操作系统实验指导

沈孝科，李中年编写

长江大学电信学院

目录

第一部分 基础知识部分

第一章.Linux 简介

1.1.Linux 操作系统的发展历史

Linux 操作系统核心最早是由 25 岁的芬兰大学生 Linus Torvalds 于 1991 年 8 月在芬兰赫尔辛基大学发布的，Linux 是 Linus 和 Minix 的混合称呼，意为 Linus 编写的类似 Minix 的系统。Linus 发布在 Intemet 上，得到了积极的回应，很快就有数百名程序员和爱好者通过 Intemet 加入 Linux 的行列，他们不断地对程序进行修改和完善，经过几年的努力，Linux 终于在全球普及，成为当今最为流行的操作系统之一。

Linux 最初是针对 Intel 架构的个人计算机开发的，但现在不仅个人桌面版的用户极多，在服务器领域也得到越来越多的应用，例如 Sun 公司的 Sparc 工作站和 DEC 公司的 Alpha 工作站等。此外，在嵌入式开发方面 Linux 更是具有其他操作系统无可比拟的优势。

Linux 的源代码是自由分发的，是完全公开的，也是完全免费的，你可以很方便地从网上下载。Linux 与 Intemet 同步发展壮大。Linux 的目标是 POSIX 兼容性。Linux 不仅涵盖了 UNIX 的所有特征，而且融合了许多其他操作系统的功能，这些特征包括：真正的多任务、虚拟存储、快速的 TCP/IP 实现、共享库和多用户。Linux 运行在保护模式并且完全支持 32 位和 64 位多任务。它能运行主要的 UNIX 工具软件、应用程序和网络协议。

Linux 还拥有一个完全免费的、遵从 X/Open 标准的 X-Window 的实现。Linux 内核的版权归 Linus Torvalds 所有。这个版权受 GNU(Gnu is Not UNIX)通用公共许可证(General Public License，GPL)的保护。你可以根据自己的需要对它进行必要的修改，无偿地使用，无约束地继续传播。

可以说 Linux 是一个高效和灵活的通用操作系统。采用 Linux 模块化的设计结构，既能充分发挥不断提高的硬件性能，又能跨不同平台使用，使得在 Linux 上开发的应用软件可以用很低的代价在不同的硬件平台上使用。Linux 操作系统也是一个多用户和多任务操作系统，它能保证 CPU 时刻处于使用状态，从而保持 CPU 的最大利用率。

现在 Linux 已经成为一个完整的类 UNIX 操作系统，它的核心版本在不断地更新，它有一个可爱的吉祥物——一只小企鹅(企鹅取自 Linus 家乡芬兰的吉祥物)，现在几乎每种版本的 Linux 都带有这个标志。

事实上 Linux 的确稳定并富有竞争力。许多大学与研究机构都使用 Linux 来完成他们的日常计算任务，很多中小型网站也在其服务器上运行 Linux，家庭的应用就更多了。Linux 主要用来浏览 Web，管理 Web 站点，撰写与发送 E-mail，以及玩游戏，它是一个具有专业水平的操作系统。

1.2.Linux 操作系统的特点

在使用方法上，Linux 与 UNIX 系统很相像，但 Linux 系统无论从结构上还是应用上都有其自身的特点。Linux 的内核特点是短小精悍，具有更高的灵活性和适应性。Linux 最大的特色在于源代码完全公开。所有的原始程序源码都可得到，包括整个核心及所有的驱动程序、发展工具及所有应用程序。在符合 GNU GPL(General Public License)的原则下，任何人皆可自由取得、散布，甚至修改源代码。除此之外，与其他操作系统相比，Linux

还具有下列特色。

1．Linux 是一个多用户、多任务的操作系统：

在 Linux 系统中，多个用户可同时在相同计算机上操作(通过终端或虚拟控制台)。Linux 在 386/486/Pentium/PentiumPro 上以保护模式运行，是真正的多任务操作系统，可同时执行多个进程，具有进程间内存地址保护，因此当某个进程出错时，不会波及整个系统。同时也提供了进程间的通信方式，使各进程能协同工作以满足用户的要求。

2．支持多种文件系统：

Linux 能支持多种文件系统，如 Ext2FS、ISOFS、Minix、Xenix、FATl6、FAT32、NTFS 等十多种文件系统。而且它自己还有一个先进的文件系统，提供最多达 4TB 的文件存储空间，文件名可以长达 255 个字符。

3．符合 POSIX　l003.1 标准

POSIX l003．1 标准定义了一个最小的 UNIX 操作系统接口，任何操作系统只有符合这一标准，才有可能运行 UNIX 程序。Linux 完全支持 POSIX 1003.1 标准，能运行 UNIX 上的丰富的应用程序。另外，为了使 UNIX SystemV 和 BSD 上的程序能直接在 Linux 上运行，Linux 还增加了部分 SystemV 和 BSD 的系统接口，使 Linux 成为一个完善的 UNIX 程序开发系统。

4．具有较好的可移植性

Linux 系统核心只有小于 10%的源代码采用汇编语言编写，其余均是采用 C 语言编写，因此具备高度的可移植性。

5．支持多平台和多处理器

Linux 虽然最初是在 Intel x86 系列 CPU 上开发的，由于它不断地发展，因而可在许多不同的 CPU 上执行。同时还支持多处理器的体系结构，如 SMP。

6．全面支持 TCP/IP 网络协议

Linux 具有较强的网络功能，包含 ftp、telnet、NFS 等。同时支持 Appletalk 服务器、Netware 客户机及服务器、Lan Manager(SMB)客户机及服务器。其他支持的网络协议有：TCP、IPv4、IPX、DDP 和 AX.25 等。

1.3.目前流行 Linux 版本

“内核”是 Linux 的关键，Linux 内核主要包括：进程管理、存储管理、设备管理、文件系统、网络通信，以及系统初始化(引导)等功能。内核拥有自己的版本号，以版本 2．4．2 为例，2 代表主版本号，4 代表次版本号，2 代表改动较小的末版本号。在版本号中，次版本号为偶数的版本表明这是一个稳定的版本，若为奇数一般是指加入了一些新的东西，该内核只是开发过程中的一个快照，相当短暂，是一个测试版本。本书的实验系统就是基于 Linux 的 2.4.2 版本的内核实现的。

由于 Linux 本身只提供了操作系统的核心，并没有提供给用户各种应用程序，如编译器、系统管理工具、网络工具、Office 套件、多媒体、绘图软件等，普通用户就无法在此平台上展开工作，因此以 Linux Kernel 为核心再集成搭配各式各样的应用程序或工具组成一套完整的操作系统，即称为 Linux 发行版。目前最流行的正式版本有下列几种。

1．RedHat

这是目前世界上最流行的 Linux 发行套件。RedHatLinux(红帽 Linux)是 RedHat 公司发行的，它安装简易、使用方便、功能强大，特别是图形用户界面特别适合于初学者。RedHat Linux7.2 基于 Linux2.4 内核，是本书实验系统选用的版本。目前最新的版本是 Red Hat Linux

9.0 版。

2．Caldera OpenLinux

Caldera OpenLinux 是最早关注简易安装方法的 Linux 正式版本之一，同时，它还在正式版本中集成了办公软件。现有最新版本是 Caldera Open Linux 2.2。

3．TurboLinux

TurboLinux 公司是以推出高性能服务器而著称的 Linux 厂商，在美国有很大的影响。它在亚洲是占市场最大的商业版本，在中国、日本和韩国都取得了巨大的成功。现在较流行的版本是 Turbo Linux 6.0，它是基于 Linux 2.2 内核。

4．红旗 Linux

国内最出名的 Linux 发行版本要数红旗了，红旗 Linux 采用图形用户界面，简洁实用的菜单结构，类似 Windows 的界面和操作方式。最新的红旗 Linux 桌面版 4.0 加快了系统开机和启动的速度，为用户集成了包括上网、图形图像处理、多媒体应用，以及娱乐游戏等进行操作(例如中文编辑和打印等)。

1.4.RedHat 9.0 的安装

Linux 可以直接在裸机上安装，也可以在硬盘上与其他操作系统，如 MS-DOS、Windows、0S/2 等共存。安装前首先将硬盘分区，然后将 Linux 和 Windows 等不同的操作系统分别装到各自的分区。

如果在裸机上安装，或者只安装 Linux 系统，大概需要 20 分钟-60 分钟，安装时间取决于具体计算机的运行速度、Linux 的版本等条件。如果要在计算机上同时保留两个或多个操作系统，则可能要花费更多的时间。一个硬盘上最多只能有 4 个主分区，所以最多只能安装 4 个不同的操作系统。

1.4.1.安装前的准备

在安装之前，要整理并记录待安装计算机的硬件情况。具体地说，需要了解如下硬件数据。

- CPU：对 Linux 来说，要求 CPU 至少为 386。
- 内存：对 RedHat 来说，内存是多多益善(至少要有 8MB)。
- 硬盘：硬盘的个数、每个硬盘的接口类型和大小，如果有多个硬盘，哪个是主盘等信息。
- 显示卡：显存为多少，厂商名称及型号。
- 显示器：厂商及型号，显示器所允许的水平和垂直扫描频率的范围等。
- 鼠标：类型是什么?如果使用的是串口鼠标，则它接在哪个 COM 端口?
- 网络：如果需要网络功能，则需要知道主机所用的 IP 地址、子网掩码、网关地址、域名服务器的 1P 地址、主机所处域的名称、主机所用的名称以及网络类型等参数。

那么如何收集硬件资源的信息呢?可以从如下几个方面着手：

- 搜集主板、显示卡、显示器、调制解调器等计算机各种硬件设备的手册。
- 如果使用 MS-DOS5.0 以上的版本，可运行诊断工具 MSD. EXE 来收集硬件数据。
- 如果是在 Windows98/2000 或 WindowsNT 中，可双击“控制面板”中的“系统”图标，从出现的对话框中收集硬件数据。

这些收集到的系统硬件信息可以为接下来的安装工作提供参考。如果不知道上述信息

也没关系，现在流行的 Linux 安装盘都可以进行自动监测，你只要按照提示往下一步一步地进行就可以了。

1.4.2.建立硬盘分区

Linux 需要自己的分区，因此在安装之前，需要为此建立相应的分区。对硬盘重新分区，意味着将要删除硬盘上原有的一切数据，所以在重新分区前，切记要备份系统。

在硬盘上建立的分区有三种类型：主分区、扩展分区和逻辑分区。一个硬盘上最多能建立四个主分区，扩展分区本身不存储数据，而是用来建立多个逻辑分区。由于在 Linux 中文件系统和交换空间(用做虚拟内存 RAM)都需要各自占据硬盘上的一个独立分区，所以一般都要为 Linux 提供不止一个分区。独立的操作系统必须安放在主分区，所以一个磁盘系统最多可以有 4 个不同的操作系统。

下面针对分区时通常会遇到的几种情况分别进行讨论。

1．硬盘上还有未分区的空间(包括没有进行分区的硬盘)

这种情况下只需要为 Linux 建立一个分区就可以了(如在自定义类型安装时，用 Linux 的 fdisk 命令来完成)。新建分区可以是主分区(primary partition)，也可以是扩展分区(extended partition)上的分区(类似 DOS 上的逻辑分区)。启动安装光盘后，按照其上的安装指示步骤即可完成安装。

2．硬盘上有一个未使用的分区

这种情况意味着我们要使用一个未使用的分区来安装 Linux。因此首先要删除现已不用的分区，然后再建一个 Linux 分区。这些可以在自定义类型安装时，用 Linux 的 fdisk 来完成。在目前推出的 Linux 安装光盘中有对这种情况处理的选择项，所以可直接按照其指示步骤完成安装。

3．所使用的分区上还有未使用的空间(重新分区)

这种情况是最普通的，但也是最为复杂的。对这种情况，主要有两种方法：破坏性重新分区和非破坏性重新分区。

1)破坏性重新分区

破坏性重新分区较为简单，主要是删除原来的大分区，再创建几个小分区以供不同的系统使用，如图 1-1 所示。

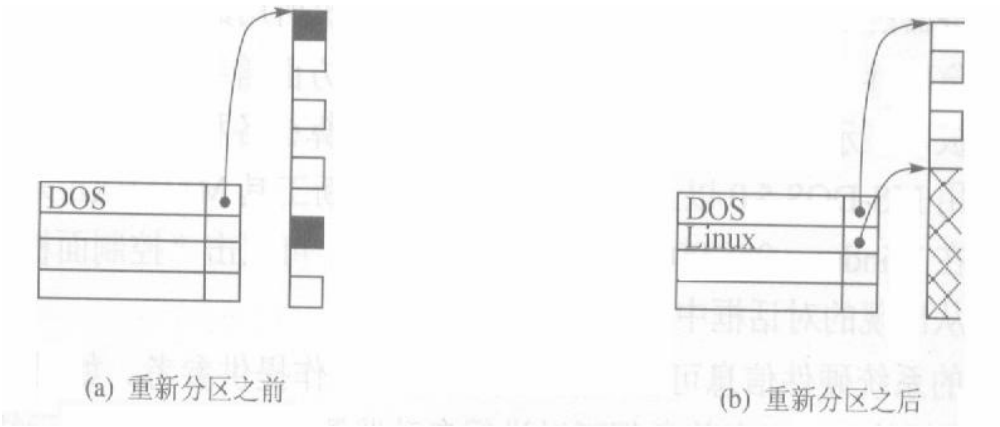


图 1-1 破坏性重新分区

具体步骤如下：

(1)备份原有分区上的数据(因为重新分区后，原来的数据将会丢失)。

(2)删除大分区。

(3)为原来所用的操作系统(如果还想用的话)和 Linux 创建不同的分区。

2)非破坏性重新分区

非破坏性重新分区较为复杂，但是保存了原来的数据且增加了新分区。这需要使用专门的分区工具来完成，如 PartitionMagic。这种方法一般包括如下三个步骤，图 1-2 所示为数据压缩前后状况。

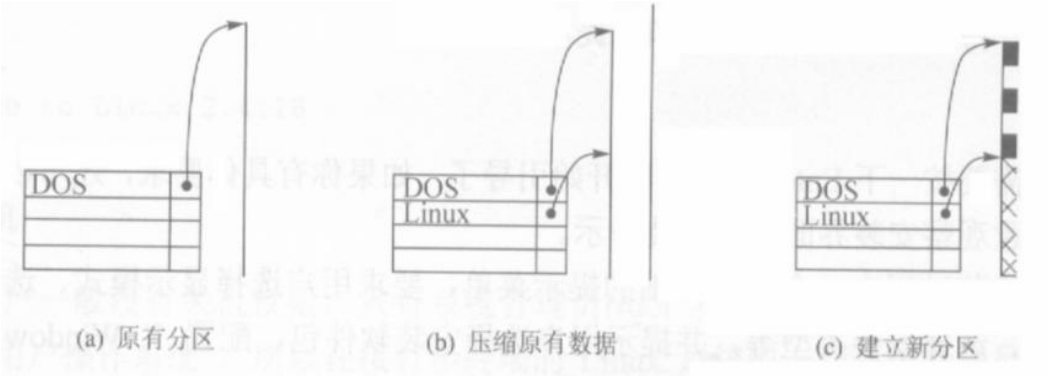


图 1-2 非破坏性重新分区

(1)压缩现有数据。这可以使自由空间尽可能地大。这一步很重要，如果做不好，则可能会限制重新分区的大小。

(2)改变分区大小。这会产生两个分区，一个分区为原来的含有数据的分区，另一个分区则是空白的。

(3)创建新分区。最简单的做法是删除新生成的分区，再创建 Linux 分区。上述这些步骤若利用分区工具 PartitionMagic，则可以很方便地完成。

1.4.3 安装类型

RedHat 提供了三种类型的安装：

- 客户机类型的安装。
- 服务器类型的安装。
- 自定义类型的安装。

1．客户机类型的安装

客户机类型的安装最为简单，只需要回答几个安装问题就可以完成安装。这对 Linux 新手尤其适合。该类型的安装首先删除硬盘上所有 Linux 分区，然后再创建 Linux 分区，并安装 Linux。如果硬盘上已经有了其他操作系统，那么本方法也可利用程序 LILO(the Linux Loader 或者 Grub 做成双启动。

2．服务器类型的安装

如果需要一个基于 Linux 的服务器，而又不愿意去做很多配置工作，那么这个方法是比较适合的。该类型的安装首先删除硬盘上所有的分区(不管它是不是 Linux 分区)，然后再创建几个 Linux 分区，并安装 Linux。该方法需要 1.6GB 左右的空间。

3．自定义类型的安装

自定义类型的安装最为灵活，可以自己决定如何分区，要安装哪些软件包，是否要用 LILO 或者 grub 来启动等。Red Hat Linux 6.0 之前的版本都是使用自定义类型的安装。

1.4.4 安装过程

在完成以上的准备工作之后，我们就可以进入安装 Linux 的具体工作了。RedHat 是一个比较成熟的 Linux 套件，提供了良好的安装界面，只要根据安装程序给出的提示，就能顺利地完成安装工作。目前，Linux 的发行方式都是以光盘的方式提供，所以在安装时一定要有光驱。

RedHat 是目前最流行的 Linux 操作系统，也是我们的实验系统的平台，因此将以 RedHatLinux 7.2 为例介绍其安装过程。Red Hat Linux 的安装方法很多，我们采用的是从 CD-ROM 安装。其安装光盘共有两张，第一张可直接从光盘启动，包含大部分的软件包和一些安装工具，第二张光盘则包含许多附加的软件包。下面是安装过程和注意事项：

(1)启动安装程序。用 Linux 的第一张光盘，从光驱引导启动系统。进入一个启动界面，显示“boot:”提示符，直接回车(按 Enter 键)，选择图形模式进行安装。

(2)选择安装界面使用的语言。

(3)选择默认的键盘设置。

(4)选择默认的鼠标设置。

(5)设置安装类型。Red HatLinux 提供了个人桌面、工作站、服务器和定制等多种安装类型，根据具体情况选择个人桌面或定制方式。

(6)进行硬盘分区。对硬盘进行分区是一件非常危险的工作，若没有必要或没有十分把握，最好先对重要的数据进行备份，以防不测。对于 IDE 硬盘，RedHatLinux 的驱动器标识符为“hdx-”，其中 hd 表明分区所在设备的类型，这里是指 IDE 硬盘。x 为盘号(a 为基本盘，b 为基本从属盘，c 为辅助主盘，d 为辅助从属盘)，• 代表分区，前 4 个分区用数字 1—4 表示，它们是主分区或扩展分区，从 5 开始就是逻辑分区。

对用户而言，无论有几个分区，分给哪个目录使用，它归根结底就只有一个根目录，一个独立且惟一的文件结构。RedHatLinux 采用了“装载”的处理方法，将一个分区和一个目录联系起来，因此每个分区都是用来组成整个文件系统的一部分。

Linux 最少需要两个分区，一个 Linux native(文件)分区，一个 Linux swap(交换)分区。其中 Linuxnative 分区是存放 Linux 系统文件的分区，它只能用 EXT2 的分区类型，在分区时应该将载入点设置为“/”目录。SWAP 分区则用做交换空间，它主要是把主内存上暂时不用的数据存起来，在需要的时候再调进内存内。一般建议分区方案如下：

● SWAP 分区　SWAP 分区至少要等于系统上实际内存的容量，一般来说它的大小是内存的两倍。

● /boot 分区　它包含了操作系统的内核和在启动系统过程中所要用到的文件，建这个分区是很有必要的，因为目前大多数的 PC 机要受到 BIOS 的限制，况且如果有了一个单独的/boot 启动分区，即使主要的根分区出现了问题，计算机依然能够启动。这个分区的大小约在 50MB~100MB 之间。

● /分区　这是根目录挂载的位置。系统运行所需要的其他文件都在该分区上，这个分区的大小约在 1. 7GB~5GB 之间假如是初次安装 Linux 系统，最好选择自动分区的方式。当然，如果安装者对 Linux 较为熟悉，也可以用系统配制的硬盘管理工具 Disk Druid 来定制所需要的分区，它不但可以根据用户的要求创建和删除硬盘分区，还可以为每个分区管理载入点。

(7)设置文件系统为 EXT2。

(8)配置引导装载程序。选择 LILO 作为引导安装程序。LILO 可以安装在第一硬盘的主引导区(MBR)或 Linux 分区的引导扇区。如果想使用 LILO 来做双启动，则需要选择前者：

如果是想用 Linux 启动软盘或其他系统引导器引导 Linux，请选择后者，即将 LILO 安装在 Linux 分区的引导扇区。

(9)网络配置。

(10)防火墙配置。

(11)选择系统默认的语言及其他支持语言。

(12)时区配置。

(13)设置 root 密码。

(14)选择软件包组。

(15)筹建引导盘。

(16)配置显卡。

(17)进行安装。

1.5.Linux 常用命令简介

1.5.1.使用常识

1．登录

Linux 是一个多任务、多用户的操作系统。当有多个用户同时使用一台机器时，Linux 可以分时地运行不同用户的应用程序。为了区分各个用户，每个用户都必须有自己独立的账号，系统要求每一名合法用户在使用 Linux 系统之前，首先必须按自己的身份登录。

在第一次登录时，可以用 root 用户及口令来登录。键入用户名 root，然后输入口令，这样就登录了。这时，就可以按 root 身份来使用 Linux。通常，用户是通过 shell 来使用操作系统的。shell 类似于 MS-DOS 下的 COMMAND.COM 命令解释器，是用户与操作系统核心之间的命令接口，负责接收用户输入的命令并将其翻译成机器能够理解的指令。按 root 方式登录时，命令窗口的提示符是“#”。root 是超级用户，对整个系统拥有一切权利。当用普通用户登录 时，则提示符是“#”。当然，提示符的形式也可以由用户重新设定。输入用户名之后，还应在提示输入口令时，将正确的口令键入，这样才能进入 Linux 系统中，向系统提交命令。

Linux 命令是区分大小写的，大多数命令都使用小写。

Linux 命令常带有各种选项。选项前一般用“—”加字符串表示，选项可以组合使用。例如：

```
$  ls  -la
```

上述命令及参数的含义是使用详细列表的形式，显示当前目录下所有的文件。

2．退出系统

在停止使用系统时，要退出系统。否则，其他用户就可能使用你的账号，做一些可能会令你后悔不及的事，比如将你的文件系统删除、修改注册口令等。

退出系统的方法有很多，可以使用 exit 或 logout 命令，或使用组合键 Ctrl+D。例如：

```
#  exit
logout
Welcome to Linux 2.4.18
login:
```

3．关机

普通用户一般没有关机权限，只有系统管理员(root 身份的特权用户)才能关闭系统。Linux 是多用户操作系统，所以在接有多终端的 Linux 系统中，除系统管理员本人之外，可能还有很多用户通过各种方式使用 Linux 主机。另外，在正常工作时，系统为提高访问和处理数据的速度，将很多进行中的工作驻留在内存中，如果突然关机，系统内核来不及将缓冲区的数据写到磁盘上，就会丢失数据甚至破坏文件系统。因此，系统管理员不能以直接关闭电源的方式来停止 Linux 系统的运行，而要按正常顺序关机。关机方法有两种：可以使用 halt 或 shutdown 命令，也可以同时键入 Ctrl+Alt+Del。

例如，如果使用 halt 命令，最后会显示已关机的信息：

The system is halted.

System halted.

这时才可以关闭电源。

1.5.2.文本编辑命令

Linux 文件可分为二进制文件和文本文件。二进制文件通常是由程序生成的，而文本文件既可以由程序生成也可以用编辑器来创建。Linux 下可运行许多种编辑器：有行编辑程序，如 ed 和 ex；也有全屏编辑程序，如 vi 和 emacs 等。

vi 是 UNIX 系统提供的标准的屏幕编辑程序，它虽然很小，但功能很强，是所有 UNIX 和 Linux 系统中最常用的文本编辑器。本节的讨论主要以 vi 为例。

利用 vi 进行编辑时，屏幕显示的内容是被编辑文件的一个窗口。在编辑过程中，vi 只是对文件的副本进行修改，而不直接改动源文件，因此用户可以随时放弃修改的结果，返回原始文件。只有当编辑工作告一段落，用户明确地发出保存修改结果的命令时，vi 才用修改后的文件取代原始文件。

1．vi 的两个模式

vi 编辑器有两种模式：命令模式和输入模式。在命令模式中，键入的是命令.这些命.令有移动光标、打开或保存文件，或者进入输入模式以及查找或替换等。在输入模式中，.键入的内容直接作为文本。只要按下 Esc 键，就可以进入命令模式。

2．vi 的使用举例

下面举例说明 vi 的使用。

(1)假如要创建或编辑文件 test，则只要输入如下命令即可：

$vi test

当 vi 激活后，首先让终端清屏，然后会显示如下状态：

"test"[New file]

这时 vi 处于命令模式。这里为节省空间，只显示了部分行，其中“—”表示空白缓冲区，而下面的“_”表示光标位置。

(2)键入 i 进入输入模式，并输入：

What is”LINUX”?In the narrow sense，——

(3)按下 Esc 键，就可进入命令模式。这时，可以通过方向键或 b 键或 f 键，将光标移动到如下所示位置：

What is”UNIX”?In the narrow_sense，

～

～

～

(4)键入 a 讲入输入模式，可在光标后输入字符，按下 Esc 键进入命令模式。

What is "LINUX"? In the narrowest_sense,

～

～

～

(5)键入进入输入模式，输入一行，键入 Esc 进入命令模式。将光标移到 y 字母下：

What iS ”LINUX"? In the narrowest sense，

it is a time-sharing operating system

～

～

(6)在命令模式下，键入 x 可删除一个字符。如果连续 5 次，那么会出现如下状态：

What is ”LINUX'’?In the narrowest sense，

it is a time-sharing operating s_

～

～

(7)在命令模式下，键入 ZZ 或“：wq”就可以保存文件并退出。

3．vi 的其他信息

以上只介绍了 vi 的几个基本操作。表 1-1—表 1-3 列出了 vi 的一些常用操作。

注意：<a>表示按键 a，而不是按三个键<、a 和>。

表 1-1 进入输入模式的方法

操作	作用
<a>	在光标后输入文本
<A>	在当前行末尾输入文本
<i>	在光标前输入文本
<I>	在当前行开始输入文本
<o>	在当前行后输入新一行
<O>	在当前行前输入新一行

注：①可以在以上命令之前加上数字表示重复次数。

②可以利用键盘上的方向键及<PageDown>等使光标定位于所需的位置。

表 1-2 删除操作

操作	作用
<x>	删除光标所在的字符
<dw>	删除光标所在的单词
<d$>	删除光标至行尾的所有字符
<D>	同<d$>
<dd>	删除当前行

注：可在删除命令前加上数字，如<5dd>表示删除 5 行。

表 1-3 改变与替换操作

操作	作用
<r>	替换光标所在的字符
<R>	替换字符序列
<cw>	替换一个单词
<ce>	同<cw>

1.5.3.文件操作命令

在 Linux 系统中，所有的数据信息都组织成文件的形式，然后保存在层次结构的树形目录中。用户的一切工作本质上就是对文件的操作。

1．目录与文件的基本操作

Linux 的文件系统结构是树状结构。执行 Linux 命令，总是在某一目录下进行的，该目录称为当前工作目录(current working directory)，通常简称为当前目录。当用户刚登录到系统中时，当前目录为该用户的主目录(home directory)。例如用户 wang 的主目录为/home/wang。用户主目录可以在/etc/passwd 文件中指定。

当引用另一个文件或目录时，可以从当前工作目录来相对定位(给出相对路径)，如 doc/f'fie.c；也可从根目录来绝对定位(给出绝对路径)，如/home/wang/doc/file.c。在 Linux 中，目录名之间用“/”分隔。在 Linux 文件系统中，根目录是用“/”表示的。另外“.”表示当前目录，而“..”表示当前目录的上一级目录。

1)常用的目录操作命令

● pwd

功能：打印当前工作目录。

例如：

$pwd

/home/wang

● cd

功能：改变当前目录。

例如，首先将当前目录改为上一级目录即/home，然后再将当前目录改为/usr/bin。这些操作结果可从动态改变的 shell 提示中看出来。

wlinux：$cd . .

wlinux：home$cd /usr/bin

wlinux：/usr/bin$

● mkdir

功能：创建目录。

例如下面创建了数个子目录：

$mkdir bin doc prog junkDir junkDir2

$ ls -CF

Bin/ doc/ junkDir/ junkDir2/ prog/

● rmdir

功能：删除目录。

例如下面删除了两个子目录 junkDir 和 junkDir2(欲删除子目录的内容应为空白)：

$rmdir junkDir junkDir2

$ ls -CF

bin/ doc/ prog/

2)常用的文件操作命令

有关文件操作命令与目录操作类似，现简述其中的三条命令。

● cat

功能：显示文件内容或合并多文件内容。不管文件长短，使用 cat 会一下子显示所有内容。

例如：

$cat junk

● cp

功能：复制文件。

例如把一文件 junk 复制到文件 junlc2 中：

$cp junk junk2

● rm

功能：删除文件。

例如删除文件 junk：

$ rm junk

2．文件权限

Linux 系统为了保护用户个人的文件不被其他用户读取、修改或执行，Linux 提供了文件权限机制。对每个文件(或目录)而言，有以下 4 种不同的用户：

- root：系统超级用户，能够以 root 账号登录。
- owner：实际拥有文件(或目录)的用户，即文件所有者。
- group：用户所在用户组的成员。
- other：以上三类之外的所有其他用户。

其中，root 用户自动拥有读、写和执行所有文件的操作权限，而其他三种用户的操作权限可以分别授予或撤销。因此，每个文件为后三种用户建立了一组 9 位的权限标志，分别赋予文件所有者、用户组和其他用户对该文件的读、写和执行权。

1)显示文件权限

可以用“ls -1”命令显示文件的权限，例如：

```
$ ls -l
drwxr-xr-x  2  wang  users  1024  Aug  18  02:49  bin
drwxr-xr-x  2  wang  users  1024  Aug  20  16:64  doc1
-rw-r--r--  1  wang  users  849   Aug  18  03:00  junk
-rw-r--r--  1  wang  users  580   Aug  18  02:56  me.txt
drwxr-xr-x  2  wang  users  1024  Aug  18  02:49  prog
total  5
```

在以上所列出的文件的长格式显示中，共有七列：

- 第 1 列表示文件权限，如 junk 的权限为-rw-r--r--。
- 第 2 列表示文件的链接数，如 junk 的链接数为 1。
- 第 3 列表示文件的所有者，如 junk 的所有者为 wang。
- 第 4 列表示文件所属的用户组，如，junk 的用户组为 users。
- 第 5 列表示文件的大小，如，iunk 的字符数为 849。
- 第 6 列表示文件的最后修改日期与时间，如 junk 的上一次修改时间为 Aug 18 03：00。
- 第 7 列表示文件本身的名称，如 junk。

文件访问权限由 10 个字符组成，如；

-rw-r--r--

第一个字符表示文件类型：-为普通文件，d 为目录，b 为块设备文件，c 为字符设备文件，1 为符号链接。后面 9 个字符每三个字符为一组，依次代表文件的所有者、文件所有者；所属的用户组以及其他用户的访问权限。每组的三个字符依次代表读、写和执行权限。系统用 r 代表读权限，w 代表写权限，x 代表可执行权限(对目录而言，可执行表示可

以进入浏览；如果没有相应权限，则用-表示。

2)设置或改变文件权限

文件所有者和超级用户可以用命令 chmod 来设置或改变文件的权限。命令 chmod 的用法有两种，其中一种如下：

chmod {a， u， g， o} [+，-，=] {r，w，x) 文件名

这里，可以用 a(d1，所有用户)、u(user，所有者)、g(group，所属用户组)、o(other，其他用户)中由一个或多个表示访问权限的赋予对象；用+、-、：分别表示增加、删除、赋予权限；用 r、w、x 组合表示读、写、执行权限。

另一种用法是用八进制数来设置权限：

chmod nnn 文件名

其中，nnn 为三个八进制数，每个八进制数分别表示所有者、同组用户与其他用户的权限，这些八进制数所对应的三位二进制数分别对应于读、写和执行权限，1 表示有相应的权限，而 0 表示没有相应的权限。例如：

chmod 755 文件名

755 代表-rwxr-xr-x，表示文件所有者具有读、写和执行权限，同组用户具有读和执行权限，其他用户具有读和执行权限。

3．文件链接

在 Linux 文件系统中，每一个文件只有一个惟一的索引节点号(inode number)即文件的内部标识符，可以有多个外部名称(用户指定的)。一个目录实际上是文件的索引节点号与其相对应的文件名的一个列表，目录中的每个文件名都有一个索引节点号与之对应。

1)查看索引节点号

1s -i 可用来查看索引节点号，例如：

```
$  ls -i
45615 f
$
```

1n 可用来为一个文件再增加一个名称，在系统内部则为文件增加一个链接，该文件名与原文件名指向同一个文件。例如：

```
$  ln  f  g
$  ln  g  h
$  ls  -il
total  54
45615  -rw-r--r--  3  wang  users  17127  Aug  20  22:09  f
45615  -rw-r--r--  3  wang  users  17127  Aug  20  22:09  g
45615  -rw-r--r--  3  wang  users  17127  Aug  20  22:09  h
```

从显示中可以看出文件 f 有三个链接。

3)删除一个文件

当用命令 rm 删除一个文件时，实际上删除的是文件的一个链接(或一个名称)。例如，以下操作使文件 f 的链接数减 1，从显示中可以看出最后显示少了一行：

```
$  rm h
$  ls  -il
total  36
45615  -rw-r--r--  2  wang  users  17127 Aug 20 22:09 f
45615  -rw-r--r--  2  wang  users  17127 Aug 20 22:09 g
```

当文件链接数为 0 时，则相应的文件索引节点才被删除，即实际删除了文件，例如：

$ rm f g

$ ls -il

total 0

4．查询文件

在通常的 UNIX 操作系统中，有三个命令可以用于从文件中查找给定的字符串，并显示相应的行。

- grep：最为常用，可用固定字符串来查询，也可用正则表达式来查询。
- egrep(extendedgrep)：扩展的 grep，可用正则表达式查询。
- fgrep(fastgrep)：快速 grep，但只能查询固定字符串。

在 Linux 操作系统中，这三个命令都已合并了，用户只要使用 grep 就可以了(当然也可以使用其他两个，事实上这两个是 grep 的链接)。

grep 命令格式：

grep[选项]字符串或正则表达式[文件列表]

以下是 grep 命令的应用举例。

显示信箱中 Emml 发送者：

$ grep 'From'

$MAIL

显示含有 fork 的 C 语言文件名：

$grep -l 'fork' *.c

除了 grep 外，Linux 还提供了其他文件查询工具，如：

nngrep 可以查询新闻组。

zgrep 可以查询压缩过的文件。

zipgrep 同 zgrep。

利用联机帮助命令 man 可以了解上述命令的具体用法。这是 Linux 下最重要的命令之一。属必须掌握的命令。其用法为：

man 命令

其用法相当简单，只需在 man 后加上要查询的命令名，系统就会给出该命令的所有使用方法和说明。这对于学习 Linux 命令的益处很大。

5．文件排序

对文本文件，可以用命令 sort 进行排序。命令 sort 可以带有各种不同的选项，从而采用不同的排序方法。其格式为：

sort [option] [FILE]

用选项“-f”，可以不区分大小写，例如：

$ls | sort -f

其中：“|”是管道命令，它将命令 1s 的输出作为命令 sort 的输入，所以 ls | sort -f 的含义，是将目录列表按字母排序显示，不区分字母的大小写。

用选项“-n”，可以按数值大小进行排序，而不是按字母顺序进行排序。例如：

$is -s | sort -n

用选项“-r”，可以用逆序排序，例如：

$ls –s | sort -nr(从这里开始)

用选项“+数值”，可以按跳过所定数值的域后的那个字段进行排序，例如：

$ls -1 | sort +4nr

6．对文件的列或域的操作

在 Linux 系统中，可以对文件中的列或域进行各种剪切和合并，常用命令有三个。

cut：从文件中选择列或域。

paste：对文件中的列或域进行合并。

join：可根据关键域对文件进行合并。

1)cut

用 cut 可以仅显示文件的字节数和文件名：

$ls -1 | cut -c29-4l, 55-

上述命令中的“-c”表示按字符(character)选取，“29-41”表示字符区间，“55-’’表示从第 55 个字符开始到结尾。而“-5’’则表示从开头到第 5 个字符。

用 cut 命令可以显示用户名、用户全名和用户的主目录名：

$ls -1 | cut c29-4l， 55-

上述命令中的“-c”表示按字符(character)选取，“29-41”表示字符区间，“55-”表示从第 55 个字符开始到结尾。而“-5”则表示从开头到第 5 个字符。

用 cut 命令可以显示用户名、用户全名和用户的主目录名：

$cut -d：-f1,5-6 /etc/passwd

上述命令中“-d：”表示域的分隔符(delimiter)是“：”，-f 之后的数字表示第几个域，1 表示第 1 个域，5-6 表示第 5 和第 6 个域。

2)paste

cut 和 paste 命令组合使用可以显示用户全名、用户名、主目录名和注册的 shell。

cut -d：-f5 /etc/passwd > /tmp/tl

cut -d：-fl /etc/passwd > /tmp/t2

cut -d：-f6 /etc/passwd | paste /tmp/tl /tmp/t2

paste 命令行中的“-”表示第三个输入文件来自标准输入。

3)join

用 join 命令可以根据共同的关键域(GID)而将/etc/passwd 和/etc/group 进行合并。

$join -t：-jl 4 -j2 3 /etc/passwd /etc/group

以上“-t”表示域的分隔符是“：”，而不是默认的制表符 Tab 或“\t”，“-j14”表示第 1 个文件的第 4 个域为共同域，“-j23”表示第 2 个文件的第 3 个域为共同域。

7．文件的压缩和解压缩操作

LINUX 中使用 gzip 和 compress 命令可以完成对文件的压缩，用 compress 命令压缩的文件后缀名为“**.Z**”。

1)compress

格式

compress 文件名

也可以使用通配符同时压缩几个文件。默认情况下，一个文件压缩之后，源文件将会被删除。如果要解压缩文件，可以使用如下命令：

uncompress 文件名

同样可以使用通配符*．Z 解压缩所有的压缩文件，在指定文件名时要包括.z 后缀。

2)gzip

gzip 命令是一种新的压缩工具，它的压缩算法不同于 compress 命令。使用 gzip 命令时，应指定压缩类型和文件名。

格式

gzip -9 文件名

上述命令中-9 选项用来告诉 gzip 命令使用最高压缩因素，是最常用的选项。gzip 命令压缩文件的扩展名是．gz，并且压缩完毕后，源文件将被删除。解压缩文件时，可以使用

gunzip 或者 gzip-d 文件名。

3)tar

tar 命令(tapearchive)在很多年以前就集成在 UNIX 系统中了。但 tar 命令的用户界面不友好，尤其是在不熟悉 tar 命令的用法时。

tar 命令的目的是建立一个单一的文档文件，就像 DOS 环境下的 ZIP 命令一样。使用 tar 命令可以将多个文件组合成为一个单一的大文件，这样就更加易于管理和备份。

格式

tar [选项] [文件名]

选项中包含很多选项，具体内容可以从 mantar 中获得。文件名中可以使用通配符。下面是两个使用 tar 的例子：

tar cvf archivel.tar /usr/src/linux

此命令将/usr/src/linux 目录下的所有文件组合成一个 archivel.tar 文件。使用 c 选项表示 tar 命令创建一个新文档，v 选项表示 tar 命令在执行时显示提示信息，f 选项表示 tar 命令使用文件名 archivel.tar 作为输出文件。

tar 命令不会自动在文件名后面加上扩展名.tar，所以使用者必须自己加上．tar 以便识别文档文件。

tar xvf archivel.tar

此命令将释放 archivel.tar 中的文件。使用 f 选项指明要释放的 tar 文件。 v 选项指示 tar 在执行时显示提示信息。- x 选项用来从文档中释放文件。

1.5.4.进程控制命令

Linux 是一个多用户、多任务操作系统。多任务是指可以同时执行多个任务。但是一般计算机只有一个 CPU，所以严格地说并不能同时执行多个任务。不过，由于 Linux 操作系统只分配给每个任务很短的运行时间片，如 20 毫秒，而且可以快速地在多个任务之间进行切换，因而看起来好像是在同时执行多个任务。

在 Linux 系统中，任务就是进程，它是正在执行的程序。进程在运行过程中要使用 CPU、内存、文件等计算机资源。由于 Linux 是多任务操作系统，可能会有多个进程同时使用同一个资源，因此操作系统要跟踪所有的进程及其使用的系统资源，以便进行进程和资源的管理。

1．Linux 的前台与后台进程

在 Linux 中，进程可以分为前台进程和后台进程。前台进程可以交互操作，即可以从键盘接收输入且可以将输出送到屏幕；而后台进程是不可以交互操作的。前台进程一个接一个地执行，而后台进程可以与其他进程同时执行。

2．进程控制

前面的所有例子中，在 shell 提示符下输入的命令都是按前台方式执行的。如果要按后台方式执行，只要在命令行末尾加上“&”即可。

1)利用两次命令指定两个后台进程

$yes > /tmp/null &

[1] 163

$yes>/tmp/trash&

[2] 1642)用 shell 内部命令 jobs 来显示当前终端下的所有进程

$jobs

[1]-Running yes>tmp/null &

[2]+Running　yes>tmp/trash&

3)用 fg 将一个后台进程转换为前台进程

$　fg　％1　　。

yes>/tmp/null

4)用组合 Ctrl+Z 来暂停执行一个进程，并转换为后台进程

$　fg　％l

yes>/tmp/null

[1]　-　　Stopped　　yes>/tmp/null

5)用 bg(重新)运行一个后台进程

$　bg　％1

[1]yes>/tmp/null

6)用 kill 命令撤销一个进程

$　kill　％1

如果想中止某个特定的进程，应该首先使用 ps 命令列出当前正在执行中的进程的清单，然后再使用 kill 命令中止其中的某一个或者全部进程。在默认情况下，ps 命令将列出当前系统中的进程，如下所示：

$　ps

PID	TTY	STAT	TIME	COMMAND
367	p0	S	0：00	bash
581	p0	S	0：01	rxvt
747	p0	S	0：00	(applix)
809	p0	S	0：18	netscape index.html
945	p0	R	0：00	ps

ps 命令会列出当前正在运行的程序以及这些程序的进程号，也就是它们的 PID。

可以使用这些信息通过向 kill 命令发出一个“-9”，也就是 SIGKILL 信号来终止某个进程：

$　kill　-9　809

在 Linux 系统中，可以定时执行一个程序。只需用 at(或 batch)、atq、atrm 来分别安排、查询、删除定时作业任务。例如，下面列出一个脚本程序，并安排其在 2：30am 执行：

$　more WholsWorking

date>home/wang/bin/out

who>>/home/wang/bin/out

$at-f/home/wang/bin/Who 土 sWorking 02：30

Job　4　will　be　executed　using　/bin/sh

进程之间可能具有父子关系，如在 shell 提示下执行的进程都是当前 shell 的子进程。shell 也是一个进程，它不断地执行用户的输入命令。一般而言，当父进程结束时，其子进程也已结束。与文件系统相似，进程之间的关系也是树状的，但是其变化较快。所有进程都是由 1 号进程 init 进程派生出来的。

这就是说，当用户退出系统后，该用户的所有进程，不管是前台的还是后台的，都将结束。如果要让一进程在用户退出系统后继续执行，则可使用 nohup 命令。例如：

$ nohup find/-name，'*game*'，-print&

[1] 320

nohup：appending output tO nohup．out'

$

3．进程的优先级

进程是有优先级的。超级用户的优先级比普通用户的要高，用户前台进程的优先级比后台进程的要高。在 Linux 中，优先数为-20(最高优先级)-19(最低优先级)。默认优先数为 0。所有用户都可以降低自己的优先级，而只有超级用户可以增加优先级。可以用 nice 命令或者系统调用来改变进程的优先级，具体使用方法可用命令“man　nice”查询。

1.5.5.系统管理

Linux 是一个功能强大而复杂的操作系统。为了能更好地发挥系统性能，需要一些系统　管理方面的知识。本节介绍超级用户、账号管理、文件系统管理等内容。

1．超级用户

UID(UserID，用户Ⅲ)称做用户标识符，它是系统分配给每个用户的用户识别号。系统通常通过 UID 而不是用户名来操作和保存用户信息。每一个 Linux 系统上都有一个 UID 为 0 的特殊用户，它通常称做超级用户或 root 用户(因为它的用户名通常为 root)。当以 root 用户身份登录时，对整个系统具有完全的访问权限。也就是说，对于 root 用户，系统将不进行任何权限检查，并且系统把所有文件和设备的读、写和执行权限都提供给了 root 用户，这使得 root 用户无所不能。

正因为如此，应当合理使用 root 账号。如果在 root 账号下使用命令不当，后果不堪设想。例如以 root 身份运行命令“/bin/rm　-rf/”则将删除整个系统。因此，一般应以普通用户身份使用系统。

1)使用 su 命令进入 root 用户身份

当需要以 root 用户身份使用时，可以用 su 命令切换成 root 用户。在执行完系统管理后，应马上用 exit 命令切换到原来状态。

命令 su 可以用来改变用户身份，如果需要切换成 root 用户，只要键入 su 并输入 root。口令就可以了。例如：

```
$　su
password：
#
```

2)使用飞选项

如果仅以 root 用户身份运行一个命令，可以使用选项-c，例如：

```
#su -c”vi/etc/passwd”
Password：
```

3)改变 shell 为 csh

如果还需要改变所使用的 shell，可以加上选项—s，例如：

```
$　su　-s　/bin/csh
Password：
#echo　$SHELL
/bin/csh
#
```

2．用户和用户组管理

用户管理是系统管理的一个重要部分。对 Linux 而言，每个用户都有一个惟一的用户名或登录名(login name)。用户名用来标识每个用户，并避免一个用户删除另一个用户的文件这类事故的发生。每个用户还必须有一个口令。

除了用户登录名和口令外，每个用户还有一些其他属性，如用户 ID、用户组 ID(groupID，GUID)、主目录(homedirectory)、登录 shell(10ginshell)等。系统上所有用户信息都存在系统文件/etc/passwd 和/etc/group 中。

1)用户管理

用户管理包括增加、修改和删除用户账号。这些工作可以通过手工编辑有关文件完成，但是最好使用用户管理工具。这里介绍一组简单的基于命令的用户管理工具：useradd，usermod 和 serdel。基于图形的用户管理工具有 control-panel 等。

(1)增加用户名

如果需要增加一个用户，可以使用 useradd 或 adduser 命令，例如，以下命令增加一个名为 john 的用户。除了用户名外，其他参数均为默认。

#adduser john

通过此操作，得到的效果如下：

● 在口令文件/etc/passwd 中，增加了一个用户 john 的条目：

john:x:506:506::/home/iohn:/bin/bash

● 如果使用了影子口令，还会在影子口令文件~tc/shadow 中，增加一个用户 john 的条目：

John:!!:10772:0:99999:7:::

● 在用户组文件/etc/group 中，增加了一个用户组 john 的条目：

john::506:

● 为用户 john 创建了主目录，并将/etc/skel 下的模板文件复制到/home/john 下。

如果需要修改 useradd 命令的默认配置，可以通过修改目录/etc/default/和/etc/skel 中的文件来完成。

(2)设置口令

为了让新增用户可以登录，还需要为他设置一个口令。这可以用 passwd 命令来完成。

例如：

#passwd john

Changing password for user john

New password：

Retype new password：

passwd：all authentication tokens updated successfully

#

(3)修改与账号有关的信息

如果需要修改与用户账号有关的信息，可以使用 usermod 命令。例如，以下命令使用户 john 的账号在 2003 年 6 月 18 日之后为无效：

#usermod -e 6/1803 iohn

命令 chfn 也可以用来修改用户的一些个人信息，例如：

$chfn

Changing finger information for john.

Password： Name ()：John Zheng

Office []： 95 Product Development Center

Office Phone []： 823162 日 8

Home Phone []： 63821235

Finger information changed.

$

命令 chsh 还可以用来改变登录 shell，例如：

$chsh—s/bin/csh john

Changing shell for john.

Password：

Shell changed.

$

(4)删除一个用户

当要删除一个用户时，可以使用 userdel 命令。例如：

#userdel john

以上 userdel 的用法只删除用户账号，并不删除主目录。如果在删除用户时还要删除其主目录，则可以加上选项“--T'。

2)用户组管理

对 Linux，每个用户都属于一个或多个用户组。如果用户属于一个用户组，则享受该用户组的权限。这样，只需配置用户组的权限，就能配置各个用户的权限了。

(1)增加一个用户组

当要增加一个用户组时，可以使用 groupadd 命令。例如，下面增加了一个名为 teachers 的用户组：

#groupadd　teachers

结果是在用户组文件/etc/group 中增加了如下一行：

teachers：X：507：

(2)修改一个用户组

当要修改一个用户组时，可以使用 groupmod 命令。用法如下：

groupmod[-g dig[O]]　[—n name]group

例如，下面将用户组 teachers 改为名为 staff 的用户组：

#groupmod-n staff teachers

结果是用户组文件/etc/group 中 teachers 行改成了如下内容：

staff：x：507：

(3)删除一个用户组

当要删除一个用户组时，可以使用 groupdel 命令。命令 groupdel 很简单，只要加上用户组名就可以了。例如，如下命令删除名为 staff 的用户组：

#groupdel staff

3．文件系统管理

数据和程序文件都存储在块设备上，例如硬盘、光盘、软盘等。设备上的文件并不是无序的，而是按一定方法组织起来的。不同组织方法也就形成了不同的文件系统，例如 ext2、ext3、FAT32、FATl6 等。

Linux 操作系统的一个重要特点是它通过 VFS(VirtualFileSystem，虚拟文件系统)支持多种不同的文件系统。Linux 使用最多的文件系统是 ext2，这是专门为 Linux 而设计的文件系统，效率高。ext3 支是在 ext2 的基础上增加了日志管理。Linux 也支持许多其他文件系统如 Minix、FAT32、FATl6 等。另外，Linux 还支持 NFS(NetworkFileSystem，网络文件系统)。

若想了解 Linux 所支持的文件系统，可以显示文件/proc/filesystems。如果需要增加或删除对某个文件系统的支持，可以重新编译内核。

对 Linux 而言，所有设备(如硬盘、光盘、软盘等)的文件系统都有机无缝地组成了一个树状文件系统。这与 MS-DOS/Windows 9x/NT/2000/XP 等不一样，不是每个分区都有独

立的驱动盘符。

由于数据和文件都位于文件系统上，如果文件系统出了问题，则后果不堪设想。因此文件系统的管理尤为重要。本节主要介绍如下几个有关文件系统方面的知识：

- 如何安装和卸载文件系统。
- 如何监视文件系统。
- 如何创建文件系统。
- 如何维护文件系统。

1)手工安装和卸载文件系统

在访问一个文件系统之前，必须首先将文件系统安装到一个目录上(除了根文件系统之外，根文件系统在启动时自动安装到根目录/上)。安装方法有两种：一种是启动时系统自动根据文件/etc/fstab 来安装：另一种是用 mount 命令或相关工具宋手工安装。这里简单介绍一下 mount 命令。

(1)用 mount 命令安装

命令 mount 的基本用途是将一个设备上的文件系统安装到某个目录上。

格式

mount -t type device dir

参数说明

device：待安装文件系统的块设备名：

type：文件系统类型(关于系统所支持的文件系统类型的信息，可参阅文件/proc/file.systems)；

dir：安装点。

例如，下面将第一硬盘第一分区的 FAT32 文件系统安装到/dosc 上，这样就可以从/dosc 处访问该文件系统了：

#mount -t vfat /dev/hdal /dosc 命令 mount 还可以用来列出所有安装的文件系统：

```
#mount
/dev/hda3 on/type ext2(rw)
none on/proc type proc(rw)
/dev/hda2 on/dosd type vfat(rw)
none on /dev/pts type devpts (rw， mode=0622)
hawk：(pid470)on/net type nfs(intr，rw，port=1023，timeo=8，retrans=110，indirect，)
/dev/hdal on/dosc type vfat (rw)
#
```

(2)用 umount 命令卸载

文件系统的卸载很容易，只要使用命令 umount 即可。

格式

umount [-nrv] [device][dir[. . .]]

例如，如果要卸载以上刚刚安装的文件系统，可以如下：

#umount /dosc

也可以如下：

#umount /dev/hdal ，

2)自动安装和卸载文件系统

除了用手工方式安装文件系统外，系统还可以自动安装和卸载文件系统，只要在文件/etc/fstab 中列出了要安装的文件系统。除了注释行外，每行描述一个文件系统。每行包括

如下一些由空格或制表符分隔的字段。

- 设备点：指定要安装的块设备或远程文件系统。
- 安装点：指定文件系统的安装点。
- 文件系统类型：Linux 支持许多文件系统，如 ext2、ext3、ext、minix、sysv、swap、xiafs、msdos、vfat、hpfs、NFS 等。
- 安装选项：这是一组以逗号隔开的安装选项。关于本地文件系统的安装选项，请参阅 mount(8)：而关于远程文件系统的安装选项，请参阅 nfs(5)。
- 备份选项：指定是否使用 dump 命令备份文件系统。如果数值为 0，表示不备份。
- 检查选项：指定在系统引导时 fsck 命令按什么顺序检查文件系统。根文件系统的值为 1，即最先检查。所有其他需要检查的文件系统的值为 2。如果没有指定数值或数值为 0 时，表示引导时不做一致性检查。

下面是一个/etc/fstab 的示例：

```
/dev/hda3       /       ext2       defaults       1 1
/dev/hdal       /dosc       vfat       defaults       0 0
/dev/hda2       /dosc       vfat       defaults       0 0
/dev/hda4       swap       swap       defaults       0 0
/dev/fd0       /mnt/floppy       ext2       noauto，user       0 0
/dev/cdrom       none       /proc       proc       defaults       0 0
none       /dev/pts       devpts       mode=0622       0 0
```

在大多数情况下，Linux 系统所使用的文件系统不经常发生变化。因此，如果将这些经常使用的文件系统存放在文件/etc/fatab 之中，则系统启动时会自动安装这些文件系统，而在系统关机时能自动卸载它们。

3)监视文件系统状态

(1)用命令 df 显示文件系统的使用情况

当要显示文件系统的使用情况时，可以使用命令 df，例如：

```
#df
Filesystem       lk-blocks       Used       Available    Use％    Mounted on
/dev/hda3       2563244       1344202       1086506       55％       /
/dev/hdal       1614272       928       1613344       0％       /dosc
/dev/hda2       2004192       1509268       494924       75％       /dosd
```

(2)用命令 du 显示目录所占空间

当要显示某一个目录及其所有子目录所占空间时，可以使用命令 du，例如：

```
#du -s    /home
310984/home
#
```

4)维护文件系统

对文件系统要定期检查。如果出现损坏或破坏的文件，则需要修补。

最常用的方法是在文件/etc/fstab 中将检查选项数值(pass number)设置为大于 0 的正整数，如 1 或 2，这样系统在启动时会自动检查文件系统的完整性。

另一种方法是直接使用 fsck 命令来检查文件系统，如果需要，还可强制该命令修改错误。这是一个前端命令，根据不同的文件系统类型，fsck 将调用不同的检查程序如 fsckext2 等。

命令 fsck 的格式

```
fsck    [-AVRTNP]    [-—s i    [-t 文件系统类型]    [-ar]文件系统[. . .  ]
```

参数说明

其中命令行选项和参数的用法如下：

-A　对/etc/fstab 中的文件系统逐个检查。通常在系统启动时使用。

-V　详细模式。列出有关 fsck 检查时的附加信息。

-R　当和-A 一起使用时，不检查根文件系统。

-T　开始时不显示标题。

-N　不执行，只显示要做什么。

-P　当和-A 一起使用时，• 并行处理所有文件系统。

-s　串行处理文件系统。

-t　文件系统类型　指定要检查文件系统的类型。

-a　不询问而自动地修复所发现的问题。

-r　在修复之前，请求确认。

文件系统指定要检查的文件系统。可以是块设备名/dev/hda2，也可以是安装点，如/usr。在 fsck 检查一文件系统时，最好先卸下这个文件系统。这可以保证在检查该文件系统时，没有其他程序同时对它进行操作。

5)建立文件系统

当增加一个新硬盘或需要改变硬盘上原来分区时，在 Linux 能使用之前，需要对磁盘进　行分区和创建文件系统。创建磁盘分区可以用 fdisk 命令，而利用 mkfs 命令可以建立或初始化文件系统。实际上，每个文件系统类型都对应有自己单独的初始化命令，mkfs 只是最常用的一个前台的程序，它根据要建立的文件系统类型调用相应的命令。

mkfs 的格式：

mkfs [-V]　[-t fstype]　[fs-option]filesys[blocks]

常用的 mkfs 命令参数如下：

-t　指定系统的类型。

-c　检查坏块并建立相应的坏块清单。

-l　文件名　从指定的文件名中读取初始坏块。

4．Linux 源代码文件安放结构

Linux 系统模块的源程序文件主要由如下几部分构成：

arch　针对不同的硬件体系结构设置的模块

fs　文件系统

init　初始化模块

ipc　进程间通信

kernel　内核

include　．h 头文件

lib　库函数

mm　存储管理

net　网络管理

drivers　驱动程序

scripts　脚本文件

documentation　系统文档

1.5.6. rpm 命令介绍

rpm(redhat package manager)是由 RedHat 公司开发的软件包安装和管理程序，同 Windows 平台上的 Uninstaller 比较相似。使用 RPM，用户可以自行安装和管理 Linux 上的应用程序和系统工具。

rpm 可以让用户直接以二进制方式安装软件包，并且可替用户查询是否已经安装了有关的库文件；在用删除程序时，它又会询问用户是否要删除有关的程序。如果使用 rpm 来升级软件，rpm 会保留原先的配置文件，这样用户就不用重新配置新的软件了。 rpm 保留一个数据库，这个数据库中包含了所有的软件包的资料，通过这个数据库，用户可以进行软件包的查询。rpm 虽然是为 Linux 设计的，但是它已经移植到 SunOS、Solaris、AIX、Irix 等其他 UNIX 系统上了。rpm 遵循 GPL 版权协议，用户可以在符合 GPL 协议的条件下自由使用及传播 rpm。

1)安装 rpm 包

rpm 软件包通常具有类似 foo-1.0-386-rpm 的文件名。其中包括软件包的名称(foo)、版本号(1．0)、发行号(1)和硬件平台(i386)。安装一个软件包只需简单地键入以下命令：

$rpm -ivh foo -1.0-i386.rpm

foo ##

rpm 安装完毕后会打印出软件包的名字(并不一定与文件名相同)， 而后打印一连串的#号以表示安装进度。

2)卸载 rpm 包

卸载软件包就像安装软件包时一样简单，键入命令如下：

$rpm -e foo

注意：这里使用软件包的名字 foo，而不是软件包文件的名字 foo-1.0-1-i386.rpm。

3)升级 rpm 包

$rpm -ivh foo-2．0-1.i386.rpm

foo ##

rpm 将自动卸载已安装的老版本的 foo 软件包，用户不会看到有关信息。因为 rpm 执行智能化的软件包升级，会自动处理配置文件，可能会显示如下信息：

Saving /etc/foo.conf as /etc/foo.conf.rpmsave

4)查询已安装的软件包

使用命令 rpm-q 来查询已安装软件包的数据库。简单地使用命令 rpm-qfoo 会打印出.foo 软件包的包名、版本号和发行号：

$rpm -q foo

foo-2.0-1

除了指定软件包名以外，还可以使用另外的选项来指明要查询哪些软件包的信息。这些选项被称之为“软件包指定选项”，具体选项内容可以用 man 命令获得帮助，即使用命令:

man rpm.

5)验证软件包

验证软件包是通过比较已安装的文件和软件包中的原始文件信息来进行的。验证主要是比较文件的尺寸、MD5 校验码、文件权限、类型、属主和用户组等。

rpm -V 命令用来验证一个软件包。可以使用任何软件包作为选项来查询你要验证的软件包。如，命令 rpm -V foo 将用来验证 foo 软件包。 ，

又如，验证包含特定文件的软件包：

rpm -vf/bin/vi

验证所有已安装的软件包：

rpm -Va

根据一个 rpm 包来验证：

rpm -Vp foo-1.0-1.i386.rpm

1.6.Linux 目录结构

Linux 文件系统是树状的结构，系统中每个分区都是一个文件系统，都有自己的目录层次。Linux 会将这些分属不同分区的、单独的文件系统按树状的方式形成一个系统的总目录层次结构。目录提供了一个管理文件方便而有效的途径，最上层是根目录，其他的所有目录都是从根目录出发而生成的，如图 1 -3 所示。微软的 DOS 和 Windows 也是采用树状结构，但是在 DOS 和 Windows 中这样的树状结构的根是磁盘分区的盘符，有几个分区就有几个树状结构，它们之间的关系是并列的。但在 Linux 中，无论操作系统管理几个磁盘分区，这样的目录树都只有一个。

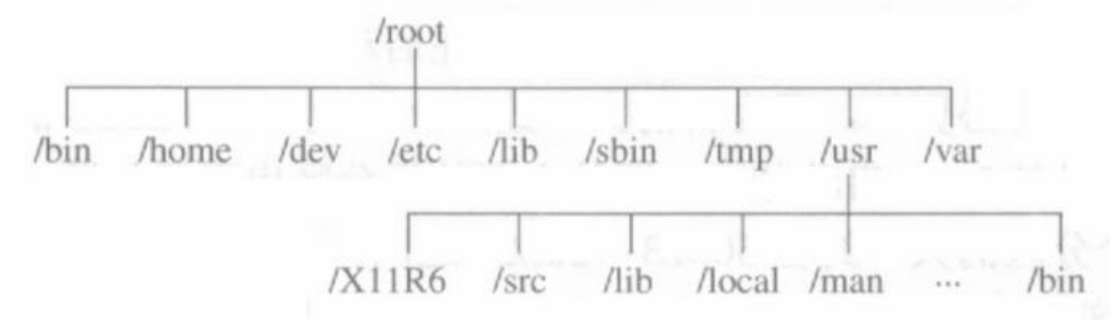


图 1 -3 Linux 的目录结构

Linux 使用标准的目录结构，在安装的时候，安装程序就已经为用户创建了文件系统和完整而固定的目录结构，并指定了每个目录的作用和其中的文件类型。

１．目录功能简介

1)．/bin

存放常用的二进制可执行命令，如 1s，mv，rm，mkdir，rmdir，gzip，tar，elnet 及 ftp 等。通常它与/usr/bin 的内容是一样的。

2)．/dev

存放与设备有关的特殊文件。基本上 UNIX 或 Linux 系统都将设备当成文件，如/dev/fd0 代表软盘，/dev/cdrom 则表示光盘。

3)．/etc

存放系统管理和配置文件，如 LILO 的参数、用户的账号和密码，以及系统的主要设置。

4)．/home

为用户设置的目录，比如用户 user 的主目录就是/home/user，可以用~user 表示。

5)．/lib

标准程序设计库，又叫动态链接共享库，在 Linux 执行或编译内核时，均会用到。

6)． /sbin

系统管理命令，存放的是系统管理员使用的管理程序，如 fdisk，mke2fs，fsck，mkswap，

mount 等等。

7)．/boot　　，

放置 Linux 核心与启动和关闭系统有关的文档，一个在后面的实验中会使用的非常重要的目录。

8)．/temp

公用的临时文件存储点。

9)．/root

系统管理员的主目录。

10)．/mm

系统提供这个目录是让用户临时装载其他的文件系统，如装载软盘的文件系统。

11)．/lost+found

这个目录平时是空的，系统非正常关机时而留下的文件会放在这里。类似于 Windows 下的.chk 文件。

12)．/proc

虚拟的目录，是系统内存的映射。可直接访问这个目录来获取系统信息。

13)．/var

这是系统在工作时预先设置的工作目录，如各种服务的日志文件和收发的邮件等。

14)．/usr

最庞大和最重要的目录之一，要用到的应用程序和文件几乎都在这个目录下。其中包含：

- /usr/X11R6　存放 Xwindow 的目录。
- /usr/bin　　众多的应用程序。
- /usr/sbin　　超级用户的一些管理程序。
- /usr/doc　　Linux 系统的说明文档(RedHat7．0 以后改放在/usr/share/doc 下)。
- /usr/include　Linux 下开发和编译应用程序所需要的头文件。
- /usr/lib　　存放常用的动态链接库和软件包的配置文件。
- /usr/man　　存放帮助文档(RedHat7．0 以后放在/usr/share/man 下)。
- /usr/src　　Linux 内核的源代码就放在这里，编译内核时必须用到。
- /usr/local/bin 本地增加的命令，通常用于软件的升级。
- /usr/local/lib 本地增加的库。

1.7.开发环境

1.6.1.如何编译简单的程序

C 语言是 Linux 下的最常用的程序设计语言，Linux 上的很多应用程序就是用 C 语言编写的。实验中的编程都是使用 C 来实现的。Linux 系统上运行的 GNUC 编译器 (GCC)是一个全功能的 ANSIC 兼容编译器。虽然 GCC 没有集成的开发环境，但堪称是目前效率很高的 C/C++编译器。

GCC 安装后的目录结构为：

/usr/lib/gcc-lib/target/version/编译器就在这个目录(及子目录)下。

/usr/birdgcc 可以从命令行执行的二进制程序在这个目录下。

/usr/target/(binIUblinclude)/库和头文件在这个目录下。

/lib/、/usr/lib 和其他目录系统的库在这些目录下。

命令格式

gcc [选项] 源文件 [目标文件]

选项含义

-o FILE 指定输出文件名，在编译为目标代码时，这一选项不是必须的。如果 FILE 没有指定，默认文件名是 a.out.

-c GCC 仅把源代码编译为目标代码。默认时 GCC 建立的目标代码文件有一个.o 的扩展名。

-static 链接静态库，即执行静态链接。

-O GCC 对源代码进行基本优化。这些优化在大多数情况下都会使程序执行得更快。

-On 指定代码优化的级别为 n，n 为 0≤n≤3 的正整数。-O2 选项告诉 GCC 产生尽可能小和尽可能快的代码。-O2 选项使编译的速度比使用-O 时慢，但通常产生的代码执行速度会更快。

-g 在可执行程序中包含标准调试信息。GCC 产生能被 GNU 调试器使用的调试信息，以便调试你的程序。GCC 提供了一个很多其他 C 编译器没有的特性，在 GCC 里可以把-g 和-O(产生优化代码)联用。

-pedantic，允许发出 ANSI/ISOC 标准所列出的所有警告。

-pedantic-errors 允许发出 ANSI/ISOC 标准所列出的所有错误。

-w 关闭所有警告，建议不要使用此项。

-Wall 允许发出 GCC 能提供的所有有用的警告，也可以用-W(warning)来标识指定的警告。

-MM 输出一个 make 兼容的相关列表。

-v 显示在编译过程中的每一步用到的命令。

例 编译 hello.c 源文件：

$ gcc hello.c –o hello

将 hello.cpp 编译为目标代码

$gcc –c hello.cpp –o hello.o

链接指定的函数库的目标代码和包含头文件，编译程序：

$gcc myapp.c –L/home/xuhong/lib –I /home/xuhong/include –lnew –o myapp

1.6.2.大型程序的编译

当用户编写大型程序时，按照 1.6.1 节的方法写进行编译的话，重新编译一次其工作量相当巨大。而 Make 命令是可自动决定一个大程序中哪些文件需要重新编译，并发布重新编译它们的命令。在 Linux 下得到广泛的应用。如果要使用 Make，必须先写一个称为 Makefile 的文件，该文件描述程序中各个文件之间的相互关系，并且提供每一个文件的更新命令。在一个程序中，可执行程序文件的更新依靠 OBJ 文件，而 OBJ 文件是由源文件编译得来的。一旦合适的 Makefile 文件存在，每次更改一些源文件，在 shell 命令下简单的键入：

make

就能执行所有的必要的重新编译任务。Make 程序根据 Makefile 文件中的数据和每个文件更改的时间戳决定哪些文件需要更新。对于这些需要更新的文件，Make 基于 Makefile 文件发布命令进行更新，进行更新的方式由提供的命令行参数控制。这些对于维护规模大

的程序来说是相当有用的。Make 程序需要一个所谓的 Makefile 文件来告诉它干什么。在大多数情况下，Makefile 文件告诉 Make 怎样编译和连接成一个程序。

我们将先给出 Makefile 的规则的格式，然后讨论一个简单的 Makefile 文件，该文件描述怎样将 8 个 C 源程序文件和 3 个头文件编译和连接成为一个文本编辑器。

1 规则的格式

一个简单的 Makefile 文件包含一系列的“规则”，其样式如下：

目标(target)…: 依赖(prerequiries)…

<tab>命令(command)

…

…

目标(target)通常是要产生的文件的名称，目标的例子是可执行文件或 OBJ 文件。目标也可是一个执行的动作名称，诸如‘clean’（详细内容请参阅假想目标一节）。

依赖是用来输入从而产生目标的文件，一个目标经常有几个依赖。

命令是 Make 执行的动作，一个规则可以含有几个命令，每个命令占一行。注意：每个命令行前面必须是一个 Tab 字符，即命令行第一个字符是 Tab。这是不小心容易出错的地方。

通常，如果一个依赖发生变化，则需要规则调用命令对相应依赖和服务进行处理从而更新或创建目标。但是，指定命令更新目标的规则并不都需要依赖，例如，包含和目标‘clern’相联系的删除命令的规则就没有依赖。

规则一般是用于解释怎样和何时重建特定文件的，这些特定文件是这个详尽规则的目标。Make 需首先调用命令对依赖进行处理，进而才能创建或更新目标。当然，一个规则也可以是用于解释怎样和何时执行一个动作。

一个 Makefile 文件可以包含规则以外的其它文本，但一个简单的 Makefile 文件仅仅需要包含规则。虽然真正的规则比这里展示的例子复杂，但格式却是完全一样。

2 一个简单的 Makefile 文件

一个简单的 Makefile 文件，该文件描述了一个称为文本编辑器（edit）的可执行文件生成方法，该文件依靠 8 个 OBJ 文件（.o 文件），它们又依靠 8 个 C 源程序文件和 3 个头文件。

在这个例子中，所有的 C 语言源文件都包含‘defs.h’ 头文件，但仅仅定义编辑命令的源文件包含‘command.h’头文件，仅仅改变编辑器缓冲区的低层文件包含‘buffer.h’头文件。

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

main.o : main.c defs.h
         cc -c main.c
kbd.o : kbd.c defs.h command.h
         cc -c kbd.c
```

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

我们把每一个长行使用反斜杠-新行法分裂为两行或多行，实际上它们相当于一行，这样做的意图仅仅是为了阅读方便。使用 Makefile 文件创建可执行的称为‘edit’的文件，键入

make

使用 Makefile 文件从目录中删除可执行文件和目标，键入：

make clean

在这个 Makefile 文件例子中，目标包括可执行文件‘edit’和 OBJ 文件‘main.o’及‘kdb.o’。依赖是 C 语言源文件和 C 语言头文件如‘main.c’和‘def.h’等。事实上，每一个 OBJ 文件即是目标也是依赖。所以命令行包括‘cc -c main.c’和‘cc -c kbd.c’。

当目标是一个文件时，如果它的任一个依赖发生变化，目标必须重新编译和连接。任何命令行的第一个字符必须是‘Tab’字符，这样可以把 Makefile 文件中的命令行与其它行分别开来。（一定要牢记：Make 并不知道命令是如何工作的，它仅仅能向您提供保证目标的合适更新的命令。Make 的全部工作是当目标需要更新时，按照您制定的具体规则执行命令。）

目标‘clean’不是一个文件，仅仅是一个动作的名称。正常情况下，在规则中‘clean’这个动作并不执行，目标‘clean’也不需要任何依赖。一般情况下，除非特意告诉 make 执行‘clean’命令，否则‘clean’命令永远不会执行。注意这样的规则不需要任何依赖，它们存在的目的仅仅是执行一些特殊的命令。象这些不需要依赖仅仅表达动作的目标称为假想目标。详细内容参见假想目标;参阅命令错误可以了解 rm 或其它命令是怎样导致 make 忽略错误的。

1.6.3.调试程序

无论是多么优秀的程序员，都难以保证自己在编写代码时不会出现任何错误，因此调试是软件开发过程中的一个必不可少的组成部分。当程序完成编译之后，它很可能无法正常运行，或者会彻底崩溃，或者不能实现预期的功能。此时如何通过调试找到问题的症结所在，就变成了摆在开发人员面前最严峻的问题。对于 Linux 程序员来讲，目前可供使用的调试器非常多，GDB（GNU DeBugger）就是其中较为优秀的。

gdb 是一个用来调试 C 和 C++ 程序的强力调试器. 它使你能在程序运行时观察程序的内部结构和内存的使用情况。以下是 gdb 所提供的一些功能：

它使你能监视你程序中变量的值；

它使你能设置断点以使程序在指定的代码行上停止执行；

它使你能一行行的执行你的代码。

在命令行上键入 gdb 并按回车键就可以运行 gdb 了, 如果一切正常的话，gdb 将被启动并且你将在屏幕上看到类似的内容：

GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for
details.
GDB 4.14 (i486-slakware-linux), Copyright 1995 Free Software Foundation, Inc.
(gdb)

当你启动 gdb 后, 你能在命令行上指定很多的选项. 你也可以以下面的方式来运行 gdb：

gdb <fname>

当你用这种方式运行 gdb，你能直接指定想要调试的程序。这将告诉 gdb 装入名为 fname 的可执行文件。你也可以用 gdb 去检查一个因程序异常终止而产生的 core 文件, 或者与一个正在运行的程序相连。你可以参考 gdb 指南页或在命令行上键入 gdb -h 得到一个有关这些选项的说明的简单列表。

为调试编译代码(Compiling Code for Debugging)，为了使 gdb 正常工作，你必须使你的程序在编译时包含调试信息。调试信息包含你程序里的每个变量的类型和在可执行文件里的地址映射以及源代码的行号， gdb 利用这些信息使源代码和机器码相关联。

在编译时用 -g 选项打开调试选项.

gdb 基本命令

gdb 支持很多的命令使你能实现不同的功能. 这些命令从简单的文件装入到允许你检查所调用的堆栈内容的复杂命令, 表 1-5 列出了你在用 gdb 调试时会用到的一些命令. 想了解 gdb 的详细使用请参考 gdb 联机手册（命令为 man gdb）。

表 1-5. 基本 gdb 命令.

命 令 描 述

file 装入想要调试的可执行文件.

kill 终止正在调试的程序.

list 列出产生执行文件的源代码的一部分.

next 执行一行源代码但不进入函数内部.

step 执行一行源代码而且进入函数内部.

run 执行当前被调试的程序

quit 终止 gdb

watch 使你能监视一个变量的值而不管它何时被改变.

break 在代码里设置断点, 这将使程序执行到这里时被挂起.

make 使你能不退出 gdb 就可以重新产生可执行文件.

shell 使你能不离开 gdb 就执行 UNIX shell 命令.

gdb 支持很多与 UNIX shell 程序一样的命令编辑特征，你能象在 bash 或 tcsh 里那样按 Tab 键让 gdb 帮你补齐一个唯一的命令，如果不唯一的话 gdb 会列出所有匹配的命令，你也能用光标键上下翻动历史命令。

下面用一个实例教你一步步的用 gdb 调试程序。被调试的程序相当的简单，但它展示了 gdb 的典型应用。

下面列出了将被调试的程序。这个程序被称为 greeting ，它显示一个简单的问候，再用反序将它列出。

```
#include <stdio.h>

main ()
{
char my_string[] = "hello there";

my_print (my_string);
my_print2 (my_string);
}

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

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

size = strlen (string);
string2 = (char *) malloc (size + 1);
for (i = 0; i < size; i++)
string2[size - i] = string[i];
string2[size+1] = `';
printf ("The string printed backward is %s ", string2);
}
```

用下面的命令编译它：

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

这个程序执行时显示如下结果：

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

输出的第一行是正确的，但第二行打印出的东西并不是我们所期望的。我们所设想的输出应该是：

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

由于某些原因，my_print2 函数没有正常工作，让我们用 gdb 看看问题究竟出在哪儿，先键入如下命令：

gdb greeting

注意： 记得在编译 greeting 程序时把调试选项打开。

如果你在输入命令时忘了把要调试的程序作为参数传给 gdb，你可以在 gdb 提示符下用 file 命令来载入它：

(gdb) file greeting

这个命令将载入 greeting 可执行文件就象你在 gdb 命令行里装入它一样。

这时你能用 gdb 的 run 命令来运行 greeting 了。当它在 gdb 里被运行后结果大约会象这样：

(gdb) run

Starting program: /root/greeting

The string is hello there

The string printed backward is

Program exited with code 041

这个输出和在 gdb 外面运行的结果一样。问题是：为什么反序打印没有工作？为了找出症结所在，我们可以在 my_print2 函数的 for 语句后设一个断点，具体的做法是在 gdb 提示符下键入 list 命令三次，列出源代码：

(gdb) list

(gdb) list

(gdb) list

技巧：在 gdb 提示符下按回车健将重复上一个命令.

第一次键入 list 命令的输出如下：

```
1 #include <stdio.h>
2
3 main ()
4 {
5 char my_string[] = "hello there";
6
7 my_print (my_string);
8 my_print2 (my_string);
9 }
10
```

如果按下回车,gdb 将再执行一次 list 命令，给出下列输出：

```
11 my_print (char *string)
12 {
13 printf ("The string is %s ", string);
14 }
15
16 my_print2 (char *string)
17 {
18 char *string2;
```

19 int size, i;

20

再按一次回车将列出 greeting 程序的剩余部分：

21 size = strlen (string);

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

23 for (i = 0; i < size; i++)

24 string2[size - i] = string[i];

25 string2[size+1] = `';

26 printf ("The string printed backward is %s ", string2);

27 }

根据列出的源程序，你能看到要设断点的地方在第 24 行,在 gdb 命令行提示符下键入如下命令设置断点：

(gdb) break 24

gdb 将作出如下的响应：

Breakpoint 1 at 0x139: file greeting.c, line 24

(gdb)

现在再键入 run 命令，将产生如下的输出：

Starting program:/root/greeting

The string is hello there

Breakpoint 1, my_print2 (string = 0xbfffdc4 "hello there") at greeting.c :24

24 string2[size-i]=string[i]

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

(gdb) watch string2[size - i]

gdb 将作出如下回应：

Watchpoint 2: string2[size - i]

现在可以用 next 命令来一步步的执行 for 循环了：

(gdb) next

经过第一次循环后，gdb 告诉我们 string2[size - i]的值是“h”。gdb 用如下的显示来告诉你这个信息：

Watchpoint 2, string2[size - i]

Old value = 0 `00'

New value = 104 `h'

my_print2(string = 0xbfffdc4 "hello there") at greeting.c:23

23 for (i=0; i<size; i++)

这个值正是期望的，后来的数次循环的结果都是正确的。当 i=10 时，表达式 string2[size - i]的值等于“e”，size - i 的值等于 1，最后一个字符已经拷到新串里了。

如果你再把循环执行下去，你会看到已经没有值分配给 string2[0]了，而它是新串的第一个字符，因为 malloc 函数在分配内存时把它们初始化为空(null)字符。所以 string2 的第一个字符是空字符。这解释了为什么在打印 string2 时没有任何输出了。现在找出了问题出在哪里，修正这个错误是很容易的。你得把代码里写入 string2 的第一个字符的的偏移量改为 size－1，而不是 size。这是因为 string2 的大小为 12，但起始偏移量是 0,，串内的字符从偏移量 0 到偏移量 10，偏移量 11 为空字符保留。为了使代码正常工作有很多种修改办

法。一种是另设一个比串的实际大小小 1 的变量。这是这种解决办法的代码：

```
#include <stdio.h>

main ()
{
char my_string[] = "hello there";

my_print (my_string);
my_print2 (my_string);
}

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

my_print2 (char *string)
{
char *string2;
int size, size2, i;

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

第二章.进程控制

在开始真正的课程之前，我们先看一下进程在大学操作系统课本里的标准定义：“进程是可并发执行的程序在一个数据集合上的运行过程。”这个定义非常严谨，而且难懂，如果你没有一下子理解这句话，就不妨看看下面这个并不严谨的解释：我们大家都知道，硬盘上的一个可执行文件经常被称做程序，在 Linux 系统中，当一个程序开始执行后，在开始执行到执行完毕退出的这段时间里，它在内存中的部分就可以被称做一个进程。

进程是多任务操作系统的核心概念，Linux 与 Unix 进程的概念基本相似。本章先介绍进程描述符(Process Descriptor)，然后介绍进程的调度时机及调度算法，最后描述进程的创建与消亡过程。

2.1. 进程与进程描述符

Linux 是一个多任务的操作系统，也就是说，在同一个时间内，可以有多个进程同时执行。如果读者对计算机硬件体系有一定了解的话，会知道我们大家常用的单 CPU 计算机实　际上在一个时间片断内只能执行一条指令，那么 Linux 是如何实现多进程同时执行的呢?原　来 Linux 使用了一种称为“进程调度”(process scheduling)的手段。首先，为每个进程指　派一定的运行时间，这个时间通常很短，短到以毫秒为单位，然后依照某种规则，从众多进程中挑选一个投入运行，其他的进程暂时等待，当正在运行的那个进程时间片耗尽，或执行完毕退出，或因某种原因暂停时，Linux 就会重新进行调度，挑选下一个进程投入运行。因为每个进程占用的时间片都很短，所以从我们使用者的角度来看，就好像多个进程同时运行一样了。

Linux 进程与传统 Unix 进程的概念没有多大区别，Linux 也没有真正意义上的线程概念。但通过 clone()系统调用，可以支持轻量级进程(1ightweightprocess)。如果一个子进程是轻量级进程，那么它可以和父进程共享页表、打开文件表等信息，以减少创建进程时的开销。借助 clone()系统调用，Linux 有一套在用户模式下运行的线程库--pthread。

Linux 还支持内核线程的概念，内核线程永远在核心态运行，没有用户空间。页面换出、刷新磁盘缓存等工作都由内核线程完成。

为了管理进程，操作系统内核必须对每个进程的相关信息进行详细的描述，如进程的优先级、阻塞原因、打开的文件等，这正是进程描述符的作用。进程描述符又称为进程控制块，在 Linux 中由一个 task_struct 结构表示。请读者现在打开 Source lnsight，按热键 F7 浏览所有全局符号(Source lnsight 把源码中所有的宏、函数、变量、结构等名字统称为“符号”，浏览所有全局符号的功能，在阅读 Linux 内核源码的过程中是非常有用的)，然后输入符号名 “task struct'，Source lnsight 就会帮助我们找到这个结构是在 sched．h 里面定义的。这个结构非常复杂，在 2.4.20 版本的内核代码中，它有将近 100 个成员，各个成员用来准确描述进程在各方面的信息，这些信息主要包括以下几个部分。

1．进程标识

包括进程的标识号(pid)、进程的用户标识、进程的组标识等。每个进程的标识号是唯一的。

2．调度相关信息

这部分内容与进程调度有关，具体内容可参考 2.2 节和相关的书籍。另外，进程描述符中需要有结构保存当前进程被换出时寄存器的状态，以便该进程恢复运行时可从正确的

状态继续运行。

3．进程虚拟空间信息

Linux 的每个用户态进程都有自己的私有地址空间，task_struct 的成员 mm 指向一个 mm_struct 结构。

4．文件相关信息

包含进程与文件系统交互的信息，具体内容查询教科书中相关章节。

5．信号处理信息

Linux 支持传统的 Unix 信号语义。该部分记录了信号的处理函数及信号掩码等信息。

6．记帐信息及统计信息

系统的资源是有限的，每个进程对每种资源的使用都有一个限值。另外，还有统计信息来记录系统需要的信息，如页面异常次数、CPU 使用时间等。

7．描述进程间关系的指针

进程并不是孤立存在的，Linux 把所有的进程描述符通过一个双向链表链接在一起。使用宏 for_each_task 就可以在需要的时候遍历所有进程，对每个进程进行某种操作，for_each_task 的定义如下所示：

```
#define for_each_task(p)    \
        for(p=&init_task；(p=p->next_task)!=&init_task；)
```

这个宏实际上是一条循环语句，init_task 指针指向整个双向链表表头，同时它又被用来作为循环的结束条件，这充分体现出了双向链表的优势。

进程描述符还拥有指向其父进程描述符、子进程描述符、兄弟进程描述符的指针。此外很多场合需要根据 pid 号能够快速找到进程，因此，系统以 pid 为关键字建立了一个 hash 表，hash 函数值相同的进程通过进程描述符的 pidhash_next 和 pidhash_pprev 成员链接在一起。

Linux 的运行分为两种模式——核心态和用户态。内核总在核心态下运行，而普通进程通常在用户模式下运行，只有通过系统调用才能切换到核心态运行。每个进程拥有两个栈：用户模式栈与核心模式栈，顾名思义，它们分别是在用户模式和核心模式下使用的。进程描述符和进程核心栈的空间分配在一起，内核为它们分配两个连续的物理页帧。下面的 C 代码说明了这一点：

```
union task_union{
struct task_struct task；
unsigned long stack[2048]；
}  ；
```

图 2-1 可以更清楚地说明这一点。

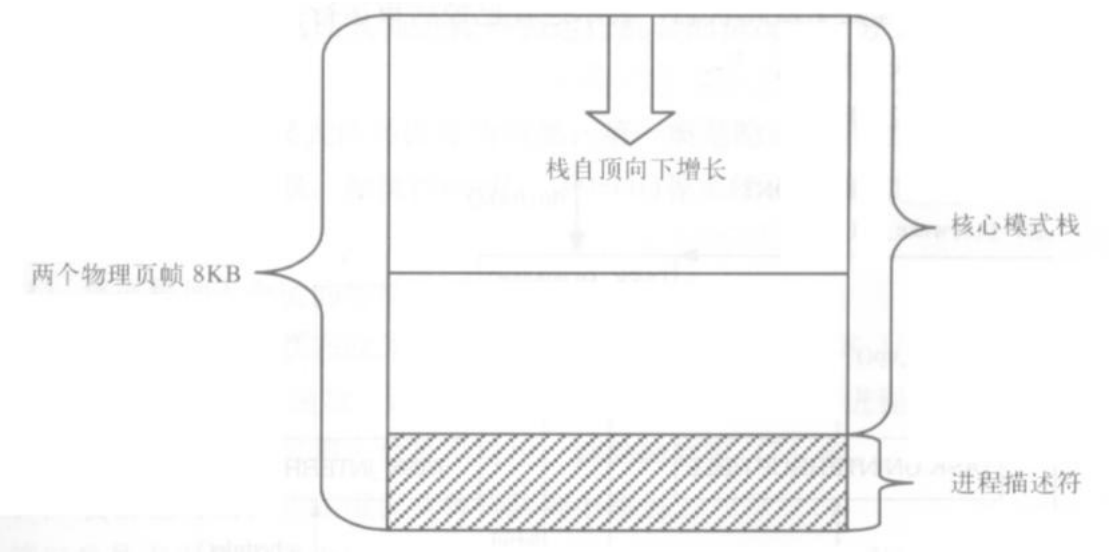


图 2-1 核心栈与进程描述符示意图

可以看到，因为进程描述符已经占用了 1KB 多的空间，所以核心栈的有效空间是 6KB 多，合理的设计使得这个容量已经足够了。

还有一个在内核源码中比较常用的宏 current。这个宏将被编译成为一条汇编指令，目的是得到当前正在处理器上执行的进程的进程描述符指针。如通过调用代码 cunent->pid，就可以得到当前正在执行的进程的 pid。

2. 2. 进程状态及切换时机

2.2.1 Linux 的进程状态

Linux 的进程状态有五种，它们分别是：

TASK_RUNNING：表示进程具备运行的资格，要么正在运行，要么就是等待被调度执行。进程描述符有一个 run_list 成员，所有处于 TASK_RUNNING 状态的进程都通过该成员链在一起，称为可运行队列。

TASK INTERRUPTIBLE 和 TASK_UNINTERRUPTIBLE：这两种状态均表示进程处于睡眠状态。进程进入睡眠状态一般是因为所请求的资源不能满足，但是，TASK_INTERRUPTIBLE 除了资源满足时可以被唤醒外，还可以被信号唤醒，而 TASK_UNINTERRUPTIBLE 就不行。

TASK_STOPPED：进程处于暂停状态，主要用于调试目的，如正在运行的进程收到 SIGSTOP 信号就将进入 TASK_STOPPED 状态。

TASK_ZOMBIE：进程处于僵死状态，表示进程已经结束运行并释放了大部分占用的资源，但 task_struct 结构还未被释放。

进程状态的转换如图 2-2 所示。

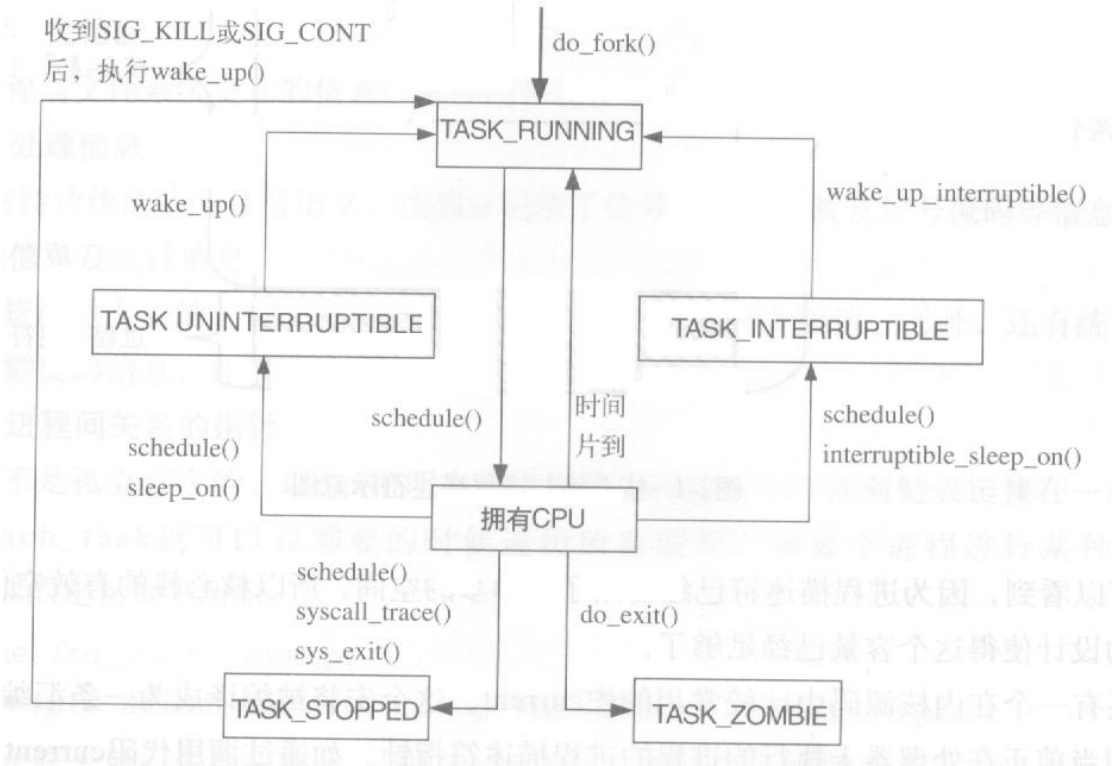


图 2-2 进程状态

为了便于管理，linux 内核使用一个“运行队列”，以链表的形式把所有处于 TASK_RUNNING 状态的进程联结在一起。对于其他状态的处理，Linux 采用了下面的方法来分别处理：

TASK_STOPPED 或者 TASK_ZOMBIE 状态的进程，没有一个专门的链表来维护它们的信息，也没有对它们进行分组。因为父进程要得到子进程的 ID 是很容易的，不需要遍历链表。

TASK_INTERRUPTIBLE 和 TASK_UNINTERRUPTIBLE 这两种状态的进程被细分为几类，每一类对应的是一种特定事件，并且采用链表的方法，形成多个等待队列。这样，操作系统可以在事件发生的时候遍历等待队列，以便唤醒恰当的进程，把该进程转化为 TASK_RUNNING 状态。等待队列在内核中的用途非常广泛，特别是在中断处理、进程同步或者定时的时候，尤其发挥着重要的作用，本书将在后面章节中详细讨论等待队列及其应用。

2.2.2.进程的切换时机

当前进程放弃 CPU 从而其他进程得到运行机会的情况可以分为两种：进程主动地放弃 CPU 和被动地放弃 CPU。

进程主动放弃 CPU 又大体可以分为两类：第一类是隐式地主动放弃 CPU。这往往是因为需要的资源目前不能获取，如执行 read()、select()等系统调用的过程中。这种情况下的处理过程如下：

(1)将进程加入合适的等待队列。

(2)把当前进程的状态改为 TASK_INTERRUTIBLE 或 TASlUNINTERRUTIBLE。

(3)调用 schedule()函数，该函数的执行结果往往是令当前进程放弃 CPU。

(4)检查资源是否可用，如果不可用，则跳转到第(2)步。

(5)资源已可用，将该进程从等待队列中移去。

第二类是进程显式地主动放弃 CPU，如系统调用 sched_yield()、sched_setscheduler()、pause()及 nanosleep()均会导致当前进程让出 CPU。

进程被动放弃 CPU 又分成两种情形，其一是当前进程的时间片已经用完，其二是刚被唤醒的进程的优先级别高于当前进程。两种情形均会导致当前进程描述符的 need_resched 被置 1。

从进程调度时机的角度来讲，也可以分成两种情形。一种是直接调用 schedule()调度函数，例如上面提到的进程主动放弃 CPU 的第一类情形。另一种是间接调用 schedule()调度函数，例如进程被动放弃 CPU 的情形。当进程描述符的 need_resched 被置 1 时，并不立即直接调用 schedule()调度函数。而是在随后的某个时刻，当进程从内核态返回用户态之前检查 neelresched 是否为 1，如果为 1，则调用 schedule()函数，开始重新调度。

下面我们来看看 schedule()函数是怎样完成进程调度的。

2.3 进程的调度算法

我们在前面看到，Linux 的进程调度的关键函数是 schedule()，该函数的作用是选出一个可运行的进程，并且让该进程开始占用 CPU。

在 2.1 节中介绍了进程描述符，也就是 Linux 内核的 task_struct 这个结构。该结构的如下成员变量与调度有关：

(1)po1icy 标识进程的调度策略。有以下三种类型：

SCHED_OTHER：普通进程。

SCHED_FIFO：实时进程，采用先进先出的调度算法。当调度程序把处理器分配给一个进程时，该进程的描述符就留在运行队列链表的当前位置，可以一直使用处理器直到被更高优先级的实时进程所取代。

SCHED_RR：实时进程，采用轮转调度算法。当调度程序把处理器分配给一个进程的时候，这个进程的描述符被放在运行队列末尾，确保处理器的时间可以公平分配给所有同一优先级的 SCHED_RR 进程。

(2)rt_priority 实时进程的优先级，普通进程不使用这个成员。

(3)nice 普通进程的优先级。

(4)counter 进程目前的 CPU 时间配额。

对于普通进程来讲，CPU 时间的分配是典型的时分策略。在某个时刻，运行队列中的每个进程都有一个 counter 值，当所有运行队列中的 counter 值都变为 0 以后，表明一轮已经结束，每个进程的 counter 根据其 nice 重新赋值，然后开始新的一轮执行过程。拥有 CPU 的进程每次时钟中断时 counter 值减一。

我们打开 Source lnsight，查找符号名 schedule，可以看到 schedule()函数的代码在文件 sched.c 中，其执行过程大致如下：

(1)检查是否有软中断服务请求，如果有，则先执行这些请求。

(2)若当前进程调度策略是 SCHED_RRN_counter 为 0，则将该进程移到可执行进程队列的尾部并对 counter 重新赋值。

(3)检查当前进程的状态，如果为 TASK_INTERRUPTIBLEJi 该进程有信号接收，则将进程状态置为 TASK_RUNNING；若当前进程的状态不是 TASK_RUNNING，则将其从可执行进程队列中移出。然后将当前进程描述符的 need_resched'陕复成 0。

(4)现在进入了函数的核心部分。可运行进程队列的每个进程都将被计算出一个权值，权值主要是利用 goodness()函数计算的，关于 goodness()函数的讨论后面将详细描述。最终最大的权值保存在变量 c 中，与之对应的进程描述符保存在变量 next 中。

(5)检查 c 是否为 0。若为 0，则表明所有可执行进程的时间配额都已用完，此时对所有进程的 counter 重新“充电”，然后重新执行第(5)步。

(6)如果 next 进程就是当前进程，则结束 shedule()函数的运行。否则进行进程切换，CPU 改由 next 进程占据。goodness()函数的代码也在 sched.c 中，它计算进程的当前权值。该函数的第一个参数是待估进程的描述符，返回值 c 则比较真实地反映了待估进程“值得运行的程度”。c 的取值范围如下所示：

c=-1000：永远不必选择待估进程。当运行队列里只有一个进程时，选择该值。

c=0：待估进程的时间片用完，在其他进程的时间片用完之前不会选择它。

0<c<1000：待估进程的时间片还没有用完，剩余的时间片可以看做优先级。

c>1000：待估进程是实时进程，应该优先执行。

如果该进程是实时进程，它的权值为 1000+rt_priority，1000 是普通进程权值无法到达的数字，因而实时进程总可以优先得到执行。对于普通进程，它的权值为 counter+20-nice，如果它又是内核线程，由于无须切换用户空间，则将权值加一作为奖励。

如果需要切换进程，则调用 switch_to()这个宏，开始执行新的进程。可以想像，switch_to()的代码应该是和处理器相关的，因此，这个宏是用汇编语言实现的，主要做了保存当前现场、恢复新进程的现场等工作。对系统底层实现比较感兴趣的读者可以自行参考这部分代码，用 Sourcelnsight 即可轻松找到 switch_to()宏的定义在 System.h 中。

一个优秀的调度算法必须在算法的计算量和算法的有效性之间做出平衡，因此，很难构造一种理论模型，证明某种算法是较优的，而且也没有哪种算法在任何情况下都是最优的。Linux 内核的调度算法能够适应一般的应用，但是在实时系统中，或者是进程数量很大的情况下，并不能发挥出最高的效率。这时可以重新改写该算法并重新编译内核，以适应那些特殊的场合。

2．4 进程的创建与消亡

2．4．1　进程的创建

我们在使用 Unix 或者 Linux 系统的时候，每次在终端下面输入一行命令，就由 shell 进程接收这个命令并创建一个新的进程，这个新的进程还可以通过 fork()系统调用，继续创建自己的子进程，系统中的多个进程构成了一棵进程树。那么，细心的读者一定会考虑到：这棵树的根是哪个进程呢?事实上，在 Linux 系统启动的时候，最早产生的进程是 idle 进程，其 pid 号为 0，该进程会创建一个内核线程，该线程进行一系列初始化动作后最终会执行/sbin/init 文件，执行该文件的结果是运行模式从核心态切换到了用户态，该线程演变成了用户进程 init，pid 号为 1。init 进程是一个非常重要的进程，一切用户态进程都是它的后代进程。

我们可以用 pstree 命令查看进程树，看到的将是如下所示的内容，其中 init 进程作为整棵树的根出现。

```
init(1)-+-crond(98)
|—emacs(387)
|—gpm(146)
|—inetd(110)
|—kerneld(18)
|—kflushd(2)
|—klogd(87)
```

```
├─kswapd(3)
├─login(160)─────bash(192)─────emacs(225)
├─lpd(121)
├─mingetty(161)
├─mingetty(162)
├─mingetty(163)
├─mingetty(164)
├─login(403)─────bash(404)─────pstree(594)
├─sendmail(134)
├─syslcgd(78)
```

fork/exec 是典型的 Unix 新进程的产生模式。通常先用 fork()创建一个新进程，然后新进程通过调用 exec 系列函数执行真正的任务。下面是一段示例代码：

```
#include <stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main(void)
{
pid_t   pid;
if((pid=fork())<0){
      printf(”fork failed＼n”);
exit(1);
)
else if(pid==0){/*子进程执行进入此部分*/
execlp("echoall"，"echoall"，(char*)0);
}
else{   /*父进程*/
printf("fork successed!＼n");
exit(0);
}
}
```

如果函数 fork()调用成功，当前进程就拥有了一个子进程。该函数返回两个值，其中在子进程中返回 0，在父进程中返回的是子进程的 pid 值。

事实上，可以实现创建子进程的系统调用有三个：clone()、vfork()和 fork()。对于 fork()和 clone()这两个系统调用，如果读者有 Linux 的程序设计经验的话，应该不会陌生。fork()采用了“写时拷贝”的策略，允许父子进程能读相同的物理页，但是如果两者当中任何一个试图对某个物理页进行写操作，那么内核就建立一个新的物理页并将旧的物理页的内容完全复制过去；clone()的不同之处在于它允许父子进程共享所有内核中的数据结构，包括页表和打开文件表，这样，clone()创建的是一种“轻量级进程”，类似于其他操作系统中“线程”的概念。

vfork()则是一个比较老的函数调用，子进程共享父进程的空间包括页表，而父进程被挂起直到子进程执行 exec 系列函数或子进程退出时为止。

由于在大多数情况下，fork()或者 vfork()系统调用之后马上要进行的系统调用就是 exec 系列。因此，在合适的场合，与 fork()时的“写时拷贝” 策略相比，vfork()无疑开销更小。

这三个系统调用最终都会调用 do_fork()函数完成主要工作。该函数的第一个参数

clone_flags 可由多个标志位组成，常见的标志位有：

CLONE_VM　　子进程、父进程共享进程空间

CLONE_FS　　子进程、父进程共享文件系统信息

CLONE_FILES　　子进程、父进程共享打开的文件

CLONE_VFORK　　父进程想在子进程释放空间时被唤醒

clone()对应的 clone_flags 可能是多个标志位的组合，这取决于具体情况。

vfork()对应的 clone_flags 值是 CLONE_VFORK，CLONE_VM 和 SIGCHILD 这三个值的组合。

fork()对应的 clone_flags 值是 SIGCHILD，SIGCHILD 的作用是子进程终结或暂停时给父进程发信号。

我们可以用 Sourcelnsight 找到 do_fork()函数在 fork.c 文件中，其执行过程大致如下：

(1)调用 alloc_taslstruct~》为子进程描述符分配空间。严格地讲，此时子进程还未生成。

(2)把父进程描述符的值全部赋给子进程描述符。

(3)检查是否超过了资源限制，如果是，则结束并返回出错信息，更改一些统计量的信息。

(4)修改子进程描述符中某些成员的值，使其正确反映子进程的状况，如进程状态被置成 TASK_UNINTERRUPTIBLE。

(5)调用 get_pid()函数为子进程得到一个 pid 号。

(6)依次调用 copy_files()、copy_fs()、copy_sighand()、copy_mm()来分别复制父进程的文件处理、信号处理及进程空间的信息。以上函数的具体行为取决于 clone_flags 参数，例如调用 copy_mm()时，如果 clone_flags 包含 CLONE_VM 标志，则子进程共享父进程的空间，而不会进行复制。关于 copy_mm()可参看本书中内存管理相关的内容。

(7)调用 copy_thread()初始化子进程的核心模式栈，核心栈保存了进程返回用户空间的上下文。此处与平台相关，以 i386 为例，其中很重要的一点是存储寄存器 eax 值的位置被置 0，这个值就是执行系统调用后子进程的返回值。

(8)将父进程当前的时间配额 counter 分一半给子进程。

(9)利用宏 SET_LINKS 将子进程插入所有进程都在其中的双向链表。调用 hash_pid()函数将子进程加入相应的 hash 队列。

(10)调用 wake_up_process()将该子进程插入可运行队列。至此，子进程创建完毕并在可运行队列中等待被调度运行。

(11)如果 clone_flags 包含 CLONE_VFORK 标志，则将父进程挂起直到子进程释放进程空间。进程描述符中有一个信号量 vforlsem 可以起到将进程挂起的作用。

(12)返回子进程的 pid 值，该值就是系统调用后父进程的返回值。

2.4.进程的销毁

进程运行结束后，需要被操作系统销毁。最普遍的情况是，进程正常运行结束后显式或隐式地调用 exit()函数(所谓隐式调用，是指编译器会在程序的结束点自动插入一个对 exit 函数的调用)。或者，当进程收到某种信号时，该信号的处理函数会结束当前进程运行并将其销毁，对于最常使用的 shell 命令之一 kill，在不指定信号的情况下，默认信号 SIGTERM 将被发给相应进程并导致该进程结束运行。另一种常见的情况是，用户程序访问了非法地址空间，使内核向进程发送 SIGSEGV 信号从而导致进程结束运行。无论进程因为什么原因而结束，进程销毁的最终动作都是调用函数 do_exit()。

do_exit()函数的实现代码在文件 exit.c 中，其执行流程大致如下：

(1)设置标志，表明进程正在被销毁。

(2)如果进程在定时器队列或信号量队列中等待，则将其移出。

(3)调用_exit_mm()、_exit_files()、_exit_fs()和 exit_sighand()释放进程所占用的各种资源。被释放的资源一般先将其共享计数减一，如果此时还有别的进程使用该资源，则共享计数不为 0，此时直接返回，如果为 0，才真正释放资源。_exit_mm()的动作是释放进程地址空间，过程中会检测该进程是否由 vfork()创建，如果是，则唤醒父进程。

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

(5)调用 schedule()调度函数切换到别的进程。

在 do_exit()执行后，进程处于 TASK_ZOMBIE 状态，也就是僵死状态，只留下了一个 task_struct 结构，之所以不立即释放该结构是因为父进程往往要通过 wait()之类的系统调用来检查子进程是否结束，而调用之前显然不能让子进程消失得无影无踪。wait()系统调用会回收处于 TASK_ZOMBIE 状态的子进程的 task_struct 结构及 pid 号，这时子进程才会退出僵死状态，从系统中彻底消失。

当然，父进程可能先于子进程结束，这时 init 进程就变成了子进程的父进程，从而保证 task_struct 结构总能得到回收。

第三章.SYS V 进程间通信

SYS V(SystemV)是传统 Unix 的一大流派，像 POSIX 一样提出了一系列的标准，其中包括进程间通信的标准。Linux 支持多种进程间通信的方式，如管道、信号、SYS V 进程间通信(1PC)和套接字等。本章主要介绍 SYSV 进程间通信的三种标准：信号量、消息队列和共享内存，以及 Linux 对这三种标准的支持。SYS V 进程间通信的内核实现在接口和数据结构两方面与库函数有许多相似之处，在下面的内容中请注意这两者之间的区别。

3.1.信号量、消息队列和共享内存的共同特性

我们首先来看一下消息队列、信号量、共享内存在操作接口上的共同之处。三者均有 XXXget()函数及 XXXctl()函数，其中 XXX 代表 msg、sem、shm 之中的任何一者，这里只讨论函数共同拥有的参数或标志位，特定的则不涉及。

下面使用术语 IPC 资源来表示单独的消息队列、共享内存或是信号量集合。

XXXget()函数有两个共同的参数：key 和 ofiag。key 既可由 ftok()函数产生，也可以是 IPC_PRIVATE 常量，key 值是 IPC 资源的外部表示。oflag 包括读写权限，还可以包含 IPC_CREATE 和 IPC_EXCL 标志位。它们组合的效果如下：

(1)指定 key 为 IPC_PRIVATE，保证创建一个唯一的 IPC 资源。

(2)设置 ofiag 参数的 IPC_CREATE 标志位，但不设置 IPC_EXCL。如果相应 key 的 IPC 资源不存在，则创建一个 IPC 资源，否则返回已存在的 IPC 资源。

(3)oflag 参数的 IPC_CREATE 和 IPC_EXCL 同时设置。如果相应 key 的 IPC 资源不存在，则创建一个 IPC 资源，否则返回一个错误信息。

XXXctl()均提供 IPC_SET、IPC_STAT 和 IPC_RMID 命令。前两者用来设置或得到 IPC 资源的状态信息，IPC_RMID 用来释放 IPC 资源。

信号量、消息队列和共享内存都是先通过 XXXget()创建一个 IPC 资源，返回值是该 IPC 资源的 ID。在以后的操作中，均以 IPC 资源 ID 为参数以对相应的 IPC 资源进行操作。注意，IPC 资源 ID 不是局限于进程的，而是全局的。只要权限允许，别的进程就可以通过 XXXget()取得已有的 IPC 资源 ID 并对其操作，从而使进程间通信成为可能。

在内核中，每一类 IPC 资源都有一个 ipc_ids 结构的全局变量用来描述同一类资源的公有数据。这三个全局变量分别是 semid_ds、msgid_ds 和 shmid_ds。

ipc_ids 结构的定义如下：

```
struct ipc_ids
{
int       size;        /*entries 数组的大小*/
int       in_use;      /*entries 数组已使用的元素个数*/
int       max_id;
mnsigned  short       seq;
mnsigned short        seq_max;
struct semaphore      sem;         /*内核信号量，控制对 ipc_ids 结构的访问*/
spinlock_t        ary;         /*自旋锁，控制对数组 entries 的访问*/
struct    ipc_id      *entries;
)  ;
```

```
structipc__id
{
struct   kern_ipc_perm*   p;
)  ;
```

数组 entries 的每一项指向一个 kern_ipc_perm 结构。kern_ipc_perm 结构表示每一个 IPC 资源的属性，用来控制操作权限。其定义如下：

```
struct   kern_ipc_perm
{
key_t      key;       /*键值，由用户提供，为 XXXget()所用
uid_t      uid;       /*创建者的用户 ID*/
gid_t      gid;       /*创建者的组 ID*/
uid_t      cuid;      /*所有者的用户 ID*/
gid_t      cgid;      /*所有者的组 ID*/
mode_t     mode;      /*操作权限，包括读、写等*/
unsigned   long      seq;
)  ;
```

因为每一个 IPC 资源均拥有一个 kern_ipc_perm 结构，而且每个 IPC 资源描述符的第一个成员就是 kern_ipc_perm 结构。因此，可以认为数组 entries 的每一非空项均指向一个 IPC 资源。当创建一个 IPC 资源时，均会调用函数 ipc_addid()从相应 ipc_ids 结构的 entries 数组中找出第一个未使用的项，然后返回其下标 index。注意，index 并不是返回的 IPC 资源 ID，但它们之间存在如下关系：

IPC 资源 ID = SEQ_MULTIPLIER * seq + index

SEQ_MULTIPLIER 是可用资源的最大数目，seq 是 ipc_ids 结构中的 seq。每当分配一个 IPC 资源时，ipc_ids 结构中的 seq 就增一。

当知道 IPC 资源 ID 时，可通过公式：(IPC 资源 ID ％ SEQ_MULTIPLIER)(a％b 得到 a 除以 b 的余数)得到其在 entries 数组中的 index，从而找到相应的 IPC 资源。

采用上述办法的理由是，IPC 资源(包括 ID 号)是一个动态申请和释放的资源，不希望 ID 号被释放后又立即被分配给别的资源，因为这样可能会导致误用。

3.2.信号量

信号量是具有整数值的对象，它支持 P/V 原语。进程可以利用信号量实现同步和互斥。需要注意的是，SYSV 信号量是用户空间的操作，最终需要内核的同步机制支持。

SYS V 支持的信号量实质上是一个信号量集合，由多个单独的信号量组成。在后面的叙述中，SYS V 信号量称为信号量集合，而单个的信号量则直接称为信号量。支持信号量集合的原因是，进程往往需要在同时具备多个资源的情况下才能工作，典型的例子是银行家算法。

信号量集合在内核中用结构 sem_array 表示，其定义如下：

```
struct   sem_array
{
struct   kern_ipc_perm   sem_perm;
time_t      sem_otime;      /*最近一次的操作时间*/
time_t      sem_ctime;      /*最近一次的改变时间*/
```

```
atruct  sem     *sem_base;
struct sem_queue     *sem_pending;      /*挂起操作队列*/
struct  sem_queue     **sem_pending_last;
structsem_undo     *undo;
unsignedlong    sem_nsems;
}  ;
```

sem_base 指向第一个信号量，sem_nsems 表示信号量的个数。信号量集合中的每一个信号用结构 sem 表示，其定义如下：

```
struct  sem
  {
    int     semval;      /*信号量的当前值*/
    int     sempid;      /*最近对信号量操作的进程的 pid*/
    }  ;
```

信号量的初始值可以调用函数 semctl()进行设置，用户可以调用函数 semop()对信号量集合中的一个或多个信号量进行操作。semop()函数原型如下：

```
int  semop(int  semid,  struct  sembuf  *opsptr,  size_t  nops);
```

其中，参数的含义如下所示：

semid　IPC 资源 ID

opsptr　操作的集合

nops　数组 opsptr 的大小

需要强调的是，内核必须保证操作数组 opsptr 原子地执行；要么完成所有的操作，要么什么也不做。每一个操作都是 sembuf 结构变量，该结构在函数库和内核中均有定义，其定义如下：

```
struct  sembuf
{
unsigned shott sem num；/*在 sem_array.sem_base[]数组中的下标*/
short     sem_op;
short     sem_flg;
}  ;
```

sem_num 指明是对哪一个信号操作。

sem_flg 指明一些操作标志位，可以有下述值：

SEM_UNDO　当进程结束但还拥有信号量资源时，应将信号量资源返还给相应的信号量集合。内核有一个 sem_undo 结构用于跟踪这方面的情况，进程描述符有一个 sem_undo 成员记录进程这方面的信息。在下面关于 sem_op 的讨论中省略掉了这方面的信息

IPC_NOWAIT　当操作不能立即完成时，若 IPC_NOWAIT 被设置，进程就立即返回：否则，进程进入睡眠状态等待时机成熟时被唤醒完成该操作

sem_op 指定具体的操作，它的值有如下含义：

(1)若大于 0，则将该值加到信号量的当前值上。

(2)若等于 0，则用户希望信号量的当前值变为 0。如果值已经是 0，则立即返回。如果不是 0，则取决于 IPC_NOWAIT 是否被设置。

(3)若小于 0，则要看信号量的当前值是否大于等于 sem_op 的绝对值。如果大于等于，就从信号量的当前值中减去 sem_op 的绝对值。如果小于，则取决于 IPC_NOWAIT 是否被设置。当进程的信号量操作不能完成睡眠时，需要将一个代表着当前进程的 sem_queue 结构链入相应的信号量集合的等待队列，即 sem_array 结构的 sem_pending 队列。sern_queue

结构定义如下：

```
struct    sem_queue
{
struct sem_queue       *next/       /*队列中的下一个元素*/
struct sem_queue       **prev;        /*队列中的前一个元素*/
struct    task_struct    *sleeper;     /*睡眠进程的描述符*/
struct sem_undo       *undo;
int       pid;         /*睡眠进程的 pid*/
int       status;
struct sem_array       *sma;         /*sem_queue 结构所属的信号量集合*/
int       id;
struct sembuf       *sops;         /*挂起的操作数组*/
int       nsops;         /*挂起的操作个数*/
int       alter;
}   ;
```

图 3-1 显示了上述数据结构之间的关系。

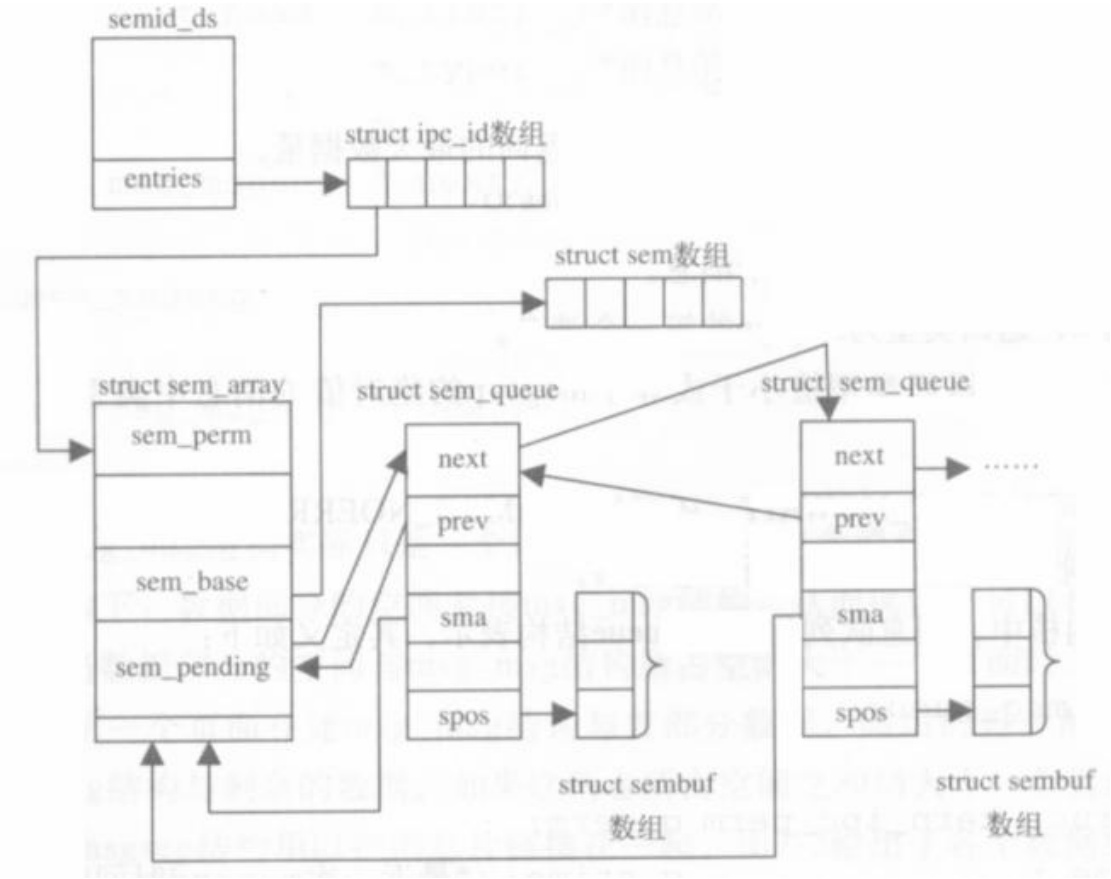


图 3-1 信号量各数据结构之间的关系

3.3.消息队列

具有写入权限的进程可以往消息队列中写入一个消息，而具备读出权限的进程则可以

从消息队列中读出一个消息，这就是消息队列支持进程通信的方式。

除前面提到的两个函数外，用户还可以使用另外两个函数操作消息队列。msgsnd()函数将消息放入队列中，其原型如下：

int msgsnd(int msqid， const void *msgp， size_t msgsz， int msgflg)；

其中参数的含义如下所示：

msqid：消息队列的资源 ID 号。

msgp：消息缓冲区的首地址。消息缓冲区由两部分组成，首先是消息的类型，然后才是数据部分。

msgsz：消息缓冲区的长度。

msgflg：可以是 0，也可以是 IPC_NOWAIT。

另一个函数 msgrcv()从某个消息队列中读一个消息并将其移出消息队列。其原型如下：

int msgrcv(int msqid，void*msgp，int msgsz，long msgtyp，int msgflg)；

其中参数的含义是：

msgp：接收消息的缓冲区首地址。

msgsz：接收缓冲区的大小，这是函数能返回的最大数据量。

msgtyp：指定接收消息的类型。分为三种情况：

• 值为 0，返回队列中的第一个消息。

• 值大于 0，返回类型为 msgtyp 的第一个消息。

• 值小于 0，则返回类型值小于或等于 msgtyp 的绝对值的消息中类型值最小的第一个消息。

msgflg：可以是 IPC_NOWAIT，还可指定为 MSG_NOERROR。MSG_NOERROR 允许消息长度大于接收缓冲区长度时截断消息返回。

在 Linux 内核中，消息队列用 msg_queue 结构表示，其定义如下：

```
struct msg_queue
{
struct kern_ipc_perm q_Perm;
time_t       q_stime;  /*最近一次 msgSnd 时间*/
time_t      q_rtime;  /*最近一次 msgrcv 时间*/
time_t       q_ctime;        /*最近的改变时间*/
unsigned long       q_cbytes;        /*队列中的字节数*/       .
unsigned long       q_qnum;        /*队列中的消息数目*/
unsigned long       q_qbytes;        /*队列中允许的最大字节数*/
pid_t       q_lspid;        /*最近一次 msgsnd()发送进程的 pid*/
pid_t       q_lrpid;        /*最近一次 msgrcv()接收进程的 pid*/
struct    list_head       q_messages;        /*消息队列*/
struct list_head       q_receivers;     /*待接收消息的睡眠进程队列*/
struct list_head       q_senders;        /*待发送消息的睡眠进程队列*/
)  ;
```

若 IPC_NOWAIT 未被设置，则当消息队列的容量已满时，发送消息的进程会进入睡眠状态，并添加到相应的 q_senders 队列，而当消息队列中无合适的消息时，接收进程会进入睡眠状态，并添加到相应的 q_receivers 队列。消息队列中的每个消息都链入 q_message 队列中，每个消息用一个 msg_msg 结构描述，该结构定义如下：

```
struct msg_msg
{
```

```
struct list_head    m_list;      /*消息队列链表*/
long        m_type;      /*消息的类型*/
int         m_ts;        /*消息的长度*/
struct msg_msgseg    *next;      /*链接属于这个消息的下一个消息片*/
}   ;
struct    msg_msgseg
{
struct    msg_msgseg       *    next;
}   ;
```

可以看出，msg_msg 结构实际只是一个消息头部，并不包含消息的数据部分。Linux 对数据部分的处理如下：数据部分的空间紧接 msg_msg 结构，从而保证了可以找到属于该消息的数据，但是当数据部分的空间与 msg_msg 结构所占空间大于一个页面时，则将其以页面为单位分片。第一个页面存储 msg_msg 结构与首部分数据，随后的再分配空间则存储 struct msg_msgseg 结构与剩余的数据。如果这两者所占空间之和仍大于一个页面，则继续分配下去。msg_msgseg 结构用以把消息片链接在一起。图 3-2 给出了各个数据结构之间的互相联系。

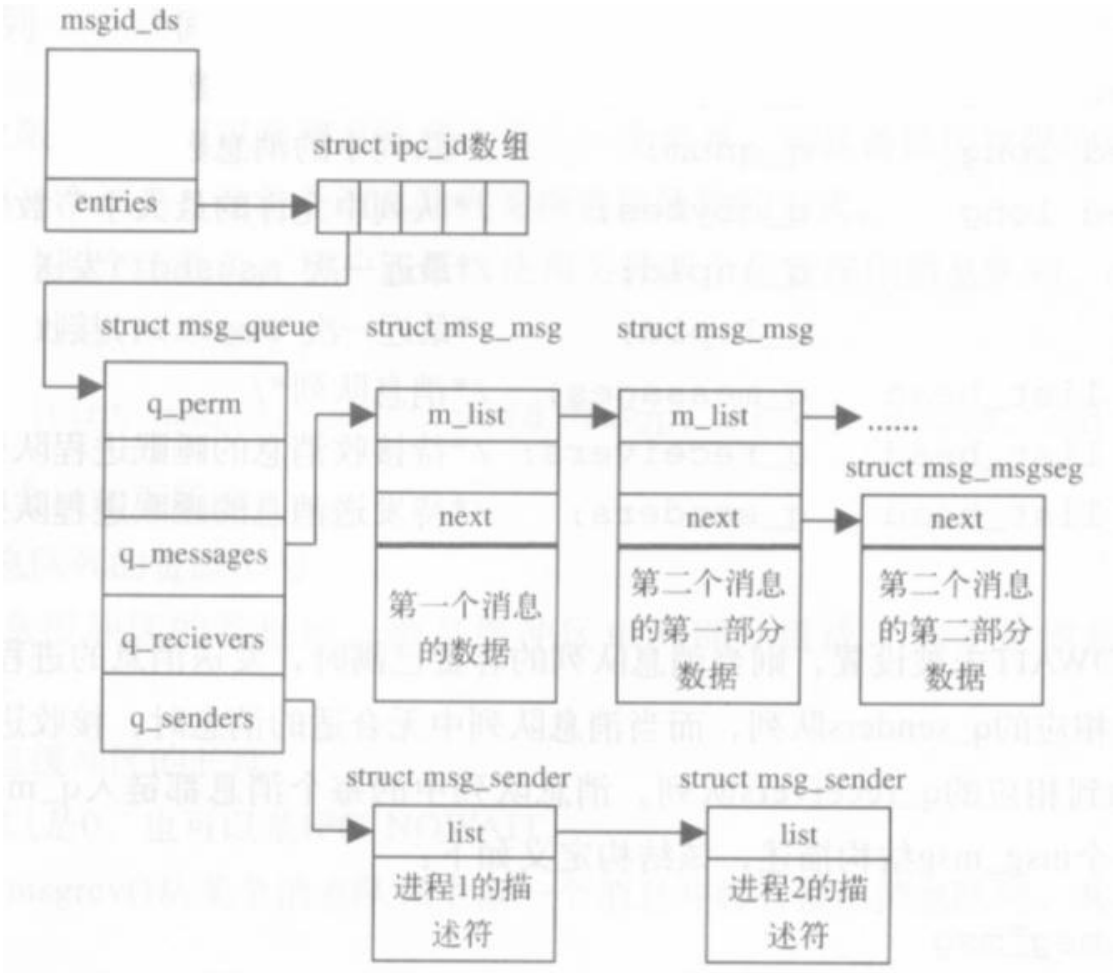


图 3-2 消息队列各数据结构之间的关系

3.4.共享内存

共享内存是多个进程共享的一块内存区域。不同的进程可把共享内存映射到自己的一块地址空间，不同的进程进行映射的地址空间不一定相同。映射后，进程对该段地址空间的写操作就是对共享内存的写操作。同样，映射了该共享内存的其他进程，就可以看见该变化，从而实现进程通信。但是共享内存没有提供进程同步与互斥的机制，所以往往需要和信号量配合使用。

与其他进程通信方式相比，共享内存在进行数据交换方面效率是比较高的。假设进程 A 给进程 B 发送数据，我们来比较一下消息队列与共享内存的传送效率。如果采用消息队列， A 进程调用 msgsnd()需要从用户态切换到核心态，进行数据传递(这里的开销往往是比较大的)然后再返回用户态，B 进程调用 msgrcv()需要从用户态切换到核心态然后再返回用户态，还有不可避免的数据传递的开销，总共需要 4 次切换。如果采用共享内存方式，一旦映射已经建立，A 往共享内存写数据，B 立即能看见，根本无须运行状态的切换。

shmget()函数有一个参数指定共享内存区域的大小，该函数建立的共享内存区在内核中用 shmid_kernel 结构表示，其定义如下：

```
struct    shmid_kernel
    {
    struct kern_ipc_perm       shm_perm;
    struct file*       shm_file;
    int       id;
    mnsigned long       shm_nattch;        /*已建立映射的数目*/
    unsigned long       shm_ segsz;        /*共享内存区的大小*/
    time_t       Shm atim;
    time_t       Shm_dtim;
    time_t       Shm_ctim;
    pid_t       shm_cprid;
    pid_t       shm_lprid;
    }   ;
```

shmget()创建的共享内存区域并没有立即分配物理内存，而是创建一个文件对象 shm_file 来描述该区域，而该文件属于 shin 文件系统。shm 文件系统是一个内存文件系统，它不依赖于磁盘文件的内容，记载该文件系统的相关信息随着关机彻底消失。在后面的介绍中我们可以看到 shm_file 实际上是由一个个页面组成的。

进程调用 shmat()函数建立进程地址空间与共享内存区的映射。选取进程地址空间的哪一段区间进行映射，可由用户指定也可委托内核进行选择。shmat()函数找到区间后，进程分配一个 vm_area_struct 结构描述该区间，vm_area_struct 结构的各项被初始化，其中 file 成员被初始化为 shm_file，而 vm_ops 成员被初始化为 shm_vm_ops。变量 shm_vm_ops 定义如下：

```
    static    struct    vm_operations_struct    shm_vm_ops =
    {
    open：        shm_open，
    close：        shm_close，
    nopage：      shmem_nopage，
    }   ;
```

需注意的是，共享内存区在第一次调用 shmat()后仍然没有分配物理内存，Linux 在此采取的仍然是“懒惰”策略。当进程第一次访问该映射共享内存区内的地址时，将触发页面异常， 最终将调用 shmem_nopage()函数。该函数处理的过程大致如下：

(1)先根据文件和文件位置查找页面缓存(page cache)，因为别的进程可能已经为映射的共享内存区页面申请了一个物理页帧。如果找到，修改本进程页表即可；否则继续下一步。

(2)检查被映射的共享内存区页面是否被访问过，但已被换出到交换分区。如果是，则调入该页面，修改进程页表；否则继续下一步。

(3)若被映射的共享内存区页面从未被访问过，则向内存子系统申请一个物理页帧，修改进程页表。进程可以调用 shmdt()函数解除地址空间与共享内存区的映射关系，其处理较为简单，主要是修改页表及释放 vm_area_struct 结构(该结构用于描述虚空间的区)。

图 3-3 显示出各个主要数据结构之间的联系。

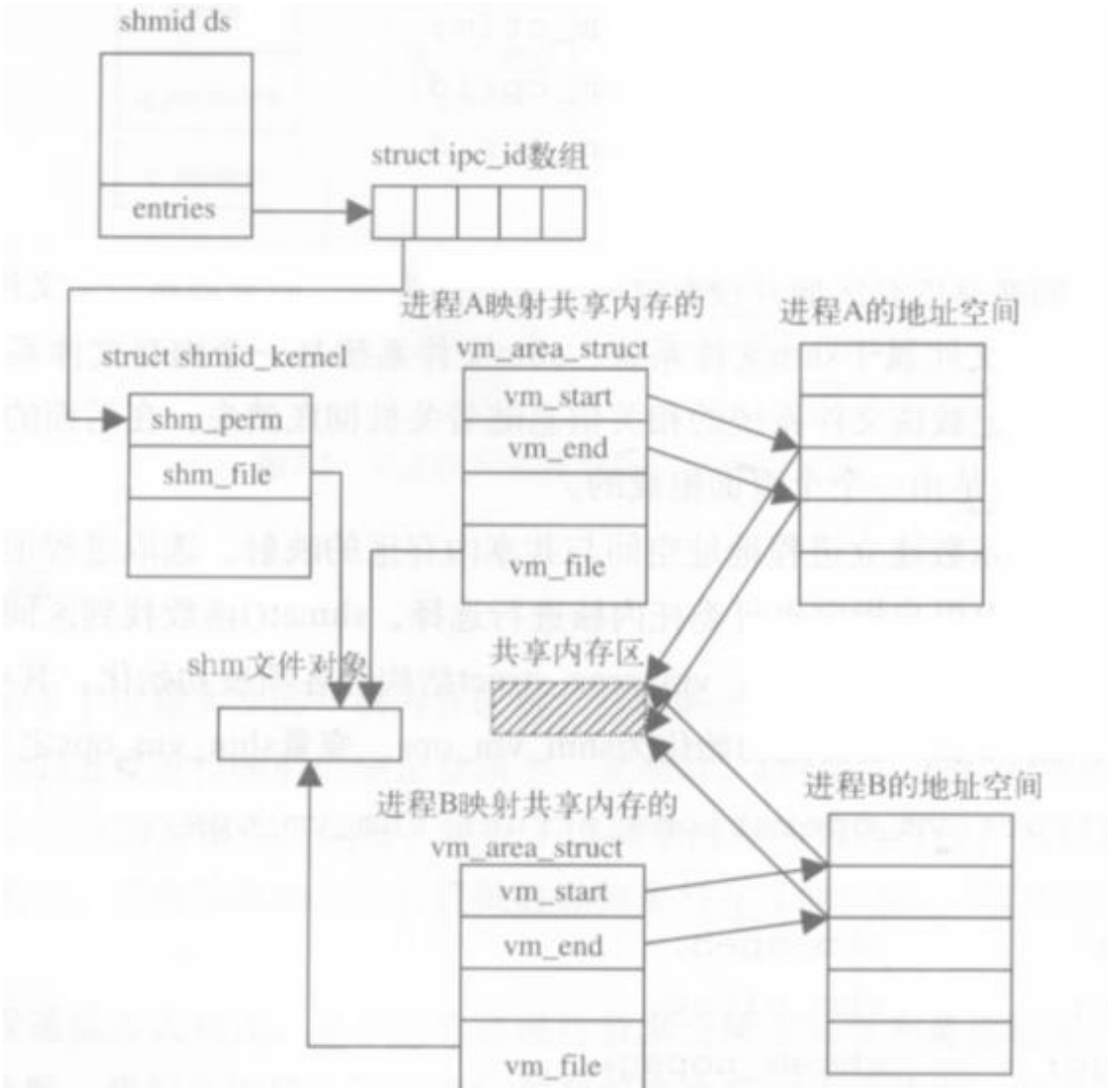


图 3-3 共享内存各数据结构之间的关系

第四章.文件系统

Linux 的最重要特征之一就是支持多种文件系统，这样它就更加灵活并可以和许多其他种操作系统共存。目前我们比较常见的 EXT2、EXT3、NTFS、VFAT 以及 ISO9660 等文件系统，都能够被 Linux 所支持。

为了把每个实际文件系统从操作系统和系统服务中分离中来，它们之间通过一个接口层—虚拟文件系统（VFS，Virtual File System）来通信。VFS 使得 Linux 可以支持多个不同的文件系统，每个具体的文件系统都实现了 VFS 的通用接口。由于软件将 Linux 文件系统的所有的细节进行了转换，因此 Linux 内核的其他部分及系统中运行的程序将看到统一的文件系统。Linux 的虚拟文件系统允许用户同时能透明地安装许多不同的文件系统。

VFS 这样一个纯软件中间层是一种优秀的设计思想，加入中间层无疑会使文件子系统的可扩展性、可维护性变得更好。本章先介绍 VFS 的运作原理，然后再介绍一个物理文件系统—EXT2，最后将介绍几个操作系统调用的实现。

4.1.Linux 文件系统概述

文件是数据的集合，文件系统不仅包含文件中的数据而且还有文件系统的结构。文件系统负责在外存上管理文件，并把对文件的存取、共享和保护等手段提供给操作系统和用户。它不仅方便了用户使用，保证了文件的安全性，还可以大大地提高系统资源的利用率。Linux 最早的文件系统是 Minix，它所受限制很大而且性能低下。其文件名最长不能超过 14 个字符且最大的文件不超过 64MB。第一个专门为 Linux 设计的文件系统称为扩展文件系统(ExtendedFileSystem)或 EXT。它出现于 1992 年 4 月，虽然能够解决一些问题，但性能依旧不好。1993 年扩展文件系统第二版或 EXT2 被设计出来并添加到 Linux 中，它是系统的标准配置。

Linux 的文件系统的功能非常强大，能支持 Minix、EXT、EXT2、UMSDOS、MS-DOS、VFAT 等多达 15 种文件系统，并且能够实现这些文件系统之间的互访。Linux 的文件系统和 Windows 的不一样，没有驱动器的概念，而是表示成单一的树状结构。如果想增加一个文件系统，必须通过装载(mount)命令将其以一个目录的形式挂接到文件系统层次树中。该目录称为安装点或者安装目录。若要删除某个文件系统，使用卸载(unmount)命令来实现。当磁盘初始化时(使用 fdisk)，磁盘中将添加一个描述物理磁盘逻辑构成的分区结构。每个分区可以拥有一个独立文件系统，如 EXT2。文件系统将文件组织成包含目录，软连接等存在于物理块设备中的逻辑层次结构。包含文件系统的设备叫块设备。Linux 文件系统认为这些块设备是简单的，它并不关心或理解底层的物理磁盘结构。将磁盘的物理结构映射为线性块集合的工作由块设备驱动宋完成，由它将对某个特定块的请求映射到正确的设备上去，此数据块所在硬盘的对应磁道、扇区及柱面数都被保存起来。不管哪个设备持有这个块，文件系统都必须使用相同的方式来寻找并操纵此块。Linux 文件系统不管(至少对系统用户来说)系统中有哪些不同的控制器，控制着哪些不同的物理介质，且这些物理介质上有几个不同的文件系统。每个实际文件系统和操作系统之间通过虚拟文件系统 VFS 来通信，Linux 的文件系统结构如图 4-1 所示。

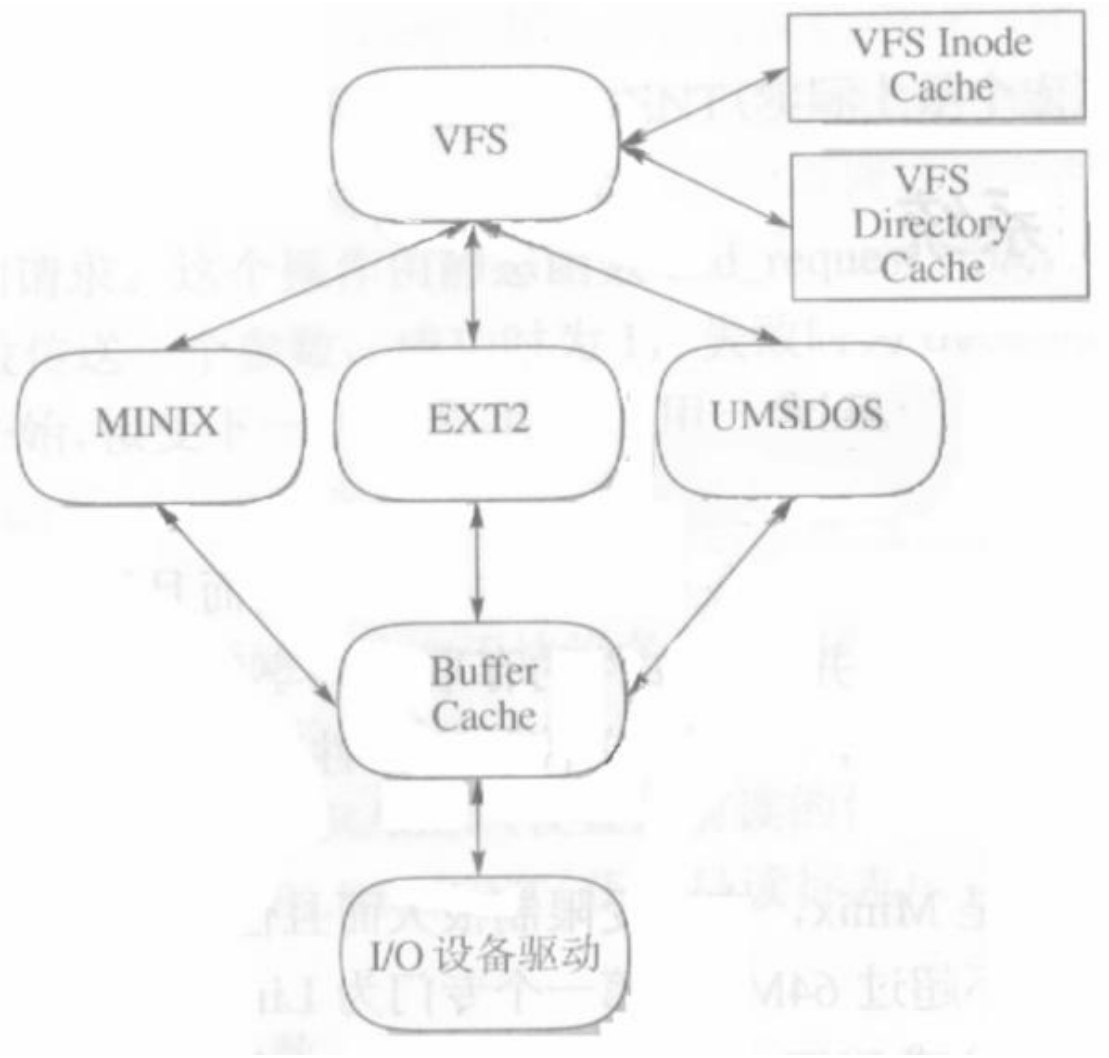


图 4-1　Linux 文件系统的结构

在各种物理文件系统与 I/O 设备之间，通过缓冲来实现快速高效的文件访问服务，并独立于底层介质和设备驱动。当 Linux 安装一个文件系统并使用时，VFS 为其缓存相关信息。此缓冲中数据在创建、写入和删除文件与目录时如果被修改，则必须谨慎地更新文件系统中对应内容。这些缓存中最重要的是 BufferCache，它被集成到独立文件系统访问底层块设备的例程中。当进行块存取时，数据块首先被放入 BufferCache，并根据其状态保存在各个队列中。此 BufferCache 不仅缓存数据，而且帮助管理块设备驱动中的异步接口。

4.2.虚拟文件系统 VFS

Linux 系统的最大的特点之一就是能支持多种不同的文件系统，每一种文件系统都有自己的组织结构和文件操作函数，相互之间差别很大。Linux 对上述文件系统的支持是通过虚拟文件系统 VFS 的引入而实现的。VFS 是物理文件系统与服务例程之间的一个接口层，它对 Linux 的每个文件系统的所有细节进行抽象，使得不同的文件系统在 Linux 核心以及系统中运行的进程看来都是相同的。

1. VFS 的功能

(1)记录可用的文件系统的类型；

(2)将设备同对应的文件系统联系起来；

(3)处理一些面向文件的通用操作；

(4)涉及针对文件系统的操作时，VFS 把它们映射到与控制文件、目录，以及 inode 相关的物理文件系统。

2．VFS 的数据结构

VFS 的结构与 UNIX 文件系统的模型一致，使用了超块和 inode 来描叙文件系统。在超块中描述了系统中已安装文件系统的相关信息，VFS inode 说明了系统中的文件和目录以及 VFS 中的内容和拓扑结构。

1)VFS 超块

每个已安装的文件系统由一个 VFS 超块表示，它包含如下信息：

- Device　表示文件系统所在块设备的设备标识符。
- Inode pointers　这个 mounted inode 指针指向文件系统中第一个 inode。而 covered inode 指针指向此文件系统安装目录的 inode。根文件系统的 VFS 超块不包含 covered 指针。
- Blocksize　以字节记数的文件系统块大小，如 1024 字节。
- Superblock operations　指向此文件系统一组超块操纵例程的指针。这些例程被 VFS 用来读写 inode 和超块。
- File System type　这是一个指向已安装文件系统的 file_system_type 结构的指针。
- File System specific　指向文件系统所需信息的指针。

2)VFS inode

VFS 中的每个文件、目录等都用且只用一个 VFSinode 表示。每个 VFSinode 中的信息通过文件系统相关例程从底层文件系统中得到。VFSinode 仅存在于核心内存，并且只要对系统有用，它们就会被保存在 VFSinodecache 中。每个 VFSinode 主要包含下列域：

- Device　包含此文件或此 VFSinode 代表的任何东西的设备的设备标志符。
- Inode number　文件系统中惟一的 inode 号。在虚拟文件系统中 device 和 inode 号的组合是惟一的。
- Mode　表示此 VFSinode 的存取权限。
- Userids　所有者的标志符。
- Times　VFSinode 创建、修改和写入时间。
- Blocksize　以字节计算的文件块大小，如 1024 字节。
- Inode operations　指向一组例程地址的指针。这些例程和文件系统相关，且对此 inode 执行操作，如截断此 inode 表示的文件。
- Count　使用此 VFS inode 的系统部件数。一个 count 为 0 的 inode 可以被自由丢弃或重新使用。
- Lock　用来对某个 VFS inode 加锁，如用于读取文件系统时。
- Dirty　表示这个 VFS inode 是否已经被写过，如果是则底层文件系统需要更新。

3．文件系统功能的实现

1)VFS 的初始化

系统启动和操作系统初始化时，物理文件系统将其自身注册到 VFS 中。物理文件系统除了可以构造到核心中之外，也可以设计成可加载模块的形式，通过 mount 命令在 VFS 中加载一个新的文件系统。当 mount 一个基于块设备且包含根目录的文件系统时，VFS 必须读取其超块。每个文件系统类型的超块读取例程必须了解文件系统的拓扑结构，并将这些信息映射到 VFS 的超块结构中。VFS 在系统中保存着一组已安装文件系统的链表及其 VFS 超块。每个 VFS 超块包含一些信息以及一个执行特定功能的函数指针。

2)文件操作的实现

当某个进程发布了一个面向文件或目录的系统调用时，首先使用系统调用遍历系统的VFS inode。为了在虚拟文件系统中找到某个文件的 VFS inode，VFS 必须依次解析此文件名字中的中间目录直到找到此 VFS inode。然后内核将调用 VFS 中相应的函数，这个函数处理一些与物理结构无关的操作，并且把它重定向为真实文件系统中相应的函数调用，而这些函数调用则用来处理那些与物理结构相关的操作。

VFS 界面由一组标准的、抽象的文件操作构成，以系统调用的形式提供给用户程序如read()、write()、lseek()等。不同的文件系统通过不同的程序来实现各种功能，但是具体的文件系统与 VFS 之间的界面是有明确定义的，这个界面的主体就是 fs_operations 数据结构。每种文件系统都有自己的 fs_operations 数据结构，结构中的成分几乎全是函数指针，如 read 就指向具体文件系统用来实现读文件操作的入口函数。在访问文件时，每个进程通过 open()与具体的文件建立连接，如图 4-2 所示。

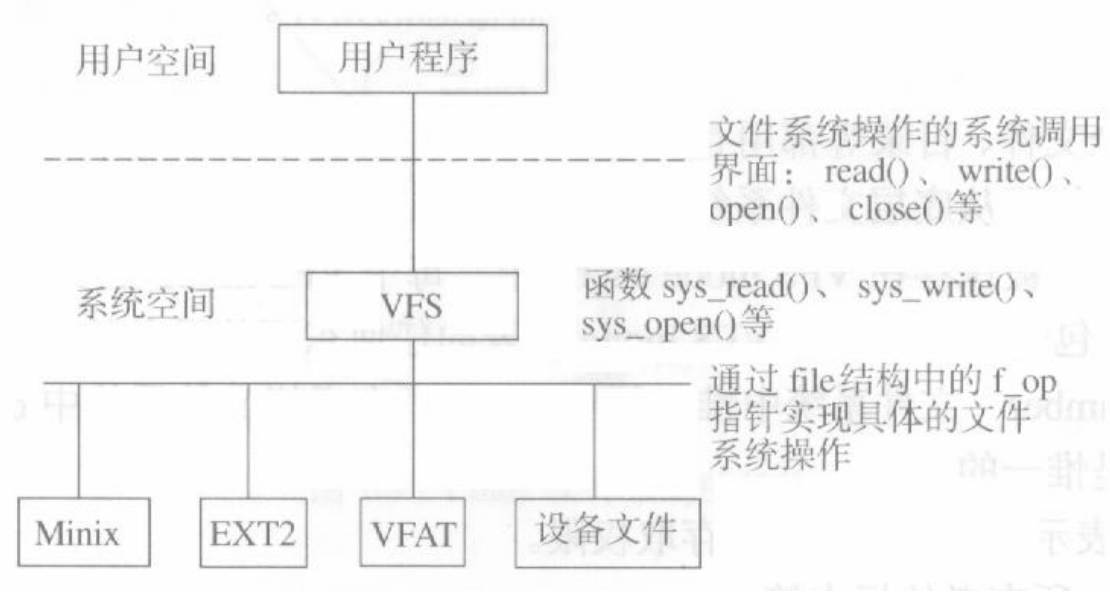


图 4-2 VFS 与具体文件系统之间关系的示意图

3)VFS 的缓冲机制

访问任何一个文件，首先要读取它的索引节点，因此索引节点的访问频率极高。为了加速对所有已安装文件系统的访问，VFS 采用 inode cache 来实现，这样当虚拟文件系统访问一个 inode 时，系统将先在 VFS inode cache 中查找。如果某个 inode 不在 inode cache 中则必须调用一个文件系统相关例程来读取此 inode。这个 inode 的读操作将把它放到 inode cache 中以备下一次访问。不经常使用的 VFS inode 将从 cache 中移出。VFS inode cache 以散列表形式实现，散列值可通过包含此文件系统的底层物理设备标识符和 inode 号计算出来。其入口指向具有相同散列值的 VFS inode 链表。

VFS 还支持一种目录 cache 以便对经常使用的目录对应的 inode 进行快速查找。目录cache 不存储目录本身的 inode，仅仅保存全目录名和其 inode 号之间的映射关系。目录 cache 也由散列表组成，每个入口指向具有相同散列值的目录 cache 入口链表。散列函数使用包含此文件系统的设备号以及目录名称来计算在此散列表中的偏移值或者索引值，这样能很快找到被缓存的目录。为了保证 cache 的有效性和及时更新，VFS 保存着一个最近最少使用 (LRU)的目录 cache 入口链表。

4.3.EXT2 文件系统

对文件系统而言，文件仅是一系列可读写的数据块。文件系统并不需要了解数据块应该放置到磁盘上什么位置，这些都是设备驱动的任务。无论何时，只要文件系统需要从包含它的块设备中读取信息或数据，它将请求底层的设备驱动读取一个基本块大小整数倍的数据块。EXT2(第二代扩展文件系统)由 ReyCard 设计，它是 Linux 界中设计最成功的文件系统，它很好地继承了 UNIX 文件系统的主要特色，如普通文件的三级索引结构，目录文件的树状结构和把设备作为特别文件等。

EXT2 文件系统的块大小是在创建(使用 mke2fs)时设置的，每个文件的大小也是以块为单位进行分配的，因此是块大小的整数倍。磁盘上除了包含数据块之外，还有些块用于存储描述文件系统结构的信息。EXT2 通过一个 inode 结构来描叙文件系统中文件，并确定此文件系统的拓扑结构。inode 结构描叙文件中数据占据哪个块以及文件的存取权限、文件修改时间及文件类型。EXT2 文件系统中的每个文件用一个 inode 来表示，且每个 inode 有惟一的编号。文件系统中所有的 inode 都被保存在 inode 表中。EXT2 目录仅是一个包含指向其目录入口指针的特殊文件(也用 inode 表示)。

1．EXT2 文件系统的布局

Linux 文件系统是一个逻辑的自包含的实体，它含有 inode，目录和数据块。Linux 将整个磁盘划分成若干分区，每个分区被当做独立的设备对待；一般需要一个主分区 native 和一个交换分区 swap。主分区用于存放文件系统，交换分区用于虚拟内存。主分区内的空间又分成若干个组。每个组内都包含有一个超级块的拷贝，以及 inode 和数据块等信息。

2．描述 EXT2 文件系统的数据结构

1)EXT2 超级块(Super-block)

超级块主要用来描述目录和文件在磁盘上的静态分布，包括大小和结构。如给定块大小、inode 总数、每组内 inode 节点数、空闲块和 inode 数等。超级块存放在 include/linux/ext2_fs.h 中的定义 struct ext2_super_block 结构内。在 Linux 启动时，根设备中的超级块被读入内存中，在某个组的超级块或者 inode 受损时，可以用来恢复系统。超级块对于文件系统的维护至关重要。一般，只有块组 0 的超级块才读入内存，其他块组的超级块仅仅作为备份。

超级块包含如下信息：文件系统的安全信息、兼容性；是否是一个真正的 EXT2 文件系统超块；是否应对此文件系统进行全面检查；超块的拷贝；以字节记数的文件系统块大小；每个组中块数目；文件系统中空闲块数；文件系统中空闲 inode 数；文件系统中第一个 inode 号，即指向“/”目录的目录入口的 inode。

2)EXT2 的组描述符

每个数据块组都拥有一个描述它的数据结构，包括本组的块分配位图的所在的块号、inode 分配位图的块号、inode 表的起始块号等。组描述符放置在一起形成了组描述符表。这是关于文件系统的备份信息以防文件系统崩溃。

3)EXT2 的位图

Linux 文件系统用位图来管理磁盘块和 inode，位图分为块位图(block bitmap)和 inode 位图。块位图占用一个磁盘块，当某位为“1”时，表示磁盘块空闲，为“0”时表示磁盘块被占用。inode 位图也占用一个磁盘块，当它为“0”时，表示组内某个对应的 inode 空闲，为“1”时表示已被占用。位图使系统能够快速地分配 inode 和数据块，保证同一文件的数据块能在磁盘上连续存放，从而大大地提高了系统的实时性能。

在创建文件时，文件系统必须在块位图中查找第一个空闲 inode，把它分配给这个新

创建的文件。在该空闲 inode 分配使用后，就需要修改指针，使它指向下一个空闲 inode。同样地，inode 被释放后，则需要修改指向第一个空闲 inode 的指针。

4)EXT2 的 inode 表

inode 是 EXT2 的基本组成部分。文件系统的每个文件或目录都由一个 inode 描述。每个 inode 对应于一个惟一的 inode 号。inode 包含了文件内容、在磁盘上的位置、文件的存取权限、修改时间以及类型等。同时，还有一个位图被系统用来跟踪已分配和未分配的 inode，并与之一一对应。属于同一块组的 inode 保存在同一个 inode 表中，inode 表占用若干个磁盘块，它几乎与标准 UNIX 的 inode 表相同。

下面是 Linux 关于 inode 数据结构的描述，它位于 include/linux/ext2_rs.h 中。

- Mode　它包 inode 描述的内容以及用户使用权限。EXT2 中的 inode 可以表示一个普通文件、目录文件、符号连接、块设备文件、字符设备文件和管道文件等。
- Owner lnformation　表示此文件或目录所有者的用户和组标识符。文件系统根据它可以进行正确的存取。
- Size　以字节计算的文件大小。
- Timestamps　inode 创建及最后一次被修改的时间。
- Datablocks　指向此 inode 描述的包含数据的块指针。前 12 个指针指向包含由 inode 描述的物理块，最后 3 个指针包含多级间接指针。例如两级间接指针指向一块指针，而这些指针又指向一些数据块。这意味着访问文件尺寸小于或等于 12 个数据块的文件将比访问大文件快得多。

EXT2 inode 还可以描述特殊设备文件。虽然它们不是真正的文件，但可以通过它们访问设备。所有那些位于/dev 中的设备文件可用来存取 Linux 设备。

5) EXT2 目录

在 EXT2 文件系统中，目录是用来创建和包含文件系统中文件存取路径的特殊文件。EXT2 采用动态方式管理它的目录，用一个单项链表示它的目录项，每个目录项的数据结构含以下几项：

- inode　对应每个目录入口的 inode。它被用来索引储存在数据块组的 inode 表中的 inode 数组。
- namelength　以字节计数的目录入口长度。
- name　目录入口的名称。

每个目录的前两个入口总是“.”和“..”。它们分别表示当前目录和父目录。当要删除一个目录时，要将目录项中的 inode 节点置为“0”，并把目录项从链表中删除，目录项所占 Linux 文件名的格式与 UNIX 类似，是一系列以“/”隔开的目录名并以文件名结尾。为了寻找 EXT2 文件系统中表示此文件的 inode，系统必须将文件名从目录名中分离出来。查找 EXT2 文件的一般步骤流程是：

(1)读取超级块信息，了解磁盘分区的管理信息，特别是块长度和 InodeTable 起始块号。

(2)找到根‘/’　目录信息。EXT2 文件系统中，根目录的索引节点号是固定的，即 EXT2_ROOT_INO=2，读取 2 号索引节点表信息，即可找到根目录所在的块号。在/include/Linux/ext2_fs.h 中定义 EXT2_ROOIINO，即

#define EXT2_ROOT_INO 2/*根节点*/

(3)读取根目录或普通目录所在块的信息，根据 extldir_entry_2 结构，找到子目录或文件的 inode 号。

(4)读取字目录或文件的 inode 信息，找到文件数据所在的磁盘块号。

(5)读取子目录或文件数据块的信息。如果是文件，则查找结束。如果是目录，转到(3)。EXT2 文件数据块地址存放在指针数组 i_block[l5]中。指针数组 lblock[0]～i_block[11]中直

接包含文件块 0—11 的物理块地址，而 i_block[13]中包含一次间接寻址地址，i block[14]中包含二次间接寻址地址，i_block[l5]中包含三次间接寻址地址。

3．数据块的分配

为了提高文件系统访问的效率，尽量避免碎片问题，EXT2 文件系统试图通过分配一个和当前文件数据块在物理位置上邻接或者至少位于同一个数据块组中的新块来解决这个问题。只有在这种分配策略失败时才在其他数据块组中分配空间。

当进程准备写某文件时，Linux 文件系统首先检查数据是否已经超出了文件最后一个被分配的块空间，如果是，则必须为此文件分配一个新数据块。进程将一直等待到此分配完成，然后将其余数据写入此文件。数据块分配过程如下：

(1)为了保持数据的一致性，EXT2 块分配程序首先对此文件系统的 EXT2 超块加锁，对超块的访问遵循先来先服务原则。

(2)若采用预分配数据块策略，则从预先分配数据块中取得一个。预先分配块实际上并不存在，它们仅仅包含在已分配块的位图中。

(3)若没有使用预分配策略，则 EXT2 文件系统必须分配一个新数据块。首先检查此文件最后一个块后的数据块是否空闲。如果此块已被使用，则它会在同一个数据块组中选择一个。

(4)如果找不到这样的数据块，进程将在其他数据块组中搜寻，直到找到一空闲块。并更新与预分配策略中的相关数据。

(5)找到空闲块后，块分配程序将更新数据块组中的位图并在 BufferCache 中为它分配一个数据缓存。缓存中的数据被置 0，且缓存被标记成 dirty 以显示其内容还没有写入物理磁盘。最后超块也被标记为 dirty 以表示它已被更新并解锁了。

4.4.Linux 的几个重要文件系统

1．/proc 文件系统

/proc 文件系统真正显示了 Linux 虚拟文件系统的能力。事实上它并不存在，不管/proc 目录还是其子目录和文件都不真正存在。但是我们是如何能够执行 cat /proc/devices 命令的呢?/proc 文件系统像一个真正的文件系统一样将向虚拟文件系统注册，然而当有对/proc 中的文件和目录的请求发生时，VFS 系统将从核心的数据中临时构造这些文件和目录。例如，核心的/proc/devices 文件是从描述其设备的内核数据结构中产生出来。/proc 文件系统提供给用户一个核心内部工作的可读窗口。Linux 核心模块都在/prog 文件系统中创建入口。

2．设备特殊文件

和所有 UNIX 版本一样，Linux 将硬件设备看成是特殊的文件。如/dev/null 表示一个空设备。设备文件不使用文件系统中的任何数据空间，它仅仅是设备驱动的访问入口点。EXT2 文件系统和 Linux VFS 都将设备文件实现成特殊的 inode 类型。有两种类型的设备文件：字符与块设备特殊文件。在核心内部设备驱动实现了类似文件的操作过程，我们可以对它执行打开、关闭等操作。字符设备允许以字符模式进行 I/O 操作，而块设备的 I/O 操作需要通过 BufferCache。当对一个设备文件发出的 I/O 请求，则会被传递到相应的设备驱动，这种设备文件往往并不是一个真正的设备驱动而仅仅是一个伪设备驱动，如 SCSI 设备驱动层。设备文件通过表示设备类型的主类型标志符和表示单元或主类型实例的从类型来引用。例如在系统中第一个 IDE 控制器上的 IDE 硬盘的主设备号为 3，而其第一个分区的从标识符为 1。所以执行 1s-1/dev/hdal 将有如下结果：

```
$ brw-rw---- 1root disk    3,1    Feb 21 9:09 /dev/hdal
```

在核心内部每个设备由惟一的 kdev t 结构来表示，其长度为两字节，首字节包含从设备号，而尾字节包含主设备号。上例中的核心 IDE 设备为 0x0301，表示块设备的 EXT2 inode，当 VFS 读取它时，表示它的 VFS inode 结构的 i_rdev 域被设置成相应的设备标识符。

第五章.设备管理

作为操作系统，最重要的任务之一，就是将系统硬件设备的细节从用户视线中隐藏起来。例如在虚拟文件系统中，对各种类型已安装的文件系统提供了统一的视图而屏蔽了具体底层细节。Linux 沿用了 Unix 处理设备的做法，把设备映射成为虚拟文件系统上的文件，用户可以像处理常规文件一样来操作设备。本章首先介绍设备文件的概念及相关的数据结构，最后通过 open()与 read()的实现，来讨论块设备管理是如何与 VFS 框架结合的。

5.1.设备文件的概念

在传统的 Unix 系统中，都把设备当成文件来处理，因而可以用 read()或者 write()等针对文件的系统调用对设备进行操作。设备文件一般在/dev 目录下，如/dev/fd0 表示软盘，/dev/hadl 表示第一个 IDE 硬盘的第一个分区。

Linux 下的设备大体分为三类：

(1)块设备：每次 I/O 操作以固定大小的数据块为单位，且可随机存取。

(2)字符设备：每次 I/O 操作存取的数据量不固定，并且只能顺序存取。

(3)网卡：网卡是一种特殊处理的设备，它没有对应的设备文件。

设备文件除文件名之外，还有三个主要的属性：

(1)类型：是字符设备还是块设备。

(2)主设备号：主设备号相同的设备被同一设备驱动程序处理。

(3)次设备号：用来指明具体的设备。

例如/dev/hadl、/dev/had2 等，都是块设备文件，主设备号均为 3(表示 IDE 硬盘)，次设 备号则分别为 1 和 2(表示 IDE 硬盘上的第 1 号分区和第 2 号分区)。设备文件的生成是由 mknod()系统调用完成的，它的参数是上面提到的三个属性。Linux 安装完成之后已经在/dev 目录下生成了绝大多数可能要用到的设备文件，尽管很多真正的设备尚未安装。

5.2.相关数据结构

1．设备的注册和注销

字符设备管理的主要数据结构如下：

```
struct    device_struct
      {
      const char       *name;
      struct file_operationo    *fops;
      }   ;
      static    struct    device_struct    chrdevs[MAX_CHRDEV];
```

全局数组 chrdevs[]记录了所有字符设备的名称 name，及其对应的设备操作函数接口 fops。

数组的下标则对应于设备的主设备号。设备在系统中的注册是通过函数 register_chrdev()实现的，其原型如下：

```
int    register_chrdev(unsigned    int    major,    const    char    *    name,
```

```
struct file_operations    *fops);
```

不同的字符设备要提供不同的 file_operations 实现。当设备不再使用时，可以通过 unregister_chrdev()函数注销。

块设备的管理比字符设备的管理要复杂一些。其主要数据结构有 blkdevsl)和 blk_dev[]：

```
static    struct
{
const char        *name;
struct block device_operations*bdops;  /*特定于设备的操作集*/
)    blkdevs[MAX_BLKDEV];
struct    blk_dev_struct
{
request_queue_t        requeSt_queue;          /*请求队列*/
queue_proc      *queue;
void        *data;
)    ;
struct blk_dev_struct blk_dev[MAX_BLKDEV];
```

数组的下标依然对应设备的主设备号，其中 blkdevsl)通过 register_blkdev()函数初始化，而 blk_dev[]通过 blk_dev_init()函数初始化。blkdevsl)记录设备文件名及相应的操作集合，blldev[]记录各个设备的请求队列。

2.缓冲区管理

块设备的操作是以块为基本单位(物理磁盘的基本单位是扇区)进行的，一般情况下，块的大小不会超过页面的大小。每读入一个块时，并不立即把块的数据送到应用程序的缓冲区，而是先在内核申请一个同等大小的内核缓冲区，将块读入，然后再写入程序的缓冲区。这样，下一次访问该块时就不必进行磁盘操作，从而提高了性能。对块的写操作同样要经过内核中的缓冲区，每个缓冲区由 buffer_head 结构描述，该结构定义如下：

```
struct buffer_head
{
struct buffer_head        *b_next;          /*用来链接 hash 值相同的 buffer_ head*/
unsigned long        b_blocknr;      /*块号*/
unsigned short        b_size;          /*块的大小*/
kdev_t        b_dev;
kdev_t        b_rdev;          /*设备号*/
struct buffer_head*b_this_page;  /*同属一个页面的 buffer 链表*/
struct buffer_head*b_reqnext;      /*同一个操作请求的 bu“er_head 链表*/
struct buffer_head**b_pprev;          /*用来链接 hash 值相同的 bu“er_head*/
char        *b_data;          /*buffer 所在的位置*/
struct page        *b page;          /*buffer 所属的页面*/
wait_queue_head_t    b_wait;          /*进程等待队列*/
struct inode*        b_inode;          /*该 buffer 所属的 inode 结构*/
}    ;
```

每个缓冲区由设备号和块号唯一确定，并用这两者作为 hash 关键字快速找到所在的位置。通常块的大小为 1KB，而物理页帧的大小为 4KB，所以一个物理页帧可以容纳 4 个缓冲区，图 5-1 反映了这种关系。

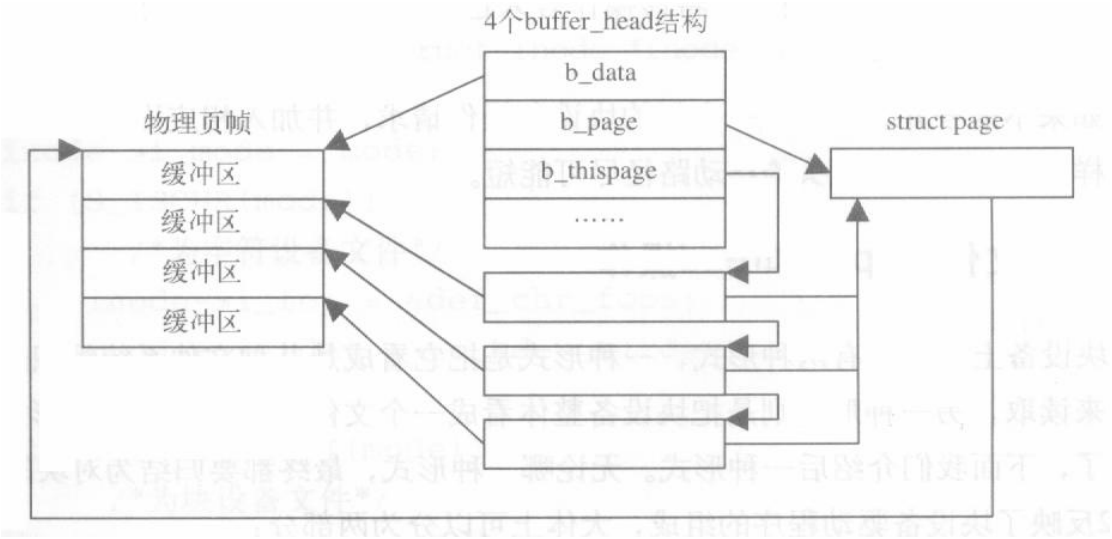


图 5-1 同属一个页面的 buffer_head 之间的关系

3.设备请求队列

每当需要对一个块操作时，该操作并不立即进行，因为磁盘的定位操作非常耗时。Linux 的做法是，通常每个块设备都维护一个请求队列，该队列的每一个成员称为操作请求。

操作请求结构的定义如下：

```
struct   request
{
int      cmd;        /*操作行为：读或写*/
struct   buffer_head   *bh;        /*buffer_head 链表*/
struct   buffer_head   *bhtail;
)   ;
```

每个操作请求都维护着一个 buffer_head 链表。每当需要对一个块进行操作时，要将相应的 buffer_head 加入设备请求队列，这个过程有两个优化措施：

(1)加入之前，先要检查已有的块设备操作请求中的块和现在操作的块是否在物理上相邻并且操作行为一致。如果是，则将该操作合入已有的操作请求，而无须生成新的操作请求。

(2)如果不能合并，则生成一个新的块设备操作请求，并加入相应设备请求队列的合适位置，这样做的目的是使磁头的移动路径尽可能短。

5.3.块设备文件的 open 和 read 操作

读取块设备上的数据有两种形式，一种形式是把它看成是某种文件系统组织起来的各个文件集合来读取，另一种形式则是把块设备整体看成一个文件来访问。前一种形式我们已经非常熟悉了，下面我们介绍后一种形式。无论哪一种形式，最终都要归结为对块的操作。

图 5-2 反映了块设备驱动程序的组成，大体上可以分为两部分：

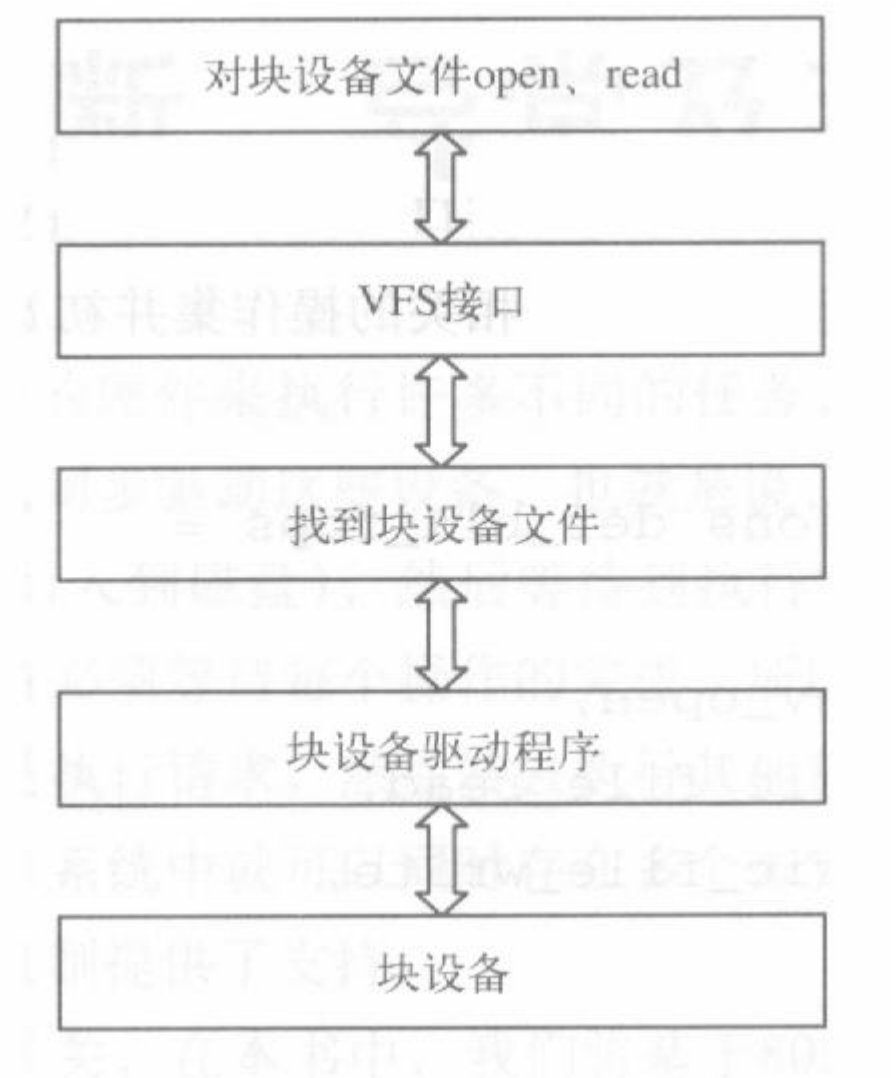


图 5-2　块设备文件的操作流程

(1) 与虚拟文件系统的接口层。

(2)真正对设备(一般是磁盘控制器)操作的部分，当有操作请求以后，会触发这部分。

1. open()函数的实现

在这里我们主要讨论该流程中设备作为一种特殊文件是如何处理的。调用 open_namei()函数的过程中，需要填充一个与设备文件相对应的 inode 对象的各成员项，通常会执行init,special_inode()函数来填充。

```
void init_special_inode(struct inode*inode，umode_t mode，int rdev)
{
inode->i_mode=mode；
if   (S_ISCHR(mode))
{      /*为字符设备文件*/
inode->i_fop=&def_chr_fops；
inode->i_rdev=to_kdev_t(rdev)；
}
else if   (S_ISBLK(mode))
{      /*为块设备文件*/
inode->i_fop=&def_blk_fopS；
```

```
inode->i_rdev=to_kdev_t(rdev);        /*读入设备号*/
inode—>i_bdev：bdget(rdev);
    }
    }
```

del_blk_fops 包含了许多块设备公共的接口，其中，blkdev_open()的作用是根据 inode 对象的设备号信息，在 blkdevs[]中找到设备相关的操作集并初始化 inode 对象的相关项，然后调用该操作集中的 open()操作。

```
struct  file_operations  def_blk_fops  =
{
open：      blkdev_open，
read：      generic  file_read，
write：      generic_file_write，
)  ；
```

在调用 dentry_open()的过程中，首先创建文件对象 f，然后根据 inode 对象的 i_fop 成员初始化对象 f 的 f_op 成员，最终将调用 f->Iop->open()，即 blkdev_open()函数。

2 read()函数的实现

在 Linux 中调用 read()函数，实际上最终调用的是 f_op->read 这个函数指针指向的函数，对于设备文件，实际上就是 generic_file_read()函数。generic_file_read()函数首先计算出要读的块。对于要读的每一块，首先检查在设备缓存中是否已经存在，若不存在，则申请一个缓冲区，并将相应的 buffer_head 加入对应设备的请求队列，然后进程进入 buffer_head 的等待队列睡眠。操作请求将在合适的时候被触发，驱动程序完成读操作后，将操作请求移出请求队列，进程被唤醒后，把设备缓冲区中的数据读到自己的缓冲区。

虽然我们在这里讨论的是块设备文件的读操作，但是对于常规文件、文件系统的元数据的读操作来说，到了块操作这一层后，它们都是相同的。

第二部分 实验部分

第六章

实验一 Linux 使用初步

一、实验目的：

本实验主要是初步熟悉 Linux 的环境，熟悉 Linux 的登陆方式和基本命能编写简单的 Linux 程序和掌握 Linux 调试程序的基本方法。

二、实验内容：

（1）简单命令的使用，编写简单的 Shell 程序

（2）编写简单的 C 语言程序，Makefile 文件的编写。

（3）使用 gdb 调试程序。

三、实验原理及相关

在 Linux 的命令体系中 Shell 占有重要的位置。shell 是用户和 Linux 内核之间的接口程序，如果把 Linux 内核想象成一个球体的中心，shell 就是包围内核的外壳，如图 6-1 所示。当从 shell 或其他程序向 Linux 传递命令时，内核会做出相应的反应。shell 是一个命令语言解释器，它拥有自己内建的 shell 命令集，shell 也能被系统中其他应用程序所调用。

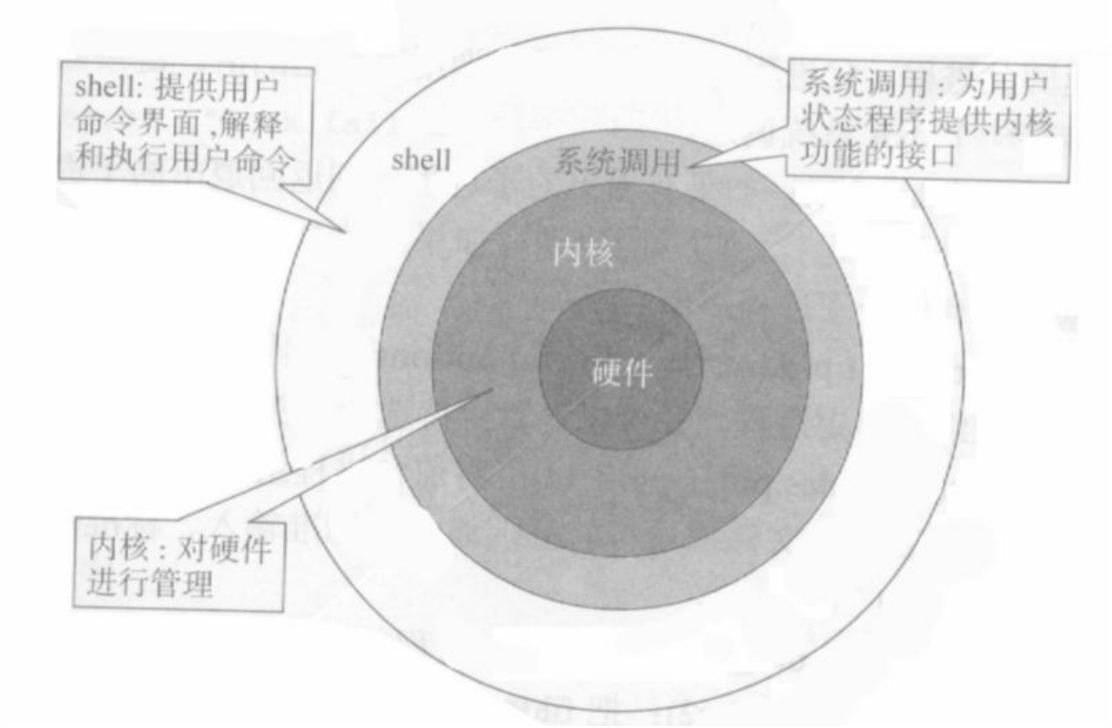


图 6-1 硬件、内核、系统调用及 shell 之间的层次关系

1．运行 sheell 命令文件的方法

shell 命令文件类似 PC-DOS 下的批命令文件。可以用任何一种编辑程序建立 shell 命令文件。可用 3 种方法执行一个已存在的 shell 文件：

1）．用"sh 命令文件”运行

假定已经建立了一个统计当前目录中的文件个数的程序 fcountl，其内容如下：

$cat fcountl

#count files 5n current directory.

ls-1 |sed|d | wc-l

可以用下面方法运行该文件：

$sh　fcountl

如果建立的命令文件有参数，可以直接跟在命令文件名之后给出。

例如：文件 fcount2 的内容为

#count files in specified directory.

1s-|$||sed |d　wc-l

运行如下命令：

$sh fcount2/bin

则把 bin 目录中的文件个数进行统计并显示出来。

2）．用“sh<命令文件名”运行

这是利用输入重新定向，使 shell 命令解释程序的输入取自一个指定命令文件。

例如：$sh<fcountl　实现的功能与 sh fcountl 相同。

3）．用 chmod 命令将命令文件变为可执行文件

一个命令文件之所以不能直接运行，是因为所有的文件在刚建立时，系统赋予它的存取权限为 644(rw-r-r--)，即不可执行。用 chmod 命令可以改变文件的存取权限，使其变为可执行的。

$chmod a+x fcount2

此时，只要直接输入文件名即可运行该命令文件。

$fcount2/usr/bin

应当指出的是，在上述 3 种 shell 命令文件运行方式中，当新建一个 shell 命令文件，对其正确性无把握时，应当使用第一种方法进行调试；当一个 shell 程序已经调试成功，应当使用第三种方法，把它的执行权固定下来。此后只要输入文件名即可执行该程序，并且可以在另一个程序中调用该程序。

2．shell 程序的变量和参数

在用 shell 程序设计语言编写的程序中，可以使用各种变量。变量的引用形式为$var。

在 shell 中，共有 6 种基本类型的变量。

1）　变量的定义

用户可以根据需要，定义自己所需的 shell 变量。

格式：变量名＝表达式

下面是一个显示目前系统注册情况的程序：

$nl loginfo　　---带行号显示文本内容

1．TIME"The current date and time：\C"

2．USERS="The numberOf users：\C"

3．ME="personal stat：\C"

4．echo"$TIME"

5．date

6．echo"$USERS"

7．wh0|wc-l

8．echo"$ME"

9．who am I

在这个程序里，TIME、USERS 和 ME 都是用户自定义的变量。上述 4、6、8 行命令的执行结果是显示变量 TIME、USERS 和 ME 的值，而 5、7、8 行命令才是显示执行的具

体命令的结果。其中变量定义中的\C 是为了以后运行程序显示这个变量时，光标不换行。

运行该命令文件的结果如下：

$loginfo

The current date and time: Sun Jan 3 13: 44:17 BJT 1995

The number of users: 3

personal stat: xiaodi　　try02 Jan 3 09:40

2）　用户可以定义的专门变量

(1)HOME　用户注册后的当前目录，也是 cd 转向的目录。即不带参数的 cd 命令等效于命令 cd　$HOME

(2)PATH　以：(冒号)分隔的目录路径的有序表，是 shell 寻找命令时要检查的路径。UNIX 系统默认的 PATH 值为：/etc:/usr/bin。

(3)MAIL　系统存放用户邮件的文件名，即存放用户邮件的邮箱。这个变量是由 mail 程序使用的。系统设置的 MAIL 变量为“/usr/spool/mail/用户注册名”。

(4)TERM　终端的类型。这个变量由依赖于终端功能的命令(如 more 等)使用。

(5)PSi　系统主提示符。当 shell 等待输入命令时，显示这个提示符。在特权用户下主提示符默认为#。普通用户下主提示符默认为$。

(6)PS2　系统的辅助提示符。在命令结束前遇到换行时，续行中使用的提示符。系统默认的提示符是>。辅助提示符是一个命令在一行写不下时才使用的。

(7)IFS　内部字段的分割符。这是由 shell 内部命令 read 经常使用的变量。系统默认的 IFS 变量的值是空格符、制表符(Tab)和回车符。

(8)TZ　定义有关时区的信息。这个变量的形式为 XXX n YYY。其中 XXX 为 3 个字母的当地时区的缩写，n 为当地时区与格林尼治标准时间的时差，YYY 为 3 个字母的当地夏令时的缩写(可没有)。这个变量是在安装系统时，根据用户的选择设置。在中国通常把这个变量设置为 BJT-8，即北京时间，与格林尼治时差 8 小时。

上述这些变量可用下面不带参数的 set 命令看到其设置情况。

$sct ↓

HOME=/usr/LiXaio

IFS= ↓

MAIL=/usr/spool/Mail/LiXiao

PATH=/bin：/usr/bin：/usr/LiXiao；

PSi=$

PS2=>

TERM=vt220

T2＝BJT-8

3)　位置参数

在 shell 程序文本中限制只能使用$1-$9 共 9 个位置参数。位置参数是在调用 shell 程序的命令行中按照确定的位置决定的变量，在程序名之后输入的每个参数之间用空格分隔。需要说明的是，$0 是一个特殊的变量，不属于位置参数，其内容是当前这个命令文件的名字。

下面用一个例子说明位置参数的用法：

$ cat pp

pp 文件的内容显示如下

echo The first positional parameter is: $1

echo The second positional parameter is: $2

echo The third positional parameter is: $ 3

echo The program is: $0

$ sh pp one two three

执行 PP 文件，运行结果如下

The first positional parameter is: one

The second positional parameter is: two

The third positional parameter is: three

The program is: pp

4). shell 预定义变量

有 6 个由 shell 自定义的变量，如表 6-l 所示：

表 6-l 自定义变量

变量	含义
#	注释符，即其后为注释信息
$#	得到传递给 shell 程序的参数个数
$*	得到传递给 shell 程序的参数
$?	得到上一条命令执行后的返回码
$$	得到进程的标识符
$!	得到后台进程的标识符

由表 5-l 中看出这些变量都由一个$符引导。下面举例说明各变量的作用。

(1) $#

这个变量记录并显示运行 shell 程序时输入的位置参数的个数。

假定 get.num 程序的内容如下；

echo The number of parameter is：$#

当该程序是可执行程序时，运行如下命令：

$get．num one two three four

运行结果为：Thenumber of parameteris：4

(2) $

这个变量将取代从第一个位置参数开始的所有位置参数的内容。该变量并不限制只使用 9 个位置参数。

例：$catshow．para

echo The parameter for this command are： $ *

当该程序是可执行程序时，运行如下命令：

$show．para how do you do

The parameter{or this command are： how do you do

(3) $?

这个变量告诉用户最近一次执行命令的返回值是什么。若为 0，表示执行成功，否则表示不成功。

例：$grep root /etc/passwd

root:：The Super User；/:/bin/sh

$echo$?

O

表示上述命令成功完成返回值为 0。

$grep LiLi/etc/passwd

$echo$?

1

其中，l 表示上述命令执行不成功。即，在/etc/passwd 文件中找不到用户。

(4) $$

这个变量记录并显示当前执行进程的进程标识。

例：执行命令$grep root/etc/passwd

$echo$$ /*返回执行 grep 命令的进程标识

2150 一进程标识号

(5) $!

这个变量告诉用户当前执行的后台进程的标识。

例；执行命令$account&

$$!

15385 一后台进程(执行 acount 命令)的进程号

(6) #

不论#符号出现在一行的什么地方，从该符号开始，以后为注释行。下面给出一个综合使用这些变量的例子，其文件名为 vartest。

$cat vartest ←显示 vartest 程序内容

#tpredefined variables test ←第一行为注释行

echo The name of this program is：$0 ←显示程序名

echoTheargumentsare $ ←显示位置参数内容

echo The number of arguments is$# ←显示位置参数个数

date&

echo The process id of the date command was $： ←显示后台命令的进程号

echo The process id of this shell is$ $ ←显示当前命令的进程号

grep root/etc/passwd ←执行命令 grep

echo The return code from grep was $? ←命令 grep 执行后的返回值显示

执行该程序并输入 n 个参数后的结果为:

$ vartest one two three

The name o{ this program ~s vartest.

The arguments are a b c.

The number of arguments is 3.

satJan2 1: 02:54 BJT 1995.

The process id of the date commmand was 413.

The process id of this shell is 411.

root:: The Super User: /: /bin/sh

The return code from grep was 0.

3 shell 语言

1). if 语句

格式：if 条件表达式；then 命令语句；fi

其中，表达式为任意逻辑表达式，能够给出返回值。shell 中大多数命令都返回一个数值，命令完成后可立即由 shell 变量得到该值($?)，这个值就是表达式的值。

如果条件表达式返回值为零，执行 then 后面的命令部分，否则不执行。例如：

$ if true;then echo hello; fi

结果显示：hello

$ if false；then echo hellO；fi

结果无任何显示。

这里需要说明的两个特殊的逻辑操作符 true 和 false。true 的返回值恒为零，所以句执行 then 后的语句，而 false 恒返回非零值，故此语句不执行。这一点正好与 C 语言相同。例如：

$ true

$ echo $?

0

而$false 命令行执行后，执行命令 echo $?，则显示 1。

if 语句的完整格式为：

if 条件表示式

then 命令

elif 条件表示式

then 命令

else 命令

fi

举一个例子，说明 if 语句的作用：

$if false；then

>echo hello

>elif true;then

2>echo goodbye

>fi

结果显示；

goodbye

2)．for 循环语句

shell 程序设计语言提供了循环结构，用于重复执行 shell 程序的某一部分。for 语句格式：

for 变量 initeml item2 item3 item4 do command done

其中，变量是用户定义的环境变量，in 后的各个 item 组成字符串表，各个串之间用空格分隔。

例如；

$for VAL in l 2 3 4；do echo $VAL;done

其结果显示：

1 2 3 4

3)．while 循环语句

while 语句将 for 和 if 的一些功能结合起来。为了介绍 while，先介绍几个命令：

（1） test 命令

test 命令检查某个条件是否成立。成立时，返回值为 o，否则，返回值为非 0。test 令的各种测试条件还可以用如下 3 种逻辑操作符连接起来：

①-a(与) ②0(或) ③!(非)

格式：test 表达式

例：$if test $VAR

>then

```
>echo hello
>fi
```

如果环境变量 VAR 没有定义，则 test 命令测试失败，if 语句不执行，否则 if 语句执行。将 VAR 用 HOME 变量替换后，该语句执行结果为：hello。这是因为用户的主目录变量(HOME)必然存在的缘故。

test 命令的另一种书写形式：将“test 表达式”用[表达式]代替。例如：

```
$if[$ HOME]
>then
>echo hellO
>fi
```

执行结果为：hello。

```
$if[-f/etc/passwd]；thenecho file exxist；fi
```

执行结果为：file exist。

若 f1＝file 1; F2＝file2

```
$if[! -f $ F1 -a -f $ F2]
>then
>echo"$F1 does't exist，  but  $ F2 does  exist"
>fi
```

下面程序列出指定目录或当前目录中的子目录。

命令文件为 lsdir，其内容如下：

```
$cat lsdir
# listing  subdirectory
if test$ #＝0
then lsdir
else
for i  do
    ls-1$i | grep'd'
done
fi
```

该程序首先对调试程序时给定的位置参数的个数进行测试。如果没有给定位置参数，程序就调用它本身，用．(即当前目录)作为参数。程序调用其自身的技术称为递归。如果调用程序时指定了目录，程序将进入循环，对指定的每个目录都以长格式列表，并找出其中首字母为 d 的各行，即把指定目录中的子目录显示出来。下面就是程序运行的结果：

```
$lsdir  ／usr／lib  ←／usr／lib 是给出的一个位置参数
drwxy-xr-x 2 bin bin 240 Nov 1 14：50 dos
drwxy-xr-x 2 bin bin 256 Nov 1 14：44 help
```

(2)exit 命令

exit 命令是结束 shell 命令文件的执行。exit 命令可以带一个变量参数，作为命令文件向发出调用的 shell 的返回值。例如：

```
$if true；then exit 6；fi
$VAL=$?
$echo $VAL
6
```

(3)expr 命令

虽然 shell 程序设计语言并不精于数值计算，但还是提供了有关的计算命令。

格式：expr(表达式　表达式……)

```
$ expr 4+5
9
```

需要说明的是，expr 表达式中仅允许整数，合法的操作符有+、-、*、 / 和%(求余)。在乘法符号(*)和除法符(/)之前必须冠以反斜杠(\)，以防止这些操作符由 expr 获得之前被 shell 解释。例如：

```
$expr 3\* 4+2\/2
13
```

(4)while 命令的格式

while 命令的格式结合了 for 和 if 某些特性，while 后面跟一个 test 命令，随后是 do-done 循环体。如果 test 命令部分取真，则执行 do-done 部分之后，继续测试 test 部分，并一直继续，直到测试结果为假时为止。

例：生成 10 个文件的程序，文件名为 filel，file2，…，filel0。

```
$VAL=1
$while[$VAL-lt 11]
>do
>touch file$VAL
>VAL='expr$VAL+1'
>done
```

4．case 命令

case…esac 是一种多重选择结构，使用户能够选择几种格式之一，并执行那种格式的。命令列表。

格式：

```
case $VAR in
格式　1)命令列表 1
;;
格式　2)命令列表 2
;;
*)命令 l
.
.
.
命令 n
;;
esac
```

举例如下：

```
$ case $ LOGNAME in
jim)echohcllOjim，welcome back
echo from your vacatiOn
;;
pat)echo pat，do not forget to read your mail
;;
steve)　echo please delete some files，　you are using
```

```
    echo too much disk space...Thanks!
    ;;
    esac
```

5．无条件控制语句 break 和 continue

这两个命令语句用于改变循环的正常操作。当在任何循环中遇到 break 时，无条件地停止该循环的执行，并转向 done、n 或 esac 语句之后的下一个命令。如果在该语句之后没有命令，程序就结束。continue 则使 shell 重新转到整个循环的开始，并继续进行另一次循环。

6．export 命令

在 UNIX 系统中，每一个进程都有一个环境。一个环境是从父进程传送给每个子进程的变量名 / 数值的列表。在父进程中，通过把用户定义的 shell 变量放在命令 export 之后作为参数就可以把这些变量放在一个进程的环境中。

格式：export　变量名 / 数值

例如，childproc 是 shell 的一个命令文件，显示环境中的变量 EDITOR 的值：

```
$cat childproc
#display editor selected
echo The editor you have seleted is$EDITOR
```

先为变量 EDITOR 赋值：$EDITOR＝vi；再用 export 命令将此值用 export 送给进程的环境：$ export EDITOR；

执行不带参数的 export，可以看见 EDITOR 变量已放人环境中：

```
    $export
    export EDITOR
```

之后，执行 childproc 过程，将显示如下结果：

```
    $childproc ↓
The editor you have seleted is vi
```

7．read 命令

read 命令的作用是从标准输入读人一行，分成若干字，给 shell 程序内部定义的变量赋值。第一个字赋给 read 命令后的第一个参数，依此类推。所有剩余的字赋值给最后一个参数。在读到文件结束符之前，read 返回的总是数值。

下面举一个例子，用 read 语句在一个通讯录文件 list 中增加新的名字和电话号码：

```
$ cat nlmknum
        Add a new name &tel to the file.
   echo Type in name please:
   read NAME
   echo Type in number please:
   read NUM
 echo $ NAME     $ NUM >> LIST
```

然后显示 list，看是否新增加了内容。

```
$ cat list
```

8．wait 命令

wait 命令使 shell 等待后台启动的所有子进程结束。wait 的返回值永远为真。

9．exec 命令

当 shell 遇到 exec 命令时，就执行作为参数给出的命令，通过启动这些命令而取代调用 exec 的 shell。

下面举一个例子说明 exec 命令的使用：

```
$ cat mydate
  # Execute the date command indirectly
  exec date
  echo "The command is never executed."
  $ mydate ↓
  Tue Jan S 08: 29:28 BJT 1998
```

由上述命令文件可知，由于 exec 语句后的参数 date 程序取代了读人并执行 mydate 程序中列出的 shell 命令，当 date 完成时，执行 mydate 这个 shell 过程所建的进程就结束了。故 echo 语句将不会被执行。

四、实验步骤：

（1）登录系统

（2）打开控制台窗口，运行 1.5 节的文件操作命令并查看结果。

（3）使用 shell 语言编写一个简单的 9×9 乘法表程序（注意：shell 文件的执行方法有两种：一是 bash shell 文件；一种是给 shell 文件加上可执行的属性）。

（4）编写一个简单的打印"Hello Linux!"的C语言程序，并编译和试看运行结果。

（5 编写Makefile 文件来编译上一步的C程序。

五、实验报告的要求：

1）实验目的

2）实验内容

3）实验结果：描述运行实验程序后观察到的实验结果并判断是否达到预期要求。

4）分析实验结果

5) 体验 Linux 与 Window 的不同使用方式

六、思考题

1) Linux 的发展状况如何？主要用来做什么的？Linux 能为我做些什么？我能用 Linux 来做些什么？

2) 我该怎么来学习 Linux？

第七章

实验二 Linux 进程控制

一、实验目的：

为了理解和掌握 UNIX(和 Linux)进程控制系统调用的功能，这里给出了进程控制的主要几个系统调用命令的格式和如何利用系统调用命令进行编程，以便通过学习，使学生理解如何创建一个进程、改变进程执行的程序、进程终止和父子进程的同步等。

二、实验内容:

UNIX 操作系统的底层接口——系统调用是用户程序与系统内核的接口。本实验主要讲述 UNIX(和 Linux)进程控制部分的主要系统调用，以及如何利用系统调用进行编程，以便通过实验对进程之间的通信有一个较深入的理解和掌握。实验内容分两个部分：

（1）利用 fork 函数完成进程产生，exit 完成进程的消亡，用 wait 函数完成父子进程的同步

（2）利用 vfork、exec 完成进程的产生。

三、实验原理及相关 API

1　进程创建的函数

进程创建的函数

UNIX 操作系统的底层接口：系统调用是用户程序与系统内核的接口。创建一个新进程的惟一方法是由一个已存在的进程通过调用 fork()、vfork()和 clone()函数来实现。已存在创建者进程叫做父进程，被创建进程叫做子进程。3 个系统调用的描述如下。

1). fork()函数

用 fork()函数创建进程时，语句调用序列如下：

#include <sys/types.h>

#include <unistd.h>

pid_t　fork(void)；

返回：调用正确完成时，给父进程返回的是被创建子进程的标识，给子进程自己返回的是 0；创建失败时，返回给父进程的是-1。

fork()函数是一个单调用双返回的函数。也就是说，该函数由父进程调用，执行成功时，在父进程中返回子进程标识，在子进程中返回 0。fork()调用后，子进程是父进程的一个复制。都是从 fork()调用语句的下一个语句开始执行。

在 Linux 环境下，fork()是采用 copy_on_write 机制实现的。仅需要创建一个进程控制块和复制父进程的页表，与父进程共享地址空间。

(1)fork()系统调用的示例

例 7.1 fork()系统调用的使用例子。

/ * file1.c 的源程序* /

```
    include   <stdio. h>
    include   <sys/types. h>
    include   <unistd. h>
int glob = 3
int main(void)
{ pid_t pid;
```

```
int loc = 3;
printf("before    fork(): glob= %d,loc= %d. \n",glob,loc);
if ((pid=fork())<0) (
        printf("fork()    error. \n")
        exit(0)
else    if (pid==0) {          / * child'context. /
                glob++
                loc--;
          printf("child process changes glob and loc: \n");
            }                              / * child end * /
            else                          / * parent'context * /
              printf("parent process doesn't change the glob \
                and loc: \n');    / * parentend* /
printf("glob= % d, loc= % d\n",glob,loc) ;/* parent and child context * /
      exit(O);
```

(2)例 7.1 的运行结果和分析

```
$ ./a.out       /*程序运行的命令行*/
```

该程序运行的情况可能有 3 种结果。若运行到 fork()时，出现错误，输出“fork() error.”，程序故障终止。

由于程序运行时产生两个进程，由 main()函数产生一个父进程以及程序执行中通过调用 fork()产生的一个子进程。当执行到 fork()后形成两个并发进程。根据系统调度情况，可能产生两种运行结果。一种可能的运行结果为：

```
before fork()：glob=3，loc=3.
child process changes the glob and loc：
glob=4,loc=2
parent process doesn't change the glob and loc：
glob=3,loc＝3
```

另一种可能的运行结果为：

```
before fork()：glob=3，loc=3.
parent process doesn't change the glob and loc：
glob=3，loc＝3.
child process changes the glob and l0c：
glob＝4，loc＝2
```

2). vfork()函数

用 vfork()函数创建进程时，语句调用序列如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t    vfork(void)；
```

返回值：正确完成时返回值与 fork()返回值一样，返回子进程的标识，否则返回-1。用 vfork()函数创建进程时，通常使用 exec()函数紧跟其后，以便为新创建进程指派另外一个可执行程序。用 vfork()创建的新进程并不完全复制父进程的数据区。vfork()与 fork()另一个不同点表现在父子进程的执行顺序上。fork()不对父子进程的执行次序进行任何限制，而 vfork()调用后，子进程先运行，父进程挂起，直到子进程调用 exec()或 exit()之后，父子进

程的执行顺序才不再有限制。否则，如果子进程在调用 exec()或 exit()之前，父进程被激活，就会造成死锁。

(1)vfork()系统调用的示例

理解 vfork()系统调用。为了进一步理解 vfork()与 fork()之间区别，下面给出一个例子。

例 7.2

```
/*file2.c 的源程序*/
    #include   <stdio.h>
    #include   <sys/types.h>
    #include   <unistd.h>
    int g1ob=3；
    int main(void)
    {   pid_t pid；
    int loc＝3；
    if((pid＝vfork())<0){
    print{(" vfork()   error＼n")；
    exit(0)；
    }
    else   if(pid＝＝0){      /*child’context*/
          glob++;
          loc--;
          printf("child process changes the glob and loc\n");
          exit(0);
    }            /*   child * /
    else                        / * parent'context * /
          printf("parent process doesn't change the glob   and loc\n");   / * parent * /
printf("glob= %d,val= %d\n”,glob,loc) ;/* parent and child ' context * /
exit(0)
}
$ ./a.out      / *程序运行的命令行* /
child process changes the glob and loc
parent process doesn't change the glob and loc
glob= 4,val= 2
```

分析：从上面的运行结果看出，vfork()创建进程成功后，父进程挂起，子进程先运行，在父进程的数据区中修改了变量 glob 和 loc 的值，并输出 child process changes the glob and loc，然后调用 exit(0)，子进程退出。父进程继续运行，执行两条输出语句。由此看出，变量 glob 和 loc 确实被子进程修改了。而且该程序的运行结果是确定的，这就是 vfork()和 fork()的不同之处。

3)．clone()函数

用 clone()函数创建进程时，语句调用序列如下：

```
#include<sys/types.h>
#include<schedle.h>
int clone(int(*fn)(void*)，void   *child_stack，int flags，void   *arg)；
```

返回值：正确完成时返回值与 fork()的返回值一样，否则返回-1。

clone()函数与 fork()很相似，创建一个新进程。但与 fork()又不完全一样，clone()函数

允许子进程与调用进程共享其执行上下文，如存储空间，文件描述符表和信号处理程序。

clone()函数的主要用来实现线程。当用 clone()创建子进程后，执行 fn(arg)的函数指定的应用。fn 是一个指向子进程开始执行时调用的一个函数的指针。arg 则是传递给 fn 函数的变量。当 fn(arg)函数返回时，子进程终止。fn 函数返回的是一个整型数，是子进程的退出码。子进程也可以显式地调用 exit()终止自己或在接收到致命错误时而终止。child_stack 变量说明子进程使用的堆栈。因为子进程可能与调用进程共享存储器但子进程的执行不可能与调用进程使用同一个栈。child stack 是调用进程为子进程建立的一个指向堆栈空间的指针。当子进程终止时，flags 低字节包含了发送给父进程信号。如果没有信号被说明，当子进程终止时，没有信号向父进程发送。

注意：clone()函数调用是 Linux 专用的，不能移植到其他系统进行多线程程序设计。

2　给进程指定一个新的运行程序的函数 exec()

一个进程调用 exec()函数来运行一个新程序。之后该进程的代码段、数据段和堆栈段就被新程序的所代替。新程序从自己的 main()函数开始执行。

exec 函数有 6 种不同的形式，任何一个都可以完成 exec()的功能，只是调用参数不同。exec()函数的 6 种基本形式如下：

include　<unistd. h>

int execve(const char * pathname, char *const argv[],char * const envp[]);

int execl(const char * pathname,const char arg 0 ,.../(char *)0 */);

int execv(const char * pathname, char ** const arg[]);

int execle(const char * pathname,const char * arg 0 ,/* (char) 0 ,char * const envp[] */);

int exclp(const char * filename,const char *~arg 0 ,.../* (char*　)0 */);

int execvp(const char * filename, char * const argv[]) ;

正确返回：0；错误返回：-1。

由说明可见，上述 6 个函数有如下几个区别：

前 4 个函数的第一个参数为可执行文件的路径名，且文件将由环境变量 PATH 所指定的路径中查找。后两个函数为文件名，且这两个函数的可执行文件不是二进制程序，而是 shell 脚本程序，由/bin/sh 解释执行。

可执行程序传递参数的方式不同：execl、execle 和 execlp 传递的参数是分离的形式，且以空指针结尾；其他 3 个函数 execv、execve 和 execvp 则要求建立一个指针数组，指向这些参数，并且把这个数组的地址作为参数进行传递。

为可执行程序传递的环境变量不同：execle 和 execve 允许将环境变量设置成字符串指针作为参数进行传递，而其他 4 个函数只能复制父进程的环境变量。一般情况下，父进程允许传递环境变量给子进程，而有些情况，父进程需要为子进程设置某些特殊环境变量。如 login 程序，需要根据一些用户设置来设置环境变量，此时就应使用这两个函数的功能。这里的环境变量是指供用户程序使用的一些数据。用户可以改变这些数据，增加新的数据或删除一些数据，以便适应各个用户的需要。

例 7.3　可以打印出传递给运行进程的参数和所使用的环境变量的程序。

```
/*file3.c 的源代码如下：*/
#include <stdio.h>
#include <unistd.h>
extern char　**environ;
int main(int argc,char * argv[])
{
```

```
    int i;
    printf("\nEnvironment: \n")
    for (i=0;environ[i]! =NULL;i++)
        printf ("% s\ n", environ[i] );
}
```

例 7.4　一个简单的 exec()调用的例子。

(1)exec()系统调用的示例

```
/*file4.c 的源代码如下：*/
#include <stdio.h>
#include<sys/types.h>
#include <unistd.h>
#include <sys/walt.h>
char   xenv_list[]＝{"USER：kate"，"PATH＝/trap"，NULL，}；
int main(void)
{
pid_t pid；
if((pid＝fork())<0){
      printf("fork error\n")；
      exit(0)；/*create subprocess is fualt，parent process exit"/
}
else   if (pid＝＝0){      /*子进程被创建，子进程的上下文*/
if(execle("/usr/john/bin/printal"，/*子进程的可执行文件的路径!*/
"printl"，"arg1"，"arg2"，(char*)0，/*arguments*/
env_list)<0)      /*environment*/
printf("execle error\n")；
exit(0)；/*子进程退出*/
}
if   (waitpid(pid，NULL，0)<0)
printf("Wait error\n")；
exit(0)；
if((pid＝fork())<0){/*再创建子进程*/
printf("fork error\n")；/*创建错误，父进程退出*/
exit(0)；
}
else if(pid＝＝0){   /*子进程创建成功时的上下文*/
if(execlp("printl"，/*子进程的可执行的文件名*/
"printl","arg1"，(char*)0)<0)      /*arguments*/
printf("execle error\n")；
exit(0)；
}
exit(0)；/*父进程退出*/
      }
```

子进程的可执行文件 printl 的源码为：

```
/*源文件名为 printl.c*/
```

```
#include   <stdio.h>
int main(int argc，char   *argv[])
{   int n；
char **ptr；
extern char **environ；
for(n＝0；n<argc；n++)
printf("arg[％d]：％s\n", n, argv[n])；
for(ptr=environ；*ptr!=0；ptr++)
printf("%s\n",*ptr)；
exit(0)；
}
```

(2)例 7.4 程序运行的结果和分析

printl.c 是主程序 file4.c 中 exec 函数指派的可执行程序的源文件名，将传递给程序的参数和环境变量进行输出。

file4.c 经编译和链接后，形成可执行文件 a.out。当执行 a.out 时，其执行结果不固定。下面是一种可能的结果。

```
$ ./a.out   /*程序运行的命令行*/
  ars[0]：printl
  arg[l]：argl
  arg[2]：arg2
  USER＝kate
  PATH＝/tmp
  arg[0]=printl
  are[l]=argl
  USER＝zlf
  HOME＝/usr/zlf
  LONGNAME＝2lf
  EDITOR＝/usr/ucb/vi
```

分析：由结果可见，程序首先调用了 execlp()，传递的参数有可执行程序的路径名、程序所需的参数及程序执行时的环境变量。故 a.out 的执行结果为 arg[o] printl(即程序)，arg[1]：argl，arg[2] :arg2(即参数)和 USER＝kate，PATH：/tmp(即环境变量)。之后，该子进程终止。系统调度另一个子进程运行。接着调用 execlp()，传递的参数有可执行程序名、程序所需的参数并继承父进程的环境变量。而且必须保证/usr/zl{/bin 在环境变量 PATH 中。之后，a.out 输出 arg[0]＝printl(即程序名)，arg[l] argl(变量名)以及环境变量的值 USER=zlf，HOME=/usr/zlf，LONGNAME=2lf，…，EDITOR=/usr/ucb/vi 等。

3　进程终止

在 Linux 系统中，进程通过执行系统调用 exit()来终止自己，格式如下：

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

或

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

进程结束时，status 是传递给父进程的该进程的结束状态码。对这两个系统调用的执行过程之间的区别通过图 7-1，就可以有一个较为直观的认识。

从图 7-1 中可以看出，exit()函数在调用 exit 系统调用之前要检查文件的打开情况，把文件缓冲区中的内容写回文件，就是图中的“清理 I/O 缓冲”一项。之后与_exit()函数一起，使进程停止运行，通过调用 exit()系统调用，清除其使用的内存空间，并释放其内核中的各种数据结构。然后，等待父进程进行善后处理。对这两个系统调用的执行过程之间的区别通过图 7-1，就可以有一个较为直观的认识。从图 7-1 中可以看出，exit()函数在调用 exit 系统调用之前要检查文件的打开情况，把文件缓冲区中的内容写回文件，就是图中的“清理 I/O 缓冲”一项。之后与_exit()函数一起，使进程停止运行，通过调用 exit()系统调用，清除其使用的内存空间，并释放其内核中的各种数据结构。然后，等待父进程进行善后处理。

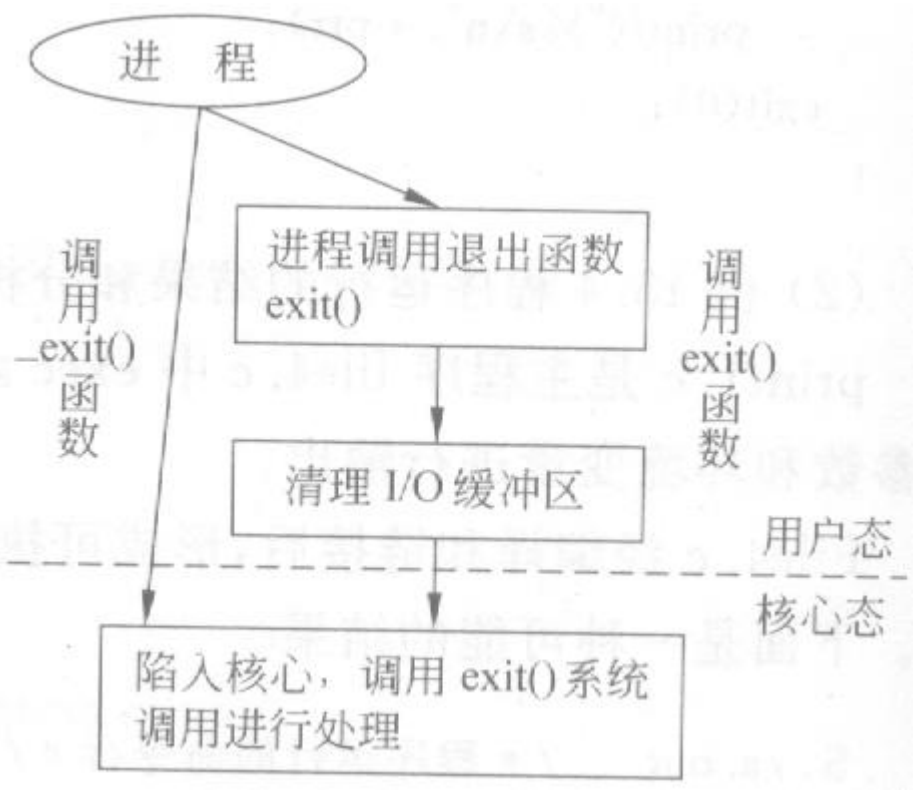


图 7-1 进程终止的系统调用

1).exit()系统调用的示例

```
/*exit_test.c*/
#include<stdlib.h>
void main()
{
   printf("this isaexitsystemcall!\n");
   exit(0);
   printf("this sentence never be displayed!\n");
}
```

2).程序运行结果和分析

编译并运行：

```
$gcc    exit_test.c-Oexit_testl
$./exit_test
this is a exit system call!
```

由于程序运行到第一个 printf 时，运行进程调用 exit(0)；使自己终止，故第二个 printf

语句没有执行。

```
/*_exit_test.c*/
#include<unistd.h>
main()
{
printf("this is a    exit system call!＼n");
printf("content in buffe");
_exit(0);
}
```

编译并运行：

```
$gcc _exit_test.c    -o _exit_test
$./_exit_test
this is a    _exit system call!
```

4　父子进程同步

当一个进程结束时，产生一个终止状态字，然后系统核心发一个 SIGCHLD 信号通知父进程。因为子进程结束是异步于父进程的，故父进程要结束之前，要同步等待子进程终止。这是通过调用系统调用 wait 或 waitpid 来实现的。

当父进程通过调用 wait 或 waitpid 同步等待子进程结束时，可能有以下几种情况：

(1)如果子进程还未结束，父进程阻塞等待；

(2)如果子进程已经结束，其终止状态字 SIGCHLD 放在指定位置等待父进程提取，这时，父进程可立即得到终止状态字并返回；

(3)如果没有子进程，父进程立即错误返回。

wait 和 waitpid 的调用形式如下：

```
    #include<sys/type.h>
#include<sys/wait.h>
pid_t wait(int * statloc);
pid_t wait((int *)0);
pid_t waitpid(pid_t pid，int * statloc，int options)
```

其中，statloc 是一个整数指针，指向存放子进程的终止状态的位置。若不关心子进程的终止状态，可传递一个空指针。

正常返回时为终止子进程的标识 pid；错误返回时为-1；其他为 0。

wait((int*)0)系统调用只用于进程同步，不关心终止进程的状态信息。也可以简单写为 wait(0)。

waitpid 则用于当有多个子进程运行时，父进程只等待指定 pid 进程终止。其参数 pid 的含义如下：

- 参数 pid<-1 时，匹配任何进程组的标识等于参数 pid 的绝对值的终止的子进程；
- 参数 pid=-1 时，匹配任何终止的子进程；
- 参数 pid＝0 时，匹配与当前进程是同一个进程组的任何子进程；
- 参数 pid>0 时，匹配其进程标识与 pid 相等的任何子进程。

另外，在调用 waitpid 时，参数 options 的取值及含义如下：

如果参数 options 中的 WNOHANG 位被设置，若没有任何子进程终止时，该调用立即以 0 返回。若有任何一个进程终止，返回终止进程的 pid，并将其终止状态存放在 statloc 中。若执行中有错误，返回-1；

如果参数 options 中的 WUNTRACED 位被设置，返回终止子进程的状态。若无终止进程，也立即返回。

wait 与 waitpid 的差别在于后者有较强的灵活性。当父进程调用 wait()时，若无子进程结束，则父进程等待；而 waitpid()函数可以选择不挂起立即返回，或指定等待，直到 N 个进程结束。

1．系统调用 wait()示例

例 7.5　父子进程通过 exit()和 wait()系统调用的使用，观察父子进程之间的同步过程。

```
/*file5.c 源程序*/
main()
{
int child;
if((child=fork())＝＝0)
{printf("child's PID is ％d＼n"，getpid())；
exit(0)；
}
printf("child's　PID to return to　parent　is ％d＼n"，child)；
wait()；
}
```

分析：这个程序是比较简单的。执行 main()函数的进程(即父进程)先创建一个子进程。若调度子进程运行，通过调用 getpid()，获得自己的进程标识，之后退出；若调度到父进程，父进程打印由 fork()调用返回的子进程标识 child，等待子进程终止后，最终执行到 main()的终点，正常结束。

例 7.6　父子进程之间的同步之例

```
/*file6.c 源程序*/
#include<stdio.h>
main()
{int pidl；
if(pidl＝fork())　/*create childl*/
{ if(fork())/*create the child2*/
{printf("parent's　context．\n")；
printf("parent is waiting the　childl　terminate．\n")；
wait(0)；
printf("parent is waiting the child2 terminate．＼n")；
wait(0)；
printf("parent terminates．＼n")；
exit(0)；
}　　.
else
/*child2*/
printf("child2's context．\n)；
sleep(5)；
printf("child2 terminate．\n)；
exit(0)；
}
```

```
}
else{ if(pidl＝＝0)/*childl*/
    {printf("childl's context。\n");
    sleep(10);
    printf("childl terminate。\n");
    exit(0);
    }
    }
}
```

分析：上述程序是父进程首先创建一个子进程，若成功，再创建另一个子进程，之后 3 个进程并发运行。究竟谁先运行，是随机的，可由运行结果知道它们的运行顺序。可能产生的一个结果是：

父进程运行输出是：parent，scontext.

parent is waiting the childl terminate.

parent is waiting the child2 terminate.

parent terminates.

子进程 1 的运行结果是：

childl’s context.

childl terminate.

子进程 2 的运行结果是：

child2’s context.

child2 terminate.

分析:上述程序是父进程首先创建一个子进程，若成功，再创建另一个子进程，之后 3 个进程并发运行。究竟谁先运行，是随机的，可由运行结果知道它们的运行顺序。可能产生的一个结果是：

父进程运行输出是：parent，scontext.

parent is waiting the childl terminate.

parent is waiting the child2 terminate.

parent terminates.

子进程 1 的运行结果是：

childl，s context.

childl terminate.

子进程 2 的运行结果是：

child2，s context.

child2 terminate.

四、实验步骤

1. 熟悉相关的原理和 API 函数。

2. 利用相关的 API（fork 和 vfork）实现进程的产生，进程的消亡和父子进程同步的过程，学生可选用不同的 API 来实现。具体实现可参考程序清单 1 中的实现。

3. 对实现的结果进行分析。

程序清单 1：

file1.c 进程的产生（fork 实现）

file2.c 进程的产生(vfork 实现)

file3.c 环境变量
file4.c 利用 exec 创建的进程
exit_test.c 进程终止
file5.c file6.c 父子进程的同步

五、实验报告的要求：

实验结束后，应整理实验报告，其内容应包括：

1、实验题目
2、程序设计思路
3、写出程序源代码
4、写出程序运行结果
5、对实验结果进行分析、总结并说明产生该结果的原因

六、思考题

（1） 进程的主要特征是什么？
（2） 系统怎么创建一个新的进程？
（3） 请使用 fork-exec 组合的传统方法编写一个 C 程序，用新创建的进程来执行“ps -f”命令。
（4） 试分析 file6.c 程序可能产生的所有结果。

第八章

实验三 Linux 进程通信(一)

一、　　实验目的：

为了理解和掌握 UNIX(和 Linux)进程通信系统调用的功能，这里给出了进程通信实现机制中使用的系统调用命令的格式和如何利用系统调用命令进行进程通信编程，以便通过学习，提高学生对进程通信系统调用的编程能力。

二、　　实验内容:

利用命名管道和无名管道编写程序实现进程间通信；利用共享内存方式编写程序实现进程间通信。

三、　　实验原理及相关管道通信机制

通过使用管道实现两个或多个进程之间的通信。所谓管道，就是将一个进程的标准输出与另一个进程的标准输入联系在一起，进行通信的一种方法。同组进程之间可用无名管道任意通信，而不同组进程之间可通过有名管道进行通信。管道通信机制分为无名管理通信和以命令行为参数的管道通信。

图 8-1 说明进程 P2 向管道写，P1 从管道读的通信的实现过程。

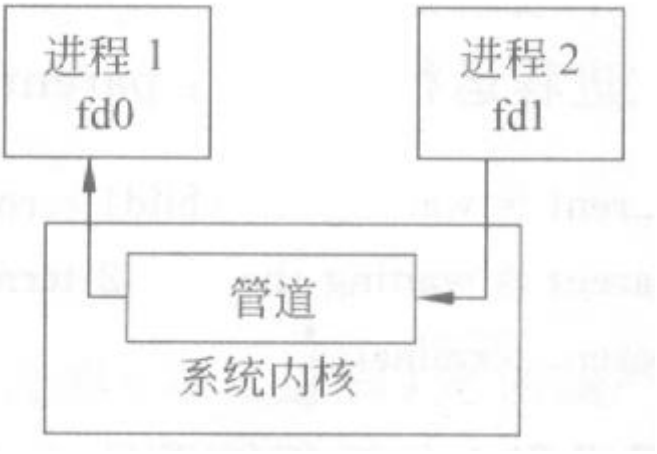


图 8-1 进程间通过管道进行通信

1．无名管道的通信

(1)创建无名管道的格式

```
#include <sys/types．h>
#include <ctype.h>
#include <unistd.h>
int pipe(int filedes[2]);
```

如果正确返回：0；错误返回：-1。

(2) 无名管道 pipe()的使用

例 8.1　使用无名管道 pipe()，进行父子进程之间的通信。

```
/*file7.c 源程序*/
charparent[]＝"a　message to pipe，communicatiOn.\n";
main()
```

```
{int pid，chanl[2]；
char buf[100]；
pipe(chanl)；
pid＝fork()；
if(pid<0)
{printf("to create child error\n")；
    exit(1)；
  }
  if(pid>0)
    {close(chanl[0])；/*父进程关闭读通道*/
    printf("parent process sendsamessagetOchild．\n")；
    write(chanl[1]，parent，sizeof(parent))；
    close(chanl[1])；
    printf(”parentprocess waits thechildtOterminate．\n")；
    wait(0)；
    printf("parentprocess terminates．\n")；
    }else{
    close(chanl[1])；/*子进程关闭写通道*/
    read(chanl[0]，buf，100)；
    printf("Themessageread bychild process fromparentis：%s．\n",buf)；
    close(chanl[0])；
    printf("child processterminates\n")；
    }
}
```

(3)观察运行结果

当父进程首先被调度时，例 8.1 运行结果是：

parent process sends a message to child.

parent process waits the child to terminates.

之后父进程阻塞等待子进程终止。当系统调度子进程运行时，输出如下信息：

The message read by child process from parent is：A message to pipe communication.

child process terminates.

之后父进程被唤醒，调度运行，输出如下信息后程序退出。

parent process terminates.

分析：例 8.1 中，父进程先使用 pipe(chanl)系统调用打开一个无名管道，之后创建。一个子进程。子进程复制父进程的打开文件表。为了正确通信，父进程关闭读通道 close (chanl[0])，子进程关闭写通道 close(chanl[1])。父进程向管道写，子进程从管道读。完成一次通信后，父子进程分别关闭自己的写/读通道，管道文件消失。使用无名管道通信时，是使用临时文件的方式实现进程之间的批量数据传输。若通信双方不是父子关系，不能直接建立通信联系，而要由创建它们的共同的父进程为它们建立管道。再通过复制父进程映象使两者建立联系。

2)．以命令行为参数的管道通信

(1) 命令格式

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

#include <sys/types.h>

#include <ctype.h>

FILE popen(const char crndstring，const char type)；

正确返回：文件结构的指针；错误返回：空指针。

int pclose(FILE *fp)；/*关闭函数*/

正确返回：cmdstring 结束时的状态；错误返回：-1。

其中，cmdstring 是 shell 要求的一个完整的命令行参数和选项。type 指出管道访问方式：读(r)或写(w)。若管道的 type 为 r，那么管道的输入端与命令行 cmdstring 的标准输出端相连。也即，父进程以 r 方式打开管道并成功地获得管道文件的指针，被创建的子进程执行命令行，向管道输出。如图 8-2 所示。若管道的 type 为 w，那么管道的输出端与命令行 cmdstring 的标准输入端相连。也即，父进程以 w 方式打开管道并成功地获得管道文件的指针，被创建的子进程执行命令行，从管道读人。popen 和 pclose 这两个函数的作用类似于 fopen 和 fclose 函数。popen 用于创建管道，函数内部完成 fork()和 exec()函数的调用，并执行命令串 cmdstring，返回一个 FILE 结构的指针，用于访问管道。pclose()用来关闭管道，关闭标准输入输出流，等待命令行执行完后返回结束时的状态。若命令行不能正确执行，返回-1。

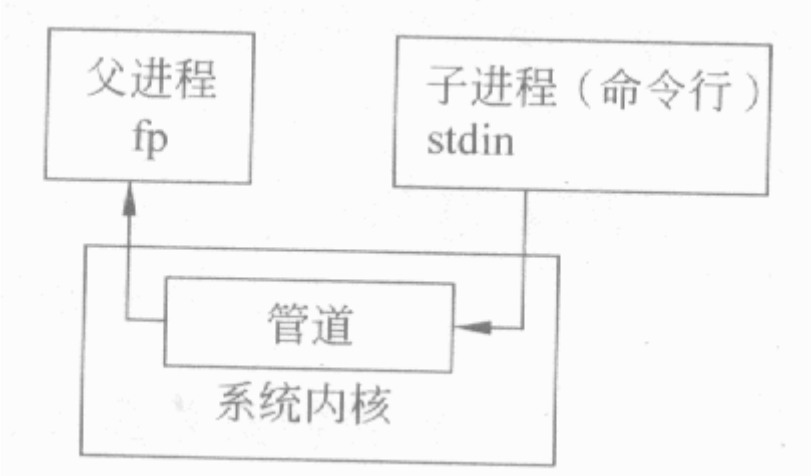


图 8-2 父进程执行 popen(command,"r")的结果

(2) 打开一个以命令行为参数的管道文件，完成进程之间的通信

例 8.2 以命令行为参数的管道文件的示例。

假设有一个可执行程序 chcase，从标准输入设备读字符，将小写字母转换成大写字母并进行输出。主程序使用 popen 创建管道，实现将某文本文件中的字母转换成大写字母。其中的文本文件名作为参数传进来。

```
/*file8.c 源程序*/
  #include<sys/wait.h>
  #include<stdio.h>
  int  main(intargc，char*argv[])
  {
    char line[MAXLINE];
    FILE* fpin，*fpout；
    /*检查输入的命令行有无带参数*/
   if(argc!＝2){  /*无*/
    fprintf(stderr，"usage：a.out<pathname>\n");
```

```
    exit(1);
     }
  /*打开文本文件*/
 if((fpin＝fopen(argv[1]，"r"))＝＝NULL){
    fprintf(stderr，"can't open％s\n"，argv[l]););
    exit(1);
  }
  /*创建写类型管道*/
 if(fpout＝popen("chcase"，"w"))＝＝NULL){
  fprintf(stderr，”popen error\n”);
  exit(1);          。          •
  /*从文件中读出数据，写入管道中*/
  while((fgets(1ine，MAXLINE，fpin)!＝NULL){
   if(fputs(1ine，fpout)＝＝EOF){
    fprintf(stderr，"fputs error    to    pipe.\n");
    exit(1);
    }
    }
    /*判断读文件是否有错误*/
     if(ferror(fpin)){
         fprintf(stderr，"fgets error.\n");
         exit(1);
         }
      /*关闭管道*/
     if(pelose(fpout)＝＝-1){
         fprintf(stderr，"pclose error.\n");
         exit(1);
        }
         exit(0);
         }
```

说明：先打开文本文件，通过函数 popen 创建一个可写管道，将命令行 chcase 的输入与管道的输出连接起来，然后向管道输入数据，那么，命令行就可以通过管道接收文本文件的数据了。

3)有名管道的通信

(1) 创建一个有名管道的系统调用 mknod。

有名管道的使用方式与无名管道不同。有名管道可被任何知道其名字的进程打开和使用。为了使用有名管道，进程要先建立它，并与它的一端相连。创建有名管道的进程叫服务器进程，存取管道的其他进程叫客户进程。通信双方必须首先创建有名管道后，才能打开管道进行读写。当文件不再需要时，要显式删除。创建一个有名管道的命令同创建一个目录文件、特别文件一样，使用如下命令：

```
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
#include<fcntl.h>
```

int mknod(const char *pathname，mode—t mode，dev_t dev)；

或

#include<sys/types.h>

#include<sys/stat.h>

int mkfifo(const char *pathname，mode_t mode)；

其中，pathname 是新创建的有名管道的文件路径名，mode 是被创建文件的类型和存取方式，dev 是文件所在的设备。对于有名管道，dev 这个参数为 0。有名管道被创建后，在系统中有一个目录项和对应的磁盘 i 节点与之对应，但并不将它打开。该系统调用隐含指定 O_CREATIO_EXCL，即创建一个新的 FIFO 文件，成功时，返回 0，不成功(即文件已经存在)，返回-1。

(2) 打开一个有名管道

由于有名管道创建时并没有打开，因此必须显式地使用如下的系统调用将文件打开。

open(pathname，oflg)；

其中，pathname 是要打开的有名管道的路径名，oflg 是文件打开时的存取方式。打开一个有名管道与打开一个普通文件没有区别，只是通信的发送者以 OWRONLY 只写方式、接收方以 ORDONLY 只读方式打开。

进程间使用有名管道实现通信时，必须有三次同步。

第一次是打开同步。当一个进程以读方式打开有名管道时，若已有写者打开过，则唤醒写者后继续前进，否则，睡眠等待写者。当一个进程以写方式打开有名管道时，若已有读者打开过，则唤醒读者后继续前进。否则等待读者。

第二次是读写同步。其同步方式与 pipe 相同。允许写者超前读者 1024 个字符。当一次写超过 1024 时，超过的字符要写入时，则写者必须等待。读者从有名管道读时，若没有数据可读则等待。若有数据可读，读完后要检查有无写者等待，若有唤醒写者。而且要求读写两方要随时检查通信的另一方是否还存在，一旦有一方不存在，应立即终止通信过程。

第三次是关闭同步。当一个写进程关闭有名管道时，若发现有进程睡眠等待从管道。读，则唤醒它，被唤醒进程立即从读调用返回。当一个读进程关闭有名管道时，若发现有进程睡眠等待向管道写，则唤醒它，并向它发一个指示错误条件的信号后返回。最后一个关闭有名管道的进程，释放该管道占用的全部盘块及相应主存 i 节点。

有名管道打开后，就可以使用读写命令进行读写，读写完成后就立即关闭。有名管道文件关闭后，文件本身并没有消失。有名管道的读、写和关闭动作与普通文件完全相同。

(3)有名管道的使用

创建有名管道的例子。

```
/*file9.c 有名管道源程序*/
  #include<fcntl.h>
  charstring[]＝"this is a examplet  show fifo communication"；
  main(argc，argv)
 int argc；
  char *argv[]；
  {
    int fd；
    charbur[256]；
    int i；
    mknod("fifo"，010777，0)；/*创建属性为 010777 的管道文件*/
    /*其中，010 为管道文件的类型，777 为允许读写执行的属性*/
```

```
if(argc＝＝2)
{
fd＝open("fifo"，O_WRONLY)；
}
else
{
fd＝open("fifo"，O_RDONLY)；
}
for(i=0；i<26；i++)
{
if(argc＝＝2){
printf("\n I have wrote：％s"，string)；
write(fd，string，45)；
string[0]+＝1;
    }
    e1se
    {
    read(fd，buf，256)；
    printf("\nThe Ontext by I have read is　：　！　％s"，buf)；
    buf[0]＝'\0';
    }
    )
    close(fd)；
    }
```

命令行格式：一个进程使用“./a.exe　1&”在后台运行，而另一个进程使用“./a.exe”在前台运行。

四、实验步骤

1. 熟悉相关的原理和 API 函数。

2. 利用命名管道和无名管道编写程序实现进程间通信，利用共享内存方式编写程序实现进程间通信。

3. 对实现的结果进行分析。

程序清单 2

file7.c　使用无名管道 pipe()，进行父子进程之间的通信

file8.c　命令行为参数的管道文件

file9.c　有名管道

五、实验报告的要求：

实验结束后，应整理实验报告，其内容应包括：

1、实验题目

2、程序的设计思路

3、写出程序源代码

4、写出程序运行结果

5、对实验结果进行分析、总结并说明产生该结果的原因

六、思考题

(1).试分析 file7.c 程序可能产生的结果。

(2).编写一个无名管道和有名管道的程序，实现有家族关系和无家族关系的进程之间的通信。父进程负责从键盘读，并将小写字母转换成大写，子进程负责将父进程读人和转换的字符进行输出。

第九章

实验四 Linux 进程通信(二)

一、实验目的：

为了理解和掌握 UNIX(和 Linux)进程通信系统调用的功能，这里给出了进程通信实现机制中使用的系统调用命令的格式和如何利用系统调用命令进行进程通信编程，以便通过学习，提高学生对进程通信系统调用的编程能力。

二、实验内容:

利用命名管道和无名管道编写程序实现进程间通信,利用共享内存方式编写程序实现进程间通信

利用消息队列、信号量、信号编写程序完成进程间通信

三、实验原理及相关 API

消息缓冲是 UNIX 系统进程之间进行大量数据交换的机制之一。消息缓冲是基于消息队列的。发送进程将消息挂人接收进程的消息队列，接收进程从消息队列中接收消息。消息是指具有类型和数量的一个数据。消息分私有的和公有的。如果消息为私有的，只能被创建消息队列的进程和其子进程访问；如果是公有的，可以被系统中知道消息队列名的所有进程访问。消息可以按类型访问，因此，不必按序访问。

1）.消息缓冲使用的数据结构

为了实现消息缓冲，系统设置了相应的数据结构，下面是其结构和作用。

(1)消息缓冲区

消息缓冲区是用来存放消息的，其结构定义如下：

```
struct msgbuf{
long mtype;       /*消息的类型，可以为正、负整数或 0*/
charmtext[N];     /*消息正文*/
};
```

(2)消息头结构

对应每一个消息缓冲区都有一个消息头结构对其进行描述，其结构定义如下：　　：

```
struct   msg{
struct msg *msgnext;     /*指向消息队列中下一个消息的指针*/
1oug   msgtype;        /*消息类型，同消息缓冲区中的类型*/
short msgts;        /*消息正文的长度*/
short   msgspot;        /*消息正文的地址*/
};
```

(3)消息头结构表

消息头结构表是由若干个消息头结构构成的数组。系统初始化时，就已开辟好。消息头结构表的定义如下：

```
struct msg msgh[MSGTQL];
```

系统初启时，所有的消息头链成一个单向空闲链，msgfp 为链头指针。

(4)消息队列头结构

为了便于管理，同一类型的消息由消息头结构组成一个消息队列。对应每个消息队列，都有一个消息队列头结构，指出该队列的一些情况。消息队列头定义在 sys/smg.h 中，其结

构如下：

```
struct msgid_ds{
struct ipc_permmsgperm；     /*消息队列访问控制结构*/
struct msg*msghrst；        /*指向队列中的第一个消息*/
struct msg*msglast;       /*指向队列中的最后一个消息*/
ushort    msgcbyte;       /*队列中当前消息正文的总字节数*/
ushort    msgqnum；        /*队列中的消息个数*/
ushort    msgqbyte;       /*队列中允许容纳的最大字节数*/
ushort    msglspid;       /*最近一次发送消息的进程标识*/
ushort    msglrpid；       /*最近一次接收消息的进程标识*/
time_t    msgstime; /*最近一次发送消息的时间*/
time_t    msgstime；       /*最近一次接收消息的时间*/
time_tmsgrtimeme       /*最近修改时间*/
```

其中，struct ipc_perm 为消息队列的拥有者和访问方式，定义如下：

```
struct ipc_perm{
ushort    uid；         /*current userid*/
ushort    gid；         /*currentgroup id*/
ushort      cuid；        /*creator user id*/
ushort    cgid；        /*creatorgroup id*/
usshort    mode；        /*access mode*/
ulong       seq；        /*slot usage sequence number*/
ke_t    key；        /*IPC KEY*/
};
```

(5)消息队列头表

所毛的消息队列头构成一个数组，叫消息队列头表，其结构如下：

```
strut msgidds msgque[MSGMNl];
```

下面用图 9-1 给出消息队列头表与消息头结构及消息之间的逻辑关系，以帮助了解和认识它们。

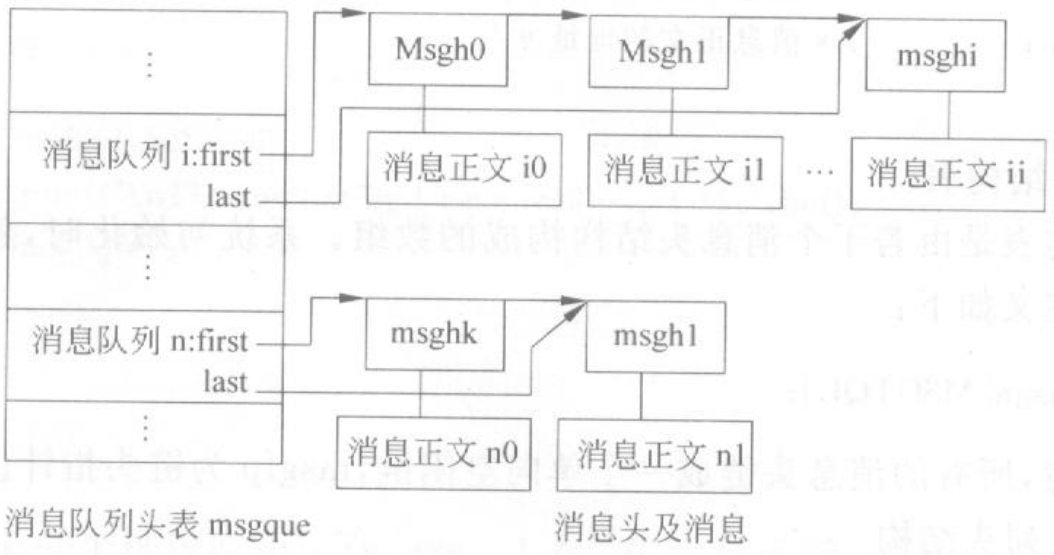


图 9-1 消息队列头表和消息头结构及消息之间的关系

(6)消息缓冲池

系统对消息正文的管理是将所有的消息正文存放在消息缓冲池中。消息缓冲池的基地址为 msg。系统初启时为其分配空间并初始化。消息头中的 msgspot 指出该消息在缓冲池中的位置。消息缓冲池的结构如下：

```
struct map{
stort msize;        /*缓冲区的大小*/
unsigned short maddr;        /*缓冲区的首地址*/
}msgmap[MSGMAP];
```

2).消息缓冲的系统调用

UNIX 系统为用户进程提供了若干条命令，包括建立一个消息队列，向消息队列发送消息，从消息队列接收消息，取或送消息队列的控制信息。

(1)建立一个消息队列

通信的双方在进行通信之前，要先申请创建一个消息队列，以便向队列中送或从队列中取消息；创建消息队列的调用语法为：

```
#include<sys/types.h>       .
#include<sys/ipc.h>
#include<sys/msg.he>
int msqid    msgget(key_t key，int msg flg);
```

其中，key 是进程要建立的消息队列的关键字，是通信双方约定的一个长整型数。

key 可以取下列两个值之一：

IPC PRIVATE：表示建立一个私有消息队列。

正整数：表示建立一个公共的消息队列。

msgflg 的低 9 位说明消息队列的主人、小组用户和其他用户对消息队列的访问权限。其他位指出消息队列的建立方式：

为 IPC_CREAT 时，若消息队列还没有建立，则创建。

为 IPC_EXCL 时，通常与 IPC_CREAT 同时使用，若消息队列还没有建立，则创建。若已经建立，则出错。IPC_EXCL 单独使用时无意义。

若 key 为 0，则创建一个消息队列；

若 key 值不为 0，只在消息队列头表中找是否有与 key 相同的消息队列。若找到，说明已有进程创建过消息队列，这时检查 msgf1g 标志，看该用户是否有权访问该消息队列，若有权访问，则返回该消息队列标识，若无权访问，创建失败，出错返回。

若没有找到与 key 相同的消息队列，且 msgflg 中含有 IPC_CREAT 标志，则创建一个消息队列，并返回该消息队列标识。

返回值：成功时，返回该消息队列标识，错误时，返回-1。

(2)向消息队列发送消息

一旦消息队列被建立，通信双方就可通过消息队列进行通信。向消息队列发送消息的调用语法为：

```
#include<sys/smg.h>
int msgsnd(int msqid，void    *msgp, size_t    msgsz，int msgflg);
```

其中，msqid 是消息队列标识符，msgp 是指向用户区要发送的消息的指针，msgsz 为要发送消息正文的字节个数，但不包括消息类型的长度。msgflg 为同步标志。当发送消息的某个条件不满足时，若 msgflg 中的 IPC_NOWAIT 标志未设置(即＝0)，发送进程阻塞等待，直到将消息挂人消息队列为止。若 IPC_NOWAIT 标志已设置，发送进程立即返回。

返回值：成功时，为实际发送的字节个数；错误时，为-1。

(3)接收消息

接收消息的调用语法为：

#include<sys/smg.h>

int msgrcv(int msqid，void msgp，size_t msgsz，long msgtyp，int msgflg)；

其中，msqid 是消息队列标识符。msgp 是用户接收消息的位置指针。msgsz 为要接收的消息正文长度。msgflg 为同步标志。如果 msgflg 中的 MSG_NOERROR 标志被设置，且消息队列中的消息长度大于要接收的消息长度，系统截取消息为要接收的长度，否则返回一个错误。当接收消息的某个条件不满足时，若 msgflg 中的 IPC_NOWAIT 标志未设置(即＝0)，接收进程阻塞等待，直到接收到消息为止。若 IPC_NOWAIT 标志已设置，发送进程立即返回。msgtyp 是要接收的消息类型。当 msgtyp 为 0 时，表示接收消息队列中的第一个消息；若大于 0，则接收消息队列中与 msgtyp 相同的第一个消息；若小于 0，则接收消息队列中类型值小于 msgtyp 绝对值且类型值最小的消息。返回值：成功时，接收消息的正文长度；错误时，为-1。

(4)取或送消息队列的控制信息

一个消息队列建立后，特权用户或被授权用户(如创建者或拥有者)允许对消息队列进行控制操作。这包括读和修改消息队列的状态信息，取或送消息队列控制信息和释放一个消息队列等。其调用语法为：

#include<sys/msg.h>

int msgctl(int msqid，int cmd，struct msgid_ds * bur)；

其中，msqid 是消息队列标识符，crud 是对消息队列使用的操作命令，buf 是用户空间中用于取或送消息队列控制信息的一个 msgid_ds 指针。cmd 操作命令可能的取值和含义：

若 cmd 为 IPC_STAT，读取消息队列的状态信息，检查用户是否允许访问，若允许，则复制 msqid 队列结构 msgid_ds 的内容到 buf 所指的用户空间，否则错误返回。

若 cmd 为 IPC_SET，修改队列状态信息，检查用户是否有权修改，若有，则将用户提供的 buf 所指的状态信息复制到 msgid_ds 结构中的 msgperm.uid，msgperm,gid，msgperm.mode 和 msgqbytes 4 个元素中。

若 cmd 为 IPC_RMID，释放消息队列，检查用户是否合法，若有权，则调用 msgfree()函数释放消息队列中的所有消息，并将消息正文字节总数清 0。若有进程等待发送或接收消息，唤醒它们。

3).利用消息缓冲机制的通信过程

为了理解进程利用消息缓冲的通信过程，下面给出两个例子。

例 9.2　创建一个私有消息队列，一个进程自己发送消息和接收消息。

创建一个消息队列，之后将一个消息“hell，world”放人消息队列。再从消息队列进行接收，并打印出所接收的消息。

```
/*file10.c 消息队列的源程序*/
#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#define    __USE_GNU
#include   <sys/msg.h>
int main(int argc，     char    *argv[])
{
int queue_id；
struct msgbuf'msg，*recv_msg；
int rc；
```

```
        int msgsz;
        queue_id＝msgget(IPC_PRIVATE，IPC_CREAT | 0600);
       if(queue_id＝=-1){
        perror("main's msgget");
        exit(1);
    printf("the created message's id＝'%d'.\n",queue id);
        //分配一个消息结构"hello，world"
        msg=(struct msgbuf*)malloc(sizeof(struct msgbuf)+strlen("hello，world"));
        msg->mtype＝1//消息队列的索引赋值为 1
        strcpy(msg->mtext，"hello，world")；//将字符串复制到消息体中
        //发送消息
        rc＝msgsnd(queue_id，msg，strlen(msg->mtext)+1，0);
        //这里的+1 是指字符串的结束符
    if (rc＝＝-1){
        perror("msgsnd");
        exit(1);
        }
        free(msg)；//释放消息占用的空间
        printf("message is placed onmessage，s queue.\n");        ：
        //接收消息
        reev_msg＝(struct  msgbuf*)malloc(sizeof(struct  msgbuf)+strlen("hellO，
word")+1);
        msgsz=strlen(recv_msg->mtext)+l;
        rc＝msgrcv(queue_id，reev_msg，msgsz，0，0);
       if (rc＝＝-1){
        perror("msgrcv");
        exit(1);
       }
    printf("received messagets mtypeis'％d'；mtext is，%s，\n"，recv_msg->mtype，
recv msg
       ->mtext);
       msgctl(queue_id，IPC_RMID，NULL)；//删除消息队列
       //free(recv_msg);
       return 0;
       }
       $./f10   /*程序运行的命令行*/
       /*gcc-o f10 file10.c*/
       the created message's id＝'983054'.
       message is placed On message's queue.
       received message's mtype is'1'；mtext is'hello，world'
```

例 9.3　创建一个公共消息队列，实现客户进程和服务者进程之间进行通信。

客户进程向服务者进程发送消息，请求服务。服务者进程接收消息，完成客户的服务请求后，再将服务结果以消息方式发送给客户。其具体实现描述为：

服务者进程用关键字 SVKEY 和标志 IPC_CREAT 调用 msgget()建立一个消息队列，得

到其队列标识符 msqid 之后，用 msqid 调用 msgrcv()接收类型为 REQ 的消息。客户进程用关建字 SVKEY 调用 msgget()得到消息队列标识符 msqid，之后用 msqid 调用 msgsnd()将自己的 pid 发送到消息队列(SVKEY)中，表示其所请求的服务。然后调用 msgrcv()等待服务结果消息的到来。服务者进程接收到请求服务的消息后进行服务工作，完成服务后向客户进程发回一条消息，消息的类型为客户的标识 pid，消息正文是服务者进程自己的标识 pid。

客户进程收到服务结果信息后，显示必要信息后，结束两者的通信过程。下面给出客户进程和服务者进程通信实例的源程序。

(1) 客户进程的通信过程

```
/*客户进程的通信程序 client.c*/
    #include <sys/types.h>
    #include <sys/ipc.h>
    #include <sys/msg.h>
    #define     SVKEY     75
    #denne    REQ     1
    struct msgform{
    longmtype;       /*消息类型*/
    charmtext[256]     /*消息正文*/
    }
    main()
    {   struct msgform msg;
    int msqid，pid，*pint;
    msqid＝msgget(SVKEY，0777);
    pid＝getpid();       /*获得当前进程的标识*/
    pint＝(int   )msg->mtext;       /*取消息正文的首地址*/
    pmt＝pid;       /*将客户进程的 pid 复制到消息缓冲区*/
    msg->mtype＝REQ;       ，
    msgsnd(msqid，&msg，sizeof(int)，0);
    msgrcv(msqld，&msg，256，pid，0);       /*这里的 pid 作为消息类型*/
    printf("client receive serverts serviceresuhis servert's:%d.\n，*pint);
    }
```

(2)服务者进程的通信过程

```
    /*服务者进程的通信程序为 server.c*/
      #include   <sys/types.h>
      #include   <sys/ipc.h>
      #include   <sys/msg.h>
      #denne    SVKEY     75
      #denne    REQ     1
      int msqid;
      main()
      {
      int i，pid，*pint;
      msqid＝msgget(SVKEY，0777|IPC_CREAT);
      for(;;)
      {msgrcv(msqld，&msg，256，REQ，0);
```

```
printf("server is doing the service fOr a client.\n");
pint＝(int*)msg・mtexti;
pid＝*pint；/*获得客户进程的 pid，以便进行服务*/
printf("serverreceive client's service request is client's pid：％d.\n"，pid)；
msg->mtype＝pid；
* pint＝getpid()；
/*将服务者服务结果用服务者的 pid 发送给客户*/
msgsnd(msqid，&msg，sizeof(int)，0)；
}
}
$./client     /*程序运行的命令行*/
client receive server's service result is server's pid：3771.
$./server     /*程序运行的命令行*/
server is doing the service for a client.
server receive client's service request is client's pid：3770.
```

2.　信号量机制

在 UNIX 系统 V 中，一个或多个信号量构成一个信号量集合。使用信号量机制用来实现进程之间的同步和互斥，允许并发进程一次对一组信号量进行相同或不同的操作。

每个 P、V 操作不限于减 1 或加 1，而是可以加减任何整数。在进程终止时，系统可根据需要自动消除所有被进程操作过的信号量的影响。

1).信号量机制中使用的数据结构

(1)信号量的数据结构

系统中的每个信号量都有一个数据结构定义在 sys/sem.h 中。结构定义如下：

```
struct sem{
    ushort    semval；        /*信号量的当前值*/
    short sempid       /*最近一次对该信号操作的进程标识*/
    ushort semncnt       /*等待信号量值增加的进程数*/。
    ushort semzcnt       /*等待信号量值等于 0 的进程数*/
    }
```

(2)信号量集合的数据结构

系统为每个信号量集合保持一个头结构定义在 sys/sem.h 中。结构定义如下：

```
struct semid_ds{
    struct    ipc_permsem_perm；     /*对信号量的访问方式，见消息缓冲*/
    time_t sem_otime；        /*最后操作信号量集合的时间*/
    time_t sem_ctime；        /*最后修改信号量集合的时间*/
    struct sem*sem_base；        /*指向集合中第一个信号量的指针*/
    ushort sem_nsems；        /*集合中的信号量个数*/
    }
```

2).信号量集合的建立

任何进程在使用信号量之前，通过调用如下的函数申请建立一个信号量集合：

```
#include    <sys/types.h>
#include    <sys/ipc.h>       .
#include    <sys/sem.h>
int semget(key_t key，int nsems，int semflg)
```

其中，key 为用户进程指定的信号量集合的关键字，nsems 为信号量集合中的信号量数，

semflg 为访问标志。

返回值：成功时，返回信号量集合的标识号，失败时，返回-1。

semflg 由以下成分组成：

IPC_CREAT，创建一个新的信号量集合。如果该标志没有设置，将查找与 key 相关的信号量集合。

IPC_EXCL 与 IPC_CREAT 一起使用，确保创建一个新的信号量集合。若该段已经存在，出错。低 9 位为三类用户的访问方式的定义。

3).对信号量的操作

通过调用 semop()函数，进程对一信号量集合中的一个或多个信号量执行 P/V 操作，其操作命令由用户提供的信号量操作模板(sembuf)定义，该模板的结构如下：

```
struct   sembuf{
ushort   sem_num;        /*信号量的序号*/
short    sem_op      /*具体执行的操作(即 P 或 V 操作)*/
short    sem_flg;     /*访问标志*/
};
```

调用语法为：

```
#include<sys/sem.h>
int semop(int semid，struct sembuf*sops，unsigned nsops);
```

其中，semid 是进程调用 semget 后返回的信号量集合的标识，sops 是用户提供的信号量操作模板数组(sembuf)的指针，nsops 为数组 sembuf 中的元素数(即一次需进行的操作数)。正常返回值为 0。

4).对信号量执行控制操作

当要读取和修改信号量集合的有关状态信息，或撤销信号量集合时，调用命令 semctl()对信号量执行控制操作。其调用语法为：

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

其中，semid 是信号量集合的标识，semnum 是要处理信号量集合中的信号量个数，cmd 为要执行的操作命令，arg 为控制操作需要的参数，是一个指向联合 semun 的指针。semun 定义为：

```
union semun{
int val；/*value for SETVAL*/
struct   semid_ds *buf/*buffer for IPC_SET&IPC_STAT*/
ushort   array[]；/*buffer for IPC_INFO*/
}arg;
```

cmd 的可能取值有：

GETPID	返回最近一个对信号量操作的进程标识
GETVAL	返回索引为 semnum 的信号量的值
GETALL	返回一个集合中所有信号量的值到 arg.array 中
GETNCNT	返回睡眠等待信号量值为正的进程的数
GETZCNT	返回睡眠等待信号量值为 0 的进程数
SETVAL	设置索引为 semnum 的信号量的值
SETALL	按照数组 arg.array，设置一个集合中所有信号量的值
IPC_STAT	若为读，可将指定 semid 的信号量头结构读人 arg.buf 中获得信号量的值

IPC_SET　　按照 arg.buf 设置用户标识、组标识和访问权限。若不是给定的用户标识，立即返回

IPC_RMID　删除指定标识的信号量集合

5).信号量的使用示例

例 9.3　用于进程互斥共享文件的信号量的使用。

```
/*互斥共享文件的例子 file11.c 源程序　*/
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<time.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<1inux/sero.h>
#define NUM PROCS 5
#define SEM ID 250
#define FILE_NAME　“/tmp/sem_MUTEX”
#define　DELAY　400000
//各子进程互斥写文件的通用函数
void　update_file(int sem_ set_id，char　*file_name_path，　int number)
{
  struct sembuf sem_op；
  FILE*file；
  //相当于执行 P 操作，申请写文件
  sem_op.sem_num=0；
  sem_op.sem_nop=-1;
  sem_op.sem_flg＝0；
  semop(sem_set_id，&sem_op，1)；
  //向文件写，写人的数据是进程的标识
  file＝fopen(file_name_path，"w")；
if (file){
  fprintf(file，"％d＼n"，number)；
   printf("％d＼n",number)；
  fclose(file)；
  }
  //相当于执行 V 操作，释放文件的使用权
  sem_op.sem_num=0；
  sem_op.sem_op=1；
  sem_op.sem_flg＝0；
  semop(sem set id，&sem_op，1)；
  }
  //子进程准备写文件的通用函数
  void do_child_loop(int sem_set_id，char *file_name)
  {
  pid_tpid＝getpid()；//得到本进程的标识
```

```
        inti，j;
        for(i＝0；i<3；i++){
        update_file(sem_set_id，file_name，pid)；
        for(j＝0;j<200000;j++)；/*暂停一段时间*/
        }
        }
    //主函数
        int main(int argc，char**argv)
        {
        int sem set id，ehild_pid；
        union semun sem_val
        int i，rc
        //创建一个信号量集合，标识为 225，只有一个信号量
        sem_set_id=semget(SEM_ID，1，IPC_CREAT|0600)；
        if(sem_set_id＝＝-1){
        perror("maint's semget error")；
        exit(1)；
        }
        //把该信号量的值设置为 1
        sem_val.val=1；
        rc＝semctl(sem_set_id，0，SETVAL，sem_val)；
if(rc＝＝-1){
        perror("main：setctt")；
        exit(1)；        }
        //建立一些子进程，以便竞争并互斥地向文件中写
        for(i＝0；i<NUM_PROCS；i++){
        child_pid＝fork()；
        switch(child_pid){
        case   -1：
        perror("fOrk()")；
        exit(1)；
        case    0：
        do_child_loop(sem_set_id，FILE_NAME)；
        exit(0)；
        default：break；
        )
    }//创建子进程的循环结束
        //父进程等待子进程结束
        for(i＝0；i<10；i++){
        int child_status；
              wait(&child_status)；
        }
        printf("main is done")；
        fflush(stdout)；
```

```
return  0;
}
$./f1n /*程序运行的命令行*/
17636
17636
17636
17637
17637
17637
17638
17638
17638
17639
17639
17639
17640
17640
17640
main is done
*/
```

四、实验步骤

1.熟悉相关的原理和 API 函数。

2. 利用消息队列实现发送消息和接收消息实现进程间的消息发送和接收;利用消息队列、信号量、信号实现客户进程和服务者进程之间进行通信程序。

3.对实现的结果进行分析。

程序清单 3

file10.c 利用消息队列实现发送消息和接收消息

client.c server.c 实现客户进程和服务者进程之间进行通信

file11.c 进程互斥共享文件的信号量

五、实验报告的要求：

实验结束后，应整理实验报告，其内容应包括：

1、实验题目

2、程序的设计思路

3、写出程序源代码

4、写出程序运行结果

5、对实验结果进行分析、总结并说明产生该结果的原因

六、思考题

(1). 分析进程之间是如何实现互斥共享文件的

(2). 编程序，实现一个服务器和多个客户之间通过消息队列进行通信,查看通信的效果如何。

(3). 例 9.3 的结果是如何产生的?

第十章

实验五 Linux 设备管理

一、实验目的

了解 linux 设备管理结构和命令。

理解 Linux 操作系统模块机制，学习编写内核模块。

了解一个简单驱动程序的编写。

二、实验原理

（一）linux 设备管理概述

Linux 的设备管理的主要任务是控制设备完成输入输出操作，所以又称输入输出（I/O）子系统。它的任务是把各种设备硬件的复杂物理特性的细节屏蔽起来，提供一个对各种不同设备使用统一方式进行操作的接口。Linux 把设备看作是特殊的文件，系统通过处理文件的接口—虚拟文件系统 VFS 来管理和控制各种设备。

Linux 的设备管理系统支持三类设备：块设备、字符设备和网络设备。

（1）块设备将数据按可寻址的块为单位进行处理。

（2）字符设备是以字符为单位进行数据传输，字符设备通常只容许顺序访问。

（3）网络设备则可通过 BSD 套接口访问数据。

字符设备指那些没有缓冲区，以字符流形式发送或接收数据的设备，比如键盘和鼠标。字符设备无需编址和寻址操作。而块设备以数据块为单位读写数据，在读写过程中，需要数据缓冲区支持。典型的块设备有硬盘，光盘等。网络设备在 Linux 系统中做专门处理，常用的有网卡，串行通信口等。Linux 系统中，将每个设备描述为主设备号和从设备号。主设备号和设备的驱动程序一一对应，从设备号用以区别同一类型的多个设备。通过主、从设备号，Linux 在设备驱动程序和设备特殊文件(device special file) 之间建立映射。

对设备的识别使用设备类型、主设备号、次设备号。

- 设备类型：字符设备或者块设备。
- 主设备号：按照设备使用的驱动程序不同而赋予设备不同的主设备号。主设备号是与驱动程序一一对应的。
- 次设备号：同时还使用次设备号来区分一种设备中的各个具体设备。次设备号用来区分使用同一个驱动程序的个体设备。

所有的设备文件均放在/dev/目录下，下图为一个 ls –l /dev/命令的截图。

```
brw-rw----   1 root     floppy     2,   0 Jul 18   1994 fd0
brw-rw----   1 root     floppy     2,   1 Jul 18   1994 fd1
brw-rw----   1 root     disk       3,   0 Jul 18   1994 hda
brw-rw----   1 root     disk       3,   1 Jul 18   1994 hda1
brw-rw----   1 root     disk       3,   2 Jul 18   1994 hda2
brw-rw----   1 root     disk       3,   3 Jul 18   1994 hda3
brw-rw----   1 root     disk       3,   4 Jul 18   1994 hda4
crw-rw----   1 root     daemon     6,   0 Jul 18   1994 lp0
crw-rw-rw-   1 root     sys        1,   3 Jul 18   1994 null
crw-rw-rw-   1 root     tty        5,   0 Jul 18   1994 tty
crw--w--w-   1 pc       book       4,   0 Jul 18   1994 tty0
crw--w--w-   1 pc       book       4,   1 Aug 30 15:16 tty1
```

其中 fd 为软盘（floppy disk），hd 为硬盘（hard disk）hda 为第一个硬盘，hda1 为第一个硬盘的第一个分区，tty 为终端。

小结：

- 从抽象的观点出发，Linux 的设备又称为设备文件。
- 设备文件也有文件名，设备文件名一般由两部分组成
- 第一部分 2～3 个字符，表示设备的种类，如串口设备是 cu，并口设备是 lp，IDE 普通硬盘是 hd，SCIS 硬盘是 sd，软盘是 fp 等。
- 第二部分通常是字母或数字，用于区分同种设备中的单个设备，如 hda、hdb、hdc…分别表示第一块、第二块、第三块 IED 硬盘。而 hda1、hda2…表示第一块硬盘中的第一、第二个磁盘分区。
- 设备文件一般置于/dev 目录下，如/dev/hda2、/dev/lp0 等。

1．Linux 设备管理层次结构

下图为一个硬件 I/O 的原理图：

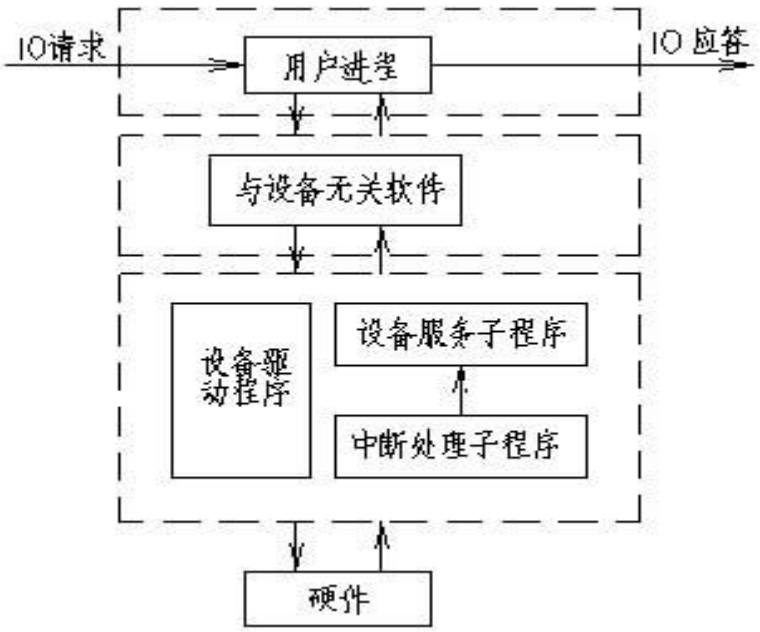


图 I/O 系统的层次结构

系统对设备的控制和操作是由设备驱动程序完成的。设备驱动程序是由设备服务子程

序和中断处理程序组成。设备服务子程序包括了对设备进行各种操作的代码，中断处理子程序处理设备中断。

2. 与设备无关的软件

在 Linux 系统中，设备均视为文件，与设备无关的软件的功能大部分是由文件系统去完成，这些软件主要功能如下：

(1) 负责把设备名与相应驱动程序对应起来，通过主设备号找到相应的设备驱动程序，通过从设备号确定具体的物理设备。

(2) 对设备提供保护机制，它像对文件保护那样采用通常的 rwx 权限机制。

(3) 屏蔽块设备间扇区大小的差异，向高层软件提供统一大小数据块。

(4) 采用缓冲技术，从而解决数据交换速度匹配问题。

(5) 负责独占设备的分配和释放。

3. 设备驱动程序

Linux 设备驱动程序是操作系统内核和机器硬件之间的接口。它驻留在内存，是内核的一部分。它的主要任务是从与设备无关软件中接收抽象的命令，在其控制的设备上完成指定操作。具体功能有：

①对设备进行初始化；

②使设备投入运行和退出服务；

③从设备接收数据并将它们送回内核；

④将数据从内核送到设备；

⑤检测和处理设备出现的错误；

linux 将设备视为文件，设备文件都是由 device_struct 数据结构来描述的，该结构定义于 fs/devices.c，其中 name 是某类设备的名字，fops 是指向文件

```
structdevice_struct{
const char *name;
struct file_operations *fops;
};
```

Linux 为每个驱动程序设有一个称为 file_operation 的数据结构，其中包含指向驱动程序内部大多数函数的指针，即为上表的*fops 指针。该操作表提供一个设备驱动程序的入口点，其数据结构定义如下：

```
#include <linux/fs.h>
struct file_operations {
int (*lseek)(struct inode *inode,struct file *filp, off_t off,int pos);
int (*read)(struct inode *inode,struct file *filp, char *buf, int count);
int (*write)(struct inode *inode,struct file *filp, char *buf,int count);
int (*readdir)(struct inode *inode,struct file *filp, struct dirent *dirent,int count);
int (*select)(struct inode *inode,struct file *filp, int sel_type,select_table *wait);
int (*ioctl) (struct inode *inode,struct file *filp, unsigned int cmd,unsigned int arg);
int (*mmap) (void);
int (*open) (struct inode *inode, struct file *filp);
void (*release) (struct inode *inode, struct file *filp);
int (*fsync) (struct inode *inode, struct file *filp);
};
```

其中定义的数据项都是函数指针，在初始化时需要将各个函数接口指定到自己编写的设备驱动的对应函数接口上。对设备的操作跟操作文件方式一样，如 open、read、write、close。

（二） Linux的I/O控制

Linux的I/O控制方式有三种：查询等待方式、中断方式和DMA(内存直接存取)方式.

1．查询等待方式

- 查询等待方式又称轮询方式（polling mode）。
- 对于不支持中断方式的机器只能采用这种方式来控制I/O过程，所以Linux中也配备了查询等待方式。
- 例如，并行接口的驱动程序中默认的控制方式就是查询等待方式。
- 如函数lp_char_polled()就是以查询等待方式向与并口连接的设备输出一个字符。

```
static inline int lp_char_polled(char lpchar, int minor)
{
    int status, wait = 0;
    unsigned long count    = 0;
    struct lp_stats *stats;
    do {                                    /*  查询等待循环  */
        status = LP_S(minor);
        count ++;
        if(need_resched)
            schedule();
    } while(!LP_READY(minor,status) && count < LP_CHAR(minor));
    if (count == LP_CHAR(minor)) { /*  超时退出  */
        return 0;
    }
    outb_p(lpchar, LP_B(minor)); /*  向设备输出字符  */
      .
      .
}
```

2．中断方式

- 在硬件支持中断的情况下，驱动程序可以使用中断方式控制I/O过程。
- 对I/O过程控制使用的中断是硬件中断，当某个设备需要服务时就向CPU发出一个中断脉冲信号，CPU接收到信号后根据中断请求号IRQ启动中断服务例程。
- 在中断方式中，Linux设备管理的一个重要任务就是在CPU接收到中断请求后，能够执行该设备驱动程序的中断服务例程。
- 为此，Linux设置了名字为irq_action的中断例程描述符表：

```
static struct irqaction *irq_action[NR_IRQS+1];
```

- NR_IRQS表示中断源的数目。
- irq_action［］是一个指向irqaction结构的指针数组，它指向的irqaction结构是各个设备中断服务例程的描述符。

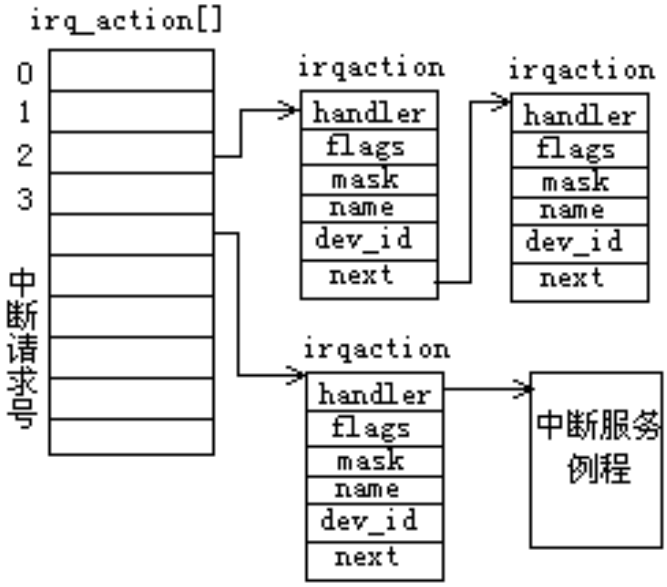
```
struct irqaction {
    void (*handler)(int, void *, struct pt_regs *);    /*  指向中断服务例程  */
    unsigned long flags;          /*  中断标志  */
```

```
    unsigned long mask; /*  中断掩码  */
    void *dev_id; /*
    struct irqaction *next; /*  指向下一个描述符  */
};
```

- 在驱动程序初始化时，调用函数request_irq（）建立该驱动程序的irqaction结构体，并把它登记到irq_action[]数组中。
- request_irq（）函数的原型如下：

```
int request_irq(unsigned int irq,
                void (*handler)(int, void *, struct pt_regs *),
                unsigned long irqflags,
                const char * devname,
                void *dev_id);
```

- 参数irq是设备中断求号，在向irq_action[]数组登记时，它做为数组的下标。
- 把中断号为irq的irqaction结构体的首地址写入irq_action[irq]。这样就把设备的中断请求号与该设备的服务例程联系在一起了。
- 当CPU接收到中断请求后，根据中断号就可以通过irq_action[]找到该设备的中断服务例程。



设备中断管理

（三） 字符设备与块设备管理

在 Linux 中，一个设备在使用之前必须向系统进行注册，设备注册是在设备初始化时完成的。

1．字符设备管理

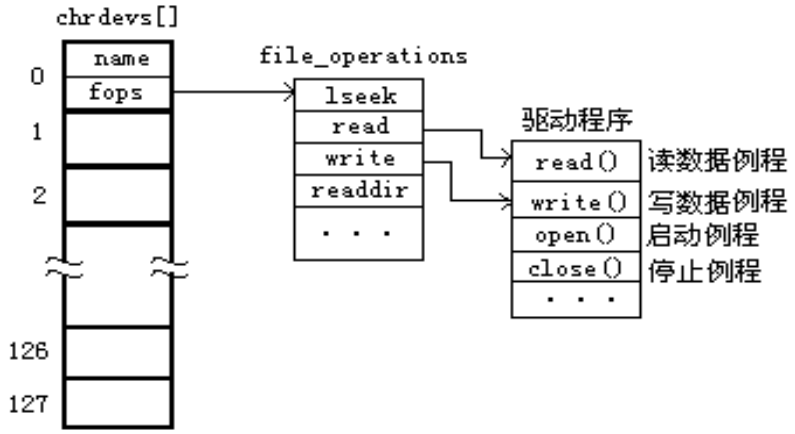
- 在系统内核保持着一张字符设备注册表，每种字符设备占用一个表项。
- 字符设备注册表是结构数组 chrdevs[]：

```
#define MAX_CHRDEV 128
static struct device_struct chrdevs[MAX_CHRDEV];
```

- 注册表的表项是 device_struct 结构：

```
struct device_struct {
    const char * name;     /*  指向设备名字符串  */
    struct file_operations * fops; /*  指向文件操作函数的指针  */
};
```

- 在字符设备注册表中，每个表项对应一种字符设备的驱动程序。所以字符设备注册表实质上是驱动程序的注册表。
- 使用同一个驱动程序的每种设备有一个唯一的主设备号。所以注册表的每个表项与一个主设备号对应。
- 在 Linux 中正是使用主设备号来对注册表数组进行索引，即 chrdevs[]数组的下标值就是主设备号。
- device_struct 结构中有指向 file_operations 结构的指针 f_ops。file_operations 结构中的函数指针指向设备驱动程序的服务例程。
- 在打开一个设备文件时，由主设备号就可以找到设备驱动程序。



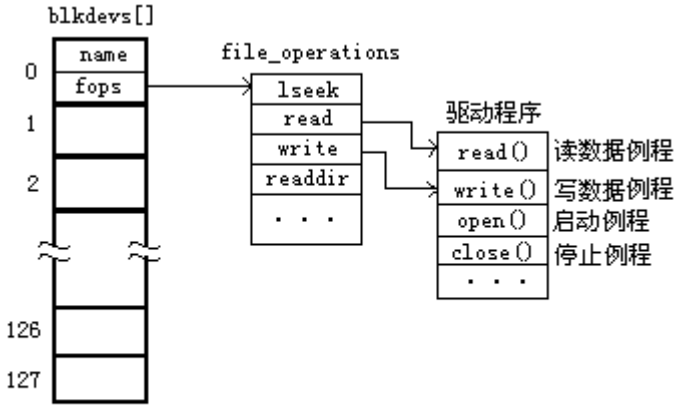
字符设备注册表

2．块设备管理

- 块设备在使用前也要向系统注册。
- 块设备注册在系统的块设备注册表中,块设备注册表是结构数组 blkdevs[]
- 它的元素也是 device_struct 结构

```
static struct device_struct blkdevs[MAX_BLKDEV]
```

- 在块设备注册表中，每个表项对应一种块设备，
- 注册表 blkdevs[]数组的的下标是主设备号。



块设备注册表

- 块设备是以块为单位传送数据的，设备与内存之间的数据传送必须经过缓冲。
- 当对设备读写时，首先把数据置于缓冲区内，应用程序需要的数据由系统在缓冲区内读写。
- 只有在缓冲区内已没有要读的数据，或缓冲区已满而无写入的空间时，才启动设备控制器进行设备与缓冲区之间的数据交换。
- 设备与缓冲区的数据交换是通过 blk_dev[]数组实现的：

  struct blk_dev_struct blk_dev[MAX_BLKDEV];
- 每个块设备对应数组中的一项，数组的下标值与主设备号对应。
- 数组元素是 blk_dev_struct 结构：

```
struct blk_dev_struct {
    void (*request_fn)(void);
            struct request * current_request;
            struct request      plug;
            struct tq_struct plug_tq;
    };
```

request_fn ：指向设备读写请求函数的指针

current_request：指向 request 结构的指针。

当缓冲区需要与设备进行数据交换时， 缓冲机制就在 blk_dev_struct 中加入一个 request 结构。每个 request 结构对应一个缓冲区对设备的读写请求。request 结构中有一个指向缓冲区信息的指针，由它决定缓冲区的位置和大小等。

（四） Linux 模块机制

驱动程序一般是内核的一部分，需要直接控制硬件。由于硬件多种多样，需要 linux 支持的驱动程序很多。但是由于 Linux 的单块结构（monolithic）使得其可扩展性较差，因此 linux 引入了模块机制，驱动程序以模块的形式装入内核，不需要重新编译内核。模块(module)实在内核空间运行的程序，实际上是一种目标对象文件，没有链接，不能独立运行，但是其代码可以在运行时链接到系统中作为内核的一部分运行或从内核中去下，从而可以动态扩展内核的功能。利用 linux 源码编译生成内核时，如某功能允许“m”选项（其他为“y”， “ n”），说明可以以模块形式存在，多数设备驱动程序以模块的方式挂接到内核，系统启动时已将若干模块挂入了内核，其他驱动程序在需要时，自动调入内核。因此用户只要有权限，就可以编写模块挂入内核。

三、实验内容与要求

1、 Linux 设备了解

① 观察/dev/目录， ls –l /dev/

② 安装光驱或 windows 系统下的可用驱动盘

Linux 下，有些设备需要手工安装，如光驱或 windows 系统下的可用驱动盘。首先观察已安装的驱动盘

```
mount     //可显示安装好的驱动盘
mount    /dev/cdrom    /mnt/cdrom //安装光驱
   或  mount /dev/hda2 /mnt/c    //安装系统的 C 盘
umount    /dev/cdrom    //卸除光驱
```

③ 观察磁盘信息

```
df //获取当前安装的文件系统
du //获取磁盘的使用信息
```

④ 到/proc 文件中观察设备的运行情况

```
cat /proc/dev/..
```

2、写一个空内核模块，编译测试。

```
#include<linux/kernel.h>
#include<linux/module.h>
int init_module(){
   printk("It is a null kernel module\n");
   return 0;
}
void cleanup_module(){
   printk("I am a kerenl module, now exiting ...\n");
}
```

编译：

```
gcc -c -Wall -D__KERNEL__ -DMODULE testmodule.c
```

运行：

```
insmod testmodule.o     //装入模块
lsmod                   //检查装入是否成功
rmmod testmodule        //卸载模块
lsmod                   //检查卸载是否成功
```

3、从内核源代码中，找出一个设备驱动程序源代码，简单分析其功能结构。

四、思考题

1、 串口、并口的设备文件名？

2、 可否写出软驱，如何卸除？

3、 printk 与 printf 的区别？

4、 在编译模块的过程中，不加入宏“__KERNEL__”和“MODULE”，行不行？

附录

A. Linux 常用函数

A.1 进程管理函数

●pid_t fork()

功能：创建一个新进程，运行与当前进程相同的程序。

参数表：空。

返回值：若失败，返回值为一 l，否则，父进程返回子进程的 pid 号，子进程返回 0。

头文件：<sys/types.h>

●pid_t waitpid(pid_tpid，int*status,intoptions)

功能：使得父进程的执行得以保持，直到子进程退出(即使父进程先处理完，也要等到子进程结束)。子进程退出时，父进程收集子进程的信息，并继续执行直到退出。

参数表：

pid：子进程的 id 号。

status：返回参数，保留子进程结束时的状态信息。

options：进程等待选项，可以为 WCONTINUED、WNOHANG、WNOWAIT 或者 WNUTRACED。

返回值：如果成功，返回被等待的子进程的 pid，否则，若指定了 WNOHANG 标志，并

且子进程状态无效，则返回 0，其余的失败将返回—1。

头文件：<sys/types.h><sys/wait.h>

●exec 系列函数

完成对其他程序的调用。用户在调用 exec 函数时，进程的当前映像将被替换成新的程序。换言之，如果成功调用了一个 exec 函数，函数的调用不会返回——而新的进程将覆盖原进程所占用的内存空间。通常用子进程调用运行 exec 函数，父进程等待直到子进程结束。

exec 系列函数有 execl、execlp、execle、execv 等。

A.2 文件管理函数

●int stat(char*path,struct stat sbuf)以及 int lstat(Char*path，struct stat sbuf)

功能：这两个函数用这些信息来填充一个类型为 struct stat 的结构。这两个函数的区别在于 lstat 不对符号链接进行追踪，而只是返回链接本身的信息；而 stat 对符号链接进行追踪，直到链按的最末端。

参数表：

path：需要获得信息的文件路径。

sbuf：返回的文件信息(若调用成功)。

返回值：如果该函数被正确执行，则返回 o；否则返回一 1。

头文件：<sys/stat.h>

●int open(Char*path，int flags,mode_tmode)

功能：打开一个文件，并返回文件的文件描述符。

参数表：

path：要打开的文件名。

flags：打开文件的方式。可用的值有 O_CREAT、O_RDONLY、O_WRONLY、O_RDWR、O_NONBLOCK、O_APPEND、O_TRUNC、O_EXECL、O_SHLOCK 以及 O_EXLOC 或者它们之间合适的组合。

mode：打开文件的权限，请参考附录 C 中的 chined 命令一节。

返回值：如果成功地打开参数 path 指定的文件.则返回该文件的文件描述符，否则，返回-l。

头文件：<fcntl.h>

●int read(int fd，void *buf，int count)

功能：从指定文件中读取特定长度的内容至某缓冲区。但要注意该文件必须具有读权限。

参数表：

fd：指定文件的文件描述符。

buf：目标缓冲区。

count：需要读入的数目(以字节记)。

返回值：实际读入的字节数。

头文件：<fcntl.h>

●int write(int fd,void*buf,int tcount)

功能：将指定的数据以指定的大小写入指定的文件。同样要求该文件必须具有写权限。

参数表：

d：指定文件的文件描述符。

buf：源缓冲区。

count：需要写入的数目(以字节记)。

返回值：实际写入的字节数。

头文件：<fcntl,h>

●int close(int fd)

功能：关闭指定的文件。

参数表：

fd：指定文件的文件描述符。

返回值：如果成功地关闭文件，则返回 0：否则，返回-1。

头文件：<fcntl.h>

●long lseek(int fd,int count,int flags)

功能：将指定文件的读写指针移动特定的偏移量。

参数表：

fd：目标文件的文件描述符。

count：偏移量的大小。

flags：文件指针移动选项，可选值有：SEEK_SET、SEEK_CUR、SEEK_END，分别表示文件的开始位置、文件指针的当前位置以及文件的结束位置。

返回值：文件指针的新位置。

头文件：<fcntl.h><io.h>

A.3 进程间通信

●int pipe(int fd[2])

功能：创建一个管道和两个文件描述符。

参数表：

fd：管道文件的描述符数组。其中 fd[0]文件描述符将用于读操作，而 fd[1]文件描述符

将用于写操作。

返回值：成功返回值是 0，如果创建失败则将返回-1。

头文件：标准头文件组。

A.4　多线程库

●int　pthread_create(pthread t*thread，const pthread attr_t*attr，void* (*routines)(void*)，void*arg)

功能：创建以函数 routines 为线程体，以 arg 为参数，具有 attr 线程属性的线程。

参数表：

thread：返回参数，新线程的句柄。

attr：新生成线程的属性，如果值为 NULL，则具有默认的线程属性设置。

routines：线程指定运行的函数，该函数必须具有 void*返回值。

arg：该线程运行函数的参数。

返回值：如果成功地创建该线程，则函数返回 0，否则，返回一个非 0 的错误码。

头文件：<pthread.h>

●int pthread join(pthread_t thread,void**status)

功能：等待一个线程结束，并将结束时的状态写入 status。

参数表：

thread：被等待的线程。

status：输出参数，线程结束时的状态。

返回值：如果线程成功结束，则返回 0，否则，返回非 0 的错误码。

头文件：<pthread.h>

●int pthread_mutex_lock(pthread_mutex_t　*mutex)

功能：如果信号量 mutex 未加锁，则为其加锁，否则，该线程阻塞直到 mutex 解锁。

参数表：

mutex：需要加锁的信号量。

返回值：如果成功加锁，则返回值为 0，否则，返回非 0 的错误码。

头文件：<pthread.h>

●int pthread_mutex_unlock(pthread_t*mutex)

功能：为指定的信号量解锁。

参数表：

mutex：需要加锁的信号量。

返回值：如果成功解锁，则返回值为 0，否则，返回非 0 的错误码。

B. Linux 常用内核函数

B.1 驱动程序入口和入口点

●module_init(x)

功能：驱动程序初始化的入口点。

参数表：x 为启动时或加载模块时要运行的函数。

描述：如果在操作系统启动时就确认把这个驱动程序加载内核或以静态形成链接，则 module init 将其初始化例程加入到“__initcall.int''代码段，否则将用 init_module 封装其初始化例程，以便该驱动程序作为模块来使用。

●module_exit(x)

功能：驱动程序退出的出口点。

参数表：x 为驱动程序被卸载时要运行的函数。

描述：当驱动程序是一个模块，用 rmmod 卸载一个模块时 module_exlt()将用 cleanup_moduleO 封装清除代码。如果驱动程序是静态地链接进内核，则 module_exit 函数不起任何作用。

B.2.原子和指针操作

●atomic read(v)

功能：读取原子变量。

参数表：v 为指向 atomic_t 类型的指针。

描述：原子地读取 v 的值。注意要保证 atomic 的有用范围只有 24 位。

●atomicc set(v，i)

功能：设置原子变量。

参数表：v 为指向 atomic—I 类型的指针，i 为待设置的值。

描述：原手地把 v 的值设置为 i。注意要保证 atomic 的有用范围只有 24 位。

●void atomic_add (inti，atomic_t*v)

功能：把整数增加到原子变量。

参数表：i 为要增加的值，v 为指向 atomic_t 类型的指针。

描述：原子地把 i 增加到 v。

注意要保证 atomic 的有用范围只有 24 位。

●void atomic_sub(inti，atomic_t *v)

功能：减原子变量的值。

参数表：伪要减取的值，v 为指向 atomlc_t 类型的指针。

描述：原手地从 V 减取 i。注意要保证 atomic 的有用范围只有 24 位。

B.3 内存操作

●void kmalloc(size_t size，int flags)

功能：分配内存。

参数表：size 为所请求内存的字节数，flags 为要分配的内存类型。

描述：kmalloc 是在内核中分配内存常用的一个函数。flags 参数的取值如下：

GFP_USER 代表用户分配内存，可以睡眠，GFP KERNEL 分配内核中的内存，可以睡眠；

GFP_ATOMIC 表示分配但不睡眠，在中断处理程序内部使用。

另外，设置 GFP—DMA 标志表示所分配的内存必须适合 DMA，例如在 i386 平台上，就意味着必须从低 16MB 的地址空间分配内存。

●void kfree(const void *objp)

功能：释放以前分配的内存。

参数表：objp 为由 kmalloc()返回的指针。

B.4. 双向链表操作

●void list_add(struct list_head new，struct list_head head)

功能：增加一个新元素。

参数表：new 为要增加的新元素，head 为增加以后的链表头。

描述：在指定的头元素后插入一个新元素，用于栈的操作。

●void list_add_tail(struct list_head*new，struct list_head*head)

功能：增加一个新元素。

参数表：new 为要增加的新元素，head 为增加以前的链表头。

描述：在指定的头元素之前插入一个新元素，用于队列的操作。

●void list del{struct list_head*entry}

功能：从链表中删除一个元素。

参数表：entry 为要从链表中删除的元素。

●void list_del_init(struct list_head*entry)

功能：从链表删除一个元素，并重新初始化链表。

参数表：entry 为要从链表中删除的元素。

●int list_empty(Structlist head*head)

功能：测试一个链表是否为空。

参数表：head 为要测试的链表。

●void list_splice(Structlist_head*list，structlist_head，head)

功能：把两个链表合并在一起。

参数表：list 为新加入的链表，head 为第一个链表。

●宏 list_entry(ptr，type，member)

功能：获得链表中元素的结构。

参数表：pos 为指向 list_head 的指针，type 为一个结构体，而 member 为结构 type 中的一个域，其类型为 list_head。

●宏 list_for_each(pos，head)

功能：扫描链表。

参数表：pos 为指向 list_head 的指针，用于循环计数，head 为链表头。

B.5 类似 C 标准库函数

当编写驱动程序时，一般情况下不能使用 C 标准库的函数。Linux 内核也提供了与 C 标准库函数功能相同的一些函数，但二者还是稍有差别。

下面列出一些常用的类似 c 标准库函数，其原型和含义请读者自行查询 c 标准库的有关文档。

●接收数据到字符串中的函数

int sprintf(char*buf， const char*fmt， ....)

int vsprintf(char*buf， const char*fmt，va_list args)

int printk(. . .)

●内存操作函数

void* memset (void *s, char c, size_t count)

void* memcpy(void*dest，const void* src， size_t count)

char* bcopy(const char* src，char* dest， int count)

void* memmove(void *dest，const void* src，size_t count)
int memcmp(const void *cs，const void* ct，size_t count)
void* memecan(void *addr，unsigned char c，size_t size)
●字符串相关函数
char* strcpy(char* dest，const char* src)
char* strncpy(char dest，const char* src，size_t count)
char* strcat(char*dest，　const char* src)
char* strncat(char *dest, const char* src, size_t count)
int strcmp(const char*cs，const char*ct)
int strncmp(const char*cs，const char*ct，size_t count)
char* strchr(const char*s, char c)
size_t strlen(const char* s)
slze_t strlen(const char* s，size_t count)
slze_t strspn(consC char*s，const char* accept)
char* strpbrk(const char* cs，const char* ct)
char* strtok(char*s，const char*ct)
unsigned long　simple_strtoul(const char*cp，char**endp，unsigned int base)
●访问用户内存的函数
get_user_byte(addr)
put_user_byte(x，addr)
get user word(addr)
put_user_word(x，addr)
get user_long(addr)
put_user_long(x，　addr)
●检测超级用户权限
suser()
fsuser()
●注册设备驱动的函数
int register_chrdev(unsigned int major，　const　char*name，struct file_o perations*fops)
int unreglster_chrdev(unsigned int major，const char*name)
int register blkdev(unsigned int major，　const char*name，struct file_o perations*fops)
int unregister_blkdev(unsigned int major，const char*name)