了驱动程序执行请求的全部信息。bio 结构定义在<linux/bio.h>中。

```

struct bio {
    sector_t          bi_sector;
    struct bio        * bi_next;           //请求链表
    struct block_device * bi_bdev;
    unsigned long     bi_flags;           //状态、命令
    unsigned long     bi_rw;              //读或写
    unsigned short    bi_vcnt;            //bio_vec 偏移个数
    unsigned short    bi_idx;             //bvl_vec 索引
    unsigned short    bi_phys_segments;   //合并后片断数目
    unsigned short    bi_hw_segments;
    unsigned int      bi_size;            //I/O count
    unsigned int      bi_hw_front_size;
    unsigned int      bi_hw_back_size;
    unsigned int      bi_max_vecs;       //bvl_vecs 最大数目
    struct bio_vec     * bi_io_vec;       //vec 链表
    bio_end_io_t      * bi_end_io;
    atomic_t          bi_cnt;            //pin 记数
    void              * bi_private;
    bio_destructor_t  * bi_destructor;    //销毁函数
};

```

request 结构是块设备进行数据请求所使用的一个重要的数据结构,该结构包含很多数据成员。下面简单列出几个常见的数据成员并加以介绍。

```

struct request {
    struct list_head queuelist;
    sector_t hard_sector;
    unsigned long hard_nr_sectors;
    unsigned int hard_cur_sectors;
    struct bio * bio;
    unsigned short nr_phys_segments;
};

```

- `queuelist`。是一个内核链表结构,将请求连接到请求队列中。
- `hard_sector`。保存下一个将传输的扇区位置。
- `hard_nr_sectors`。保存了等待传输扇区的总数量。
- `hard_cur_sectors`。保存了当前 `bio` 中剩余的扇区数目。
- `bio`。代表该请求的 `bio` 结构链表。
- `nr_phys_segments`。表示当相邻的页被合并后,这个请求所占用的物理内存中段的数目。

8.4 开发实例

前面已经了解了字符设备和块设备驱动程序的基本结构,下面以两个具体的实例来了解一下这两类驱动程序的编写方法。

8.4.1 字符设备驱动程序

下面是一个简单的字符设备驱动的例子。在这个字符设备驱动程序中,需要给其添加模块初始化函数,该函数用来完成对所控设备的初始化工作,并调用 `register_chrdev()` 函数注册字符设备。与模块初始化函数对应的就是模块卸载函数,需要调用 `unregister_chrdev()` 函数。另外,在 `read()` 和 `write()` 函数中涉及了内核空间与用户空间的内存交互,需要使用 `copy_to_user()` 和 `copy_from_user()` 函数。`ioctl()` 函数是特殊的控制函数,可以向设备传递控制信息或从设备取得状态信息。

对字符设备文件进行 `open` 系统调用时,将调用驱动程序的 `open()` 函数。打开字符设备的定义如下:

```
static int my_open(struct inode* inode, struct file* file)
{
    printk("my_device_open(%p,%p)\n", inode, file);
    return 0;
}
```

当最后一个打开设备的用户进程执行 `close` 系统调用时,内核将调用驱动程序的 `release()` 函数,完成释放设备的处理工作。

```
static void my_release(struct inode* inode, struct file* file)
{
    printk("my_device_release(%p,%p)\n", inode, file);
}
```

当对设备文件进行 `read` 操作时,将调用此函数,完成内核空间向用户空间的数据传

递。函数实现如下所示：

```
static ssize_t my_read(struct file* f, char* buf, int size, loff_t off)
{
    int length;
    int count=size;
    if(count<0)
        return-EINVAL;
    if(mydev.usage)
        return-EBUSY;
    mydev.usage=1;
    if(mydev.data==0)
        return 0;
    length=strlen(mydev.data);
    if(length<count)
        count=length;
    copy_to_user(buf,mydev.data,count+1);
    mydev.new_msg=0;
    mydev.usage=0;
    return count;
}
```

当对设备文件进行 write 操作时,将调用此函数,完成用户空间向内核空间的数据传递。函数实现如下所示:

```
static ssize_t my_write(struct file* f, const char* buf, int size, loff_t off)
{
    int count=size;
    if(count<0)
        return-EINVAL;
    if(mydev.usage|mydev.new_msg)
        return-EBUSY;
    mydev.usage=1;
    kfree(mydev.data);
    mydev.data=kmalloc(sizeof(char) * (count+1),GFP_KERNEL);
    if(!mydev.data)
        return-ENOMEM;
    copy_from_user(mydev.data,buf,count+1);
    mydev.usage=0;
    mydev.new_msg=1;
    return count;
}
```

对字符设备进行控制,函数实现如下所示:

```
static int my_ioctl(struct inode *inod, struct file *f, unsigned int arg1,
                    unsigned int arg2)
{
    int ret=0;
    switch(arg1)
    {
        case MYDEV_RESET:
            kfree(mydev.data);
            mydev.data=NULL;
            mydev.usage=0;
            mydev.new_msg=0;
            break;
        case MYDEV_NEW_MSG:
            if(mydev.new_msg)
                return 1;
            break;
        case MYDEV_LENGTH:
            if(mydev.data==NULL)
                return 0;
            else
                return strlen(mydev.data);
            break;
        default:
            return -ENOTTY;
    }
    return ret;
}
```

初始化字符设备驱动的 file_operations 结构体,方法如下:

```
struct file_operations my_fops={
    .read=my_read,
    .write=my_write,
    .ioctl=my_ioctl,
    .open=my_open,
    .release=my_release,
};
```

字符设备驱动程序注册时所做的工作,函数实现如下所示:

```
static int __init mydev_init(void)
{
    int result;
    result=register_chrdev(my_major,DEVICE_NAME,&my_fops); //注册设备驱动
    if(result<0)
        return result;
    if(my_major==0)
        my_major=result;
    printk(KERN_INFO "Register MYDEV. major-number=%d\n",result);
    return 0;
}
```

卸载驱动程序所做的工作,函数实现如下所示:

```
static void __exit mydev_exit(void)
{
    printk(KERN_INFO "Unregister MYDEV\n");
    unregister_chrdev(my_major, DEVICE_NAME); //注销设备驱动
}
```

虚拟字符设备驱动程序的结构已经介绍完毕,读者可以结合前面所学知识,编写一个完整的字符设备驱动程序,并编写一个测试程序进行测试。

8.4.2 块设备驱动程序

在 Linux 源码目录树下的 drivers/block/rd.c 文件,是一个 RAM disk 驱动程序。在 Linux 中可以将一部分内存当作分区来使用,称为 RamDisk。对于一些经常被访问,并且不会被更改的文件,可以将它们通过 RamDisk 放在内存中,能够明显地提高系统性能。下面是 Linux 内核源码中 RamDisk 驱动程序的部分源码

对块设备进行初始化,将块设备注册到内核中。函数清单如下。

```
static int __init rd_init(void)
{
    int i;
    int err=-ENOMEM;
    //检查块大小。块大小必须小于一个内存页面的大小且要大于 512 字节,是 2 的 N 次幂
    if(rd_blocksize>PAGE_SIZE || rd_blocksize<512 ||
```

```

        (rd_blocksize & (rd_blocksize-1))) {
    printk("RAMDISK: wrong blocksize %d, reverting to defaults\n", rd_blocksize);
    rd_blocksize=BLOCK_SIZE;                //使用默认值
}

for(i=0; i<CONFIG_BLK_DEV_RAM_COUNT; i++) {
    rd_disks[i]=alloc_disk(1);                //分配硬盘描述符
    if(!rd_disks[i])
        goto out;

    rd_queue[i]=blk_alloc_queue(GFP_KERNEL);    //分配请求队列描述符
    if(!rd_queue[i]) {
        put_disk(rd_disks[i]);
        goto out;
    }
}

//注册块设备
if(register_blkdev(RAMDISK_MAJOR, "ramdisk")) {
    //注册块设备
    err=-EIO;
    goto out;
}

//初始化请求队列,并将磁盘加入到队列中去
for(i=0; i<CONFIG_BLK_DEV_RAM_COUNT; i++) {
    struct gendisk * disk=rd_disks[i];        //注册请求队列的 BIO 请求函数
    blk_queue_make_request(rd_queue[i], &rd_make_request);    //绑定请求函数
    blk_queue_hardsect_size(rd_queue[i], rd_blocksize);    //硬件扇区尺寸设置
    disk->major=RAMDISK_MAJOR;                //主设备号
    disk->first_minor=i;
    disk->fops=&rd_bd_op;
    disk->queue=rd_queue[i];                //每个盘一个请求队列
    disk->flags |= GENHD_FL_SUPPRESS_PARTITION_INFO;
    sprintf(disk->disk_name, "ram%d", i);
    set_capacity(disk, rd_size*2);
    add_disk(rd_disks[i]);                //添加 gendisk
}

```

```

    printk("RAMDISK driver initialized: "
           "%d RAM disks of %dK size %d blocksize\n",
           CONFIG_BLK_DEV_RAM_COUNT, rd_size, rd_blocksize);

    return 0;
out:
    while(i--) {
        put_disk(rd_disks[i]);
        blk_cleanup_queue(rd_queue[i]);
    }
    return err;
}

```

RamDisk 还提供 `block_device_operations` 结构体中两个成员函数 `open()` 和 `ioctl()` 的实现,这里不进行分析。

处理 BIO 请求的函数,实现过程如下面程序所示:

```

static int rd_make_request(request_queue_t *q, struct bio *bio)
{
    struct block_device *bdev=bio->bi_bdev;
    struct address_space * mapping=bdev->bd_inode->i_mapping;
    //起始的扇区号
    sector_t sector=bio->bi_sector;
    //传输数据的总长度大小
    unsigned long len=bio->bi_size>>9;
    int rw=bio_data_dir(bio);
    struct bio_vec *bvec;
    int ret=0, i;

    if (sector+len>get_capacity(bdev->bd_disk))
        goto fail;

    if (rw==READA)
        rw=READ;
    //处理 BIO 中的每一个 segment
    bio_for_each_segment(bvec, bio, i) {
        ret |=rd_blkdev_pagecache_IO(rw, bvec, sector, mapping);
        sector+=bvec->bv_len>>9;
    }
    if (ret)

```

//超过容量


```

        goto fail;
    bio_endio(bio, bio->bi_size, 0); //处理结束
    return 0;
fail:
    bio_io_error(bio, bio->bi_size);
    return 0;
}

```

卸载函数,注销设备,并释放先前申请的资源。

```

static void __exit rd_cleanup(void)
{
    int i;
    for(i=0; i<CONFIG_BLK_DEV_RAM_COUNT; i++) {
        struct block_device * bdev=rd_bdev[i];
        rd_bdev[i]=NULL;
        if(bdev) {
            invalidate_bdev(bdev, 1);
            blkdev_put(bdev);
        }
        del_gendisk(rd_disks[i]); //删除 gendisk
        put_disk(rd_disks[i]); //释放对 gendisk 的引用
        blk_cleanup_queue(rd_queue[i]); //清除请求队列
    }
    unregister_blkdev(RAMDISK_MAJOR, "ramdisk");
}

```

8.5 本章小结

本章主要介绍了 Linux 字符设备及块设备驱动程序的开发方法。首先介绍了设备驱动程序的设计基础知识。接下来介绍了字符设备及块设备驱动程序的结构。最后介绍了两个驱动程序实例,一个是虚拟字符设备驱动,另一个是 RAMDisk 块设备驱动程序,来帮助读者了解驱动程序的开发流程及方法。

思考题

1. Linux 2.6 和 Linux 2.4 内核的字符设备驱动的注册方法有哪些不同?
2. 分析字符设备驱动的静态注册方法与动态注册方法。

进一步阅读

[1] Jonathan Corbet, Alessandor Rubini, Greg Kroah-Hartman 著. 魏永明, 耿岳, 钟书毅译. Linux 设备驱动程序(第 3 版). 北京: 中国电力出版社, 2006

[2] Linux 下 USB 驱动编写

<http://www.linux-usb.org/>

<http://www.ibm.com/developerworks/cn/linux/l-usb/index1.html>

<http://www.ibm.com/developerworks/cn/linux/l-usb/index2.html>

第9章 网络设备驱动程序

9.1 网络设备驱动程序设计基础

网络设备(Network Device)是指提供网络功能的设备。常见的网络设备有集线器(Hub)、网桥(Bridge)、交换机(Switch)、路由器(Router)和网卡(Network Interface Card, NIC)。网卡是我们所熟悉的设备,也叫网络适配器。它是局域网中最基本的部件之一,是连接计算机与网络的硬件设备,能使计算机通过网络介质接收和发送数据。每个网卡有一个唯一的 MAC(Media Access Control, 介质访问控制)地址。我们日常使用的网卡为以太网网卡。

网络设备驱动程序的一些基本要求如下。

(1) 数据包的发送和接收。网络设备将计算机要发送的数据分解成适当大小的数据包再向网络发送;网络设备也从网络接收发送到计算机的数据帧,然后传给上层协议处理。

(2) 中断处理。操作系统必须提供驱动程序响应中断的能力。操作系统在硬件中断发生后调用驱动程序的中断处理程序。网络设备对数据的发送和接收处理一般通过中断的方式实现。Linux 系统支持中断的共享,即多个设备共享一个中断。

(3) 时钟。在实现驱动程序时,很多地方会用到时钟。如超时处理、没有中断机制的硬件轮询等。在网络设备驱动程序中,如果硬件不支持中断,那么驱动程序可以利用定时器,通过轮询的方式,查询是否有需要发送或接收的数据,并根据查询结果进行相应处理。

实现 Linux 网络设备驱动功能主要有两种形式:一是通过内核加载,即内核启动时就加载网络设备驱动;二是通过模块加载。模块加载过程如图 9.1 所示,其实现过程如下。

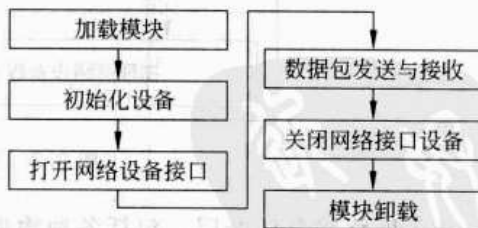


图 9.1 网络设备驱动模块加载实现过程

(1) 通过模块加载命令 insmod 把网络设备驱动程序加载到内核中。

(2) 对设备进行初始化,将设备的 device 数据结构插入到 dev_base 链表的末尾,调用 register_netdev() 函数在 Linux 系统中注册该网络设备。

(3) 打开网络设备。

(4) 进行数据包的发送与接收。

(5) 关闭网络设备,释放硬件资源。

(6) 执行模块卸载命令 `rmmmod`,卸载网络驱动程序模块。

网络驱动程序与块设备驱动程序有以下不同之处:块设备(如硬盘)在 `/dev` 目录下以特殊文件形式存在,而网络设备不是以文件形式存在,因此网络设备不支持一般的文件操作(如读文件、写文件等),它有自己的内存名字空间,提供特定的函数接口供调用;块设备只响应来自内核的请求,而网络驱动程序异步地接收外部数据。内核要求块设备驱动程序向其发送缓冲区数据,网络设备则向内核请求把从外部获得的数据包发送给内核。

网络驱动还支持大量的管理任务,如设置地址、修改传输参数、维护流量和错误统计等。

9.2 Linux 网络设备驱动程序的结构

Linux 网络设备驱动程序(如图 9.2 所示)由 4 部分组成:网络设备媒介层、网络设备驱动层、网络设备接口层及网络协议接口层。

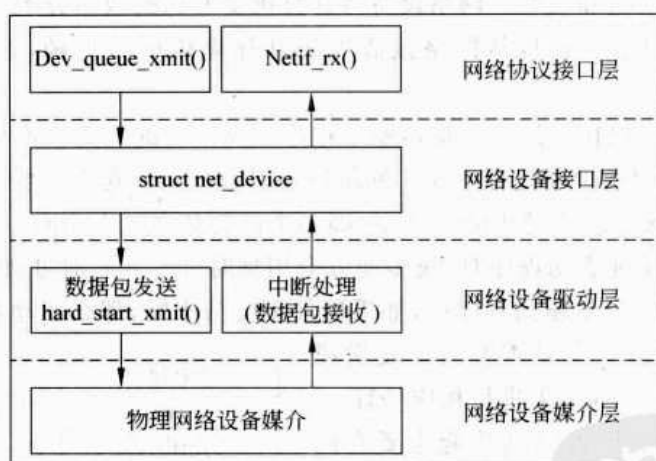


图 9.2 网络设备驱动结构

(1) 网络设备媒介层。包括各种物理网络设备和传输媒介。

(2) 网络设备驱动层。网络设备驱动主要包括网络设备的初始化、数据包的发送和接收。

(3) 网络设备接口层。Linux 系统用 `Net_device` 结构表示网络设备接口。`Net_device` 结构保存所有与硬件有关的接口信息,各协议软件主要通过 `Net_device` 结构来完成与硬件的交互。全部网络设备接口组成一个由内核管理的以 `dev_base` 为头指针的设备链表,每个网络设备接口对应一个存在的网络设备,一个网络设备接口可以是物理接

口,也可以是软件虚拟接口,如环路设备(Loopback Device)等。

(4) 网络协议接口层。提供网络接口驱动程序的抽象接口。该接口为上层协议层提供服务,同时它隐藏了网络设备驱动程序的实现细节,相邻层的通信只需要通过特定函数调用实现,使通信过程相对透明,简化了开发的难度。

网络设备初始化过程如下。

(1) 探测物理设备。初始化例程探测它的设备及硬件位置(I/O 端口和 IRQ 线),并根据硬件的特征检查硬件是否存在。

(2) 配置资源。完成对硬件资源的配置。

(3) 初始化 net_device 结构。调用与网络设备相关的 setup() 回调函数(如以太网卡使用 ether_setup)来初始化 net_device 结构中的变量。

(4) 注册设备。在网络设备全局链表中插入一个 net_device 网络设备接口来表示该网络设备。

数据包的发送和接收是实现 Linux 网络驱动程序的两个关键的过程,这两个过程处理的好坏将直接影响到驱动程序的整体运行质量。

网络协议接口层发送数据前,对于以太网网卡驱动程序,需要调用 hard_header 函数,用来建立硬件帧头。hard_header 函数传送的参数包括数据的 SKB (struct sk_buff)、设备指针、协议名、目的地址、源地址和数据长度。hard_header 函数调用 skb_push 函数来填入硬件帧头。如果硬件帧头中的信息还不完全,则返回负数,再由网络协议接口层进一步建立硬件帧头。

网络设备驱动程序发送数据时,系统调用 hard_start_xmit 函数,发送的数据放在一个 sk_buff 结构中。一般的驱动程序将数据传输到硬件发出去,特殊的设备如环路设备把数据组成一个接收数据再回送给系统。如果发送成功,则在 hard_start_xmit 函数里释放 SKB (支持 DMA 发送的设备在收到发送成功中断后释放 SKB),返回 NETDEV_TX_OK;否则返回错误。

对于接收数据的处理,设备收到数据后产生一个中断,中断处理程序申请一块缓冲区 (SKB),从硬件读出数据放置到该缓冲区。然后填充 SKB 中的一些信息:判断收到的帧的协议类型,填入 skb->protocol;把指针 skb->mac.raw 指向硬件数据,然后丢弃硬件帧头;设置 skb->pkt_type,表示链路层数据类型。数据类型如下。

- PACKET_BROADCAST。链路层广播。
- PACKET_MULTICAST。链路层组播。
- PACKET_SELF。发给自己的帧。
- PACKET_OTHERHOST。发给其他主机的帧。

最后调用 `netif_rx()` 函数把数据传送给上层协议层处理。

驱动程序还提供对设备参数进行设置和读取的方法。设置方法如下几种。

(1) `dev->set_mac_address()`。当用户调用 `ioctl` 类型为 `SIOCSIFHWADDR` 时要设置设备的 mac 地址。

(2) `dev->set_config()`。当用户调用 `ioctl` 的类型为 `SIOCSIFMAP` 时, 系统会调用该方法。用户会传递一个 `ifmap` 结构包含需要的 I/O、中断等参数。

(3) `dev->do_ioctl()`。参数类型在 `SIOCDEVPRIVATE` 和 `SIOCDEVPRIVATE + 15` 之间时, 系统调用该方法来设置或读取设备信息。

(4) `dev->get_stats()`。返回一个 `enet_statistics` 结构, 包含统计信息。

网络驱动程序实现时要注意的问题如下。

1. 中断共享

设备申请中断的函数接口为:

```
int request_irq(unsigned int irq, void (* handler) (int irq, void * dev_id, struct
pt_regs * regs), unsigned long irq_flags, const char * devname, void * dev_id);
```

其中, `irqflags` 表示中断的属性。如果有多个设备共享同一个中断号, 那么所有共享该中断号的设备都必须将 `irqflags` 设置为 `SA_SHIRQ` 属性。

2. I/O 端口

每个驱动应该避免误用 I/O 端口。有关端口的内核函数如下。

(1) `check_region(int io_part, int off_set)`: 参数 1 为 I/O 端口的基址, 参数 2 为 I/O 端口偏移量。函数功能是通过查看系统的 I/O 表来检查是否有别的驱动程序占用该 I/O 端口。

(2) `request_region(int io_part, int off_set, char * devname)`: 参数 1 为 I/O 端口的基址, 参数 2 为 I/O 端口偏移量, 参数 3 为设备名。函数功能是设备申请 I/O 端口, 如果该端口未被占用, 则申请成功。在 `/proc/ioprots` 文件中可看到已登记的 I/O 端口。

3. 流量控制

对于发送数据的流量控制, 设备数据结构里的 `tx_queue_len` 参数表示发送时最多可以缓存的数据包数。另外, 通过高层协议窗口控制流量, 达到了窗口大小, 高层协议就不再发送数据。对于接收数据的流量控制, `netif_rx()` 函数对缓存的数据包数有限制。此外, 高层协议也有一个最大的等待接收处理的数据量。

9.3 网络设备驱动程序的内核相关数据结构

9.3.1 net_device 结构

在 include/linux/netdevice.h 里定义了网络设备的数据结构 net_device。

分配 net_device 结构的函数接口为：

```
struct net_device alloc_netdev(int sizeof_priv, const char * name, void (* setup)
(struct net_device * ));
```

第一个参数为私有数据区长度，第二个参数为接口名，第三个参数为网络设备接口初始化函数。

net_device 接口的细节如下。

(1) 全局信息。

```
char name[IFNAMSIZ];
```

设备名。如果设备名包含 %d 形式的字符串，那么 register_netdev 函数会将设备名替换成一个唯一的数字，数字从 0 开始递增。

```
unsigned long state;
```

表示设备状态。

```
struct net_device * next;
```

指向全局设备链表中的下一个设备接口结构。

```
int (* init)(struct net_device * dev);
```

网络设备初始化函数。如果该参数被设置的话，register_netdev 函数回调用它来完成设备初始化。现在驱动程序开发者基本不使用该功能，一般在执行 register_netdev 函数前就已经初始化了硬件。

(2) 硬件信息。

```
unsigned long rmem_end;
```

```
unsigned long rmem_start;
```

```
unsigned long mem_end;
```

```
unsigned long mem_start;
```

设备内存信息，表示设备共享内存的起始和结束地址。设备发送和接收数据的内存区域不同时，mem_start、mem_end 参数用来发送数据，rmem_start、rmem_end 参数用来接收

数据。

```
unsigned long base_addr;
```

网络设备接口的 I/O 基址。可以用 ifconfig 命令来改变 I/O 基址的值。

```
unsigned char irq;
```

表示中断号。

```
unsigned char if_port;
```

表示多端口设备上正在使用的端口。

```
unsigned char dma;
```

表示设备的 DMA 通道。

(3) 接口信息。

```
unsigned mtu;
```

表示最大传输单元长度。以太网的 mtu 为 1500 字节数。可以用 ifconfig 命令改变 mtu 值。

```
unsigned short type;
```

表示接口硬件类型,地址解析时用该值判断硬件支持哪种硬件地址。

```
unsigned short hard_header_len;
```

表示硬件头信息长度,数据包中 IP 头前的字节数。以太网的硬件信息长度为 14。

```
unsigned long tx_queue_len;
```

表示设备传输队列允许的最大帧数。

```
unsigned char addr_len;
```

```
unsigned char broadcast[MAX_ADDR_LEN];
```

```
unsigned char dev_addr[MAX_ADDR_LEN];
```

表示硬件地址长度和设备硬件地址。

```
unsigned short flags;
```

接口标志,在 linux/if.h 中定义,包括如下标志。

```
IFF_UP
```

表示只读属性。网络设备接口处于活动状态并准备接收数据时,内核将接口设置为该属性。

IFF_BROADCAST

表示网络设备接口支持广播。

IFF_DEBUG

表示调试模式。

IFF_LOOPBACK

表示环路设备。

IFF_POINTTOPOINT

表示建立点对点的连接,由驱动程序或 ifconfig 命令设置,如 plip 和 ppp 驱动需要设置它。

IFF_NOARP

表示网络设备接口不能进行地址解析。

IFF_PROMISC

设置该标志将激活混杂模式。默认情况下,以太网接口使用一个硬件过滤器来确保它只接收广播数据包与直接发送到接口硬件地址的数据包。但数据包侦听器如 tcpdump 会在接口上设置混杂模式,以检查通过传输介质的所有数据包。

IFF_MULTICAST

表示接口支持组播。

IFF_ALLMULTI

表示接口接收所有的组播数据包。

IFF_MASTER

IFF_SLAVE

这两个标志由负载均衡代码使用。

IFF_PROTSEL

IFF_AUTOMEDIA

表示设备能在多种媒体介质间切换。实际使用中,内核不使用这两个标志。

IFF_DYNAMIC

IFF_RUNNING

表示接口已经启动并正在运行。

IFF_NOTRALLERS

该标志只是为了与 BSD 兼容, Linux 不使用。

int features;

表示网络设备特征, 包括如下标志。

NETIF_F_SG

NETIF_F_FRAGLIST

这两个标志控制分散/聚合 I/O 的使用。

NETIF_F_IP_CSUM

NETIF_F_NO_CSUM

这两个标志告诉内核不要对通过接口传输的数据包进行检验。如果接口能够校验 IP 数据包, 则设置 NETIF_F_IP_CSUM。如果接口不需要校验, 则设置 NETIF_F_NO_CSUM。

NETIF_F_HIGHDMA

设备在使用高端内存时使用该标志。

(4) 设置方法。

int (* open) (struct device * dev);

打开接口。该函数应该登记其所需要的所有系统资源, 打开硬件及完成其他所需要的设置。

int (* stop) (struct device * dev);

停止接口。

int (* hard_start_xmit) (struct sk_buff * skb, struct device * dev);

初始化包的传输。

int (* hard_header) (struct sk_buff * skb, struct device * dev, unsigned short type, void * daddr, void * saddr, unsigned len);

该函数功能为建立硬件信息头, 在 hard_start_xmit 前被调用。

int (* rebuild_header) (void * eth, struct device * dev, unsigned long raddr, struct sk_buff * skb);

该函数用在传输数据包前、完成地址解析之后, 重新建立硬件头。

void (* set_multicast_list) (struct device * dev);

用来设置组播列表。

```
int(* set_mac_address)(struct device* dev, void* addr);
```

设置 MAC 地址。

```
int(* do_ioctl)(struct device* dev, struct ifreq* ifr, int cmd);
```

对接口执行一系列控制命令。

```
int(* set_config)(struct device* dev, struct ifmap* map);
```

改变接口配置,可以改变设备的 I/O 地址和中断号。

```
void(* header_cache)(struct neighbour* neigh, struct hh_cache* hh);
```

根据 ARP(地址解析协议)查询的结果填充 hh_cache 结构。

```
void(* header_cache_update)(struct hh_cache* hh, struct device* dev, unsigned char* haddr);
```

在 hh_cache 结构的目标地址发生变化时进行更新。

```
int(* change_mtu)(struct device* dev, int new_mtu);
```

改变 mtu 即最大传输单元数。

(5) 工具成员。

net_device 结构中的其余数据由接口使用,保存一些有用的状态信息。某些成员由 ifconfigs 和 netstat 使用,用来为用户提供当前的配置信息。这些成员如下。

```
unsigned long trans_start;  
unsigned long last_rx;
```

保存 jiffies 值。驱动程序在传输开始时更新 trans_start 值,在接收数据包时更新 last_rx 值。网络子系统使用 trans_start 来检测传输是否被锁住。

```
int watchdog_timeo;
```

在网络层确定传输已经超时并且调用 tx_timeout 函数之前的最小时间(以 jiffies 为单位)。

```
void* priv;
```

和 filp->private_data 等价,由 alloc_netdev 设置,并且不能被直接访问。

```
struct dev_mc_list* mc_list;  
int mc_count;
```

用来处理组播。mc_count 表明 mc_list 所包含的项的数目。

```
spinlock_t xmit_lock;
int xmit_lock_owner;
```

xmit_lock 用来避免对 hard_start_xmit 函数的多次调用。xmit_lock_owner 用来获得 xmit_lock 的 CPU 编号。

9.3.2 sk_buffer 结构

Linux 系统通过 sk_buff 在网络各层之间传递数据。sk_buff 组织成双向链表。sk_buff 提供管理缓冲区的方法,常用的方法如下。

```
struct sk_buff* __alloc_skb(unsigned int size,gfp_t priority, int fclone, int node);
```

申请一个 sk_buff 并对它初始化,返回申请到的 sk_buff。

```
struct sk_buff* skb_get(struct sk_buff* skb);
```

声明一个对 sk_buff 的引用,返回 sk_buff 指针。

```
int skb_cloned(const struct sk_buff* skb);
```

复制一个 sk_buff。

```
int skb_shared(const struct sk_buff* skb);
```

如果有多个用户用到该 sk_buff,函数返回 true。

```
void skb_queue_head(struct sk_buff_head* list, struct sk_buff* newsk);
```

在 sk_buff 链表头部插入一个新的 sk_buff。

```
void skb_queue_tail(struct sk_buff_head* list, struct sk_buff* newsk);
```

在 sk_buff 链表尾部插入一个新的 sk_buff。

```
void skb_insert(struct sk_buff* old, struct sk_buff* newsk, struct sk_buff_head* list);
```

在 sk_buff 链表的 old 元素前插入一个元素。

```
void skb_append(struct sk_buff* old, struct sk_buff* newsk, struct sk_buff_head* list);
```

在 sk_buff 链表的 old 元素后插入一个元素。

```
unsigned char* skb_push(struct sk_buff* skb, unsigned int len);
```

将 sk_buff 缓冲区长度增加 len。

```
unsigned char* skb_pull(struct sk_buff* skb, unsigned int len);
```

从 sk_buff 缓冲区头部将缓冲区长度增加 len。

```
unsigned char * skb_put(struct sk_buff* skb, unsigned int len);
```

在 sk_buff 缓冲区头部将缓冲区长度减去 len。

9.4 开发实例

以 Realtech 8193 网卡驱动为例。代码来自 Linux 2.6.18 中的 8139too.c 文件,驱动程序的主要模块和函数代码分析如下。

1. 驱动模块的加载和卸载

Realtech 8193 网卡私有数据结构:

```
struct rtl8139_private {
    void __iomem * mmio_addr;           //内存映射地址
    int drv_flags;
    struct pci_dev * pci_dev;           //PCI 设备结构体指针
    u32 msg_enable;
    struct net_device_stats stats;      //网络设备状态
    unsigned char * rx_ring;
    unsigned int cur_rx;                //下一个接收数据包的缓冲区索引
    unsigned int tx_flag;
    unsigned long cur_tx;               //下一个发送数据包的缓冲区索引
    unsigned long dirty_tx;
    unsigned char * tx_buf[NUM_TX_DESC]; //发送数据缓冲区
    unsigned char * tx_bufs;
    dma_addr_t rx_ring_dma;             //接收数据 DMA 地址
    dma_addr_t tx_bufs_dma;
    signed char phys[4];                //物理地址
    char twistie, twist_row, twist_col;
    unsigned int watchdog_fired : 1;
    unsigned int default_port : 4;      //默认端口
    unsigned int have_thread : 1;
    spinlock_t lock;                    //自旋锁
    spinlock_t rx_lock;
    chip_t chipset;
    u32 rx_config;
    struct rtl_extra_stats xstats;
    struct delayed_work thread;         //工作队列
    struct mii_if_info mii;
```

```

    unsigned int regs_len;
    unsigned long fifo_copy_timeout;
};

```

PCI 设备驱动结构:

```

static struct pci_driver rtl8139_pci_driver={
    .name      =DRV_NAME,
    .id_table  =rtl8139_pci_tbl,           //此驱动所支持的网卡系列
    .probe     =rtl8139_init_one,         //初始化网络设备的回调函数
    .remove    =__devexit_p(rtl8139_remove_one), //注销网络设备的回调函数
#ifdef CONFIG_PM
    .suspend   =rtl8139_suspend,          //设备挂起函数
    .resume    =rtl8139_resume,           //设备恢复函数
#endif
};

static int __init rtl8139_init_module (void)
{
    ...
    //注册 PCI 设备
    return pci_register_driver(&rtl8139_pci_driver);
}

static void __exit rtl8139_cleanup_module (void)
{
    //注销 PCI 设备
    pci_unregister_driver (&rtl8139_pci_driver);
}

module_init(rtl8139_init_module);           //加载驱动模块
module_exit(rtl8139_cleanup_module);        //卸载驱动模块

```

2. PCI 设备探测函数 probe, 初始化网络设备

```

static int __devinit rtl8139_init_board (struct pci_dev* pdev,
                                         struct net_device**dev_out)
{
    ...
    //分配并设置网络设备
    dev=alloc_etherdev (sizeof (* tp));

```

```

...
//声明为内核设备模块
SET_MODULE_OWNER(dev);
SET_NETDEV_DEV(dev, &pdev->dev);
//初始化私有结构中的各成员值
tp=netdev_priv(dev);
tp->pci_dev=pdev;
//设置 PCI 设备使能位
pci_enable_device (pdev);
...
//网卡 I/O 内存资源的起始地址、结束地址、标志及大小
pio_start=pci_resource_start (pdev, 0);
pio_end=pci_resource_end (pdev, 0);
pio_flags=pci_resource_flags (pdev, 0);
pio_len=pci_resource_len (pdev, 0);

mmio_start=pci_resource_start (pdev, 1);
mmio_end=pci_resource_end (pdev, 1);
mmio_flags=pci_resource_flags (pdev, 1);
mmio_len=pci_resource_len (pdev, 1);
//申请内存空间,配置网卡的 I/O 内存资源
pci_request_regions (pdev, DRV_NAME);
...
pci_set_master (pdev);
//I/O 内存地址映射
#ifdef USE_IO_OPS //PIO 模式
ioaddr=ioport_map(pio_start, pio_len);
...
dev->base_addr=pio_start;
tp->mmio_addr=ioaddr;
tp->regs_len=pio_len;
#else //MMIO 模式
ioaddr=pci_iomap(pdev, 1, 0);
...
dev->base_addr=(long) ioaddr;
tp->mmio_addr=ioaddr;
tp->regs_len=mmio_len;
...
}

```



```

static int __devinit rtl8139_init_one (struct pci_dev *pdev,
                                     const struct pci_device_id *ent)
{
    ...
    rtl8139_init_board (pdev, &dev);
    ...
    //网络设备回调函数赋值
    dev->open=rtl8139_open;
    dev->hard_start_xmit=rtl8139_start_xmit;
    dev->poll=rtl8139_poll;
    dev->weight=64;
    dev->stop=rtl8139_close;
    dev->get_stats=rtl8139_get_stats;
    dev->set_multicast_list=rtl8139_set_rx_mode;
    dev->do_ioctl=netdev_ioctl;
    dev->ethtool_ops=&rtl8139_ethtool_ops;
    dev->tx_timeout=rtl8139_tx_timeout;
    dev->watchdog_timeo=TX_TIMEOUT;
    ...
    dev->irq=pdev->irq;
    ...
    register_netdev (dev);           //注册网络设备
    pci_set_drvdata (pdev, dev);
    ...
}

```

3. 注销网络设备

```

static void __rtl8139_cleanup_dev (struct net_device *dev)
{
    struct rtl8139_private *tp=netdev_priv(dev);
    struct pci_dev *pdev;
    assert(dev != NULL);
    assert(tp->pci_dev != NULL);
    pdev=tp->pci_dev;
    //取消地址映射
    #ifdef USE_IO_OPS
        if(tp->mmio_addr)
            ioport_unmap (tp->mmio_addr);
    #else

```

```

        if(tp->mmio_addr)
            pci_iounmap (pdev, tp->mmio_addr);
    #endif
    pci_release_regions (pdev);           //释放 PCI 资源
    free_netdev(dev);                     //释放网络设备
    pci_set_drvdata (pdev, NULL);         //PCI 设备中的设备指针赋空值
}

```

```
static void __devexit rtl8139_remove_one (struct pci_dev *pdev)
```

```

{
    struct net_device *dev=pci_get_drvdata (pdev);
    assert(dev!=NULL);
    flush_scheduled_work();
    unregister_netdev (dev);             //注销网络设备
    __rtl8139_cleanup_dev (dev);
    pci_disable_device (pdev);           //停用 PCI 设备
}

```

4. 打开网络设备

```
static int rtl8139_open (struct net_device *dev)
```

```

{
    struct rtl8139_private *tp=netdev_priv(dev);
    int retval;
    ...
    //申请中断, IRQF_SHARED 标识允许共享中断
    retval=request_irq (dev->irq, rtl8139_interrupt, IRQF_SHARED,
                        dev->name, dev);
    ...
    //申请缓冲区
    tp->tx_bufs=pci_alloc_consistent(tp->pci_dev, TX_BUF_TOT_LEN,
                                     &tp->tx_bufs_dma);
    tp->rx_ring=pci_alloc_consistent(tp->pci_dev, RX_BUF_TOT_LEN,
                                     &tp->rx_ring_dma);
    ...
    //初始化缓冲区
    rtl8139_init_ring (dev);
    //初始化硬件
    rtl8139_hw_start (dev);
    //允许网卡开始传输包
}

```

```

        netif_start_queue (dev);
        ...
    }

```

5. 关闭网络设备

```

static int rtl8139_close (struct net_device * dev)
{
    ...
    netif_stop_queue (dev);          //停止网卡传输数据包
    ...
    RTL_W8 (ChipCmd, 0);             //停止接收和发送数据进程
    RTL_W16 (IntrMask, 0);          //清中断标志
    ...
    //释放缓冲区
    pci_free_consistent (tp->pci_dev, RX_BUF_TOT_LEN,
        tp->rx_ring, tp->rx_ring_dma);
    pci_free_consistent (tp->pci_dev, TX_BUF_TOT_LEN,
        tp->tx_bufs, tp->tx_bufs_dma);
    ...
}

```

6. 硬件处理数据包发送

```

static int rtl8139_start_xmit (struct sk_buff * skb, struct net_device * dev)
{
    ...
    //计算下一个传输描述符的入口
    entry=tp->cur_tx%NUM_TX_DESC;
    ...
    //获得自旋锁的同时把标志寄存器的值保存到变量 flags 中,并关闭本地中断
    spin_lock_irqsave (&tp->lock, flags);
    //设置硬件发送参数
    RTL_W32_F (TxStatus0+ (entry* sizeof (u32)),
        tp->tx_flag|max(len, (unsigned int)ETH_ZLEN));
    //记录发包开始时间
    dev->trans_start=jiffies;
    ...
    //解锁并恢复本地中断状态
    spin_unlock_irqrestore (&tp->lock, flags);
    ...
}

```

7. 中断处理接收数据包、发送数据包

```
static irqreturn_t rtl8139_interrupt (int irq, void* dev_instance)
{ ...
    status=RTL_R16 (IntrStatus);
    ...
    //接收数据包的中断处理
    if(status & RxAckBits){
        if(netif_rx_schedule_prep(dev)) {
            RTL_W16_F (IntrMask, rtl8139_norx_intr_mask);
            __netif_rx_schedule (dev);
        }
    }
    ...

    //发送数据包的中断处理
    if(status & (TxOK|TxErr)) {
        rtl8139_tx_interrupt (dev, tp, ioaddr);
        ...
    }
}
```

除上述函数外,还有接收数据包、发送数据包的具体处理函数,以及驱动程序的挂起、恢复函数等。因篇幅原因,在此略去,不作分析。

9.5 本章小结

本章介绍了 Linux 网络设备驱动。首先介绍了 Linux 网络驱动程序的基本要求;然后介绍了 Linux 网络驱动程序结构,阐述了 Linux 网络驱动程序的基本过程和相关注意事项;接着介绍了网络驱动程序的内核相关数据结构,包括 net_device 结构和 sk_buffer 结构的细节;最后介绍了 Realtech 8193 网卡驱动程序,对驱动程序的主要模块和函数代码作出了分析。

思考题

1. Linux 支持中断的共享。对于两个共享同一中断号的网络设备,如果同时产生中断,网络设备驱动程序该如何处理?
2. 以太网的 MAC 地址解析过程是怎样的?

3. 网络的通信模式可分为单播、广播和多播。网卡驱动程序对多播是如何支持的?
4. 网卡从支持的网络可分为基于有线网络的网卡和无线网卡。无线网卡驱动程序与基于有线网络的网卡驱动程序相比,有什么相同和不同之处?
5. 随着网络技术的发展,网络带宽也在不断提高。目前主流的以太网卡主要有 10Mbps 以太网卡、100Mbps 以太网卡、10Mbps/100Mbps 自适应网卡和 1000Mbps 千兆以太网卡 4 种。找一个千兆以太网卡驱动程序源代码,分析其实现过程,说明其为实现千兆高速带宽做了哪些工作。

进一步阅读

1. Corbet, J & A. Rubini & G. K. Hartman 著, 魏永明, 耿岳, 钟书毅译. Linux 网络设备驱动. 中国电力出版社
2. 王玉东, 冯茜, 张效义. 基于 Linux 的无线网卡驱动程序. 信息工程大学学报. 2005, 6(2): 79-80, 87
3. 孙翔, 刘乃安. 基于 NDIS 规范的 WLAN 无线网卡驱动程序的设计与实现. 无线电工程. 2004, 34(6): 23-25, 33

实验 1 Linux 系统管理

一、实验目的

1. 学习使用 Linux 文件系统。
2. 学习 Linux 进程的相关操作命令。
3. 学习 gcc、make 和 gdb 等编程工具链的使用。

二、实验内容——Linux 文件管理实验

通过本实验可以掌握 Linux 系统中文件管理的概念,包括常用命令格式、文件类型、目录结构、文件系统及其安装 Linux 的文件系统等。

1. 实验内容

- (1) 显示当前目录。
- (2) 将当前目录改变到根目录。
- (3) 列出当前目录的内容。
- (4) 在当前目录下,创建名为 ABC 的新目录。
- (5) 在 ABC 目录下创建名为 abc 的新文件,并使用 vi 之类的编辑器在该新文件中输入一些内容。
- (6) 显示当前目录内容。
 - ① 按照长清单格式。
 - ② 显示所有文件,包括隐藏文件。
- (7) 在屏幕上显示文件 abc 的内容。
- (8) 删除 ABC 目录下的 abc 文件。
- (9) 删除当前目录中的 ABC 目录。
- (10) 在用户主目录下创建名为 memos 的目录。
- (11) 利用 cp 命令把用户主目录下的所有文件复制到目录 memos 下。
- (12) 在 memos 目录下新建目录 backup。
- (13) 利用 mv 命令,把 memos 下的两个文件移到 backup 下。
- (14) 删除 memos 目录及下面所有的文件和目录。
- (15) 从根目录开始查找名为 passwd 的文件。

- (16) 查找系统中所有以字母 a 开头的文件。
- (17) 用 grep 打印出 etc/passwd 文件中所有带 root 的行。
- (18) 将用户根目录下的所有文件排序,忽略大小写,并将结果输出到 result 文件中。

2. 安装和卸载文件系统

- (1) 通过查看 /var/log/messages 文件,查找 Linux 系统已识别的设备代号。
- (2) 用 fdisk 在一个未使用的设备上(例如 U 盘)创建新的分区。
- (3) 在其中的一个设备上创建一个 ext3 分区。
- (4) 将这个新的分区挂载到 mnt/ext3 目录下。
- (5) 在这个新的文件系统中试验各种基本文件操作。
- (6) 卸载该文件系统。

3. 背景知识

- (1) 文件系统的基本命令。

我们上机时通过用户界面与计算机打交道。用户界面定义了用户与计算机交流的不同方式,常见的有图形界面、命令界面及编程界面。不同的用户可以选择自己喜欢的界面。Linux 提供了几种不同的用户界面,其中命令界面是 Linux 系统下最简单,但也是功能最强的用户界面。

文件可以用来存放信息和数据,最常见的操作几乎都与文件相关,例如创建删除文件、打开/关闭文件、读/写文件等。下面将对这些常用的命令进行简单的介绍。

- ls

句法: ls [OPTION]...[FILE]...

功能: 列出一个或多个文件(默认为当前目录下)的信息。

OPTION 的常用选项如下。

- -a, --all。列出目录下所有的文件,包括文件名以“.”开头的隐藏文件。
- -l。使用长清单格式,其中包括文件权限、大小和拥有权等信息。
- -r, --reverse。反序列表。
- -R, --recursive。递归列出子目录中的内容。

- pwd

句法: pwd

功能: 打印当前工作目录的全路径名(绝对路径)。

- cd

句法: cd [-PL] [dir]

功能: 改变当前目录为 dir。

关于 dir 的几点说明如下。

若 dir 默认,则改变当前目录为 \$ HOME 变量的值,其中 \$ HOME 变量存放着当前用户的主目录,即该用户刚登录后所在的目录。

dir 为“.”,代表当前的目录。

dir 为“..”,代表当前目录的父目录。

若 dir 以字符“/”开头,则 dir 指出的是绝对路径;否则 dir 指出的是相对路径,目录应改为从当前目录开始的 dir 目录。

OPTION 的常用选项介绍如下。

- -P。使用物理目录结构,而不使用紧跟着的符号链接。
- -L。强制使用后面的符号链接。
- mkdir

句法: mkdir[OPTION] DIRECTORY...

功能: 创建一个或多个目录。其中子目录名为最多 255 个任意字符(除了字符“/”)的字符串。

OPTION 的常用选项介绍如下。

- -m,--mode=MODE。设置许可模式(形式如 chmod,而不是 umask 中的 rwxrwxrwx)。
- -p,--parents。如果不存在错误,则根据需要创建父目录。
- rmdir

句法: rmdir [OPTION]... DIRECTORY...

功能: 删除一个或多个空的目录。任何一个 DIRECTORY 参数必须是一个空目录,否则命令会指出这不是一个空目录。DIRECTORY 参数支持通配符。

OPTION 的常用选项介绍如下。

-p 或--parents: 如果目录由多个路径名组成,从最后一个路径名开始依次删除,直到所有的路径名都被删完。例如,命令 rmdir -p a/b/c 按照 rmdir /a/b/c、rmdir a/b、rmdir a 的顺序删除目录。

- rm

句法: rm [OPTION]...[FILE]...

功能: 删除命令中所列出的每个文件。默认时,该命令不删除目录。其中 FILE 是要删除的文件。

OPTION 的常用选项介绍如下。

- -f 或--force。忽略不存在的文件,并且无需用户对每个文件确认删除。
- -i 或--interactive。删除每个文件时提示用户是否要删除。
- -r 或-R 或--recursive。以递归方式删除目录中的内容,包括所有文件和子目录。

- cp

句法: cp [OPTION]...SOURCE DEST

功能: 复制一个或多个文件或目录。若命令的最后一个参数 DEST 是一个已存在的目录,则 cp 把每个 SOURCE 文件复制到该目录下(保持原来的文件名不变)。

OPTION 的常用选项介绍如下。

-r 或-R 或-recursive: 该命令默认时不复制目录,复制目录要用该选项。

- mv

句法: mv [OPTION]...SOURCE DEST

功能: 移动或更名文件或目录。若命令的最后一个参数名是一个已存在的目录,则将在 SOURCE 中指定的文件移动到该目录下。

- cat

句法: cat [OPTION] [FILE]...

功能: 该命令包含两项功能,一是在标准输出(通常是屏幕)上显示文件内容,二是连接两个或多个文件,然后通过输出重定向符把合并后的内容存入新的文件。

- chmod

句法: chmod [OPTION]...MODE FILE...

功能: 改变文件(包括目录文件)的访问许可权限

MODE: 权限设定字符串,格式为[ugoa...][[+-=][rwxX]...][, ...],其中 u 表示该档案的拥有者,g 表示与该档案的拥有者属于同一个群体(group)者,o 表示其他以外的人,a 表示这三者皆是。+表示增加权限、-表示取消权限、=表示唯一设定权限。r 为可读、w 为可写、x 为可执行。

- find

句法: find [FILE...] [EXPRESSION]

功能: 查找文件和目录。

- grep

句法: grep [OPTIONS] PATTERN [FILE...]grep [OPTIONS] [-e PATTERN/-F FILE] [FILE...]

功能: 在一个或多个输入文件(若在命令中未给出文件,则是标准输入文件)中查找含有与指定模式相匹配的字符串的行。

OPTION 的常用选项如下。

- -i. 不区分大小写的搜索。默认情况区分大小写。

- -w. 只匹配整个单词,而不是字符串的一部分(如匹配'magic',而不是'magical')

- sort

句法: sort [OPTION]...[FILE]...

功能: 排序给出的所有文件并合并至标准输出。如不指定输入文件或使用“-”,则表

示排序内容来自标准输入。

OPTION 的常用选项如下。

- -o FILE。将合并排序结果输出到 FILE 文件,而不是标准输出。
- -r。倒序排序。
- -d。按字典顺序排序,比较时仅字母、数字、空格和制表符有意义。
- -f。将小写字母与大写字母同等对待。

(2) Linux 文件系统的安装和卸载。

Linux 允许根据分区将使用的文件系统创建成为不同的分区类型。下面是对不同文件系统及其使用方法的简单描述。

EXT2	支持标准 UNIX 文件类型(常规文件、目录和符号链接等)。支持长达 255 个字 符的文件名
EXT3	EXT2 的升级版,可方便地从 EXT2 迁移至 EXT3。主要优点是在 EXT2 的基 础上加入了记录数据的日志功能,且支持异步的日志
ReiserFS	一种新型的文件系统,通过完全平衡树结构来容纳数据,包括文件数据、文件 名以及日志支持。Reiser FS 支持海量磁盘和磁盘阵列,因此搜索速度快且效率 高
JFS	IBM 的 JFS 文件系统提供了基于日志的字节级文件系统,该文件系统是为面 向事务的高性能系统而开发的,与非日志文件系统相比,它的优点是其快速重 启能力
swap	用于支持虚拟内存的交换空间
vfat	是与 Windows 9x/2000/NT 的 FAT 文件系统的长文件名兼容的文件系统

在 Linux 系统中,设备名称通常都在 /dev 目录下。这些设备名称的命名都是有规则的。例如, /dev/hda1 是一个设备名称,其中 hd 即 Hard Disk(硬盘),代表 IDE 硬盘,hd 也可以用 sd 代替,即 SCSI Drive,代表 SCSI 硬盘;a 则是代表第一个设备,通常 PC 上可以连接 4 块 IDE 硬盘,所以 IDE 硬盘对应的设备名称分别为 hda、hdb、hdc 和 hdd。同样,如果使用 SCSI 硬盘,则设备名称为 sda、sdb 等。此外,hda1 代表 hda 的第一个硬盘分区,hda2 代表 hda 的第二个分区,依此类推。

通过查看 /var/log/messages 文件,可以找到 Linux 系统已辨认出来的设备代号。

① 查看和更改现有文件系统分区

- fdisk

句法: fdisk filesystem

功能: fdisk 是各种 Linux 发行版本中最常用的分区工具,可以用来查看现有设备的分区情况及对设备进行分区。输入如 fdisk /dev/sda1 即可进入 fdisk 的欢迎界面,等待用户输入相应参数

主要参数如下。

- n。新增一个分区。

- p。显示分区列表。
- w。保存所做修改后退出。

② 建立文件系统

- mkfs

句法: `mkfs [-V] [-t fstype] [fs-options] filesystem [blocks]`

功能: 在特定的 partition 上建立 Linux 文件系统。

OPTION 的常用选项如下。

- filesystem: 预备检查的硬盘分区, 如 `/dev/sda1`。
- -V。详细显示模式。
- -t。给定文件系统的格式, Linux 的预设值为 EXT2。
- -c。在制作档案系统前, 检查该分区是否有坏轨。
- block。给定 block 的大小。

例子: 在 `/dev/hda5` 上建一个 ext3 的文件系统, 并检查坏块。

```
mkfs -V -t ext3 -c /dev/hda5
```

注意, 只有 root 用户才能建立、安装或卸载文件系统。

③ 安装文件系统

创建文件系统后, 需要使用命令 `mount` 将该文件系统安装到主文件系统中。

Linux 系统中有一个 `/mnt` 目录, 专门用作挂载点 (Mount Point) 目录。建议在实际应用中, 将设备挂装到此目录的子目录中。

- mount

句法: `mount [-afFhnrVw] [-L<label>] [-o<选项>] [-t<fstype>] [device] [mount point]`

功能: `mount` 可将指定设备中指定的文件系统加载到 Linux 目录下 (也就是装载点)。

OPTION 的常用选项如下。

- -t fstype。制定需要安装的文件系统类型, 这与 `mkfs` 中的 -t 选项是一样的。
- -L<label>。加载文件系统标签为 <label> 的设备。
- mount point。通常这是一个空目录名, 并且是专门为安装新文件系统而准备的。

一般来说, 在 Linux 中都将设备挂装到 `/mnt` 目录之下。

例子: 将 `hda1` 上的 ext2 分区挂载到 `/mnt/ext2` 空目录下。

```
mount -t ext2 /dev/hda1 /mnt/ext2
```

④ 卸载文件系统

在关闭系统之前, 为了保证文件系统的完整性, 所有安装的文件系统都必须被卸载。

通常,在/etc/fstab 文件中定义的文件系统都能够自动卸载。但是,手工安装的文件系统,在关闭系统之前必须手工卸载。有时,也需要在系统工作过程中手工卸载某个文件系统。手工卸载文件系统必须使用 umount 命令。

- umount

句法: umount <分区名或分区的安装点>

例子: 将已经安装到/mnt/floppy 目录下的软盘卸载,可以使用以下命令。

```
umount /mnt/floppy
```

注意,对于正在使用的文件系统,不能使用 umount 命令卸载。

三、实验内容——Linux 进程和作业控制

Linux 的进程和作业控制体现了 Linux 多任务的特性。

1. 背景知识

1) 进程和作业

每运行一个程序, Linux 系统就启动一个进程, Linux 把正在运行的程序称为“进程”,也可以称为“作业”。可以分别使用 ps 命令和 jobs 命令来查看进程和作业的信息。如果想要观察动态的进程控制和调度信息,可以使用 top 命令。下面先简单介绍这三个命令的使用。

(1) ps 命令。

格式: ps [Options] [pids]

功能: 显示当前进程的信息。

常用选项可以通过 man ps 命令查看。

例子: 要查看当前进程的信息,可在 Shell 提示符下输入命令 ps -l, 屏幕将显示栏目表。可以观察到,用户 root 当前正在运行的进程是 bash 和 ps 命令自身,即 bash 与 ps 命令同时运行。当 root 输入 ps 命令后, bash 执行该命令, ps 完成运行后,控制返回 bash 进程, bash 显示提示符,等待下一个命令。

(2) jobs 命令。

格式: jobs

功能: 列出当前正在运行的作业信息,该命令没有选项和参数。

(3) top 命令。

格式: top [options]

功能: 主要用于显示系统动态的进程控制和进程调度,此外,还可查看内存动态使用的实时信息。由它提供的内存动态使用信息包括总的可用内存、未占用的内存、已使用的

内存、共享的内存和用于缓存的内存。

常用选项说明如下。

- d。指定屏幕刷新的时间间隔。
- q。使 top 不延迟地刷新,如果调用者拥有超级用户的特权,top 将以最高可能的优先级来运行。
- S。累积模式,每一进程都以 CPU 时间列出。
- s。top 将在安全模式下运行,以避免交互命令的潜在危险。
- i。不理睬任何空闲或僵尸进程就开始运行 top 命令。
- c。显示命令行而不只是命令名。

在输入 top 命令后,top 命令将以全屏方式显示处理器状态等动态信息。默认情况下,5s 更新一次显示。

2) 进程控制

这里介绍的进程控制主要是指运用 Linux 系统提供的命令,撤销进程、改变进程的优先级及将进程切换到后台运行的方法。

(1) 撤销进程。

有两种撤销进程的方法,一种是使用 kill 命令,另一种是使用 Ctrl+C 键。下面先介绍 kill 命令的使用。

格式: kill[options]pid,其中,pid 是进程标识。

功能: 终止(或撤销)一个进程。

常用选项说明如下。

- -s<信号名>。指定发送的信号,信号是以信号名或数字给出的。
- -p。指 kill 只输出已命名的 pid,而且不送信号给它。
- -l。输出一个信号名列表。

如要撤销一个进程,则只需在命令 kill 后输入欲撤销进程的进程标识(pid)即可。kill 命令除可用于撤销进程外,还可用于撤销作业。可利用 Ctrl+C 键来撤销一个当前正在前台运行的进程。

(2) 指定进程的优先级。

nice 命令可以指定进程的优先级,也可以通过该命令来改变一个进程的优先级。下面介绍该命令的使用方法。

格式: nice [options] [COMAMND arg...]

功能: 通过修改优先级来运行一个程序。

常用选项说明如下。

- -n 或 -adjustment=ADJUST。指定程序运行优先级的调整值。
- -help。输出帮助信息。

- --version。输出版本信息。

3) 作业控制

Linux 系统提供了许多命令使用户可以方便地运行和撤销作业、在指定时间运行作业及将作业在前后台之间切换等。下面对这些作业控制的方法进行介绍。

(1) 前台作业和后台作业,以及前后台作业之间的切换。

使一个作业放到后台运行的一种方法是先用 Ctrl+Z 键挂起该作业,然后在 Shell 提示符下输入命令 bg;而要将该作业再次切换回前台运行,则只需在 Shell 提示符下输入命令 fg 即可。如果系统中正在运行多个作业,也可使用命令 fg 和 bg 对这些作业进行前后台切换,但必须在命令后跟上 % 符号和作业号。例如有 3 个作业,3 号作业在前台运行,1、2 号作业在后台运行,如果要将 2 号作业放到前台运行而把 3 号作业放到后台运行,则只需分别输入命令 fg%2 和命令 bg%3 即可。

使一个作业在后台运行的另一种方法是在要运行的程序命令的最后加上 & 符号。

(2) 挂起作业和撤销作业。

作业可以被挂起,挂起作业只是暂时停止该作业的运行,用户可在需要时在前台或后台恢复其运行。恢复被挂起的作业不会改变作业的状态,该作业将从被中断处继续运行。挂起作业的方法是使用 Ctrl+Z 键。撤销作业是指终止该作业的运行。可以用 kill 命令来撤销一个作业,但在命令 kill 后必须先输入 % 符号,再输入作业号。

(3) 指定时间运行作业。

使用 at 命令可以在一个特定的时间执行作业或删除队列中的文件。

格式: at [-v] [-q<队列名>] [-f<文件名>] [-mldbv] TIME

功能: 在指定时间执行作业或删除队列中的文件。

(4) 作业的顺序执行。

用户可以在一个命令行输入多个命令来使作业顺序执行。只需要用分号(;)或与号(&)分开这些命令。系统将按命令行的顺序执行这些作业,并在屏幕上显示所有输出结果。例如,要顺序执行以下三个命令。

- ls-l>file1(将 /root 目录下文件的详细内容输出到 file1 中)。
- cat file1(将 file1 中的内容输出到屏幕上)。
- cp file1 file2(将 file1 中的内容复制到 file2 中)。

可以在 Shell 提示符下输入命令 ls-l>file1;cat file1;cp file1 file2 达到同样目的。

2. 实验指导

(1) 显示系统正在执行的进程的信息。

① 先在后台运行一个作业 yes>/dev/null,即在 Shell 提示符下输入命令 yes>/dev/null&。

问题 1: 请记录该作业在你机器上的进程号。

② 输入 `ps -e` 命令显示系统正在执行的所有进程的信息。

问题 2: 请记录当前 `bash` 的进程号

(2) 在前台启动一个作业并将它放到后台运行。

① 在前台启动一个作业 `yes>/dev/null`, 在 Shell 下输入命令 `yes>/dev/null`, 按 `Ctrl+Z` 键, 挂起该前台作业。

② 在 Shell 下输入命令 `fg %2`, 将作业[2]重新放到前台运行。

③ 使用 `Ctrl+Z` 键再次将前台作业挂起。

④ 输入命令 `bg`, 将该作业放到后台运行。

问题 3: 请记录每一步操作完之后 `bash` 的显示结果。

(3) 撤销正在运行的作业。

① 输入命令 `ps`, 显示正在执行的进程的信息。

② 输入命令 `kill %1`, 撤销正在运行的作业[1], 再输入 `jobs` 命令查看运行结果。

③ 输入命令 `kill <code>PID</code>`, 撤销正在运行的作业[PID], 再用 `jobs` 命令显示进程的运行状态。例如某个正在运行作业的 PID 为 2296, 就可以使用 `kill 2296` 来撤销该作业。

④ 输入命令 `ps`, 显示正在执行的进程的信息。

问题 4: 请记录每一步执行完之后 `bash` 的输出结果。

(4) 定时启动一个作业并查看作业运行的结果。

① 输入重定向命令 `cat>work` 建立作业文件, 输入要启动的作业命令, 按 `Ctrl+D` 键结束键盘输入, 其操作过程如下。

```
[root@localhost/root]# cat>work
ls -l
date
[root@localhost/root]#
```

② 输入命令 `date` 显示当前时间, 然后输入命令 `at -f work. 0810+5 minutes`, 表明 5 分钟后运行 `work` 文件中的作业。

③ 以交互方式从键盘输入命令, 并在 5 分钟后运行作业, 其操作过程如下。

```
[root@localhost/root]# at 08: 15
at>ls -l
at>date
<Ctrl>+D
at>< EOT>                                (屏幕显示)
[root@localhost/root]# at -l
12 2002-06-06 08: 15 a                    (屏幕显示)
```

13 2002-06-06 08:15 a (屏幕显示)

(5) 删除上面的 13 号作业。

① 输入命令 `at -d 13`, 删除用户排在队列中的 13 号作业, 然后输入命令 `at -l` 以查看队列中的作业。

```
[root@ localhost /root]# at -d 13
```

```
[root@ localhost /root]# at -l
```

```
12 2002-06-06 08:15 a
```

② 查看作业运行的结果。

当时间进行到 2007/8/6 08:15 时, 屏幕显示信息如下。

```
you have a mail in /var/spool/mail/root
```

问题 5: 请写出定时作业的运行结果。

(6) 使用 `top` 命令。

① 分别用两种方法改变 `top` 的更新延迟时间为 10s, 并仔细观察处于全屏幕交互状态时 `top` 命令的输出信息。

② 使用 `top` 的交互命令实现只显示系统当前优先级最高的前 5 个进程的信息, 然后恢复到默认状态。

③ 使用 `top` 的交互命令移去 `SIZE` 字段并增加 `DSIZE` 字段的显示。

④ 使用 `top` 的交互命令隐藏进程和 CPU 状态显示, 然后恢复。

⑤ 使用 `top` 的交互命令以驻留内存使用程序对进程进行排序, 然后恢复到默认状态。

⑥ 使用 `top` 的交互命令改变 `top` 进程的优先级, 输入 `nice` 的值为 6, 观察 `top` 进程的 `NI` 字段值及 `PRI` 字段值的变化。

⑦ 使用 `top` 的交互命令撤销一个进程并观察 `top` 输出信息的变化。

问题 6: 查阅资料并写出每一步的命令。

3. 练习与思考

(1) 用 `top` 命令查看系统的进程调度动态信息, 注意体会交互命令的功能及各字段的含义。

(2) 什么是前台和后台? 什么是前台作业和后台作业?

(3) 将作业放到后台运行有哪两种方法? 如何操作?

(4) 试叙述调整进程优先级的两种方法。

(5) 试叙述撤销作业和挂起作业的区别, 其具体的操作方法是什么?

四、实验内容——GNU C 编程工具的使用

GNU C 的编译器是 gcc。gcc 软件包支持 C、C++ 和 Objective C。gcc 的可执行文件在 /usr/bin/gcc 和 /usr/lib/gcc-lib/target/version 目录下, /lib 和 /usr/lib 目录下是库文件, /usr/include 目录下是头文件。

1. 使用 gcc 编译和链接

gcc 编译 C 程序要经过预处理、编译、汇编和连接 4 个步骤。gcc 的基本用法如下。

```
gcc [option] [filename]
```

在 gcc 后面可跟多个编译选项,同时执行多个操作。gcc 有一百多个编译选项,但大多数一般不会用到,这里介绍主要的常用选项。如果了解更多的选项和更具体的说明,可以参考 gcc 的联机帮助,输入 man gcc 即可。

例如,当需要将文件 test.c 编译成 test.o 文件时,输入 gcc -c test.c 即可。

gcc 的主要选项如表 1 所示。

表 1 gcc 的常用命令

功 能	描 述
-x language	选择语言
-c	只预处理、编译和汇编源程序,产生目标代码,不进行链接
-S	只对文件进行编译,不汇编和链接
-E	只对文件进行预处理,不编译、汇编和链接
-o [file1] file2	将 file1 作为输出文件的文件名,如果没有 -o,则默认的可执行文件名为 a.out,目标文件名为 file2.o,汇编文件名为 file2.s
-L library	指定使用的库文件
-I directory	将 directory 目录加到头文件的目录列表中,且优先搜索。可以有多个 i 选项
-W	禁止警告信息
-Wall	显示附加的警告信息
-g	产生用于调试和排错的扩展符号表,供 gdb 调试用。不能和优化选项同时用
-p	产生 prof 所需的信息
-pg	产生 gprof 所需的信息
-O	对编译进行优化,使执行程序更短,执行速度快,需要更多的时间和内存
-O2	比 O 选项更好的优化
-v	显示 gcc 的版本

2. 使用 make 和 makefile 实现自动编译

当一个系统包含多个模块时,多个模块之间存在各种依赖关系,例如一个模块要调用另外多个模块中的函数或引用其中定义的变量。此时要生成最后的可运行文件,就需要按照一定的顺序编译连接各个模块。另外,当其中的某些模块被改动,与其有依赖关系的模块也需要重新编译连接。因此,要维护复杂一些的系统的工作也是很复杂的。

Linux 提供了 make 命令和 makefile 文件的方式来维护目标程序。make 是个独立工作的程序,它按照 makefile 中描述的模块依赖关系,根据模块程序的修改情况重新编译链接目标代码,实现维护工作的自动化。

makefile 文件中的基本单位是目标(target),每一个目标包括两部分:该目标依赖的文件或模块,以及实现该目标的一组命令。

格式是:每个目标后跟一个“:”,其依赖的文件或其他目标跟在后面,以空格隔开。另起一行,输入实现这一目标的 shell 命令。在命令前的第一个输入必须是<tab>键,不能以空格开头。Shell 命令可以有多行,但后续的行必须以<tab>开头。如果要把一行命令分几行输入,则以反斜杠(\)连接。在 makefile 中的文件名可以使用“*”和“?”通配符。

3. 使用 make 命令

建立 makefile 后,就可以调用 make 命令生成和维护目标文件。make 命令的格式如下:

```
make [option] [-macrodef] [target]
```

option 指定 make 的工作方式,macrodef 指定执行 makefile 时的宏值,target 是需要更新的目标。具体的用法是用 man make 命令,参见在线帮助或有关参考书。

当 make 单独使用时,默认的文件名是 makefile,也可以使用自己的文件名,格式为:make -f makefile -name 或 -file=filename 或 -makefile=filename。如果没有指定 target,make 文件会从文件的第一个目标开始维护。检查它所依赖的文件或目标,如果是目标,则继续检查该目标的文件,直至所有的依赖文件都是源文件,然后再沿原路径上溯,检查是否目标所依赖的文件已被更新,如果目标文件比依赖文件建立时间早,则算过期,需要重新生成。如果所有文件都不需要更新,则 make 给出提示“目标文件已经是最新的”。

4. gdb 调试工具

gdb 是一个 GNU 调试程序,可以用来调试 C 和 C++ 程序。它的主要功能如下。

(1) 监视程序中变量的值。

(2) 设置断点。

(3) 单步执行程序。

gdb 的启动:

```
gdb filename
```

也可以没有 filename,而在进入 gdb 后再通过 file 命令装入文件。

注意:为了使 gdb 能正常工作,需要在编程序时,打开调试选项,命令为 gcc -g filename。

启动后,gdb 的提示为:

```
(gdb)
```

此时可以输入命令了。

下面给出部分常用命令和选项的分类及汇总。

(1) 断点命令如表 2 所示。

表 2 断点命令

功 能	命 令 语 法	备 注
设置断点	break 设置断点的行号	
设置临时断点	tbreak 设置断点的行号	
清除断点	clear 断点的行号	
清除断点或自动表达式	delete 断点编号或表达式	
设置有条件的断点	condition 断点编号 条件表达式	
让所设断点暂时失效	disable 断点编号	使多个断点失效,用空格分开
恢复暂时失效的断点	enable 断点编号	

(2) 监视命令如表 3 所示。

表 3 监视命令

功 能	命 令 语 法	备 注
设置要监视的表达式	watch 要监视的表达式	
设置要监视的表达式	awatch 要监视的表达式	当表达式的值变化或被读取时,程序暂停,显示表达式的值
设置要显示的表达式	display 表达式	当程序运行到断点时,显示该表达式的值
查询表达式的类型	whatis 表达式	

续表

功 能	命 令 语 法	备 注
打印	print 表达式 print \$ 符号 print \$ \$ 符号 print 开始表达式@连续 内存空间大小	打印表达式的值 打印给定符号的前 1 个符号的值 打印给定符号的前 2 个符号的值 打印内存中某一连续空间的值,主要用于打印 数组类型的表达式的值

(3) 文件命令如表 4 所示。

表 4 文件命令

功 能	命 令 语 法	备 注
加载调试文件	file 文件名	
列出文件内容	list [参数]	参数指示开始显示的行。 行号:从当前文件的该行开始 文件名 行号:从指定文件的指定行开始 函数名:显示指定的函数 文件名 函数名:指定文件的指定函数 * 地址:显示包含该地址的文件 行号 1 行号 2:从行号 1 显示到行号 2
向后查找字符串第一个 所在的行	forward 字符串	查找到的行号保存在 \$ _ 中,用 print \$ _ 显示
同上	search 字符串	
向前查找字符串第一个 所在的行	reverse-search 字符串	

(4) 程序控制命令如表 5 所示。

表 5 程序控制命令

功 能	命 令 语 法	备 注
运行程序	run	
继续执行	cont [N]	N:忽略断点的次数是 N-1
继续运行,不进入子程序	next [N]	N:执行该操作的次数 执行后,显示下一条执行的指令
单步运行,进入子程序	step [N]	N:同上 执行后,显示下一条执行的指令

(5) 堆栈及信号命令如表 6 所示。

表 6 堆栈及信号命令

功 能	命 令 语 法	备 注
显示堆栈内容	backtrace 堆栈层数	可以用来显示函数调用关系
设置信号处理函数	handle 信号 信号处理函数	信号用符号或数字表示 all:表示除 SIGTRAP 和 SIGINT 之外的所有信号 信号处理函数如下。 stop:重新进入调试器,显示提示信息 print:显示提示信息 pass:传递给调试程序 ignore:pass 和 stop 的组合

实验2 剪裁内核

一、实验目的

1. 了解 Linux 源代码的获取方式。
2. 学习 Linux 内核的配置方法。

二、实验内容

下载新版本的内核源码,在自己的机器上解压缩,观察内核源代码的组成结构。然后自行根据需要配置并编译这个版本的内核源码,并尝试用编译出的内核重新引导系统。

三、准备知识

1. Linux 内核源代码的获取

Linux 内核源代码的官方提供网站只有一个: <http://www.kernel.org/>。

在下载网页上,有两种类型的代码包: `linux-*.*.*.tar.gz` 和 `linux-*.*.*.tar.bz2` (*.*.* 代表的是内核版本,如 Linux 2.6.18)。两个代码包的内容是一样的,只是压缩程序不同。gz 是用 gzip 压缩的,而 bz2 是用 bzip2 压缩的。bzip2 的压缩能力比 gzip 强。

Linux 的下载工具有 gftp, kget 和 wget 等。命令行的 wget 在开发过程中使用比较方便,它既支持 FTP,也支持 HTTP,还支持断点续传。例如,用 wget 下载 Linux 2.6.18 内核的命令如下, -c 选项表示使用断点续传。

```
$wget -c http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.18.tar.bz2
$wget http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.18.tar.bz2.sign
```

下载完后,可先验证一下内核源码包的电子签名:

```
$gpg -verify linux-2.6.18.tar.bz2
```

如果没有问题,就可解压内核压缩包了。命令如下:

```
$tar jxvf linux-2.6.18.tar.bz2
```

如果是 gzip 格式的源代码包,则使用如下解压命令:

```
$tar zxvf linux-2.6.18.tar.gz
```

解压后可参照第 1 章介绍的 Linux 内核源代码的目录结构,总体浏览一下。注意,通常每个有源码文件的目录下都有 Makefile 和 Kconfig(2.6 内核,2.4 用 config.in)两个文件,用于组织内核的配置和编译。顶层目录下的 Makefile 是整个内核配置和编译的总体控制文件,负责启动配置工具、组织子目录编译。另外,还可设置目标 CPU 的体系结构和交叉编译命令等。

如果是初次接触 Linux 内核源代码,可通过顶层目录下的 readme 文件了解相关的内核介绍和编译命令。但需注意该文件主要是针对 i386 平台的,对其他的体系结构不一定完全适用。

2. Linux 内核的配置方式

Linux 内核源码支持 20 多种处理器架构,还有各种各样的驱动程序等配置选项,所以在内核编译前,针对目标平台的配置工作相当复杂。为此,Linux 内核源代码专门提供了一个配置系统。

为了将讨论的问题明确化,而且对内容本身不产生影响,在本章下面的论述中,凡是涉及到与具体 CPU 体系结构相关的内容,都以 ARM 为例。

Linux 内核的配置系统由三个部分组成,Makefile、配置文件和配置工具。

- Makefile。分布在 Linux 内核源代码中,定义 Linux 内核的编译规则。
- 配置文件(config.in/Kconfig)。(通过配置界面)给用户提供了配置选择的功能,在 Linux 2.4 版以前文件名是 config.in,在 Linux 2.6 版以后文件名改为 Kconfig。
- 配置工具。包括配置命令解释器和配置用户界面。其中配置命令解释器负责对配置脚本中使用的配置命令进行解释;配置用户界面为用户提供配置内核选项的使用界面,它有三种:make config 提供基于字符界面;make menuconfig 提供基于 Ncurses 图形界面;make xconfig 提供基于 X windows 图形界面的用户配置界面。

位于顶层目录下的 Makefile 是整个配置系统的总体控制文件。它不仅通过控制各子目录下的 Makefile 来定义 Linux 内核的编译规则,而且还允许使用者通过 make 命令来启动配置工具来对繁杂的内核选项进行可视化的设置。

配置工具按照配置文件的内容为用户提供各种配置选项,用户通过配置工具界面来对各选项分别进行设置。设置的结果保存在顶层目录的 .config 文件中,同时,配置工具也会根据设置的结果产生一些头文件和符号连接,这将在真正编译内核时决定 Makefile 中的相关变量值,进而决定所生成的内核组成。

Linux 内核配置系统支持多种内核配置方式,有命令行方式,也有图形界面方式,下面分别予以介绍。在实际配置时,选用其中的一种即可。

(1) make config。该命令为文本方式。通过命令接口,依次提示每个选项让用户设

置。如果, config 文件存在, 它会根据该文件来确定提示时的默认值。

(2) make oldconfig。该命令为文本方式。通过命令接口, 对, config 文件中已经设定的选项不再提示用户选择。只有当碰到, config 文件中有先前没有确定的选项时才提示用户手工设定。注意它与上面 make config 的区别。make config 会要求用户手动设定所有的选项, 即使以前已经设定过。

(3) make menuconfig。该命令显示以光标为基础的、终端式的配置菜单。如果, config 文件存在, 它会据此来设定提示时的默认值, 这一点与 make config 是一样的。

(4) make xconfig。该命令显示以 Tk 为基础的 X Window 配置菜单。如果, config 文件存在, 它会据此来设定提示时的默认值, 这一点与 make config 和 make menuconfig 是一样的。

对已经设置好的配置选项, 如果需要进行变更, 但又不想丢失已有的配置, 可以通过手工备份, config 文件实现。具体的做法如下。

- ① 将, config 文件复制到一个可靠的地方, 如果需要可更改文件名。
- ② make menuconfig 修改配置并编译使用。
- ③ 需要时, 可以将在第①步中备份的, config 文件复制回内核源码顶层目录, 注意名称不得改变。
- ④ 通过 make oldconfig 命令恢复原来的配置。

四、Linux 内核的配置选项解析

前面介绍的 4 种方式都可以用来设置内核, 但由于 make menuconfig 方式的光标菜单运行时不依赖于 X11 的图形软件环境, 可以运行在字符终端上, 所以一般开发者通常都会选用这种方式。

在各级子菜单项中, 当选择相应的配置时, 通常有三种可能的选择, 其含义如下。

- Y。将该功能编译进内核。
- N。不将该功能编译进内核。
- M。将该功能编译成内核模块, 在需要时再动态加载到内核。

在每一个选项前面都有一对括号: []、< > 或者是 ()。在光标选中选项时, 可通过空格键改变括号中的内容。[] 中要么是空, 要么是“*”, 而< > 除了是空和“*”外, 还可以是 M。空表示 N(不将该功能编译进内核), “*”表示 Y(将该功能编译进内核), 而 M 则表示将该功能编译成内核模块。() 表示要按 Enter 键后在出现的新界面进行输入或选择。

内核配置是否适当, 将直接关系到 Linux 系统是否能够高效地运行。因此, 内核配置的需要根据具体情况具体分析。对各选项的选择原则是将与内核其他部分关系较远且不经常使用的功能编译为内核模块, 这有利于减小内核运行时的内存需求, 并减少该功能的

环境设置对内核的影响;对不需要的功能要去掉;与内核关系紧密而且经常使用的功能要直接编译进内核,以提高运行效率。

随着内核功能的不断扩充,内核的配置选项也在不断的发生变化。相对 Linux 2.4 内核来说, Linux 的 2.6 内核的配置选项有了较大改变。就是同为 Linux 2.6 内核,各版本之间的配置选项也会有些变化。下面为 Linux 2.6.18 内核的一些选项。

(1) Code maturity level options(代码成熟度选项)。

本菜单只包含了一个用于配置代码成熟度的选项。

Prompt for development and/or incomplete code/drivers [Y/n/?]

默认情况下是选择的,这将会在设置界面中显示正在开发或者还没有完成的代码与驱动。大多数情况下,应该选择它,因为有许多设备需要选择这个选项才能进行配置。

(2) General setup(常规设置)。

本菜单主要包含一些通用的配置选项。

Local version-append to kernel release (LOCALVERSION) [2.6.18]

该选项在内核版本后面加上自定义的版本字符串(小于 64 字符),可以用 `uname -a` 命令看到。

Automatically append version information to the version string (LOCALVERSION_AUTO) [Y/n/?]

默认情况下是选择的,自动在版本字符串后面添加版本信息,编译时需要有 perl 及 git 仓库支持。

Support for paging of anonymous memory (swap) (SWAP) [Y/n/?]

默认情况下是选择的,选择该选项将打开虚拟内存的页交换(swap)功能。

System V IPC (SYSVIPC) [Y/n/?]

默认情况下是选择的,该选项将为进程提供通信机制,这将使系统中各进程间有交换信息与保持同步的能力。嵌入式系统一般都是多任务的,所以一定要选。

POSIX Message Queues (POSIX_MESSAGE) [Y/n/?]

默认情况下是选择的,它提供 POSIX 标准的消息队列通信方式,这是 POSIX IPC 中的一部分,应该选择。

BSD Process Accounting (BSD_PROCESS_ACCT) [Y/n/?]

默认情况下是选择的,将进程的统计信息写入文件的用户级系统调用,主要包括进程的创建时间、创建者和内存占用等信息。嵌入式系统中通常不需要。

BSD Process Accounting version 3 file format (BSD_PROCESS_ACCT_V3) [N/y/?]

使用新的第三版文件格式,可以包含每个进程的 PID 和其父进程的 PID,但是不兼容旧版本的文件格式。一般不选。

Auditing support (AUDIT) [Y/n/?]

默认情况下是选择的,用于资源审核的,某些内核模块(如 SELinux)需要它,只有同时选择其下面的子项才能对系统调用进行审计。一般不选。

Enable system-call auditing support (AUDITSYSCALL) [Y/n/?]

默认情况下是选择的,用于系统调用的审核,一般不选。

Kernel .config support (IKCONFIG) [N/y/?]

把内核的配置信息与相关的文档说明编译进内核中,以后可以通过 scripts/extract-ikconfig 脚本来提取这些信息。

Cpuset support (CPUSETS) [Y/n/?]

创建和管理 CPUSETs,主要对大型 SMP 或者 NUMA 系统有用。

Initramfs source file(s) (INITRAMFS_SOURCE) []

initrd 已经被 initramfs 取代,可能通过选择该选项指定 initramfs 的源文件。

Optimize for size (Look out for broken compilers!) (CC_OPTIMIZE_FOR_SIZE) [Y/n/?]

默认情况下是选择的,编译时优化内核尺寸(使用-Os 而不是-O2 参数编译),对嵌入式系统可能有用。

(3) Configure standard kernel features (for small systems)。

为小型系统配置标准的内核特性。一般不需要。

(4) Loadable module support(可加载模块支持)。

本菜单包含对内核动态加载模块的一些配置选项。

Enable loadable module support (MODULES) [Y/n/?]

默认情况下是选择的,用以打开对可加载模块的支持,通常要选择。

Module unloading (MODULE_UNLOAD) [Y/n/?]

默认情况下是选择的,允许卸载已经加载的模块。通常要选择。

Forced module unloading (MODULE_FORCE_UNLOAD) [N/y/?]

允许强制卸载正在使用中的模块,对系统有危害,一般不选。

Module versioning support (MODVERSIONS) [Y/n/?]

默认情况下是选择的,允许使用其他内核版本的模块。

Source checksum for all modules (MODULE_SRCVERSION_ALL) [Y/n/?]

为所有的模块校验源码。一般不选。

Automatic kernel module loading (KMOD) [Y/n/?]

默认情况下是选择的。当内核在某些任务中要使用一些被编译为模块的驱动或特性时,需要先使用 modprobe 命令来加载它,然后内核才能使用。如果选择了这个选项,在内核需要一些模块时它可以自动调用 modprobe 命令来加载需要的模块,这样内核就可以自动解决模块的依赖关系。因此,通常应该选择。

(5) Block layer(块设备层)。

本菜单用于对块设备层的配置。

Support for Large Block Devices (LBD) [Y/n/?]

如果想给自己的机器配上大的磁盘或希望有个 raid 或 loopback 设备,则选择。

Support for tracing block io actions (BLK_DEV_IO_TRACE) [Y/n/?]

默认情况下是选择的。能够跟踪给定序列的块设备层的行为,并观察在一个块设备序列中发生的任何阻塞。

Support for Large Single Files (LSF) [Y/n/?]

可以处理非常大的文件(大于 2TB)。

(6) IO Schedulers。

本菜单用于对大型块设备中设备层的 I/O Schedulers(I/O 调度器)配置。

Anticipatory I/O scheduler (IOSCHED_AS) [Y/n/m/?]

默认情况下是选择的。适用于大多数环境,但不太适合数据库应用。

Deadline I/O scheduler (IOSCHED_DEADLINE) [Y/n/m/?]

默认情况下是选择的。通常与 Anticipatory 相当,但更简洁小巧,更适合于数据库应用。

CFQ I/O scheduler (IOSCHED_CFQ) [Y/n/m/?]

默认情况下是选择的。为所有进程分配等量的带宽,适合于桌面多任务及多媒体应用。

() Default I/O scheduler

1. Anticipatory (DEFAULT_AS)
2. Deadline (DEFAULT_DEADLINE)
3. CFQ (DEFAULT_CFQ)
4. No-op (DEFAULT_NOOP)

默认 IO 调度器为 Anticipatory。可以指定为以上四者之一,也可以设为空。

(7) Processor type and features(处理器类型及特性)。

本菜单包含系统平台的列表及其相关的配置选项。对于选择不同的体系结构,会显示不同的提示信息。

Symmetric multi-processing support (SMP) [Y/n/?]

提供对多 CPU 系统的支持。如果是单 CPU 的机器,选择 N,内核会运行得快些。如果是多 CPU 的机器,且这里选择了 Y,则下面 Enhanced Real Time Clock Support 的选项也需要选择 Y。

Generic x86 support (X86_GENERIC) [Y/n/?]

不仅包含最优的已选 x86 变量,也包含了一些普通的最优化。这将使得内核在 x86 CPU 上运行的更好。

HPET Timer Support (HPET_TIMER) [Y/n/?]

支持内核的内置计时器对 HPET 的使用。

Maximum number of CPUs (2-255) (NR_CPUS) [32]

描述内核所能支持的 CPU 的最大数量(2~225 之间)。

SMT (Hyperthreading) scheduler support (SCHED_SMT) [Y/n/?]

当处理带有 HyperThreading 的 Intel Pentium 4 芯片时,可提高 CPU 调度中决策的制定。

Multi-core scheduler support (SCHED_MC) [Y/n/?]

当处理多核 CPU 芯片时,可提高 CPU 调度中决策的制定。

Preemption Model

1. No Forced Preemption (Server) (PREEMPT_NONE)
 2. Voluntary Kernel Preemption (Desktop) (PREEMPT_VOLUNTARY)
 3. Preemptible Kernel (Low-Latency Desktop) (PREEMPT)
- choice[1-3]:

优先级模型选项,可从上面选择其一。

Preempt The Big Kernel Lock (PREEMPT_BKL) [Y/n/?]

该选项通过锁上大内核优先权而缩短内核的反应时间。

若正在为一个 desktop 系统创建内核,则选择 Y;若不确定,则选择 N。

Machine Check Exception (X86_MCE) [Y/n/?] ?

当处理器发现问题时,允许它通知内核。

Toshiba Laptop support (TOSHIBA) [M/n/y/?]

该选项可以增加一个驱动,该驱动可以通过东芝 BIOS 安全进入东芝笔记本电脑 CPU 的系统管理模式。

Dell laptop support (I8K) [M/n/y/?] ?

该选项可以增加一个驱动,该驱动可以安全进入 Dell 笔记本电脑 CPU 的系统管理模式。

Enable X86 board specific fixups for reboot (X86_REBOOTFIXUPS)[N/Y?]

该选项使得芯片集或主板能够进行修订以重启后可以正常工作。一般不选。

(8) Firmware Drivers(硬件驱动)。

该菜单包含了硬件驱动配置的一些选项。

BIOS Enhanced Disk Drive calls determine boot disk (EDD) [M/n/y/?]

该选项使得 BIOS 能提高磁盘驱动服务真实模式的 BIOS,决定用哪个磁盘 BIOS 来尝试启动。该信息将被输出。

EFI Variable Support via sysfs (EFI_VARS) [Y/n/m/?]

该选项可以得到 EFI(可扩展的硬件界面)的变量信息,且可以通过该界面读、写或者创建 EFI 变量。

BIOS update support for DELL systems via sysfs (DELL_RBU) [M/n/y/?]

如果是 DELL 的系统,该选项选择 M 可以出现 BIOS 更新的选项。

Dell Systems Management Base Driver (DCDBAS) [M/n/y/?]

DELL 系统管理基本驱动可在一定的 DELL 系统上,为系统管理软件在 SMI 和主机控制行为提供界面。

High Memory Support

1. off (NOHIGHMEM)

2. 4GB (HIGHMEM4G)

3. 64GB (HIGHMEM64G)

choice[1-3]:

高内存支持,从以上三项中根据具体情况任选其一。

64 bit Memory and IO resources (EXPERIMENTAL) (RESOURCES_64BIT) [Y/n/?]

使得内存和 IO 资源成为 64 位。

Allocate 3rd-level pagetables from highmem (HIGHPTE) [Y/n/?]

该选项将使得用户空间页表达到高内存以提高内存利用率。

Math emulation (MATH_EMULATION) [N/y/?]

Linux 可以仿效一个数学处理器,用于浮点运算。486DX 和奔腾处理器都有内置的数学处理器,而 486SX 和 386 则没有。如果机器本身没有或者用户不确定的话,则选择 Y。

MTRR (Memory Type Range Register) support (MTRR) [Y/n/?]

如果在 PCI 或 AGP 总线上有一个 VGA 卡。一般会选择。

Boot from EFI support (EFI) [Y/n/?]

该选项可以使得内核通过从硬件发送过来的系统配置信息在 EFI 平台上启动,这也同时使得内核可以使用任何有效的 EFI 运行时服务。

Use register arguments (REGPARM) [Y/n/?]

使用-mregparm=3 编译内核。这种命令 gcc 使用一个更为有效的函数来调用 ABI,一般会选择。

Enable seccomp to safely compute untrusted bytecode (SECCOMP) [N/y/?]

一般会选择。只有嵌入式系统不选择。

Timer frequency

1. 100 HZ (HZ_100)

2. 250 HZ (HZ_250)

3. 1000 HZ (HZ_1000)

choice[1-3]:

时钟频率,从以上三个选项中选择其一。

kexec system call (EXPERIMENTAL) (KEXEC) [Y/n/?]

系统调用 kexec 能够关闭当前的内核而开启另一个。很像重启,但是又不依赖于硬件。可以开启任何的内核而不仅仅是 linux。

kernel crash dumps (EXPERIMENTAL) (CRASH_DUMP) [N/y/?]

通过 kexec 开启后产生碰撞堆 crash dump。

Support for hot-pluggable CPUs (EXPERIMENTAL) (HOTPLUG_CPU) [Y/n/?] ?

通过关闭和开启 CPU 使得能够在 SMP 系统上挂起。CPU 通过 /sys/devices/system/cpu 来控制。

Compat VDSO support (COMPAT_VDSO) [N/y/?] ?

移除高映射的 VDSO 而使用随机 VDSO。如果现在正在运行 Linux 2.3.3 或其之后的版本,则选择 N。

(9) Power management options(电源管理选项)。

本菜单适用于配置电源管理相关的选项。电源管理能使计算机在不用的时候进入“睡眠”,以节约电能。ARM 平台下主要包括电源管理的支持和高级电源管理仿真。

Power Management support (PM) [Y/?]

电源管理有 APM 和 ACPI 两种标准,且不能同时使用。即使关闭该选项,X86 上运行的 Linux 也会在空闲时发出 HLT 指令使 CPU 进入睡眠状态。

Legacy Power Management API (PM_LEGACY) [Y/n/?]

传统的电源管理 API,如软关机和系统休眠等接口。

Power Management Debug Support (PM_DEBUG) [N/y/?]

仅供电源管理调试时使用。

(10) ACPI (Advanced Configuration and Power Interface) Support。

本菜单包含对高级配置和电源界面的支持选项。

ACPI Support (ACPI) [Y/n/?]

ACPI 为需要 ACPI 平台的 Linux 提供支持,并且采用现有的操作系统配置和电源管理软件。这将使得内核增加 70KB。

(11) APM(Advanced Power Management)BIOS Support。

本菜单包含对高级电源管理 BIOS 的支持选项。

APM (Advanced Power Management) BIOS support (APM) [Y/n/m/?]

APM 是一个为节约电量而使用一些不同技术的 BIOS 的规范。这对具有 APM BIOS 的使用电池电源的笔记本电脑很有用。

(12) CPU Frequency scaling。

本菜单包含 CPU 频率度量的选项。

CPU Frequency scaling (CPU_FREQ) [Y/n/?] ?

CPU 频率度量允许用户在 CPU 运转时改变时钟速度,这是一种节约电量的好方法。
因为 CPU 使用速度越低,CPU 用电量越少。

CPUFreq processor drivers

CPU 频率处理器驱动。

shared options

共享选项。

(13) Bus options (PCI,PCMCIA,EISA,MCA,ISA)(总线支持)。

本菜单包含设定总线的支持选项。

PCI support (PCI) [Y/n/?]

找出是否有一个 PCI 主板。

ISA support (ISA) [Y/n/?] ?

找出在主板上是否有 ISA 插槽。

MCA support (MCA) [N/y/?] ?

MCA 存在于一些 IBM PS/2 机器和笔记本电脑上。是一种类似于 PCI 或 ISA 的总线系统。

NatSemi SCx200 support (SCx200) [N/m/y/?] ?

该选项提供对 AMD 的 Geode 处理器的基本支持。

(14) PCCARD (PCMCIA/CardBus) support。

本菜单包含对 PCCARD 的支持选项。

PCCard (PCMCIA/CardBus) support (PCCARD) [Y/n/m/?]

如果希望对 Linux 计算机配置 PCMCIA- or PC-cards,则选择 Y。

PC-card bridges

PC-card 桥。

(15) PCI Hotplug Support。

本菜单包含对 PCI Hotplug 的支持选项。

Support for PCI Hotplug (EXPERIMENTAL) (HOTPLUG_PCI) [Y/n/m/?] ?

如果主板有一个 PCI Hotplug 控制器,则此处选 Y,这将在机器充电和运行时,允许增加或删除 PCI 卡。

ACPI PCI Hotplug driver (HOTPLUG_PCI_ACPI) [y/M/n/?] ?

如果系统支持 PCI Hotplug 使用 ACPI,则此处选择 Y。如果想把该驱动编译成模块,则选择 M,若不确定则选择 N。

CompactPCI Hotplug driver (HOTPLUG_PCI_CPCI) [N/y/?] ?

如果系统通过 PICMG 2.1 规格说明书支持 CompactPCI hotswap 的系统卡,则此处选择 Y,如果不确定则选择 N。

SHPC PCI Hotplug driver (HOTPLUG_PCI_SHPC) [N/m/y/?] ?

如果主板有 SHPC PCI Hotplug 控制器,则此处选择 Y。如果想把该驱动编译成模块,则选择 M,若不确定则选择 N。

(16) Executable file formats.

本菜单包含对可扩展文件格式的选项。

Kernel support for ELF binaries (BINFMT_ELF) [Y/n/?] ?

ELF(可扩展可链接格式)是一个为通过被不同技术和操作系统所使用的库准备的。选择 Y 可以使得内核运行 ELF 二进制文件并使之增大 13KB。目前基本已代替传统的格式 QMAGIC 和 ZMAGIC。

Kernel support for a.out and ECOFF binaries (BINFMT_AOUT) [N/m/y/?] ?

提供对 a.out 和 ECOFF 二进制格式的支持。

Kernel support for MISC binaries (BINFMT_MISC) [Y/n/m/?] ?

提供对 MISC 二进制格式的支持。

(17) Networking(网络)。

本菜单包括网络协议及其相关配置选项。

Networking options

网络选项。

Network packet debugging (NETDEBUG) [N/y/?]

在调试不合格的包时加上额外的附加信息,通常不选。

Packet socket (PACKET) [Y/n/m/?]

这种 Socket 可以让应用程序(如 tcpdump、iptables 等)直接与网络设备通信,而不通过内核中的其他中介协议。

Packet socket: mmaped IO (PACKET_MMAP) [Y/n/?]

让 Packet socket 驱动程序使用 IO 映射机制以使连接速度更快。

Unix domain sockets (UNIX) [Y/n/m/?]

一种仅运行于本机上的效率高于 TCP/IP 的 Socket, 简称 UNIX socket。可用来进行进程间通信 (Inter Process Communication, IPC)。

IPsec user configuration interface (XFRM_USER) [Y/n/m/?]

为 Ipsec (可在 IP 层加密) 之类的工具提供用户配置接口。

PF_KEY sockets (NET_KEY) [M/n/y/?]

用于可信任的密钥管理程序和操作系统内核内部的密钥管理进行通信, IPsec 依赖于它。

TCP/IP networking (INET) [Y/?]

TCP/IP 协议, 应该选择。

下面是很多关于 IP 组播、高级路由和策略路由等的设置, 可根据需要选择。

The IPv6 protocol (IPv6) [M/n/y/?]

关于 IPv6 协议的设置。后面也有很多具体的选项。

Network packet filtering (replaces ipchains)

网络包过滤器。

Network packet filtering (replaces ipchains) (NETFILTER) [Y/n/?]

网络过滤器是为过滤和破坏通过 Linux 盒的网络包的框架。一般要选。

IP: Netfilter Configuration

IP 网络过滤器配置。

Connection tracking (required for masq/NAT) (IP_NF_CONNTRACK) [M/n/y/?]

保存通过机器的网络包的记录, 以找出它们是怎样关系到连接的。

IP tables support (required for filtering/masq/NAT) (IP_NF_IPTABLES) [M/n/?] ?

IP 表是一个可扩展的包识别框架。

IPv6: Netfilter Configuration (EXPERIMENTAL)

DECnet: Netfilter Configuration

Bridge: Netfilter Configuration

DCCP Configuration (EXPERIMENTAL)

IPv6 网络过滤器配置

DECnet 网络过滤器配置

桥网络过滤器配置

DCCP 配置

SCTP Configuration (EXPERIMENTAL)

SCTP 配置

TIPC Configuration (EXPERIMENTAL)

TIPC 配置

QoS and/or fair queueing

QoS 和 fair 序列

Network testing

网络测试

Bluetooth subsystem support

蓝牙子系统支持

(18) 驱动程序选项。

因为驱动程序的选项层次很多,而且对嵌入式系统来说也比较重要,所以单独放到下一节进行介绍。

(19) File systems(文件系统)。

本菜单用于选择 Linux 内核支持的文件系统的类型。在选择时需要了解各种文件系统的特性,根据自己的需要作出决定。

Second extended fs support (EXT2_FS) [Y/n/m/?]

Ext2 格式的文件系统,它是 Linux 默认的标准文件系统。

Ext3 journalling file system support (EXT3_FS) [M/n/y/?]

Ext3 是日志型文件系统,安全可靠,适用于多任务系统。

Reiserfs support (REISERFS_FS) [M/n/y/?]

Reiserfs 性能几乎全面超越 EXT2,小文件(小于 4KB)性能非常突出。创建和删除文件速度最快。

JFS filesystem support (JFS_FS) [M/n/y/?]

IBM 的 JFS 文件系统。

XFS filesystem support (XFS_FS) [M/n/y/?]

XFS 文件系统产生的碎片少,多线程并发读写时效果好,大文件(>64KB)时性能尤佳。由于 XFS 在内存中缓存尽可能多的数据,且仅当内存不足时才会将数据刷到磁盘,所以应当仅在确保电力供应不会中断的情况下才使用 XFS。

OCFS2 file system support (EXPERIMENTAL) (OCFS2_FS) [M/n/y/?]

Minix fs support (MINIX_FS) [M/n/y/?]

ROM file system support (ROMFS_FS) [M/n/y/?]

这是一个很小的应用目标为 RAM Disk 的只读文件系统,但它也可用于其他只读存储介质。

Quota support (QUOTA) [Y/n/?]

磁盘配额支持,限制某个用户或者某组用户的磁盘占用空间,Ext2/Ext3/Reiserfs 都支持它。

Kernel automounter support (AUTOFS_FS) [M/n/y/?]

内核自动加载远程文件系统,不选,需要时可选下面这项。

Kernel automounter version 4 support (also supports v3) [M/n/y/?]

新的(v4)内核自动加载远程文件系统的支持,也支持 v3。

Filesystem in Userspace support (FUSE_FS) [M/n/y/?]

本选项允许在用户空间实现一个文件系统,如果打算开发一个自己的文件系统,或者使用一个基于用户空间的文件系统,可选择此项。

下面的各项是一些子菜单,分别用于设定一种特定设备或类型的文件系统,因为内容太多,仅给出其功能说明。

CD-ROM/DVD Filesystems. 设置 CD-ROM/DVD 的文件系统。

DOS/FAT/NT Filesystems. 支持 DOS/Windows 的文件系统。

Pseudo filesystems. 伪文件系统,如 proc、虚拟内存文件系统等的设置。

Miscellaneous filesystems. 杂项文件系统。虽称为“杂项”,但对嵌入式系统来说,却常常是“主项”,因为嵌入式系统中常用的 JFF2 和 cramfs 等文件系统都在该菜单下。

Network File Systems. 网络文件系统,包括 NFS、SMB 和 NCP 等网络文件系统的选择和设置。

Partition Types. 高级磁盘分区类型,不确定可以全不选。

Native Language Support. 本地语言支持。为了减少内存占用,只选择必要的选项,且选择 M 编译成模块形式。

(20) Kernel hacking(内核调试)。

本菜单用于设置内核所支持的调试选项。

Show timing information on printk (PRINTK_TIME) [N/y/?]

在 printk 的输出中包含时间信息,可以用来分析内核启动过程各步骤所用时间。

Magic SysRq key (MAGIC_SYSRQ) [Y/n/?]

选中该选项将使内核即使处理调试状态时,也能接受一些预定义的功能键,完成一些特定的内核功能,如将缓冲区中的数据 flush 到外存。如果读者还不了解这些功能键的定义,那就不要选择该选项,以免产生不可预测的后果。

Kernel debugging (DEBUG_KERNEL) [Y/n/?]

在调试驱动程序或内核时,可选择该项,以便于错误的定位。

(21) Page alloc debug is incompatible with Software Suspend on i386。

在 i386 上,页分配调试和软件挂起是矛盾的。

(22) Security options(安全选项)。

本菜单用于设置跟系统安全相关的选项。

Enable access key retention support (KEYS) [Y/n/?]

在内核中保留 authentication token 和 access key。

Enable different security models (SECURITY) [Y/n/?]

允许内核选择不同的安全模型,如果未选中则内核将使用默认的安全模型。

Socket and Networking Security Hooks (SECURITY_NETWORK) [Y/n/?]

允许安全模型通过 Security Hook 对 Socket 与 Networking 进行访问控制。

XFRM (IPSec) Networking Security Hooks (SECURITY_NETWORK_XFRM) [N/y/?]

启用 XFRM 安全 Hook。

Default Linux Capabilities (SECURITY_CAPABILITIES) [Y/n/m/?]

启用与默认的 Linux 的兼容性。

Root Plug Support (SECURITY_ROOTPLUG) [N/m/y/?]

一个简单的 Linux 安全模块,在特定的 USB 设备不存在时它简单地禁止一切 egid == 0 的进程运行。

NSA SELinux Support (SECURITY_SELINUX) [Y/n/?]

美国国家安全局 (National Security Agency, NSA) 开发的 SELinux (安全增强 Linux),需要时可选择。选择后还需要进行下面的启动、调试选项。

(23) Cryptographic options(加密选项)。

本菜单用于配置应用程序开发所能使用的加解密操作的 API。另外,还可能对各种加密算法的选项进行设置。

Cryptographic API 下面列出内核提供的加密 API 支持。这些加密算法被广泛地应用于驱动程序通信协议等机制中。子选项可以全不选,内核中若有其他部分依赖它,会自动选上。

(24) Hardware crypto devices。

本菜单包含对硬件 crypto 设备的支持。

(25) Library routines(库子程序)。

本菜单用于选择内核需要的压缩和校验算法的实现函数。只有那些不包含在内核源码中的第三方内核模块才需要在这里设置。对内核本身来说,可以全不选。内核中若有依赖它的地方,会自动选上。

- CRC-CCITT functions。传送 8 位字符,欧洲标准。
- CRC16 functions。传送 8 位字符,美国标准。
- CRC32 functions。用于点对点的同步数据传输中,传输网络数据包所必需的。
- CRC32c(Castagnoli, et al)Cyclic Redundancy-Check。用于点对点的同步数据传输中,如 SCSI 设备。

五、Linux 内核的设备驱动程序配置选项

Linux 的设备驱动程序绝大多数都放置在选项配置界面的 Device Drivers 菜单下。在对各种按设备类型归类后,又分别设置相应的子菜单。下面对各子菜单的内容进行简要的介绍。

(1) Generic Driver Options。

该菜单对应 drivers/base 目录中的 Kconfig 配置文件,用于确定各种固件驱动程序,以及驱动程序的调试信息是否编入内核映像。

(2) Connector-unified userspace <—> kernelspace linker。

该菜单对应 drivers/connector 目录中的 Kconfig 配置文件,用于选择用户空间和内核空间连接器,工作在 netlink socket 协议的顶层。通常不选。

(3) Memory Technology Devices (MTD)。

该菜单对应 drivers/mtd 目录中的 Kconfig 配置文件,适用于特殊的存储技术装置,如常用于数码相机或嵌入式系统的闪存卡等。

(4) Parallel port support。

该菜单对应 drivers/parport 目录中的 Kconfig 配置文件,用于并口设备驱动程序的支持(如传统的打印机接口)。

(5) Plug and Play support。

该菜单对应 drivers/pnp 目录中的 Kconfig 配置文件,用于即插即用支持。这里的选项与 PCI 设备无关,主要包含计算机外围设备的热插拔功能。

(6) Block devices。

该菜单对应 drivers/block 目录中的 Kconfig 配置文件,用于块设备的驱动程序的设计。包括软驱、RAMDisk 等。

(7) ATA/ATAPI/MFM/RLL support。

该菜单用于配置对 ATA/ATAPI/MFM/RLL 的支持。

(8) SCSI device support.

该菜单对应 drivers/scsi 目录中的 Kconfig 配置文件,用于配置各种 SCSI 设备的驱动。

(9) Old CD-ROM drivers.

Support non-SCSI/IDE/ATAPI CDRom drives (CD_NO_IDESCSI) [N/y/?]

(10) Multi-device support(RAID and LVM)。

该菜单对应 drivers/md 目录中的 Kconfig 配置文件,用于配置通过一个逻辑设备对多个物理设备的支持(RAID 和 LVM)。RAID 和 LVM 的功能是使多个物理设备组建成一个单独的逻辑磁盘(设备)。

(11) Fusion MPT device support.

该菜单对应 drivers/message/fusion/目录中的 Kconfig 配置文件,用于配置对 Fusion MPT 设备(SCSI 适配器)的支持。

(12) IEEE 1394(FireWire) support.

该菜单对应 drivers/ieee1394 /目录中的 Kconfig 配置文件,用于配置对 IEEE 1394 接口的支持。

(13) I2O device support.

该菜单对应 drivers/ message/i2o/目录中的 Kconfig 配置文件,用于配置对 I2O(智能 I/O)设备的支持。I2O 设备使用专门的 I/O 处理器负责中断处理、缓冲存取和数据传输等繁琐任务以减少 CPU 占用。

(14) Network device support.

该菜单对应 drivers/net/目录中的 Kconfig 配置文件,用于配置对网络适配器的设置,包括物理层(如以太网接口)参数和链路层协议(如 PPP)等。

(15) ARCnet devices.

该菜单用于配置对 ARCnet 设备的支持。

(16) PHY device support.

该菜单用于配置对 PHY 设备的支持。

(17) Ethernet(10or100Mbit)。

该菜单用于配置对 10 或 100Mb/s 的 Ethernet 的支持。

(18) Ethernet(1000Mbit)。

该菜单用于配置对 1000Mb/s 的 Ethernet 的支持。

(19) Ethernet(10000Mbit)。

该菜单用于配置对 10Gb/s 的 Ethernet 的支持。

(20) Token Ring devices.

该菜单用于配置对令牌网设备的支持。

(21) Wireless LAN(non-hamradio)。

该菜单用于配置对无线 LAN 的支持。

(22) PCMCIA network device support。

该菜单用于配置对 PCMCIA 网络设备的支持。

(23) Wan interfaces。

该菜单用于配置对 Wan 的支持。

(24) ATM drivers。

该菜单用于配置对 ATM 设备的支持。

(25) ISDN subsystem。

该菜单对应 drivers/isdn/目录中的 Kconfig 配置文件,用于配置对 ISDN(Integrated Service Digital Network,综合业务数字网)的驱动。

(26) Telephony Support。

该菜单用于配置对 linux 通过电话卡获得 IP 这种应用的支持。

(27) Input device support。

该菜单对应 drivers/input/目录中的 Kconfig 配置文件,用于配置对各种输入设置的驱动。包括鼠标、键盘、游戏杆和触摸屏等及其相关的硬件接口。

(28) Hardware I/O ports。

该菜单配置对硬件 I/O 端口的支持。

(29) Character devices。

该菜单对应 drivers/char/目录中的 Kconfig 配置文件,用于配置对各种字符设备的驱动。包括串口、伪终端、并口打印机、PCMCIA 接口的字符设备和看门狗等。

(30) Serial drivers。

该菜单用于配置对 8250/16550 和兼容系列的支持。

(31) Non-8250 serial port support。

该菜单用于配置对非 8250 系列端口的支持。

(32) IPMI。

该菜单用于配置对 IPMI 的支持。

(33) Watchdog Cards。

该菜单配置对看门狗设备的支持。

(34) Ftape, the floppy tape device driver。

该菜单用于配置对软盘设备的支持。

(35) PCMCIA character devices。

该菜单用于配置对 PCMCIA 特性设备的支持。

(36) TPM devices。

该菜单用于配置对 TPM 设备的支持。

(37) I2C support。

该菜单对应 drivers/i2c/ 目录中的 Kconfig 配置文件,用于配置对 I2C 设备的接口、算法及硬件参数的设置。I2C 是 Philips 公司极力推动的,微控制应用中使用的低速串行总线协议,可用于监控电压、风扇转速和温度等。SMBus(系统管理总线)是 I2C 的子集。

(38) SPI support。

该菜单对应 drivers/spi/ 目录中的 Kconfig 配置文件,用于配置对 SPI(串行外围接口)的驱动。SPI 常用于微控制器与外围设备(传感器、eeprom、flash、编码器和模数转换器等)之间的通信。例如,MMC 和 SD 卡就通常需要使用 SPI。

(39) Dallas's 1-wire bus。

该菜单对应 drivers/w1/ 目录中的 Kconfig 配置文件,用于配置对一线总线的驱动。

(40) Hardware Monitoring support。

该菜单对应 drivers/hwmon/ 目录中的 Kconfig 配置文件,用于配置对系统硬件的监视程序。目前的主板大多都有一个监控硬件健康的设备,用于监视温度、电压和风扇转速等,可按照自己主板实际使用的芯片选择相应的子项。该功能需要 I2C 的支持。

(41) Misc devices。

该菜单对应 drivers/misc/ 目录中的 Kconfig 配置文件,用于配置对各种杂项设备的驱动配置,包括各类 ATA/ATAP 接口设备驱动,通常包括 IDE 硬盘和 ATAPI 光驱。

(42) Multimedia devices。

该菜单对应 drivers/media/ 目录中的 Kconfig 配置文件,用于配置对多媒体设备的驱动,包括视频捕获卡、音频接收卡和数字视频广播设备(DVB 卡或机顶盒)等。

(43) Video Capture Adapters。

该菜单用于配置对视频截图适配器的支持。

(44) Encoders and Decoders。

该菜单用于配置对编码器和解码器设备的支持。

(45) V4L USB devices。

该菜单用于配置对 V4L USB 设备的支持。

(46) Radio Adapters。

该菜单用于支持对无线适配器的支持。

(47) Digital Video Broadcasting Devices。

该菜单用于配置对数字视频播放设备的支持。

(48) Graphics support。

该菜单对应 drivers/video/ 目录中的 Kconfig 配置文件,用于配置对图形设备/显卡

支持,包括 Framebuffer 的驱动程序。

(49) Console display driver support.

该菜单用于配置对控制台显示器的支持。

(50) Logo configuration.

该菜单用于配置对 logo 的支持。

(51) Backlight & LCD device support.

该菜单用于配置对 Backlight 和 LCD 的支持。

(52) Sound.

该菜单对应 sound/ 目录中的 Kconfig 配置文件,用于配置对声卡的驱动。包括各种音频处理芯片。

(53) Advanced Linux Sound Architecture.

该菜单用于配置对先进的 linux 声音体系的支持。

(54) Generic devices.

该菜单用于配置对一般设备的支持。

(55) Open Sound System.

该菜单用于配置对公放声音系统的支持。

(56) USB support.

该菜单对应 drivers/usb/ 目录中的 Kconfig 配置文件,用于配置对 USB Host 和 Device 的驱动。

(57) MMC/SD Card support.

该菜单对应 drivers/mmc/ 目录中的 Kconfig 配置文件,用于配置 MMC/SD 卡的驱动。

(58) LED devices.

该菜单用于配置对 LED 设备的支持。

(59) EDAC-error detection and reporting(RAS)(EXPERIMENTAL).

该菜单用于配置对 EDAC 的支持。

(60) Real Time Clock.

该菜单用于配置对实时时钟的支持。

(61) DMA Engine support.

该菜单用于配置对 DMA 引擎的支持。

六、Linux 2.6 内核编译步骤

(1) 下载源码并解压。

从 <ftp://ftp.kernel.org/pub/linux/kernel/v2.6> 中下载 linux-2.6.x.tar.gz 到 /usr/

src 目录。用 su 命令切换为 root 用户,键入下面命令:

```
#cd/usr/src  
#tar xzvf linux-2.6x.tar.gz
```

(2) 进入源代码目录。

```
#cd linux
```

(3) 可以选择下面的一个命令来配置内核。

```
#make menuconfig  
#make xconfig  
#make oldconfig
```

对内核进行配置。配置完毕后,保存退出。

(4) 编译内核。

```
#make
```

(5) 编译模块。

```
#make modules
```

(6) 安装模块。

```
#make modules_install
```

(7) 安装内核。

```
#make install
```

到此内核编译完成,重新启动计算机,启动新内核。如果启动项中没有新内核的选项,可以通过修改引导程序的配置文件进行解决。

实验3 内核同步机制

一、实验目的

通过改造 pipe 系统调用理解内核同步机制。

二、实验内容

1. 改造 pipe 系统调用,使 pipe 获得的文件句柄可以用来进行简单同步操作。
2. 增加三个针对 pipe 的 ioctl 命令。
3. SET_EVENT_COUNT,设置 pipe 上要等待的 event count。
4. NOTIFY_EVENT,通知 event,count 减 1,必要时唤醒在 pipe event 上等待的线程。

5. WAIT_EVENT,阻塞等待 event,count 减 1,直到 count 变为 0 结束阻塞。

类似以下程序:

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

int main()
{
    int pipe_fd[2], ret;
    int i;
    pipe(pipe_fd);
    ret=ioctl(pipe_fd[0], SET_EVENT_COUNT, 4);
    for(i=0; i<4; i++)
        if(fork()==0) {
            sleep(10);
            fprintf(stderr, "child %d\n", i);
            ret=ioctl(pipe_fd[0], NOTIFY_EVENT);
            exit(0);
        }

    ret=ioctl(pipe_fd[0], WAIT_EVENT);
    fprintf(stderr, "parent %d\n", ret);
    return 0;
}
```

父进程将阻塞等待子进程发送 4 次 NOTIFY_EVENT 后被唤醒。

三、准备知识

管道系统调用是标准的 UNIX 系统调用,几乎被包含在了所有的 UNIX 标准中。

父进程通过 pipe 调用创建一个管道,这个管道由两个文件描述符表示。fd[0]被用来读,fd[1]被用来写。

ioctl 是一个系统调用,可以为设备驱动或者特殊的文件系统提供额外的命令和参数。

```
int ioctl(int d, int request, ...)
```

其中 d 是要操作的文件句柄,request 是要求的操作,后面的参数是变长参数。

通过支持 ioctl,设备驱动或者特殊的文件系统,可以提供标准系统调用语义以外对文件系统额外语义的操作。

四、参考代码

1. 改造 pipe_inode_info 数据结构

```
struct pipe_inode_info {  
    wait_queue_head_t wait;  
    wait_queue_head_t event_wait;  
    atomic_t count;  
    unsigned int nrbufs, curbuf;  
    struct page * tmp_page;  
    unsigned int readers;
```

2. 修改 pipe.c,增加 pipe_wait_event 函数来等待时间

```
static int pipe_wait_event(struct pipe_inode_info * pipe)  
{  
    int ret=0;  
    //定义等待队列  
    DEFINE_WAIT(wait);  
    while(atomic_read(&pipe->count)>0) {  
        prepare_to_wait(&pipe->event_wait, &wait,  
                        TASK_INTERRUPTIBLE);  
        //检查是否有信号要处理  
        if(signal_pending(current)) {
```

```

ret=-EINTR;
break;
}
schedule();
}
finish_wait(&pipe->event_wait, &wait);
return ret;
}

```

3. 增加 pipe_wakeup_event, 用来唤醒等待在某一个事件上的进程

```

static void pipe_wakeup_event(struct pipe_inode_info * pipe)
{

```

```

    atomic_dec(&pipe->count);
    wake_up_interruptible(&pipe->event_wait);
}

```

4. 改造 pipe_ioctl, 增加相应的 ioctl 参数

```

#define IOCTL_PIPE_EVENT_WAIT          0xFFFF
#define IOCTL_PIPE_EVENT_WAKEUP        0xFFFE
#define IOCTL_PIPE_EVENT_SET           0xFFFD

static int
pipe_ioctl(struct inode * pino, struct file * filp,
            unsigned int cmd, unsigned long arg)
{
    mutex_unlock(&inode->i_mutex);

    return put_user(count, (int __user *) arg);
case IOCTL_PIPE_EVENT_WAIT:
    pipe=inode->i_pipe;
    return pipe_wait_event(pipe);
case IOCTL_PIPE_EVENT_WAKEUP:
    pipe=inode->i_pipe;
    pipe_wakeup_event(pipe);
    return 0;
case IOCTL_PIPE_EVENT_SET:
    pipe=inode->i_pipe;
    atomic_set(&pipe->count, arg);
    return 0;
default:

```

```
return-EINVAL;
```

```
}
```

5. 修改 pipe 结构, 分配函数 alloc_pipe_info

```
if(pipe) {  
    init_waitqueue_head(&pipe->wait);  
    init_waitqueue_head(&pipe->event_wait);  
    atomic_set(&pipe->count, 0);  
    pipe->r_counter=pipe->w_counter=1;  
    pipe->inode=inode;  
}
```

五、实验思考题

在 i386 Linux kernel 中, 切换进程时浮点寄存器(Float Point Unit, FPU)的保存和恢复不是完全在进程切换时进行(Lazy FPU)的。尝试把保存和恢复放在进程切换时进行, 比较这样做的性能变化。



实验 4 进程间通信（内存映射、共享内存、信号量、管道和信号等）

实验 4.1 内核进程间通信的实现

一、实验目的

1. 掌握 Linux 内核进程间通信的设计。
2. 了解向内核中添加新的系统调用的方法。

二、实验内容

流管道(stream pipe)是一种全双工的管道。单个流管道就能向父、子进程提供双向的数据。流管道最早出现在 4.2BSD 和 SVR3.2 系统中,但至今在 Linux 内核中还没有实现。

本实验要求读者在 Linux 2.6 内核中基于 pipefs 实现流管道。

实验要求如下。

- (1) 增加系统调用 `int s_pipe(int filedes[2])`,并实现 `do_s_pipe()`。
- (2) 返回的 `filedes[0]`和 `filedes[1]`既可用于读操作也可用于写操作。
- (3) 父进程调用 `s_pipe` 系统,调用后创建子进程。父进程和子进程要求能够使用 `s_pipe` 返回的 `fd[]`进行双向的数据通信。

三、准备知识

Linux 内核中所有的系统调用都维护在一张名为 `sys_call_table` 的表中,每一个系统调用函数在表中的顺序即为这个系统调用号。这个号码在用户进程提出系统调用请求时被传递到内核,用来唯一地标识一个系统调用。所以,向内核中添加一个新的系统调用首先需要分配一个新的系统调用号,并将其发布(通常使用 `include/asm/unistd.h` 文件)。然后修改系统调用表,把新的系统调用函数添加到系统调用号所对应的位置。这样,用户就可以根据这个号码来使用相应的系统调用。例如:

```
#define __NR_s_pipe 324
#define SYS_s_pipe __NR_s_pipe
```



```

int main()
{
    ...
    if((ret=syscall(SYS_s_pipe, fd))!=-1) {
        printf("s_pipe error: %s\n", strerror(errno));
        return ret;
    }
    ...
}

```

四、参考代码

(代码针对 Linux 2.6.18 内核版本)

增加新的系统调用函数 sys_s_pipe()。

```

asmlinkage int sys_s_pipe(unsigned long __user * fildes)
{
    int fd[2];
    int error;

    error=do_s_pipe(fd);
    if(!error) {
        if(copy_to_user(fildes, fd, 2*sizeof(int)))
            error=-EFAULT;
    }
    return error;
}

```

在文件 arch/i386/kernel/syscall_table.S 的 sys_call_table 中添加 sys_s_pipe 函数名。

```
.long sys_s_pipe
```

流管道功能实现。

```

struct file* create_s_pipe(void)
{
    int err;
    struct inode* inode;
    struct file* f;
    struct dentry* dentry;
    struct qstr name= { .name="" };

```

```

f=get_empty_filp();
if(!f)
    return ERR_PTR(-ENFILE);
err=-ENFILE;
inode=get_pipe_inode();
if(!inode)
    goto err_file;

err=-ENOMEM;
dentry=d_alloc(pipe_mnt->mnt_sb->s_root, &name);
if(!dentry)
    goto err_inode;

dentry->d_op=&pipefs_dentry_operations;
dentry->d_flags &= ~DCACHE_UNHASHED;
d_instantiate(dentry, inode);
f->f_path.mnt=mntget(pipe_mnt);
f->f_path.dentry=dentry;
f->f_mapping=inode->i_mapping;

f->f_flags=O_RDWR;
f->f_op=&rdwr_pipe_fops;
f->f_mode=FMODE_READ | FMODE_WRITE;
f->f_version=0;

return f;

err_inode:
    free_pipe_info(inode);
    iput(inode);
err_file:
    put_filp(f);
    return ERR_PTR(err);
}

void free_s_pipe(struct file * f)
{
    free_pipe_info(f->f_dentry->d_inode);

```

```

        dput(f->f_path.dentry);
        mntput(f->f_path.mnt);
        put_filp(f);
    }

int do_s_pipe(int * fd)
{
    struct file * f1, * f2;
    int error;
    int fd1, fd2;

    f1=create_s_pipe();
    if(IS_ERR(f1))
        return PTR_ERR(f1);
    f2=create_s_pipe();
    error=PTR_ERR(f2);
    if(IS_ERR(f2))
        goto err_s_pipe1;

    error=get_unused_fd();
    if(error<0)
        goto err_s_pipe2;
    fd1=error;

    error=get_unused_fd();
    if(error<0)
        goto err_fd;
    fd2=error;

    fd_install(fd1, f1);
    fd_install(fd2, f2);
    fd[0]=fd1;
    fd[1]=fd2;

    return 0;

err_fd:
    put_unused_fd(fd1);
err_s_pipe2:

```

```

        free_s_pipe(f2);
err_s_pipe1:
        free_s_pipe(f1);
        return error;
}

```

在文件 include/asm-i386/unistd.h 中发布 s_pipe 的系统调用号码,并修改系统调用函数总数

```

#define __NR_s_pipe 324
#define NR_syscalls 325

```

实验 4.2 进程间通信的应用

一、实验目的

1. 了解 Linux 操作系统用户进程的编程。
2. 了解 Linux 操作系统中进程间通信系统调用的使用方法。

二、实验内容

编写一个生产者—消费者程序组,要求同时支持多个生产者和消费者,生产者与消费者之间使用共享内存进行数据传递,并使用信号量对数据的内容进程保护。

三、准备知识

系统 V 的 IPC 有各自的命名空间,并且都使用一个正整数来标识。在使用进程间通信的系统调用时,首先必须创建或获得一个 IPC 标识,然后才可以对这个标识所标志的 IPC 进行操作。例如以信号量为例,首先需要设定一个互相通信的进程所公共使用的 key,然后利用 semget() 函数由指定的 key 得到信号量的标识;最后就可以使用 semop() 函数对信号量进行操作。这里 key 的选取可以有很多种方法,例如可以直接指定一个数字,但这样做有可能会产生 key 的冲突。一个更好的方法是使用类似 ftok() 的函数来将一个目录名转化为一个 key。由于不相关的进程通常不会选用相同的目录名,这样的做法有效地减少了 key 的冲突。

四、参考代码

```

#include<stdio.h>
#include<sys/types.h>

```

```

#include<linux/ipc.h>
#include<linux/sem.h>
#include<linux/shm.h>
#include<errno.h>
#include<time.h>
#include "ipc.h"

Int semid;
Struct sembuf sops;
int shmid;
void* shmaddr;

static inline key_t ipc_get_key(void)
{
    key_t key;

    key=ftok(".", 1234);
    if(key==--1)
        return 1234;
    return key;
}

P(short flags)
{
    sops.sem_num=0;
    sops.sem_op=-1;
    sops.sem_flg=flags;
    if(semop(semid, &sops, 1)==--1)
        fprintf(stderr, "%s: semop error: %s\n", __FUNCTION__,
                strerror(errno));
}

V(short flags)
{
    sops.sem_num=0;
    sops.sem_op=1;
    sops.sem_flg=flags;
    if(semop(semid, &sops, 1)==--1)
        fprintf(stderr, "%s: semop error: %s\n", __FUNCTION__,

```

```

        strerror(errno));
    }

void writer(void)
{
    P(IPC_NOWAIT | SEM_UNDO);
    sprintf(shmaddr, "This is for shared memory(%ld).", time(NULL));
    V();
}

void reader(void)
{
    P(IPC_NOWAIT | SEM_UNDO);
    printf("reader: [%s]\n", shmaddr);
    V();
}

int main(int argc, char * argv[])
{
    int opt=0;
    key_t key=ipc_get_key();

    if(argc>1 && argv[1])
        opt=1;

    if((semid=semget(key, 5, 0666 | IPC_CREAT))== -1) {
        fprintf(stderr, "semget error: %s\n", strerror(errno));
        return -1;
    }

    if((shmid=shmget(key, 64, 0666 | IPC_CREAT))== -1) {
        fprintf(stderr, "shmget error: %s\n", strerror(errno));
        return -1;
    }

    if((shmaddr=(void *)shmat(shmid, shmaddr, 0))== (void *)-1) {
        fprintf(stderr, "shmat error: %s\n", strerror(errno));
        return -1;
    }
}

```



```

        if(opt)
            writer();
        else
            reader();

        if(shmdt(shmaddr)---1) {
            fprintf(stderr, "shmdt error: %s\n", strerror(errno));
            return -1;
        }

        return 0;
    }
}

```

五、实验思考题

在用户空间实现 atomic 和 spinlock, 并编写应用程序测试比较原子操作 atomic 和 spinlock 的性能差异。

实验5 proc 文件系统

一、实验目的

1. 了解 proc 文件系统编程。
2. 了解内核模块编程。

二、实验内容

编写一个内核模块,模块载入时会向 proc 文件系统中添加一个名为 hello 的文件。用户可以往该文件中写入一个字符串。当用户读取该文件时,文件将返回用户前一次输入的字符串。当模块被卸载时,该文件也从 proc 文件系统中删除。

当插入模块时,可以用 cat 命令向新创建的文件中写入字符串,按 Ctrl+D 组合键可以结束输入。

```
# cat> /proc/hello
Hello
^D
```

这时显示 /proc/hello 文件的内容时,内核会将用户上次输入的字符串返回给用户。

```
# cat /proc/hello
last message: hello
```

三、准备知识

内核模块的相关知识已经在第 7 章中进行了阐述,这里将对 proc 文件系统进行介绍。

1. proc 文件系统介绍

proc 文件系统是一个特殊的文件系统,内核通过 proc 文件系统将内核的一些信息输出给用户。/proc 目录下每个文件的读写都可以绑定到一个内核函数上,当文件被读写时,被绑定的相应内核函数就会被触发,由这些函数处理文件的输出与输入。用户通过 proc 文件系统可以了解进程的地址空间信息、系统的硬件信息、系统的中断信息和系统

的 I/O 信息等。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。但是也有一些 proc 文件被设置成可写,用户就可以通过写这些 proc 文件将一些参数传递给内核。对于 proc 文件系统的更详细介绍可参考本书的第 5 章。

2. 创建 proc 文件

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 文件系统编程中,最重要的数据结构是 `proc_dir_entry`,它的定义为:

```
struct proc_dir_entry {
    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,常用到的一些成员如下。

- name。文件名。
- mode。文件的类型和权限。
- nlink。读文件的链接数。
- read_proc。读操作函数。
- write_proc。写操作函数。

- owner。该文件的拥有者模块。

在 proc 文件系统中创建一个新文件时,需要将该结构初始化,再将该结构传递给内核。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 文件只有单个内存页面供用户数据传输。这样就把用户文件的大小限制在 4KB。start 参数在这里就是用来实现大数据文件的。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断创建更大的文件,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,指向的是用户空间的缓冲区。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 文件系统中了。加载的第一步是获得一个 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;mod 是文件的保护掩码(默认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,如果要创建 /proc/test 文件,可以使用:

```
create_proc_entry("test", 0, NULL);
```

当获得了 proc_dir_entry 后,第二步就是对这个数据结构设置 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 文件系统还提供了多个函数, 方便在 proc 文件系统中创建或者删除文件或目录, 主要的函数如下。

- 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。

这几个函数用法和 VFS 中的相应函数的用法类似, 不再详细解释, 读者可以查询相应的手册了解具体的用法。

四、参考代码

```
#include<linux/init.h>
#include<linux/module.h>
#include<linux/proc_fs.h>
#include<asm/uaccess.h>

MODULE_LICENSE("GPL");

#define STRINGLEN 1024
char global_buffer[STRINGLEN];
struct proc_dir_entry * hello_file;

//读取/proc/hello 时被调用的函数
int proc_read_hello( char * page, char ** start, off_t off,
int count, int * eof, void * data)
{
    int len;
    try_module_get(THIS_MODULE); //增加模块引用计数
    len=sprintf( page, "last message: %s\n", global_buffer);
    module_put(THIS_MODULE);
    return len;
}

//写/proc/hello 时被调用的函数
```

```
int proc_write_hello( struct file* file, const char* buffer,
                    unsigned long count,
                    void* data)
```

```
{
    int len, retval;
    try_module_get(THIS_MODULE);
    //防止用户输入的字符串过长
    if(count >= STRINGLEN)
        len = STRINGLEN - 1;
    else
        len = count;
    //将用户输入复制到内核空间
    retval = copy_from_user(global_buffer, buffer, len);
    global_buffer[len] = '\0';          //字符串结束符
    module_put(THIS_MODULE);
    return len;
}
```

//模块初始化函数

```
static __init int hello_init()
{
    //创建一个 proc_dir_entry
    hello_file = create_proc_entry("hello", 0644, NULL);
    strcpy(global_buffer, "hello");
    //设置文件的读写函数
    hello_file->read_proc = proc_read_hello;
    hello_file->write_proc = proc_write_hello;
    hello_file->owner = THIS_MODULE;
    return 0;
}
```

//模块清理函数

```
static __exit void hello_exit()
{
    remove_proc_entry("hello", NULL);
}
```

//注册模块初始化与注册函数


```
module_init(hello_init);  
module_exit(hello_exit);
```

五、实验思考题

本实验的参考代码假设了用户的输入不能超过 1024 字节。如果用户的输入超过 1024 字节,代码会将用户的输入截断。修改参考代码使得用户的输入可以不受此限制。

实验 6 虚拟内存的管理

一、实验目的

理解 Linux 虚拟内存管理的机制。

二、实验内容

本实验的内容包括以下两部分。

1. 模块

编写一个内核模块,分别实现如下几个函数。

- static void mtest_dump_vma_list(void): 依次列出当前进程各段的读、写或执行权限。
- static void mtest_find_vma(unsigned long addr): 找到虚地址 addr 所在的 vma, 通过 printk 打印该段的起始地址、终止地址和段标志等信息。
- static void mtest_find_page(unsigned long addr): 找到虚拟地址 addr, 显示出其对应的物理地址。

为了测试和使用以上函数,在内核模块的初始化函数中创建一个名为 mtest 的 /proc 文件。mtest 文件绑定的写函数 mtest_write 允许用户程序写入一串字符串。mtest_write 函数的定义为:

```
static ssize_t mtest_write(struct file * file, const char __user * buffer, size_t
count, loff_t * data)
```

2. 测试程序

为了测试以上模块的正确性,要求编写一个称为 test 的应用程序,它通过打开 /proc 文件 mtest,调用模块内的相关函数并给出相应的测试信息。

三、准备知识

进程的虚拟存储空间管理的相关数据结构 vm_area_struct 和 mm_struct,请参照第 4 章中的相关介绍。关于 /proc 文件的相关知识请参阅第 5 章文件系统的相关介绍。内核模块的相关知识参阅第 7 章的相关介绍。

另外,读取当前进程的 mm 结构,需要首先使用 `down_read(&mm->mmap_sem)` 对其控制信号量进行 P 操作,在使用完成后,需要使用 `up_read(&mm->mmap_sem)` 对该信号量进行 V 操作。

四、参考代码

1. 模块的参考代码

```
#include<linux/kernel.h>
#include<linux/module.h>
#include<linux/init.h>
#include<linux/proc_fs.h>
#include<linux/mm.h>
#include<linux/highmem.h>
#include<asm/uaccess.h>

static void
mtest_dump_vma_list(void)
{
    struct mm_struct * mm=current->mm;
    struct vm_area_struct * vma;
    printk("mtest_dump_vma_list\n");
    down_read(&mm->mmap_sem);
    for (vma=mm->mmap; vma; vma=vma->vm_next) {
        printk("VMA 0x%lx- 0x%lx ",
            vma->vm_start, vma->vm_end);
        if (vma->vm_flags & VM_WRITE)
            printk("WRITE ");
        if (vma->vm_flags & VM_READ)
            printk("READ ");
        if (vma->vm_flags & VM_EXEC)
            printk("EXEC ");
        printk("\n");
    }
    up_read(&mm->mmap_sem);
}

static void
mtest_find_vma(unsigned long addr)
{

```

```

struct vm_area_struct * vma;
struct mm_struct * mm=current->mm;

printk("mtest_find_vma\n");

down_read(&mm->mmap_sem);
vma=find_vma(mm, addr);
if(vma && addr >= vma->vm_start) {
    printk("found vma 0x%lx-0x%lx flag %lx for addr 0x%lx\n",
           vma->vm_start, vma->vm_end, vma->vm_flags, addr);
} else {
    printk("no vma found for %lx\n", addr);
}
up_read(&mm->mmap_sem);
}

```

```

static void
mtest_find_page(unsigned long addr)
{
    struct vm_area_struct * vma;
    struct mm_struct * mm=current->mm;
    unsigned long kernel_addr;
    struct page * page;
    printk("mtest_find_page\n");
    down_read(&mm->mmap_sem);
    vma=find_vma(mm, addr);
    if(vma && addr >= vma->vm_start) {
        pgd_t * pgd;
        pud_t * pud;
        pmd_t * pmd;
        pte_t * pte;
        spinlock_t * ptl;
        pgd=pgd_offset(mm, addr);
        if(pgd_none(*pgd) || unlikely(pgd_bad(*pgd))) {
            printk("no pgd found for 0x%lx\n", addr);
            goto out;
        }
        pud=pud_offset(pgd, addr);
        if(pud_none(*pud) || unlikely(pud_bad(*pud))) {

```

```

        printk("no pud found for 0x%lx\n", addr);
        goto out;
    }
    pmd=pmd_offset(pud, addr);
    if(pmd_none(*pmd) || unlikely(pmd_bad(*pmd))) {
        printk("no pmd found for 0x%lx\n", addr);
        goto out;
    }
    pte=pte_offset_map_lock(mm, pmd, addr, &ptl);
    if(!pte)
        goto out;
    if(!pte_present(*pte)) {
        printk("pte not present for 0x%lx\n", addr);
        pte_unmap_unlock(pte, ptl);
        goto out;
    }
    page=pfn_to_page(pte_pfn(*pte));
    get_page(page);
    pte_unmap_unlock(pte, ptl);
    kernel_addr= (unsigned long)page_address(page);
    printk("va 0x%lx ->pa 0x%lx \n",
        addr,
        kernel_addr+ (addr&~PAGE_MASK));
    put_page(page);
} else {
    printk("no vma found for %lx\n", addr);
    goto out;
}
out:
    up_read(&mm->mmap_sem);
}

//proc fs stuff
static ssize_t
mtest_write(struct file* file, const char __user* buffer, size_t count, loff_t* data)
{
    char buf[128];
    unsigned long val, val2;
    if(count>sizeof(buf))

```

```

        return-EINVAL;
    if(copy_from_user(buf, buffer, count))
        return -EINVAL;
    if(memcmp(buf, "listvma", 7)==0)
        mtest_dump_vma_list();
    else if(memcmp(buf, "findvma", 7)==0) {
        if(sscanf(buf+7, "%lx", &val)==1) {
            mtest_find_vma(val);
        }
    }
    else if(memcmp(buf, "findpage", 8)==0) {
        if(sscanf(buf+8, "%lx", &val)==1) {
            mtest_find_page(val);
        }
    }
    else if(memcmp(buf, "writeval", 8)==0) {
        if(sscanf(buf+8, "%lx %lx", &val, &val2)==2) {
            mtest_write_val(val, val2);
        }
    }
    return count;
}

static struct
file_operations proc_mtest_operations={ .write=mtest_write };

static struct proc_dir_entry*mtest_proc_entry;

static int __init
mtest_init(void)
{
    mtest_proc_entry=create_proc_entry("mtest", 0777, NULL);
    if(mtest_proc_entry==NULL) {
        printk("Error creating proc entry\n");
        return-1;
    }
    mtest_proc_entry->proc_fops=&proc_mtest_operations;
    return 0;
}

```



```
static void
_exit mtest_exit(void)
{
    remove_proc_entry("mtest", NULL);
}
```

```
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("mtest");
```

```
module_init(mtest_init);
module_exit(mtest_exit);
```

2. 应用程序参考代码

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

```
int main()
{
    void volatile * p1, * p2;
    FILE * file;
    file=fopen("/proc/mtest", "w");
    if(file==NULL) {
        fprintf(stderr, "can not open /proc/mtest\n");
        return 1;
    }
    p1=mmap(0, 10 * 4096, PROT_READ, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
    if(p1==MAP_FAILED)
        perror("mmap");

    printf("mmap readonly area %p, length %lx\n", p1, 10 * 4096);
    fprintf(file, "listvma");
    fflush(file);

    p2=mmap(0, 10 * 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
    if(p2==MAP_FAILED)
        perror("mmap");
    printf("mmap read/write%p, length %lx\n", p2, 10 * 4096);

    fprintf(file, "findvma%p", p2);
```

```

fflush(file);

fprintf(file, "findpage%p", p2);
fflush(file);

* (long *)p2=0x100;

fprintf(file, "findpage%p", p2);

fflush(file);

fprintf(file, "findpage%p", p2);
fflush(file);

fprintf(file, "writeval%p%lx", p2, 0x1234);
fflush(file);

fprintf(file, "writeval%p%lx", p2, 0x4321);
fflush(file);
system("dmesg-c");

printf("value of address%p changed to%lx\n", p2, * (unsigned long *)p2);
munmap((void *)p1, 10 * 4096);
munmap((void *)p2, 10 * 4096);
fclose(file);
return 0;
}

```

五、实验思考题

采用与以上实验类似的方法,可进一步通过查看进程的页表,并找到该进程的内存中相关页的 page 结构体,显示出这此物理页的一些管理信息。

实验7 字符、块设备和网卡驱动

实验7.1 字符设备驱动

一、实验目的

通过本实验的学习,了解 Linux 操作系统中的字符设备驱动程序结构,并能编写简单字符设备的驱动程序,以及对所编写的设备驱动程序进行测试,最终了解 Linux 操作系统如何管理字符设备。

二、准备知识

字符设备驱动程序主要包括初始化字符设备、字符设备的 I/O 调用和中断服务程序。在字符设备驱动程序的 `file_operations` 结构中,需要定义字符设备的基本入口点。

- `open()` 函数
- `release()` 函数
- `read()` 函数
- `write()` 函数
- `ioctl()` 函数
- `select()` 函数

另外,注册字符设备驱动程序的函数为 `register_chrdev()`。`register_chrdev()` 函数的原型如下:

```
int register_chrdev(unsigned int major,           //主设备号
                    const char * name,           //设备名称
                    struct file_operations * ops); //指向设备操作函数指针
```

其中 `major` 是设备驱动程序向系统申请的主设备号。如果 `major` 为 0,则系统为该驱动程序动态分配一个空闲的主设备号。`name` 是设备名称,`ops` 是指向设备操作函数的指针。

注销字符设备驱动程序的函数是 `unregister_chrdev()`,原型如下:

```
int unregister_chrdev(unsigned int major, const char * name);
```

字符设备注册后,必须在文件系统中为其创建一个设备文件。该设备文件可以在

/dev 目录中创建,每个设备文件代表一个具体的设备。

使用 mknod 命令来创建设备文件。创建设备文件时需要使用设备的主设备号和从设备号作为参数。

阅读教材相关章节知识,了解字符设备的驱动程序结构。

三、实验内容

根据教材提供的实例,编写一个简单的字符设备驱动程序。要求该字符设备包括 open()、write()、read()、ioctl() 和 release() 这 5 个基本操作,并编写一个测试程序来测试所编写的字符设备驱动程序。

四、实验指导

(1) 驱动程序编写后,编写一个 Makefile 文件来编译此驱动程序。

Makefile 格式如下:

```
obj-m:=mydev.o
```

```
all:
```

```
make-C/lib/modules/$ (shell uname-r)/build M=$ (shell pwd) modules
```

```
clean:
```

```
make-C/lib/modules/$ (shell uname-r)/build M=$ (shell pwd) clean
```

(2) 使用 # insmod***.ko 来安装此驱动程序,并通过 # cat /proc/devices 命令查看新加载设备的设备号。

(3) 接下来需要创建设备文件,使用命令 # mknod /dev/***c 254 0。

(4) 最后运行测试文件来测试驱动程序。

实验 7.2 块设备驱动

一、实验目的

通过本实验的学习,了解 Linux 操作系统中的块设备驱动程序结构,并能编写简单的块设备的驱动程序,以及对所编写设备驱动程序进行测试,最终了解 Linux 操作系统如何管理块设备。

二、准备知识

块设备驱动程序主要包括初始化块设备、块设备的 I/O 调用和中断处理程序。块设备的 I/O 调用 ioctl() 函数、open() 函数、release() 函数与字符设备类似。关于块设备的

数据读取,需要使用 request 机制来实现。块设备驱动程序的入口点包括以下几个函数。

- request()函数(处理请求函数)
- ioctl 函数
- media_change 函数
- revalidate_disk 函数

块设备驱动程序的注册可以通过调用函数 register_blkdev()来完成。register_blkdev()函数的原型如下:

```
int register_blkdev(unsigned int major,          //主设备号
const char * name,                               //设备名称
```

注销块设备驱动程序的函数是 unregister_blkdev(),原型如下:

```
(int unregister_blkdev(unsigned int major,const char * name);
```

阅读教材相关章节知识,了解块设备的驱动程序结构。

三、实验内容

分析教材提供的实例,编写一个简单的 ramdisk 块设备驱动程序。要求该块设备包括 open()、request()、ioctl()和 release()等基本操作。并编写一个测试程序来测试所编写的设备驱动程序。

四、实验指导

将块设备驱动加载,并给这个块设备分区、创建文件系统,观察是否可用。

实验 7.3 网络设备驱动

一、实验目的

通过本实验的学习,了解 Linux 操作系统中的网络设备驱动程序的结构,并能编写简单的网络设备的驱动程序,以及对所编写网络设备驱动程序的测试,最终了解 Linux 操作系统是如何管理网络设备的。

二、准备知识

网络设备驱动程序的一些基本要求如下。

- (1) 数据包的发送和接收。
- (2) 中断处理。

(3) 提供时钟机制。

Linux 网络设备驱动程序由 4 部分组成：网络设备媒介层、网络设备驱动层、网络设备接口层及网络协议接口层。网络设备媒介层包括各种物理网络设备和传输媒介。对于网络设备接口层，Linux 系统用 `Net_device` 结构表示网络设备接口。`Net_device` 结构保存所有与硬件有关的接口信息，各协议软件主要通过 `Net_device` 结构来完成与硬件的交互。网络设备驱动层主要包括网络设备的初始化、数据包的发送和接收。网络协议接口层提供网络接口驱动程序的抽象接口。

网络驱动程序的基本方法如下。

(1) 初始化(initialize)。检测设备；配置和初始化硬件；初始化 `net_device` 结构；注册设备。

(2) 打开(open)。这个方法在网络设备被激活时被调用，进行资源的申请和硬件的激活等。`open` 方法的另一个作用是如果驱动程序作为一个模块被装入，则要防止模块卸载时设备处于打开状态。在 `open` 方法里要调用 `MOD_INC_USE_COUNT` 宏。

(3) 关闭(close)。释放某些系统资源。如果是作为模块装入的驱动程序，在 `close` 方法里应该调用 `MOD_DEC_USE_COUNT` 宏，减少设备被引用的次数，以使驱动程序可以被卸载。

(4) 发送(hard_start_xmit)。网络设备驱动程序发送数据时，系统调用 `dev_queue_xmit` 函数，发送的数据放在一个 `sk_buff` 结构中。一般的驱动程序将数据传输到硬件发出去，特殊的设备如 `loopback` 把数据组成一个接收数据再回送给系统，或如 `dummy` 设备直接丢弃数据。如果发送成功，则在 `hard_start_xmit` 方法里释放 `sk_buff`，返回 0，否则返回 1。

(5) 接收(reception)。驱动程序并不存在一个接收方法。有数据收到应该是驱动程序来通知系统的。一般设备收到数据后都会产生一个中断，在中断处理程序中驱动程序申请一块 `sk_buff`，从硬件读出数据放置到申请好的缓冲区里。接下来填充 `sk_buff` 中的一些信息。最后调用 `netif_rx()` 函数把数据传送给上层协议层处理。

网络驱动程序常用的系统支持如下。

(1) 内存申请和释放。

在 `include/linux/kernel.h` 中声明了 `kmalloc()` 和 `kfree()` 函数，用于在内核模式下申请和释放内存。

```
void* kmalloc(unsigned int len, int priority);  
void kfree(void* __ptr);
```

`kmalloc()` 函数申请空间有大小限制，长度是 2 的整次方。可以申请的最大长度也有限制。

(2) request_irq()和 free_irq()函数。

```
int request_irq(unsigned int irq, void(* handler)(int irq, void* dev_id, struct pt_regs* regs), unsigned long irqflags, const char* devname, void* dev_id);
```

申请中断函数。其中,irqflags 表示中断的属性。如果有多个设备共享同一个中断号,那么所有共享该中断号的设备都必须将 irqflags 设置为 SA_SHIRQ 属性。

```
void free_irq(unsigned int irq, void* dev_id);
```

释放中断函数。

(3) I/O。

每个驱动应该避免误用 I/O 端口。有关端口的内核函数如下。

- check_region(int io_part, int off_set)。参数 1 为 I/O 端口的基址,参数 2 为 I/O 端口偏移量。函数功能是通过查看系统的 I/O 表来检查是否有别的驱动程序占用该 I/O 端口。
- request_region(int io_part, int off_set, char* devname)。参数 1 为 I/O 端口的基址,参数 2 为 I/O 端口偏移量,参数 3 为设备名。函数功能是设备申请 I/O 端口,如果该端口未被占用,则申请成功。在 /proc/ioprots 文件中可看到已登记的 I/O 端口。

三、实验内容

分析教材提供的实例,编写一个简单的网络设备驱动程序。要求该设备包括 open()、request()、ioctl()和 release()等基本操作。并编写一个测试程序来测试所编写的设备驱动程序。

四、实验指导

将网络设备驱动程序加载,进行网络数据包的发送和接收,观察驱动是否可用。

根据功能的划分,网卡驱动程序主要有如下功能。

- (1) 打开,关闭,初始化。
- (2) 数据包传输和接收,读,写。
- (3) ioctl,参数设置,地址解析。
- (4) 中断处理。

五、实验思考题

- (1) 编写一个 USB 接口的摄像头驱动程序。
- (2) 编写一个 USB 无线网卡驱动程序,使该 USB 无线网卡能正常工作。
- (3) 找一个路由器驱动程序源代码并进行分析,然后说明如何编写路由器驱动程序。

参考文献

1. 周明德. 保护方式下的 80386 及其编程. 北京: 清华大学出版社, 1993
2. 汤子瀛、哲凤屏、汤小丹. 计算机操作系统, 第二版. 西安: 西安电子科技大学出版社, 1996
3. Uresh Vahalia. UNIX 高级教程系统技术内幕. 清华大学出版社, 1999
4. 罗宇, 邹鹏, 吴刚. 操作系统. 北京: 电子工业出版社, 2003
5. 费翔林, 骆斌, 谢立. 操作系统教程. 北京: 高等教育出版社, 2003
6. 何炎祥, 李飞, 李宁. 计算机操作系统. 北京: 清华大学出版社, 2004
7. 蒋静, 徐志伟. 操作系统原理·技术与编程. 北京: 机械工业出版社, 2004
8. 骆耀祖主编. Linux 操作系统分析教程. 清华大学出版社, 北京交通大学出版社, 2004
9. 王克义, 鲁守智, 蔡建新, 王文保. 微机原理与接口技术教程. 北京: 北京大学出版社, 2004
10. 徐德民. 操作系统原理——Linux 篇. 北京: 国防工业出版社, 2004
11. 徐虹, 何嘉, 张钟澍. 操作系统实验指导——基于 Linux 内核. 清华大学出版社, 2004
12. 孟庆昌. 操作系统. 北京: 电子工业出版社, 2005
13. 刘胤杰, 岳浩等. Linux 操作系统教程. 机械工业出版社, 2005
14. Daniel P. Bovet, Marco Cesati. Understanding the Linux Kernel, 3rd Edition. O' Reilly November 2005
15. 王星, 胡爱群, 张天放, 王质礼. Linux 下网络设备驱动程序的研究与开发[J], 微计算机应用. Vol. 26 No. 5 Sept, 2005
16. 蔡斌, 万柳. 基于 Linux 的网络设备驱动程序的分析[J], 计算机应用. Vol. 27 No. 4 July, 2006
17. 陈向群, 杨芙清. 操作系统教程(第 2 版). 北京: 北京大学出版社, 2006
18. 陈莉君, 康华. Linux 操作系统原理与应用. 北京: 清华大学出版社, 2006
19. Jonathan Corbet, Alessandor Rubini & Greg Kroah-Hartman 著. 魏永明, 耿岳, 钟书毅译. Linux 设备驱动程序(第三版). 北京: 中国电力出版社, 2006
20. (美)Robert Love 著. 陈莉君, 康华, 张波译. Linux 内核设计与实现(第 2 版). 北京: 机械工业出版社, 2006
21. 陈忠文主编. Linux 操作系统实训教程. 中国电力出版社, 2006
22. 孟庆昌, 牛欣源. Linux 教程(第 2 版). 电子工业出版社, 2007
23. Corbet, J&A. Rubini, G. K. Hartman. Linux Device Drivers(third edition)[M], O'Reilly, 2005