

操作系统原理 Linux 篇

操作系统原理 Linux 篇.....	1
第 1 章 操作系统概述.....	4
1.1 操作系统的地位及作用.....	4
1.2 操作系统的功能.....	5
1.3 操作系统的分类.....	7
第 2 章 Linux 概述.....	9
2.1Linux 的发展及背景.....	9
2.2Linux 的性能和特点.....	10
2.3Linux 内核.....	11
2.4Linux 下常用命令介绍.....	12
2.5 Linux 下程序设计基础.....	12
第 3 章 进程管理.....	17
3. 1 进程的基本概念.....	17
3. 2 进程状态和进程控制.....	18
3. 3 进程状态和进程控制.....	22
3. 4 进程的同步和互斥.....	24
3. 5 P、V 操作.....	25
3. 6 进程通信.....	32
3. 7 死锁.....	34
第 4 章 Linux 进程管理.....	37
4. 1Linux 进程概述.....	37
4. 2Linux 进程的状态和标识.....	40
4. 3Linux 的进程调度.....	44
4. 4 Linux 进程的创建和撤销.....	47
4. 4 Linux 信号.....	52
4. 4 Linux 管道.....	56
4. 7 IPC 信号量机制.....	60
4. 8 IPC 消息队列.....	67
4. 9 IPC 共享内存.....	74
第 5 章 存储管理.....	79
5. 1 存储管理的目的和功能.....	79
5. 2 地址重定位.....	80
5. 3 分区存储管理.....	81
5. 4 分页存储管理.....	87
5. 5 存储扩充技术.....	89
5. 6 分段存储管理.....	92
5. 7 段页式存储管理.....	94
第 6 章 Linux 存储管理.....	97
6. 1 80x86 的分段机制.....	97
6. 2 选段符与段描述符.....	100
6. 3 80x86 的分页机制.....	102
6. 4 Linux 的分段和分页结构.....	103
6. 5 Linux 进程地址空间管理.....	107
6. 6 Linux 物理空间管理.....	110

6. 7 内存的分配与释放.....	113
第7章 文件管理.....	114
7. 1 文件与文件系统.....	114
7. 2 文件的组织结构.....	115
7. 3 文件的目录结构.....	117
7. 4 文件存取与操作.....	119
7. 5 文件存储空间的管理.....	120
7. 6 文件的共享和保护.....	123
第8章 Linux 文件管理.....	124
8. 1Linux 文件系统概述.....	125
8. 2 EXT2 文件系统.....	126
8. 3 EXT2 的 inode 和文件结构.....	128
8. 4 虚拟文件系统 VFS.....	130
8. 5 文件系统的安装与注册.....	132
8. 6 文件管理和操作.....	134
第9章 设备管理.....	137
9. 1 设备与设备管理.....	137
9. 2I/O 控制方式.....	138
9. 3 缓冲技术	141
9. 4 设备的分配	142
9. 5 设备处理程序与 I/O 进程	144
第10章 Linux 设备管理.....	145
10. 1Linux 设备分类与识别.....	145
10. 2 设备驱动程序与设备注册.....	146
10. 3Linux 的 I/O 控制方式.....	149
10. 4Linux 设备 I/O 操作.....	151

第 1 章 操作系统概述

1.1 操作系统的地位及作用

1.1.1 操作系统的地位

计算机有软、硬件组成,如果没有软件,则计算机就等于一堆废铁。

软件可分为:

系统程序——管理计算机和应用程序(其中操作系统是其核心);

应用程序——解决用户问题的程序。

为了说明各个组成部分在计算机系统中的地位和作用,以及各个部件之间的关系,通常使用系统视图进行形象的描述,下图 1.1 给出了计算机系统视图。

图 1.1 计算机系统视图

从计算机系统视图中可以清楚地看到计算机系统的**层次结构**。计算机系统各个层次之间存在着一种单向服务的关系,即每一个内层都向其外层提供了一组**接口**。这里提到的“接口”与计算机硬件之间的硬接口在概念上虽然是相同的,但是它们并不像硬接口那样通过硬件的电气连接完成其功能。而是由指令、程序和数据结构等形成的一种接口,通常把这种接口称为“软接口”。通过软接口,内层以事先约定好的方式为外层提供服务,外层则通过该接口使用内层提供的服务来完成本身的功能。

下面从内向外简要说明各个层次的特性以及层次间接口的作用。

1. 硬件系统

硬件系统指组成计算机基本结构的**5 个部分**,即运算器、控制器、主存储器以及输入设备和输出设备。

运算器和控制器通常集成在一个芯片上,成为中央处理器(CPU)。CPU 是执行程序时进行运算和控制的装置,它直接控制着计算机各个部件的工作,是硬件系统的核心。

主存储器(内存)是存放系统中运行的程序和数据部件。

输入输出设备(外围设备)是用于实现计算机系统与外界信息交换的各种硬件设备。

硬件是操作系统存在的物质基础。**硬件层提供给操作系统的接口是机器的指令系统。操作系统的程序使用指令系统提供的机器指令所提供的机器指令所具有的功能,实现对硬件的直接管理和控制。**

2. 操作系统

操作系统是靠近硬件的软件层,其功能是直接控制和管理系统资源(包括软件、硬件)。计算机系统的硬件在操作系统的管理和控制下,其功能得以充分发挥。从用户观点看,引入操作系统后,计算机系统成为一台硬件系统功能更强、服务质量更高、使用更方便的机器。操作系统与其他系统软件一起向用户提供了一个良好的工作环境,用户无需了解许多与硬件和系统软件的细节,就能方便的使用计算机。

操作系统在硬件系统上运行,它常驻内存内,并提供给上层两种接口:操作接口和编程接口。操作接口由一系列操作命令组成,用户通过操作接口可以方便地使用计算机。编程接口由一系列的系统调用组成各种程序可以使用这些系统调用让操作系统为其服务,并通过操作系统来使用硬件和软件资源。所以其他程序是在操作系统提供的功能基础上运行的。

3. 系统应用软件

系统应用层由一系列的语言处理程序和系统服务程序构成。这些程序不是常驻内存的，而是存放在磁盘或其他外存储设备上，仅当需要运行这些程序时，才把它们装入内存。应用程序的主要功能是为用户编制应用软件、加工和调试程序以及处理数据提供必要服务。图 1.1 中给出了组成系统应用软件的几种典型程序。

系统应用层程序在操作系统的支持下工作，它们一般都使用机器指令以及操作系统提供的系统调用来编制程序。对上层它们提供了编制源程序的语句和语法或调试命令、系统维护命令等。

系统应用层层的程序有效地扩充了计算机系统的功能。它们与操作系统一起组成系统软件整体，起到了简化程序设计、扩大计算机处理能力、提高计算机使用效率、充分发挥各种资源功能的作用。因此，可以把这些系统应用程序看作是操作系统功能的延伸，甚至可以把它们看作操作系统的一部分。但是它与操作系统的不同之处在于，其运行环境与普通用户应用程序一样，它们仍然要通过操作系统才能使用和控制系统资源。

4. 应用软件

计算机层次结构的最外层是应用程序。这些程序是计算机用户为了使用计算机完成某一特定工作，或者解决某一具体问题而编制的程序。这些软件主要是使用其下层的系统应用程序提供的服务来实现自己的特定功能。

1.1.2 操作系统的作用

主要体现在两方面：

1. 屏蔽硬件物理特性和操作细节，为用户使用计算机提供了便利

- 指令系统（成千上万条机器指令，它们的执行由微程序的指令解释系统实现的）。计算机问世初期，计算机工作者就是在裸机上通过手工操作方式进行工作。
- 计算机硬件体系结构越来越复杂。

2. 有效管理系统资源，提高系统资源使用效率

如何有效地管理、合理地分配系统资源，提高系统资源的使用效率是操作系统必须发挥的主要作用。**资源利用率、系统吞吐量两个重要的指标。**

计算机系统要同时供多个程序共同使用。操作解决资源共享问题！！如何分配、管理有限的资源是非常关键的问题！

操作系统定义：操作系统是计算机系统中最基本的系统软件，它用于有效地管理系统资源，并为用户使用计算机提供了便利的环境。

1.2 操作系统的功能

目前计算机系统的应用领域日益扩大，计算机系统的功能也越来越强，由于操作系统在计算机系统中的核心地位，所以计算机系统的功能在很大程度上由操作系统的功能来决定。

1.2.1 单道系统与多道系统

- 单道系统：在早期的计算机系统中，每次只能运行一个程序，一个程序常常就是一道用户作业，所以这种系统称为单道程序设计系统，简称单道系统。在单道系统的内存中，除了操作系统的程序外只能存放一道用户作业，处理机和所有的系统资源仅为这一道作业服务。

缺点：处理机运行时间的大量浪费，严重影响了整个计算机系统的使用效率。

- 多道系统：在计算机中同时存放若干道用户程序，这些程序轮流使用处理机在计算机中交替地运行。多道系统的技术核心是实现了处理机与设备的并行工作。最大限度地提高了处理机的使用效率。但是也使得操作系统的功能变得十分复杂。

在多道系统中操作系统必须实现有限资源在多个程序间的合理分配，并尽可能地提高系统资源的使用效率。

本课程教授的操作系统原理，主要针对多道系统介绍操作系统对系统资源有效的管理及其有关技术。

1.2.2 操作系统的功能

1) 处理机的管理：

- 进程控制
- 进程调度
- 进程通信
- 进程同步与互斥
- 死锁（dead lock）

2) 存储器的管理

- 存储分配
- 地址映射
- 存储保护
- 内存扩充
- 内存共享

3) 设备管理：

- 设备的分配
- 设备的管理和控制（设备驱动程序）
- 为用户使用设备提供统一的操作接口（只要指名设备，操作方式（读/写））
- 充分发挥设备和主机的并行工作能力（缓冲和虚拟技术）

4) 文件管理：

- 文件的组织（可以有效地分配和回收文件的存贮空间，存取文件时准确地定位）
- 文件的保护和共享
- 文件的操作与用户的接口

1.3 操作系统的分类

从操作系统所适应的硬件机器规模的角度：大型机操作系统、小型机操作系统和微型机操作系统；

从使用计算机系统的用户数目的角度：单用户操作系统和多用户操作系统。

例如：MS-DOS 是一个单用户单任务微型机操作系统，MS-Windows 是一个单用户多任务操作系统，而 UNIX、Linux 是多任务多用户操作系统。

从操作系统的设计目标及其功能，通常把操作系统分为 3 类：批处理操作系统、分时操作系统和实时操作系统。

1.3.1 批处理操作系统

为了减少人为干预，提高计算机的利用率，把需要机器运行的若干程序按一定顺序组织在一起，然后成批地交给计算机，让计算机自动地、按顺序逐个运行各个程序。又分为单道批处理系统和多道批处理系统。

没有人机交互。

主要设计目标：减少人为干预，提高计算机的利用率。

1.3.2 分时操作系统

分时系统，一台计算机主机连接多个终端机，每个用户使用一台终端运行自己的程序。所谓分时，就是把处理机的时间分成若干小的时间片，把每个时间片轮流分配给各个程序。时间片的长短由系统确定。

特征：同时性、独立性、及时性、交互性。

主要设计目标：

- 1、及时响应用户请求，所以响应时间是分时系统的衡量指标。
- 2、提高资源利用率

分时操作系统主要用于小型机、工作站和高档微型机。UNIX 就是一个分时系统。

1.3.3 实时操作系统

实时就是“立即”、“现在”的意思，是指对随时发生的外部事件能及时做出响应和处理。

实时操作系统可分为两类：实时控制系统和实时信息处理系统。

主要设计目标：实时响应及处理能力高、高可靠性、安全性。

实时操作系统也可以连接多个终端，各个终端也可以与系统发生交互，但是它与分时系统不同，其差别主要体现在以下 3 个方面：

(1) 目标不同。

分时系统：通用性较强的计算机系统；

实时系统：提供一种特殊用途的专用计算机系统。

(2) 交互能力不同。

分时系统具有较强的交互会话能力，可以运行任何用户程序，可以应用户的不同请求给予响应。
实时系统中的应用程序是预先设计好的，只能响应预先约定好的用户请求。

(3) 响应时间不同。

分时系统：以人能就接受的程度来确定响应时间，通常是秒数量级；

实时系统：以控制过程或信息处理过程所能容忍的延迟来确定，通常是毫秒或微秒数量级。

(4) 可靠性不同。

1.3.4 其他操作系统

通用操作系统：为了使计算机系统适应范围更广、处理能力更强，有些系统兼有实时、分时和批处理中的两种或三种处理能力，从而形成了通用操作系统。在具有三种处理能力的系统中，实时任务处理级别最高；在没有实时任务的情况下，处理分时任务；如果没有分时任务，则再执行批处理任务。

网络操作系统：是实现网络通信与网络资源管理的操作系统。

分布式操作系统：分布式系统是由多个分散的计算机网络连接而成的统一的计算机系统。但是它具有与计算机网络不同的特征：分布性、并行性、透明性、共享性（每个计算机上的资源都可供共享）、健壮性。参见（清华 范策）

嵌入式操作系统：嵌入式系统是用于控制、监视或者辅助操作机器和设备的装置。嵌入式操作系统大多用于机电设备、仪器等上的专用控制方面，它大多采用**微内核结构**。

第 2 章 Linux 概述

2.1 Linux 的发展及背景

2.1.1 Linux 的发展历史

- 1、1991 年，年轻的荷兰大学生 Linus B.Torvalds 在 PC 机上开发出的一个简单的操作系统内核程序，由此揭开了 Linux 发展的历史。
- 2、1994 年 3 月，由 Linus 领导的世界各地的爱好者共同开发的第一个功能完整、性能稳定的 Linux 内核版本 Linux1.0 问世。
- 3、为了方便用户安装和使用 Linux 操作系统，一些商业软件公司把 Linux 内核与各种实用程序，如编译器、编辑器、窗口管理等组合在一起，形成了各种发行套件，这就是当前出现的各种不同名称的 Linux 发行版本，如 Red Hat Linux、Slackware Linux、Turbo Linux、Debian Linux、Xteam Linux、红旗 Linux 等。

Linux 目前已经成为一个备受关注的稳定的、健壮、功能强大的多用户多任务操作系统，它可以在诸如 i386、Sparc、Alpha、Mips、M68k、PPC 等众多的计算机平台上运行。

2.1.2 Linux 与 GNU

当前国际上提供给用户使用的计算机软件有 3 种形态：商业软件（commercial software）、共享软件（Shareware）和自由软件（freeware 或 free software）。

- 商业软件：有软件开发商出售并提供技术服务，用户只有使用权，不允许进行非法拷贝、扩散和修改。
- 共享软件：由软件开发者向用户免费提供软件试用版并规定一定的试用期，用户在试用期结束后若继续使用，必须向开发者支付费用后，开发者才继续向用户提供升级软件和技术服务。
- 自由软件：由开发者提供软件全部源代码，任何用户都有使用、拷贝、扩散、修改该软件，同时用户也有义务将自己修改过的程序代码公开。

1984 年，在自由软件的积极倡导者 Richard Stallman 的组织下，提出了基于自由软件思想的 GNU（GNU's Not UNIX，不是 Unix 的 Unix）的计划。开发这个系统的目的是为了所有计算机用户都可以自由获得和免费使用这个系统，任何人都可以免费获得这个系统的源代码，并可以相互自由拷贝。为了推行 GNU，Richard Stallman 建立了美国自由软件基金会 FSF（Free Software Foundation），并制定了一份公用版权协议 GPL（General Public License）。GNU 也有自己的版权声明，称为 Copyleft，以区别于一般的 Copyright。

在 GNU 的实施过程中，推出了许多著名软件，如 GCC。

Linus 最初在互联网上推出 Linux 的时候，是按完全自由软件的思想进行源代码扩散的。他要求所

有的 Linux 的源代码必须公开，而且任何人均不得从 Linux 交易中获利。然而很快意识到这样做，实际上阻碍了 Linux 的发展，于是他决定转向具有 GPL 版权的 GNU。GPL 确定了 Linux 自由版权的地位，除了规定有自由软件的各项许可权之外，还允许用户出售自己的程序拷贝。

事实证明，Linux 在版权上的转变对于 Linux 的进一步发展起到了十分重要的作用。

2.2Linux 的性能和特点

2.2.1 Linux 的优越性能

1. 与 UNIX 兼容的强劲功能。

Unix 是 20 世纪 70 年代由贝尔实验室推出的小型机操作系统。80 年代经过多所大学、研究所、工业实验室的完善和发展，成为世界各地计算机网络通信系统和开发工作站的主流操作系统。Linux 参照 POSIX（Portable operating System Interface for Unix）1003.1 标准开发，所以它与 Unix 在功能上完全兼容，而且具有最新的扩展功能，在 Unix 上运行的软件大部分不需要做任何修改就可以在 Linux 上运行。

2. 简单廉价的运行条件。

Unix 虽功能强大，但是对运行环境要求较严格，通常运行在小型机和高档微机上，此外 Unix 系统价格较高。而 Linux 解决了这些问题。

3. 性能完善的网络功能。

Unix 成为主流操作系统，是因为它具有完善的网络管理功能。Linux 不但继承了 UNIX 全部的网络管理功能，而且进一步予以加强和完善。Linux 内核支持 Ethernet，PPP，SLIP，NFS，AX.5，IPX/SPX（Novell），NCP（Novell）等网络协议。

4. 可任意裁减的内核

计算机技术人员可以根据应用的需要只使用内核的一部分功能，也可以在内核中增加新功能。Linux 系统最小可以裁减到只有 1.4MB。Linux 的这个特点为嵌入式系统开拓了美好的前景。

5. 完善的技术支持体系

由于 Linux 主要通过互联网进行推广和传播，所以 Linux 的技术支持体系更具特色。

以下列举几个 Linux 著名的英文网站：

<http://www.linuxhp.com/>

<http://src.doc.ic.ac.uk/>

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

下载 Linux 内核的 FTP 服务器有：

<ftp.cs.helsinki.fi:/pub/software/linux/kernel>

<ftp.funet.fi:/pub/linux/PEOPLE/Linus>

<ftp.kernel.org>

下载 Linux 运行所需要的工具程序服务器有：

<Sunsite.unc.edu:/pub/Linux>

<Tsx-11.mit.edu:/pub/Linux>

2.2.2 Linux 的技术特点

1. 多用户多任务
2. 可靠的保护机制

Linux 提供的保护机制:

- 在运行机制上, Linux 提供了两种执行状态, 用户态 (自己内存空间, 用户指令) 和内核态 (全部内存空间, 全部指令)。
- 认证监督机制。系统中的每个用户都有自己唯一的 ID 标识。(一般用户, 超级用户)
- 存储保护机制

3. 多平台

Linux 内核源代码使用的是高级语言 C 语言, 所以它的移植性很强。

4. 设备独立于内核
5. 支持多种文件系统

由于 Linux 采用了虚拟文件系统 VFS, 所以它可以支持多种不同的物理文件系统。

6. 完善的虚拟存储技术
7. 支持多种硬件设备

由于 Linux 自由软件的性质, Linux 程序人员和设备制造商都为 Linux 系统开发了大量的设备驱动程序。

2.3Linux 内核

2.3.1Linux 内核的版本

Linux 软件分为**两种版本**: 一种是 Linux 内核的版本, 如 Linux2.0.2、Linux2.4.7 等; 另外一种是 Linux 发行套件的版本。发行套件的版本号与内核的版本号是相对独立的, 它们由发布者自行规定, 例如 RedHat5.1, RedHat6.0, RedHat9.0 等。各软件公司发行套件的版本编号方式虽然不同, 但它们使用的内核的版本是统一的。

Linux 的内核版本号由 3 组数字组成, 形式为 x.yy.zz, 例如, linux2.0.35, linux2.2.1 等。其中:

- 第一组数字 x 是主版本号, 当前是 0~2, 它表示 Linux 内核在功能上有重大改进的不同阶段。
- 第二组数字 yy 是次版本号, 从 0~99, 其中偶数号表示内核经过改进是稳定的; 而奇数号表示内核还处于开发过程中, 是实验性的源代码, 本身并不稳定。
- 最后一组数字 zz 是修订号, 从 0~99, 它表示各主次版本的增补级。例如 Linux2.0.35, 修订号是 35, 表示该版本经过 35 次增补。修订号的增加只表示排除了发现的内核的缺陷, 内核的功能并未改变, 如从 Linux2.0.2 到 Linux2.0.3 内核的功能相同的。

2.3.2 Linux 内核的组成及功能

Linux 采用模块化程序设计方法，其内核由若干功能相对独立的程序模块组成。其主要优点在于对内核功能的增加和修改十分方便，而且任何一个模块的改动都不会影响其他模块的功能。

Linux 内核主要由 5 个子系统组成：进程管理，存储管理，文件管理，网络管理，进程间通信。图 2.1 给出了 Linux 内核组成示意图和各个子系统之间的关系。

2.4 Linux 下常用命令介绍

2.5 Linux 下程序设计基础

Linux 内核编程实战经验谈

蓝森林 <http://www.lslnet.com> 2001 年 6 月 16 日 18:09

作者：李艳彬

当前，在国产自主知识产权的操作系统这面大旗的倡导下，IT 界掀起了一浪高过一浪的 Linux 编程热潮。Linux 以其源码开放、配置灵活等不可多得的优越性吸引着越来越多的编程爱好者深入 Linux 的内核开发。笔者近来实践过一个 Linux 的实时化改造课题任务，积累了一点 Linux 内核编程的实战经验，在这里想就编译内核、增加系统调用等方面的问题和感兴趣的爱好者共做切磋。

编译内核

在 Linux 编程的实践中，经常会遇到编译内核的问题。为什么要编译内核呢？

其一，可以定制内核模块。Linux 引入了“动态载入模块”的概念，使用户可以把驱动程序以及非必要的内核功能代码编译成“模块”，由系统在需要时动态载入，不需要时自动卸载，从而提高了系统的效率和灵活性。

其二，可以定制系统功能。当添加某种设备时、增加系统功能时、系统暴露出缺陷需要打“补丁”时，当新版内核出现准备用来升级时，编译内核是不可避免的。而且，编译内核正是 Linux 独有的“系统级 DIY”的魅力所在！

好，现在就让我们一起开始——编译内核！

(1) 安装源码

首先要确定自己 Linux 系统是否已安装了内核源码：

```
# rpm -q kernel-source
kernel_source-2.2.5-16
```

如果证实没有安装，则需要找来安装盘或从网上下载 kernel-source-2.2.5-15.i386.rpm 并安装：

```
# rpm -Uhv kernel-source-2.2.5-15.i386.rpm
```

如果是升级到新版本，则需要找来升级包（linux-2.2.16.tar.gz），自己解压安装：

```
# cd /usr/src
```

进入源码目录。

```
# rm -rf linux
```

删除以前的链接。

```
# tar xzvf linux-2.2.16.tar.gz
```

解压升级包。

```
# ln -s linux-2.2.16 linux
```

重建目录链接。

（2）配置内核

进入内核源码所在目录：

```
# cd /usr/src/linux
```

先清除多余的（一般是以前编译生成的）文件：

```
# make mrproper
```

开始配置内核（如果对各选项不是很熟悉的话，建议按回车键）：

```
# make config
```

（3）编译内核

清除以前生成的目标文件及其他文件：

```
# make clean
```

理顺各文件之间的依存关系：

```
# make dep
```

编译压缩的内核：

```
# make bzImage
```

编译模块：

```
# make modules_install
```

（4）装新内核

将新内核文件复制到用于存放启动文件的 /boot 目录：

```
# cp /usr/src/linux/System.map /boot/System.new
```

```
# cp /usr/src/linux/arch/i386/boot/bzImage /boot/vmlinuz.new
```

进入启动目录：

```
# cd /boot
```

给新内核建立链接：

```
# rm System.map
```

```
# ln -s System.new System.map
```

```
# rm vmlinuz
```

```
# ln -s vmlinuz.new vmlinuz
```

编辑 LIL0 的配置文件 /etc/lilo.conf ，使 LIL0 能启动新内核：

```
# vi /etc/lilo.conf
```

在文件末加入以下部分：（后两行内容要与旧内核相应行保持一致）

```
image=/boot/vmlinuz.new
```

```
lable=new
```

```
root=/dev/hda3
```

```
read-only
```

重写 LIL0 的启动扇区，使改动生效：

```
# lilo
```

（5）重启系统

```
# reboot
```

当重启后出现 lilo: 提示时输入新内核的标号（按 TAB 键可显示所有的标号）：

```
lilo: new
```

```
OK!! boot new.....
```

```
.....
```

一切运行正常，新内核引导成功！

以上步骤在 pentium III/64M/20G、Red Hat Linux 6.0 (2.2.5-15) 机上测试通过。

增加系统调用

在实际编程中，尤其是当我们需要增加或完善系统功能的时候，我们经常会用到系统调用函数。系统调用函数通常由用户进程在用户态下调用，内核通过 system_call 函数响应系统调用产生的软中断，在正确访问核心栈、系统调用开关表之后陷入到操作系统内核中进行处理。

系统调用是用户进程由用户态切换到核心态的一种常见方式。利用编写系统调用函数来直接调用了部分操作系统内核代码，也是 Linux 内核编程者必修之功。下面笔者以在 Linux 中创建一个名为 print_info 的系统调用函数为例，来说明如何为内核增加系统调用。

需要以下几个基本步骤：

1、编写系统调用函数

编辑 sys.c 文件：

```
# cd /usr/src/linux/kernel
```

```
# vi sys.c
```

在文件的最后增加一个系统调用函数：

```
asmlinkage int sys_print_info(int testflag)
{
    printk(" Its my syscall function!\n");
    return 0;
}
```

该函数有一个 int 型入口参数 testflag, 并返回整数 0。

2、修改与系统调用号相关的文件

编辑入口表文件：

```
# cd /usr/src/linux/arch/i386/kernel
```

```
# vi entry.S
```

把函数的入口地址加到 sys_call_table 表中：

arch/i386/kernel/entry.S 中的最后几行源代码修改前为：

```
.....  
.long SYMBOL_NAME(sys_sendfile)  
.long SYMBOL_NAME(sys_ni_syscall) /* streams1 */  
.long SYMBOL_NAME(sys_ni_syscall) /* streams2 */  
.long SYMBOL_NAME(sys_vfork) /* 190 */  
rept NR_syscalls-190  
.long SYMBOL_NAME(sys_ni_syscall)  
.endr
```

修改后为：

```
.....  
.long SYMBOL_NAME(sys_sendfile)  
.long SYMBOL_NAME(sys_ni_syscall) /* streams1 */  
.long SYMBOL_NAME(sys_ni_syscall) /* streams2 */  
.long SYMBOL_NAME(sys_vfork) /* 190 */  
.long SYMBOL_NAME(sys_print_info) /* added by I */  
.rept NR_syscalls-191  
.endr
```

修改相应的头文件：

```
# cd /usr/src/linux/include/asm
```

```
# vi unistd.h
```

把增加的 sys_call_table 表项所对应的向量，在 include/asm/unistd.h 中进行必要申明，以供用户进程和其他系统进程查询或调用。

```
#define __NR_putpmsg 189  
#define __NR_vfork 190  
#define __NR_print_info 191 /* added by I */
```

3、编译内核，再重新启动

4、测试

编写用户测试程序 (test.c)：

```
# vi test.c
```

```
#include
```

```
#include
```

```
extern int errno;
```

```
_syscall1(int,print_info,int,testflag)
```

```
main()
```

```
{
```

```
int i;
```

```
i= print_info(0);  
if(i==0)  
printf("i=%d , syscall success!\n",i);  
}
```

如果要在用户程序中使用系统调用函数，那么在主函数 `main` 前必须申明调用 `_syscall`，其中 1 表示该系统调用只有一个入口参数，第一个 `int` 表示系统调用的返回值为整型，`print_info` 为系统调用函数名，第二个 `int` 表示入口参数的类型为整型，`testflag` 为入口参数名。

编译测试程序：

```
# gcc -o test test.c
```

执行测试程序：

```
# ./test
```

```
Its my syscall function!
```

```
i=0, syscall success!
```

```
ok!!!增加系统调用函数成功!
```

以上步骤在 `pentium III/64M/20G`、`Red Hat Linux 6.0 (2.2.5-15)` 机上测试通过。

第3章 进程管理

3.1 进程的基本概念

3.1.1 程序的顺序执行

程序执行时，必须按照某种先后顺序逐步进行。（单道程序系统中）

特点：顺序性、封闭性、可再现性。

3.1.2 程序的并发执行

若干逻辑上相互独立的多道程序在计算机中交替执行。（多道程序）

特点：

1. 间断性
2. 失去程序封闭性
3. 具有不可再现性

例：P31-32 出现三种情况

- a. A 在 B 前，N 的值为： $n+1; n+2; 0$ 当 n 的初值为 0 $(1, 2, 0)$
- b. A 在 B 后，N 的值为： $0; 0; 1$ $(0, 0, 1)$
- c. A 在 B 之间，N 的值为： $0; 1; 0$ $(0, 1, 0)$

3.1.3 进程的定义和特征

1. 进程的概念
进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动，它是进行系统资源分配、调度的一个独立单位。
2. 进程的五大特征 P33
 - a. 动态性：程序的一个执行过程，运动中的程序。
 - b. 并发性：建立进程的程序是可并发执行的。
 - c. 异步性：并发程序相互制约。
 - d. 独立性：在一个数据集上，作为一个基本单位，具有独立性。
 - e. 结构性：每个进程都由其相对应的数据结构及某独立表项。
进程（程序、数据、PCB）

} 基本特征

3. 引入进程的利弊

利：多道程序并行执行，改善了系统资源的利用率，提高了系统的吞吐量

弊：1) 空间开销

2) 时间开销

3. 2 进程状态和进程控制

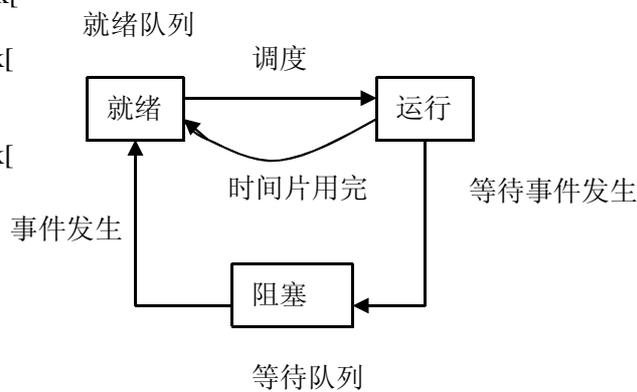
3.2.1 进程的状态及转换

1. 代表进程生命周期的三种状态

{ Empty：表已经获得初处理器以外的，所有运行所需的资源的进程。
 示空缓冲单已经获得处理器和其他资源，正在执行的进程
 运行的总数
 阻塞 =n 正在运行的进程，因某种原因而暂停运行，等待某个事件的发生

```

Think;
V(Chopstick[
i+1])
V(Chopstick[
i])
P(Chopstick[
i+1])
Eat;
P(Chopstick[
i])
thinkeri
1
5
4
3
2
1
2
5
3
4
eat
Think
thinker
Perform
write
operation
  
```



```

V(wmutex)
P(wmutex)
writer
Readcount=0
V(rmutex)
Readcount-
  
```

V(wmutex)

P(rmutex)

A

A. 状态间的演变 (4 种情况)

Perform

3.2.2 进程的实体 operation

V(rmutex)

1. 进程的实体

Readcount
I 2. 进程控制块 (Process Control Block——PCB)

P(wmutex) { 程序 进程运行时对应的执行代码

{ 数据集 运行时必需的数据资源

Readcount=0
进程数据结构 记录进程存在, 保持进程所需数据集合, 完成进程控制的重要结构

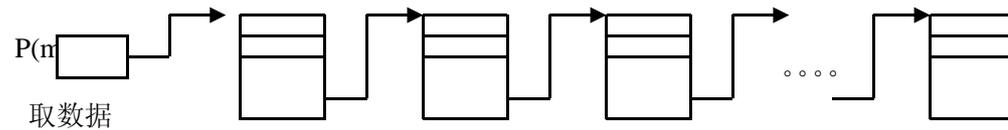
P(rmutex)

● 定义: PCB 是系统为描述和控制进程的运行而为每个进程定义的一个数据结构, PCB 中记录了操作系统所需全部的关于进程的描述和控制信息, 它是进程存在的唯一标志。

● PCB 结构 (内容) 标识信息、调度信息、处理机信息、控制信息

● PCB 的组织方式 (根据不同的状态来组织): 索引表结构, 或链接表结构。

P(full)



3.2.2 进程控制

V(empty)

进程控制的任务就是通过控制进程的状态的变化, 使各个进程有条不紊地推进, 从而实现对系统中全部进程的有效管理。进程控制是由进程使用操作系统提供的一组系统调用来实现的, 且这些系统调用都具有原语的特性, 故也称为进程控制原语。

➤ 原语: 是执行一定功能的程序段, 它的执行不可中断, 好像执行一条指令一样。原语具有原子操作性, 即一个操作的所有动作, 要么全做, 要么全不做。这种原子性通过屏蔽中断来实现。

送数据

1. 创建原语 (create)

V(mutex)
1) 主要功能: 为被创建的进程建立一个 PCB, 并填入相应的初始项。

2) 创建原语的主要操作 P39

a. 先向系统申请一个空闲的 PCB

b. 再根据父进程提供的参数将子进程的 PCB 的初始化

full: 将此 PCB 插入就绪队列

c. 将 PCB 插入就绪队列

d. 返回一个进程标识符

初值 = 0

Empty: 表

示空缓冲单

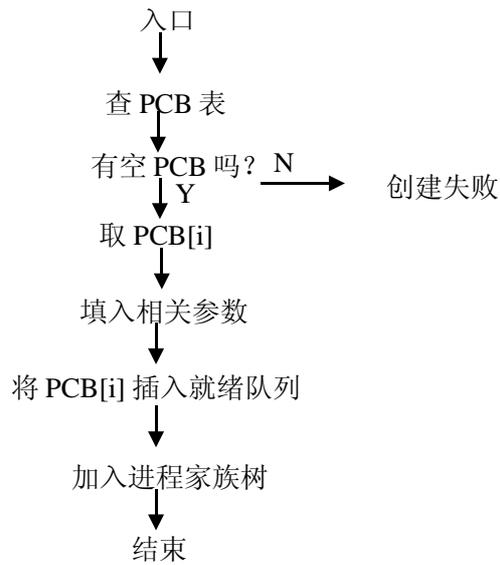
元的总数

初值 = n

有界缓冲区

n > 0

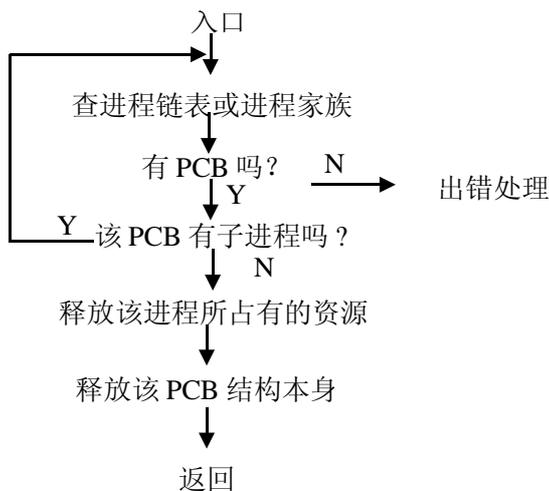
就绪



2. 撤销原语 (destroy)

- 1) 撤销原因: a 任务完成; b 发生错误; c 父进程被撤销
- 2) 主要功能: 收回被撤销的进程占用的所有资源, 并撤销它的 PCB。
- 3) 主要操作:
 - a. 找到要撤销的进程 PCB, 如果有子孙进程, 需要遍历进程树, 找到所有的子孙进程的 PCB
 - b. 释放及其子孙进程所占用的全部资源。
 - c. 删除其子孙进程。
 - d. 把进程 PCB 的从其所在的队列中删除, 并回收 PCB
 - e. 如果撤销的是运行进程, 则需要从就绪队列中选择一个进程投入运行。

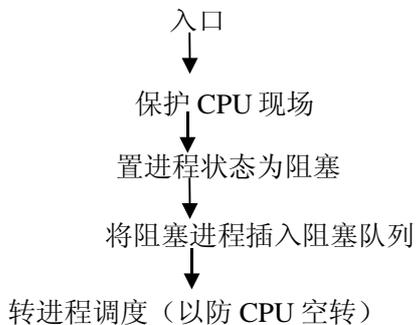
注: 撤销进程原语应由父进程或祖先进程发出。进程一般不能自己撤销自己。



3. 阻塞原语 (block)

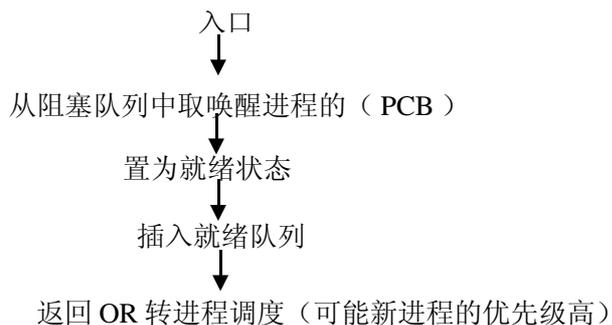
- 1) 主要功能: 将进程有执行状态或其他状态转为阻塞状态。

- 2) 挂起原因：资源缺乏、等待 I/O 操作等。
- 3) 挂起方式：进程将自身挂起、将指定标识的进程挂起、将某进程的全部或部分子进程挂起等。
- 4) 主要操作：如果该进程正处于运行状态，则
 - a. 中断处理机，保存该进程的 CPU 现场
 - b. 将该进程插入等待队列中
 - c. 从就绪队列中选择一个新的进程投入运行



4. 唤醒原语 (wakeup)

- 1) 主要功能：由“发现者”进程将处于阻塞状态的进程有阻塞状态变为就绪状态。（发现者进程与被唤醒的进程之间存在制约关系，或者是监视事件发生地系统进程）
- 2) 主要操作：
 - a. 从相应阻塞队列中查找等待事件的阻塞进程，改变被恢复的进程的 PCB 的内容
 - b. 把该进程从阻塞队列中撤出，并将该进程插入就绪队列。



3. 3 进程状态和进程控制

操作系统进程管理的核心任务是为了多个进程合理地分配处理机资源，这就是处理机调度。因为处理机调度是面向进程的，所以又称进程调度。

3.3.1 进程调度的功能

在系统运行过程中，由于多个进程需要轮流使用处理机，所以完成分配处理机任务的进程调度是系统中最频繁的工作，也是操作系统核心的重要组成部分。

进程调度的主要功能是：

- (1) 记录当前进程的情况，如进程名、指令计数器、状态寄存器及所有通用寄存器等现场信息，将这些信息记录在它的进程控制块中。
- (2) 根据一定的调度算法，确定就绪队列中哪一个进程能获得处理机，以及占有多长时间
- (3) 回收和分配处理机。当前进程转入适当的状态后，系统回收处理机，然后把处理机分配给调度算法选中的下一个进程。

3.3.2 进程调度性能标准

在系统完成进程调度时，最关键的是调度算法。操作系统的设计目标和工作效率主要进程调度算法性能决定。

面向用户的指标：周转时间短、响应时间快、截止时间的保证、优先权准则。

1. 响应时间：从用户提出请求，到系统首次产生响应所经历的时间。（常用于分时 OS）

2. 周转时间：（批处理系统）从作业提交给系统开始，到作业完成为止。

作业周转时间：外存+内存+处理器+I/O

进程周转时间：内存+处理器+I/O，进程从第一次进入就绪队列开始，到进程运行完毕所经历的时间。

带权周转时间：作业的周转时间与系统为他提供的实际服务时间之比。

3. 截止时间：（实时系统）：某任务必须开始执行的最迟时间。

4. 运行时间：处理器+I/O，指进程获得处理器，处于 CPU-I/O 的执行期

例：P1, P2, P3 单独运行时间 21, 6, 3

当运行方向为：P1, P2, P3。周转时间分别为：21, 27, 30。平均周转时间为 26；

当运行方向为：P3, P2, P1。周转时间分别为：3, 9, 30。平均周转时间为 14；

面向系统的指标：系统的吞吐量高、处理机的利用率好、各类资源的平衡利用

3.3.3 进程调度方式

进程调度方式是指系统把处理机分配给一个进程使用时，让进程如何占有处理机，以及它能占有多长时间。通常有两种方式：

1. 非剥夺方式：不能从正在运行的进程夺走处理器控制权，除非它运行完毕或因某种原因阻塞。
2. 剥夺方式：按某种原则，将正在运行的进程强撤销，并将处理器分配给其他就绪进程。
剥夺原则：1) 优先级原则；2) 短进程优先原则；3) 时间片原则；4) 强制性剥夺

3.3.4 进程调度算法

进程调度算法又称调度策略，它是指在就绪状态的进程中，按照什么原则选择一个进程，并把处理机分配给该进程使用。

主要解决的两个问题：

- a. 选择方式：选择哪个进程（随算法不同而不同）；
- b. 调度方式：选中它以后，如何给它分配 CPU，及其占用 CPU 的时间（剥夺、非剥夺）；

1. 先进先出（FIFO）

- a. 按照进程进入就绪队列的先后次序来分配处理机
- b. 非剥夺方式调度方法
- c. 优缺点

2. 短执行进程优先（SCBF）

- a. 按照就绪队列中的进程预期执行时间的长短来分配处理机
- b. 剥夺方式调度方法
- c. 优缺点

3. 优先级调度（最常用）

- a. 按照就绪队列中的进程的优先级的高低来分配处理机（优先权：静态、动态）
- b. 剥夺/非剥夺方式调度方法
- c. 优缺点

4. 时间片轮转法

系统将所有就绪进程按到达时间的先后顺序排成一个队列，依次轮流调度各个进程，调度算法从队列头选择一个进程执行，且仅执行一个时间片，当时用完一个时间片后，释放 CPU，并加入到就绪队列尾部，等待继续调度。

难点和关键：时间片的长短的选择。

5. 多级队列反馈调度算法

3. 4 进程的同步和互斥

进程间存在两种形式的制约关系：

- a. 直接相互制约：进程合作——同步关系（进程的推进速度之间的制约；进程间通信）
- b. 间接相互制约：资源共享——互斥关系

1. 几个概念

- 1) 临界资源：一次仅允许一个进程使用的资源（硬件、软件）。
- 2) 临界区：访问这段临界资源的那段程序。
- 3) 进程互斥：多个进程在共享临界资源时的相互制约关系。
- 4) 进程同步：系统中有多个进程共同完成一项任务，这些进程之间存在着直接制约关系，每一个进程要依赖于其他进程所产生的结果才能继续运行。共同完成一个任务的若干进程称为合作进程。

2. 进程同步和互斥的区别

- 1) 互斥进程在单独运行时都可以得到正确结果；同步进程不能单独运行。

- 2) 互斥进程不规定执行先后顺序；同步进程必须按照严格的顺序执行。
- 3) 互斥进程不知道对方的存在；同步进程不仅知道其他进程的存在，而且还要通过与其它进程的通信来达到相互的协调。
- 3. 同步机制必须遵循如下准则：
 - 1) 空闲让进
 - 2) 忙则等待
 - 3) 有限等待：避免‘死等’（死锁）
 - 4) 让权等待：退出临界区，立即释放处理机，让位给等待进程。避免‘忙等’。

3. 5 P、V 操作

3.5.1 信号量

- 1. 信号量的定义：信号量 S（Semaphore）是一个记录性变量。

```

Type semaphore=record
      value : integer;
      L : list of process;
end
  
```

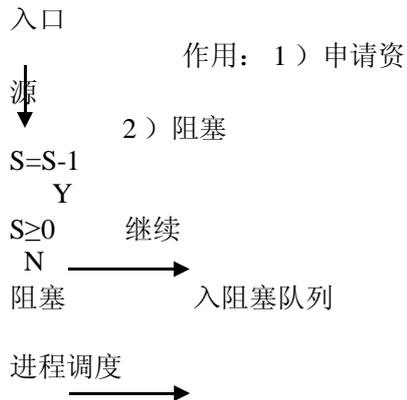
- 2. S.value 的取值及意义

初始值：表示系统中某类资源的数目	}	<ul style="list-style-type: none"> >0：表示系统当前可用的该类资源的数目 <=0：其绝对值表示系统中因请求该类资源而被阻塞的进程数目
------------------	---	--

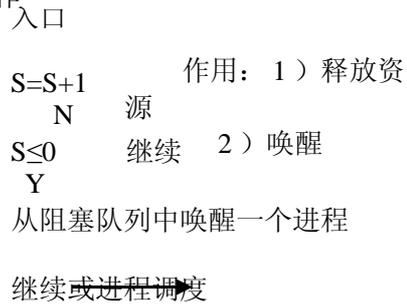
- 3. 信号量的操作

除初始化之外，信号量仅能由 P、V 两条原语，即 P、V 两种操作来改变。

P 操作



V 操作



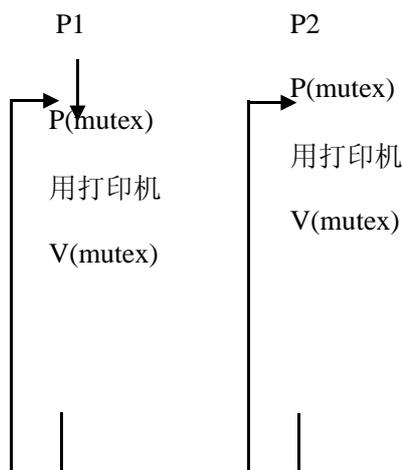
3.5.2 信号量的应用

1) 用信号量实现进程互斥

用信号量解决几个进程互斥进入临界区的问题，几个进程共享一个公用信号量 `mutex`（互斥信号量）。

每个进程进入临界区必须先执行 `P(mutex)`，退出临界区后执行 `V(mutex)`。对于 n 个进程同时共享一个临界资源，则 `mutex` 的取值为 $1 \sim -(n-1)$ 。

举例：有 1 台打印机 2 个进程，则 `mutex` 的取值为？



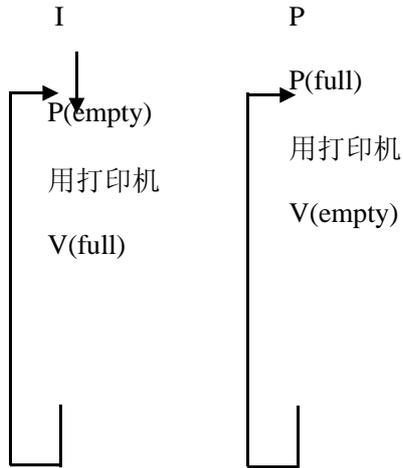
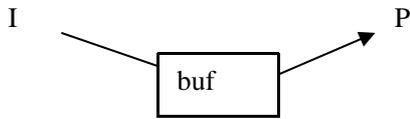
2) 用信号量实现进程同步 P47

设立与资源相关的私有信号量（资源信号量）。

设缓冲区只有一个缓冲单元

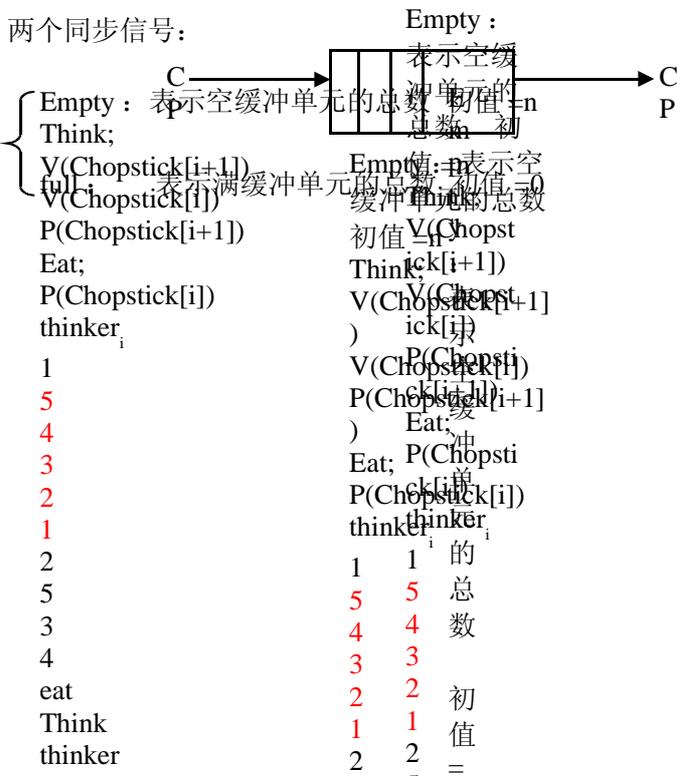
Empty=1 表示空单元个数。

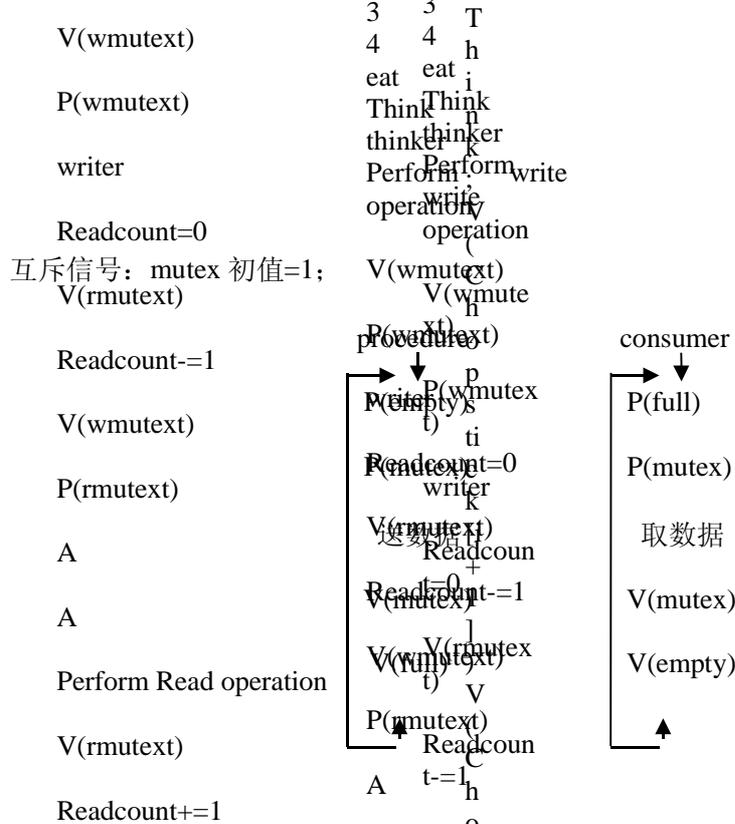
Full=0 表示装满信息的单元个数。



3.5.3 经典进程同步机制

1.“生产者——消费者问题”简称PC问题





注意:

1. 每个程序中实现互斥的 P(mutex), V(mutex) 必须成对出现。
2. Empty 和 full 的 P, V 操作必须成对出现, 但它们处于不同的程序中。
3. 每个程序中的多个 P 操作不能颠倒顺序, 否则可能引起死锁。

2. 读者—写者问题 (The Reader-Writer Problem)

问题描述:

一个数据文件或记录可被多个进程共享, 其中, 允许多个 Reader 进程同时读一个共享对象, 因为读操作不会使文件混乱, 但决不允许一个 writer 进程和其他 reader 或 writer 进程同时访问共享对象。所谓读者—写者问题, 是指保证一个 writer 进程必须和其他进程互斥地访问共享对象的同步问题。该问题首先在 1971 年由 Courtois 等人解决。

利用记录型信号量解决读者—写者问题:

设置变量:
 readcount: 记录正在读的进程的数目。
 设置信号量:
 wmutex: 保证 reader 与 writer 读、写时的互斥。
 Rmutex: 保证对 readcount 的互斥访问。

进程流程图: (pascal 描述见书)

```

procedure
    )
    取数据
    t=0
    a
    V(mutex)
    P(rmutex)
    )
    V(empty)
    送数据
    Reader
    h
    procedure
    consumer
    P(empty)
    P(full)
    P(mutex)
    P(mutex)
    full : 表示满缓冲单元的总数 初值 =0
    Empty : 表示空缓冲单元的总数 初值 =n
    有界缓冲区
    n>0
    V(mutex)
    V(mutex)
    V(full)
    h
    V(empty)
    n
    上表示进
  
```

缓冲单元的总数
 初值 $P(\text{empty})$
 Empty : 表示空
 缓冲单元的总数
 初值 $=n$
 有界缓冲区
 $n > 0$
 ---- $V(\text{mutex})$
 就绪 $V(\text{full})$
 4
 e
 full :
 表示满缓
 冲单元的
 总数 初
 值 $=0$
 Empty :
 表示空缓
 冲单元的
 总数 初
 值 $=n$
 有界缓冲
 区 e
 $n > 0$
 ---- P
 就绪
 r
 f
 o
 r
 m

 w
 r
 i
 t
 e
 o
 p
 e
 r
 a
 t
 i
 o
 n

 V
 (
 w
 m
 u
 t
 e
 x
 t)

 P
 (
 w
 m
 u
 t
 e
 x
 t)

 w
 r
 i

e
r

R
e
a
d
c
o
u
n
t
=
0

V
(
r
m
u
t
e
x
t)

R
e
a
d
c
o
u
n
t-
=
1

V
(
w
m
u
t
e
x
t)

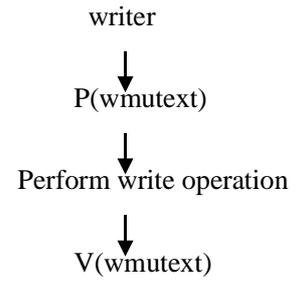
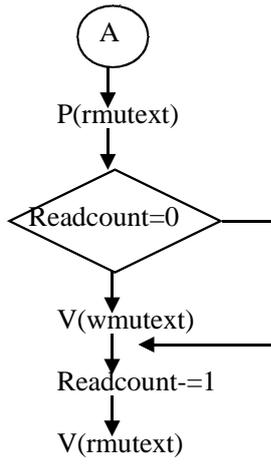
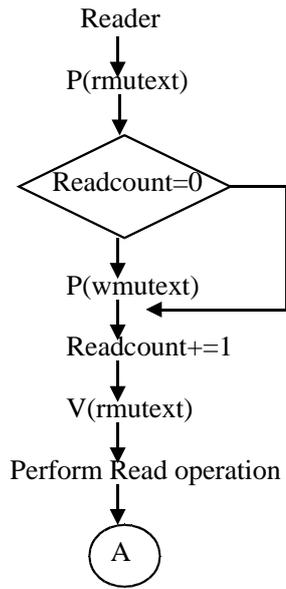
P
(
r
m
u
t
e
x
t)

A

A

P
e
r
f
o
r
m

R



a
d
o
p
e
r
a
t
i
o
n

V
(
r
m
u
t
e
x
t)

R
e
a
d
c
o
u
n
t
+
=1

P
(
w
m
u
t
e
x
t)

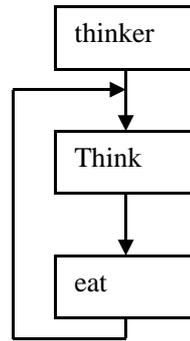
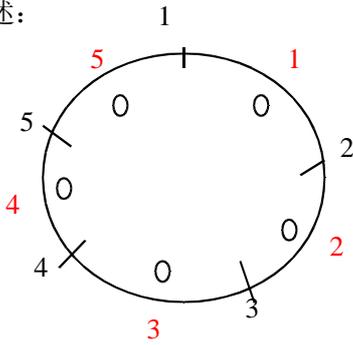
R
e
a
d
c
o
u
n
t
=0

P
(
r
m
u
t
e
x
t)

R
e
a
d

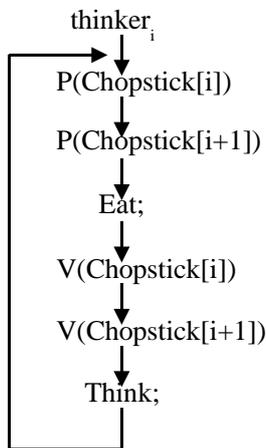
3. 哲学家进餐问题

问题描述：



解决方案一：

将 5 支筷子看作临界资源，设置互斥信号量 chopstick[i]。



思考：该方案可能导致死锁，why?

作业：解决这样的死锁问题的几种方法，见 p78。写出实现这几种方法的流程图或 pascal 描述。

r
c
o
n
s
u
m
e
r

P
(
f
u
l
l
)

P
(
u
t
e
x
)

取
数
据

V
(
m
u
t
e
x
)

V
(
e
m
p
t
y
)

P

o
c
c
e
d
u
r
e

3. 6 进程通信

通常把并发进程之间相互交换信息称为进程通信。（进程间的互斥与同步就是一种进程间的通信方式）

3.6.1 进程通信机制

1. 进程通信的类型
2. 高级通信的方式

低级通信：交换信息量小。常用变量、数组等方式。例进程同步和互斥 P、V 操作。
高级通信：交换信息量大。采用缓冲、管道、信箱、共享区等方式。

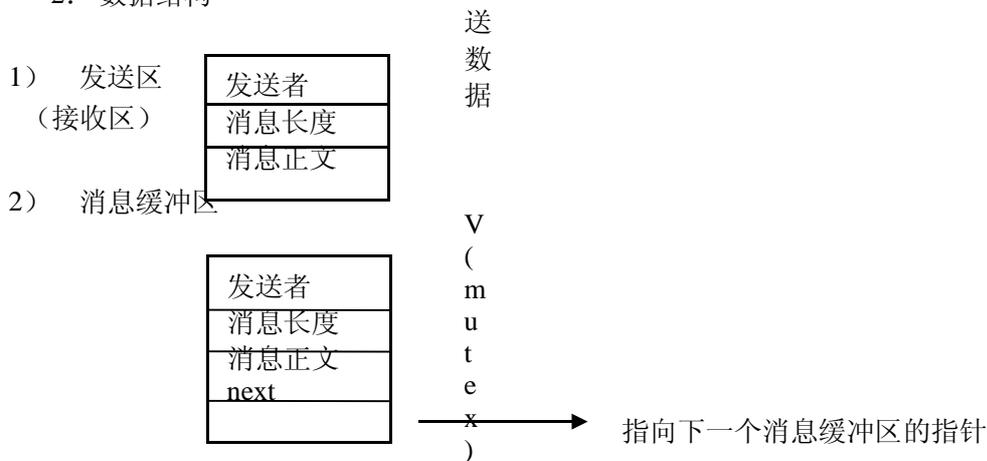
P
(
e
m
p
t
)
P
(
m
u
t
e
x
)

根据通信实施方式和数据存取方式可分为三类:

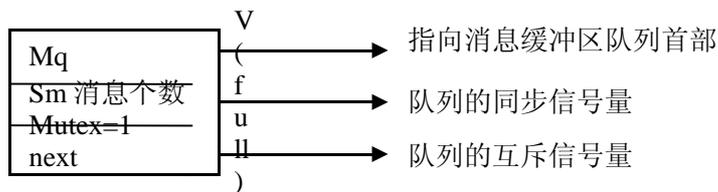
- 共享存储器系统: 相互通信的进程共享某些数据结构和共享存储区 (如寄存器、数组等)。
- 消息传递系统
- 管道通信系统 (共享文件) 基于文件系统, 用一个打开的共享文件连结两个相互通信的进程。
(以自然字符流方式读写)

3.6.2 消息通信

1. 基本思想: 把需要在进程间传递的一组信息看作一个消息。在系统中设置一个存放消息的存储区域, 称为消息缓冲区, 它可以同时存放一定数量的消息。
2. 数据结构



- 3) 系统中每个进程都设置一个消息队列, 进程的 PCB 中设置一个指向其缓冲队列的指针。对该队列的操作要遵循同步互斥原则。Pcb 中相应的变量如下:



3. 通信原语 (参见教材)

- 1) send 原语
- 2) receive 原语

3.6.3 信箱通信

1. 信箱: 是一种公共的存储区, 是通信的一种中间实体。
2. 组成:

- 信箱头: 描述信箱 (信箱名称、信箱大小、信箱方向以及拥有该信箱的进程名等)。
 - 信箱体: 存放消息, 由若干格子组成, 每个格子放一信件。 (格子数目及大小在创建信箱时确定)
- 单向信箱: 只发不答
- 双向信箱: 即发也收
- 公用信箱: 与多个进程通信 (按优先级处理)

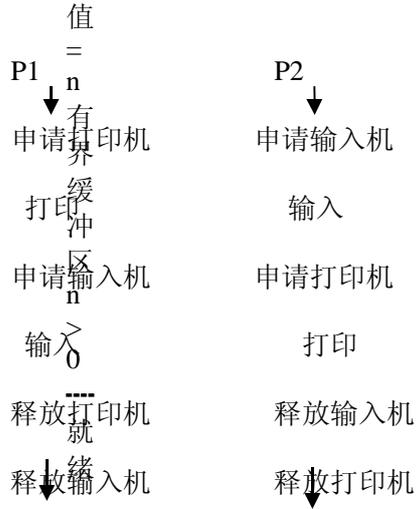
表示不满缓冲单元的总数
初值 =

3. 7 死锁

3.7.1 死锁的定义:

所谓死锁,是指多个进程因竞争资源而造成的一种僵局,若无外力作用,这些进程都无法向前推进,死锁是计算机系统和进程所处的一种状态。

例 1: 一台打印机、一台输入机



例 2: 解释为什么生产者消费者问题中两个 p 操作不能调换位置

3.7.2 死锁产生的必要条件

- 1、 互斥条件
- 2、 不剥夺条件
- 3、 请求与保持条件
- 4、 循环保持条件（进程资源图）：存在一种进程资源的循环等待。

3.7.3 死锁的预防

死锁预防：设置某些条件，破坏产生死锁的必要条件中的一个或多个即可。

- A. 破坏互斥条件：（应保证对临界资源的互斥访问，故不大可能实现）

- B. 破坏不剥夺条件：当进程资源请求不能立即满足时，必须释放所有已获得的资源。
- C. 破坏请求与保持条件：静态地一次性分配资源
- D. 破坏循环等待条件：有序资源分配法。
 - 1) 对资源进行编号；
 - 2) 进程申请资源时，必须以编号递增方向申请。

3.7.4 死锁的避免

死锁的避免：在资源的动态分配过程中，用某种方法防止系统进入不安全状态。

举例：详见书。结合课本中的事例，详细讲解银行家算法及安全性算法

系统 {

- 安全状态： 在某个时刻，系统能按某种顺序，如 $\langle P_1, P_2, \dots, P_n \rangle$ 来为每个进程分配需要的资源，直至最大需求，是每个进程都能顺利地完 成。则称此时的系统状态为安全状态。 $\langle P_1, P_2, \dots, P_n \rangle$ 为安全序 列。
- 不安全状态： 在某个时刻，系统不存在这样一个安全序列，则称此时的系统状 态为不安全状态。

3.7.5 死锁的检测

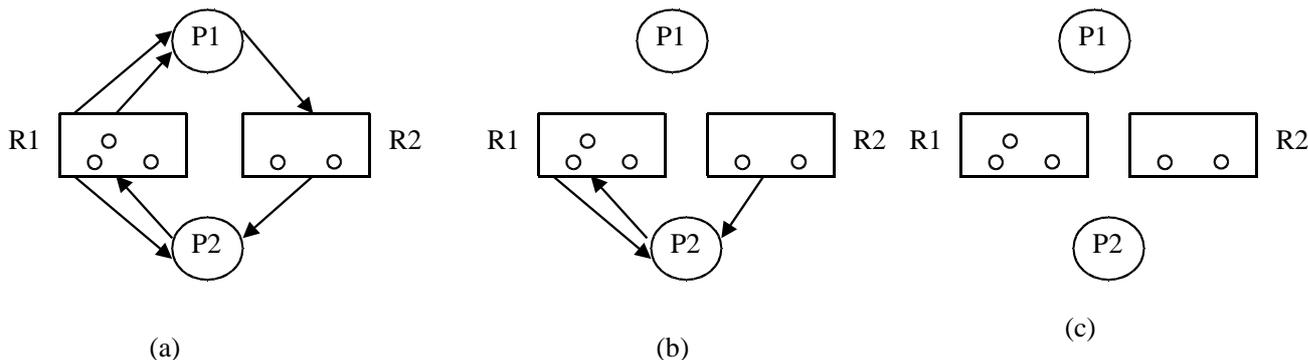
死锁定理： S 为死锁状态的充分条件是，当且仅当 S 状态的资源分配表是不可完全简化的。利用资源分配图，对其进行简化。

{

- 可完全简化 \Rightarrow 不死锁
- 不可完全简化 \Rightarrow 死锁（死锁定理）

简化方法： 寻找过程资源图中既不孤立又不阻塞的节点（进程节点）删去请求边和分配边。具体如下：

- 1) 在资源分配表中，找一个既不阻塞也非独立的进程点 P_i ，在顺利情况下， P_i 可获得所需资源而继续执行，直至运行完毕，再释放其占有的所有资源。（a）—（b）
- 2) P_1 释放资源后，使 P_2 获得资源而继续运行， P_2 完成后又释放出所有的全部资源。（b）—（c）
- 3) 在进行一系列的简化后，若能消去图中所有边，使所有进程都成为孤立点。则称该图可完全简化，否则称该图为不可完全简化。



3.7.6 死锁的解除

将系统从死锁状态解脱出来，有两种方法：

- 1) 资源剥夺法（抢占）：当发生死锁后，从其他进程剥夺足够数量的资源给死锁进程，以解除死锁状态。
- 2) 撤销进程法（终止）：采用强制手段，从系统中撤销一个或部分死锁进程，并剥夺这些进程的资源，供其他死锁进程使用。

第4章 Linux 进程管理

4. 1Linux 进程概述

4.1.1 Linux 进程的组成

1. Linux 进程组成：由正文段（text）、用户数据段（user segment）和系统数据段。

- 正文段：存放进程要执行的指令代码。Linux 中正文段具有只读属性。
- 用户数据段：进程运行过程中处理数据的集合，它们是进程直接进行操作的所有数据，包括进程运行处理的数据段和进程使用的堆栈。
- 系统数据段：存放反映一个进程的状态和运行环境的所有数据。这些数据只能由内核访问和使用。在系统数据段中包括进程控制块 PCB。

在 Linux 中，PCB 是一个名为 task_struct 的结构体，称为任务结构体。

2. 任务结构体的主要内容

任务结构体 `task_struct` 的大小约 1000B。它定义在源代码的 `include/linux/sched.h` 下。下面把它的成员的功能和作用归纳为 9 个方面予以说明：

- 1) 进程的状态和标志 `state` 和 `flags`。
- 2) 进程的标识。表示进程标识的成员项：`pid`, `uid`, `gid` 等。
- 3) 进程的族亲关系，`p_opptr`、`p_pptr`、`p_cptr`、`p_vsptr`、`p_osptr`
- 4) 进程间的链接信息。`Next_task`、`prev_task`、`next_run`、`prev_run`。
- 5) 进程的调度信息。`Counter`、`priority`、`rt_priotity`、`policy`。
- 6) 进程的时间信息。表示系统中各种时间成员项：`start_time`、`utime`、`stime`、`cutime`、`cstime`、`timeout`。表示定时器的成员项：`it_real_value`、`it_real_incr`、`it_virt_value`、`it_virt_incr`、`it_prof_value`、`it_prof_incr`、`real_timer`。
- 7) 进程的虚拟信息。`Mm`、`ldt`、`saved_kernel_stack`、`kernel_stack_page`。
- 8) 进程的文件信息 `Fs`、`file`。
- 9) 与进程通信有关的信息：`signal`、`blocked`、`sig`、`exit_signal`、`semundo`、`semsleeping`。

3. 任务结构体的管理

1) `task[]` 数组

为了管理系统中所有进程，系统在内核空间设置了一个指针数组 `task[]`，该数组的每一个元素指向一个任务结构体，所以 `task` 数组又称 `task` 向量。如下图 4.1 所示。

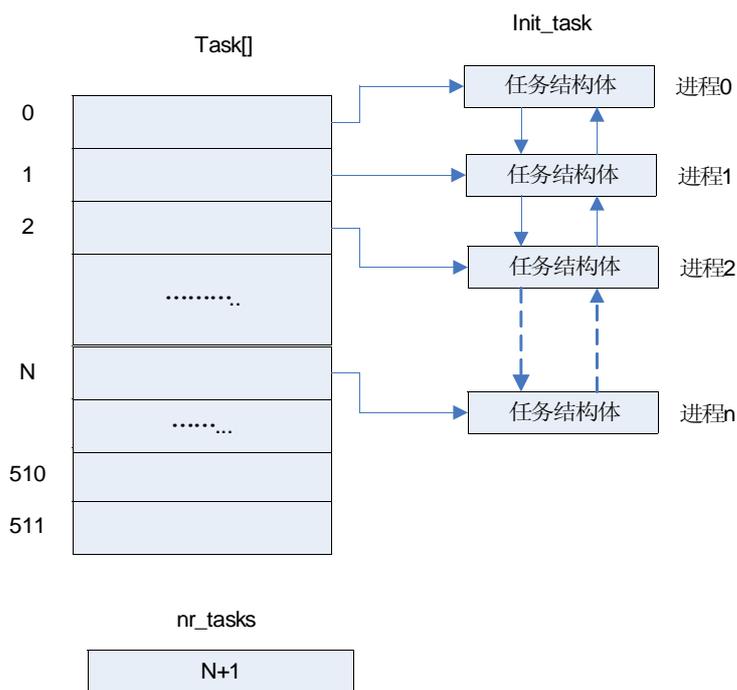


图4.1 linux任务结构体的管理

Task 数组的大小决定了系统中容纳进程最大数量。在 linux 内核源代码 kernel/sched.c 中 task 数组的定义如下：

```
Struct task_struct *task[NR_TASKS]={&init_task};
```

可以看到：

- task[]数组是一个指向 task_struct 结构的指针数组。
- 其中符号常量 NR_TASKS 决定了数组的大小，在 /include/linux/task.h 中，它的默认值被定义为 512：#define NR_TASKS 512
- task[]数组的第一个指针指向一个名字为 init_task 的结构体，它是系统初始化进程 init 的任务结构体。

2) nr_tasks

为了记录系统中实际存在的进程数，系统定义了一个全局变量 nr_tasks，其值随系统中存在的进程数目而变化。在 kernel/fork.c 中，它的定义及初始化如下：

```
Int nr_tasks =1;
```

3) 双向循环链表

Linux 中把所有进程的任务结构相互连成一个双向循环链表，其首结点就是 init 的任务结构体 init_task。这个双向链表是通过任务结构体中的两个成员项指针相互链接而成：

```
Struct task_struct *next_task; /*指向后一个任务结构体的指针*/
Struct task_struct *prev_task; /*指向前一个任务结构体的指针*/
```

宏定义：

SET_LINKS：插入一个任务结构体

REMOVE_LINKS：删除一个任务结构体

For_each_task：遍历所有任务结构体

4.1.2 Linux 进程在处理机上的执行状态

在 Linux 系统中，用户不能直接访问系统资源，如处理机、寄存器、存储器和各种外围设备。因此提供了两种不同指令：

- 一般指令：供用户和系统编程使用，不能直接访问系统资源；
 - 特权指令：供操作系统使用，可以直接访问和控制系统资源；
- 为了区分处理机在执行那种指令，通常将处理机的执行状态又分为两种：

- 管态（内核态、系统）
- 目态（用户态）

由目态转变为管态的情况可能有：

- 进程通过系统调用向系统提出服务请求
- 进程执行某些不正常的操作时，如除数为 0、超出权限的非法访问、用户堆栈溢出等。
- 进程使用设备进行 I/O 操作时，当操作完成或设备出现异常情况时。

4.1.3 进程空间和系统空间

进程的虚拟地址空间：Linux 操作系统运行在多道环境下，多个进程能够同时在系统中并发活动。为了防止进程间的干扰，系统为每个进程都分配了一个相对独立的虚拟地址空间，又称为进程的虚拟内存空间。

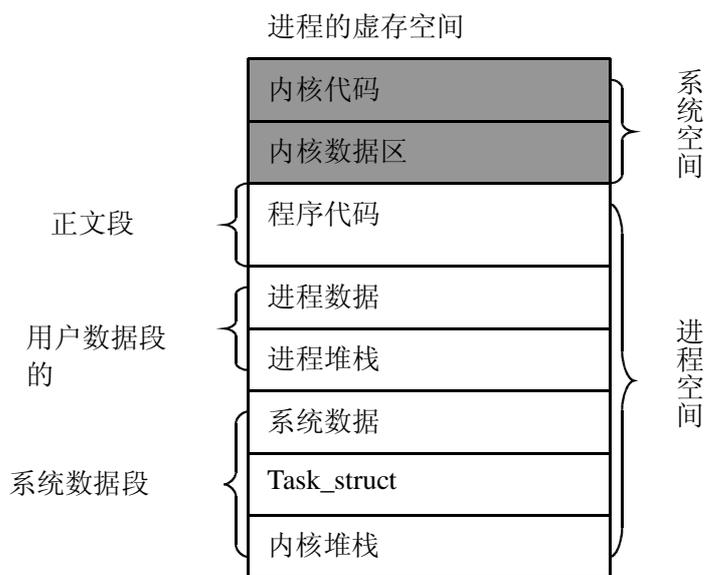
进程的虚拟地址空间包含进程本身的代码和数据等，同时还包括操作系统的代码和数据。故进程的虚拟地址空间分两部分：进程空间和系统空间。

进程空间（如图）：

注：

- 内核堆栈：进程在需要使用内核功能而通过系统调用运行内核代码时，需要使用堆栈保存数据。这个堆栈是由系统内和使用的，故称内核堆栈。
- 其中系统数据段，只能在内核态执行

系统空间：操作系统的内核映射到进程的虚拟地址空间。（可供多个进程共享，只能内核态执行）



4.1.4 进程上下文和系统上下文

进程上下文: 进程的运行环境动态变化，在 linux 中把**进程的动态变化的环境总和**称为进程上下文。

- 当前进程
- 进程切换
- 上下文切换
- 进程通过系统调用执行**内核**代码时，内核运行是为进程服务，所以此时内核运行在进程上下文中。

系统上下文: 内核除了为进程服务，也需要完成操作系统本身任务，如响应外部设备的中断、更新有关定时器、重新计算进程优先级等。故把**系统在完成自身任务是的运行环境称为系统上下文 (system context)**。

4. 2Linux 进程的状态和标识

4.2.1 Linux 进程的状态及转换

linux 中进程状态分为 5 种。

每个进程在系统中所处的状态记录在它的任务结构体的成员项 **state** 中。进程的状态用符号常量表示，它们定义在 `/include/linux/sched.h` 下：

```
#define TASK_RUNNING      0      可运行态（运行态、就绪态）
#define TASK_INTERRUPTIBLE 1      可中断的等待态
#define TASK_UNINTERRUPTIBLE 2    不可中断的等待态
```

```
#define TASK_ZOMBIE      3      僵死态
#define TASK_STOPPED    4      暂停态
```

1. 运行态 (running) —— 运行, 该进程称为当前进程 (current process)

实际上 linux 并没有该状态, 而是将其归结在可运行态。系统中设置全局指针变量 `current`, 指向当前进程。

2. 可运行态 (Running) —— 就绪

Linux 中把所有处于运行、就绪状态的进程链接成一个双向链表, 称为可运行队列 (run_queue)。使用任务结构体中的两个指针:

```
Struct task_struct *next_run; /*指向后一个任务结构体的指针*/
```

```
Struct task_struct *prev_run; /*指向前一个任务结构体的指针*/
```

该链表的首结点为 `init_task`。系统设置全局变量 `nr_running` 记录处于运行、就绪态的进程数。

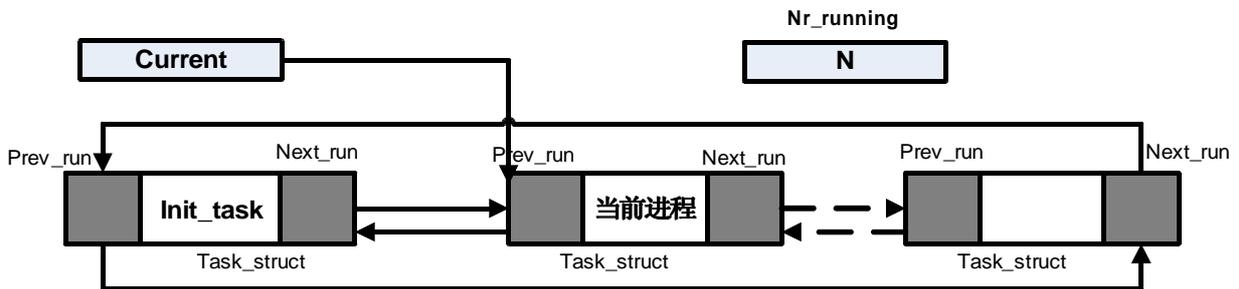


图4.4 linux的可运行队列

3. 等待态 (wait) —— 阻塞

在 linux 中将该状态进一步划分为: 可中断的等待态 (interruptible) 和不可中断的等待状态 (uninterruptible)。

- 可中断的等待态的进程可以由信号 (signal) 来解除其等待态, 收到信号后进程进入可运行态。
- 不可中断的等待状态的进程, 一般都是直接或间接在等待硬件条件, 只能用特定的方式来解除其等待状态, 如是用 `wakeup()`。

处于等待态的进程根据其等待的事件排在不同的等待队列中。Linux 中等待队列是由一个 `wait_queue` 结构体组成的单向循环链表。该结构体定义在 `include/linux/wait.h` 中, 如下所示:

```
Struct wait_queue {
    Struct task_struct *task; /*指向一个等待态的进程的任务结构体*/
    Struct wait_queue *next; /*指向下一个 wait_queue 结构体*/
}
```

注:

- 与可运行队列不同, 等待队列不是直接由进程的任务结构体组成队列, 而是由于任务结构体对应的 `wait_queue` 构成。
- 每个等待队列都有一个指向该队列的队首指针, 它一般是个全局指针变量。如图 4.5。

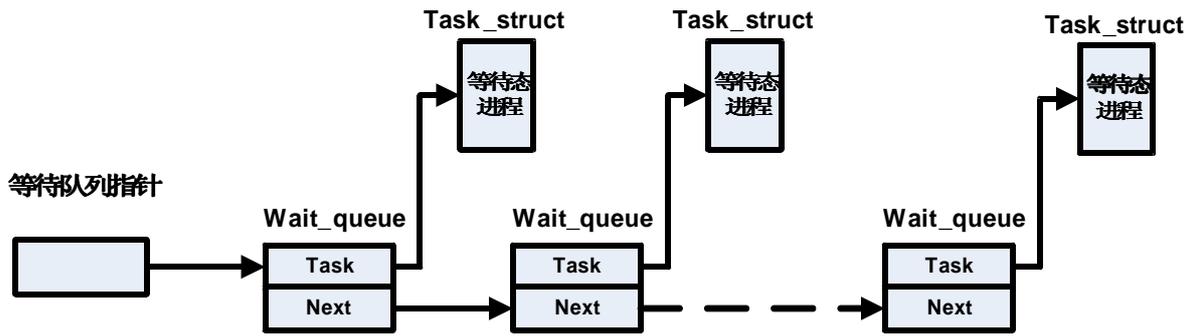


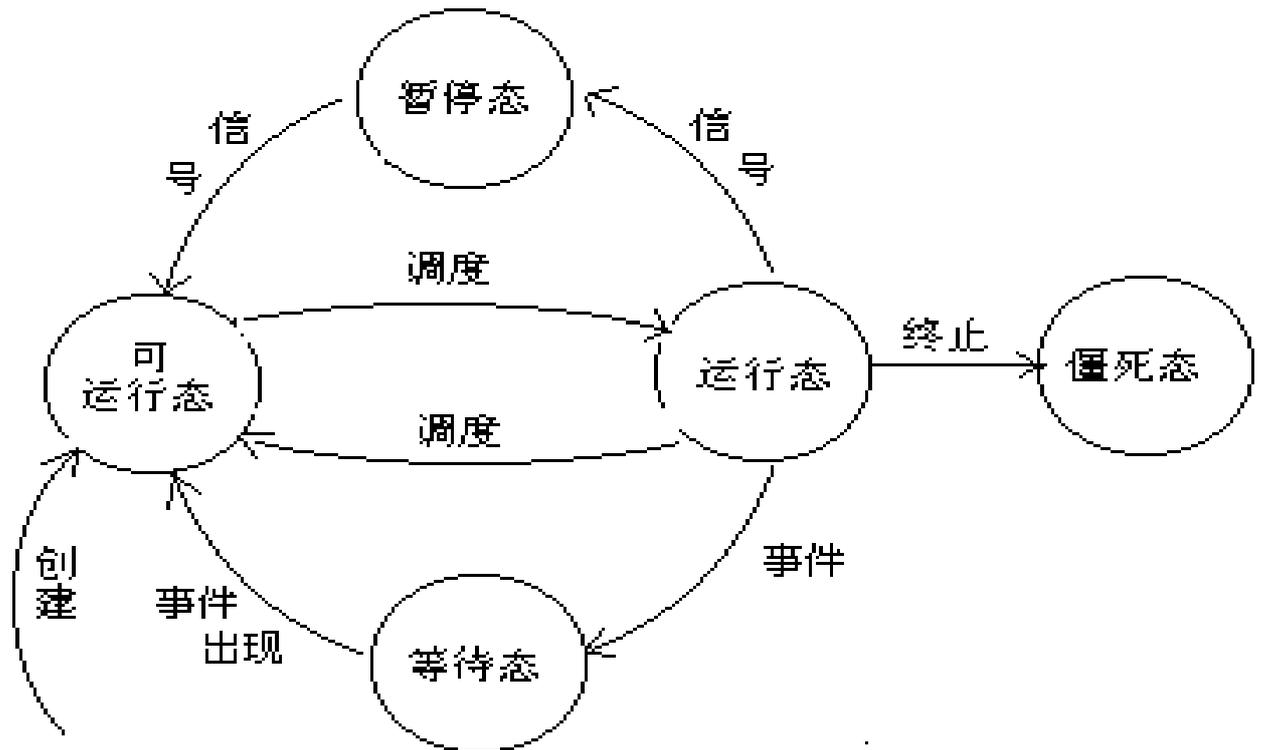
图4.5 ; linux等待队列

4. 暂停态 (stopped)

暂停态：进程由于需要接受某种特殊处理而暂时停止运行所处的状态。通常，进程在接受到外部进程的某个信号 (SIGSTOP、SIGSTP、SIGTTOU) 而进入暂停态。通常正在接受调试的进程就处于暂停态。

5. 僵死态 (zombie)

僵死态：进程的运行已经结束，但是由于某种原因它的进程结构体仍在系统中。



Linux进程状态的转换

图 4.6 Linux 进程转换图

4.2.2 Linux 进程的标识

Linux 中，进程的标识是系统识别进程的依据，也是进程访问设备和文件时的凭证。Linux 为每个进程设置多种标识，不同的标识的用途不同。Task_struct 中记录着进程的各种标识：

Int pid	进程标识号
Unsigned short uid , gid;	用户标识号，组标识号
Unsigned short euid , egid;	用户有效标识号，组有效标识号
Unsigned short suid , sgid;	用户备份标识号，组备份标识号
Unsigned short fsuid , fsgid;	用户文件标识号，组文件标识号

注：

1. Pid 32 位，但是为了与 UNIX 兼容（16 位），故 linux 也使用 16 位，最大值 32767。Pid 按照进程创建的先后顺序依次赋予进程，即前面进程 PID 值加 1。当达到最大值时，重复使用已经撤销进程的 PID。
2. 设置 Uid, gid 目的：文件保护。Linux 把文件的所有用户分为 3 类：所有者、同组用户、其他用户。
3. 一般情况下 euid=uid; egid= gid; fsuid= uid; fsgid= gid;
4. Euid 和 egid: 在进程企图访问特权数据或代码时，系统内核需要检查进程的有效标识 Euid 和 egid。需要其他进程服务时，这两个指将变为服务进程 uid 和 gid。参见（徐德民 p82）
5. fsuid , fsgid: 在进程企图访问文件时，系统内核需要检查进程的文件标识 fsuid , fsgid。需要其他进程服务时，这两个指将变为服务进程 uid 和 gid。参见（徐德民 p82）
6. 没有将 euid 和 fsuid, egid 和 fsgid 统一，原因：防止具有访问特权后，用户对系统造成破坏。
7. Suid 和 sgid 是 POSIX 标准要求的标识。当用户执行系统调用而使其用户标识 uid 或组标识 guid 改变时，suid 和 sgid 保存原来的值，以便恢复。

4.2.3 进程标识哈希表

Linux 的进程表示哈希表提供了按照哈希算法从进程 PID 快速查找对应任务结构体的方法，实现哈希算法的哈希函数定义为带参数的宏 pid_hashfn(x)，如下所示：

```
#define pid_hashfn(x) (((x)>>8)^(x))&(PIDHASH_SZ-1)???
```

其中：参数 x 就是进程的标识 PID，计算结果的哈希值用于检索对应的任务结构体。例如 PID 为 228 的哈希值是 100，PID 为 27536 的哈希值是 123，PID 为 27535 的哈希值是 100。

为了解决哈希值冲突的问题，Linux 把具有相同哈希值的 PID 对应的进程组成一个个双向循环链表。在 task_struct 中的两个成员项：

```
Struct task_struct * pidhash_next; /*指向后一个任务结构体的指针*/  
Struct task_struct * pidhash_prev; /*指向前一个任务结构体的指针*/
```

1. 进程标识哈希表

Linux 使用一个称为 pidhash[] 的指针数组管理这些链表，称为进程标识哈希表。该表中记录各个链表首结点地址，数组元素的下标与链表的哈希值相同。在 include/linux/sched.h 中 pidhash[] 数组定义如下：

```
Struct task_struct * pidhash[PIDHASH_SZ];
```

从定义中可以看出，

- pidhash 数组由 PIDHASH_SZ 个元素组成，每个元素是指向一个进程任务结构体的指针。
- 数组元素个数 PIDHASH_SZ 是系统中最多可容纳的进程数 NR_TASKS 除以 4，定义如下：

```
#define PIDHASH_SZ (NR_TASKS>>2)? ? ?
```

图 4.7Linux 进程标识哈希表

2 进程标识哈希表操作函数：

Hash_pid(): 进程创建时，将其任务结构体插入哈希链表

Unhash_pid(): 进程撤销时，将其任务结构体从哈希链表中删除。

Find_task_by_pid(): 根据 PID 相应进程的任务结构体。

4. 3Linux 的进程调度

4.3.1 Linux 进程调度策略

Linux 是一个同时具有分时和实时系统特征的操作系统。Linux 在进程调度中采用的是可抢占的调度方式。

Linux 中的进程分为普通进程和实时进程。实时进程的优先级高于普通进程。对实时进程和普通进程采用不同的调度策略。

Linux 为每个进程都规定了一种调度策略，并记录在其任务结构体 policy 成员项中。Linux 调度策略有 3 种，它们以符合常量的形式定义在 /include/linux/sched.h 中，其定义及意义如下所示：

```
#define SCHED_OTHER 0 普通进程的时间片轮转算法（根据优先权选择下一个进程）
```

```
#define SCHED_FIFO 1 实时进程的先进先出算法（适用于响应时间要求比较严格的短小进程）
```

```
#define SCHED_RR 2 实时进程的时间片轮转算法（适用于响应时间要求比较严格的较大进程）
```

因此在 linux 的可运行队列中，从调度策略来分 SCHED_FIFO 的实时进程具有最高优先级，其次是 SCHED_RR 的实时进程，而 SCHED_OTHER 的普通进程优先级最低。

4.3.2 Linux 进程调度依据

Linux 的进程调度采用了优先级和权值的方法。Linux 用以下四个数据作为调度依据，它们记录在进程的任务结构体中：

- Policy 是进程的调度策略
- Priority 是普通进程的优先级。它是 [0~70] 之间的数，数值越大优先级越高。Priority 除表示进程的优先级，还表示分配给进程使用 CPU 的时间片。
- Rt_Priority 是实时进程的优先级。策略为 SCHED_FIFO 的实时进程的 rt_Priority 大于 SCHED_RR 实时进程
- Counter 中存放的是进程还需要使用 CPU 运行时间的计数值，它是动态变化的，它的初始值就是 Priority。

4.3.3 Linux 进程调度的加权处理

普通进程和实时进程在同一个可运行队列中，为了保证实时进程优于普通进程执行，linux 采用了加权处理的方法：在进程调度过程中，每次选取下一个运行进程时，调度程序首先给可运行队列的每个进程赋予一个权值（weight）。普通进程的权值就是它的 counter 值，而实时进程的权值是它的 rt_Priority 值加 1000。

Linux 使用内核函数 goodness（）对进程进行加权处理，它的源程序在/kernel/sched.c 中，下面给出了去掉其中多处理机（SMP）部分后简化的程序代码。

```
Static inline goodness(struct task_struct *p, struct task_struct *prev, int this_cpu)
{
    Int weight;
    If(p->policy!=SCHED_OTHER)      /*若当前进程是实时进程*/
        Return 1000+p->rt_Priority; /*返回权值为 rt_Priority +1000*/
    Weight=p->counter;              /*若当前进程是普通进程*/
    .....
    .....
    Return weight;                  /*返回权值为 counter */
}
```

4.3.4 Linux 进程调度方法

实时进程的优先级大于普通进程的优先级，故只有当可运行队列的所有实时进程都运行完成后，普通进程才能得到运行。

linux 普通进程的优先级由 Priority 和 counter 共同决定。在进程运行过程中 Priority 保持不变，体现了进程的静态优先级概念；而 counter 不断减少，表示了进程的动态优先级。采用**动态优先级**的方法，使得一个进程占用 CPU 的时间越长，counter 的值越小。这样使得每个进程都可以公平地分配到 CPU。

4.3.5 Linux 进程调度时机

Linux 进程调度是由 Schedule（）完成的。该函数定义在/kernel/sched.c 中。执行该函数 的情况可以分为两种：

- 在某些系统调用函数中直接调用 Schedule（）。
- 在系统运行过程中，通过检查调度标志而执行该函数。进程调度标志是一个名为 need_resched 的全局变量，当它的值为 1 时，表明需要执行调度函数。

下面介绍几种需要执行进程调度的时机：

1. 进程状态发生变化时

Linux 进程状态不断发生变化，在下列状态转换是需要执行进程调度：

1) 当前进程进入等待状态

例如，运行太的进程可以通过执行系统调用 `sleep_on()` 主动放弃 CPU 而进入等待状态。
`Sleep_on()`的部分源代码如下：

```
Current->state=state;          /* 把当前进程状态设置为等待状态*/
Save_flags(flags);
_add_wait_queue(p,&wait);      /*把当前进程加入等待队列*/
Sti ();
Schedule();                    /*执行进程调度*/
Cli();
```

2) 运行态下的进程运行结束后

一般通过调用内核函数 `do_exit()` 终止运行进程并转入僵死状态。该函数部分源码：

```
.....
Current->state = TASK_ZOMBIE; /*把当前进程设置为僵死状态*/
.....
Schedule (); /*执行调度程序*/
.....
```

3) 使用 `wake_up_process()` 将处于等待状态的进程唤醒，然后将它置于可运行状态。该函数部分源码：

```
Save_flags(flafs);
Cli();
p->state = TASK_RUNNING; /*把进程置为可运行态*/
if (! p->next_run)
    add_to_runqueue(p); /*加入到可运行队列*/
restore_flags(flags);
if(p->counter>current->counter+3)
    need_resched = 1; /*调度标志置位，执行进程调度*/
```

4) 当一个进程的程序接受调试时。

调试进程向被调试进程发送 `SIGSTOP` 信号，被调试进程处理该信号时调用内核函数 `do_signal()`。
部分源码：

```
.....
Current->state = TASK_STOPPED /*把当前进程置为暂停态*/
Notify_parent(current);
Schedule (); /*执行进程调度*/
.....
```

5) 当被调试的进程接收到调试进程发送的 `SIGCONT` 信号时，执行 `send_sig()`，其中使用 `wake_up_process()` 解除被调试进程的暂停态而重新进入可运行态。

```
If (sig==SOGKILL||sig==SIGCONT) )
{
    If(p->state==TASK_STOPPED) /*若进程为暂停态*/
        wake_up_process(p);
```

2. 当前进程时间片用完时

在进程时间片运行完时，需要将 CPU 重新分配给下一个被选中的进程，这个过程是在时钟中断中实现。在时钟中断处理程序中调用了内核函数 `update_process_times()`，它用于更新进程的各个时间信息，其中包括下面语句：

```

p->counter -= ticks
if(p->counter<0)
{
    P->counter=0;
    Need_resched=1;
}

```

3. 进程从系统调用返回用户态时

当进程从系统调用返回用户态时，需要执行内核的汇编例程 `ret_from_sys_call`，其中包括对 `need_resched` 标志进行检测的指令。

```
Cmpl $0, need_resched
```

```
Jne reschedule
```

当 `need_resched=1` 时，就转移到 `reschedule`。

```
Reschedule:
```

```
Call SYMBOL_NAME(schedule)
```

4. 中断处理后，进程返回用户态时

同 3，当中断处理结束后，也需要执行内核的汇编例程 `ret_from_sys_call`

4. 4 Linux 进程的创建和撤销

4.4.1 Linux 进程的族亲关系

1. 进程树

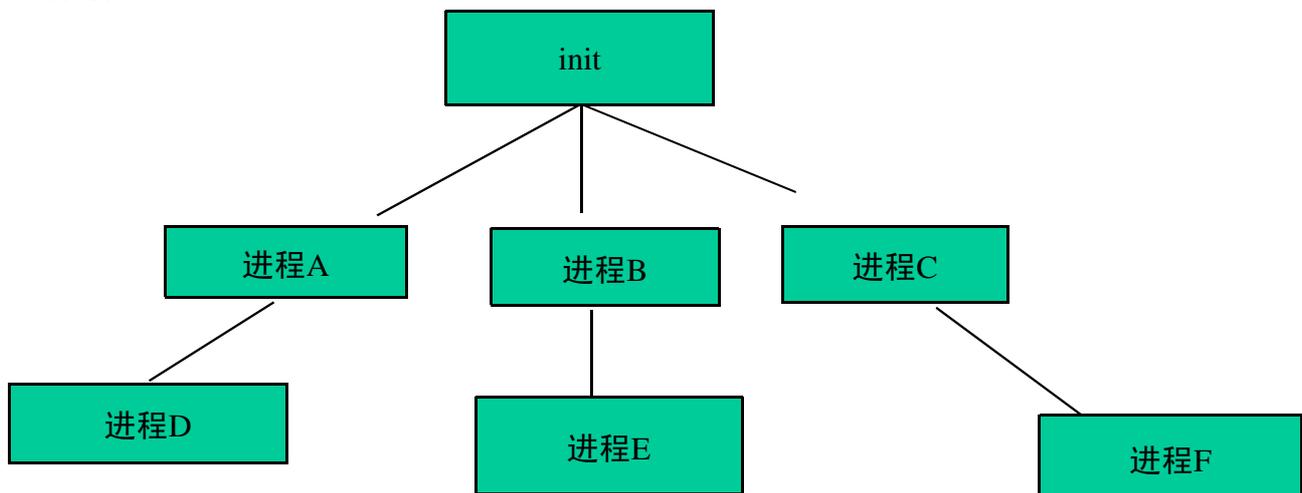


图 4.8 linux 系统的进程树

在系统加电启动后,系统只有一个进程，它就是由系统创建的初始进程，又称 `init` 进程。`init` 进程的任务结构体名字为 `init_task`。`init` 进程是系统中所有进程的祖先进程，进程标识号 `PID` 为 1。

- 进程之间的父子关系
- 进程之间的兄弟关系：按照创建时间确定，先者为兄，后者为弟；

2. Linux 进程的族亲关系

在每个进程的任务结构体中设置了 5 个成员项指针：

```
Struct task_struct *p_opptr /*指向祖先进程任务结构体指针*/  
Struct task_struct *p_pptr /*指向父进程任务结构体指针*/  
Struct task_struct *p_cprr /*指向子进程任务结构体指针*/  
Struct task_struct *p_ysptr /*指向弟进程任务结构体指针*/  
Struct task_struct *p_osptr /*指向兄进程任务结构体指针*/
```

图 4.9 Linux 进程的族亲关系

说明：

在任务结构体中只有一个指向子进程指针，它指向该进程子进程中最年轻的一个，其他子进程通过兄弟关系指针与父进程的关系。

4.4.2 Linux 进程的创建

Linux 中，除 init 进程是启动时由系统创建的，其他进程都是由当前进程使用系统调用 **fork ()** 创建的。

- 子进程被创建后，通常要继承（共享）父进程所有的资源。包括虚拟存储空间的内容、打开的文件、专用的信号处理程序等。
- 写时拷贝（copy on write）技术：子进程和父进程共享同一个虚拟空间，只是这两个进程中某个进程需要虚拟内存写入时，这时才建立属于该进程的那部分虚拟空间，并把原虚拟空间的那部分内容拷贝到新建的虚拟内存区域中，然后在新建的属于自己的虚拟空间区域中写入信息。这样既可以使子进程继承父进程的资源，又可以在需要时建立自己的存储空间，避免子进程存储空间的无谓浪费，有效地节省了系统时间。
- 父进程和子进程在“分裂”后，运行时都在继续执行 **fork ()** 程序的代码。但是为了区分两个进程：父进程执行完 **fork ()** 时返回值是子进程的 PID 值；而子进程执行 **fork ()** 的返回值是 0。

例：

```
#include <sys/types.h>  
#include <unistd.h>  
Main()  
{  
    Pid_t val;  
    Printf("PID before fork():%d\n",(int) getpid());  
    Val=fork();  
    If(val!=0)  
        Printf("parent process PID:%d\n",(int) getpid());  
    Else  
        Printf("child process PID:%d\n",(int) getpid());  
}
```

程序执行结果：

PID before fork(): “此时父进程的 PID”

parent process PID: “此时父进程的 PID”

child process PID: “此时子进程的 PID”

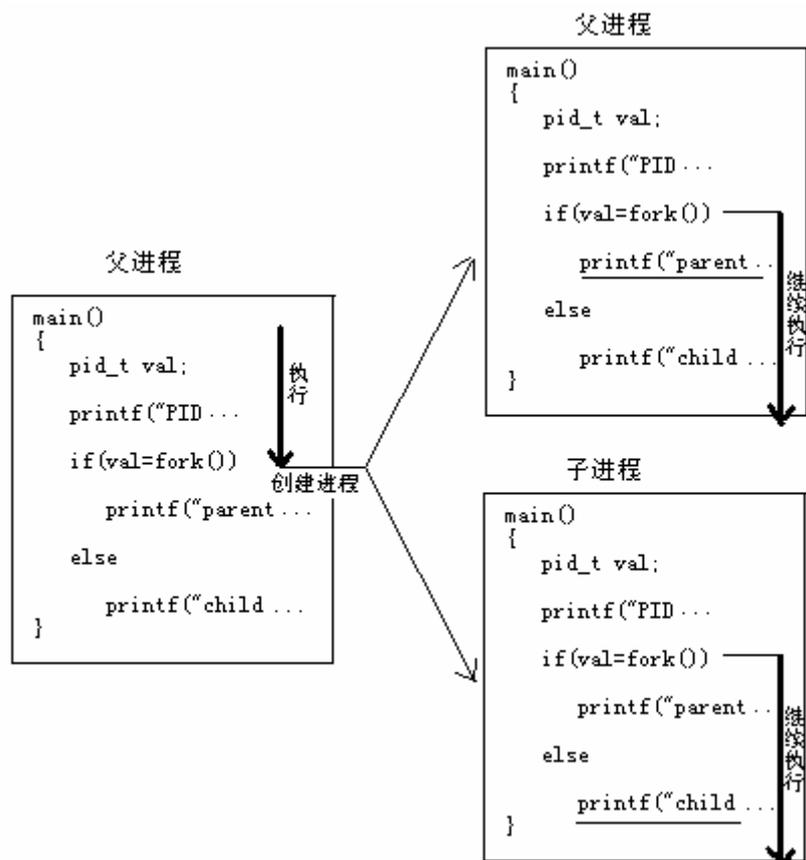


图 4.10 fork () 进程的“分裂”过程

4.4.3 Linux 进程创建的过程

Linux 中进程都是由当前进程使用系统调用 **fork ()** 创建的，而实际上是 **fork ()** 中进一步调用内核函数 **do_fork ()** 来完成的。**Fork ()** 的源代码定义在 **/kernel/fork.c** 中。下面给出 **do_fork ()** 内核函数的主要功能：

1) 在内存空间为新进程（子进程）分配任务结构体使用的空间，然后把当前进程（父进程）的任务结构体的所有内容拷贝到子进程的任务结构体中。

2) 为新进程在其虚拟内存建立内核堆栈。虽然子进程共享父进程的虚拟空间，但是两个进程在进入核心态后必须有自己独立的内核堆栈。

3) 对子进程任务结构体中的部分内容进行初始化设置，例如进程的链接关系（族亲关系）等，主要是与父进程不同的那些数据。

4) 把父进程的资源管理信息拷贝给子进程，如虚拟内存信息、文件信息、信号信息等，前面说过，这里的拷贝是建立两个进程对这些资源的共享关系。

5) 把子进程加入可运行队列中，由调度进程在适当的时机调度运行，也就是在这个时候，当前进程分裂为两个进程。

7) 无论哪个进程使用 CPU 运行, 都继续执行 do_fork()函数的代码, 执行结束后返回它们各自的返回值。

从以上过程可以看出, do_fork()函数的功能首先是建立子进程的任务结构体和对其进行初始化, 然后继承父进程资源, 最后设置进程的时间片并把它加入可运行队列。

4.4.4 Linux 进程的执行

1. 系统调用 exec ()

在系统中创建一个进程的是需要该进程完成一定的任务, 通过执行它实现所需的功能。在 Linux 系统中, 使进程执行的唯一方法是使用系统调用 **exec ()**。

Exec () 由多种使用形式, 它们只是在参数上不同, 而功能相同。下面介绍一种最常用的形式:

```
int execl (path, arg0, ..., argn, (char*) 0)
char *path, *arg0, ..., *argn;
```

其中:

- Path: 指出要执行的程序文件的完整路径名。
- arg0: 指出要执行的程序文件的文件名。
- Arg1, ..., argn: 程序的参数, 这些参数的个数可以不同。
- 该系统调用将引起另一个程序的执行, 故它成功调用后不需返回, 只有在调用失败时, 返回值 -1;

例: 在程序运行中要执行 linux 命令 “ls -l”, 程序编制如下:

```
#include <stdio.h>
Main()
{
    Execl(“/bin/ls”, “ls”, “-l”, 0);          /*调用成功, 则执行 ls 命令*/
    Printf(“can only get here on error/n”); /*出错时, 返回该程序, 显示错误信息*/
}
```

2. 子进程执行新的程序

一个进程使用 fork () 建立子进程后, 让子进程执行另外一个程序的方法也是通过使用 exec () 系统调用。程序的一般形式为:

```
Main ()
{
    Pid_t val;
    Val =fork();
    If(val!=0)
    {
        ..... /*执行父进程的语句*/
    }
    Else
    {
        exec (.....) ; /*执行程序*/
    }
    .....
}
```

```
Exit (1) ;  
}
```

Linux 采用“写时拷贝”技术，所以子进程在创建时是共享父进程的正文段和数据段，但是在执行 `exec()` 时，子进程将建立自己的虚拟空间，并把参数 `arg0` 制定的可执行程序映像装入子进程的虚拟空间，这就形成了子进程自己的正文段和数据段。

4.4.5 Linux 进程的终止和撤销

进程终止的两种情况：

- 进程完成自身的任务而自动终止：方法 1，使用系统调用 `exit()` 显示终止进程；方法 2，执行程序中的 `main()` 的结尾而隐式地终止进程。
- 进程被内核强制终止：如进程运行出现致命错误，或者收到不能处理的信号时。

1. 进程的终止 `exit()`

Linux 中终止进程是通过调用内核函数 `do_exit()` 实现的，该函数定义在 `kernel/exit.c` 中。下面结合 `do_exit()` 函数的部分源代码对进程终止和撤销的过程说明如下：

1) 函数首先设定当前进程的标志，即任务结构体的 `flags` 成员项设定为 `PF_EXITING`，表示该进程将要终止

```
Current->flags = PF_EXITING;
```

2) 释放系统中该进程在各个管理队列中的任务结构体。

```
Del_timer(&current->real_timer);/*从动态计时器系列中释放该进程的任务结构体*/
```

```
Sem_exit(); /*从 IPC 进程间通信的信号量机制中释放该进程的任务结构体*/
```

3) 释放进程使用的各种资源。

```
Kerneld_exit(); /*释放内核*/
```

```
_exit_mm(current); /*释放虚拟空间*/
```

```
_exit_files(current); /*释放打开的文件*/
```

```
_exit_fs(current); /*释放 fs 结构体*/
```

```
_exit_sighand(current); /*释放信号*/
```

```
Exit_thread(); /*释放线程*/
```

4) 把进程的状态设置为僵死状态。

```
Current->state = TASK_ZOMBIE;
```

5) 把退出码置入任务结构体。

```
Current->exit_code =code;/*正常退出时，退出码为 0；异常退出时，为非 0 值，通常是错误编号  
*/
```

6) 变更进程族亲关系。

```
Exit_notify();/*把要撤销的进程的子进程的父进程赋予 init 进程*/
```

7) 执行进程调度，选择下一个使用 CPU 的进程。

```
Schedule ();
```

2. 进程的撤销 `release()`

一个进程终止后不能自己撤销自己，而只能由他的父进程或祖先进程来撤销它：

父进程在建立子进程后，通常要使用系统调用 `wait()` 等待子进程的终止。在一个进程终止时，把退出码置入任务结构体后，内核就把该进程终止的信号发给它的父进程。父进程就收到信号后，结束等待状态，继续执行 `wait()` 程序，检查到该子进程的状态为僵死态，则调用 `release()` 函数，

撤销该子进程，将其任务结构体释放，则该子进程从此彻底消亡。

如果父进程在子进程之前终止，则为了不使其子进程成为孤立进程，则将其的子进程赋予 `init` 进程，使 `init` 进程成为这些进程的父进程，并由其负责撤销这些子进程。

4. 4 Linux 信号

4.5.1 信号的作用和种类

1. 信号机制

采用信号机制在进程间通信时传送的只是一个称为信号的数值，它不能传送更多的其它信息。

- 信号的主要作用是把系统中发生的某些事件通知给进程。
- 信号的主要特征是它的异步性：即什么时候出现信号是不可预知的。
- 信号的数量与运行机器平台有关，它与硬件机器的字长相对应。如 `80x86` 的字长 32 位，则信号有 32 种。（本节以 `80x86` 平台为例介绍 `linux` 信号机制）
- 在 `include/asm-i386/signal.h` 中定义的系统变量 `NSIG` 记录着信号的数量。
- 从 `linux2.2` 开始在一般信号的基础上又增设了实时信号，实时信号的数量与一般信号相同。
- 系统中每个信号都是一个整数，称为信号值。为了清楚地表示信号的意义，`linux` 通过宏定义给每个信号都定义了一个符号常量，称为信号名。`Linux` 的信号名的组成以 `SIG` 打头，后面跟着表示信号意义的英文缩写。对于 `80x86`，信号的宏定义在 `include/asm-i386/signal.h` 中。

2. 信号种类

表 4.1 `linux` 的信号种类及其意义

信号值	信号名	信号意义	缺省处理
1	<code>SIGHUP</code>	进程的控制终端或控制进程已结束	终止进程
2	<code>SIGINT</code>	用户键入 <code>ctrl-c</code>	终止进程
3	<code>SIGQUIT</code>	从键盘来的终止信号 (<code>quit</code>)	终止进程、 <code>core</code> 转储
4	<code>SIGILL</code>	进程执行了非法指令或企图执行数据段	终止进程、 <code>core</code> 转储
5	<code>SIGTRAP</code>	跟踪中断、执行 <code>trap</code> 指令	终止进程、 <code>core</code> 转储
6	<code>SIGABRT</code>	进程发现错误并调试 <code>abort</code>	终止进程、 <code>core</code> 转储
7	<code>SIGBUS</code>	进程访问非法地址、地址对齐出错	终止进程、 <code>core</code> 转储
8	<code>SIGFPE</code>	进程浮点运算错误、溢出、除数为 0 等	终止进程、 <code>core</code> 转储
9	<code>SIGKILL</code>	强制终止进程（本信号不能屏蔽）	终止进程（不能忽视）
10	<code>SIGUSR1</code>	保留给用户自行定义信号	终止进程
11	<code>SIGSEGV</code>	进程访问内存越界，或无访问权限	终止进程、 <code>core</code> 转储
12	<code>SIGUSR2</code>	保留给用户自行定义信号	终止进程
13	<code>SIGPIPE</code>	进程向无读者的管道执行写操作	终止进程
14	<code>SIGALRM</code>	时钟定时信号，由系统调用 <code>alarm</code> 发出	终止进程
15	<code>SIGTERM</code>	结束信号，由 <code>kill</code> 命令产生	终止进程
16	<code>SIGSTKFLT</code>	进程发现堆栈溢出错误	终止进程、 <code>core</code> 转储
17	<code>SIGCHLD</code>	子进程结束或终止	忽视
18	<code>SIGCONT</code>	让暂停的进程继续执行	终止进程
19	<code>SIGSTOP</code>	暂停进程的执行（不能屏蔽）	暂停进程（不能忽视）
20	<code>SIGTSTP</code>	用户键入暂停（通常是 <code>ctrl-z</code> ）	暂停进程
21	<code>SIGTTIN</code>	后台作业要从用户终端 (<code>stdin</code>) 读数据	暂停进程

22	SIGTTOU	后台作业写用户终端 (stdout)	暂停进程
23	SIGURG	套接字 (socket) 有“紧急”数据到达	忽视
24	SIGXCPU	进程使用 CPU 超时	终止进程、core 转储
25	SIGXFSZ	进程处理文件超长	终止进程、core 转储
26	SIGVTALRM	虚拟时钟信号 (计算进程占用 CPU 时间)	终止进程
27	SIGPROF	类似 SIGALRM/ SIGVTALRM (计算进程占用 CPU 时间以及系统调用的时间)	终止进程
28	SIGWINCH	终端窗口大小已改变	忽视
29	SIGIO	I/O 准备就绪, 可以进行输入/输出操作	忽视
30	SIGPWR	系统电源失效	
31	SIGUNUSED	未使用	

0 号值: 作为特殊情况来处理

综合上表, 可以将信号产生的情况归纳为 3 种主要情况:

1. 进程在执行过程中发生了某种错误, 此时系统的相应错误标志被置位, 系统内核识别到错误标志后, 就向有关进程发送相应信号, 通知进程发生了运行错误。
2. 系统或用户发出的控制进程终止或暂停的信号。
3. 内核需要控制进程的运行而产生的信号。

4.5.2 信号的处理

1. 在进程的任务结构体 `task_struct` 中有两个成员项用于处理接收的信号:

Unsigned long signal;

Unsigned long blocked;

它们都是位域 (Bitmap) 形式的 32 位 unsigned long 型变量, 每一位 (bit) 对应一种信号。变量的第 0 位对应信号值为 1 的 SIGHUP, 第 1 位对应信号值为 2 的 SIGINT, 依此类推。

- 1) Signal: 存放进程收到且尚未处理的信号。

- 进程可以同时接收多个信号
- 每种信号在 signal 中只有一位, 故不能识别接收了一个还是多个同一个信号
- 信号没有优先级, 可以以任意顺序处理接受到的信号

- 2) Blocked: 通过将 blocked 中的某一位设置为 1, 来屏蔽某种信号的处理。但是有两个不能屏蔽的信号 (SIGKILL 和 SIGSTOP) 是不能被屏蔽的, blocked 中它们对应的位始终为 0;

2. 进程接收到信号后的两种处理方式:

- 交给内核进行处理 (缺省方式)
- 由进程自行处理

详见表 4.1。

注

- 1). 其中 core 转储指把该进程内存中的有关信息进行转储 (dump), 生成 core 文件。在使用 gdb 调试工具对程序进行调试时, 通常需要使用 core 文件。
- 2). 进程接收到信号后有其自行处理成为信号的捕获, 但是信号 SIGKILL 和 SIGSTOP 不能有进程捕获, 他们必须由内核进行处理。
- 3) 信号无论是由内核或是进程处理, 都可以被忽视, 即不进行任何处理, 但是信号 SIGKILL 和 SIGSTOP 不能被忽视。

4.5.3 信号处理函数

1. 数据结构

当进程接收到信号，并且该信号没有被阻塞的话，进程就执行信号处理函数完成对信号的处理，每种信号都有其对应的处理函数，进程对所有信号处理函数集中由 `signal_struct` 结构体来管理，进程任务结构体中成员项 `sig` 指向该结构体。在 `include/linux/sched.h` 中定义了 `signal_struct` 结构体：

```
Struct signal_struct{
    Int count;
    Struct sigaction action[32];
};
```

其中，

- `count`: 共享处理信号函数的计数值。一般是子进程继承父进程的信号机制时的计数。
- `action[]`: 是该进程的信号处理函数表，32 个元素对应 32 种信号。该数组是 `sigaction` 结构体，它定义在 `include/asm-i386/signal.h` 中

```
Struct sigaction {
    _sighandler_t sa_handler;
    Sigset_t sa_mask;
    Unsigned long sa_flags;
    Void(*sa_restorer)(void);
};
```

其中

- `sa_handler` 是指向信号处理函数的指针,通常是用户自行设定的信号处理函数。当 `sa_handler` 的值是系统定义的以下符号常量时，它不是信号处理函数的入口地址，其值和意义如下：
 - `SIG_DEL` 0 缺省处理，由内核执行系统设定的信号处理函数
 - `SIG_IGN` 1 忽视信号，不进行信号处理
 - `SIG_ERR` -1 信号处理时返回的错误，一般用于判断函数的返回值是 否正确。
- `sa_mask` 是一个信号屏蔽码，当进程处理某一个信号时，它被逻辑加（OR）到接收进程的信号屏蔽码 `blocked` 上，进程信号屏蔽码的这种改变只是在信号处理期间有效，其目的是在进程执行信号处理过程中屏蔽其它到达的信号。
- `sa_flags` 是信号处理标志，主要有
 - `SA_ONESHOT` 信号到达时，启动信号处理函数
 - `SA_NOMASK` 不使用 `sa_mask` 改变进程的信号屏蔽码
- `sa_restorer` 是一个函数指针，目前未用，保留以供扩充。

进程信号处理的有关数据结构之间的关系如图 4.11。

Task_struct

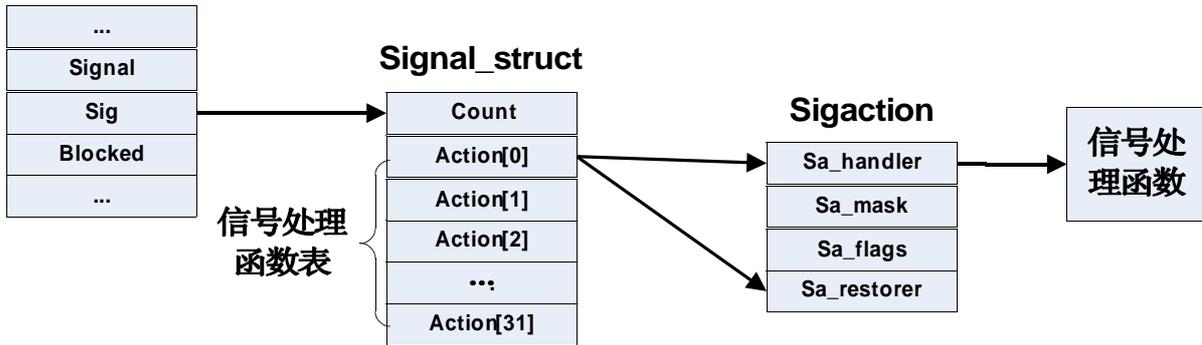


图 4.11 Linux 信号的数据结构

2. 处理函数 signal

Linux 系统提供了用户自己设置信号处理函数的方法，它由系统调用 `signal()` 完成。在 `signal()` 中进一步调用内核函数 `sys_signal()` 实现函数设置的功能。该内核函数定义在 `kernal/signal.c` 中：

```
Asmlinkage unsigned long sys_signal (int signum, _sighandler_t handler);
```

参数说明：

- `signum`：信号值，指明要设置哪个信号的函数；
- `handler`：用户设置的处理函数的首地址。（也可以是 `SIG_DEL`、`SIG_IGN`）

函数简要说明：

```
Struct sigaction tmp; /*用于暂存信号处理函数的有关信息。*/
```

```
...
```

```
If (signum<1|| signum>32) return -EINVAL; /*判断 signum 给定的信号值是否合理*/
```

```
If (signum==SIGKILL||signum==SIGSTOP) return -EINVAL; /*若为这两个信号，则不能被捕获，即用户不能为它们设定处理函数*/
```

```
If (handler! =SIG_DFL&& handler! =SIG_IGN) /*若信号不是指定为缺省处理或
```

```
{ 忽视，则确认给定的处理函数使
```

```
用存储空间的有效性*/
```

```
Err = verify_area(VERIFY_READ, handler, 1);
```

```
If (err) return err;
```

```
}
```

经过上面的检查确认后，开始使用 `tmp` 设置进程的信号处理函数结构体。

```
Memset (&tmp,0,sizeof(tmp)); /* 首先把该结构的存储空间全部清 0*/
```

```
Tmp.sa_handler =handler; /*把参数 handler 指定的信号函数处理函数首地址置入 tmp 的 sa_handler*/
```

```
Tmp.sa_flags = SA_ONESHOT|SA_NOMASK; /*设置 sa_flag*/
```

```
Current->sig->action[signum-1]=tmp; /*把 tmp 的内容复制到当前进程的处理信号函数表中与指定信号对应的数组元素中。*/
```

```
Check_pending(signum); /*设置当前进程任务结构体的 signal 成员项*/
```

```
Return (unsigned long) handler; /*返回 handler 的值，即原信号处理函数的首地址*/
```

3. 程序例

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

```

Int count=0;
Void ctrl_c_count(int);
Main()
{
    int c;
    void(*old_handler)(int);
    old_handler=signal(SIGINT,ctrl_c_count);
    while((c=getch())!="\n");
    printf("Ctrl_C count=%d\n",count);
    signal(SIGINT,old_handler);
}
Void ctrl_c_count(int dump)
{
    Printf("Ctrl_C\n");
    Count++;
}

```

4. 4 Linux 管道

4.6.1 管道的概念

管道是 linux 进程通信的一种手段，使用管道通信时，两个进程中的一个进程（写管道进程）从管道的一端把数据送入管道，另一个进程（读管道进程）从管道的另一端得到这些数据。管道实际上就是一种共享文件，所以管道机制是以文件系统为基础实现的。数据在管道中以先进先出的方式，并以字符流的形式传送。



图 4.12 管道通信示意图

管道分两种：无名管道、命名管道。它们的内部结构是一致的，但是用方式不同。

- 无名管道：只能在父子进程之间通信
- 命名管道：可以在任意进程间通信。

4.6.2 无名管道

linux 管道可以在终端的命令行中使用，也可以在程序中使用。

1. 终端使用

在终端键入 Linux 命令时，可以使用无名管道连接两个命令，例如

```
$ls -l | more
```

其中“|”就是管道命令符，它的作用是把第一条命令的输出与第二条命令的输入连接在一起。

2. 程序中使用

在程序设计中无名管道的建立由系统调用 `pipe()` 实现，其定义：

```
#include <unistd.h>
```

```
int pipe(int fildes[2]);
```

其中 `fildes[]` 是具有两个元素的 `int` 型数组。在调用 `pipe()` 建立一个无名管道后，使用两个文件标识号来表示管道的两端（一端写，一端读），并记入 `fildes[]` 中。其中 `fildes[0]` 是读取管道的文件标识号，`fildes[1]` 是写入管道的文件标识号。

说明：

- 父子进程使用无名管道通信是建立在子进程继承父进程资源的基础上。父子进程通信时，必须先建立管道，再创建子进程。
- 使用管道时必须确定管道通信的方向，且一旦确定后不能改变。
- 父子进程中一个进程只能使用一个文件标识号，所以另一个不使用的标识号可以使用系统调用 `close()` 关闭它。



图 4.13 使用无名管道的进程通信

例：在该例中，父进程建立管道后创建一个子进程。子进程的任务是把一组字符串信息写入管道，父进程在子进程完成任务终止后，从管道中读取信息并显示在显示器上。

```
#include <stdio.h>
Main()
{
    Pid_t pid;
    Int fds[2];
    Char buf1[50],buf2[50];
    Pipe(fds);      /*建立无名管道*/
    If ( (pid=fork ()) <0) /*创建子进程失败，程序终止*/
    {
        Printf ( "Fork () Error\n" );
        Exit (1) ;
    }
    Else if(pid==0) /*子进程*/
```

```

    {
        Close(fds[0]); /*关闭不使用的文件标识号*/
        Sprintf(buf1,"these are transmitted data\n"); /*把信息写入缓冲区 buf1*/
        Write(fds[1],buf1,50); /*把缓冲区 buf1 中的信息写入管道*/
        Exit (1) /*子进程终止*/
    }
    Else
    {
        Close(fds[1]); /*关闭不使用的文件标识号*/
        Wait (0) ; /*等待子进程结束*/
        Read(fds[0],buf2,50); /*读管道信息并把信息置入 buf2*/
        Printf ( "%s\n", buf2) ; /*显示 buf2 中的信息*/
    }
}

```

继续说明：

- 无名管道与一般文件不同，它没有纳入文件系统的目录，不占用外存空间，仅使用内存作为数据传输的缓冲区。
- 缓冲区的大小决定每次写入管道的字节数。该值由全局符号常量 PIPE BUF 确定。缺省值为一个物理页面。
- 由文件系统管理，但此时没有文件系统知识基础，故暂时不讲详细管理

4.6.2 命名管道

又称 FIFO 管道。命名管道与无名管道的区别：命名管道有文件名，在文件系统中可见；可以实现任意进程间的通信。

1. 终端使用

使用 mkfifo 建立一个命名管道。

例：\$mkfifo myfifo; /*建立一个名字为 myfifo 的管道。

此时使用 ls 命令，就可以查看到该文件信息。

```
$ls -l myfifo
```

```
prw-r--- wang user 0 fen 22 13: 45 myinfo /*第一个字符 p，表示是 FIFO 文件
```

在命名管道建立后，就可以用它在两个进程间进行通信。如：

```
$cut -c1-5< myfifo&
```

```
$cat file1 >myfifo
```

其中 cat 命令把文件 file1 的内容写入管道 myfifo，命令 cut 从管道 myfifo 中读出文件的内容进行裁剪后显示每行的前 1~5 个字符。

2. 程序中使用

1) 建立命名管道

方法一：调用 C 函数 mkfifo () 实现的，其定义如：

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

```
int mkfifo(const char *path, mode_t mode);
```

其中：

path: 指明要创建的命名管道的路径和名字;

mode: 指明管道访问的权限。

创建成功返回 0，否则为负数。

另一种方法：使用 linux 系统调用 `mknod()`

`mknod()` 可以建立任何类型的文件，在建立命名管道时使用的形式如下：

`mknod(path, mode|S_FIFO,0)`;

参数含义与上同，其中 `S_FIFO` 表示建立 FIFO 文件。

这两种方法作用基本相同：建立命名管道的目录结构、inode 节点、file 文件结构体等。

2) 打开管道

由于任何进程都可以通过命名管道进行通信，所以在使用命名管道时，必须先打开它，由系统调用 `open()` 实现。

`open(char *path, int mode)`

其中：

path: 指明要使用的命名管道的路径和名字;

mode: 指明管道访问的模式：`O_RDONLY`(只读)、`O_WRONLY` (只写)

创建成功返回文件标识号，否则为负数。

注：在使用文件操作对管道进行各种操作时，要使用文件标识号，而不是管道名。

例：有两个程序，其中 `wrfifo.c` 把一组信息写入管道，另一个程序 `rdfifo.c` 把管道中的信息读出后显示在屏幕上。

*/*读管道程序 rdfifo.c*/*

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

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

```
#include <fcntl.h>
```

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

```
#include <stdlib.h>
```

```
#include <limits.h>
```

```
Main(void)
```

```
{
```

```
    Int fd,len;
```

```
    Char buf[PIPE_BUF];
```

```
    Mode_t mode=0666;
```

```
    If(mkfifo("fifo1",mode)<0)
```

```
    {
```

```
        Printf("mkfifo error\n");
```

```
        Exit(1);
```

```
    }
```

```
    If((fd=open("fifo1",O_RDONLY))<0)
```

```
    {
```

```
        Printf("pipe open error\n");
```

```
        Exit(1);
```

```
    }
```

```
    While(len=(read(fd,buf,PIPE_BUF-1))>0)
```

```
        Printf("read fifo pipe: %s",buf);
```

```
    Close(fd);
```

```

}

/*写管道程序 wdfifo.c*/
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
Main(void)
{
    Int fd,len;
    Char buf[PIPE_BUF];
    Mode_t mode=0666;
    If((fd=open ( "fifo1",O_WRONLY))<0
    {
        Printf("pipe open error\n");
        Exit(1);
    }
    For (i=0; i<3;i++)
    {
        Len=sprintf(buf,"write fifo pipe from %d at %d times\n",getpid(),i+1);
        Write(fd,buf,len+1);
    }
    Close(fd);
}

```

在 linux 终端上运行这两个程序:

```
$./rdfifo&
```

```
$/wrfifo
```

结果:

```
Write fifo pipe from 945 at 1 times
```

```
Write fifo pipe from 945 at 2 times
```

```
Write fifo pipe from 945 at 3 times
```

4. 7 IPC 信号量机制

Linux 的信号量机制有两种:

- 其本身设置的信号量机制
- 引进 UNIX SYSTEM V 的 IPC (Internal Process Communication) 中的信号量机制

本节介绍后者, 其涉及到的函数和数据结构分别定义在 Linux 源文件的 ipc/sem.c 和 include/linux/sem.h

4.7.1 信号量与信号量集合

IPC 信号量机制的工作原理与之前原理部分介绍的信号量的工作原理基本相同。但是它更完善、更方便使用。

1. 信号量

定义：

系统中每个信号量对应一个信号量结构体 `sem`，其定义如下：

```
Struct sem
{
    Short  semval;    /*信号量的值*/
    Unshort sempid; /*记录对信号量最后一次实施操作进程的PID*/
}
```

PV 操作

（为了解决死锁）IPC 信号量机制在这方面做了改进，它可以使用原语一次对多个信号量进行操作。为此，引进了信号量集合的概念。

2. 信号量集合

在 IPC 信号量机制中，把进程需要访问资源对应的信号量组成一个信号量集合，并可以使用操作原语一次性地对信号量集中的多个信号量进行 PV 操作。

信号量数组：在 IPC 信号量机制中把多个信号量组成一个信号量集合，该集合由信号量结构体 `sem` 组成，称为信号量数组。

信号量集合描述符：系统中的每个信号量集合用一个描述符描述其特征和记载其管理信息。其定义如下：

```
Struct semid_ds
{
    Struct_ipc_perm  sem_perm;    /*对信号量集合的访问权限*/
    Time_t          sem_otime;    /*最后一次对信号量集进行操作的时间*/
    Time_t          sem_ctime;    /*最后一次修改信号量集的时间*/
    Struct sem       *sem_base;    /*指向信号量数组*/
    Struct sem_queue *sem_pending; /*指向等待队列头*/
    Struct sem_queue *sem_pending_last; /*指向等待队列尾*/
    Struct sem_undo  *undo;        /*进程终止时需要使用 sem_undo 结构体中的信
                                    息，对信号量集合进行有关操作*/
    Unshort          sem_nsems;    /*信号量集合中信号量的数目*/
}
```

3. 信号量集合的集中

IPC 对系统中的所有信号量集合进行集中管理，把所有的信号量集合描述符组织在一个 `semary[]` 数组中，其定义如下：

```
Static struct semid_ds *semary[SEMMNI];
```

其中 `SEMMNI` 为数组大小，是系统中可以设置的信号量集合的最大数目，其缺省值为 128，宏定义如下：

```
#define SEMMNI 128
```

4.7.2 信号量集合的创建和检索

1. semget ()

系统为每个信号量集合设定了一个唯一的标识号 ID，IPC 提供了创建信号量集合和获取信号量集合标识号的系统调用 `semget ()`，其原型定义如下：

```
int semget (key_t key, int nsems, int semflg);
```

该系统调用正常则返回值为信号量集合的标识号，出错则返回负数。

- **Key:** 要创建或要获取的信号量集合的标志键值，可以由用户指定，也可使用符号常量 `IPC_PRIVATE` 由系统给定。
- **Nsems:** 指出要创建的信号量集合中包含的信号量的个数。
- **Semflgs:** 操作标志；指定了信号量集合的访问权限和操作模式。其取值或符号常量及意义如下：
 - 0400 允许创建者读
 - 0200 允许创建者写
 - 0040 允许创建者同组用户读
 - 0020 允许创建者同组用户写
 - 0004 允许其它所有的进程读
 - 0002 允许其它所有的进程写
 - `IPC_CREAT(00001000)` 创建新的信号量集合
 - `IPC_EXCL(00002000)` 检索信号量集合

前六项指定了信号量集合的访问权限，后两项指定了操作模式。访问权限和操作模式可以使用逻辑与 (&) 组合在一起表示复合属性。

2. 创建信号量集

若操作模式设定为 `IPC_CREAT`：

- 则当系统中尚没有建立与 `key` 对应的信号量集合，则建立这个新的集合，并返回新集合的标识号；
- 当系统中已经存在与键值 `key` 对应的信号量集合，则返回这个集合的标识号；
- 当不能创建，则返回 -1

若操作模式设定为 `IPC_CREAT|IPC_EXCL`：

- 与前不同的是，当系统中已经存在与键值 `key` 对应的信号量集合，则返回错误值 `-EEXIST`。

例：建立一个键值为 `KEY`，包含 1 个信号量，允许任何进程读写的信号量集合时，调用函数的形式为：`id=semget(KEY,1,0666|IPC_CREAT)`；

3. 检索信号量集

检索一个信号量集合的标识号时，只需将 `semflg` 中的操作模式设为 `IPC_EXCL`。若存在，返回集合的标识号，否则，返回 -1；

4.7.3 信号量 PV 操作

IPC 中没有对信号量分别设置 P 和 V 操作原语，而是统一由具有原语性质的系统调用 `semop ()` 实现的，通常称其为信号量操作函数。其定义如下：

```
int semop (int semid, struct sembuf *sops, unsigned nsops);
```

- Semid: 实施 pv 操作的信号量集合的标识号
- Nsops: 本次实施操作的信号量的个数
- Sops: 指向一个信号量操作数组。因为每次对信号量集合中实施操作的信号量个数不同，不同信号量实施的操作不同，所以必须指明本次操作是对哪些信号量，实施哪些操作。该数组的元素个数就是 Nsops。

Sops 中的每个元素是一个 sembuf 结构体，它由系统定义：

```
Struct sembuf
{
    Ushort sem_num;    /*指出信号量在信号量数组中的下标*/
    Short   sem_op;    /*指出操作的种类*/
    Short   sem_flg;   /*指出操作的标志*/
}
```

说明：

Sem_op 的值决定操作的类型：

- ✧ Sem_op 的值是负数：表示进程请求资源，则实施 P 操作，把 semval 的值减去 sem_op 绝对值。
- ✧ Sem_op 的值是正数：表示进程释放资源，则实施 V 操作，把 semval 的值加上 sem_op 以上两种情况下，若对信号量集实施操作后，所有信号量的值 semval 均大于等于 0，则函数返回 0，表示进程所需的多个资源都可用，此时进程可以继续运行；否则，只要有一个 semval 结果为负数，则表示进程需要的这种资源不可用，进程被阻塞，并将本次操作加入该信号量集合的等待队列。
- ✧ Sem_op 的值为 0。此时若 semval 也是 0，则函数返回，调用 semop 的进程继续执行；若 semval 非 0，则进程被阻塞。

sem_flg：控制进程的执行。通常取值 0；若指定为 IPC_NOWAIT，则在执行 semop 操作时，即使出现需要进程阻塞的情况，也不阻塞，而是继续运行。

4.7.4 信号量操作等待队列

1. 信号量操作等待队列

上节提到如果进程执行信号量操作时被阻塞，则将把该次操作被加入到信号量集合的等待队列。IPC 每个信号量集合都有一个等待队列，分别由其描述符中的成员向 sem_pending 和 sem_pending_last 指向其头部和尾部。该等待队列是由 sem_queue 结构体组成的双向循环链表。Sem_queue 结构体定义如下：

```
Struct sem_queue
{
    Struct sem_queue *next;    /*指向队列后一个节点*/
    Struct sem_queue *prev;    /*指向队列前一个节点*/
    Struct wait_queue *sleeper; /*指向被阻塞进程*/
    Struct sem_undo *undo;
    Int pid; /*实施操作的进程的 PID*/
    Int status;
    Struct semid_ds *sma; /*指出对哪个信号量集合实施操作*/
    Struct sembuf *sops; /*指向是进程阻塞的操作数组*/
}
```

```

    Int          nsops; /*指出操作数组中操作的个数*/
}

```

2. 将信号量操作移出等待队列

上节我们介绍了，进程执行信号量操作时，出现进程阻塞，操作放入信号量操作等待队列的情况。下面我们介绍如何将进程唤醒、将操作冲等待队列移出的情况：

若某个进程在执行 `semop()` 时没有阻塞，函数将检查该信号量集的操作等待队列：

- 若无操作等待，则返回；
- 若有操作等待，则依次重新执行这些操作：
 - 1) 若进程仍需等待，则操作保留在等待队列中；
 - 2) 若进程可以继续执行，则通过该 `sem_queue` 结构体中的 `sleeper` 在进程等待队列中找到该进程，并将其唤醒；再将该 `sem_queue` 结构体从操作等待队列中删除。

4.7.5 信号量控制操作

IPC 信号量机制提供了可以对信号量集合进行多种控制操作的系统调用 `semctl()`，它能实现对信号量的初始化、查询、修改、删除等功能，当然只有具有相应权限的进程才能执行相应的操作。其原型如下：

```

Int semctl (int semid, int semnum, int cmd, union semun arg) ;

```

Semid: 指向操作对象，即信号量集合标记号。

Semnum: 信号量的索引号，指明信号量在信号量数组中的下标。

Cmd: 指定各种不同操作。

Arg: 用于传递执行各种控制操作时所需的参数。`union semun` 是 IPC 定义的一个联合体。其定义：

```

Union semun
{
    Int val;
    Struct semid_ds *buf;
    Unshort *array;
    Struct seminfo *_buf;
    Void*_pad;
}

```

下面介绍 `cmd`，对应的符号常量和意义所在：

IPC_STAT: 读取信号量集合 `semid_ds` 结构体的内容，并把它写进参数 `arg` 给出的 `semnum` 联合体中成员项 `buf` 指定的 `semid_ds` 结构中。此时无视参数 `Semnum`。

IPC_SET: 修改信号量集合 `semid_ds` 结构体的成员项 `sem_perm` 结构中的某些成员项的值；同时 `sem_ctime` 被自动更新。只有信号量集合创建者、同组用户和超级用户可以执行该操作。修改值取自参数 `arg` 给出的 `semun` 联合体中 `buf` 指定的 `semid_ds` 结构体的第一个成员项。

IPC_RMID: 从内核中删除信号量集合。只有信号量集合创建者和超级用户。在该信号量集合中等该信号量操作的所有进程被唤醒，并得到错误信息 `EIDRM`。

GETPID: 获取信号量数组中以参数 `semnum` 做为索引值指定信号量的 `sempid` 值。它是对信号量最后一个执行 `semop()` 操作的进程的 `PID`。

SETVAL: 设置信号量数组中以参数 `semnum` 做为索引值指定信号量的值。`Arg` 的 `semun` 联合体成员项 `val` 中给出要设置的值。

GETVAL: 获取信号量数组中以参数 `semnum` 做为索引值指定信号量的 `semval` 值。其值写入 `arg` 的

semun 联合体成员项 val 中。该值也作为函数返回值返回。

SETALL: 设置信号量集合中所有信号量的值，设置值存放在 Arg 的 semun 联合体成员项 array 中。

GETALL: 获取信号量集合中所有信号量的值，并存放在 Arg 的 semun 联合体成员项 array 中。

GETNCNT: 得到等待以 semnum 为索引指出的信号量的值为非 0 的进程的个数。

4.7.6 信号量的程序例

下面给出两个使用 IPC 信号量机制的程序例。

程序例 1: 创建一个信号量集合，并对信号量进行 P 操作。

```
/*mksem.c*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
Int main (void)
{
    Int semid;          /*信号量集合标识号*/
    Int nsems=1;       /*信号量集合中信号量的个数*/
    Int flags =6;      /*对信号量集合的访问权限，允许所有进程进行读写*/
    Struct sembuf buf; /*信号量操作数组*/

    /*创建信号量集*/
    Semid = semget (IPC_PRIVATE,nsems,flags) ;
    If(semid<0) /*若返回值为负，则出错*/
    {
        Printf ( “semapher ceate failer! \n”);
        Exit(EXIT_FAILURE);
    }
    Printf ( “semapher ceated: %d\n”,semid) ;

    /*设置操作数组个成员项的值*/
    Buf.sem_num=0; /*下标为 0，因为只有一个信号量*/
    Buf.sem_op =1; /*信号量加 1 操作*/
    Buf.sem_flg=IPC_NOWAIT; /*进程不阻塞*/

    If ( (semop (semid, &buf,nsems) <0) /*执行信号量操作*/
    {
        /*信号量操作失败*/
        Printf ( “semapher operation failer! \n”);
        Exit(EXIT_FAILURE);
    }
}
```

```

    System("ipcs -s"); /*执行键盘命令 ipcs -s, 显示 IPC 信号量*/
    Exit(EXIT_SUCCESS); /*成功退出*/
}

```

程序运行结果如下:

```

$./mksem
semapher ceated: 520
-----semapher arrays-----
Key      semid  owner  perms  nsems  status
0x00000000  520   wang   666    1

```

程序例 2: 通过信号量控制函数删除信号量集合。

```

/*sctlsem.c*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>
Int main (int argc,char *argv[ ])
{
    Int semid; /*信号量集合标识号*/
    If (argc !=2)
    {
        Puts ( "USAGE: sctl<semaphore id>" );
        exit(EXIT_FAILURE);
    }
    Semid=atoi(argv[1]); /*从命令行参数得到信号量集合标识号*/
    /*删除信号量集合*/
    If ( (semctl (semid, 0, IPC_RMID) ) <0) /*调用信号量控制函数*/
    {
        /*返回负值, 失败退出*/
        Printf ( "semapher control failer! \n" );
        exit(EXIT_FAILURE);
    }
    Else
    {
        Puts ( "semapher removed!" );
        System("ipcs -s"); /*执行键盘命令 ipcs -s, 显示 IPC 信号量*/
    }
    Exit(EXIT_SUCCESS); /*成功退出*/
}

```

程序运行时, 需要给出所要删除的信号量集合的标识号, 程序运行结果如下:

```

$./ sctlsem.c 520
semapher removed!

```

4. 8 IPC 消息队列

前面介绍了两种 linux 中进程通信的方法：信号和管道。它们的缺点是，信号每次只能传递一个数据；管道只能在两个进程间单向交换信息。接下来我们将介绍两种高级通信机制 IPC 的消息队列和共享内存。

本节介绍 IPC 消息队列，其中涉及的函数和数据结构分别定义在 ipc/msg.c 和 include/linux/msg.h。

4.8.1 消息队列的结构

IPC 的消息队列的工作原理与前面原理部分介绍过的进程间的消息通信机制基本一致。IPC 消息队列一般用于客户机/服务器（C/S）模型中，客户机进程向服务器发送请求服务的消息，服务器进程接收到消息后执行客户机请求。

1. 消息

IPC 中一个消息由消息头和消息正文组成。消息头是一个 msg 结构体。其定义如下：

```
struct msg
{
    struct msg *msg_next; /*指向下一个消息*/
    long msg_type; /*消息类型：大于 0,是通信双方约定的消息标志*/
    char *msg_spot; /*消息正文地址*/
    time_t msg_time; /*消息发送时间*/
    short msg_ts; /*消息正文大小，最大长度由 MSGMAX 决定，缺省为 4057 字节*/
}
```

2. 消息队列

Linux 中可以根据进程的需要建立多个 IPC 消息队列，消息队列的最大数量由符号常量 MSGMNI 决定，缺省为 128。系统对所有消息队列统一管理。每个消息队列有唯一的标识号。每个消息队列是由消息结构体构成的单向链表。

描述消息队列的数据结构 struct msqid_ds，称为消息队列描述符。与消息队列一一对应。其定义如下：

```
struct msqid_ds
{
    Struct ipc_perm msg_perm; /*访问权限*/
    Struct msg *msg_first; /*指向消息队列头*/
    Struct msg *msg_last; /*指向消息队列尾*/
    Time_t msg_stime; /*最近发送消息的时间*/
    Time_t msg_rtime; /*最近接收消息的时间*/
    Time_t msg_ctime; /*最近修改消息的时间*/
    Struct wait_queue *wait; /*等待接收消息的进程等待队列*/
    Struct wait_queue *wait; /*等待发送消息的进程等待队列*/
    Unshort msg_cbytes; /*队列中当前字节数*/
    Unshort msg_qnum; /*队列中消息数*/
}
```

```

Unshort msg_qbytes;      /*队列最大字节数,不能超过 MSGMN (缺省 16384) */
Unshort msg_lspid;      /*最近发送进程的 PID*/
Unshort msg_lrpid;      /*最近接收进程的 PID*/
}

```

说明：IPC 消息队列用于多个进程间的数据交换，其中包含多个进程发送来的消息，这些消息又要传递给不同的进程。故发送进程在发送的消息中需要设定一个类型 `msg_type`（详见消息结构体），接收进程在消息队列中按照类型就可以找到对方进程发来的消息。

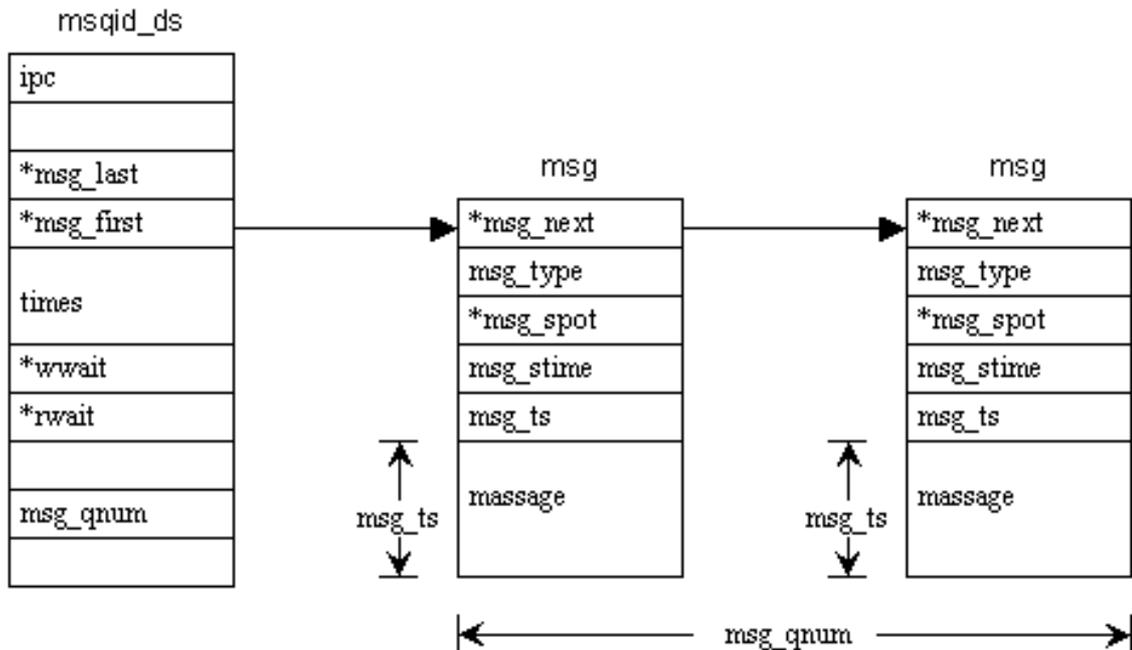


图 4.16 IPC 消息队列的结构

4.8.2 消息队列的生成与控制

1. 建立及检索消息队列

在程序中使用 `msgget()` 建立一个消息队列或者检索消息队列的标识号。其定义为：
`Int msgget (key_t key, int msgflg) /*msgflg 的值与 semget () 中的 semflg 完全相同*/`
 调用成功则返回消息队列的标识号，否则返回-1；

2. 消息队列的控制

对消息队列的控制操作由系统调用 `msgctl()` 实现，其定义如下：

```
Int msgctl (int msqid, int cmd, struct msqid_ds *buf)
```

Msqid: 消息队列的标识号。

Cmd: 操作类型。

Buf: 用于向函数传递数据和从函数的得到的结果数据。

说明：`cmd` 参数的种类与 `semctl()` 类似，但制定操作的意义不同。主要有：

MSG_STAT 或 IPC_STAT: 把指定消息队列的描述符内容拷贝到 `buf` 指定的 `msqid_ds` 结构体中。

IPC_SET: 允许修改制定消息队列描述符内容。该操作用于修改消息队列的访问权限，即 `msg_perm` 的成员项。超级用户可以修改 `msg_dbyte` 的值。变更后，`msg_ctime` 的值自动更新。

IPC_RMID: 删除消息队列及其数据结构，该操作只有超级用户或消息队列生成者进行。

MSG_INFO 或 IPC_INFO: 把与消息队列有关的最大值数据输出到 msginfo 结构体中。在该结构体中记录着 IPC 消息队列的消息数、字节数和消息的字节最大数。

4.8.3 消息的发送与接收

进程使用系统调用 msgsnd () 向消息队列写消息；接受进程使用系统调用 msgrcv () 从消息队列读取消息。

Int msgsnd (int msqid, struct msgbuf *msgp, int msgsz, int msgopt) ;

Int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, int msgtyp, int msgopt) ;

Msgsnd () 调用成功返回 0，msgrcv () 调用成功返回值为接收到的消息的长度；两个函数调用失败返回错误信息。

- **Msqid:** 消息队列标识号。
- **Msgp:** 指向一个 msgbuf 结构体 其定义为：

```
Struct msgbuf
{
    Long mtype;    /*消息类型*/
    Char mtext[1]; /*消息正文*/
}
```

说明：

1. msgbuf 结构体，相当于原理部分所提的消息发送区或消息接收区。
 2. Mtext，只有一个字节，并不表示消息正文本身，仅指名消息正文地址。
 3. 用户可以自行定义一个结构体来代替 Msgbuf 结构体。（参见程序例）
- **Msgsz:** 消息正文长度。
 - **Msgopt:** 操作模式。
 - 取值为 0 时**，表示忽略该参数。
 - 在接收函数中**，取值为 IPC_NOWAIT，表示即使消息队列没有需要的消息，进程也不阻塞，返回值为 ENOMEG，通知进程消息不存在。
 - 在发送函数中**，取值为 IPC_NOWAIT，表示即使消息队列已满，进程也不阻塞，返回值为 ENOMEG，通知进程重新发送消息。
 - **Msgtyp:** 从消息队列读取消息的方式。
 - Msgtyp 取值为 0**，读取消息队列的第一个消息。
 - Msgtyp 取值大于 0**，表示消息类型，这时读取队列中符合该类型的第一个消息。若 msgopt 指定为 MSG_EXCEPT，则读取队列第一个不符合该类型的消息。
 - Msgtyp 取值小于 0**，则表示读取小于其绝对值的类型值最小的第一个消息。当一个消息从消息队列中读取后，由系统自动从队列中将其删除。

4.8.4 消息队列的程序例

下面给出使用消息队列在进程间通信的程序例，其功能是在进程间完成文件的传送。一个发送进程把

两个文件送入消息队列，另外两个接收进程分别从消息队列各自接收一个文件。

该例由3个源程序文件组成：**user.h**是共用的头部文件，**filesnd.c**的功能是把两个文件读出并发送到消息队列，**filercv.c**的功能是从消息队列读取文件。

发送进程执行**filesnd.c**，两个接收进程执行**filercv.c**。

```
/*user.h*/
#define F_KEY          "copyfile" /*指定生成IPC键值的文件名*/
#define F_TYPE1        1          /*第一个文件的消息类型*/
#define F_TYPE2        2          /*第二个文件的消息类型*/
#define F_TYPE_END1    3          /*第一个文件接收完毕的消息类型*/
#define F_TYPE_END2    4          /*第二个文件接收完毕的消息类型*/
#define BUF_SIZE       256
```

```
Struct msg_data /*替代上节所讲msgbuf结构体,作为发送区(接收区)*/
{
    Long      type;
    Int       size;
    Char      buf[BUF_SIZE];
}
```

/*filesnd.c 数据发送程序*/

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include "user.h"
```

```
Close_all(int msgid, int fd1, int fd2)
```

```
{
    Close(fd1); /*关闭文件*/
    Close(fd2); /*关闭文件*/
    Msgctl (msgid, IPC_RMID,0) ; /*删除消息队列*/
}
```

```
Main (int argc, char **argv)
```

```
{
    Int fd1, fd2, msgid;
    Int end1=1, end2=1;
    Key_t key;
    Struct msg_data data1,data2;
    If (argc !=3) /*检查命令行参数*/
    {
        Printf (" usage: argv[0] File1 File2 <CR>\n") ;
        Exit (-1) ;
    }
```

```

}
If ( (fd1=open (argv[1], O_RDONLY) ==-1) /*打开第一个文件，只读*/
{
    Printf(“%s can’t opened\n”,argv[1]);
    Exit (-1) ;
}
If ( (fd2=open (argv[2], O_RDONLY) ==-1) /*打开第二个文件，只读*/
{
    Printf(“%s can’t opened\n”,argv[2]);
    Exit (-1) ;
}
If ( (key=ftok (F_KEY,'a') ) ==-1) /*生成 IPC 键值*/
{
    Close(fd1);
    Close(fd2);
    Printf(“Sender: create Key Error.\n”);
    Exit (-1) ;
}
If((msgid=msgget(key,IPC_CREAT|IPC_EXCL|0666))== -1) /*生成消息队列*/
{
    Close(fd1);
    Close(fd2);
    Printf(“Sender: create Message queue Error.\n”);
    Exit (-1) ;
}
Printf(“\nSender: create Message queue created.\n”);
Printf(“Copy %s and %s\n”,argv[1],argv[2]);
Data1.type=F_TYPE1;          /*确定两个文件的消息类型*/
Data2.type=F_TYPE2;
Data1.size= Data2.size=-1;
While(data1.size||data2.size) /*读取和发送两个文件*/
{
    If(data1.size)
    {
        Data1.size=read(fd1,data1.buf,BUF_SIZE) /*读取第一个文件*/
        If (data1.size== -1)
        {
            Close_all(msgid,fd1,fd2);
            Printf(“Sender: error read file%s \n”,argv[1]);
            Exit (-1) ;
        }
        /*写入消息队列*/
        If (msgsnd (msgid, (struct msgbuf*) &data1,sizeof(struct msg_data),0) ==-1)
        {

```

```

        Printf("Sender: error message send file%s \n",argv[1]);
        Exit (-1) ;
    }
}
If(data2.size)
{
    Data2.size=read(fd2,data2.buf,BUF_SIZE) /*读取第二个文件*/
    If (data2.size==-1)
    {
        Close_all(msgid,fd1,fd2);
        Printf("Sender: error read file%s \n",argv[2]);
        Exit (-1) ;
    }
    /*写入消息队列*/
    If (msgsnd (msgid, (struct msgbuf*) &data2, sizeof(struct msg_data),0) ==-1)
    {
        Printf("Sender: error message send file%s \n",argv[2]);
        Exit (-1) ;
    }
}
}
Printf("Sender: data transmission to message queue complete \n");
Printf("Sender: waiting for receiver to finish reading\n");
/*等待接收进程结束*/
While(end1||end2)
{
    Struct msgbuf rec;
    If(msgrcv(msgid,&rec,sizeof(struct msgbuf),TYPE_END1,IPC_NOWAIT)!=-1)
        End1=0;
    If(msgrcv(msgid,&rec,sizeof(struct msgbuf),TYPE_END2,IPC_NOWAIT)!=-1)
        End2=0;
}
Printf("\nSender: disconnection of the two receiver processes \n");
Close_all(msgid,fd1,fd2);
}

```

/*filercv.c 数据接收程序*/

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include < sys/stat.h >

```

```

#include "user.h"

Main (int argc, char **argv)
{
    Int fd, ret, msgid;
    Int no;
    Int type;
    Key_t key;
    Struct msg_data data;
    Struct msgbuf dis_msg;
    If (argc != 3) /*检查命令行参数*/
    {
        Printf (" usage: filerv 1|2 File_name <CR>\n") ;
        Exit (-1) ;
    }
    No=atoi(argv[1]);
    If ( (no != 1) &&(no != 2))
    {
        Printf (" usage: %s 1|2 File_name <CR>\n", argv[0]) ;
        Exit (-1) ;
    }
    If (no == 1) type = F_TYPE1;
    Else type = F_TYPE2;
    If((key=ftok(F_KEY,'a')) == -1) /*生成 IPC 键值*/
    {
        Printf("receiver: create Key Error.\n");
        Exit (-1) ;
    }
    If ( (msgid=msgget (key, IPC_EXCL) ) == -1) /*得到键值对应的消息队列标识号*/
    {
        Printf ( "receive: create message queue error\n" ) ;
        Exit(-1);
    }
    Printf ( "\nreceive %d connected to the message queue \n",no,getid() ) ;
    If ( (fd2=open (argv[2], O_WRONLY) == -1) /*打开文件，只写*/
    {
        Printf("'%s can't opened\n",argv[2]);
        Exit (-1) ;
    }
    While (1)
    {
        /*从消息队列接收消息*/
        If(msgrcv(msgid, (struct msgbuf*) &data, sizeof(struct msg_data),type, 0) != -1)
        {
            Printf("Receiver %d error during reception.\n",no);
        }
    }
}

```

```

        Exit (-1) ;
    }
    If(data.size==0) break;
    Ret = write(fd,data.buf,data.size) /*把读取的消息写入文件*/
    If (ret==-1)
    {
        Close (fd) ;
        Msgctl (msgid, IPC_RMID, 0) ;
        Printf ( "receiver %d error during writing\n", no) ;
        Exit (-1) ;
    }
}
Close (fd) ;
Printf ( "receiver %d disconnection\n", no) ;
If (no==1)
    Dis_msg.mtype = TYPE_END1;
else
    Dis_msg.mtype = TYPE_END2;
Msgsnd(msgid,&dis_msg,sizeof(struct msgbuf),0);/*发送接收完毕消息*/
}

```

该程序分别由三个进程执行，使用的命令和运行结果如下：

```
#!/filesnd dbfile1 dbfile2 &
```

```
Sender: message queue create;
```

```
Sender: data transmission to message queue completed
```

```
Sender: waiting for receiver to finish reading
```

```
#!/filercv 1 file1&
```

```
Receiver 1 connected to the message queue
```

```
Receiver 1 disconnection
```

```
#!/filercv 1 file2
```

```
Receiver 2 connected to the message queue
```

```
Receiver 2 disconnection
```

```
Sender: disconnection of the two receiver processes.
```

4. 9 IPC 共享内存

共享内存是进程通信中**速度最快**的一种。该机制的实现与 linux 的存储管理密切相关。(可以考虑暂时不详细讲)。本节中出现的函数和数据结构分别定义在 linux 源文件 ipc/shm.c 和 include/linux/shm.h 中。

4.9.1 共享内存

基本思想：通过允许多个进程访问同一个内存区域实现进程之间的数据传送。进程不能直接访问物理内存空间，所以需要将共享的内存空间映射到通信进程的虚拟地址空间，这个过程对用户进程透明。（attach 结合, detach 脱离）

共享内存实现机制与信号量、消息管理机制一致。每一个共享内存有唯一标识，称为内存标记号，每个区域内存由一个数据结构记录他的管理信息，称为内存描述符 `shmid_ds` 结构体，定义如下：

```
Struct shmid_ds
{
    Struct ipc_perm shm_perm;      /* 访问权限*/
    Int shm_segsz;                /* 共享内存大小（字节）*/
    Time_t shm_atime;             /* 最近一次结合时间*/
    Time_t shm_dtime;             /* 最近一次分离时间*/
    Time_t shm_ctime;             /* 最近一次修改时间*/
    Unsigned short shm_cpid;      /* 创建者 pid*/
    Unsigned short shm_lpid;      /* 最后一次对共享内存进行操作的进程的 pid*/
    Short shm_nattch;             /* 当前结合进程的数量*/
    Unsigned short shm_npages;    /* 共享内存使用的页面数*/
    Unsigned long shm_pages;      /* 共享内存页表指针*/
    Struct vm_area_struct *attaches; /* 供结合进程使用的虚拟内存描述符*/
}
```

此处不对参数进行详细介绍，因为部分与前面介绍内容雷同，一部分设计内存管理。

4.9.2 共享内存的生成与控制

1. Shmget () 建立或检索共享内存

`Shmget (key_t key, int size ,int shmflg) ;`

- 此处不对参数进行详细介绍，因为部分与前面介绍内容雷同。
- **Size:** 共享内存大小，要说明的是：检索时，size 要小于共享内存实际大小。

2. Shmctl () 控制函数

`Shmctl (int shmid, int cmd, struct shmid_ds *buf)`

- 此处不对参数进行详细介绍，因为部分与前面介绍内容雷同。

➤ **Cmd** 指出的对共享内存的操作种类如下：

IPC_STAT: 获取共享内存的管理信息，并将其复制到 `buf`。

IPC_SET: 修改共享内存描述符内容。

IPC_RMID: 删除共享内存，必须在共享内存与所有进程分离后。

SHM_LOCK: 禁止共享内存的页面交换；

SHM_UNLOCK: 允许共享内存的页面交换；

4.9.3 共享内存的结合与分离

1. 与共享内存结合

系统调用 `shmat()`

```
int shmat (int shmid, char *shmaddr, int shmflg) ;
```

Shmid: 共享内存的标识号;

Shmaddr: 共享内存存在进程虚拟空间的起始地址。若为 0，则由系统在进程虚拟空间分配共享内存区域。

Shmflg: 操作特性。若为 `SHM_RDONLY`，只能对共享内存进行读取操作；若为 0，则可以读写。对内存没有只写操作。

2. 与共享内存分离

```
int shmdt (char *shmaddr)
```

Shmaddr: 进出虚拟空间中共享内存的首地址。执行该函数将释放进程虚拟空间中共享内存占用的区域，并修改 `shmid_ds` 中的相关项。

4.9.4 共享内存的程序例

下面给出使用共享内存实现进程通信的程序例。它由两个程序组成，第一个用于建立共享内存，第二个给出与共享内存结合和分离的程序方法。

*/*mkshm.c 生成和初始化内存共享区域*/*

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFSZ 4096 /*共享内存的大小*/
int main (void)
{
    int shmid;
    if((shmid=shmget (IPC_PRIVATE,BUFSZ,0666) )<0
    {
        printf("share memory error\n");
        exit(-1);
    }
    printf("segment created:%d",shmid);
    system("ipcs -m");
    exit(0);
}
```

程序运行结果如下：

```
./mkshm
```

```
Segment created: 48033
```

```
----shared memory segments-----
```

Key	shmid	owner	perms	bytes	nattch	status
0x00000000	48033	xm	666	4096	0	

```
/*atshm.c -与内存结合的程序*/
```

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

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

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

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

```
#include <stdlib.h>
```

```
Main (int argc, char *argv[])
```

```
{  
    Int shmid;          /*共享内存标识号*/  
    Char *shmbuf;      /*共享内存在进程虚拟空间的地址*/  
  
    If (argc != 2)  
    {  
        Printf ( "USAGE: atshm shmid\n" );  
        Exit(-1);  
    }  
    Shmid=atoi ( argv[1] );  
    /*与共享内存结合*/  
    If((shmbuf = shmat(shmid,0,0))<(char*)0)  
    {  
        Printf("shm attach error\n");  
        Exit(-1);  
    }  
    Printf("segment attach at %p\n",shmbuf);  
    System("ipcs -m"); /*显示结合信息*/  
  
    If ( ( shmdt ( shmbuf ) ) <0 )  
    {  
        Printf("shm detach error\n");  
        Exit(-1);  
    }  
    Printf("segment deached\n");  
    System("ipcs -m"); /*显示分离信息*/  
    Exit ( 0 ) ;  
}
```

```
程序运行结果:
```

```
$/ atshm 48033
```

```
Sgement attched at 0x40012000
----shared memory segments-----
Key      shmid    owner   perms  bytes  nattch  status
0x00000000 48033   xdm    666   4096   1
```

```
Sgement deatched
----shared memory segments-----
Key      shmid    owner   perms  bytes  nattch  status
0x00000000 48033   xdm    666   4096   0
```

4.9.5 IPC 对象

通常把 Linux 中使用的 System V IPC 的 3 种通信机制称为 IPC 对象 (IPC object)。

这三种机制有一定的统一性:

首先: 它们有统一的管理方式, 每一种机制都有描述符记录管理信息;

其次: 有格式统一的操作接口, 即系统调用。如生成函数 `***get()`、控制函数 `***ctl()`; 而且这些函数使用的符号常量也是一致的。

1. IPC 对象的键值

IPC 对象的另外一个重要特征: 系统使用一个键值和一个标识号来识别每一个 IPC 对象。键值和标识号都是唯一的, 且一一对应。为什么要使用两种识别方法呢?

因为: IPC 的标识号是创建者进程运行过程中动态生成的, 在其他进程中不可见, 这样就无法使用其 IPC 对象进行通信; 而键值是一个静态的识别码, 对所有进程都可见。

生成键值: 调用 c 函数 `ftok()`。定义:

```
Key_t ftok(char * pathname, char proj);
```

Pathname: 一个已经存在的路径和文件名;

Proj: 任意字符

使用同一个 IPC 对象进行通信的相关进程可以使用该函数和相同参数的到该对象的键值。(为了使得相关进程能够得到生成键值的参数, 一般的做法是把这些参数组织在一个头部文件中, 然后将其包括到进程程序的头部文件中。

2. IPC 对象的访问权限

为了限制进程对 IPC 对象的使用, 防止出现混乱。IPC 对其对象规定了一定的访问权限。结构体 `ipc_perm` 记录着 IPC 对象的访问权限等静态属性。其定义为:

```
Struct ipc_perm
{
    Key_t    key;      /*IPC 对象键值*/
    Unshort uid;      /*所有者用户标识*/
    Unshort gid;      /*所有者组标识*/
    Unshort cuid;     /*创建者用户标识*/
    Unshort cgid;     /*创建者组标识*/
    Unshort mode;     /*访问权限和操作模式*/
    Unshort seq;      /*序列号*/
}
```

进程访问 IPC 对象时，系统首先要根据该进程的用户标识和组标识，对照访问权限来确定本次访问是否合法。

3. IPC 对象的终端命令

在 linux 中提供了终端上查看和删除 IPC 对象的命令。常用的有 `ipcs` 和 `ipcrm` 两条命令。

➤ Ipcs 查看命令

常用格式：`ipcs [-asmq]`

选择项

-a 显示所有 IPC 对象信息

-s 显示信号量机制信息

-m 显示共享内存信息

-q 显示消息队列信息

➤ Ipcrm 删除命令

常用格式：`Ipcrm[shm|msg|sem] id` `/*id 为 IPC 对象的标识号*/`

第 5 章 存储管理

5. 1 存储管理的目的和功能

1. 存储管理目的：

- 1) 为多道程序的并发执行提供良好的环境，使每道程序都能在不受干扰的环境中运行。
- 2) 便于用户使用存储器，是用户从存储器的分配、保护和共享等繁琐的事务中解脱出来。
- 3) 提高存储器的利用率，以提高系统的吞吐量。
- 4) 从逻辑上扩充内存空间，使大的程序能在小的内存空间运行或允许更多的程序并发执行。

2. 存储管理的主要功能

主存储器的存储空间一般分为两部分：一部分是系统区，用于存放操作系统的程序和数据；另一部分是用户区，用于存放应用程序与用户的程序和数据。

存储管理主要是对用户区进行管理。

1) 内存分配

- a. 为每道程序分配内存空间，使他们“各得其所”
- b. 提高存储器的利用率，以减少不可用的存储空间。（即“零头”）
- c. 允许正在运行的程序申请附加内存空间，以适应程序或数据动态增长的需求。

为此存储分配机制应具有以下功能：

- (1) 记录每个存储区（分配单位）的状态，作为内存分配的依据

(2) 能动态的分配内存：指在进程运行期间，根据系统或用户的请求，分配其所需要的内存空间，并修改相应的空闲存储区表。

(3) 及时回收系统或用户进程释放的存储区。

2) 内存保护

内存保护的任务是确保每道程序都在自己的内存空间运行，互不干扰，为此系统需要每道程序都能：

- a. 不访问 os 的任何部分，包括程序区或数据区
- b. 执行中的进程不会转移到其他进程的 program 中去执行。
- c. 未经特殊安排，不能访问其他进程中的数据，若在执行中发生了上述情况，系统应能立即抛弃这样的指令。上述检查需要硬件完成，使用软件不仅会显著增减 cpu 开销，且大大降低了进程的运行速度。

3) 内存共享

4) 地址映射（讲述作业从源程序到装入的过程）

5) 内存扩充

内存扩充的任务是从逻辑上来扩充内存容量，是用户认为系统所拥有的内存空间比实际的空间大。为实现此任务，系统必须具有下述功能：

- a. 请求调入功能
- b. 置换功能

虚拟存储器——os 把主存和辅存两者融为一体，为用户提供一个超过实际主存容量的存储器。其容量由计算机的地质结构来决定。

5. 2 地址重定位

1. 作业的地址空间

名字空间：用户在使用汇编语言或高级语言编制作业的源程序时，一般要使用符号名来指定程序转移的目的地、子程序的入口地址以及要访问的数据等在作业中的位置。因此这个作业空间称为名字空间。

作业的逻辑地址空间（相对地址空间）：源程序经过汇编或编译后，形成目标程序，每个目标程序都是以 0 为基址顺序进行编址的，原来用符号名访问的单元用具体的数据——单元号取代。这样生成的目标程序占据一定的地址空间，称为作业的逻辑地址空间，简称逻辑空间。在逻辑空间中每条指令的地址和指令中要访问的操作数地址统称为**逻辑地址**。

作业的物理地址空间（绝对地址空间）：内存是由若干个存储单元组成的，所有存储单元顺序编号，每个存储单元有一个编号，这种编号可唯一标识一个存储单元，称为内存地址（或物理地址）。程序装入内存后，它们占用的主存区域是由绝对地址来确定指令和数据的位置的，通常把这些绝对地址的集合形成的作业空间称为作业的物理地址空间（绝对地址空间）。

图 5.2 作业地址空间的转换

	名空间		逻辑地址空间		物理空间
		0b		1000b	
		100b		1100b	Mov R1,[200]
	Mov R1,[data]				
		200b		1200b	
data	6817		6817		6817
		299b		1299b	

2. 地址映射（重定位）

逻辑地址和物理地址。这两者在多道程序环境下是不一致的，因此存储管理必须提供地址映射功能（重定位），用于把逻辑地址转换为物理地址。

重定位 { 静态重定位：有装配程序来完成。物理地址=起始地址+逻辑地址
 动态重定位：在程序执行过程中，由硬件地址映射机构来完成。基地址寄存器 BR，虚地址寄存器 VR，内存地址 $MR=BR+VR$

静态重定位：优点，不需要硬件支持、简单、速度快。

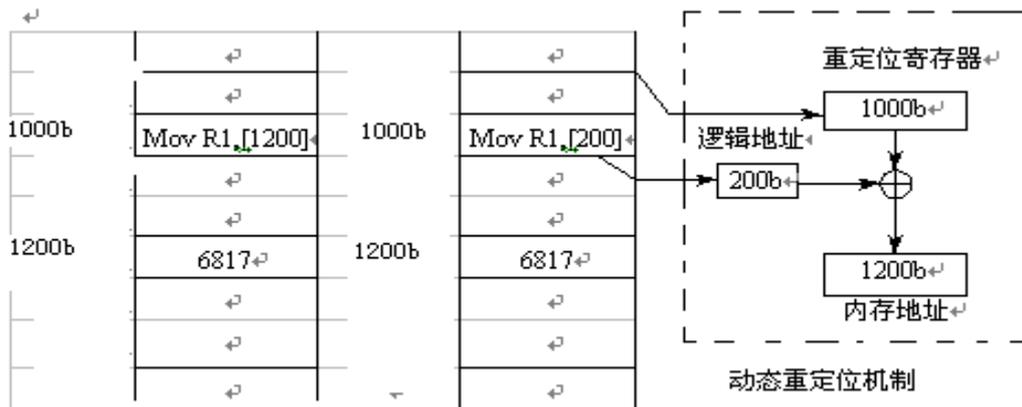
缺点，无法进行地址变换，无法实现虚拟存储器

- 1) 将程序一旦装入内存后就不能再移动，
- 2) 必须在程序执行前将有关部分全部装入
- 3) 必须占用连续的内存空间

动态重定位：缺点，需要硬件支持，且实现存储管理的软件算法比较复杂。

优点，为实现虚拟存储器提供了基础，有利于存储空间的共享

- 1) 可以将程序分配到不连续的存储空间
- 2) 在程序运行之前可以只装入它的部分代码即可投入运行，然后在程序运行期间，根据需要动态申请分配内存
- 3) 为用户提供一个比主存的存储空间大的多的地址空间。



a) 采用静态重定位后的内存空间

b) 采用动态重定位时内存空间及地址重定位示意图

5. 3 分区存储管理

单道系统：单一连续存储管理。

多道系统：最简单的方法是将内存用户区划分成若干区域，每个区域分配给一个用户作业，把用户作业一次性全部装入内存，并限定它们只能在自己的区域内运行。

5.3.1 固定分区管理

1. 基本思想：将内存空间划分为若干分区（划分原则由系统操作员或 OS 确定）每个分区中驻留一道程序，这些分区的长度可以不同，但分区的个数和每个分区的长度、位置是固定的。
2. 分区说明表：PDT

分区号	大小	内存始址	状态
1	12K	20K	已分配
2	32K	32K	已分配
3	64K	64K	已分配
4	128K	128K	未分配

3. 分配策略:

在调度作业时，存储器管理根据所需量在分区说明表中找出一个足够大的分区分配它，然后用重定位装配程序装入它。如果找不到合适得分区，则通知作业调度模块，另外选择一个作业

分配：依次查找分区说明表中的信息，将分区大小满足作业请求容量，并且使用状态为空闲的第一个分区分配给该作业。同时将该分区的状态改为已使用。如图。

回收：作业运行完毕后，将释放的分区收回，设分区使用状态为空闲。

4. 优、缺点:

优点：能实现多个作业共享内存，保证多道运行、数据结构简单，分配回收算法容易实现等；

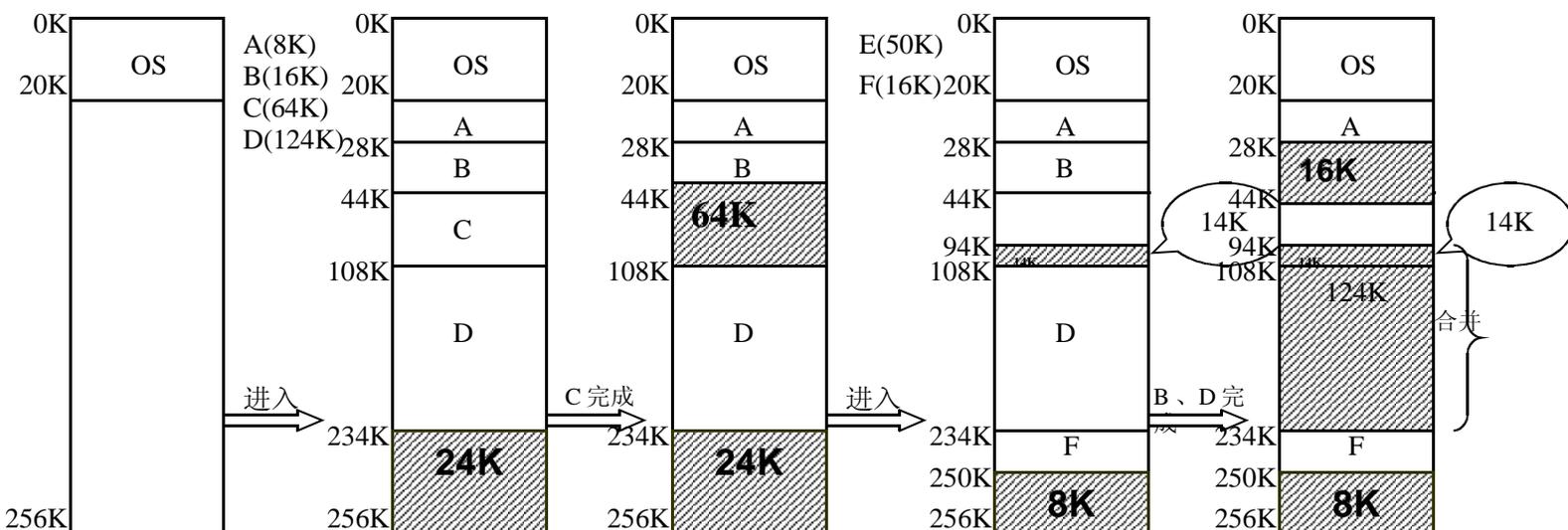
缺点：内存利用不充分，小作业占用大分区，造成内碎片现象；作业的大小受到分区大小严格限制；

注：

内碎片：在固定分区方式中，一个分区分配给作业后，分区中未使用的空间区称为内碎片，又称内零头。

5.3.2 可变分区管理

1. 基本思想（量体裁衣）：根据作业大小动态地划分分区，使分区大小正好适应作业的需要。克服了内碎片。



2. 与固定分区的区别:

- 1) 系统运行过程中，作业装入时建立
- 2) 个分区的大小不确定
- 3) 内存中分区的数目可以（一般）不定

3. 主存占有表和空闲说明表（链）

分区说明表：记录占用内存分区的情况；

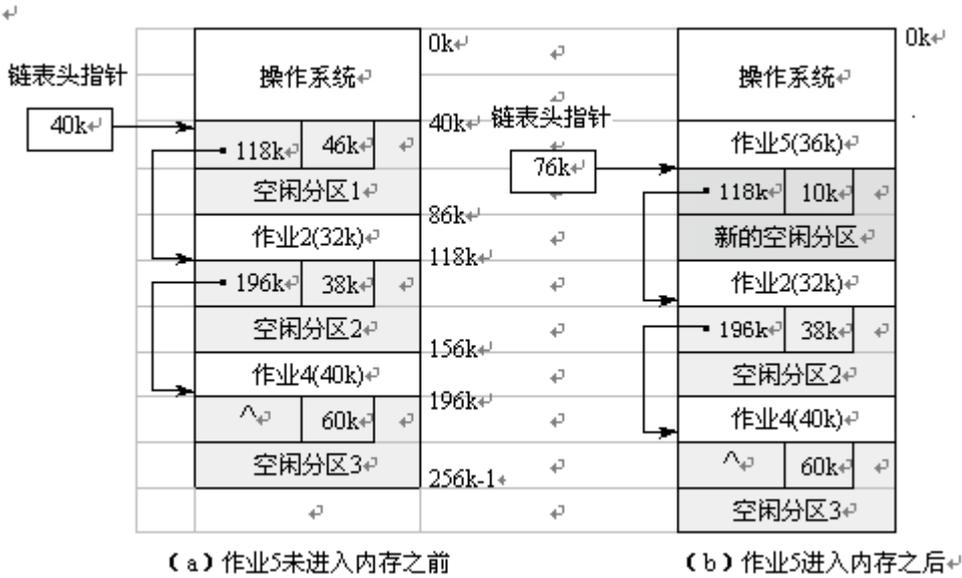
空闲说明表（空闲分区链）：将内存中空闲分区单独构成一个空闲分区表和空闲分区链；

4. 分配策略：主要问题：分配、回收

1) 首次适应算法（FF）：按始地址升序排列。从链首开始分配。

分配：从空闲分区表（空闲分区链）首部开始顺序查找，直到找到第一个能满足其大小要求的空闲地址为止。

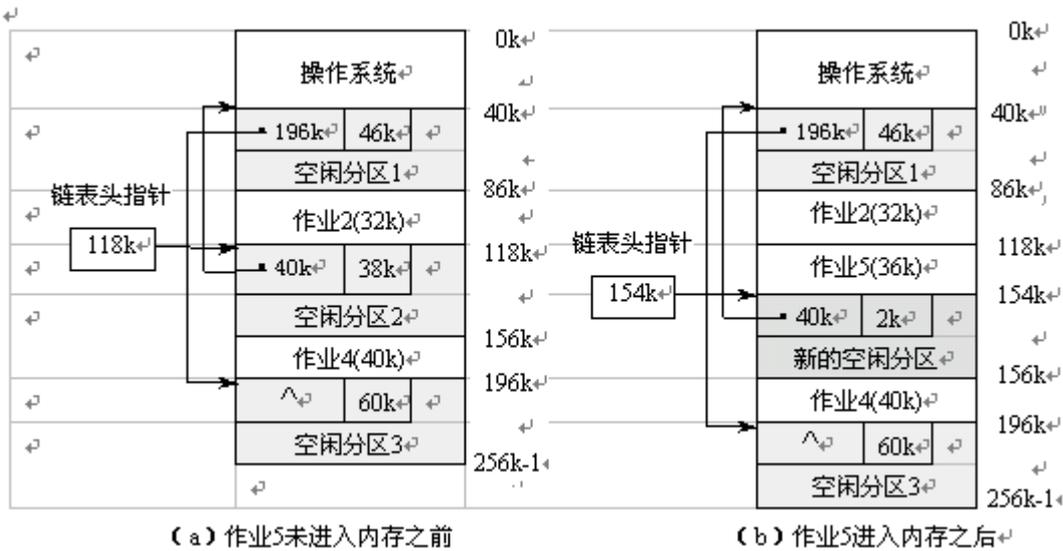
优点：算法简单，查找速度快，大作业易满足要求。



2) 最佳适应算法（BF）：按分区大小递增的顺序排列。

分配：从空闲分区表（链）首开始查找，直到找到第一个能满足其大小要求的空闲地址为止。

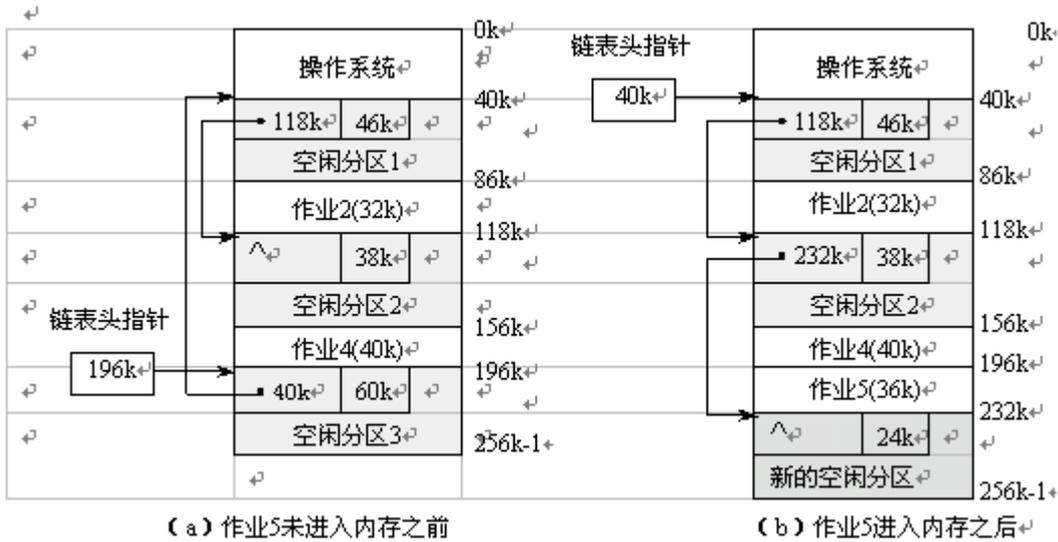
注意：此算法看起来最佳，其实不然。



3) 最坏适应算法（WF）按分区大小递减的顺序排列。

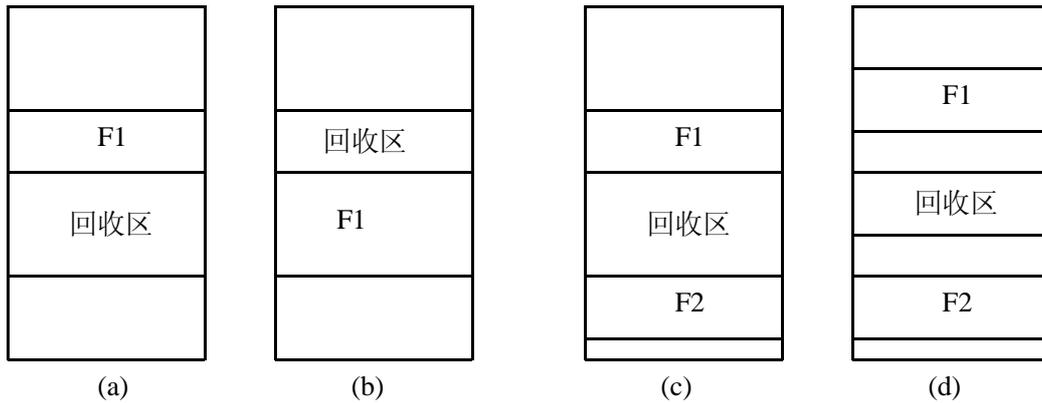
分配：总是把空闲链中的第一个分区，即将最大的空闲分区分配给作业；

缺点：大作业难以满足要求。



5. 分区回收

当进程运行完毕释放内存时，系统根据释放区的首址，从空闲链中找到相应的插入点，此时可能出现以下四种情况。



处理方法：

- (a) 不再为回收区分配新表项，而只需修改 F1 的大小。
- (b) 用回收区的首址作为新空闲区的首址，大小为两者之和。
- (c) 将三个分区合并，区 F1 的首址，取消 F2 的表项
- (d) 新建一个表项。

注：

外碎片：主存中的一个空闲区域在分配给作业后，一般总是剩余一个更小的空闲区。当这样的小分区不能再装入一个作业时，即不能被利用时，它们也成为主存碎片，这样的分区在作用使用的分区之外，所以称为外碎片。

内存紧凑技术（拼接技术）：通过移动各个作业分区的存储位置，把多个外碎片拼接成一个较大的空闲区，从而可以用于存放另一个新的作业。

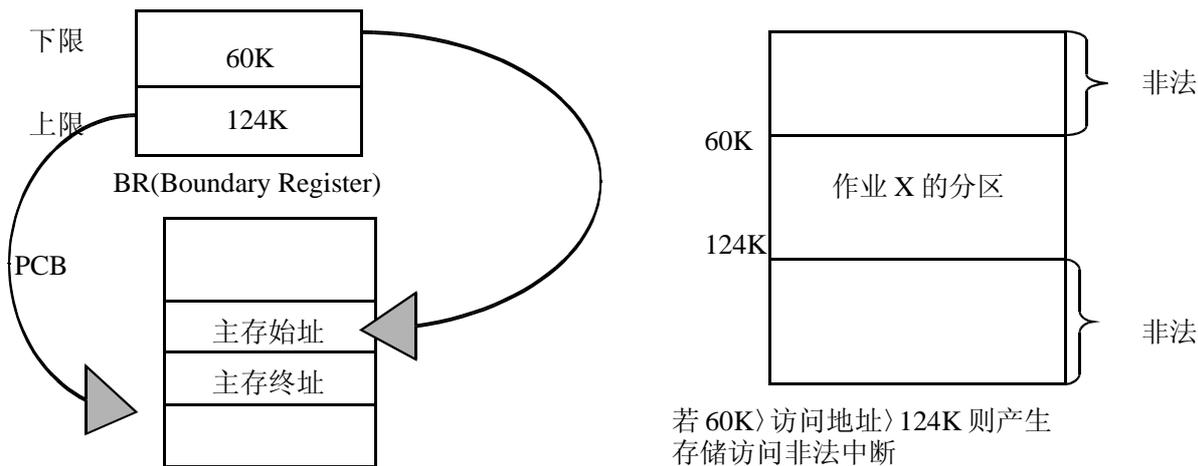
采用动态重定位技术，一个作业在内存中移动后，只要改变重定位寄存器的内容即可。

5.3.3 分区管理存储保护

存储保护是为了防止一个作业有意或无意地破坏操作系统或其他作业。

1. 界限寄存器保护（上下界保护法）（上、下限）（PCB）

如图示：把作业 X 分配在 60K 到 124K 的一个分区内，当调度到该作业在 CPU 上执行时，由 OS 把这对寄存器分别设置成 60K 和 124K。在作业运行过程中形成的每一个访问地址，与这两个寄存器的值比较，进行地址有效性检验，发现非法访问时产生中断。



2. 基址，限长寄存器法（基址、限长）

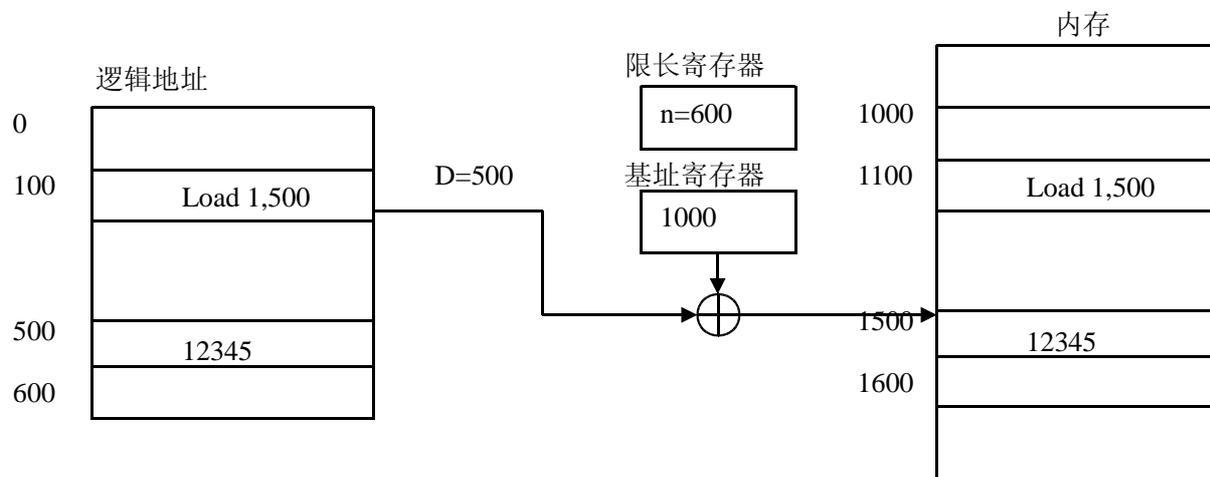
限长寄存器中的值代表可以使用的最大地址位移量（即相对地址或逻辑地址），基址寄存器用以存放运行作业的起始地址。

Load 1,500——取逻辑地址 D=500 号单元的数到 1 号寄存器中。

在 CPU 执行该指令时，由硬件对所访问的逻辑地址 D 进行检查：

若 $D > n$ (n 为限长值)，则说明地址越界，即所要访问的内存地址超出本作业所占用的存储空间，这将产生保护中断，控制转给 OS 去进行出错处理。

若 $D < n$ ，则说明地址合法，做重定位工作。



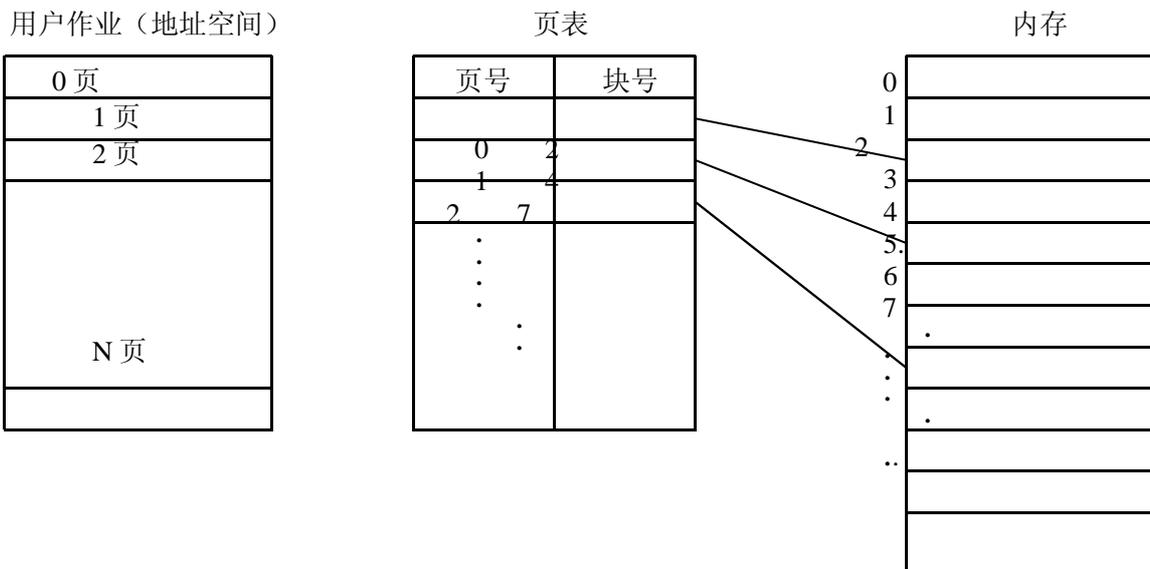
3. 保护键法（锁、钥匙）（状态子 PSW）

(详细参见参考文献1, P104)

对每个分区分配一个唯一的保护健，它是一个 N 位的二进制代码，相当于该分区的一把锁，而在程序状态寄存器 PSW 种，设置有保护健字段，它相当于一把钥匙，在执行存储访问指令时，先要检查被访问的单元所在分区的锁和钥匙是否相符。若符合则允许访问，否则不可以。因此，当某一分区分配给作业时，同时分配给他一个唯一的键码，当该作业执行时，PSW 中的保护字段置相应的键码。这样，它在访问本作业的存储区时，由于锁钥匙相配可以顺利地进行。如果该作业在运行中不正确的访问时，则会因为钥匙不匹配而产生保护中断，并把控制转给 OS 进行出错处理。为了保证 OS 能够访问内存的所有单元，可以规定一种”万能钥匙”，如当 PSW 中保护健字段全为 0 时，则不进行钥锁匹配检查。这样，就可以访问所有内存空间。

5. 4 分页存储管理

5.4.1 纯分页存储管理



a. 页 (页面) 和物理块 (帧)

分页存储管理，把进程的地址空间划分为大小相等的片段，成为页或页面。相应地，内存空间也分成与页面相同大小的若干块，成为物理快或帧。

b. 页面大小

页面大小通常在 512k~4k，大小必须适中。Why? (四个因素刘乃琦 P84)

因为：若页面太小，一方面课时内存碎片小，减少了内存碎片的总空间，有利于提高内存的利用率；但另一方面，也会使每个进程要求较多的页面，从而引起页表过长，占用大量内存；此外，还会降低页面换进换出的效率。若选择较大的页面虽然可减少页表的长度，提高页面换进换出的效率，但却又会使页面碎片增大。因此，页面大小应选择适中。

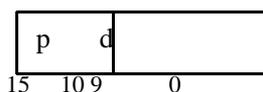
页面碎片：分页方式中也会出现内存碎片，它们都分布在每个作业的最后页面内部，故称为页面碎片。

c. 逻辑地址结构

页面大小为 $2^{10}=1K$ 举例：逻辑地址 5000

最多可以有 $2^6=64$ 页 可转换为 $4*1024+904$

地址结构 { 页号 p
位移量 w (即，页内相对地址)



分页后，作业相对地址转变为页面地址的公式：

$$p=[A/L]\text{取整}$$

$$d=A\%L$$

其中：A 作业地址空间的相对地址；L 为页面大小。

5.4.2 页表

1. 页表——os 为每个进程建立一张页面与物理块号的对照表，以便实现地址映射；

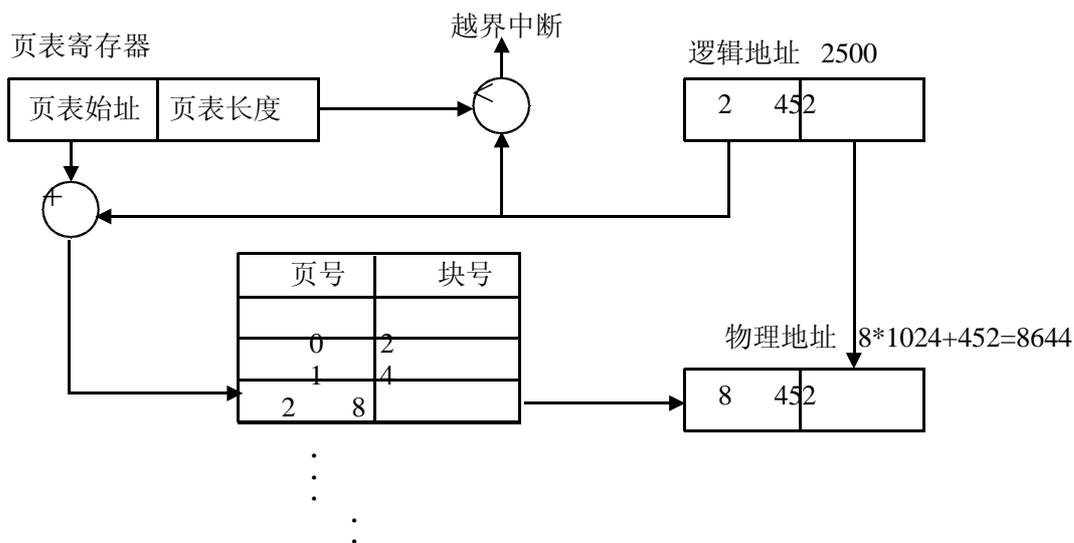
*页表技术实质上是动态重定位技术的一种延伸。

2. 页表的组织

页表在内存中占有一块固定的存储区，页表的大小有进程或作业的长度决定。如，对于一个每页长为 1k，大小为 20k 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 20 个存储单元。

3. 地址变换机构、地址映射。（两个地址寄存器 PAR、PLR）

举例：设一个 3 页长的进程具有页号 0,1,2，其对应的块号为 2, 3, 8。设每个页面长度为 1K，指令 Load



1,2500 的虚拟地址为 100，是求出该条指令所对应的物理地址。并分析其执行过程。

解：PMT 如图所示

页号	块号
0	2
1	3
2	8

由虚地址 100 可知，指令 Load 1,2500 在第 0 页的第 100 单元之中，由于第 0 页对应第 2 块，因此，该指令在内存中的地址为 2048+100=2148。

当 CPU 执行到第 2148 单元的指令时，要从有效地址 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，地址变换机构首先将 2500 转换为页号和页内相对地址的形式：即 P=2,W=452。

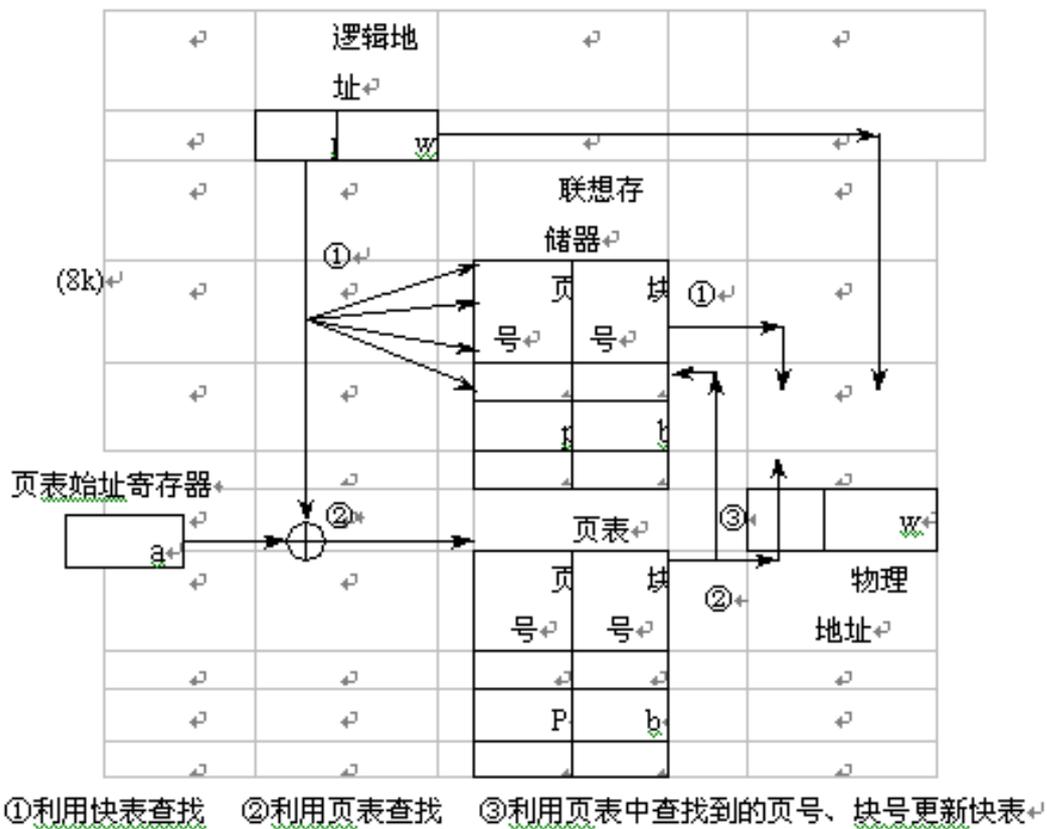
由页表，可知第 2 页所对应的块号为 8，最后，将块号与页内相对地址 W=452 相连，得到待访问的物理内存地址为 8644。

4. 地址越界保护

5.4.3 快表

从上面介绍的地址变换过程可以看出：如果把页表全部放在内存，那么存取一个数据时，至少要访问二次内存。一次是访问页表，形成实际内存地址；另一次是根据形成的内存地址存取数据。显然，这比通常执行指令的速度要慢得多，使计算机的运行速度几乎降低一半。

应用联想存储器和页表相结合的方式，可有效地提高系统动态地址转换的速度，是一种行之有效的方法。这个联想存储器称为快表。



采用快表和页表相结合的分页地址变换过程示意图

5. 5 存储扩充技术

5.5.1 覆盖技术

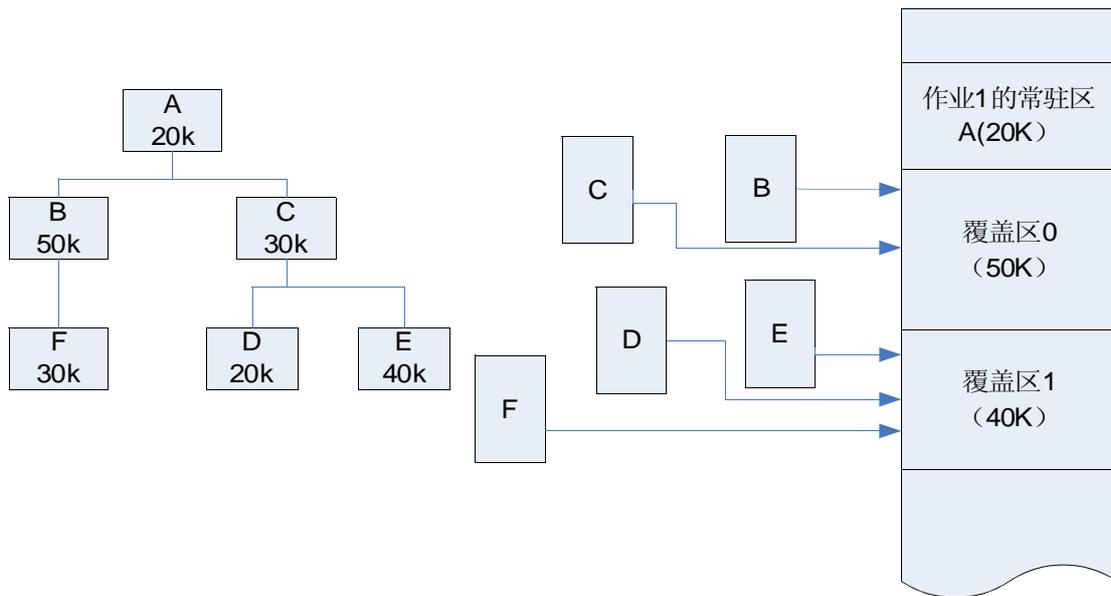
覆盖是指一个作业的若干程序段，或几个作业的某些部分共享某一个存储空间。

覆盖技术的实现是把程序化分为若干个功能上相对独立的程序段，按照其自身的逻辑结构使那些不会同时执行的程序段共享同一快内存区域。程序段先保存在磁盘中，当有关程序段的前一部分执行结束后把后续程序调入内存，覆盖前面的程序段。

覆盖不需要任何来自操作系统的特殊支持，可以完全由用户实现，即覆盖技术是用户程序自己附加的控制。覆盖技术要求程序员提供一个清楚的覆盖结构，即程序员要把一个程序化分成不同的程序段，并规定好他们的执行和覆盖的顺序。操作系统则根据程序员提供的覆盖结构，完成程序段之间的覆盖。

覆盖可以有编译程序提供支持：被覆盖的块是由程序员或编译程序预先（在执行前）确定的。总之，覆盖可以从用户级彻底解决内存小装不下程序的问题。

例：



覆盖技术要求用户清楚地了解程序的结构，并制定各程序段调入内存的先后次序，以及内存中可以覆盖掉的程序段的位置等。对用户是不透明的。因此通常用于系统程序的内存管理上，因为系统软件设计者容易了解系统程序的覆盖结构。

例如：把磁盘操作系统分为两部分，一部分是操作系统中经常用到的基本部分，它们常驻内存且占用固定区域，另一部分是不太经常使用的部分，他们存放在磁盘中，当调用它们时才被调入内存覆盖区。

5.5.2 交换技术

定义 对换——指把内存中占时不能运行的进程或暂时不用的程序和数据换出到外存上，以腾出足够的内存空间，把已具备运行条件的进程或进程所需要的程序和数据，换入内存。

如果对换是以整个进程为单位，称之为“整体对换”或“进程对换”。这种对换被广泛地应用于分时系统；如果对换是以“页”或“段”为单位，则分别称之为“页面对换”或“分段对换”，又称为“部分对换”。这种对换方法是实现请求分页及请求分段式虚拟存储器的基础。

与覆盖技术相比，交换技术的特点是交换过程对用户是透明的，但需要更多的硬件支持。

5.5.3 虚拟存储技术

- 解决内存不足：
- a. 加内存条。
 - b. 覆盖技术、整体对换
 - c. 虚拟技术

虚拟存储器的基本概念

1) 常规内存管理方式的特征

- a. 一次性
- b. 驻留性

一次性和驻留性是许多不用或暂时不用的程序（数据）占据了大量的内存空间，将使一些需要运行的作业无法装入运行。现在的问题是：一次性、驻留性是否是存储器管理所必需的。

2) 局部性原理

早在 1968 年，Denning . p 就指出过，程序在执行始将呈现出局部性规律，即在一较段时间内，程序的执行仅局限于某个部分，相应地，它所访问的存储空间页局限于某个区域。

局限性（时间局限性、空间局限性）**局部性原理（principle of locality）：在一个作业运行的某一段时间，它所访问的地址空间往往只集中在某几页，而不是整个程序的所有部分都具有平均的访问概率，这种现象称为局部性原理。**

3) 虚拟存储器的引入

当一个作业的地址空间很大时，不能别全部装入内存，但基于局部性原理，允许只将当前要运行的那部分程序和数据先装入内存便启动运行，其余部分仍驻留在外存上，在需要时，在通过调入功能和置换功能将其调入内存。

Def: 虚拟存储器是只具有请求调入功能和置换功能，能从逻辑上对内存容量进行扩充的一种存储器系统。

在多道系统中使用虚拟存储技术，不管物理内存空间多大，系统都可以为每个用户的作业提供很大的独立的虚拟空间。例如 linux 操作系统中可以为每个用户作业提供高达 4G 的虚拟空间。

虚拟存储空间中的地址称为虚拟地址。虚拟地址空间大小由虚拟地址的长度决定。由于虚拟空间比物理内存的容量大的多，所以系统提供的虚拟地址的长度大于主存的绝对地址的长度，例如在 80x86 中绝对地址是 32 位的，而虚拟地址是 46 的。

4) 虚拟存储器的特征

- a. 多次性
- b. 对换性
- c. 离散性：多次性和对换性必须建立在离散分配的基础上，故虚拟存储管理必须建立在离散分配系统基础上。

5.5.4 请求式分页存储管理

1. 数据结构——页表（PMT）

在请求分页系统中的页表，时在分页系统的页表的基础上增加如下几项：

页号	物理块号	状态位	访问字段	修改位	外存地址
----	------	-----	------	-----	------

状态位: 用于指出该页是否已调入内存

访问字段: 用于记录本页在一段时间内被访问的次数, 提供给置换机构参考

修改位: 表示该页在调入内存后是否被修改过, 由于内存中的每一页都在外存上保留一份副本, 因此, 若未被修改, 在置换该页时就不需要将该页写回到外存上; 否则, 必须将该页重写到外存上, 以保证外存中所保留的始终是最新的副本。

外存始址: 指出该页在外存上的地址, 供调入该页时使用。

2. 缺页中断和地址映射过程 (参见书)

3. 页面淘汰算法

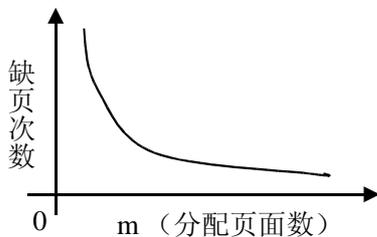
- 优化 or 最佳淘汰算法 (OPT): 淘汰将永不再使用的页面, 或最长时间内不再访问的页面。(无法实现, 仅作为标准)
- 最近最就不用算法 (LRU: Least Recently Used Replacement): 当需要置换一页面时, 选择在最近一段时间内最久不用地页面予以淘汰。
- 近似 LRU 算法——NRU (not Recently Used)
- FIFO 算法: 淘汰最先进入内存的页面, 即在内存中驻留时间最长的页面。

缺点:

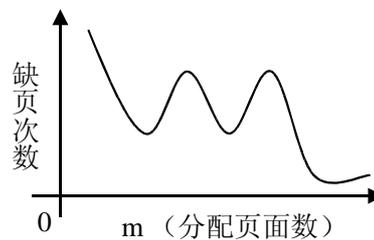
a. 内存利用率不高——由于该算法基于 CPU 按线性顺序访问的地址空间的, 而许多时候, CPU 不是按线性顺序访问地址空间, 如执行循环时, 那些在内存中停留时间长最长的页往往是经常被访问的页。

b. 会出现异常现象——Belady 现象: 一般来说, 内存帧越多, 一个作业发生缺页的次数就越少。(如果给一个进程分配了它所需要的全部页面, 则不会发生缺页现象) 但 Belady 举出了反例, 例如使用 FIFO 算法时, 在未给进程或作业分配足它所需要的页面时, 有时会发生分配的页面数越多, 缺页次数反而增加的奇怪现象。这种现象称为 Belady 现象。P83

a) 正常情况



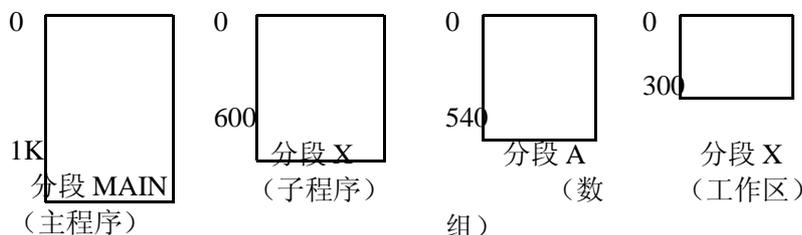
b) Belady 现象



5. 6 分段存储管理

1. 基本思想:

从固定分区到可变分区, 进而又发展到分页系统的原因, 主要都是为了提高内存利用率, 然而, 分段存储管理方式的引入, 则主要是为了满足以下的一系列要求。



(1) 地址结构

分段系统，作业的地址空间（二维）由若干逻辑分段组成，每个段是一组逻辑意义完整的信息集合。每段都有自己的名字，且每段都是首地址为 0 的连续的一维地址空间。

所以，整个作业空间是二维的，具有如下地址结构。



一个作业最多可以有 2^8 个段，每段最长为 2^{16}

(2) 段表

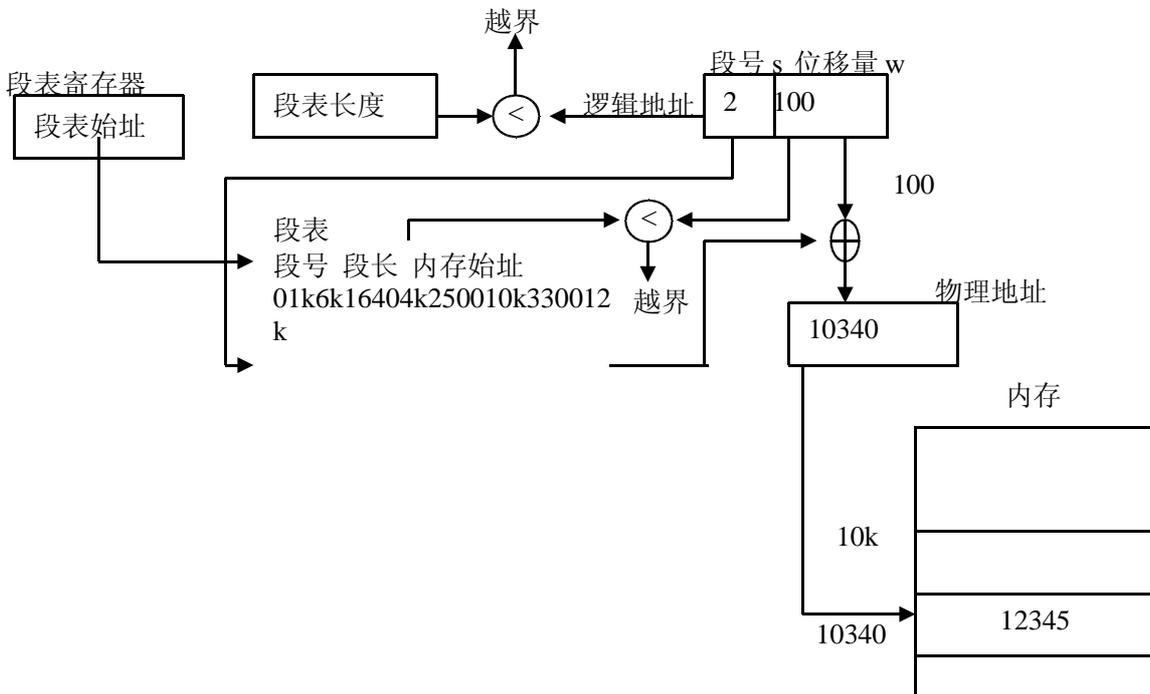
分段式存储管理以段为单位进行内存分配，每段分配一个连续的内存区，各段之间的内存不一定连续，且各个内存区也不等长。内存的分配和释放随需要动态进行。在将一个进程的各个段离散地放入内存和不同的物理区中后，为能使程序正确运行，即能从物理内存中找出每个逻辑段对应的位置，需要在系统中为每个进程建立一张段映射表，简称“段表” (SMT: segment mapping table)。每个段在段表中占有一个表项，其中记录了该段在内存中的起始地址、段的长度。段表可实现从逻辑段到物理内存的映射。

SMT: [段号][装入标志位][段长][内存始址]

(3) 地址变换机构

为了实现从进程的逻辑地址到物理地址的变换功能，在系统中设置了段表寄存器，用于存放段表的始址和段表的长度。在进行地址变换时，系统将段号与段表长度进行比较，若段号太大表示访问越界，便产生越界中断信号；若未越界，则根据段表始址和该段的段号，计算出该段对应的段表项的位置，从而读出该段在内存中的起始地址，然后再检查段内地址是否超过该段的段长，若超过，同样发出越界中断信号；若未越界，则将该段内存起始地址与段内地址相加，得到要访问的内存物理地址。

与页式管理相同，段式管理时的地址变换过程必须经过二次以上的内存访问。即首先访问段表以计



算得到访问指令或数据的物理地址，然后才是对物理地址进行取数据或数据操作。为了提高访问速度，也可引入快表。

(4) 存储共享

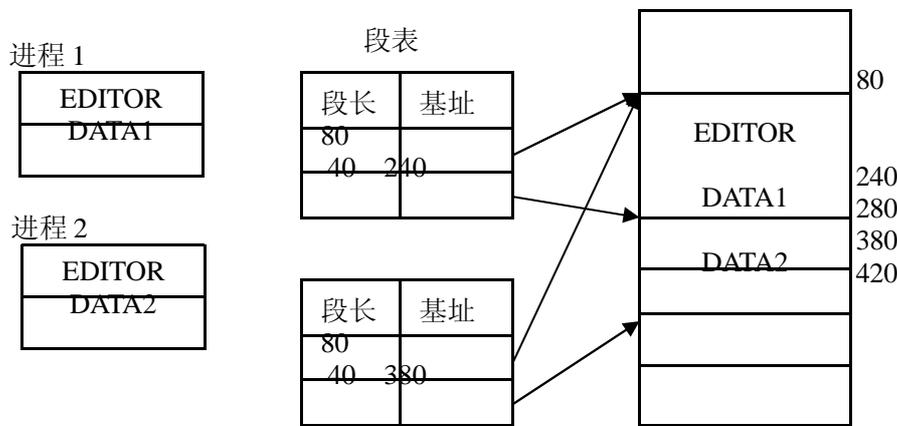
只需在每个进程的段表中，由相应表项来指向该共享段在内存中的物理区即可。

如下图：

(5) 分页和分段的主要区别

- 页是信息的物理单位，分页仅仅是由于系统管理的需要，而不是用户的需要；而段是信息的逻辑单位，它含有一组具有相对完整意义的信息，是处于用户的需要。
- 页的大小固定且由系统确定，把逻辑地址分为页号和页内地址两部分功能，由机器硬件实现；而段的长度却不固定，由用户在变成是确定，或由编译程序在对源程序进行编译时，根据信息的性质来划分。
- 分页的作业地址空间是一维的，即单一的线性地址空间；而分段的作业地址空间则是二维的。

5.7 段页式存储管理



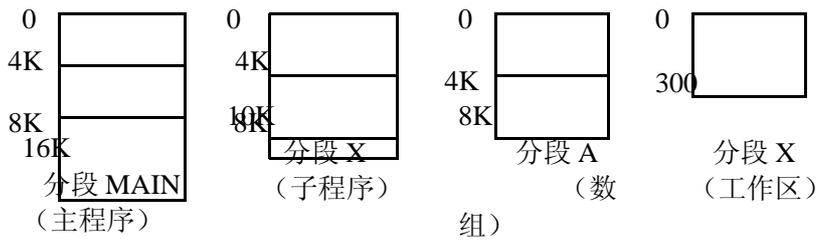
5.7.1 段页式存储管理原理

分页系统能有效地提高内存利用率，而分段系统则能很好地满足用户需要，如果对两种存储管理方式“各取所长”，可结合成一种新的存储管理方式，它具有分段系统的优点，又能象分页系统那样很好地解决“碎片”问题。这个系统别称为“段页式系统”。

1. 基本原理

段页式系统的基本原理是分段和分页原理的结合。先分段后分页，每段具有段名。

例如：该作业由三段，页面大小为 4K



2. 地址结构

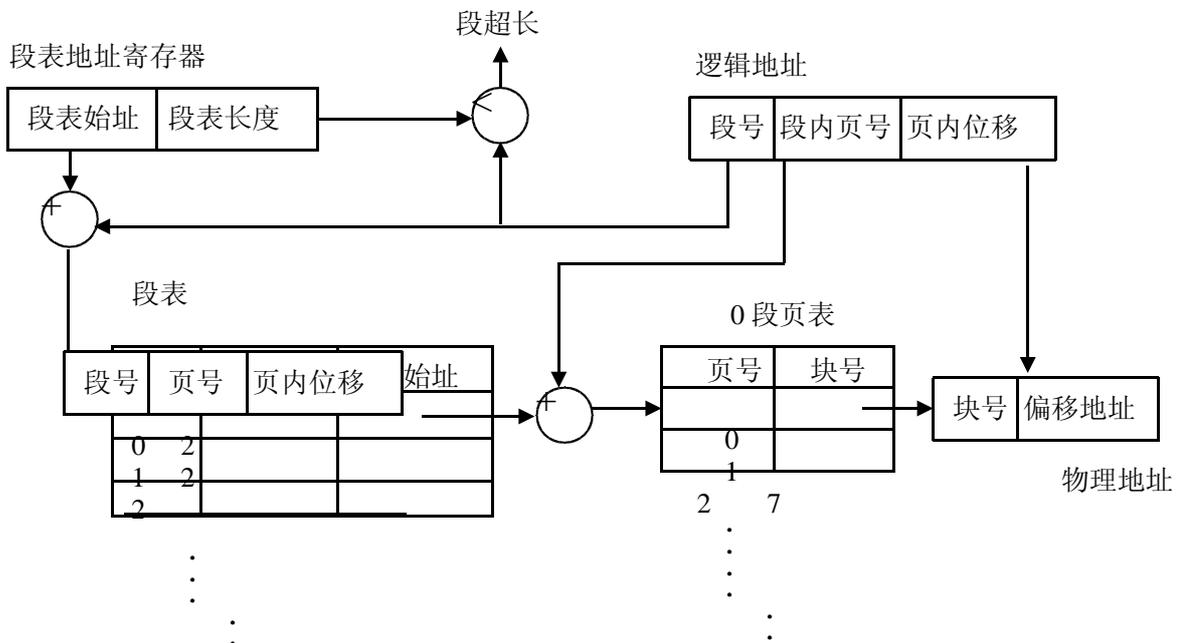
段页式系统中，其地址结构由段名、段内页号和页内地址三部分组成。

3. 为实现地址映射，必须配置段表和页表，段表的内容略有变化，它不是内存始址和段长，而是页表始址和页表长度。



5.7.2 地址转换

在段页式系统中，需配置 1 段表寄存器，其中存放段表始址和段表长度，进行地址变换时，首先利用段号 S ，将之与段表长度进行比较，若未越界，便利用段表始址和段号求得该段表项的位置，并从中找到页表始址，并利用逻辑地址中的页号 P 获得对应页表项的位置，从而取出该页的物理块号 b ，由块号 b 和页内地址构成物理地址。



在段页式系统中，为了获得一条指令或数据，需要三次访问内存。第一次，从内存中取得页表地址；第二次，从内存中取的物理块号形成物理地址；第三次才能取得所需的指令和数据。为了提高速度，必须引入超高速缓冲 Cache。

第 6 章 Linux 存储管理

本章前三节作为学习 linux 存储管理的基础知识，介绍了 Intel 80x86 系列机器实现虚拟存储的分段机制；然后结合 80x86 介绍 linux 的分段分页结构、进程地址空间管理、物理空间管理以及存储空间分配。

6. 1 80x86 的分段机制

Linux 最早在 intel80386 机器上开发，当前也主要运行在 intel 80386、80486 和 pentium 系列机器上。按照管理把该机器系列统称 **80x86**，也称 **i386**。

80x86 机器中用于控制和管理内存的硬件机制称为**存储器管理单元 MMU (memory manage unit)**。

在 MMU 控制下，80x86 具有**两种存储管理模式**：

- 实地址模式：cpu 只能寻址 1MB 空间，MS-DOS 操作系统就工作在实地址模式下；
- 受保护的虚地址模式（保护模式）：80x86 提供了虚拟存储的硬件机制，它是 OS 实现多任务存储管理以及提供存储保护的硬件基础。Linux 就是在该模式下实现其各种功能的。

6.1.180x86 的虚拟存储空间

1. 物理地址和虚拟地址

物理地址：通常出现在地址总线上的地址称为物理地址。内存空间有物理地址确定，所以内存又称为物理地址空间。内存的最大容量由物理地址长度决定。80x86 机器的地址总线为 32 位，故有它确定的物理地址空间的范围可达 $2^{32}B$ ，即 4GB。

虚拟地址：用户作业的地址空间是由逻辑地址确定的，逻辑地址就是机器指令中使用的地址，故逻辑地址空间的最大容量由机器地址长度决定。80x86 指令地址字长度 48 位，其中 46 位用于寻址，因此 80x86 的逻辑地址地址空间最大可达 64T。故 80x86 提供给程序人员使用的逻辑地址空间远远大于实际的物理地址空间，是一个虚拟的存储空间。所以逻辑地址就是虚拟地址。

为了支持多道进行并发执行以及实现存储保护，80x86 存储管理机制把 64T 的逻辑地址空间分成性质不同的两部分：

- 全局地址空间：所有进程共享的空间，通常存放 os 的程序和数据。
- 局部地址空间：由系统分配给各个进程使用，用于存放进程各自的程序和数据。

2. 地址转换方式

80x86 在保护模式下提供了两种地址转换方式：

- 分段机制
- **分段+分页的两级地址转换方式。（本书只介绍这种转换方式）。**

3. 分段分页地址转换

在系统运行时，逻辑地址必须转换成物理地址才能访问物理内存。80x86 中分段分页的地址转换机制中需要两级地址转换：

- **第一级：由分段机制完成逻辑地址向线性地址的转换。**分段机制把逻辑地址空间分成若干个相互

独立的地址空间，它称为线性地址空间。线性地址空间的地址称为**线性地址**。每个线性地址空间从0开始编址，80x86中线性地址是32位的，因此每个线性地址空间最大4G。在多道系统中，**每个线性地址可以供一个用户进程使用，这就保证了每个进程都有自己独立的虚拟存储空间。**故：

逻辑地址：面向用户，是用户在程序设计时使用的地址空间，也就是用户程序中指令访问的地址空间；

线性地址：面向进程，使进程使用的地址空间，对用户透明。

➤ **第二级：由分页机制完成线性地址向物理地址的转换**

说明：

1) 在80x86中，虚拟地址空间的全局和局部地址空间都被划分成不同段。64T的虚拟地址空间最多可以分16K个段。每个段的最大长度可达4G；

14 位	32 位
------	------

2) 全局地址空间和局部地址空间各自最多可以有8K段，分别称为全局段和局部段。

3) 由于虚拟存储空间不是实际存在的，所以其中的段的数量，段的大小是不确定的，是根据需要随时建立的。当然不能超过上面提到的最大限制。

6.1.2 段描述符表

80x86的段描述符表就是前面原理部分提到的段表。段描述符就是段表表项，**每个段描述符的长度为8B**。80x86提供两种不同的段描述符：

- 全局描述符表 GDT (Global Descriptor Table)：描述全局段，故系统中只有一个全局描述符表
- 局部描述符表 LDT (Local Descriptor Table)：描述各个进程的各个局部段。故系统中有多少进程就有多少 LDT。

1. 全局描述符表 GDT

1) 80x86中，全局描述符表 GDT 中一般包括3种不同种类的描述符：

- 系统内核代码段和数据段的描述符
- 进程状态段 TSS (Task State Segment) 的描述符。系统中每个进程都有一个 TSS 段，用于保存该进程的上下文。所有进程的 TSS 段描述符都集中的保存在 GDT 中。
- LDT 描述符：对各个进程的局部描述符表 LDT 进行描述的描述符，记录各个进程的局部描述符 LDT 在线性地址空间的位置和长度。每个进程 LDT 描述符在 GDT 中的位置，由系统记录在该进程 TSS 段的一个 16 位的位域中，该位域的值称为 LDT 选择符。

2) 全局描述符表寄存器 (GDTR)：记录 GDT 在线性地址空间中的位置。

32 位	表基址	16 位 表限
------	-----	---------

其中：

表基址：指出 GDT 的起始地址；

表限：表限的值加 1 位 GDT 的长度。

说明：

16 位的表限确定了 GDT 的最大地址空间为 64K，由于每个描述符长度为 8B，因此 GDT 最多可以拥有 8K 个描述符。每个描述符对应一个段，因此系统中最多可以有 8K 个全局段。

2. 局部描述符表 LDT：描述进程各个局部段的。

3. 定位 LDT

因为单处理机中某个时刻只能有一个进程处于运行状态，所以在某个时刻只有当前进程的 LDT 处于活动（使用）状态。系统需要能迅速定位活动的 LDT。

80x86 通过局部描述表寄存器 LDTR 和相应的 LDT 高速缓存来实现定位 LDT。

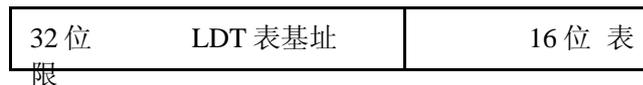
➤ **局部描述表寄存器 LDTR：**是一个 16 位寄存器，它的作用是指出当前进程的 LDT 描述符在 GDT 中的位置，也就是前面提到的 LDT 选择符。在进程切换时，从进程的 TSS 中取出 LDT 选择符，并将它放入 LDTR 中；

在地址映射时，首先从 LDTR 中得到 LDT 选择符；然后 GDT 中根据 LDT 选择符找到 LDT 描述符（即得到 LDT 的位置等信息），接下来才能找到 LDT（即段表）。显然这样是会严重影响程序运行速度的。故引入了 LDT 高速缓存。

➤ **LDT 高速缓存：**保存从 GDT 中找到的 LDT 描述符。

引入 LDT 高速缓存后，第一次仍需要完成上面提到的三个步骤，但是第一次完成时，将从 GDT 中找到的该进程的 LDT 描述符保存在了 LDT 高速缓存中。这样以后地址映射时，可以直接从高速缓存中得到 LDT 的位置等信息。

LDT 高速缓存：



说明：

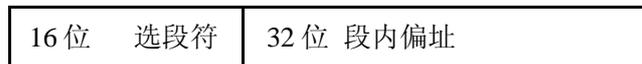
16 位的表限确定了 LDT 的最大地址空间为 64K，由于每个描述符长度为 8B，因此 LDT 最多可以拥有 8K 个描述符。每个描述符对应一个段，因此系统中最多可以有 8K 个局部段。

图 6.6 局部描述符表 LDT 的定位机制。

6.1.3 逻辑地址向线形地址的转换

1. 基本过程（参照原理部分）

在 80x86 中逻辑地址的格式如下所示：



在 80x86 中，程序执行中读取每一条指令时都需要访问代码段。这时系统把逻辑地址中的选段符装入寄存器 CS 中，把段内偏址装入指令计数器 EIP 中，然后经过硬件分段机制把逻辑地址转成指向代码段的 32 位线性地址。

程序中执行读写数据的指令时，系统把指令中逻辑地址的选段符装入 DS 寄存器中，把段内偏址装入某个通用寄存器中，然后经过硬件分段机制把逻辑地址转成指向数据段的 32 位线性地址。

2. 段描述符寄存器（参照原理部分，快表）

选段寄存器：上述中设计到的寄存器，因为包含了选择某个段的索引值，故又称其为选段寄存器。

80x86 中有 6 个段寄存器：CS、DS、SS、ES、FS 和 GS；

段描述符寄存器：根据程序的局部性原理，为了提高地址映射速度，减少访问内存的次数。80x86 分段机制在高速缓存中，为每个选段寄存器都配置了一个对应的段描述符寄存器，用于保存当前访问的那些段的段描述符。

图 6.7 选段符寄存器和段描述符寄存器

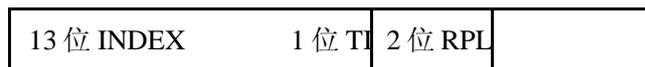
6. 2 选段符与段描述符

6.2.1 选段符

选段符的作用：确定所选的段描述符在段描述符表中的位置。

选段符由 3 个部分组成，如下所示：

其中：



- RPL (Reques Privilege Level)：请求特权级，即访问者的特权级，分为 0、1、2、3 四级，0 级最高，3 级最低。Linux 中仅使用其中两个级别：0 级为内核级，3 为用户级。
- TI (Table Indictor)：描述符表的种类。0 为访问全局描述符表 GDT，1 为访问局部描述符表 LDT。
- INDEX：索引值。指出段描述符在描述符表中的偏移量。INDEX 13 位，则可以最大可描述 8K 个段。

如何确定描述符在描述符表中的偏移地址？

INDEX 是描述索引表的索引值，实际上是描述符在描述符表中的序号。每个描述符的长度为 8B，所以描述符相对于表基地址的偏移地址实际是：INDEX 乘以 8。而实际该值并不需要计算，只要将 INDEX 的值左移 3 位，低地址部分 3 位补 0，形成的 16 位地址就是偏移地址。

6.2.2 段描述符

在 80x86 线性地址空间中，所有的段根据它们所存放的内容可以分为两类：

- 常规段：存放代码、数据、堆栈的段，无论是用户的还是系统的。
- 系统段：存放系统管理用的各种数据结构，如进程状态段 TSS、中断向量表、系统调用表等。

相应地，段描述符也有两种：常规段描述符、系统段描述符。**本书中只介绍常规段描述符。**

64 位常规段描述符主要分为以下四个部分：



1. 段基址：32 位（分布在段描述符 16~39 位和 56~63 位）指出段在线性地址空间的起始地址。
2. 段限：20 位（分布在段描述符 0~15 位和 48~51 位）段限加 1 是段的长度。
3. 访问属性：8 位（40~47 位）定义了段的类型、操作属性及保护特性。
4. 辅助特性：4 位（52~55 位）定义了段的其他属性。

下面介绍访问属性

7	6	5	4	3	2	1	0
P	DPL	S	E	C/E	R/W	A	

P (present)：存在位。

DPL (Descriptor privilege Level)：段的访问特权级。它是段本身的访问特权分为 0、1、2、3 四级，0 级最高，3 级最低。Linux 中仅使用其中两个级别：0 级为内核级，3 为用户级。

S (Segment)：段种类。0：系统段，1：常规段。

E (Executive)：执行位。表示段的类型：0 表示数据段；1 表示代码段。

C/ED：相容性/扩展位。

代码段 (S=1, E=1) 中，C=1 表示在满足一定条件下段可以执行。C=0 表示段不能执行；

数据段 (S=1, E=0) 中，对堆栈段而言：ED =1 表示向上扩展堆栈。ED =0 表示向下扩展堆栈。

R/W：读写位。

代码段 (S=1, E=1) 中，R/W =1 表示可读可执行。R/W =0 表示不可读但可执行；

数据段 (S=1, E=0) 中，R/W =1 表示可读可写。R/W =0 表示可读禁止写。

A：访问位。A=0 表示该段尚未被访问；A=1 表示该段已被访问过。

下面介绍辅助特性：

G (granularity)：粒度，指段的单位长度，G=1 表示段长度以页面（4K）为单位，在常规段描述符中 G=1；G=0 表示段长度以字节为单位，在系统段描述符中 G=0。

D：操作长度，只用于代码段描述符中，D=1 位 32 位代码段，D=0 为 16 位代码段。

R：系统保留(值为 0)

U：可以由 os 系统程序员自行定义使用，linux 未用。

6.2.3 分段机制的存储保护

1. 地址越界保护

其实 GDT、LDT 中的段限就是用于地址越界保护。

2. 存取控制保护

存取控制保护从两个方面来保证信息安全：

➤ 设置对存储区域的访问权限

RPL：规定访问者的访问权限；

DPL：规定段本身的访问权限；

访问者的访问权限必须段本身的访问权限时，才能访问段。

➤ 设置对存储区域的操作权限

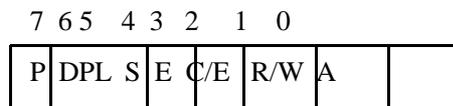
通过段描述符中 R/W 实现。

6. 3 80x86 的分页机制

6.3.180x86 的分页机制

80x86 中，逻辑页面大小为 4KB。

80x86 中采用两级页表：简单描述就是：每个页表占用一个物理页面（4KB），每个页表项占 4B，则每个页表 1K 个表项。因此一张页表可以覆盖 $1K * 4K = 4M$ 的地址空间。建立进程时，系统根据进程逻辑



页面的数目提供不同张数的页表，这样不同尺寸的进程就可以使用不同张数的页表。但是此时必须为每个进程建立一个页表目录，记录该进程的各个页表的存储位置。所以需要使用两级页表。

因为 80x86 中页表目录中的表项也占用 4B，所以每个页面可以记录 1K 个页表项，所以采用两级页表可以覆盖 $1k * 1K * 4K = 4G$ 的地址空间。也就是能覆盖最大容量的线性地址空间。

采用二级页表的好处：覆盖更大容量的线性地址空间、节省空间（因为如果采用一级页表，为了满足最大容量 4G 的需要，必须为 4G 空间中的每个页建立页表项。）。

6.3.2 分页机制的地址转换

1. 地址结构



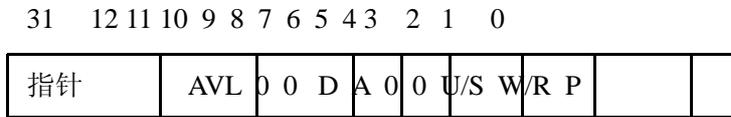
2. 地址转换

图 6.10 80x86 两级分页机制的地址转换（此图必须画）

图中：

- CR3 控制寄存器：记录页表目录在物理内存中的起始地址。该寄存器是一个 32 位寄存器，它的低 12 位总是 0，这样可以保证页表目录在物理地址空间总是按页面对齐的。
- 页表目录域和页表域中记录的都是查找项的索引值，即序号，因每个表项为 4B，所以需要乘 4，得到表项中的偏移地址。

6.3.3 页表目录与页表表项



1. 指针。页表目录中：是页表指针，指向某一页表。
 页表中：是页面指针，指向某一物理页面。
 实际上它们分别是页表或物理页面起始地址的高 20 位，该地址的低 12 位总为 0，这就保证了页表和物理页面在物理地址空间总是按页面对齐的，即它们总是位于 4K 页面的边界上。
2. AVL：供操作系统自行定义使用。
3. D：修改位（仅对页表表项有意义）
4. A：访问位。
5. U/S：访问权限，表示页表或页面本身的访问权限。U/S=1 是用户级，U/S=0 是系统级。
6. W/R：读/写保护位，操作限制。1 表示允许读写；0 表示只读
7. P：存在位。
8. 表项中 0 的位域保留以备扩充。

6.3.4 分页机制的存储保护

从两方面进行：

- 地址越界保护。访问到的页表项为 0 时，地址越界。（可以参见徐德民 p194 195 的例题）
- 存取控制保护。U/S 和 W/R

6.3.5 快表 TLB

从上述的分页机制可以看出，采用两级页表，需要访问 3 次内存。为此 80x86 中也引进了快表。它放在处理器芯片中的高速缓存中，称为转换旁视缓冲存储器 TLB（Translation Look-aside Buffers）。其中存放 32 个最近使用的页表项（页面地址）。

6. 4 Linux 的分段和分页结构

本节介绍 Linux 运行在 80x86 上时，其内核如何利用硬件机制实现对存储空间的分段分页管理。

6.4.1 Linux 的分段结构

1. 用户区和内核区

80x86 的分段机制把 64T 的虚拟地址空间分为最大长度为 4G 的线性地址空间。Linux 把每个线性地址空间提供给一个进程使用，所以每个线性地址空间就是用户的虚拟内存空间。在一个进程的线性地址空间中包含若干全局段和局部段。

- 内核代码段和内核数据段：由于对一个进程而言，只有它的虚拟存储空间是可见的。因此为了操作系统功能，系统内核必须包括在进程的虚拟地址空间中。Linux 把内核的代码和数据映射到线性地址空间的全局段中，它们就形成了内核代码段和内核数据段。
- 用户代码段和用户数据段：进程本身的代码和数据映射到进程线性地址空间的局部段，形成用户代码段和用户数据段。

为了保护内核，linux 把虚拟内存分成两部分：

- 内核区：包括内核代码段和内核数据段、内核使用的堆栈、全局数据结构。
- 用户区：包括用户代码段和用户数据段、进程堆栈、进程数据结构等

在 80x86 上 linux 进程虚拟空间中：

- 用户区：地址从 0x00000000~0xbfffffff，其大小为 3G；
- 内核区：地址从 0xc0000000~0xffffffff，其大小为 1G；

说明：

由于对每个进程而言，只有它的虚拟存储空间是可见的，而且每个进程的虚拟存储空间都是从 0 开始的相对地址空间，所以从系统角度看，每个进程在虚拟空间中的内核区和用户区的分布是相同的。

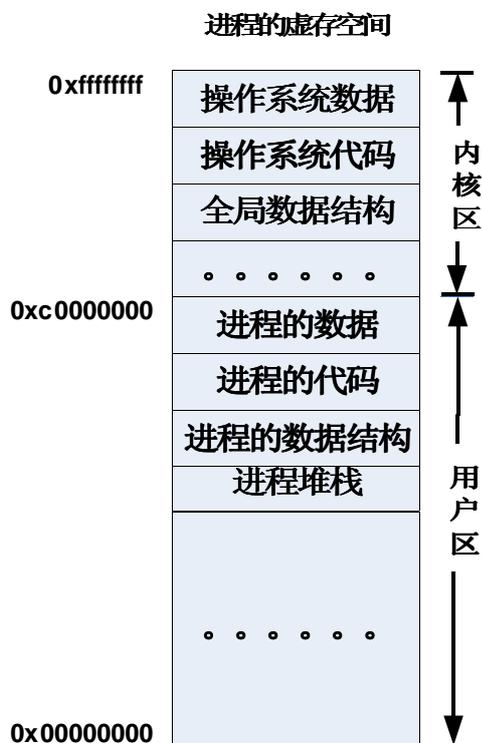


图6.13 进程的虚拟内存

2. GDT

由于大部分进程都只有一个代码段和一个数据段，为了提高地址映射速度，从 linux2.2 开始，把进程的一个代码段和一个数据段的描述符放到了 GDT 中，这样就可以直接从 GDT 中取得描述符，而不必再通过 GDT 访问 LDT。只有进程需要建立更多段时，才把它们的描述符放到 LDT 中。

在 linux 内核 2.2 的 arch/i386/kernel/head.s 文件中定义了 GDT 的内容。

NULL
未使用
内核代码段描述符
内核数据段描述符
用户代码段描述符
用户数据段描述符
未使用
未使用
任务0的TSS描述符
任务0的LDT描述符
任务1的TSS描述符
任务1的LDT描述符
.....
任务n的TSS描述符
任务n的LDT描述符

6.4.2 Linux 的三级分页结构

图6.14 linux的GDT

1. Linux 的三级分页结构

Linux 在 80x86 机器平台运行时，因为 80x86 是 32 位的，所以采用两级分页结构就可以有效地节省内存空间。但是如果 Linux 运行在 64 位机器平台时，如 Alpha 等，地址总线 64 位，寻址更大空间，此时二级页表将出现页表目录过大的问题。

为此，64 位机器的分页机制提供的是三级分页结构。Linux 运行在 64 位机器平台时也采用三级分页结构。

页目录：PGD (page directory)

页中间目录：PMD (page Middle directory)

页表：PTE (page table)

地址结构，以 alpha 机器为例：



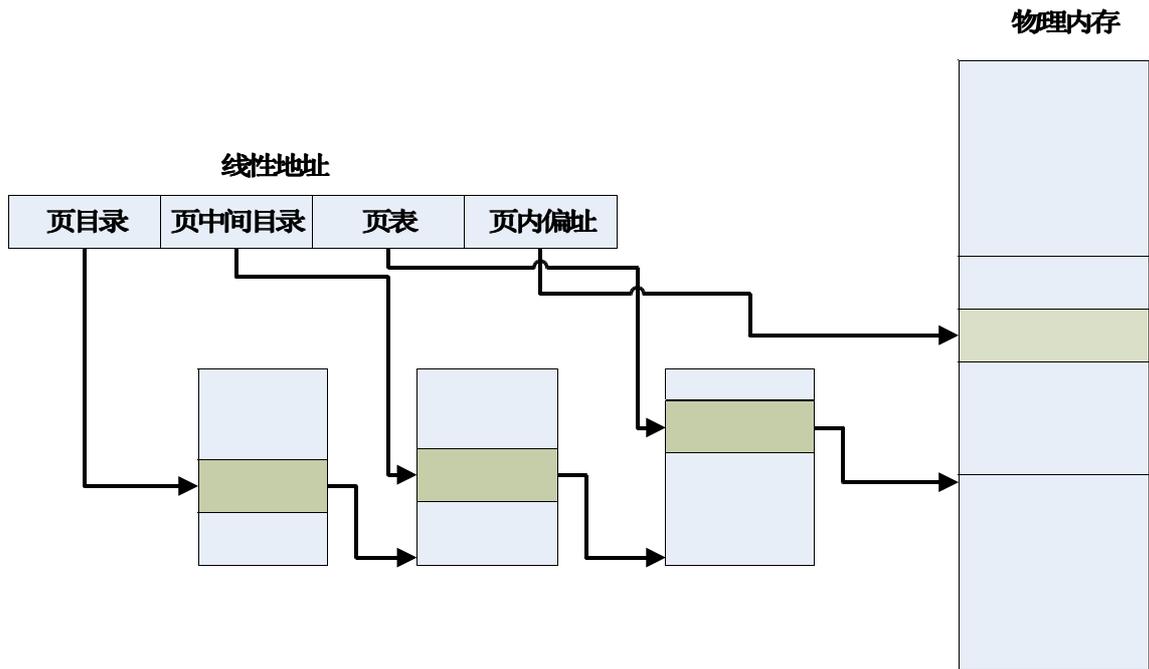


图6.15 Linux的三级分页结构

2. linux 与硬件无关的分页机构

linux 提供了与硬件无关的分页机构。所谓与硬件无关是指这种分页结构仅是一种存储管理模型。当 linux 运行在某种机器时，通过对存储管理模型有关参数的设置来适应机器的硬件。Linux 内核提供的与硬件无关的分页机构是三级分页存储管理模型。

Linux 内核把三级分页存储管理模型转换为两级分页结构的具体做法是：把三级分页存储管理模型的页中间目录域长度定义为 0。同时定义中间目录表只有一个表项。在 linux 源代码文件的 `/include/asm-i386/pgtable.h` 中有如下定义：

```
#define PTRS_PER_PTE 1024
#define PTRS_PER_PMD 1
#define PTRS_PER_PGD 1024
```

3. Linux 页面大小

Linux 中页面的尺寸由宏定义的符号常量 `PAGE_SIZE` 指定，在 80x86 机器中页面长度为 4KB，定义在 `include/asm-i386/page.h` 中：

```
#define PAGE_SHIFT 12
#define PAGE_SIZE (1UL<< PAGE_SHIFT) /*把 1 左移 12 位，即 4K*/
```

在 alpha 等 64 位机中，页面大小为 8K，定义在相应的 `page.h` 中：

```
#define PAGE_SHIFT 13
#define PAGE_SIZE (1UL<< PAGE_SHIFT) /*把 1 左移 13 位，即 8K*/
```

6.4.3 内核页表和进程页表

1. CR3 控制寄存器

进程切换时，linux 要重新设置 CR3 控制寄存器，使它指向新进程的页目录表。该任务由内核的汇编语言函数 `startup_32()` 完成。

2. 页目录表

linux 内核为了配合硬件分页机制设置了相应的数据结构，其中页目录表定义为一个具有 1024 个元素的数组：

```
Pgd_t swapper_pg_dir[1024] /*每个元素指向一个页表*/
```

3. 内核页表和进程页表

内核区和用户区的分界地址由符号常量 PAGE_OFFSET 确定，如 80x86 中 PAGE_OFFSET 的值是 0xc0000000 (3G)。所以此时页目录表的前 768 个表项对应用户区，后 256 项对应内核区（所有进程的内核表项完全相同）。在 linux 内核中使用 paging_init() 函数对页目录表进行初始化，它定义在 arch/i386/mm/init.c 中。

图 6.16 linux 进程的内核页表和进程页表

6. 5 Linux 进程地址空间管理

本节介绍 linux 对进程虚拟空间的管理方法和内核提供的对进程虚拟空间进行管理的数据结构及有关函数。

6.5.1 进程地址空间用户区的管理

Linux 内核对进程地址空间的管理主要是对用户区的管理。在 linux 中，系统为每个进程定义了一个 mm_struct 结构体来描述其虚拟内存的用户区。Mm_struct 结构体的首地址记录在每个进程的任务结构体的成员项 mm 中。

mm_struct 结构体定义在 /include/linux/sched.h 中，如下所示：

```
struct mm_struct
{
    Int count; /*共享进程数*/
    Pdg_t *pgd; /*进程页目录表*/
    Unsigned long context; /*进程上下文地址*/
    Unsigned long start_code,end_code,start_data,end_data; /*代码段、数据段的首、终地址*/
    Unsigned long start_brk,brk, start_stack,start_mmap; /* start_stack 堆栈首地址*/
    Unsigned long arg_start,arg_end,env_start,env_end; /*参数区、环境变量去的首、终地址*/
    Unsigned long rss,total_vm,locked_vm; /* total_vm 占用的虚拟内存总量*/
    Unsigned long def_flags;
    Struct vm_area_struct *mmap; /*指向该进程虚拟区域链表首地址*/
    Struct vm_area_struct *mmap_avl; /*指向该进程虚拟区域 AVL 树的根*/
    Struct semaphore mmap_sem;
}
```

6.5.2 虚拟区域

虚拟区域: linux 把进程虚拟内存用户区进一步划分若干个连续的区域, 把性质不同的信息置入不同的存储区域内, 这些连续的存储区域就称为虚拟区域。这个区域内的信息具有相同的操作和访问特性。

Linux 对每个虚拟区域用 `vm_area_struct` 结构体来描述, 它定义在 `/include/linux/mm.h` 中:

```
struct vm_area_struct
{
    Struct mm_struct *vm_mm;           /*指出该虚拟区域属于哪个进程的虚拟内存*/
    Unsigned long vm_start;           /*该虚拟区域的首地址*/
    Unsigned long vm_end;             /*该虚拟区域的终止地址*/
    Pgprot_t vm_page_prot;           /*规定虚拟区域的各个页面的保护特性*/
    Unsigned short vm_flags;          /*该虚拟区域操作特性*/
    Short vm_avl_height;              /*表示 ALV 树的高度*/
    struct vm_area_struct *vm_avl_left; /*相邻的低地址虚拟区域*/
    struct vm_area_struct *vm_avl_right; /*相邻的高地址虚拟区域*/
    struct vm_area_struct *vm_avl_next; /*指向下一个虚拟区域*/
                                        /*在虚拟区域用于特殊的情况下时, 需要链接成双向链表*/
    struct vm_area_struct *vm_avl_next_share; /*指向双向链表的下一个虚拟区域*/
    struct vm_area_struct *vm_avl_prev_share; /*指向双向链表的前一个虚拟区域*/
    struct vm_operations_struct *vm_ops; /*指向该区域的操作函数的集合的结构体*/
    unsigned long vm_offset;           /*若虚拟区域的内容是某个文件的映射, 则
                                        vm_offset, 指出该区域相对文件首址的偏移
                                        量; 若其内容属于虚拟空间的某个共享区域,
                                        则 vm_offset, 指出该区域相对共享区域首址
                                        的
                                        偏移量*/
    struct inode *vm_inode;           /*虚拟区域可能是映射的是某个文件的内容,
                                        指向映射文件, 否则为 NULL*/
    unsigned long vm_pte;
}
```

其中: 一个虚拟区域的操作特性由 `vm_flags` 确定, 它的值是内核定义的一系列符号常量:

<code>VM_READ</code>	虚拟区域允许读
<code>VM_WRITE</code>	虚拟区域允许写
<code>VM_EXEC</code>	虚拟区域允许执行
<code>VM_SHARED</code>	虚拟区域允许被多个进程共享
<code>VM_GROWSDOWN</code>	虚拟区域可以向下延伸
<code>VM_GROWSUP</code>	虚拟区域可以向上延伸
<code>VM_SHM</code>	虚拟区域是共享存储器的一部份
<code>VM_LOCKED</code>	虚拟区域可以加锁
<code>VM_STACK_FLAGS</code>	虚拟区域作为堆栈使用

6.5.3 虚拟区域的建立和映射

1. 定义

Linux 中进程运行时，必须把磁盘上的用户程序代码和数据组成的可执行映像映射到该进程的虚拟地址空间，如果使用共享库，还需要把共享库映射到它的虚拟空间。此时系统就调用 `do_mmap()` 建立一个虚拟区域，并把有关的信息映射到该虚拟区中。

`do_mmap()` 定义在 `/mm/mmap.c` 中。其原型定义如下：

```
unsigned long do_mmap(struct file *file, unsigned long addr, unsigned long len, unsigned long prot,
unsigned long flags, unsigned long off)
```

参数说明：

- **Addr:** 指明要建立的虚拟区域在虚拟内存中的开始地址
- **Len:** 虚拟区域的尺寸
- **File:** 当映射到该区域的是文件的内容时，`file` 指向该文件的结构体的指针
若 `file` 为 `NULL`，映像到虚拟区域的是一组长度为 `len` 的空白页面，这个映射称为匿名映射。
- **Off:** 指出映射到虚拟区域的内容相对于文件起始位置的偏移量。
- **Prot:** 指定该虚拟区域的访问特性，其值得符号常量定义如下：

<code>PROT_READ</code>	<code>0x1</code>	对虚拟区域允许读
<code>PROT_WRITE</code>	<code>0x2</code>	对虚拟区域允许写
<code>PROT_EXEC</code>	<code>0x4</code>	虚拟区域代码允许执行
<code>PROT_NONE</code>	<code>0x0</code>	不允许访问虚拟区域

- **Flag:** 指定虚拟区域的属性，常用的符号常量有：
`MAP_FIXED`: 虚拟区域固定在以 `addr` 开始的位置，不允许其他虚拟区再使用这个位置。
`MAP_SHARED`: 指定了对虚拟区域的操作时作用在一组共享的物理页面上的，即其他进程的虚拟区域也作用在同一组页面上，当一个进程对共享页面修改时，其他进程可以同时感知到。如 `system V` 共享内存的进程通信就是采用这样的方法。
`MAP_PRIVATE`: 指定了对虚拟区域的写入操作将引起页面的拷贝，即 `MAP_PRIVATE` 的虚拟区域具有“写时拷贝”的属性。

2. `do_mmap()` 函数说明

该函数功能主要由 3 部分组成：

第一部分：检查调用该函数时给定的各项参数是否合理。

第二部分：建立虚拟区域的 `vm_area_struct` 结构体并进行初始化

第三部分：对进程 `mm` 结构体中的有关量进行修改

执行结束后返回已建立的虚拟存储区域首地址。

`do_mummap()` 函数：撤销对虚拟区域的映射。

6. 6 Linux 物理空间管理

6.6.1 Linux 物理内存空间

Linux 物理内存空间分成两个区域：

- 内核区：存放内核代码和数据，以及内核管理进程的数据结构等。低地址区
- 动态 RAM 区：存放各个用户进程的代码、数据等。高地址区。

需要注意：内核区映射到进程线性地址空间时在高地址区域，故内核在进程虚拟内存和物理内存所占据的位置不同。

图 6.18 linux 的物理内存空间（必须画）

6.6.2 物理页面的管理

Linux 以页面为单位来分配内存。

Linux 对每个物理页面都使用一个页面描述符（page 结构体）描述其物理特性。其定义包含在 `/include/linux/mm.h` 中，并进一步被定义为 `mem_map_t` 类型。

```
typedef struct page
{
    struct page *next;                /*双向链表的下一个*/
    struct page *prev;                /*双向链表的前一个*/
    struct page *next_hash;           /*指向 hash 表后一个*/
    struct page *prev_hash;           /*指向 hash 表前一个*/
    unsigned dirty: 16, age: 8;        /*age 记载被访问的情况；dirty：是否被修改*/
    atomic_t count;                   /*共享进程数目*/
    unsigned long flags;               /*页面状态*/

    /*当页面内容是文件的一部分*/
    struct inode *inode;               /*指向文件的 inode */
    unsigned long offset;              /*指出在文件中的偏移量 */

    /*系统把所有 page 结构体集中组成一个 mem_map 数组*/
    /*在 mem_map 数组中的下标*/
    unsigned long map_nr;
    unsigned long swap_unlock_entry;
    struct wait_queue *wait;
    struct buffer_head * buffers;
} mem_map_t;
```

6.6.3 空闲页面管理——buddy 算法

为了提高访问页面的速度，以及满足使用连续面的要求（如较大的线性数组要求连续的页面）。操作系统在分配内存时要尽量保留连续的页面，所以实施分配时不能以单一页面分配，而是以多个页面为单位分配。根据这个思想，linux 对内存空间的管理和分配采用了 Buddy 算法。Buddy 是“伙伴”、“搭档”的意思。

Buddy 算法的基本思想：以多个页面为单位管理和分配空闲区域。

1. 空闲页块组

- 它把物理内存中的所有页面按照 2 的整数次幂 (2^n) 进行划分，linux2.0 中年 (0~5) 对物理内存进行 6 次划分 (1, 2, 4, 8, 16, 32)。这样划分后形成大小不同的存储块，称为页面块 (页块)。
- 包含一个页面的块称为 1 页块，包含两个页面的块称为 2 页块，依次类推。将每种页块按照它们的先后顺序两两结合成一对对的 buddy“伙伴”，如：
1 页块中：0 和 1、2 和 3、4 和 5、…….; 就是一对对的 1 页块 buddy“伙伴”
2 页块中：0~1 和 2~3、4~5 和 6~7、8~9 和 10~11、……. 就是一对对的 2 页块 buddy“伙伴”
- **对空闲区域的管理按照页块大小分组进行管理。**
系统设置了一个静态数组 free_area[] 来管理各个空闲页块组。在/mm/page_alloc.c 中。

```
#define NR_MEM_LISTS 6
```

```
Static struct free_area_struct free_area[NR_MEM_LISTS];
```

```
Struct free_area_struct
```

```
{
```

```
    Struct page *next;           /*空闲链表下一个节点*/
```

```
    Struct page *prev;          /*空闲链表前一个节点*/
```

```
    Unsigned int *map;           /*指向相应页块的位图，其位于内存 bitmap 区*/
```

```
}
```

该数组共 6 个元素，指向 1、2、4、8、16、32 六种页面块。

2. 两种管理方法：位图法和空闲页块组链表。

1) 位图法

Linux 对内存页面块的每种划分都对应一个位图 map，图 6.19 给出了 1、2、4 页块位图示意图。

在位图中每一位表示一对 buddy 页块的使用情况，方法：

如：1 对都空闲，则该位为 0；

1 对都占用（全部或部分），则该位为 0；

1 对中，1 组空闲而另一组被占用（全部或部分），则该位为 1；

2) 空闲页块组链表

系统按照 buddy 关系把具有相同大小的空闲页面块组成空闲页面块，每个空闲页块组用一个双向循环链表进行管理。例，见图 6.19

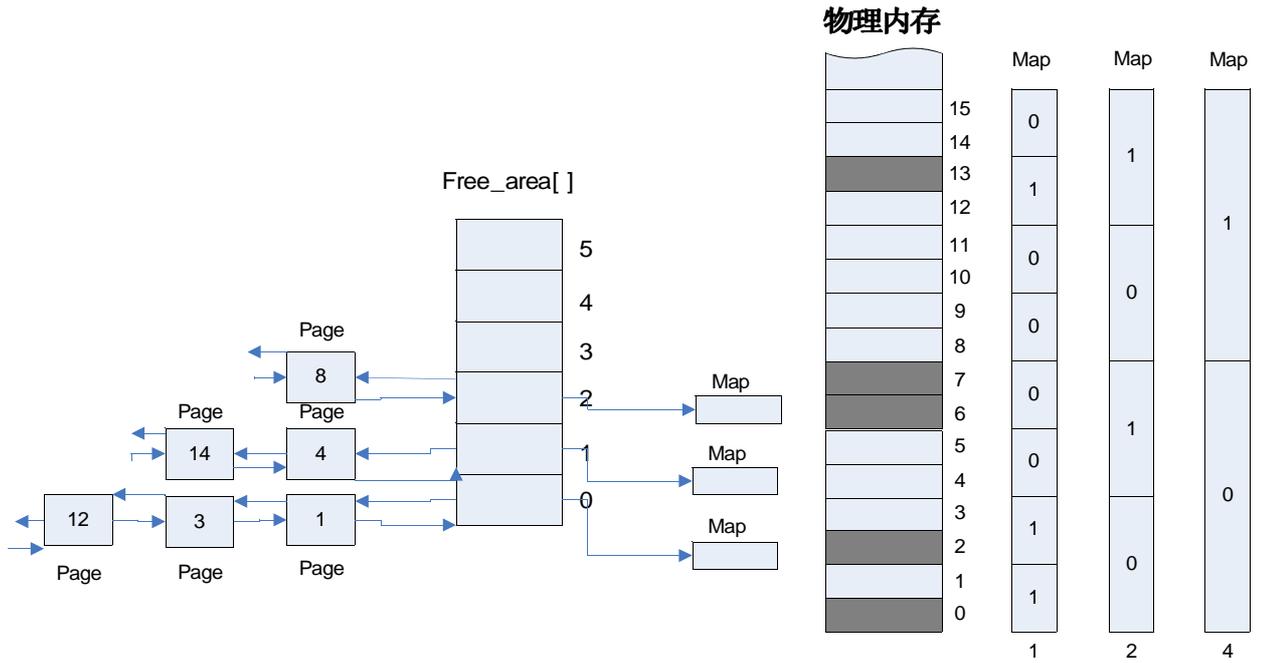


图6.19 buddy算法演示

3. 分配和释放管理

用事例说明：图 6.20（图 6.19 中，若系统请求 3 个页面） 和图 6.21（图 6.20 中，页面 13 被释放。）

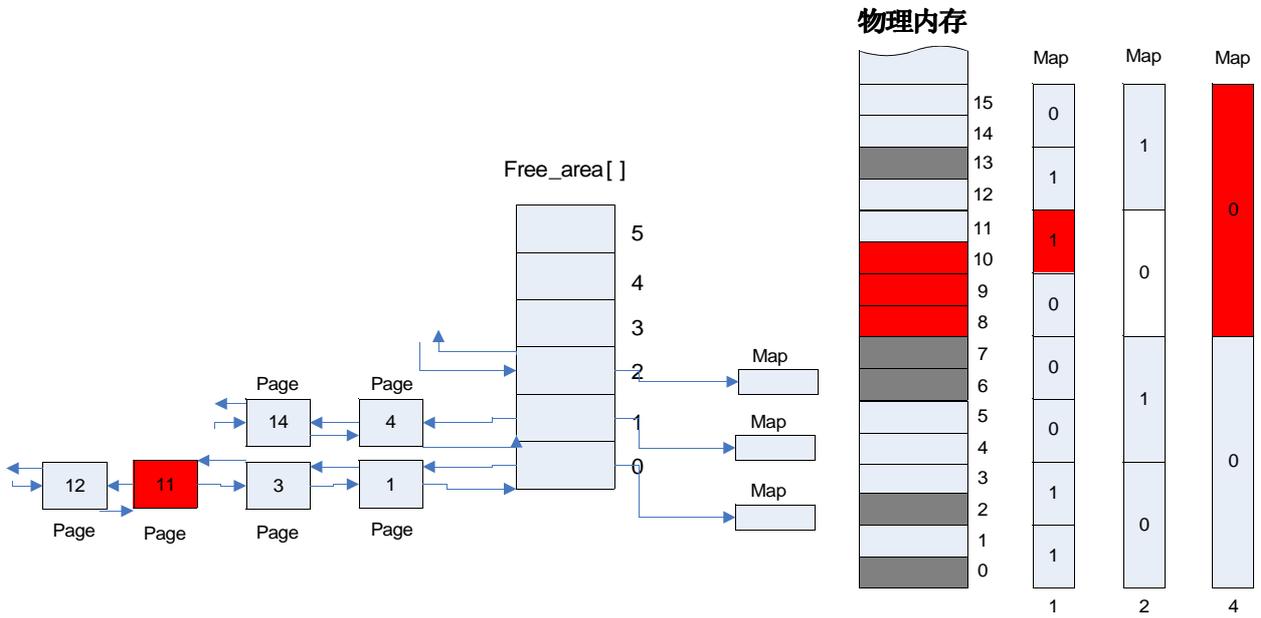


图6.20 buddy算法的页面分配

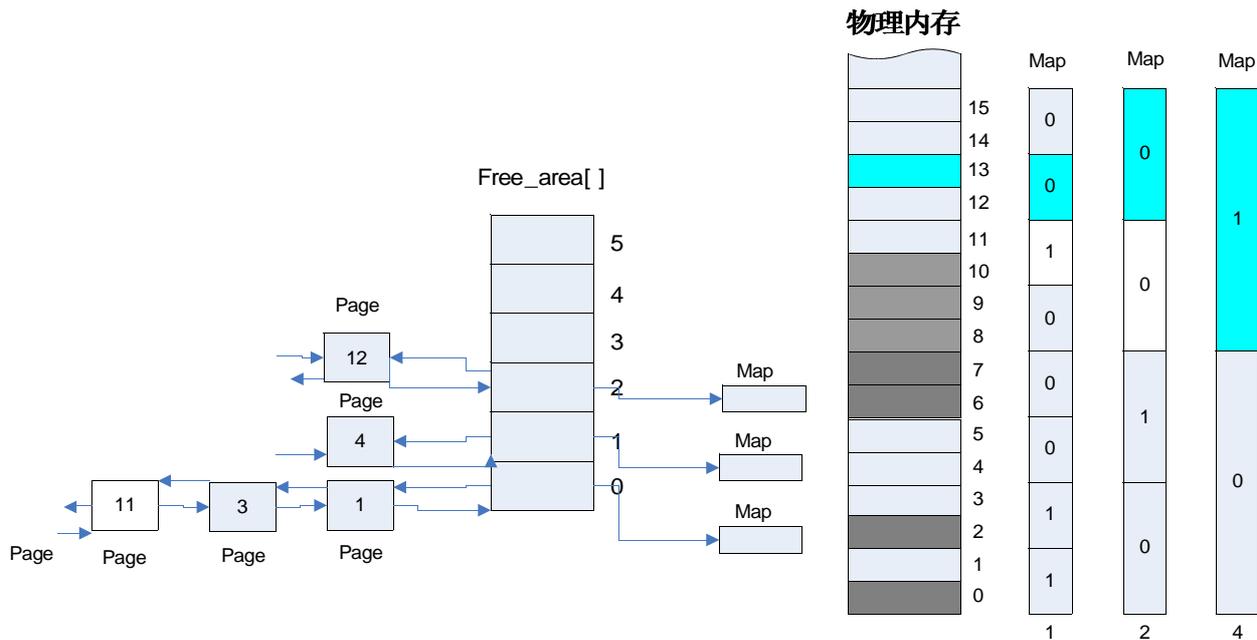


图6.21 buddy算法的页面释放

6.7 内存的分配与释放

Linux 中设置了多个用于虚拟内存和物理内存分配和释放的函数，本节主要介绍其中两对：面向物理内存分配和释放的函数 `kmalloc()` 和 `kfree()`；面向虚拟内存的分配和释放的函数 `vmalloc()` 和 `vfree()`；

6.7.1 物理内存分配的数据结构

Linux 中 `kmalloc()` 和 `kfree()` 用于分配和释放小于 128K 的连续物理内存空间。使用它可以分配到 [32B, 128KB] 的连续的内存空间。它在 Buddy 算法的基础上又设置了专门的数据结构来管理内存。

在使用 `kmalloc()` 和 `kfree()` 分配和释放内存是以块为单位进行的。可以分配的块单位记录在 `blocksize` 表中，它是一个静态数组，定义在 `/mm/kmalloc.c` 中：

```
#if PAGE_SIZE == 4096
Static const unsigned int blocksize[]={
32,64,128,252,508,1020,.....,131072-16,0
}
```

对页面大小为 4K 的机器，块单位共 13 种，它们近似于 2 的次幂。

可以块的大小与页面大小不一致，可能小于或等于大于页面。

如小于一个页面时，需要将一个页面再次按照第一次使用该页面的块单位来划分页面。所以每个页面又需要一个 `page_descriptor` 结构体（页描述符）来记录它的划分情况，它放在页面首部。

以下略

6.7.2 物理内存分配函数

略

6.7.3 虚拟内存分配函数

略

第7章 文件管理

7.1 文件与文件系统

7.1.1 文件

文件：具有文件名的一组信息组合，包括两部分：

文件体：文件本身的信息；

文件说明：文件存储和管理信息；如：文件名、文件内部标识、文件存储地址、访问权限、访问时间等；

7.1.2 文件的种类

文件系统在管理文件时还要识别和区分文件的类型，如果是文件系统所确认的文件类型，则可根据类型对文件进行合理的操作。

(1) 按用途分类

系统文件、库文件和用户文件。

(2) 按保护级别分类

根据限定的使用文件的权限：执行文件、只读文件和读写文件等。

(3) 按信息流向分类

物理设备的特性决定了文件信息的流向：输入文件、输出文件和输入输出文件。

(4) 按文件的性质分类

根据文件的性质分：普通文件、目录文件、设备文件等。

(5) 按文件的组织结构分类

由用户组织的文件称逻辑文件：流式文件和记录式文件。

文件在存储介质上的组织方式称文件的物理结构（物理文件）：顺序文件、链接文件和索引文件等。

7.1.3 文件系统及其功能

1. **文件系统：**操作系统中管理文件的机构，提供文件存储、提供文件在外存中的组织方式，以及文件访问控制等功能。文件系统的三个部分：管理软件、被管理软件、相关的数据结构

2. 文件系统的功能：

(1) 目录管理

文件目录是实现按名存取的一种手段。

建立一个新文件，应与该文件有关的一些属性登记在文件目录中；

读一个文件，应从文件目录中查找指定文件是否存在并核对是否有权使用。

一个好的目录结构应既能方便检索，又能保证文件的安全。

(2) 文件的组织

用户按信息的使用和处理方式组织文件，称为文件的逻辑结构或称为逻辑文件。把逻辑文件保存到存储介质上的工作由文件系统来做，这样可减轻用户的负担。根据用户对文件的存取方式和存储介质的特性，文件在存储介质上可以有多种组织形式。把文件在存储介质上的组织方式称为文件的物理结构或称为物理文件。因此，当用户要求保存文件时，文件系统必须把逻辑文件转换成物理文件，而当用户要求读文件时，文件系统又要把物理文件转换成逻辑文件。

(3) 文件存储空间的管理

要把文件保存到存储介质上时，必须记住哪些存储空间已被占用，哪些存储空间是空闲的。文件只能保存到空闲的存储空间中，否则会破坏已保存的信息。当文件没有必要再保留而被删除时，该文件所占的存储空间应成为空闲空间。

(4) 文件操作

为了保证文件系统正确地存储和检索文件，规定了在一个文件上可执行的操作，这些可执行的操作统称为“文件操作”。文件系统提供的基本文件操作有建立文件、打开文件、读文件、写文件、关闭文件和删除文件等。“文件操作”是文件系统提供给用户使用文件的一组接口，用户调用“文件操作”提出对文件的操作要求。

(5) 文件的共享、保护和保密

在多道程序设计的系统中，有些文件是可以共享的，例如，编译程序、库文件等。实现文件共享既节省文件的存放空间，又可减少传送文件的时间，但必须对文件采取安全保护措施。既要防止有意或无意地破坏文件，又要避免随意地剽窃文件。

7. 2 文件的组织结构

7.2.1 文件的逻辑结构

逻辑结构：它是用户所观察到的文件组织形式，是用户可以处理的数据及结构，它独立于物理特性，又称为文件组织（file organization）。

分类：

- a. 有结构记录式文件（数据库）：包含若干顺序排列的记录
 - ◆ 变长记录：数据项/字段不同；数据项本身不同。

- ◆ 定长记录：文件长度=记录总个数×记录长
- b. 无结构字符流式文件：文件的信息不组成记录，文件的长度即为字符总个数。（源程序、文本文件）

7.2.2 文件的物理结构

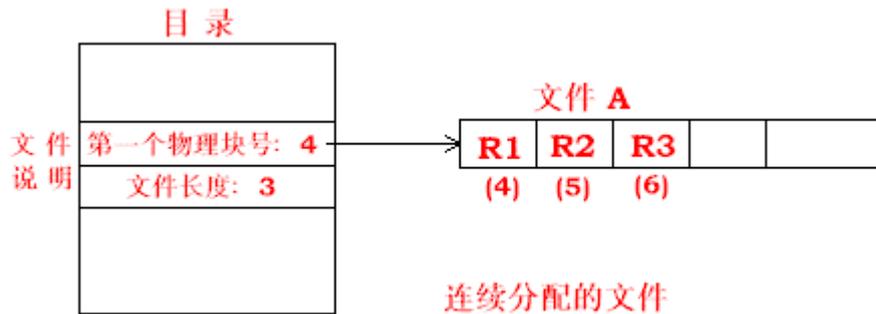
1. 物理结构：文件在外存上的实际的组织形式。

- 1) 物理块：以物理块为基本单位分配和传输信息，物理块大小由存储设备和 OS 确定。
- 2) 物理块大小与逻辑记录大小之间不一定一致。

2. 文件物理结构的几种形式：1) 顺序结构 2) 链接结构 3) 索引结构

一、顺序结构（顺序文件或连续文件）

一个文件在逻辑上连续的信息被存放到磁盘上依次相邻的块上。逻辑记录顺序与磁盘块的顺序相一致。
例：



优点：存取速度快、结构简单、支持顺序存取和随机存取。

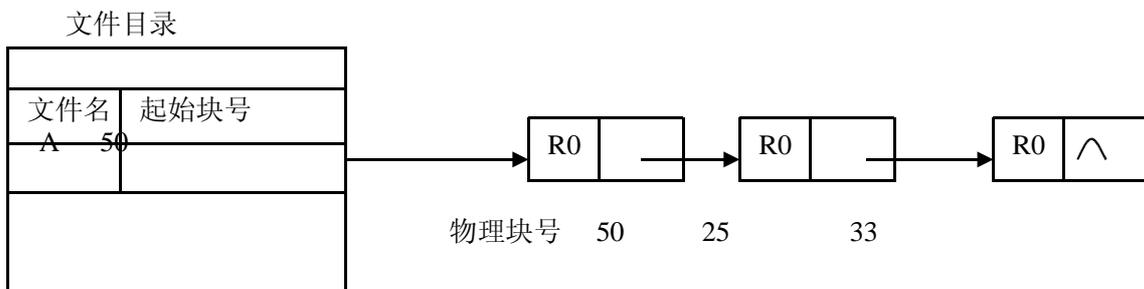
存在的问题：

- (1) 磁盘存储空间的利用率不高, 容易产生碎片。
- (2) 对输出文件很难估计需多少磁盘块。
- (3) 影响文件的扩展。

二、链接结构（链接文件或串联文件）

基本思想：将文件存放在外存中若干个物理块中，这些物理块不必连续。每个物理块的最后一个单元用作指针，指向下一个物理块的地址。最后一块中的指针可用特殊字符（例如“—1”）表示文件到此结束，从而将同一个文件的物理块链接起来。

优点：解决了顺序结构中的所有问题。



磁盘上所有空闲块都可以被利用；

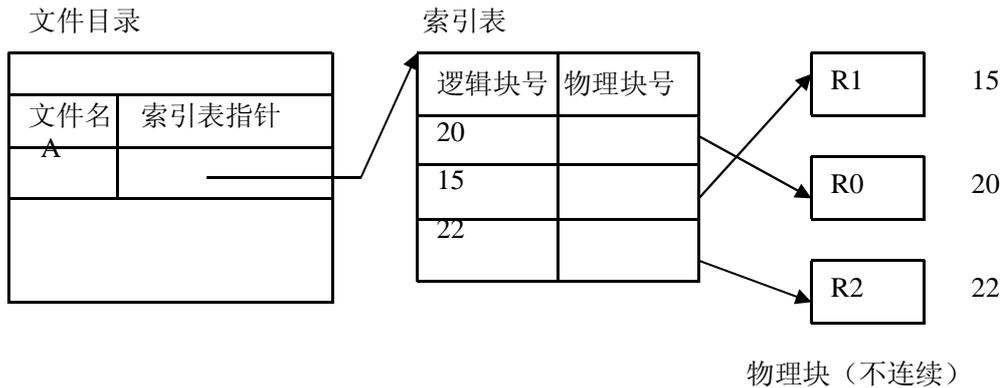
建立文件时也不必事先考虑文件的长度，文件可继续扩展；
便于在文件的任何位置插入一个记录或删除一个记录。

缺点：采用随机存取方式是低效的。文件只能按指针链接顺序访问，故存取速度慢；
可靠性问题，如指针出错。
链接指针占用一定的空间。

读出一块信息时，应将其中的指针分离出来，保证用户使用信息的正确性。

2、索引结构（索引文件）

基本思想：为每一个文件建立一张索引表，每一表项记录文件所在的一个物理块。



优点：能方便地实现文件的扩展、记录的插入和删除。

缺点：必须增加索引表占用的空间和读写索引表的时间。索引表的查找策略对文件系统效率影响很大。

索引表的管理：当索引表非常大时，需要多个磁盘块存放，各磁盘块之间可用指针链起来。当随机存取某个记录时，可能要沿链搜索才能找到该记录的存放地址，很费时间。

解决方案：**多级索引、混合索引(下例适用 Unix System V)**

7.3 文件的目录结构

7.3.1 文件说明、目录文件

1. 文件说明 FCB：记录文件的管理、描述信息。与文件一一对应。：

➤ 基本信息

- 文件名：字符串，通常在不同系统中允许不同的最大长度。可以修改。有些系统允许同一个文件有多个别名(alias)；
- 别名的数目；
- 文件的物理地址：包括哪个设备或文件卷 volume，以及各个存储块位置；
- 文件逻辑结构
- 文件物理结构
- 文件类型：可有多种不同的划分方法，如：
 - ◇ 有无结构（记录文件，流式文件）
 - ◇ 内容（二进制，文本）

- ◇ 用途（源代码，目标代码，可执行文件，数据）
- ◇ 属性 attribute（如系统，隐含等）
- ◇ 文件组织（如顺序，索引等）

□ 文件长度（当前和上限）：以字节、字或存储块为单位。可以通过写入或创建、打开、关闭等操作而变化。

➤ 访问控制信息

- 文件所有者（属主）：通常是创建文件的用户，或者改变已有文件的属主；
- 访问权限（控制各用户可使用的访问方式）：如读、写、执行、删除等；

➤ 使用信息

- 创建时间
- 最晚一次读访问的时间和用户
- 最晚一次写访问的时间和用户

2. 索引节点 i_node

在UNIX、Linux 操作系统中，把文件说明（文件描述符）信息分成两个部分：

- a) 符号文件目录：由文件名和文件内部标识组成的树状结构，按文件名排序；
- b) 基本文件目录（索引节点 i_node 目录）：由其余文件说明信息组成的线性结构，按文件内部标识排序；

3. 目录文件：有所有文件目录组成的一个专门的独立的文件。目录文件是操作系统管理文件的重要依据。

7.3.2 文件目录结构

1. 一级目录：整个目录组织是一个线性结构，系统中的所有文件都建立在一张目录表中。所有文件都登记在同一个文件目录中。它主要用于单用户操作系统。它具有如下的特点：

- 结构简单；
- 文件多时，目录检索时间长；
- 有命名冲突：如多个文件有相同的文件名（不同用户的相同作用的文件）或一个文件有多个不同的文件名（不同用户对同一文件的命名）；

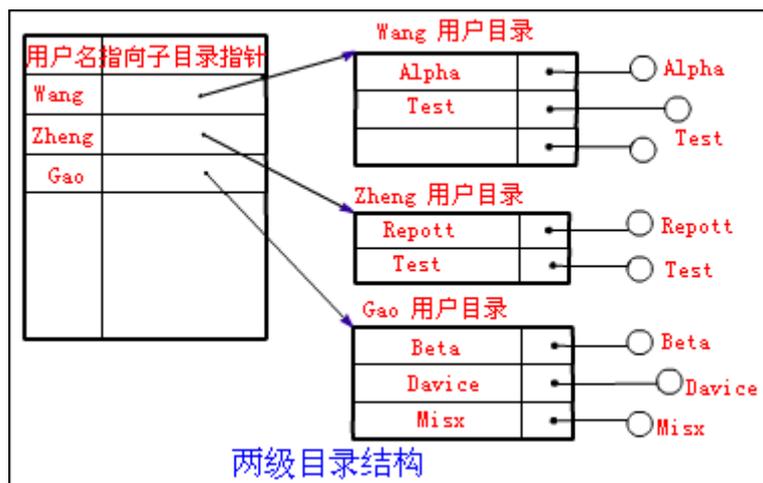
□

文件名	状态位	物理地址	文件其它属性
Alpha			
Report			
Text			

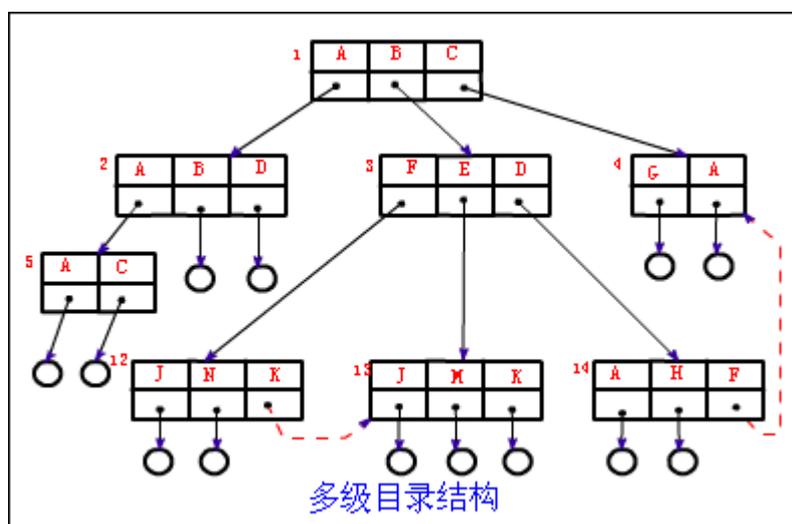
单级目录

2. 二级目录：在根目录（第一级目录）下，每个用户对对应一个目录（第二级目录），在用户目录下是该用户的文件，而不再有下级目录。适用于多用户系统，各用户可有自己的专用目录。

第一级为主文件目录，主文件目录以用户名为索引，对每个用户都设置一个指向用户文件目录的指针。第二级为用户文件目录，用户文件目录为本用户的每一个文件设置一个目录项。



3. 多级目录：或称为树状目录 (tree-like)。允许用户在自己的文件目录中根据不同类型的文件再建立子目录。在文件数目较多时，便于系统和用户将文件分散管理。适用于较大的文件系统管理。下面是几个与目录相关的概念：



- 目录树：中间结点是目录，叶子结点是目录或文件。
- 目录的上下级关系：当前目录 (current directory, working directory)、父目录 (parent directory)、子目录 (subdirectory)、根目录 (root directory) 等；
- 路径 (path)：每个目录或文件，可以由根目录开始依次经由的各级目录名，加上最终的目录名或文件名来表示；
- 相对路径：从某一目录开始，依次列出到达某文件的目录文件名，加上文件名。
- 当前目录：最近访问过的或正在使用的目录。当前目录的内容被复制在内存缓冲区内。可以提高查找文件说明信息的速度。

7. 4 文件存取与操作

1. 文件的存取方法

- **顺序存取：**按照文件的逻辑地址依次顺序存取文件中的信息；
- **随即存取：**可以从任意指定的位置开始读取文件；（编号或地址）
- **直接存取：**又称按键存取（数据库中，按照关键字查询）

2. 文件存储设备

- **顺序存取设备：**磁带。用于保存文档。
- **直接存取设备：**

3. 活动文件

活动文件：用户正在使用的文件。文件被执行“打开”操作时内存中建立该文件的控制块，它就从静止状态变为活动状态。

系统打开文件表：记录系统中所有打开文件的控制。

进程活动文件表：记录本进程中打开的所有文件。（指保存每个文件在系统活动表中的位置）

4. 文件操作（该部分不讲）

文件的功能是存储信息以备读取，不同的操作系统提供了不同的操作以对文件进行存取，以下是一些最常用的与文件有关的系统调用：

系统调用的步骤：

- a. 准备参数
- b. 调用系统调用
- c. 判返回值（* 出错，给出错误号等； * 正常返回）

1)CREATE 创建一个空的文件并设置它的一些属性

2)DELETE

当用户或系统不再需要一个文件时，就可以将其删除以释放存储空间。这个系统调用就是完成这一功能。另外，一些操作系统会自动将 n 天没有被存取的文件删除。

3)OPEN

一个进程在存取文件之前，必须先将其打开。系统调用 OPEN 的功能是将文件的属性及其在磁盘上的地址列表取出存入主存中以备后序的调用使用。

4)CLOSE

当对一个文件的所有存取操作结束，在主存中的文件属性列表和地址列表就不再需要了，因此文件就应该被关闭以释放主存空间。

5)READ

这个系统调用被用于从文件中读取数据，一般是读取当前位置的字节。调用者必须指定需要读取的字节数以及相应的缓冲区。

6)WRITE 将数据写到文件的当前位置，如果当前位置是文件尾，文件就被增长。

7)APPEND 将数据添加至文件尾。

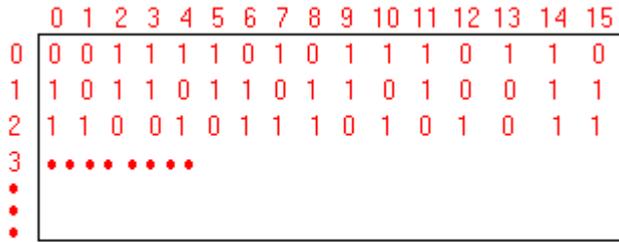
8)SEEK 在随机存取文件中，该系统调用被用于确定存取文件的当前位置。

9)SET ATTRIBUTES 文件的一些属性是由用户自己设定，并且可以在创建之后进行修改。

10)RENAME 对文件进行改名操作。

7. 5 文件存储空间的管理

1、位示图



位示图

对每个磁盘可以用一张位示图指示磁盘空间的使用情况。一个磁盘的分块确定后，根据总块数决定位示图由多少字组成，位示图中的每一位与一个磁盘块对应，某位为“1”状态表示相应块已被占用，为“0”状态的位所对应的块是空闲块。

块号 $b = (i-1) \times n + j$

- $i = (b-1) \text{DIV } n + 1$
- $j = (b-1) \text{MOD } n + 1$

2、空闲块表

系统为每个磁盘建立一张空闲块表，表中每个登记项记录一组连续空闲块的首块号和块数，空闲块数为“0”的登记项为“空”登记项。

适合采用顺序结构的文件。

序号	第一个空白块号	空白块个数	物理块号
1	2	4	2、3、4、5
2	9	3	9、10、11
3	15	5	15、16、17、18、19
4			

三、空闲块链表

1、单块连接

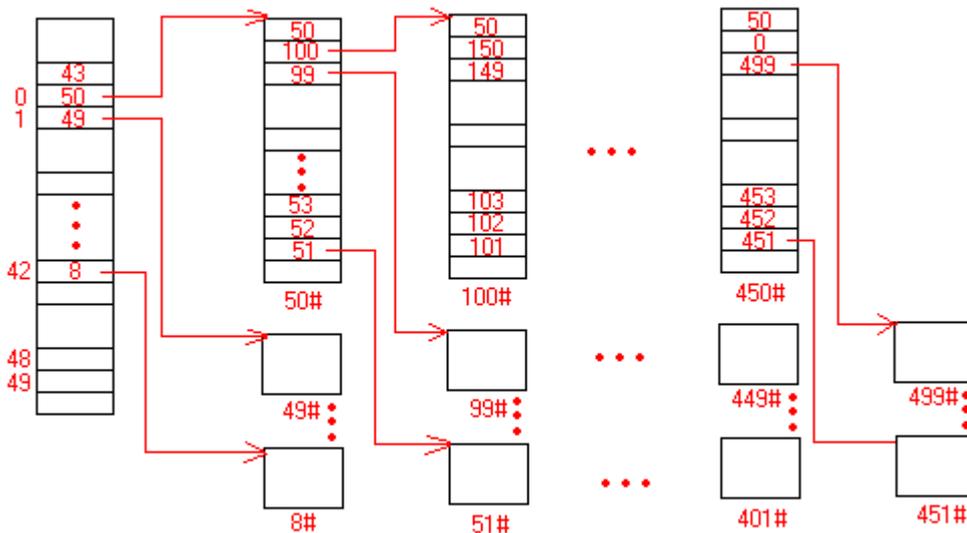
把所有空闲块用指针连接起来，每一个空闲块中都设置一个指向另一个空闲块的指针，所有的空闲块就构成了一个空闲块链。系统设置一个链首指针，指向链中的第一个空闲块，最后一个空闲块中的指针为“0”。

效率较低，麻烦费时。

2、成组连接

UNIX 系统：采用空闲块成组连接的方法。

采用成组连接后，分配回收磁盘块时均在内存中查找和修改，只是在一组空闲块分配完或空闲的磁



空闲块成组链接法

盘块构成一组时才启动磁盘读写。比单块连接方式效率高。

7. 6 文件的共享和保护

7.6.1 文件存取控制

1. 存取控制矩阵

如下图，记录所有用户对所有文件的使用权限。

文件权限	用户 1			用户 2			用户 3		
	R	W	E	R	W	E	R	W	E
文件 A	√	ρ	√	√	√	√	√	√	√
文件 B	√	ρ	√	ρ	ρ	ρ	ρ	ρ	√
文件 C	√	√	√	√	√	√	√	√	√
文件 D	√	√	√	√	√	√	√	√	√

缺点：占用较大空间；查找速度慢

2. 存取控制表

对存取控制矩阵改进，将用户进行分组。对一个文件而言，针对文件所有者，即其所在的各个用户组规定不同的存取权限，从而形成该文件的存取控制表。它比存取控制矩阵规模要小得多。

文件名	File
用户	
文件所有者	RWE
用户组 A	RW
用户组 B	RE
其他	R

3. 口令

对文件规定一个口令，放在文件说明中，并规定使用该文件的用户。当用户访问文件时，必须提供口令。验证正确后，才能访问。

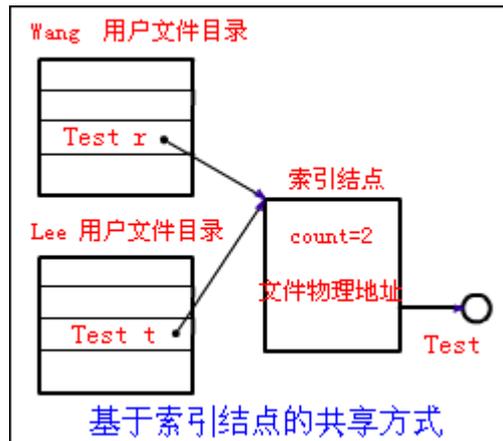
4. 加密

对文件中所有信息以密码形式重新编码存储，在读文件时，再进行译码解密。通常做法是：在用户向外存写入一个文件时，通过一个加密程序对文件的信息进行变化处理。读取文件时，通过一个解密程序把文件恢复原貌。

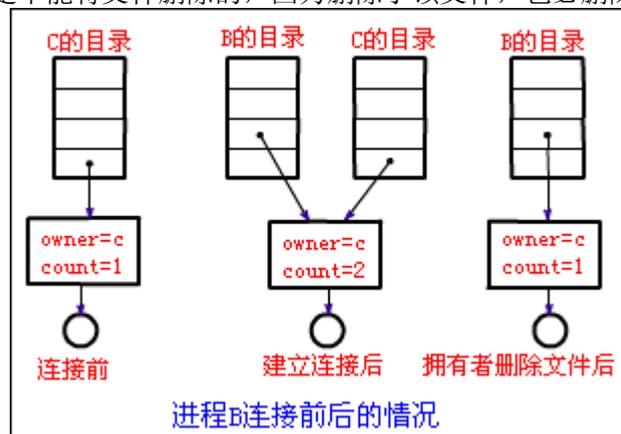
7.6.2 文件共享的实现方法

1. 基于索引结点的共享方式：

引用索引结点，即诸如文件的物理地址及其它的文件属性等信息，不再放在目录中，而是放在索引结点中。在文件目录中只设置文件名及指向相应索引结点的指针，如右图所示。



在索引结点中还应有一个链接计数 count，用于表示链接到本索引结点上的用户目录项的数目。当用户创建一个新文件时，他是该文件的所有者，此时将 count 置 1。当有用户 B 要共享此文件时，在用户 B 的用户目录中增加一目录项，并设置一指针指向该文件的索引结点，此时，文件主仍然是 C，count=2。如果用户 C 不再需要此文件，是不能将文件删除的，因为删除了该文件，也必删除了该文件的索引结点。



2. 符号链实现文件共享:

B 为了共享 C 的一个文件 F，可以由系统创建一个 LINK 类型的新文件，将新文件 F 写入 B 的用户目录中，以实现 B 的目录文件与文件 F 的链接。在新文件中只包含被链接文件 F 的路径名，称这样的链接方法为符号链接；

7.6.3 文件的备份转储

通过转储技术，定期将全部或部分文件转存在磁带、光盘作为备份。常用的转储方法有两种：全量转储、增量转储。

全量转储：把文件系统中所有文件，定期复制在磁带上。

增量转储：仅把修改过的文件和新建立的文件转储在磁带上。

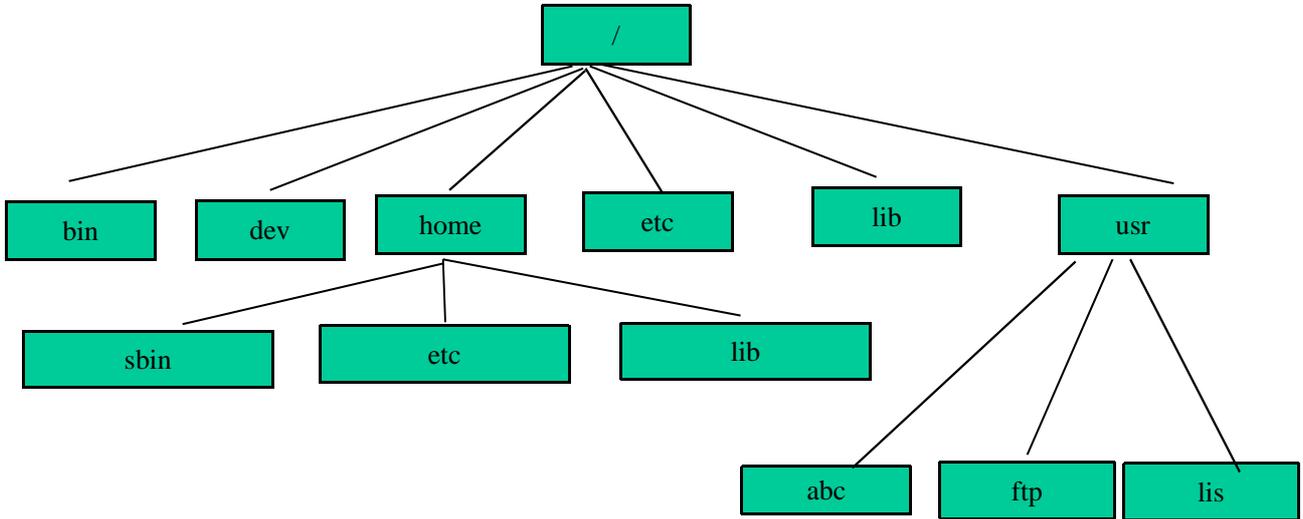
第 8 章 Linux 文件管理

本章主要介绍 EXT2 文件系统的结构特点以及 linux 虚拟文件系统 VFS 的特点及其实现技术。

8. 1Linux 文件系统概述

1. Linux 文件系统的树型结构

Linux 最上层的是根目录，用/表示。



Linux 采用目录分解的方法管理文件目录。在树型目录中的目录项是文件的符号目录。如下图所示：

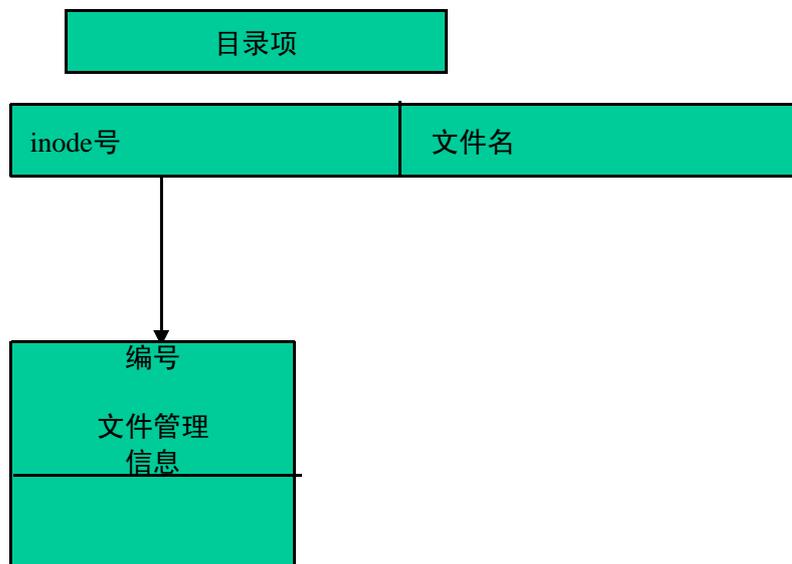


图 8.2 linux 文件系统的目录项

2. Linux 文件类型

- ◆ 普通文件
- ◆ 目录文件

- ◆ 设备文件
- ◆ 管道文件
- ◆ 链接文件（基于索引节点共享）

3. 文件的访问权限

Linux 中的每个文件都归于一个特定的用户所有，而且一个用户一般都是与某个用户组相关。所以，linux 设置了三种针对访问者身份的权限：文件所有者、与文件所有者同组的用户、其他用户。

对文件的访问限制主要体现在文件的 3 种操作上，即文件的读取、写入和执行。对 3 种访问者的 3 种操作限制形成了 9 种情况，用 9 位二进制代码表示。

```
-rwxr-xr-x 1 user wheel 3212 Dec 4 12:36 a.out
```

8. 2 EXT2 文件系统

8.2.1 EXT2 文件系统的构造

一个文件系统一般使用块设备上的一个独立逻辑分区，在这个分区内建立文件系统的树型层次结构。EXT2 文件系统由逻辑序列块序列组成。EXT2 文件系统把它所使用的磁盘逻辑分区划分为若干块组（block group），并从 0 开始依次编号。每个块组中包含若干数据块，其中存放文件内容。每个组中除数据块之外包括 5 种用于管理和控制的信息块：超级块、组描述符、块位图、inode 位图和 inode 表。这些信息位于每个块组的前部，后面是文件的数据块。

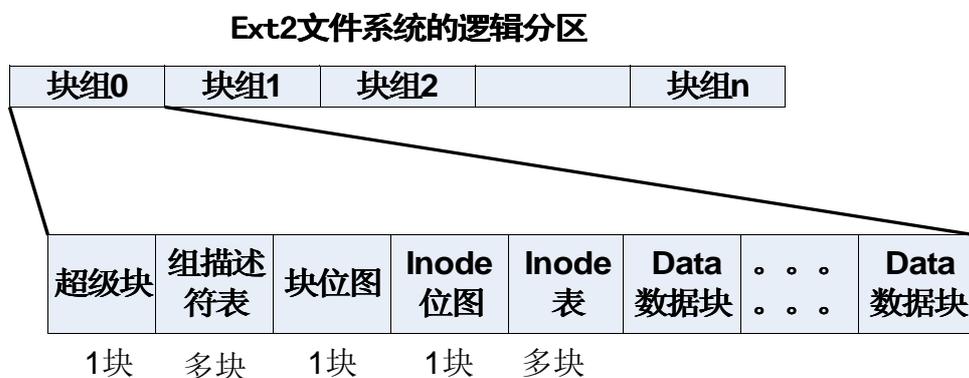


图8. 3 Ext2文件系统结构

8.2.2 EXT2 超级块（super block）

EXT2 超级块（super block）：用来描述 EXT2 文件系统整体信息的数据结构，主要描述文件在逻辑分区中的静态分布情况，以及描述文件系统的各种组成结构的尺寸、数量等。**所有块组中包含的超级块的内容是相同的。系统运行期间，需要把超级块中的内容复制到内存缓冲区内。**

在 linux 中，EXT2 文件系统超级块定义为 ext2_super_block 结构，定义在 /include/linux/ext2_fs.h 中。EXT2 超级块由两部分组成：

- 基本超级块：EXT2 文件系统得整体静态信息。
- 扩充块：反映所在块组的某些动态特性。

超级块本身占用一个物理块（1024B），基本块占 84B，扩充块占 20B，剩余的 920B 定义为元素长度为 4B 的数组 reserved[230]，作为备用。

	0	1	2	3	4	5	6	7
0	Inode数				块数			
8	保留块数				空闲块数			
16	空闲 Inode数				第一个数据块块号			
24	块长度				片长度			
32	每组块数				每组片数			
40	每组 inode数				安装时间			
48	最后写入时间				安装计数	最大安装数		
56	署名	状态		出错动作		改版标志		
64	最后检测时间				最大检测间隔			
72	操作系统				版本标志			
80	UID	GID						

图8.4 ext2 超级块

8.2.3 组描述符

组描述符：记录各个组块的描述信息（在位文件分配磁盘空间时需要使用这些信息）。这些组描述符集中在一起，就形成了组描述符表。组描述符可能占用多个物理块。和超级块一样的是：每个组块中的组描述符的内容完全相同，而且它的内容也要读入内存。

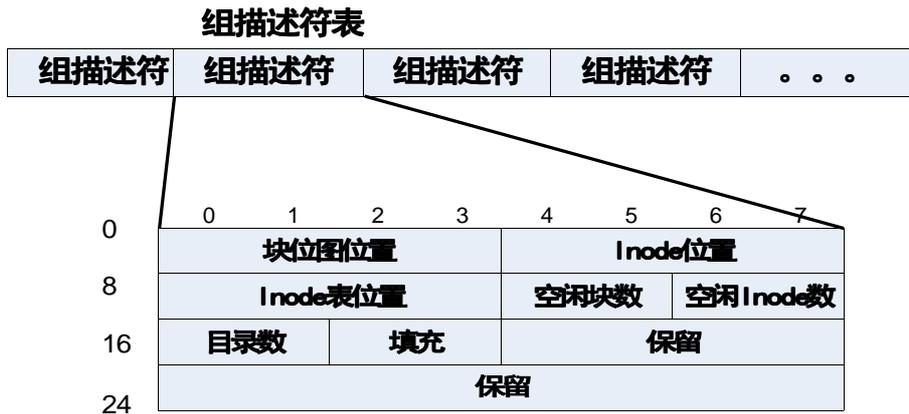


图8.5 ext2 组描述符的结构

在 linux 中，组描述符（32B）是一个 ext2_group_desc 结构，定义在/include/linux/ext2_fs.h 中：

8.2.4 块位图

块位图：记录每块组的数据块的使用情况。它占用一个物理块。

所以一个块组中数据块的最大数量是一个物理块的长度的 8 倍。例：对 1024 的物理块而言，其块位图就有 1024*8 位，即可以表示 8K 个数据块，也就是说一个块组的数据区的最大容量是 8M。如果 EXT2 文件系统使用的逻辑分区为 100M，则它可以划分为 12 个块组。

系统运行后，块位图装入一个高速缓存中，但是由于高速缓存空间有限，故只装入当前常用的 8 块（缺省值）位图。

8. 3 EXT2 的 inode 和文件结构

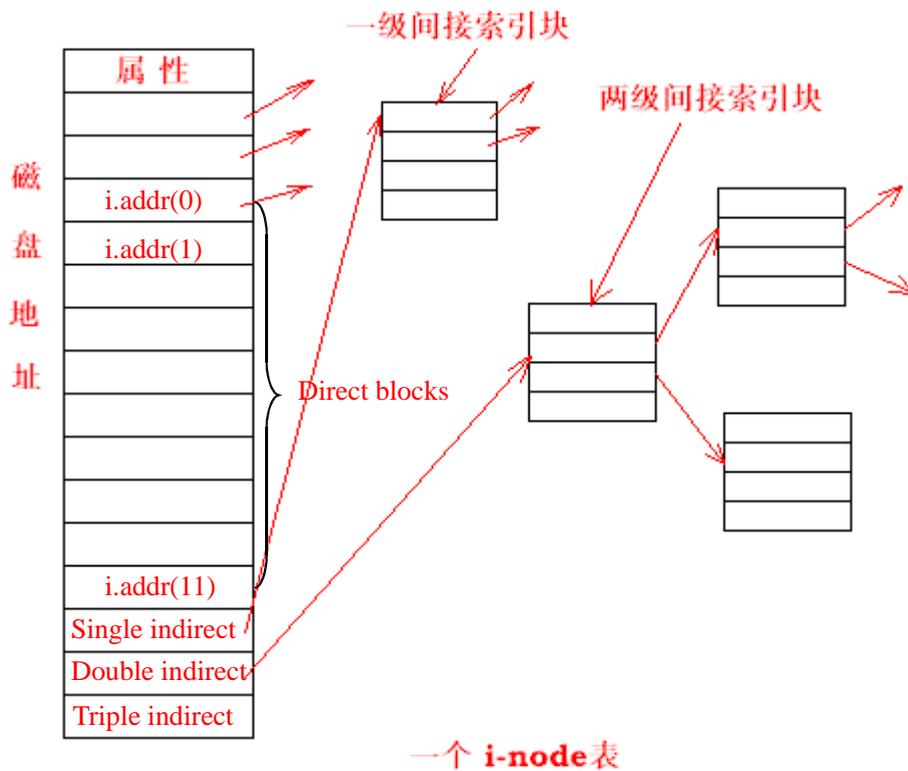
8.3.1 EXT2 文件系统 inode 结构

Linux 中 EXT2 文件系统 inode 定义为 struct ext2_inode。它定义在/include/linux/ext2_fs.h。

表 8.3 EXT2 文件系统 inode 结构的主要内容（参见汤子瀛）

8.3.2 inode 表和 inode 位图

inode 表：一个块组的所有文件的 inode 的集合。它可能占据多个物理块。每个块组可以包含的 inode 的数目



由超级块中的成员项 `s_inoders_per_group` 给出。

Inode 位图：反映了 inode 表中各个项的使用情况，它的每一位表示 inode 表的一个表项。1 使用；0 空闲。与块位图相似，inode 位图也装入一个高速缓冲中。

8.3.3 EXT2 文件的物理结构

采用混合索引结构。Inode 中 `i_block[]` 数组共有 15 个地址。如下图所示：

EXT2 文件默认的物理块尺寸为 1K，EXT2 的块地址长度 4B，所以每个间接块中的索引表可以包括 $1024/4=256$ 个物理地址。所以

1. 直接地址：允许文件不大于 12K
2. 一次间接地址：当文件大于 12K 时采用，允许文件长达 $256K+12K$
3. 二次间接地址：当文件大于 $256K+12K$ 时采用，允许文件长达 $256*256K+256K+12K$
4. 三次间接地址：当文件大于 $256*256K+256K+12K$ 时采用，允许文件长达 $256*256*256K+256*256K+256K+12K=16G+64M+256K+12K$

但是实际上 linux 是 32 位文件系统，文件尺寸最大为 4G。

EXT2 文件系统按照文件的逻辑块号为索引值查找数据块，逻辑块从 0 依次编号。

例：如何查找逻辑块号 100 对应的物理块。

8.3.4 EXT2 的目录结构

Linux 树型目录结构中，每个文件目录都是一个目录文件，每个目录项都是一个 `ext2_dir_entry` 结构体，它就是一个文件的符号目录。定义在 `/include/linux/ext2_fs.h`。

Struct `ext2_dir_entry`

```

{
    _u32 inode;           /*inode 号*/
    _u16 rec_len;        /*目录项长度*/
    _u16 name_len;       /*文件名长度*/
    Char name[EXT2_NAME_LEN]; /*文件名*/
}

```

EXT2_NAME_LEN 缺省为 255，也就是文件名最大可以用 255 个字符。另外目录项长度根据文件名长度的大小是可变的。但是必须是 4 的倍数，不用部分用\0 填充。如图 8.7 所示。

删除文件时，将相应的 inode 号字段置 0，如果相邻有空白目录项则合并。

添加文件时，找到一个长度合适的空白目录项，并写入相应信息；若空白表项使用后剩余空间大于 12B，则把剩余部分仍作为空白目录项。如果找不到合适的空白目录项，就在文件尾部建立这个文件的目录项。

8. 4 虚拟文件系统 VFS

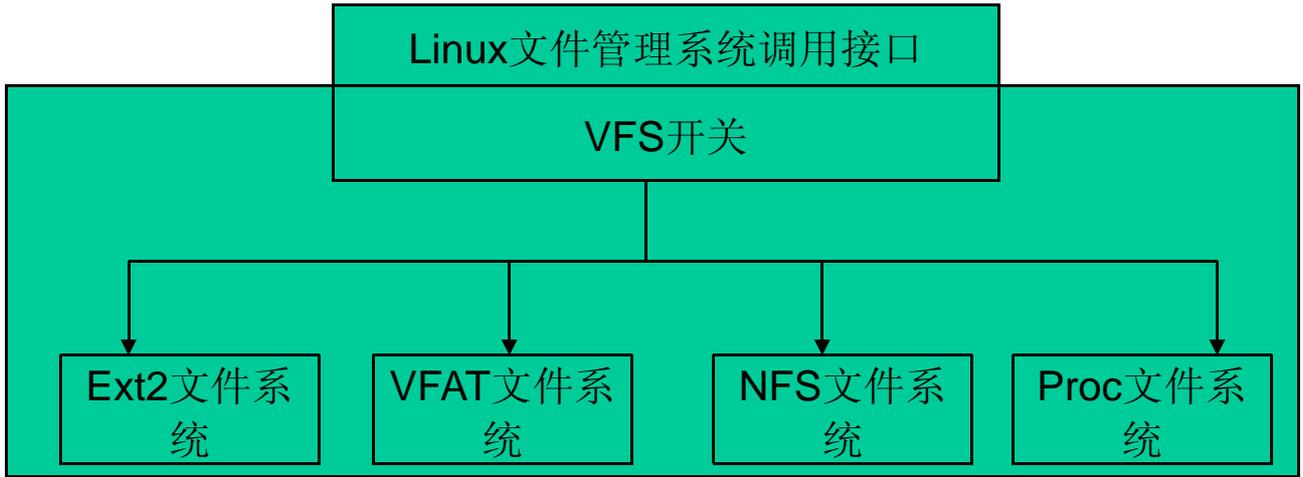
linux 支持的文件系统有;Minix、ext2、iso、nfs、smb、msdos、romfs 等多达 20 几种。为什么能支持这么多的文件系统？因为它引进了虚拟文件转换技术 VFS。Vfs 屏蔽了各种文件系统的差异，为处理各种不同文件系统提供了统一的接口，在 vfs 的管理下，linux 能访问各种文件系统，而且实现了各种文件系统之间的互访

8.4.1 VFS 的工作原理

1. 虚拟转换机制（VFS）

- **物理文件系统：**Linux 支持的各种文件系统：如 Minix、ext2、iso、nfs 等被称为物理文件系统；
- **虚拟转换机制（VFS）：**不同的物理文件系统有不同的组织结构和不同的处理方式，为了能处理各种不同的物理文件系统，操作系统需要把它们特性进行抽象，把各种不同物理文件系统转换为一个具有统一共性的虚拟文件系统，这种转换机制称为虚拟文件系统转换，即 vfs。

VFS 不是实际的文件系统，它进程提供了处理各种物理文件系统的公共接口。通过这个接口使得不同的文件系统看来都是相同的。



2. VFS 超级块和 VFS inode 结构

- VFS 超级块的作用是把在各种文件系统中的表示文件在逻辑分区中的静态分布情况转换成统一的格式。
- VFS inode 作用是不同文件系统的活动文件的管理信息，如文件类型、文件尺寸等转换成统一格式。

3. 公共操作函数接口

当进程向系统发出文件操作请求时，该文件可能是某个物理文件系统中的文件，内核将通过 VFS 公共操作函数接口转换到该文件系统的相应的操作函数。

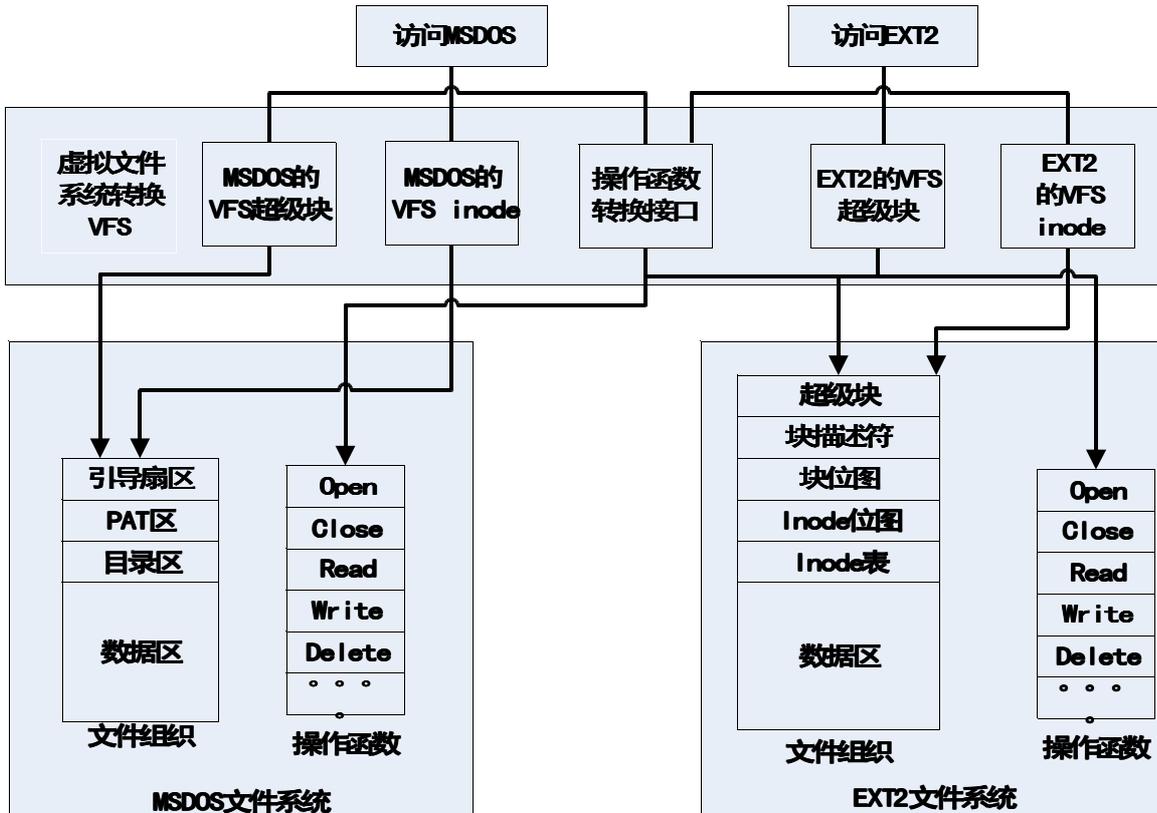


图8.9 VFS 实现文件系统的转换

8.4.2 VFS 超级块

VFS 超级块是在文件系统安装是由系统在内存建立的，对于每一种已安装的文件系统，在内存中都有与其对应的 VFS 超级块。各种文件系统的超级块都是一个 `super_block` 结构体。它里的数据是在安装时，由读超级块例程 `read_super()` 把某种文件系统的管理信息写入它的 VFS 超级块中。

VFS 超级块主要包括以下几种信息：

1. 文件系统的组织信息。如文件系统所在的设备号、块大小、块位数、文件系统署名等。设备号包括主设备号和次设备号，如 `/dev/hda1`，设备号是 `0x0301`，其中 `03` 是主设备号，`01` 是次设备号。
2. 文件系统的注册和安装信息。
3. 超级块的属性信息，表现为超级块的各种标志，如超级块标志、锁定标志、禁写标志、修改标志等
4. VFS 超级块的前面各个成员项表示的是各种文件系统的共信息，不同文件系统的特有信息则由联合体 `u` 的各个成员表示。
5. 指向对超级块进行操作的函数指针。

8.4.3 VFS 的 inode

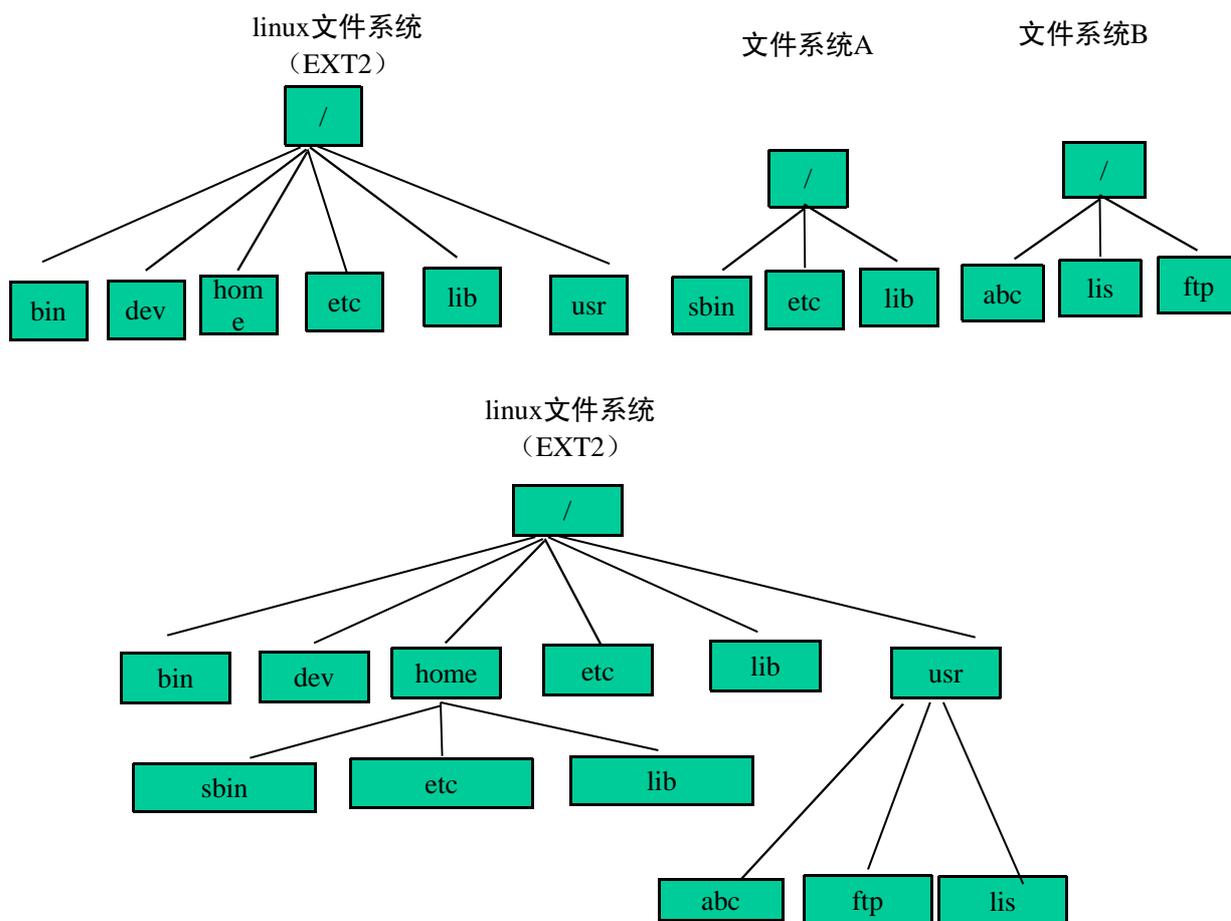
VFS inode 作用是把不同文件系统的活动文件的管理信息，如文件类型、文件尺寸等转换成统一格式。只有当前正在使用的活动文件才有 VFS inode。详细信息请自行查阅参考书。

8. 5 文件系统的安装与注册

8.5.1 文件系统的安装

1. 安装点

安装点：linux 文件系统的树型层次结构中用于安装其它文件系统的目录称为安装点或安装目录。



超级用户可以 通过下列命令来安装文件系统:

```
$mount -t msdos /dev/hdc /mnt/usr
```

Msdos: 文件系统类型

/dev/hdc: 文件系统所在设备

/mnt/usr: 安装点

卸载文件系统命令:

```
$umount dev/hdc 或 $umount /mnt/usr
```

2. 物理文件系统链表

Linux 对系统中已安装的每种物理文件系统用一个 `vfsmount` 结构进行描述，其定义在 `/include/linux/mount.h`。

8.5.2 文件系统的注册

文件系统在安装后，为了能让 linux 系统对各种物理文件进行管理，物理文件系统在安装后必须向系统内核注册。

两种方式:

- ◆ 编译系统内核时确定可以支持哪些文件系统，在系统引导时注册

◆ 系统运行中需要使用某种文件系统时进行安装并注册。

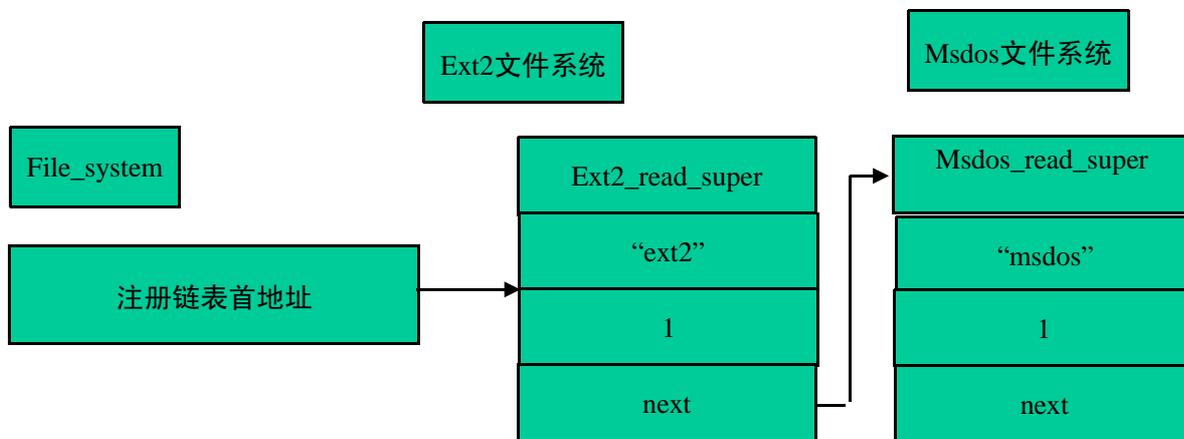
1. 注册链表

系统中所有已注册的文件系统登记在 `file_system_type` 中，组成一个链表。

```
struct file_system_type
{
    const char *name; /*指向文件系统文件系统名*/
    struct super_block *(*read_super) (struct super_block *, void *, int); /*函数指针，函数功能：在文
        件系统安装时，从外存读取该文件系统的有关数据写入 VFS 中。*/
    int requires_dev; /*是否需要设备支持：1 需要；0 不需要*/
    struct file_system_type * next; /*下一个节点*/
};
```

Linux 支持的各种物理文件系统的注册数据预先设置在它们各自的注册结构体中。

图 8.12 三种文件系统的注册链表



2. 文件系统的注册

各种文件系统的注册是通过内核提供的文件系统初始化函数实现的，如：

```
Init_ext2_fs();
```

```
Init_msdos_fs();
```

在各个文件系统初始化函数中，把文件系统的注册结构体作为参数，调用内核提供的文件注册函数 `register_filesystem()`，把文件系统注册结构体加入到注册链表中，从而完成注册功能。

8.6 文件管理和操作

8.6.1 系统对文件的管理

Linux 系统把所有打开的活动文件进行统一管理，组成“系统打开文件表”。系统打开文件表是一个双向链表，每个表项是一个 `file` 结构体，存放着活动文件的控制信息。其定义在 `/include/linux/fs.h`。

Struct file

```
{
    Mode_t f_mode; /*文件的打开模式*/
```

```

Loff_t f_pos;                /*文件的当前读写位置*/
Unsigned short f_flags;      /*文件操作标志*/
Unsigned short f_count;      /*共享该结构体的计数值*/
Unsigned long f_reada,f_ramax,f_raend,f_ralen,f_rawin;
Struct file *f_next,*f_prev; /*连接前后节点的指针*/
Struct fown_struct f_owner;
Struct inode * f_inode;     /*文件对应的 inode*/
Struct file_operation *f_op; /*指向文件操作结构体的指针*/
Unsigned long f_version;    /*文件版本*/
Void *private_data;        /*指向与文件管理模块有关的私有数据的指针*/
}

```

8.6.2 进程对文件的管理

1. 数据结构

对于进程打开的文件，由进程的两个私有结构进行管理：

- fs_struct 记录着文件系统根目录和当前目录。
- files_struct 包含着进程的打开文件表。

```

struct fs_struct
{
    atomic_t count;                /*共享此结构的计数值*/
    int umask;                      /*文件掩码*/
    struct dentry * root, * pwd;    /*根目录和当前目录的 inode 指针*/
};

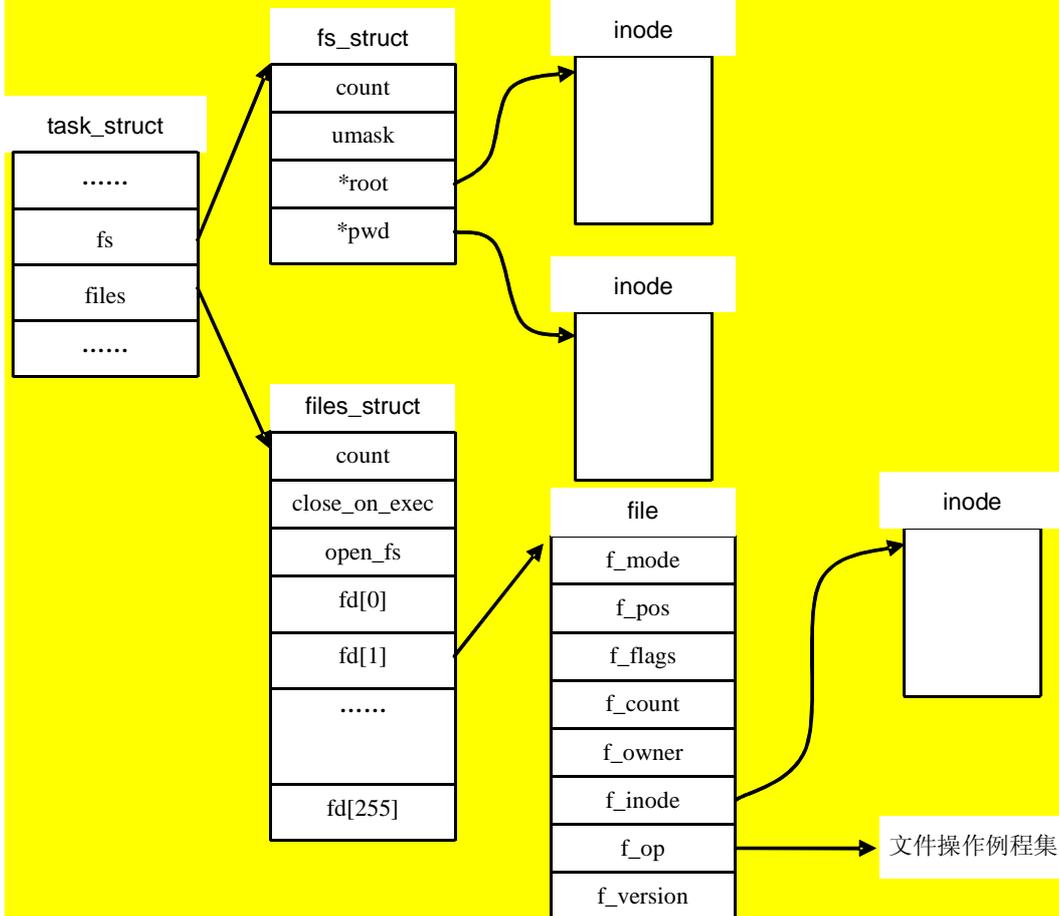
struct files_struct
{
    int count;                      /*共享计数值*/
    fd_set *close_on_exec;
    fd_set *open_fds;
    struct file * fd [NR_OPEN]; /*进程打开文件表，有 256 个元素，表示最多可以同时打开 256 个文件*/
};

```

说明：

- 当进程打开一个文件时，建立一个 file 结构体，并加入到系统打开文件表中，然后把 file 结构体的首地址写入 fd[] 数组的一个空闲元素中。
- linux 中把 fd[] 数组的下标作为一个进程的活动文件标志，称为进程标识符。

2. 当前进程要处理某个打开文件时，访问各个数据结构的步骤：



8.6.3 文件操作函数

```

struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);

```

```

unsigned int (*poll) (struct file *, struct poll_table_struct *);
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, struct dentry *, int 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 (*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);
#ifdef MAGIC_ROM_PTR
int (*romptr) (struct file *, struct vm_area_struct *);
#endif /* MAGIC_ROM_PTR */
};

```

- lseek - 移动文件指针的位置，只用于可以随机存取的设备
- read - 从字符设备读数据
- Write - 向字符设备写数据
- Open-打开设备，并初始化设备
- Release-关闭设备，并释放资源

第9章 设备管理

9.1 设备与设备管理

9.1.1 设备的分类

按从属关系分

- 系统设备：Os 生成时已登记在系统中的标准设备，如键盘、显示器等。
- 用户设备：Os 生成时未登记在系统中的标准设备，如绘图仪、扫描仪等。

按设备的资源属性分

- 独占设备：打印机（大多数低速设备）
- 共享设备：磁盘
- 虚拟设备：独占设备——>共享设备

虚拟设备：指通过虚拟技术将一台独占设备变换为若干台逻辑设备，供若干用户进程同时使用。通常

把这种经过虚拟技术处理后的设备称为虚拟设备。例如 SPOOLING 技术。

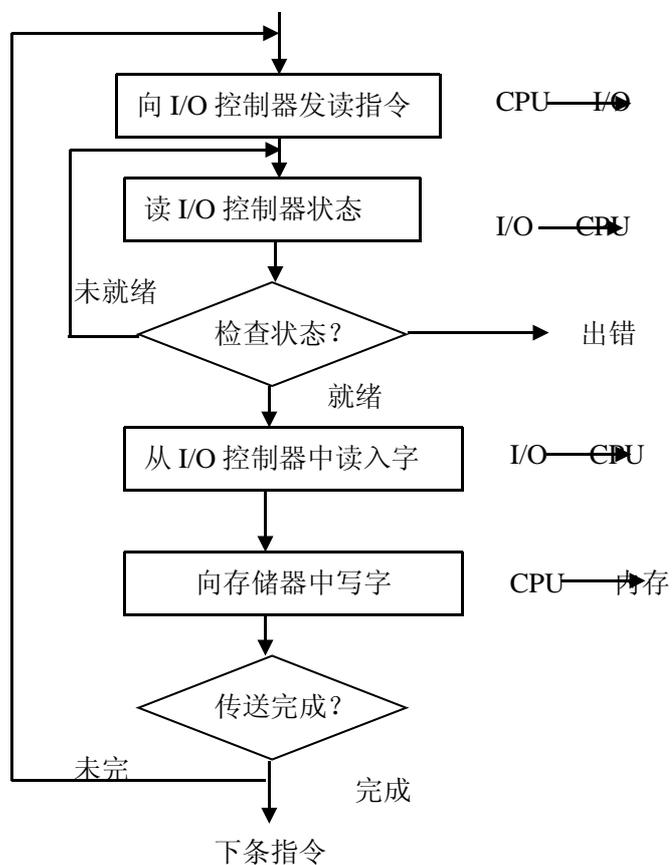
9.1.2 设备管理的目标

9.1.3 设备管理的功能

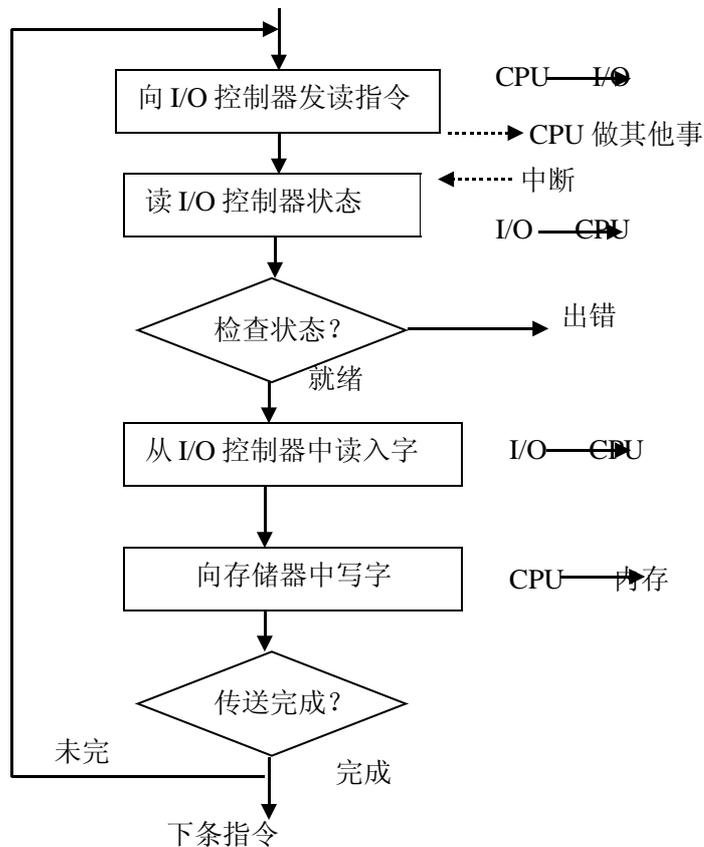
9. 2I/O 控制方式

9.2.1 CPU 控制方式

1) 程序直接控制方式——CPU 一直对 I/O 控制干预



2) 中断控制方式——CPU 以字节为单位对 I/O 控制干预



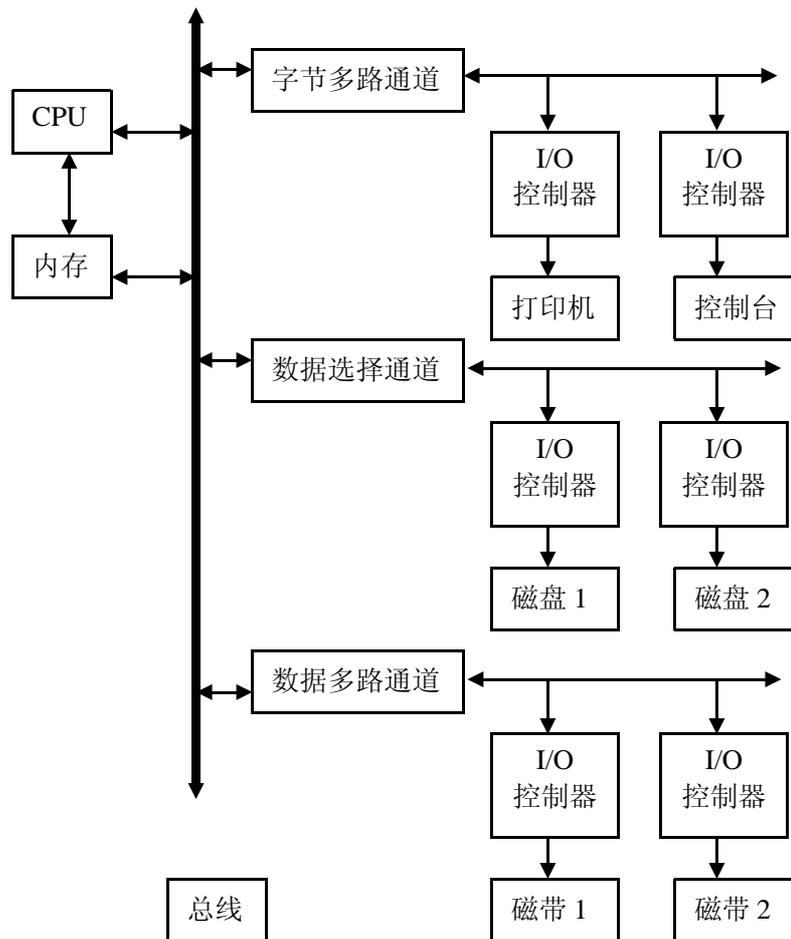
3) DMA 方式——CPU 以一组数据块干预 I/O 控制

DMA 方式又称直接存取 (Direct Memory Access) 方式。其基本思想是在外围设备和内存之间开辟直接的数据交换通道。在 DMA 方式中, I/O 控制器具有比中断方式和程序直接控制方式更强的功能。另外除控制状态寄存器和数据缓冲寄存器之外, DMA 控制器中还包括传送字节计数器、内存地址寄存器等。这是因为 DMA 方式窃取或挪用 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中的缘故。

9.2.2 通道控制方式

1. **I/O 通道:** 一种硬件机制, 指专门用于 I/O 工作的处理机, 它由自己的简单的与 I/O 操作相关的指令系统, 如数据传输、设备控制等。通道执行的程序为通道程序。

2. 通道方式的处理过程：CPU 向通道发出一条 I/O 指令，通道接收到指令后，从内存中取出本次要执行的通道程序，然后执行该通道程序，仅当通道完成了规定的 I/O 任务后，才向 CPU 发出中断信号（可以提高 CPU 与 I/O 的并行处理能力）



3. 通道的分类，按信息交换的方式，分为三类：

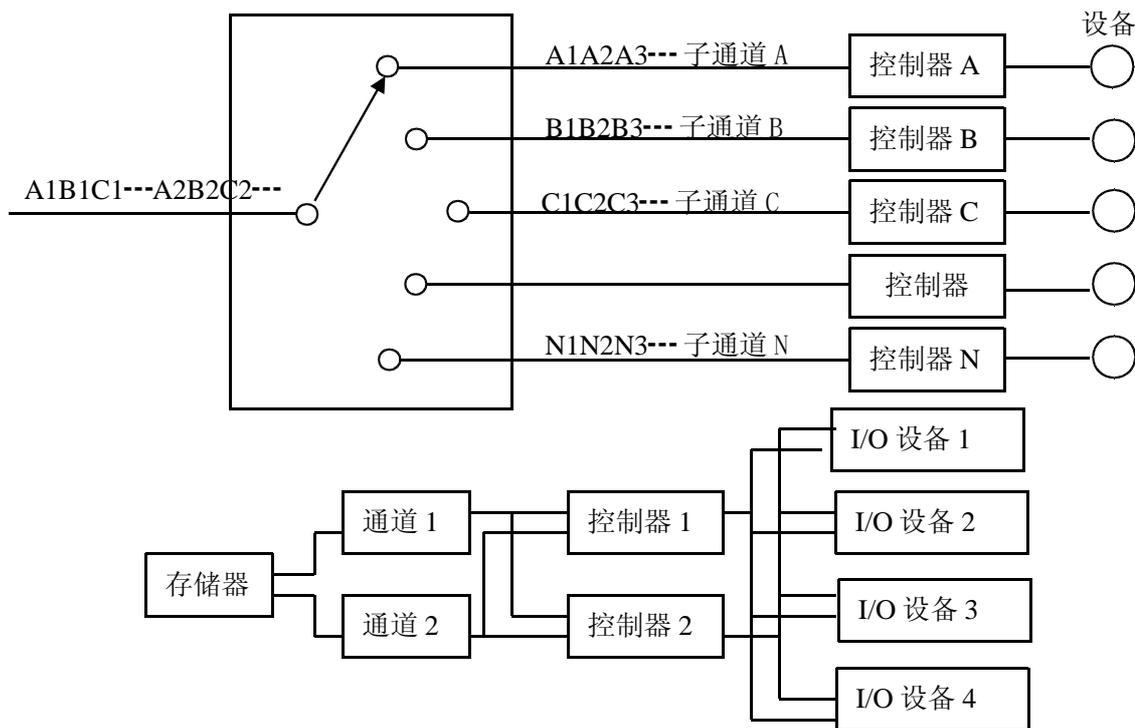
- a. 字节多路通道（中低速）（分时）：字节为单位
- b. 数据选择通道（高速）：块为单位
- c. 数组多路通道（高中速）（分时）：块为单位

4. 瓶颈问题

引起原因——通道不足

措施：

- a. 提高 I/O 设备的独立性，以减少占用通道的时间，提高 I/O 设备的独立性，最常用的方法是在设备和控制器增设缓冲
- b. 增加通路，增加设备道主机之间的通路，是解决“瓶颈”问题的最有效的方法，即可把一个设备连到多个控制器上，而一个控制器又被连到多个通道上。如下图：



9. 3 缓冲技术

9.3.1 缓冲技术的引进

1、CPU 与 I/O 设备间速度不匹配

2、减少 CPU 的中断频率

(防止每接收一次数据就中断一次, 将数据先放到缓冲区, 待满时, 再中断 CPU, 显然减少了中断次数)

3、提高 CPU 和 I/O 设备的并行性

(例如: 打印机工作时, CPU 继续进行计算工作)

9.3.2 缓冲的种类

1. 缓冲的实现方式

1) 硬件方式

一般是采用专用硬件缓冲器, 如 I/O 控制器中的数据缓冲寄存器等。(成本比较高, 除关键部件外, 一般不采用);

2) 软件方式

在内存中开辟出一个具有 N 个单元的专用缓冲区, 以便存放输入输出数据。

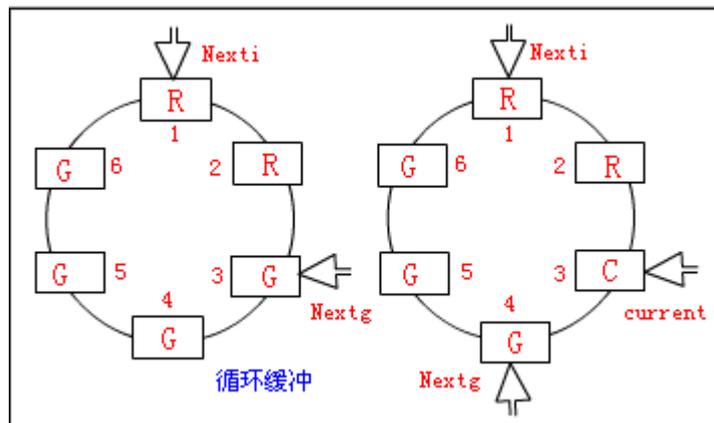
操作系统为每一个缓冲区建立一个数据结构, 称为缓存控制块 BCB (buffer control block)。操作系统通过 BCB 对每一个缓存实施具体的管理。

2. 缓冲的使用方式:

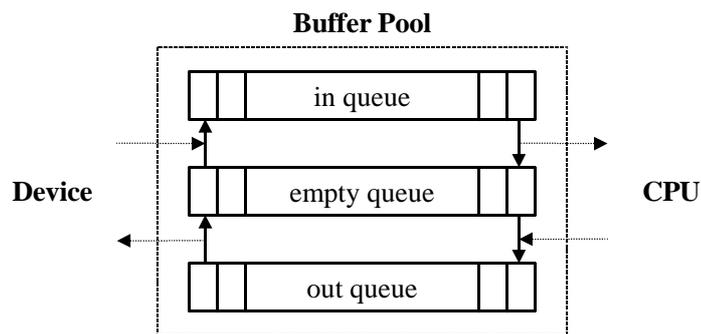
- ✓ 专用缓冲: 为某设备/进程专门设置的
- ✓ 公用缓冲: 为所有的设备/进程设置的, 为所有的设备/进程所共享

3. 缓冲的组织方式:

- ✓ 单缓冲
- ✓ 双缓冲
- ✓ 循环缓冲



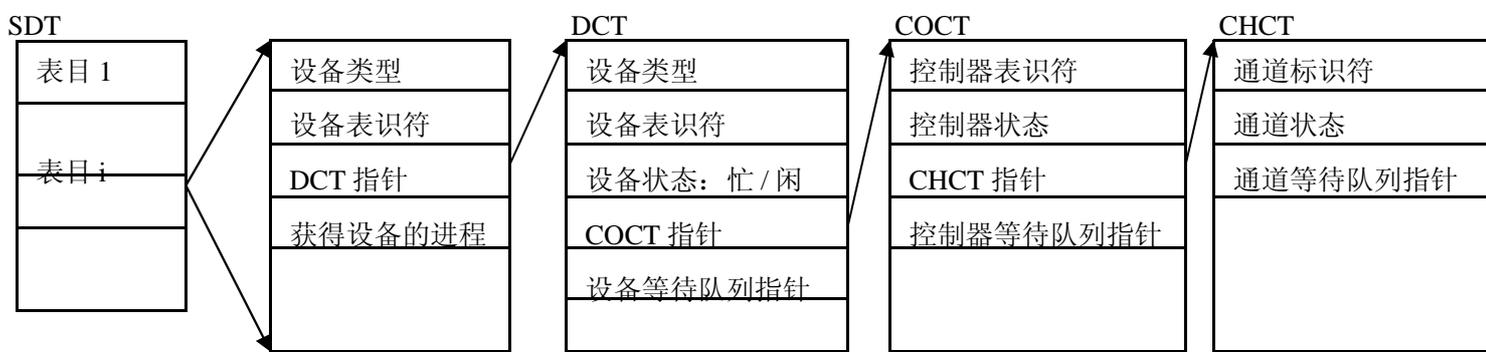
- ✓ 缓冲池:



9. 4 设备的分配

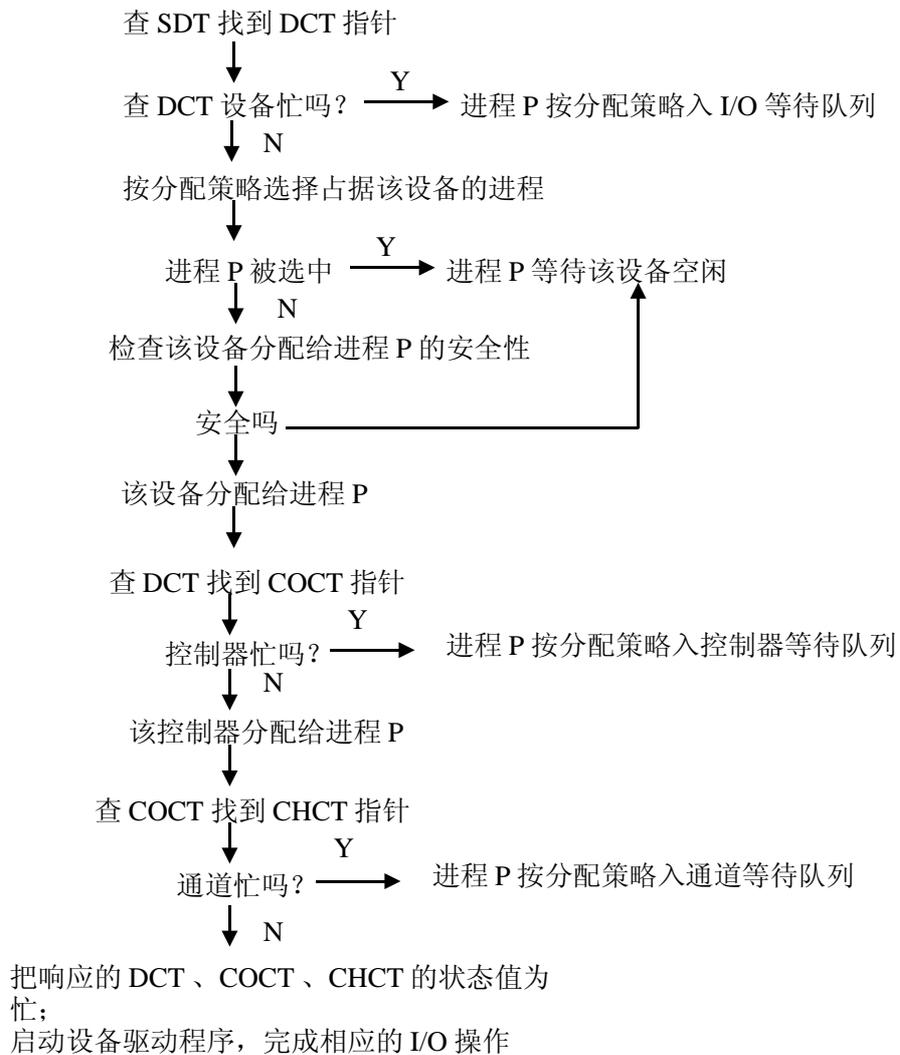
9.4.1 设备管理的数据结构

1. 设备控制表 DCT
2. 控制器控制表 COCT
3. 通道控制表 CHCT
4. 系统设备表 SDT: 记录已被连接到系统中的所有物理设备的情况, 每个物理设备占一个表目。



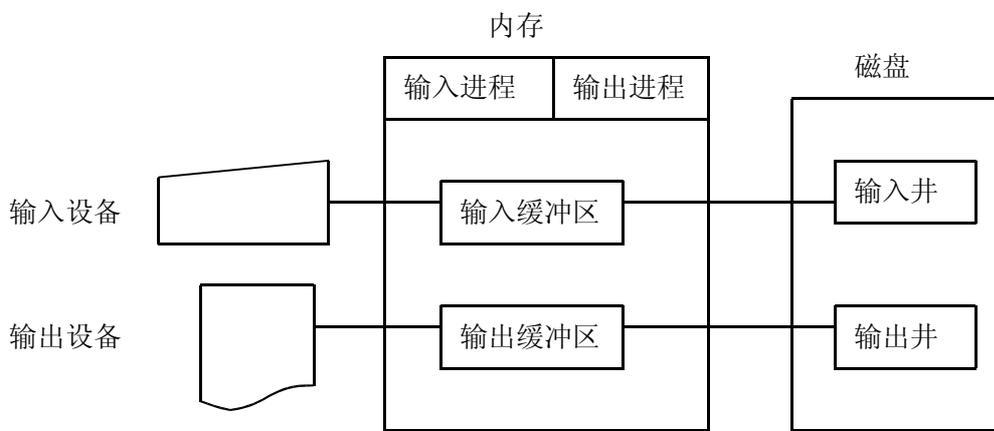
9.4.2 设备分配策略

1. 独占分配方式
2. 共享分配方式（同时分配给多个作业使用，很复杂）



3. 虚拟分配方式

虚拟技术 **SPOOLING** 技术：又称假脱机 I/O 操作，是对脱机输入输出工作的模拟，它需要缓冲技术支持并必须有高速、大容量且可随机存取的外存支持。可将一个独占设备虚拟成多台设备。



9.4.3 设备分配算法

1. 先请求先服务 (FCFS)
2. 优先级高者优先

9.5 设备处理程序与 I/O 进程

设备处理程序包括驱动程序和 I/O 中断处理程序。I/O 处理程序通常是由 I/O 进程完成的。

9.5.1 设备处理程序

设备处理程序是负责直接控制设备完成实际的 I/O 操作的程序，设备驱动程序直接和硬件设备打交道。设备驱动程序包括对设备的各种操作，在操作系统的控制下，cpu 通过执行驱动程序来实现对设备底层硬件设备的处理和操作。

9.5.2 I/O 进程

I/O 进程：为了能够在处理机上执行设备处理程序，当前许多操作系统都设置了专门完成 I/O 操作的进程，属于系统进程。在不同的操作系统中，处理 I/O 操作和配置 I/O 进程的方式不同，大体可以分为三种：

- 为每一类设备设置一个进程，它专门执行这类设备的 I/O 操作。
- 在整个系统中设置一个 I/O 进程，专门负责对系统中所有各类设备的 I/O 操作。也可以设置一个输入进程和一个输出进程，分别负责处理系统中所有的各类设备的输入或输出操作。
- 系统为每台设备建立一个 I/O 进程，它们分别执行设备各自的处理程序。

当前操作系统多数采用 (2) 方式。

I/O 进程一般在系统生成时被创建，平时处于睡眠等待状态。I/O 进程在两种情况下被唤醒：

- 当用进程发出设备请求 (I/O 请求)。
- 出现 I/O 中断。

第 10 章 Linux 设备管理

10. 1Linux 设备分类与识别

10.1.1 Linux 设备的分类

- 字符设备：以字符为单位输入输出数据的设备，并且以字符为单位对设备中的信息进行组织和处理。显示器、键盘等
- 块设备：以一定大小的数据块为单位输入输出数据，并且设备中的数据也是以物理块为单位进行组织和管理的。硬盘、软盘、光盘等。
- 网络设备：通过网络与外部进程或远程计算机进行通信的设备。网卡

10.1.2 设备文件

linux 设备管理的特点：物理设备抽象化，把物理设备看成文件，采用文件系统的接口和系统调用来管理和控制设备。Linux 设备就是一种特殊文件，称为设备文件。

1. 从设备向内存输入数据，相当于从设备文件读取数据；把数据从内存输出到设备可以看作是数据写入设备文件；启动设备时可以看作是打开设备文件；停止设备可看作是关闭设备文件。
2. Linux 的设备文件一般置于/dev 目录下。系统中每个设备文件都有设备文件名。设备文件名由两个部分组成：
 - 第一部分 2~3 个字符，表示设备的种类；
 - 第二部分通常是字母或数字，区分同种设备中的单个设备
 - IDE 普通硬盘是以 hd 命名；第一个 ide 设备是 hda，第二个是 hdb...；而 hda1, hda2 表示第一块硬盘的第一、第二个磁盘分区。每个硬盘可以最多有四个主分区，因此 1-4 命名硬盘的主分区。逻辑分区是从 5 开始的，每多一个分区，数字加 1 就可以。
 - SCSI 硬盘是用 sd 命名
 - 软盘是用 fs 命名
3. Linux 设备中有一个特殊的设备：null 设备。通常称其为“黑洞”设备，并没有实体与之对应。向 null 设备输出的一切数据都被舍弃。（相当于“回收站”）
4. 在终端使用 `ls -l /dev` 查看设备列表
5. **逻辑设备名（设备类型，面向进程），物理设备名（设备文件名，面向内核），设备无关性。**
6. 设备文件与普通文件的差异
 - 设备文件没有象普通文件那样的文件实体，不在外存占据数据块来存放数据
 - 进程访问普通文件是读写磁盘分区中的数据，访问设备文件是对硬件设备进行读写，完成设备与内存之间的数据传送

7. 网络设备并不与设备文件对应，故网络设备没有设备文件名，只有逻辑设备名。如系统的第一块以太网卡的逻辑设备名是 eth0。Linux 文件系统不能用来管理和控制网络设备。

10.1.3 Linux 设备的识别

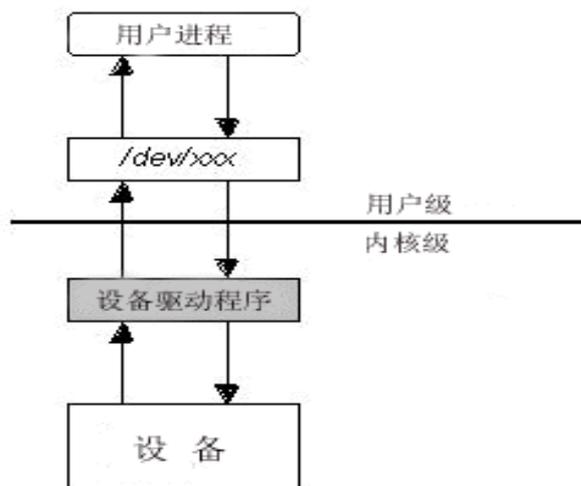
Linux 内核对设备的识别是通过：设备类型+设备号(主、次设备号)。

- 设备类型：字符设备、块设备
- 主设备号：使用同一个驱动程序的每种设备有一个唯一的主设备号
- 次设备号：区分同种设备中的各个具体设备

主次设备号值都是从 0~255

10. 2 设备驱动程序与设备注册

10.2.1 设备驱动程序



1. 功能：

- ◆ 对设备进行初始化
- ◆ 启动、停止设备的运行
- ◆ 把设备上的数据传到内存
- ◆ 把数据从内存传送到设备
- ◆ 检测设备状态

2. 驱动程序虽然是在设备生产厂家开发的，但装入系统后由内核统一管理，处于内核态，成为内核的一部分。

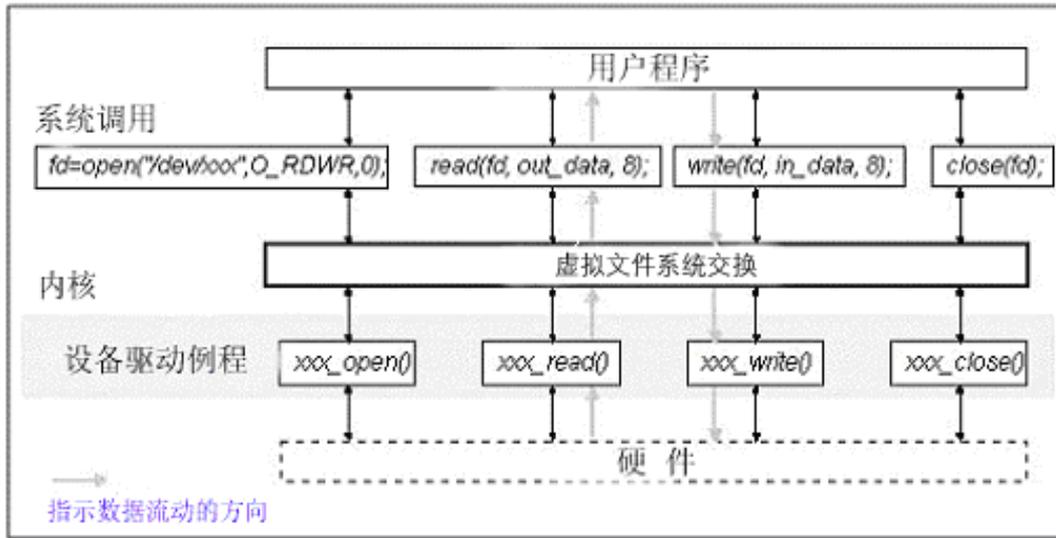
3. Linux 对设备的管理和控制是使用 VFS 提供的各种数据结构和操作函数实现的。

4. 驱动程序的编制:

Linux 中对文件的操作使用的是 VFS 虚拟文件系统的文件操作接口，即 `file_operations` 结构。`file_operations` 结构是文件操作函数指针的集合，在设备管理中，该结构体各个成员项指向的操作函数就是设备驱动程序的各个操作例程。

编制设备驱动程序的工作就是使用汇编或 c 语言编写控制设备完成各种操作的例程，然后把这些操作例程的入口地址赋予 `file_operations` 结构体的有关成员项即可。

各个操作函数的主要功能:



10.2.2 设备注册

当一种设备安装到系统时，必须向系统进行注册，注册之后才能使用，设备注册的任务是把驱动程序加载到系统中。

- 设备的驱动程序是系统在启动时装载到系统中的；
- 对于“即装即用”设备，驱动程序作为程序模块可以随时加载到系统中，

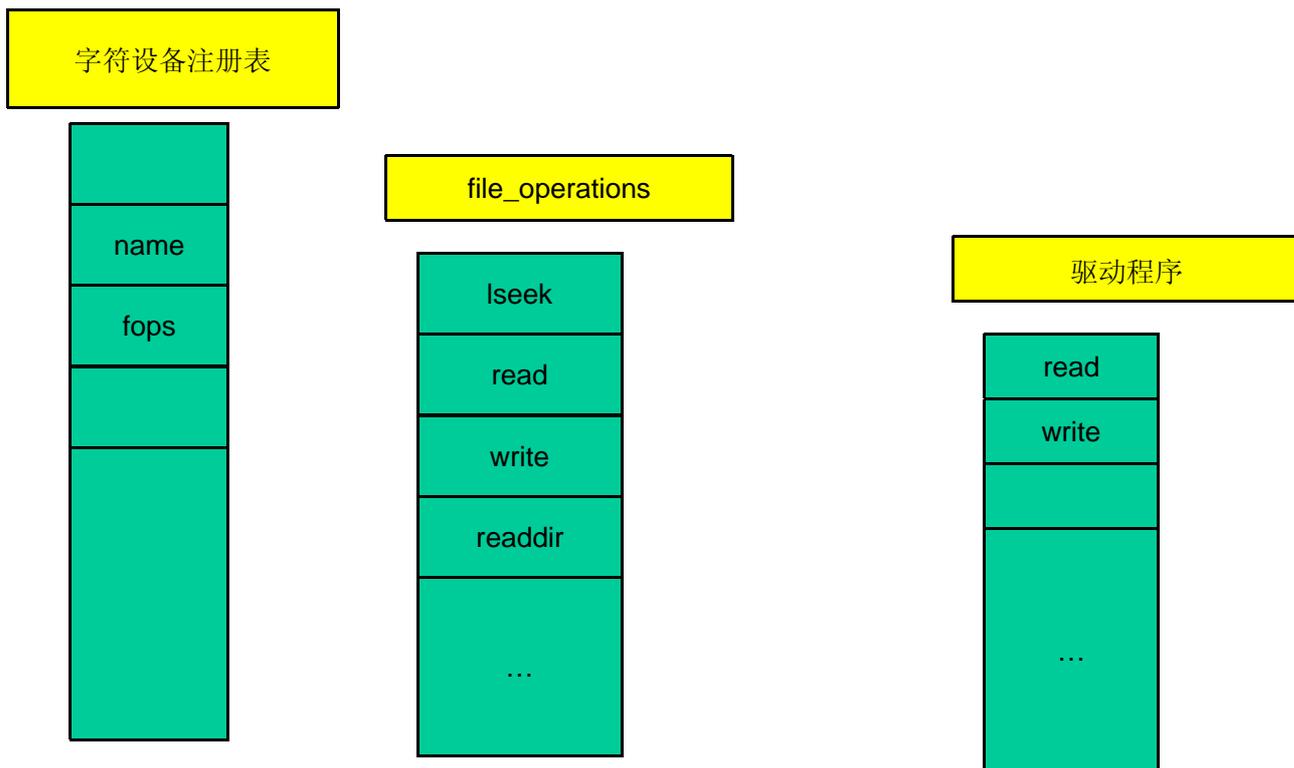
1. 设备注册表

两个设备注册表：字符设备注册表、块设备注册表。

```
#define MAX_CHRDEV 255
#define MAX_BLKDEV 255
static struct device_struct chrdevs[MAX_CHRDEV];
static struct device_struct blkdevs[MAX_BLKDEV];
```

每个注册表都有 255 个表项，每个表项表示一个设备，都是一个 `device_struct` 结构，称为设备描述符。在 `fs/device.c` 中。

```
struct device_struct {
    const char * name;          /*指向设备名字符串*/
    struct file_operations * fops; /*指向文件操作函数 file_operations 的指针*/
};
```



说明:

- 设备注册表的下标是某种设备的主设备号。使用主设备号作为索引就可以从设备注册表得到这种设备的驱动程序。
- 两个注册表的第一个表项通常为 null，因为系统中不存在主设备号为 0 的字符设备和块设备。

2. 设备注册函数

设备被注册时，系统首先构筑 `file_operations` 结构体，然后把驱动程序各个操作函数的入口地址赋予结构体的有关成员项，成员项没有对应函数时，则置为 `null`。设备注册是通过系统调用注册函数实现的。在 `fs/device.c` 中。

字符设备注册函数: `int register_chrdev(unsigned int major, const char * name, struct file_operations *fops)`

块设备注册函数: `int register_blkdev(unsigned int major, const char * name, struct file_operations *fops)`

- 若 `major=0`，注册一个新设备，由系统自动分配一个主设备号给驱动程序，成功则返回主设备号。
- 若 `major!=0`，此时注册函数可以变更主设备号为 `major` 的设备的：设备名或驱动程序。成功返回 0；

`if (major == 0)`

{

`for (major = MAX_CHRDEV-1; major > 0; major--)`

`if (chrdevs[major].fops == NULL) /*从注册表 chrdevs[]的底部开始，向上搜寻一个空表项。`

{

`chrdevs[major].name = name;`

`chrdevs[major].fops = fops;`

`write_unlock(&chrdevs_lock);`

`return major;`

}

}

`write_unlock(&chrdevs_lock);`

```
    return -EBUSY;
```

```
}
```

3. 设备注销

当设备需要撤销时，可以使用**注销函数**从设备注册表中删除。

```
int unregister_blkdev(unsigned int major, const char * name) /* 块设备注销函数*/
```

```
int unregister_chrdev(unsigned int major, const char * name) /* 字符设备注销函数*/
```

```
{
```

```
    if (strcmp(chrdevs[major].name, name))    return -EINVAL;
```

```
    chrdevs[major].name = NULL;
```

```
    chrdevs[major].fops = NULL;
```

```
    return 0;
```

```
}
```

10. 3Linux 的 I/O 控制方式

Linux 对设备的输入、输出过程实际上是在 cpu 的控制下主机（内存）与外部设备之间传送数据的过程。所以 Linux 的 I/O 控制方式有 3 种：

- 查询等待方式
- 中断方式
- DMA 方式

10.3.1 查询等待方式（轮询方式）

驱动程序不断检测设备状态，当设备准备好传送数据时，cpu 执行驱动完成一次 I/O 过程，若设备未转备好，则驱动程序反复检测设备状态，直到设备转备好。

适用于：不支持设备中断的系统、系统支持的中断数目有限时；例如并行接口（打印机接口）的驱动程序中默认的控制方式就是轮询方式。

10.3.2 中断方式

当进程向设备提出 I/O 请求时并不等待设备完成 I/O 操作，而是把 cpu 让给其它进程使用，自己则进入睡眠状态。在设备完成 I/O 操作时发出中断信号，系统根据中断信号调用相应的中断服务唤醒等待的进程继续执行后面的操作。

在机器硬件支持中断的情况下，设备驱动程序就可以使用中断方式控制设备的 I/O 操作。因此，设备驱动程序中处理包含各种操作函数外，同时还要提供进行各种中断处理的中断服务例程。CPU 接受到来自硬件的中断请求后，则通过中断请求好就能够执行该设备驱动程序的中断服务例程。

- 中断服务例程

- **中断服务例程描述表:** Linux 中, 各种中断服务例程是通过**中断服务例程描述表**进行管理。**中断服务例程描述表**是一个名为 Irq_action 的指针数组, 定义如下:

```
Static struct irqaction * irq_action[NR_IRQS+1];
```

- **中断服务例程描述符:** Irqaction 结构体;
- **机器支持硬件中断源的数目:** NR_IRQS;
- **中断服务例程描述表数组的下标:** 与中断请求号 IRQ 对应, 使用 IRQ 作为索引就可以找到该设备的中断例程描述符。

- **Irqaction 结构体**

Irqaction 结构体定义在/include/linux/interrupt.h 中, 如下所示:

```
Struct irqaction
{
    Void(*handler)(int,void*,struct pt_regs*); /*指向中断服务例程*/
    Unsigned long flags; /*中断标志*/
    Unsigned long mask; /*中断掩码*/
    Const char *name; /*设备名*/
    Void *dev_id; /*设备号*/
    Struct irqaction *next; /*指向下一个描述符: 允许多个设备共享一个中断请求号 IRQ, 一个 IRQ 对应的多个中断例程描述符链接成一个单向链表*/
}
```

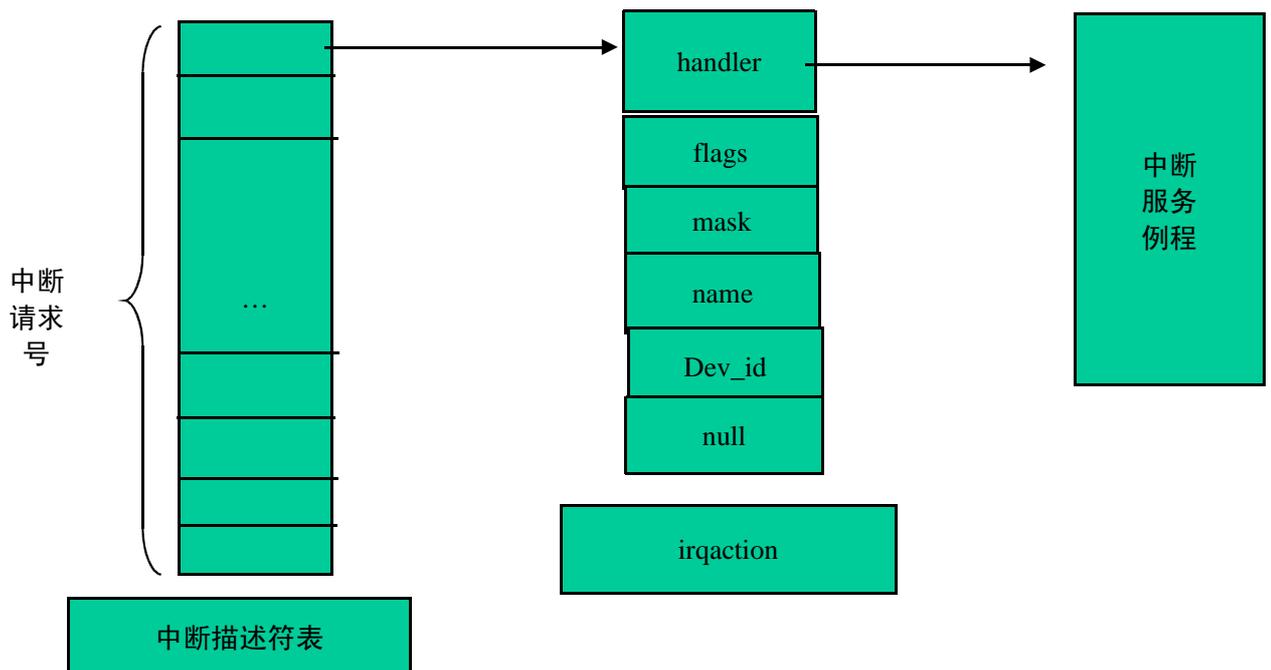


图 10.2 linux 中断例程描述表

- **Request_irq()**

在设备驱动程序加载时, 通过调用内核函数 Request_irq()建立驱动程序中断例程描述符 irqaction 结构体, 并把它登记到 Irq_action 数组中。Request_irq()与硬件相关。在 80x86 机器上 Request_irq()函数定义在 arch/i386/kernel/irq.c 中:

```
int request_irq(unsigned int irq, void (*handler)(int, void *, struct pt_regs *), unsigned long irqflags, const char * devname, void *dev_id)
```

- **free_irq ()**

撤消中断例程时, 可以使用函数:

Void free_irq ((unsigned int irq, void *dev_id)

把中断例程描述符表中下标为 irq, 设备号为 dev_id 占用的表项释放。

10.3.3DMA 方式

(略)

10. 4Linux 设备 I/O 操作

10.4.1 设备 I/O 操作

1. 设备文件的建立

调用: mknod (const char *pathname, mode_t mode, dev_t dev)

终端: mknod /dev/name type major minor

提供文件名、类型、主设备号、次设备号

2. 设备文件的打开、关闭

int open(const char *pathname,int flags) /*flag 给出的对文件的处理方式: 只读、只写、读写*/

int close(int fd) /*fd 设备的文件标识号*/

3. 的读写

int read(int fd,void *buf,int count)

int write(int fd,void *buf,int count)