何文德 邓旭东编

《操作系统原理》

实验指导书

长 沙 学 院 计 算 机 工 程 与 数 学 学 院

2018 年 2 月

前 言

《操作系统原理》课程是物联网工程专业的一门专业必修课程，操作系统原理实验是操作系统课程的重要组成部分，是理论性与实践性并重的课程。

操作系统是计算机中的核心系统软件，用于控制和管理计算机系统中的各种软件和硬件资源，使之得到有效利用；合理组织计算机工作流程，以改善系统性能；向用户提供易于使用的接口。操作系统的性能直接影响到计算机系统的工作效率，所以操作系统一直以来都是计算机领域中最活跃的分支之一。

虽然操作系统的基本概念、原理和算法对初学者比较抽象，但这些概念、原理和算法都可以在实践中实现，学习操作系统应注重理论与实践现结合。本实验指导为了做到理论教学与实践教学彼此呼应、原理讲授与实践环节紧密结合，掌握基本理论与提高编程能力相互并重，共安排了 5 个 Linux 实验项目。这些实验项目分为操作系统算法模拟、系统编程、系统分析与设计三个层次，训练的知识点有 Linu Shell 程序、进程的并发性与进程控制、信号量与 P/V 操作、请求分页虚拟存储管理、设备驱动、虚拟设备、文件系统设计、文件数据恢复等。

编者

2018 年 2 月

目录

第一部分 实验内容

1 用户接口

1.1 Linux 基本操作

1. 实验目的

(1) 了解 Linux 文件和目录结构。
(2) 掌握 Linux 常用命令操作。

2. 实验类型：实训

3. 实验学时：2

4. 实验原理和知识点

(1) 实验原理：操作系统给用户提供了易于使用的界面，Linux 也不例外。Linux 向用户提供的使用界面有命令行界面(CLI)、图形用户界面(GUI)。在命令行界面下，用户可以向系统提交命令，系统的命令解释器(或 shell)解释执行用户提交的命令，并将结果返回(通常是显示)给用户。在图形用户界面下，用户通过鼠标点击图形对象(如窗口、菜单、图符、按钮等等)向系统发出命令，系统执行该命令并给出结果。在命令行界面下操作 Linux 是本次实验的重点。
(2) 知识点：CLI、GUI、命令解释程序、Linux 目录结构、Linux 文件属性、Linux 常用命令的使用、使用 vi 编辑器编辑文本文件。

5. 实验环境（硬件环境、软件环境）：

(1) 硬件环境：Intel Pentium III 以上 CPU，128MB 以上内存，2GB 以上硬盘
(2) 软件环境：Ubuntu 13.04 操作系统。
注：Ubuntu 是 Linux 的一个免费发行的流行版本，系英国"科能软件股份有限公司(Canonical)"开发

6. 实验内容及步骤：

(1) linux 启动、登录、注销、关机。
(2) 认识 Linux 文件和目录结构。
(3) 常用命令的操作。
(4) vi 编辑器操作。

7. 思考与练习

(1) 怎样挂装和卸载一个设备（比如优盘）？
(2) 查阅资料，了解怎样修改启动配置文件 /boot/grub/grub.conf。
(3) /proc 里面存放的是什么？
(4) 怎样把程序执行的结果输出到一个文件里？

1.2 shell 程序设计

1. 实验目的

(1) 了解 shell 的编程特点，掌握 shell 程序设计的基础知识。
(2) 对 shell 程序流程控制、shell 程序的运行方式有进一步的认识和理解。
(3) 通过本实验让学生应能够基本掌握编写 shell 程序的步骤、方法和技巧。

2. 实验类型：实训

3. 实验学时：2

4. 实验原理和知识点

(1) 实验原理：
shell 是一个命令解释程序，目前几大主流 shell 类型有：Bourne shell, C shell,K shell,分别简写为 bsh, csh, ksh。用 shell 语言写出来的程序，不需要通过编译即可执行。 shell 提供了你与操作系统之间交互的环境。
(2) 知识点：
a. 可在 shell 程序中使用 linux 下常用的命令，如 ls, cat, read, cp, ps, who, mkdir 等。
b. Shell 的程序结构有：①条件测试语句 if；②循环结构 while；③分支结构 case；④函数调用：例如 func()；⑤变量定义：例如 VAR=5。

5. 实验环境（硬件环境、软件环境）：

(1)硬件环境：Intel Pentium III 以上 CPU，128MB 以上内存，2GB 以上硬盘
(2)软件环境：Red Hat Linux 9.0 以上版本操作系统。

6. 实验内容及步骤：

(1) 编写一段 shell 程序，实现文件的备份和恢复。
(2) 编写一段 shell 程序，使用一个菜单界面，方便在 Linux 下对 U 盘的加载、卸载过程。程序要求实现以下 5 个功能：①加载 U 盘；②卸载 U 盘；③查看加载后的 U 盘信息；④从 Linux 分区的硬盘中拷贝文件到 U 盘中；⑤从 U 盘拷贝文件到 Linux 分区的硬盘指定位置上。
(3) 编写一段 shell 程序，主要功能是用来存储和查询学生成绩，并提供菜单显示选项；同时可以根据用户输入的选项执行查询、添加等功能；列举/usr/include 中的文件中的详细信息，并且将文件类型及权限写入/root/filename.txt 的文件中。

7. 思考与练习

(1) 如何取得指定目录下（如/lib 目录）的文件大小，并且将文件大于 5000 个数据块的文件的详细信息写入指定的文件中（如/root/largefile.txt）。
(2) 试比较 shell 程序和 C 语言的差别。

2 进程管理

2.1 进程创建与控制

1.实验目的

(1) 加深对进程概念的理解,理解进程和程序的区别。
(2) 认识并发进程的实质。分析进程争用资源的现象，学习解决进程互斥的方法。
(3) 认识并发进程的软中断通信。掌握使用软中断控制进程的编程技术。

2.实验类型： 实训。

3.实验学时：2

4.实验原理和知识点

(1) 实验原理：程序的并发执行具有随机性和不可再现性。程序并发执行会导致资源共享和资源竞争，各程序向前执行的速度会受资源共享的制约。程序的动态执行过程用进程这个概念来描述。由于向前推进的速度不可预知，所以多个进程并发地重复执行，整体上得到的结果可能不同。但要注意，就其中某单个进程而言，其多次运行结果是确定的。
。

5.实验环境（硬件环境、软件环境）:

(1) 硬件环境：Intel Pentium III 以上 CPU，128MB 以上内存，2GB 以上硬盘
(2) 软件环境：Ubuntu 13.04 操作系统。

6.实验内容及步骤:

(1)使用系统调用 fork()创建两个子进程，观察父子进程的运行，通过运行结果的分析，认识并发进程的实质。

(2)进程间通过系统调用 signal()进行通信，通过系统调用 wait()进行同步。通过运行结果的分析，认识并发进程的软中断通信。

(3)记录实验结果。

7.思考与练习

父子进程同步实验：编写一段程序， 使用系统调用 fork()创建一个子进程，使用系统调用 wait()让父进程等待子进程结束。

2.2 进程间通信

1.实验目的

(1) 掌握管道通信原理。

(2) 通过编写 Linux 消息发送和接收程序，了解和熟悉 Linux 消息通信机制

(3) 通过编写共享存储区的通信程序，理解 Linux 共享存储区机制。

2.实验类型：实训

3.实验学时：2

4.实验原理和知识点

(1) 实验原理：

并发运行的进程之间，可以通过信号进行同步，也可以通过管道、消息通信机制、共享存储机制进行通信。

(2) 知识点：

管道、管道通信；消息队列、消息缓冲、消息的创建、发送和接收；共享存储的创建、附接和断接。

5.实验环境（硬件环境、软件环境）：

(1) 硬件环境：Intel Pentium III 以上 CPU，128MB 以上内存，2GB 以上硬盘

(2) 软件环境：Ubuntu 13.04 操作系统。

6.实验内容及步骤：

(1)使用系统调用 pipe()建立一条管道，进程间通过管道交换信息。

(2)使用系统调用 msgget()、msgsnd()、msgrcv()、msgctl()编写消息发送和接收程序。

(3)使用系统调用 shmget()、shmat()、shmdt()、shmctl()编写共享存储区的通信程序。

(4)记录实验结果。

7.思考与练习

(1) 试比较消息通信和共享存储通信的优劣。

() 采用消息通信机制，你能写一个客户/服务器通信程序吗？

2.3 调度算法

1.实验目的

(1) 了解进程调度的几种算法，并比较各种调度算法的性能。

2.实验类型：实训。

3.实验学时：4

4.实验原理和知识点

(1) 实验原理：操作系统的调度程序担负两项任务：调度和分派。前者实现调度策略，确定就绪进程/线程竞争使用处理器的次序的裁决原则，即进程/线程何时应放弃 CPU 和选择哪个进程/线程来执行；后者执实现调度机制，确定如何时分复用 CPU，处理上下文细节，完成进程/线程同 CPU 的绑定及放弃的实际工作。

(2) 知识点:先来先服务算法(FIFO)、最短作业调度算法(SJF)、最短剩余时间优先算法(SRTF)、响应比最高者优先算法(HRRF)、优先级调度算法(静态优先级和动态优先级调度算法)、轮转调度算法(RR)、多级反馈队列调度算法(MLFQ)、彩票调度算法。

5.实验环境（硬件环境、软件环境）：

(1) 硬件环境：Intel Pentium III 以上 CPU，128MB 以上内存，2GB 以上硬盘

(2) 软件环境：Ubuntu 13.04 操作系统。

6.实验内容及步骤：

(1) 熟悉几种调度算法；

(2) 根据演示的程序结果，分析程序的采用的是哪种调度算法；

(3) 编写 FIFO，SJF，SRTF，动态优先级调度算法的程序，并计算其周转时间，平均作业周转时间，平均带权作业周转时间。

7.思考与练习

(1) 试编写多级反馈队列调度算法调度程序。

2.4 P、V 原语应用程序

1. 实验目的

(1) 掌握信号量的原理及 P、V 操作。

(2) 了解生产者与消费者问题，并通过信号量解决单缓冲区生产者－消费者问题和 m 个生产者和 n 个消费者共享 k 件产品缓冲区的问题。

2. 实验类型：实训。

3. 实验学时：4

4. 实验原理和知识点

(1) 实验原理：在操作系统中，用信号量表示物理资源的实体，是一个与队列有关的整型变量。信号量仅存在赋值、P、V 三种操作，且 P、V 操作为不可分割的原子性操作。可通过信号量实现进程的互斥、进程的同步。

5. 实验环境（硬件环境、软件环境）：

(1) 硬件环境：Intel Pentium III 以上 CPU，128MB 以上内存，2GB 以上硬盘

(2) 软件环境：Ubuntu 13.04 操作系统。

6. 实验内容及步骤：

(1) 熟悉信号量的操作，及如何实现进程的互斥与同步。

(2) 了解生产者与消费者问题。

(3) 掌握解决生产者与消费者问题的伪代码。

(4) 查看程序演示结果，并分析各个过程及现象。

7. 思考与练习

(1) 试分析生产者与消费者问题中的 P、V 操作代表的含义，以及 m 个生产者 n 个消费者共享 k 件产品缓冲区问题中的锁的释放与缓冲区资源的释放操作是否能互换。

(2) 利用信号量及共享内存，编写单缓冲区生产者与消费者程序及 m 个生产者 n 个消费者共享 k 件产品的缓冲区的程序。

3 存储管理

3.1 页面置换算法

1.实验目的

通过请求页式存储管理中页面置换算法的模拟设计，培养存储管理相关算法的实现能力。

2.实验类型：实训。

3.实验学时：4

4.实验原理和知识点

(1) 实验原理：请求页式存储管理原理。

(2) 知识点：页面置换、缺页中断、缺页率、命中率、FIFO 算法、Belady 现象、OPT 算法、LRU 算法、CLOCK 算法。

5.实验环境（硬件环境、软件环境）：

(1) 硬件环境：Intel Pentium III 以上 CPU，128MB 以上内存，2GB 以上硬盘

(2) 软件环境：Ubuntu 13.04 操作系统。

6.实验内容及步骤：

使用 FIFO 算法、OPT 算法、LRU、CLOCK 算法之一设计程序,模拟请求页式虚拟存储管理中页面置换过程，计算访问命中率。

(1) 学习预备知识。

(2) 编写程序。

(3) 编辑、编译、运行和调试程序。

(4) 记录程序运行结果。

(5) 分析结果并总结。

7.思考与练习

(1) 根据实验结果，说明 FIFO、OPT、LRU、CLOCK 算法那种算法最优？

(2) 采用队列法实现 LRU 算法时，怎样减少队列元素的移动或元素间交换次数？

4 设备管理

4.1 设备驱动程序

1.实验目的

通过 Linux 字符设备驱动程序的探索，熟悉 Linux 驱动程序框架和实现方法，培养 Linux 驱动程序设计能力。

2.实验类型：实训。

3.实验学时：4

4.实验原理和知识点

(1) 实验原理：Linux 驱动程序工作原理

(2) 知识点：字符设备、主设备号、次设备号、idode 结点、驱动程序框架、模块的安装与卸载、设备文件的创建与删除。

5.实验环境（硬件环境、软件环境）：

(1) 硬件环境：Intel Pentium III 以上 CPU，128MB 以上内存，2GB 以上硬盘

(2) 软件环境：Ubuntu 13.04 操作系统。

6.实验内容及步骤：

实现一个基于主存的字符设备驱动程序，在应用程序中通过该驱动程序读写内核的内存。实实现一个用户应用程序,该程序可以向上述字符设备驱动程序读或写信息，从而实现多个用户之间的聊天功能。

(1) 学习预备知识。

(2) 编写程序。

(3) 编辑、编译、运行和调试程序。

(4) 记录程序运行结果。

(5) 分析结果并总结。

7.思考与练习

(1) 根据实验结果，说明字符设备驱动程序需实现哪些文件操作函数？

(2) 驱动程序模块怎样动态添加、怎样动态卸载？

4.2 SPOOLing 模拟程序

1. 实验目的

理解和掌握 SPOOLing 技术。

2. 实验类型：实训。

3. 实验学时：4

4. 实验原理和知识点

(1) 实验原理：SPOOLing 是用一类物理设备模拟另一类物理设备的技术，是使独占型设备变成共享设备的一种技术。

(2) 知识点：预输入程序，缓输出程序，井管理程序，输入井，输出井。

5. 实验环境（硬件环境、软件环境）：

(1) 硬件环境：Intel Pentium III 以上 CPU，128MB 以上内存，2GB 以上硬盘

(2) 软件环境：Ubuntu 13.04 操作系统。

6. 实验内容及步骤：

(1) 理解 SPOOLing 技术的设计与实现。

(2) 画出 SPOOLing 技术实现的流程图(分为输入和输出两部分)。

(3) 编写程序模拟实现 SPOOLing。

7. 思考与练习

(1) 预输入程序、缓输出程序、井管理程序在实现 SPOOLing 技术时，分别对哪些表进行了处理，这些表的用途是什么？

5 文件系统

5.1 模拟文件系统设计

1. 实验目的

通过一个简单多用户文件系统的设计，加深理解文件系统的内部功能和内部实现。

2. 实验类型：实训。

3. 实验学时：2

4. 实验原理和知识点

(1) 实验原理：Linux 文件管理原理

(2) 知识点：文件系统数据结构、文件目录、文件操作、文件存储空间管理

5. 实验环境（硬件环境、软件环境）：

(1) 硬件环境：Intel Pentium III 以上 CPU，128MB 以上内存，2GB 以上硬盘

(2) 软件环境：Ubuntu 13.04 操作系统。

6. 实验内容及步骤：

实验内容：

设计一个二级文件系统，要求实现登录、注销、退出文件系统、目录操作（创建目录、改变目录）、文件操作（创建文件、打开文件、读文件、写文件、关闭文件、删除文件）等功能。（可以不考虑文件共享、文件系统安全、管道文件与设备文件等特殊内容）。

实验步骤：

(1) 学习预备知识。

(2) 编写程序。

(3) 编辑、编译、运行和调试程序。

(4) 记录程序运行结果。

(5) 分析结果并总结。

7. 思考与练习

(1) 读写文件之前，为什么要先打开文件？打开一个文件要做哪些工作？

(2) 许多现代操作系统把文件的描述信息放在索引结构里，这样做的好处是什么？

(3) 你设计的文件系统最大能存取多大的文件？若要存取一个文件长度为 2GB 的文件，你的文件系统要做哪些改动？

5.2 文件删除与恢复

1.实验目的

通过文件删除的分析，培养文件删除与恢复程序的设计能力。掌握分区结构的获取方法、FDT 和 FAT 结构的解析方法、文件簇链恢复方法。

2.实验类型：实训。

3.实验学时：2

4.实验原理和知识点

(1) 实验原理：FAT32 文件系统原理。

(2) 知识点： FAT32、BPB、FDT、FAT。

5.实验环境（硬件环境、软件环境）：

(1) 硬件环境：Intel Pentium III 以上 CPU，128MB 以上内存，2GB 以上硬盘

(2) 软件环境：Ubuntu 13.04 操作系统。

6.实验内容及步骤：

设计 FAT32 优盘文件的恢复程序。

7.思考与练习

(1) 文件删除前后 FDT、FAT、物理扇区数据有什么变化？

(2) 怎样恢复文件的簇链？

第二部分 实验指导

1 用户接口

1.1 Linux 基本操作

1. 启动 Linux

开机启动 Ubuntu 13.10。看到屏幕出现用户名和密码信息后，选择 Administrator，在密码栏输入：123456。登录成功后进入 Unity 图形界面。

2. 认识 Ubuntu 桌面

Ubuntu 桌面包括一个启动器、应用程序和窗口，如图 1.1 所示。桌面左边一列是启动器。在启动器中，显示了多个应用程序的图标。点击应用程序图标，可快速启动相应的应用。

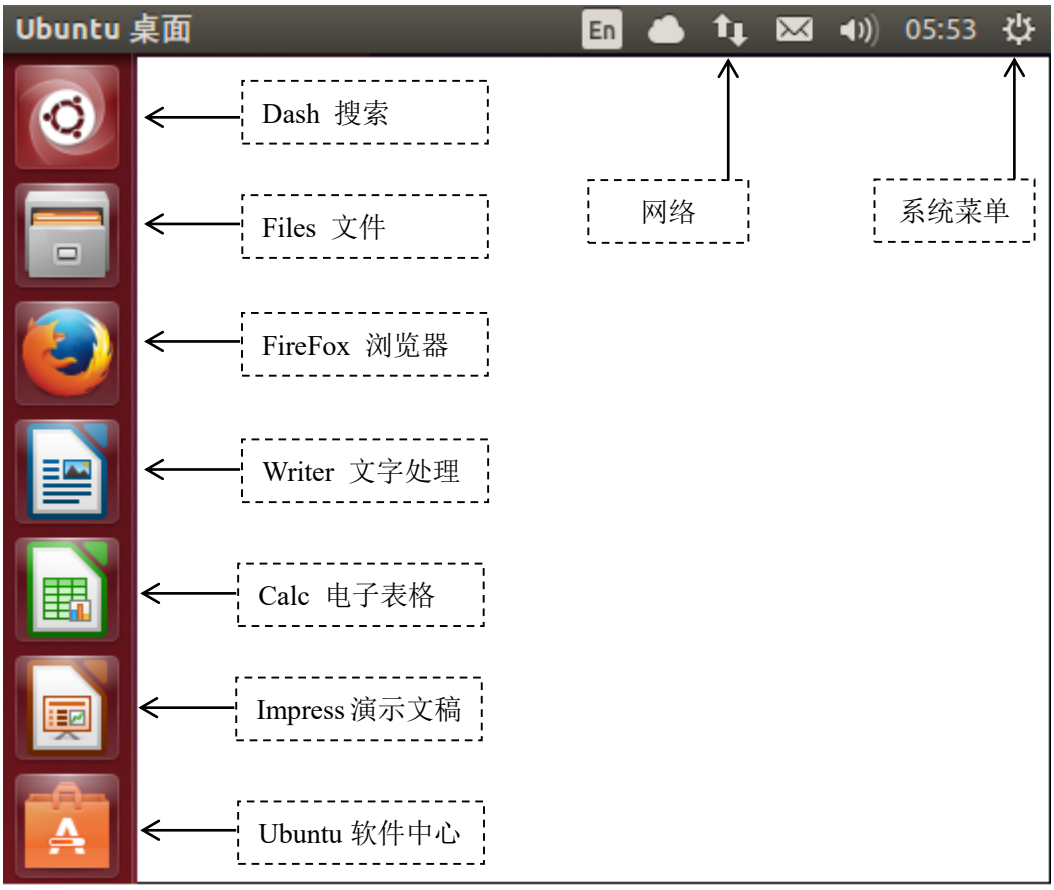


图 1.1 Ubuntu 桌面

关于 Ubuntu 桌面的其他操作，请参考 https://help.ubuntu.com/，或点击屏幕右上角的“系统菜单”，再选择“帮助”命令。

3. 使用终端

在 Ubuntu 里，终端提供命令行界面。在命令行界面下，用户可以向系统提交命令，系

统的命令解释程序（又叫 shell）解释执行用户提交的命令，并将结果返回给用户。操作步骤是：鼠标点击启动器里的搜索图标，输入 Ter，搜索，点击终端图标启动终端。或者同时按下快捷键“Ctrl+Alt+T”，也可以启动终端。进入终端后，出现$提示符，系统等待用户键入命令。

注意，Linux 的用户分为超级用户和非超级用户。超级用户，名字为 root，可以执行所有的命令。非超级用户只能执行非特权命令。超级用户的命令提示符为#，而非超级用户的命令提示符为$。在$提示符下，输入命令： sudo –i，则进入 root 命令界面#。在#提示符下，输入命令：exit，则又退出 root 命令界面，显示$提示符。

凡是需要 root 特权的命令，就使用 sudo。例如 Ubuntu 升级命令：

```
sudo apt-get upgrade
```

4. 认识 Linux 文件系统的目录结构

文件系统的**目录**就是人们熟知的**文件夹**。文件夹这个名称很形象，便于理解，是给非计算机专业用户使用的。目录是操作系统的专业术语。Linux 中，每个设备或硬盘分区构成一个文件系统。单个文件系统按树状方式形成一个目录层次结构，如图 1.2 所示。

图中，最上层的是**根目录**，用“/”表示。/bin 目录存放常用的可执行命令；/dev 目录存放设备文件；/etc 目录存放系统管理和配置文件；/home 为用户目录；/lib 目录存放标准库；/sbin 目录存放系统管理员的管理程序；/root 是超级用户主目录；/mnt 目录给用户临时装载其他文件系统；/usr 目录存放系统配置文件和软件包程序。

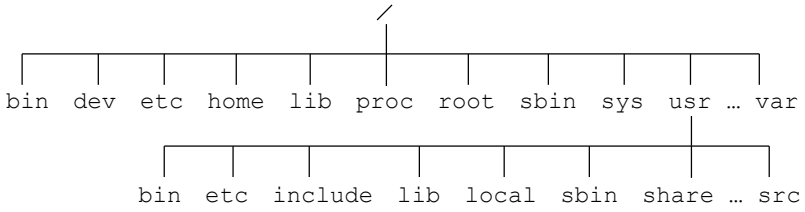


图 1.2　Linux 目录结构示意图

用户当前所处的位置称之为**当前目录**或叫**工作目录**。当前目录可以用圆点“.”表示。每个目录的上一级目录称为**父目录**，用连续两个圆点“..”表示。用户可以用 cd 命令改变当前目录。比如，把当前目录改为/usr，操作命令为：cd /usr。

上面的/usr 称为**路径名**。假设我们要访问 usr 目录下的 bin。有两条路径，一条是**绝对路径**（从根开始）：/usr/bin。另一条是**相对路径**(从当前目录开始)：./bin。也可以拐个弯来访问：../usr/bin。在图形界面下，使用文件管理器浏览文件和文件夹。点击桌面上的“文件”快捷图标，浏览文件和文件夹，认识 Linux 目录结构。

5. 文件名与文件属性

人们熟知，文件名和目录名由字母、数字、下划线等可打印字符构成，除此之外，还可以在文件名中使用**通配符**“*”和“?”。“*”代表任何字符串（包括无字符），“？”代表任一字符。比如，abc?d 的第 4 个字符为任意字符，匹配时可以表示 abc2d, 也可能表示 abcxd，等等。又比如，假定在当前目录下创建了以下文件：yourfile，myfile2，myfile3，xfile。若要列出以 myfile 开头的文件，可键入命令：ls myfile*。显示的结果为 myfile2，myfile3。从键盘键入 ls –l /etc。上述命令列出文件的基本属性信息：

```
drwxr-xr-x  5 root root     4096 10 月 17 03:03 apport
```

基本属性信息中，各字段含义如图 1.3 所示。

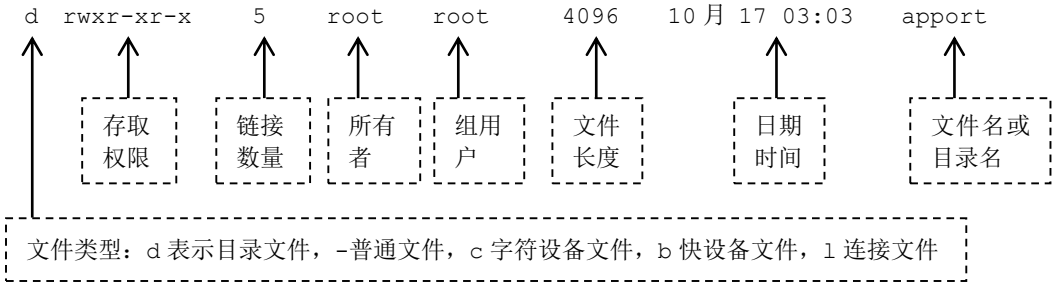


图1.3 文件属性信息

6.常用命令

(1) ls命令：列出目录内容

练习：(注意区分下列命令中的字母l与数字1，注意空格的输入)

```
ls -lF  /etc          长格式显示etc目录信息（不包括隐藏文件）
ls  -al /etc          显示etc目录所有信息（包括隐藏文件）
ls  -al /etc|more     分页显示etc目录所有信息（输入q退出more命令）
ls  -al  /etc/.*      显示/etc目录中所有以点开头的文件
ls  -lF  /etc/d*      显示/etc目录中所有以d开头的文件
ls  --help            ls命令的帮助信息
```

注：linux中，命令选项--help显示该命令的帮助信息。

(2) pwd命令：显示当前目录

练习：pwd

(3) cd命令：改变当前目录

练习：依次输入下列命令，观察结果。

```
cd /usr/bin           当前目录改为/usr/bin
ls -l                 列出当前目录内容，即列出目录/usr/bin的内容
cd ..                 进入当前目录的父目录,即/usr
ls -l                 列出当前目录/usr的内容
cd ~                  当前目录改为用户的home目录
ls ./                 列出当前目录的内容
cd /root              当前目录改为/root
ls                    列出当前目录/root的内容
```

(4) mkdir命令：建立子目录

```
练习：mkdir abc       在当前目录下创建目录abc
      mkdir abc/def1  在目录abc下创建目录def1
      mkdir abc/def2  在目录abc下创建目录def2
      ls abc          列出目录abc的内容
      mkdir /home/fun 在目录/home下创建目录fun
      ls /home        列出目录home的内容
```

(5) cp命令：复制文件

这条命令常用-fr这个“万能”命令选项，其中-f表示强制，-r表示所有子目录。

```
练习：ls abc                  先列出目录abc的内容
      cp -fr /etc/passwd abc/mypwd
          把目录/etc里的文件passwd复制到目录abc里，取名为mypwd
      ls abc                  再列出目录abc的内容（观察有何变化）
```

(6) mv 命令：文件改名或移动文件

练习：ls abc　　列出目录 abc 的内容

mv abc/mypwd abc/newpwd　　文件 mypwd 改名为 newpwd

ls abc　　再列出目录 abc 的内容（观察有何变化）

ls /home　　列出目录/home 的内容

mv abc/newpwd /home　　文件 newpwd 从目录 abc 移到目录 home

ls /home　　再列出目录/home 的内容（观察有何变化）

(7) rm 命令：删除文件或目录

练习：ls abc　　先列出目录 abc 的内容

rm -fr abc/def2　　删除目录 abc 里的子目录 def2

ls abc　　再列出目录 abc 的内容（观察有何变化）

(8) less 命令：显示文件内容

练习：less /home/newpwd　　输入 q 退出 less

(9) chmod 命令：设置文件访问权限

每个文件和目录都有访问许可权限。这些权限分为只读(r)，可写(w)，可执行(x)三种。这些权限赋予三种不同类型的用户，文件所有者(u),同组用户(g)，其他用户(o)。如图 1.4 所示。

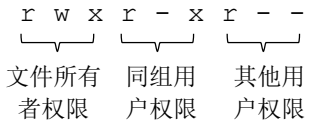


图 1.4　文件访问权限示意图

先用 ls -l 查看一下文件的访问权限。例如：

-rw-r--r-- 1 root root 483997 Jul 15 17:31 sobsrc.tgz

这里，第一个“-”表示普通文，接下来的 rw-r--r--为文件访问权。这些权限也可以用八进制表示为 644。其中八进制 6 的二进制位为 110,位为 0 代表无相应权限。所以 110 表示 rw-。余类推。

练习：chmod 777 /home/newpwd　　权限为所有用户都可读可写可执行

chmod u-w /home/newpwd　　解除所有者写权力(u 表示所有者，-为解除，w 为写权)

chmod u+wx /home/newpwd　　赋予所有者写文件和执行文件的权力

chmod g-x /home/newpwd　　解除同组用户执行权

注意：如果需要特权，就在命令前面加 sudo

(10) find 命令：查找文件

练习：find / -name passwd -print　　查找文件名为 passwd 的文件

find / -name cat -print　　查找名为 cat 的文件

(11) tar 命令：文件压缩打包命令

命令格式如下，

打包：tar -czvf 包名 被打包的目录或文件名

解包：tar -xzvf 包名

练习：tar -czvf myroot.tgz /root/*　　把目录/root 压缩打包为 myroot.tgz

cd /home

tar -xzvf /root/myroot.tgz　　在当前目录里解开包 myroot.tgz

(12) mount 和 umount 命令：挂装和卸载文件系统（课后练习）

命令格式： mount 选项 设备文件名 挂装点目录

例如，使用 U 盘，挂装命令如下，

mkdir /usb 先建一个目录，作为挂装点

sudo fdisk -l 显示 U 盘在哪个盘

假如显示 /dev/sdb1

sudo mount /dev/sdb1 /usb U 盘挂装到挂装点/usb

挂装后，访问目录/usb 即访问 U 盘。

卸载 U 盘，使用下列命令，

sudo umount /usb 或者 sudo umount /dev/sdb1

(13) ps 命令：进程列表

练习：ps -ef 查看所有进程及其进程号、系统时间

ps --help all 显示 ps 命令的帮助信息

(14) kill 命令：结束或终止进程

例如：kill 2277 终止进程号为 2277 的进程

kill -9 2278 强行终止进程号为 2278 的进程

kill -9 cat 强行终止进程名为 cat 的进程

(15) ln 命令：建立文件或目录的链接。链接分为硬链接和软链接两种，软链接又叫符号链接。

用 ln 建立硬链接命令格式是： ln 源文件 目标文件

用 ln 建立符号链接命令格式是：ln -s 源文件 目标文件

其中，-s 是 symbolic 的意思。注意，目录不能建立硬链接。

练习：ln -s /home myhome 建立 home 目录的符号链接,取名为 myhome

ls myhome

(16) 其他命令(仅供参考，课堂不练习)

uname -r	显示 linux 内核版本信息	df	文件系统磁盘空间
dmesg	系统启动信息	du	目录大小
hostname	主机名	free	内存和交换空间
who	在线登录用户	ifconfig	查看和配置网络接口参数
id	当前用户 id 信息	netstat	查看当前网络状态
rmdir	删除空目录	ping	测试网络连通情况
ftp	利用 ftp 上传和下载文件	telnet	远程登录
cat /etc/shells	显示系统可用 Shell	lsmod	查看内核加载的模块
chsh	选择 Shell	insmod	安装模块
echo $SHELL	当前正使用的 Shell	rmmod	删除模块
uptime	系统运行多长时间	grep	查找文件中的字符串
date	当前日期	dpkg	安装、更新、移除软件
chgrp	改变文件或目录所属组	chown	更改文件或目录属主和属组
adduser	增加用户名	passwd	设置用户口令（修改密码）
userdel	删除一个用户	groupadd	增加一个组用户
groupmod	修改一个组用户	groupdel	删除一个组用户

7. 重定向与管道

程序或文件的默认输入/输出方向有 stdin(键盘)，stdout(屏幕)，和 stderr(错误消息输出到屏幕上)。所谓**重定向**是指捕捉一个文件、命令、程序、脚本、或者是脚本中的代码块的输出，然后将这些输出发送到另一个文件、命令、程序、或脚本中。简单地说，

就是改变默认输入/输出方向。比如，ls 命令默认输出是屏幕。使用输出重定向“>”可以把 ls 的输出转到一个文件里。

练习：ls

ls /bin > abc　　输出重定向（新建或覆盖）。输出到 abc 文件中,屏幕不显示。

ls /bin >>abc　　输出重定向（追加方式）。 输出追加到文件 abc 的末尾。

hostname >>abc　　输出重定向（追加方式）。

cat </home/newpwd　输入重定向

管道是一种内存文件，用于两个进程之间通信。使用管道，一个命令的执行结果可以作为另一个命令的输入。比如，more 命令是分屏显示命令，而 ls 命令用于显示目录内容，当目录内容太多时，一屏显示不下来，这时可以通过管道，把 ls 的输出传送给 more 命令，然后由 more 命令一屏一屏显示。

练习： ls -lF /etc |more

man ls|more

echo "Hello" | write root　　向用户 root 发送消息 Hello.

8. vi 编辑器

vi 是一个文本编辑程序，用户使用它可以创建、打开、编辑文本文件。

(1)启动 vi： vi〈文件名〉 如：vi mytext

(2)vi 的三种模式

命令模式、插入模式、编辑模式，其相互转换如图 1.5 所示。

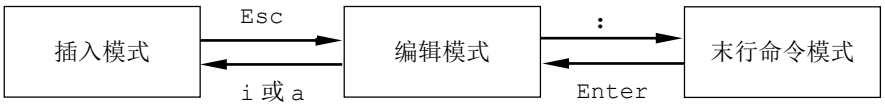


图 1.5 vi 模式及其转换

vi 启动后处于编辑模式，按 i 或 a 键进入插入模式，再按 Esc 键又退到编辑模式。若按:键则进入末行命令模式，按回车键返回编辑模式。

(3)编辑模式的命令

h、j、k、l	向左、下、上、右移动光标(每次移动一个字符)。
i	在光标前插入文本，转到输入模式
a	在光标后插入文本。转到输入模式
ctrl+f、ctrl+b	上下翻页
dd	删除行（如果要删除 3 行可键入 3dd）
yy	复制行（如果要复制 3 行可键入 3yy）
p	粘贴行 （注：先复制，光标移到目的行，然后粘贴）
x	删除光标后的字符
X	删除光标前的字符
u	恢复刚才被修改的文本

(4)插入模式用于输入文本

可打印字符	输入的文本内容
Esc	转到编辑模式

(5)末行命令模式的命令

:q!	放弃编辑内容并退出 vi
:wq!	保存文件并退出 vi

:w　　保存文件
:w filename　　另存为 filename
:r file　　读入文件 file 的内容
:/ abcd　　在文本中搜索字符串“abcd”，按 n 继续搜索

练习：创建名为 abc8 的文件，输入下列文本,保存退出。

第 1 行：Linux is a modern, free operating system based on UNIX standards.

第 2 行：First kernel developed in 1991 by Linus Torvalds.

第 3 行：last mode

修改第 3 行，在 last 后面插入 line，改行内容变为：last line mode

把第 1 行和第 2 行复制到第 4 行和第 5 行。操作步骤是：按压 Esc 键，操作光标键 K，使光标移到第 1 行，输入 2yy 完成 2 行复制。再操作光标键 j，使光标移到第 3 行。最后按压 p 键完成粘贴。输入:wq 保存并退出。

9.保存实验结果

如何保存终端窗口的实验结果呢？办法是先选取终端窗口中的内容，再复制、粘贴到一个文本文件中。操作步骤是，鼠标点击终端窗口，然后鼠标移到屏幕顶部，出现终端菜单栏，选择“编辑”，再依次点击“全选”、“复制”。鼠标点击启动器的搜索图标，输入 gedit,在 gedit 中点击鼠标右键，选择“粘贴”。鼠标移到屏幕顶部，出现 gedit 菜单栏，点击“文件”,“另存为”保存到文件中。

10.注销与关机

注销：鼠标点击屏幕右上角“系统菜单”，选择“注销“命令。

关机：鼠标点击屏幕右上角“系统菜单”，选择“关机“命令。

多用户环境下，可以在终端中使用命令关机。注意，只有**超级用户**才能关机。所以关机命令应该这样下达，

sudo shutdown -h now　关机

sudo shutdown -r now　重启

11.安装 Ubuntu（课后练习）

Ubuntu 有多种安装方法，这里只介绍怎样在 Windows 系统中安装 Ubuntu。

(1) 在线安装

从 ubuntu 官方网站 http://releases.ubuntu.com/saucy/下载 wubi.exe，运行 wubi.exe。

(2) 离线安装

从 Ubuntu 官方网站 http://releases.ubuntu.com/saucy/下载

wubi.exe 和 ubuntu-13.10-desktop-i386.iso 两个文件

下载完后复制到某个盘的根目录，比如 D:\盘。进入 D:\盘，以管理员身份运行下列命令：wubi.exe --force-wubi --32bit

（3）卸载 Ubuntu

进入 wubi 安装目录，找到 uninstall-wubi.exe 并运行。

1.2 shell 程序设计

【预备知识】

Shell 是命令解释程序的统称，是用户和 Linux 操作系统之间的交互接口，如图 1.6 所示。Shell 向用户提供命令提示符，等待接收用户提交的命令，解释命令并转到相应命令处理程序去执行。

Ubuntu 提供了多个 Shell。比如 sh，bash。看看系统有几个 Shell，输入下列命令：cat /etc/shells

系统默认的 Shell 是什么？输入下列命令：echo $SHELL

Shell 拥有自己内建的 Shell 命令集，提供 Shell 程序设计语言，可以执行 Shell 程序。Shell 程序也叫 Shell 脚本程序。

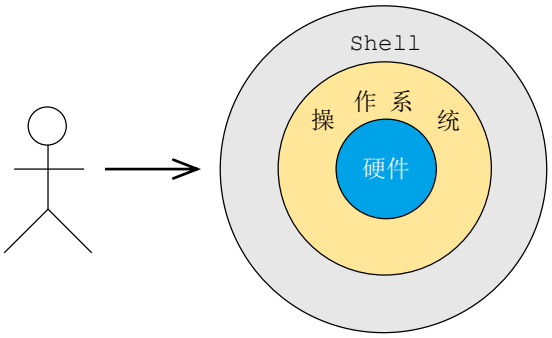


图 1.6　Shell 示意图

用 vi 或 gedit 编辑下面这段 Shell 程序，取名为 hello。

```
#!/bin/bash
x=10
y=20
let z=$x+$y
echo "hello,$z"
```

第一行，#! 说明执行该文件的 Shell。
第二行，定义变量 x。
第三行，定义变量 y。
第四行，引用变量 x 和 y，二者加运算，结果赋给变量 z。
第五行，显示字符串 hello，和变量 z 的值

怎能样执行 Shell 程序呢？有两个方法，第一个方法是指定一个 Shell 去执行，第二个方法是用 chmod 赋予 Shell 文件可执行权限。下面以上面的 hello 程序为例说明具体操作。

方法一，指定用 bash 执行 hello，命令是：bash hello

方法二，增加可执行权命令：chmod +x hello

执行程序命令：./hello

推荐使用第二种方法。

1. 变量的定义

使用变量可以保存有用的信息，也可以用于定制用户本身的工作环境，变量的定义方式如下：

USRDIR=/root

VAR=5

说明：USRDIR 和 VAR 均是变量名，“=”后面的/root 和 5 为变量的值，Shell 变量不需要声明变量的类型。

变量必须先定义后使用。使用变量需要在变量名字前加”$”符号，如$USRDIR 即可使用变量 USRDIR 的值。为了防止 Shell 误解变量值，也可以在变量名前后加花括号，如${USRDIR}，${VAR}。

2. 环境变量

定义方式与变量的定义相同，但是作用是用于用户自身工作环境的定制，在 Linux 操作

系统中，环境变量一般定义在用户初始化脚本中，可以使用 env 命令查看当前设置的环境变量，一般常用的一个环境变量为当前用户的用户目录 HOME, PS1, PATH。如需要查看某一个环境变量时，可使用如下命令:echo $VAR。如查看当前的用户目录，可使用 echo $HOME

在 Ubuntu Linux 中，用户环境初始化脚本文件为$HOME/.bashrc 文件。

3.位置参数变量

如果要向一个 shell 脚本传递信息，可以使用位置参数完成此功能。参数相关数目传入脚本的数据可以任意多，但只有前 9 个可被访问，使用 shift 命令可以改变这个限制。位置参数从 0 开始，到 9 结束，引用位置参数变量时，在参数前加”$”符号即可，如$1 代表第一个参数。$0 表示预留的保存实际脚本名字的参数。比如有一个脚本名为 comp.sh,执行时，传入两个参数 comp.sh 5 10，那么$0 等于”comp.sh”， $1 等于”5”，$2 等于 10。当使用 sh comp.sh 5 10 执行时,sh 不参与位置参数,也就是 0 号位置仍然从脚本名 comp.sh 开始

4.特定参数变量

脚本运行时，与脚本运行时的一些控制信息可以用特定参数变量表示，各特定参数变量的定义及含义如下:

$#	传递到脚本的参数个数
$*	以一个单字符串显示所有向脚本传递的参数，与位置变量不同，此选项参数可超过 9 个
$$	脚本运行的进程 ID
$!	后台运行的最后一个进程的 ID
$@	与$#相同，但是使用时加引号，并在引号中返回每个参数
$-	显示 shell 使用的当前选项，与 set 命令功能相同
$?	显示最后命令的退出状态，0 表示没有错误，任何其它值表示有错误

5.常用 shell 命令

(1) echo

使用 echo 命令可以显示文本行或变量，或者把字符串输入到文件，它的一般形式为，

echo string 或 echo $var

echo 命令常用的功能有很多，常用的有以下几个：

\c 不换行

\f 进纸

\t 跳格

\n 换行

如果是 linux 系统，那么必须使用-n 选项来禁止 echo 命令输出后换行。如，

echo -n “what is your name \c”

必须使用-e 选项才能使转义符生效.

如果想把 echo 命令输出的信息另存至文件中，可使用输出重定向行，如，

echo “My name is …” > filename.txt

这样，便将 My name is …这些字符串写入了 filename.txt 中并保存了。

补充：printf "%s %d\n" "abc" 10

(2) read

使用 read 语句可以从键盘或某一行文本中读入信息，并将其赋给一个变量，如果只指定一个变量，那么 read 将会把所有的输入赋给该变量，直到遇到第一个文件结束符或回车。如果赋给多个变量，那么将会把读入的信息，以空格作为域分隔符，分别将每个域按顺序赋变量。当定义的变量少于域的数目时，最后一个变量包含剩余的所有信息。给它的一般形式如下：

read var1 var2 …

示例：read -p "请输入两个数字" x1 x2 #-p 选项是提示

(3) cat

可以用它来显示文件内容，创建文件，还可以用它来显示控制字符，使用 cat 时不会在文件分页符处停下来，它会一下显示完整个文件，如果希望每次显示一页，可以使用 more 命令。若希望 cat 命令能显示控制字符，可以使用 cat -v filename 即可

(4)管道 |

管道符为一竖线 “|”，可以通过管道把一个命令的输出传递给另一个命令作为输入，它的一般表示形式为: 命令 1|命令 2

也就是命令 1 的输入作为命令 2 的输入如， ls | grep zengjian.txt

(5) tee

tee 命令把输出的一个副本送到标准输出，另一个副本拷贝到相应的文件中。如果希望在看到输出的同时，也将其存入一个文件中，那个这个命令最合适了，它的一般形式为:

tee -a files

-a 表示追加到文件末尾。

如 who | tee who.out

那么执行的结果将是屏幕上会显示 who 命令的结果，而且会将这些结果保存到 who.out 中。

6. 标准输入、标准输出、标准错误

在 Linux 和 Windows 操作系统中，将设备以文件的方式进行管理，叫作设备文件。操作系统启动后，会默认打开三个设备文件，分别是标准输入、标准输出、标准错误，它们的文件描述符分别为 0,1,2。其中标准输入缺省是键盘，标准输出缺省是屏幕，标准错误的缺省也是屏幕。

7. 文件重定向

在执行命令时，可以指定命令的标准输入，标准输出和错误，要实现这一点，就需要使用文件重定向。

command > filename	将标准输出重定向至文件
command >> filename	将标准输出追加重定向至文件
command < filename	Command 以 filename 文件作为输入
command << delimiter	Command 从标准输入读入，直到遇到 delemiter 分界符

command < &m	把文件描述符 m 作为标准输入
command > &m	把文件描述符 m 作为标准输出
command < &-	关闭标准输入

7. Shell 语句控制结构

Shell 的程序结构有:

（1） 条件测试语句

```
if [ 条件 ]
then
    命令行
fi
```

```
if [ 条件 ]
then
    命令行 1
else
    命令行 2
fi
```

```
if [ 条件 ]
then
    命令行 1
elif [ 条件 ]
    命令行 2
fi
```

注意，条件中，两边的中括号[和]与条件之间要有空格

字符串比较

>	大于
>=	大于或等于
=	等于
<	小于
<=	小于或等于
!=	不等于
-z	空串 (字符串长度为 0)

数值比较

-eq	等于
-ne	不等于
-gt	大于
-lt	小于
-le	小于或等于
-ge	大于或等于

逻辑操作

-a	与
-o	或
!	非

文件操作

-d	存在目录
-f	存在文件
-L	存在符号链接
-r	存在且可读
-w	存在且可写
-x	存在且可执行
-s	存在且长度为 0

（2） 循环结构

```
while  条件
do
    命令表
done
```

```
for 变量名 in 列表
do
    命令表
done
```

（3） 分支结构

```
case 值 in
 模式 1)
    命令 1
    …
    命令 n;;
 模式 2)
    命令 2
    …
    命令 m;;
esac
```

注意每个模式以两个分号”;;”结束

8. 函数调用

在定义函数时，括号中不能有参数，且没有返回类型及返回值，定义方式如下：

```
function 函数名()
{
    命令表
}
```

```
function 函数名()
{
    命令表
    return [ n ]
}
```

注意：关键字 **function 可以省略**。return 命令退出函数，返回值 n。

调用时，直接使用函数名。

【训练任务】

任务 1、编写一段 shell 程序，实现文件的备份和恢复。

参考代码如下：

```
#!/bin/sh
# backup.sh

#备份目录函数
backupdir()
{
  dirtest
  echo "Backupping..."
  tar -zcvf /tmp/backup.tar.gz $DIRECTORY
```

```
}

#恢复目录函数
restoredir()
{
  dirtest
  echo "Restroing..."
  tar -xzvf /tmp/backup.tar.gz
}

#验证目录函数
dirtest()
{
  echo "Please enter the directory name of backup file:"
  read DIRECTORY
  if [ ! -d $DIRECTORY ]
  then
    echo "Sorry,$DIRECTORY is not a directory!"
    exit 1
  fi
  cd $DIRECTORY
}

clear
ANS=Y
while [ $ANS = Y -o $ANS = y ]
do
  echo "============================="
  echo "=   Backup-Restore Menu     ="
  echo "+++++++++++++++++++++++++++++"
  echo "+    1:Backup Directory      +"
  echo "+    2:Restore Directory     +"
  echo "+    3:Exit                   +"
  echo "+++++++++++++++++++++++++++++"
  echo "Please enter a choice(0-2):"

  read CHOICE
  case "$CHOICE" in
    1) backupdir;;
    2) restoredir;;
    0) exit 1;;
    *) echo "Invalid Choice!"
       exit 1;;
  esac
```

```
  if [ $? -ne 0 ]
  then
    echo "Program encounter error!"
    exit 2
  else
    echo "Operate successfully!"
  fi
  echo "Would you like to continue? Y/y to continue,any other key
to exit:"
  read ANS
  clear
done
```

用 gedit 输入代码，存盘为 backup.sh

```
chmod +x backup.sh
./backup.sh
```

任务 2、编写一段 shell 程序，使用一个菜单界面，方便在 Linux 下对 U 盘的加载、卸载过程。程序要求实现

以下 5 个功能：

（1）加载 U 盘。

（2）卸载 U 盘。

（3）查看加载后的 U 盘信息。

（4）从 Linux 分区的硬盘中拷贝文件到 U 盘中

（5）从 U 盘拷贝文件到 Linux 分区的硬盘指定位置上。

第一步：编写菜单，和加载 U 盘以及卸载 U 盘的程序：

```
#! /bin/sh
# mountusb.sh

#退出程序函数
quit()
{
  clear
  echo "**********************************"
  echo "*  Thank you to use ,Good bye!   *"
  echo "**********************************"
  exit 0
}

#加载 U 盘函数
mountusb()
{
  clear
```

```
  mkdir /mnt/usb
  /sbin/fdisk -l | grep /dev/sd
  echo "Please Enter the device name of usb as shown above:"
  read PARAMETER
  mount /dev/$PARAMETER /mnt/usb
}

#卸载 U 盘函数
umountusb()
{
  clear
  umount /mnt/usb
}

clear
while true
do
  echo "========================================="
  echo "***     UNIX USB MANAGE PROGRAM    ***"
  echo "========================================="
  echo "        1-MOUNT USB                 "
  echo "        2-UMOUNT USB                "
  echo "        0-EXIT                     "
  echo "========================================="
  echo -e "Please Enter a Choice(0-5):\c"
  read CHOICE
  case $CHOICE in
    1) mountusb;;
    2) umountusb;;
    0) quit;;
    *) echo "Invalid Choice!Correct Choice is (0-5)"
       sleep 4
       clear;;
  esac
done
```

用 gedit 输入代码，存盘为 mountusb.sh

```
chmod +x mountusb.sh
./mountusb.sh
```

第二步：编写显示 U 盘内容的函数，并在菜单中添加选项。

参考代码：

```
#显示 U 盘内容函数
display()
{
```

```
  clear
  ls -l /mnt/usb
}
```

第三步：编写将当前目录的文件拷贝至 U 盘的函数，并在菜单中添加相应的选项：
参考代码：

```
#拷贝硬盘文件到 U 盘函数
cpdisktousb()
{
  clear
  echo "Please Enter the filename to be Copied(under current
directory):"
  read FILE
  echo "Copying,Please wait..."
  cp $FILE /mnt/usb
}
```

第四步：编写将 U 盘文件拷贝至当前目录的函数，并在菜单中添加相应的选项：
参考代码：

```
#拷贝 U 盘文件到硬盘函数
cpusbtodisk()
{
  clear
  echo "Please Enter the filename to be Copied in USB:"
  read FILE
  echo "Copying,Please wait..."
  cp /mnt/usb/$FILE .
}
```

任务 3、编写一段 shell 程序，主要功能是用来存储和查询学生成绩，并提供菜单显示选项；同时可以根据用户输入的选项执行查询、添加等功能。

假设学生姓名和成绩信息保存在当前目录 record 文件中，而且文件的每一行记录了一个学生的一条记录。

第一步：编写菜单，和添加记录以及退出程序函数。

参考代码：

```
#退出程序函数
quit()
{
  clear
  exit
}
```

```
#增加记录函数
add()
{
  clear
  echo "Enter name and score of a record."
  echo "\c"
  if [ ! -f ./record ]; then
    touch record
  fi
  read NEWNAME
  echo "$NEWNAME" >> ./record
  sort -o ./record ./record
}

clear
while true
do
  echo "***************************************"
  echo "*    STUDENT'S RECORD MENU          *"
  echo "***************************************"
  echo "#######################################"
  echo "#    1:ADD A RECORD                  #"
  echo "#    Q:EXIT PROGRAM                  #"
  echo "#######################################"
  echo -n "\tPlease Enter Your Choice {1,Q]:"
  read CHOICE
  case $CHOICE in
    1) add;clear;;
    Q|q) quit;;
    *) echo "Invalid Choice!";
       sleep 2;
       clear;;
  esac
done
```

用 gedit 输入代码，存盘为 mountusb.sh

```
chmod +x mountusb.sh
./mountusb.sh
```

第二步：编写查询记录的函数，并在菜单中增加相应的选项。

参考代码：

```
#查询函数
search()
{
```

```
 clear
 echo -e "Please Enter Name >>>"
 read NAME
 if [ ! -f ./record ];then
   echo "You must have some scores before you can search!"
   sleep 2
   clear
   return
 fi
 until [ ! -z "$NAME" ]
 do
   echo "You didn't enter a name!"
   echo "Please Enter Name >>>"
   read NAME
 done
 grep -i "$NAME" ./record 2> /dev/null
 if  [ $? = 1 ];then
    echo "Name not in record."
 fi
}
```

第三步：编写删除记录的函数，并在菜单中增加相应的选项。

```
#删除记录函数
delete()
{
 clear
 echo "Please Enter Name >>>"
 read NAME
 if [ ! -f ./record ]; then
   echo "You must have some scores before you can search!"
   sleep 5
   clear
   return
 fi
 until [ ! -z "$NAME" ]
 do
   echo "You didn't enter a name!"
   echo "Please Enter Name >>>"
   read NAME
 done
 grep -i "$NAME" ./record 2> /dev/null
 if  [ $? = 1 ];then
    echo "Name not in record."
 else
```

```
    cp record record.bak
    rm -f record
    grep -vi "$NAME" ./record.bak > record
    rm -f record.bak
    echo "Delete Successfully!"
  fi
}
```

第四步：编写显示所有记录的函数，并在菜单中增加相应的选项。

```
#显示所有记录函数
display()
{
  more ./record
}
```

任务 4、编写一个 Shell 程序，测试一个指定程序的执行时间。

2 进程管理

2.1 进程创建与控制

1. 进程的创建

【预备知识】

(1) 编译与连接程序

Linux 下常用的 C/C++语言编译器是 GCC（GNU Compiler Collection），它是 GNU 项目中符合 ANSI C 标准的编译系统，能够编译 C、C++等语言编写的程序。它除了功能强大，结构灵活外，还可以通过不同的前端模块来支持各种语言，如 Java、Pascal 等。GCC 编译过程分为四个阶段：预处理（Pre-Processing）、编译（Compiling）、汇编（Assembling）、连接（Linking）。GCC 的使用方法可以通过--help 选项或 man 命令获得。GCC 编译选项如表 2.1 所示。

表 2.1 GCC 选项

选项	说明
-c	只编译不连接，生成目标文件.o
-S	只编译不汇编，生成汇编代码
-E	只预编译
-g	在可执行文件中包含标准调试信息
-o file	把输出文件输出到 file 里
-v	显示编译器版本信息
-I dir	在头文件的搜索列表中添加 dir 目录
-L dir	在库文件的搜索列表中添加 dir 目录
-static	连接静态库
-llibrary	连接名为 library 的库文件

Gcc 编译 C 程序的最简语法格式:

gcc 程序名.c -o 输出文件名

例 1. 把 abc.c 编译连接为可执行文件 abc.exe，命令如下，

```
gcc abc.c -o abc.exe
```

例 2. 把 def.c 编译为目标文件 def.o，再连接为可执行文件 def2，命令如下，

```
gcc -c def.c
gcc def.o -o def2
```

(2) fork()系统调用

头文件：#include <unistd.h>

函数原型： pid_t fork(void);

函数功能：fork 的功能是创建**子进程**。调用 fork 的进程称为**父进程**。如图 2.1 所示。子进程是父进程的一个拷贝，它继承了父进程的用户代码、组代码、环境变量、已打开的文件

代码、工作目录及资源限制。fork 语句执行后，内核向父进程返回子进程的进程号，向子进程返回 0。**父子进程都从 fork()的下一句开始并发执行**。

返回值：

返回值==-1：创建失败。

返回值==0： 程序在子进程中。

返回值>0： 程序在父进程中。（该返回值是子进程的进程号）

编程提示： 虽然子进程是父进程的一个复制品，但父子的行为是不同的。编程时要抓住内核的返回值。通过返回值，可以知道是父进程还是子进程，因而可以编写不同行为的代码。

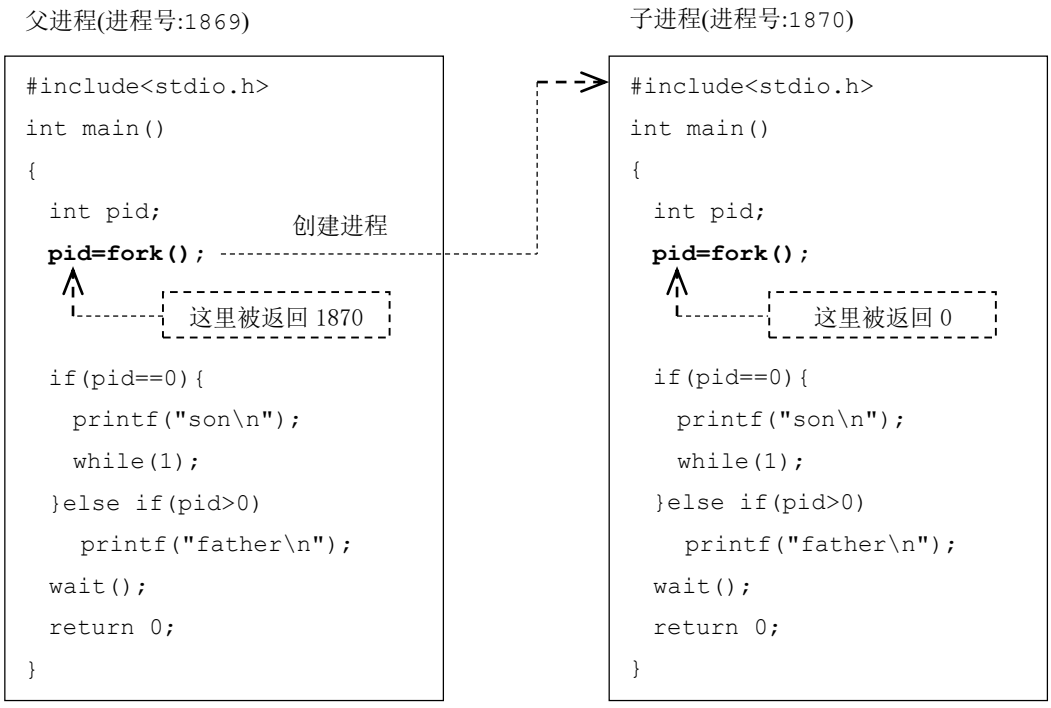


图 2.1 fork()创建进程示意图

(3) wait()系统调用

头文件：#include <sys/wait.h>

函数原型：pid_t wait(int *status);

函数功能：wait 的功能是等待子进程结束。发出 wait 调用的进程只要有子进程，就会睡眠直到子进程中的一个终止为止。若没有子进程，则该调用立即返回。

函数参数：status 是子进程退出时的状态信息。

返回值：成功则返回子进程号，否则返回-1。

练习：用 gedit 编辑图 2.1 中的源程序，保存为 e201.c

编译 gcc -o e201 e201.c

运行 ./e201

查看进程号，新建一个终端，然后键入 ps -e | grep e201

观察屏幕，是否有两个名为 e201 的进程。比较它们的进程号，判别哪个是父进程，哪个是子进程。程序中的 while(1);语句是为了不让程序退出来，以便于你观察进程状态。

用 kill 命令把这两个进程终止。

【任务 1】

编写一段程序,使用系统调用 fork()创建两个子进程 p1 和 p2。p1 显示字符'b'，p2 显示字符'c'，父进程显示字符'a'，父进程和两个子进程并发运行。观察并记录屏幕上的显示结果，分析原因。

(1)程序设计

用 while 语句控制 fork()直到创建成功。用 if 语句判别是在子进程中还是在父进程中。程序运行后会创建三个进程，它们分别是子进程 p1、p2、父进程。这三个进程并发运行。假定子进程有些任务要做，完成这些任务要花一定时间，因此，可以用一个延时函数简单地模拟这些任务。

```
//e202.c
#include <stdio.h>

void delay(int x) //延时函数
{
    int i,j;
    for(i=0;i<x;i++)
        for(j=0;j<x;j++) ;
}

int main()
{
    int p1,p2;
    while((p1=fork())==-1); //创建子进程 p1
    if(p1==0) //子进程 p1 创建成功
    {
        delay(10); //子进程 p1 延时
        putchar('b'); //子进程 p1 显示字符'b'
    }else{
        while((p2=fork())==-1); //创建子进程 p2
        if(p2==0) //子进程 p2 创建成功
        {
            delay(10); //子进程 p2 延时
            putchar('c'); //子进程 p2 显示字符'c'
        }else{
            delay(100); //父进程延时
            putchar('a'); //父进程显示字符'a'
        }
    }
    return 0;
}
```

(2)上机操作

用 gedit 输入上述源代码，取名为 e202.c。

编译 gcc -o e202 e202.c
运行 ./e202
按向上的光标键、回车，运行刚才的程序。快速重复这个步骤，观察并记录结果。

(3)问题

屏幕上是否有时显示 bac,有时显示 bca，…。为什么会这样呢？

(4) 思考与练习(课后完成)

(1) 父子进程同步实验：编写一段程序， 使用系统调用 fork()创建一个子进程，子进程求前 10 个自然数的和并打印出来，使用系统调用 wait()让父进程等待子进程结束。

(2) 已知下列Linux程序执行后，屏幕上打印结果如下，画出进程家族树（以进程号标示进程）。

```
//Linux程序
#include "stdio.h"
#include "unistd.h"
int main()
{
  int p1,p2,p3;
  p1=fork();
  p2=fork();
  p3=fork();
  //注：getpid()获取当前进程 pid
  if(p1>0 && p2>0 && p3>0)   printf("A:%d\n",getpid());
  if(p1==0 && p2>0 && p3>0)  printf("B:%d\n",getpid());
  if(p1==0 && p2==0 && p3>0) printf("C:%d\n",getpid());
  if(p1==0 && p2==0 && p3==0)printf("D:%d\n",getpid());
  if(p1==0 && p2>0 && p3==0) printf("E:%d\n",getpid());
  if(p1>0 && p2==0 && p3>0)  printf("F:%d\n",getpid());
  if(p1>0 && p2==0 && p3==0) printf("G:%d\n",getpid());
  if(p1>0 && p2>0 && p3==0)  printf("H:%d\n",getpid());
  sleep(10);
  return 0;
}
  //屏幕打印结果
  A:9850
  B:9851
  C:9852
  D:9853
  E:9854
  F:9855
  G:9856
  H:9857
```

2. 软中断通信

【预备知识】

(1) 信号

信号是一种软件中断，用来通知进程发生了异步事件。进程之间可以互相通过系统调用 kill 发送软中断信号。内核也可以因为内部事件而给进程发送信号，通知进程发生了某个事件。注意，信号只是用来通知某进程发生了什么事件，并不给该进程传递任何数据。

收到信号的进程对各种信号有不同的处理方法。处理方法可以分为三类：第一种是类似中断的处理程序，对于需要处理的信号，进程可以指定处理函数，由该函数来处理。第二种方法是，忽略某个信号，对该信号不做任何处理，就象未发生过一样。第三种方法是，对该信号的处理保留系统的默认值，这种缺省操作，对大部分的信号的缺省操作是使得进程终止。进程通过系统调用 signal 来指定进程对某个信号的处理行为。

在进程表的表项中有一个软中断信号域，该域中每一位对应一个信号，当有信号发送给进程时，对应位置位。由此可以看出，进程对不同的信号可以同时保留，但对于同一个信号，进程并不知道在处理之前来过多少个。

每个信号都有一个名字，这些名字都以 SIG 开头。编号为 1 ~ 31 的信号为传统 UNIX 支持的信号，是非实时的不可靠信号，编号为 32～63 的信号是后来扩充的，是实时的可靠信号。不可靠信号和可靠信号的区别在于前者不支持排队，可能会造成信号丢失，而后者不会。表 2.2 是编号为 1~20 的信号。

表 2.2 信号

信号名	信号值	默认动作	说明
SIGHUP	1	终止进程	通知同一会话内的各个作业,这时它们与控制终端不再关联。
SIGINT	2	终止进程	当用户按中断键(Ctrl-C)时，终端驱动程序产生此信号，信号送到前台进程组的每个进程。
SIGQUIT	3	终止进程	当用户按退出键(Ctrl-\)时，终端驱动程序产生此信号，信号送到前台进程组的每个进程。
SIGILL	4	建立 CORE 文件	执行了非法指令。通常是因为可执行文件本身出现错误, 或者试图执行数据段。堆栈溢出时也有可能产生这个信号。
SIGTRAP	5	建立 CORE 文件	由断点指令或其它 trap 指令产生。由 debugger 使用。
SIGABRT	6	建立 CORE 文件	调用 abort 函数生成的信号。
SIGBUS	7	建立 CORE 文件	非法地址，包括内存地址对齐出错。它与 SIGSEGV 的区别在于后者是由于对合法存储地址的非法访问触发的(如访问不属于自己存储空间或只读存储空间)。
SIGFPE	8	建立 CORE 文件	在发生致命的算术运算错误时发出。不仅包括浮点运算错误, 还包括溢出及除数为 0 等其它所有的算术的错误。
SIGKILL	9	终止进程	用来立即结束程序的运行。本信号不能被阻塞、处理和忽略。如果管理员发现某个进程终止不了，可尝试发送这个信号。
SIGUSR1	10,16	终止进程	用户定义信号 1 (留给用户使用)
SIGSEGV	11	终止进程	试图访问未分配给自己的内存, 或试图往没有写权限的内存地址写数据。
SIGUSR2	12,17	终止进程	用户定义信号 2 (留给用户使用)
SIGPIPE	13	终止进程	管道破裂。这个信号通常在进程间通信产生，比如采用 FIFO(管道)通信的两个进程，读管道没打开或者意外终止就往管道写，写进程会收到 SIGPIPE 信号。此外用 Socket 通信的两个进程，写进程在写 Socket 的时候，读进程已经终止。

SIGALRM	14	终止进程	时钟定时信号，计算的是实际的时间或时钟时间。 alarm 函数使用该信号。
SIGTERM	15	终止进程	程序结束信号，该信号可以被阻塞和处理。通常用来要求程序自己正常退出，shell 命令 kill 缺省产生这个信号。如果进程终止不了，用户才会尝试 SIGKILL。
SIGCHLD	17	忽略信号	子进程结束时，父进程会收到这个信号。
SIGCONT	18	忽略信号	让一个停止的进程继续执行。 本信号不能被阻塞。
SIGSTOP	19	停止进程	暂时停止进程的执行。本信号不能被阻塞，处理或忽略。
SIGTSTP	20	停止进程	停止进程的运行，但该信号可以被处理和忽略。 用户键入 SUSP 字符时(通常是 Ctrl-Z)发出这个信号。

要查询系统中的信号，可以使用命令：kill -l

要了解信号的意义及其用法，可以使用命令：man 7 signal

(2) kill()系统调用：

头文件 #include <signal.h>

函数原型 int kill(pid_t pid, int signumber);

kill 的功能是向一个进程或一个进程组发送信号。

函数参数意义如下：

pid 指定信号的接受进程或进程组。

pid==-1：信号发送到所有进程。

pid==0： 信号发送到与发送者同组的所有进程。

pid>0：信号发送到进程号为 pid 的进程。

pid<-1：信号发送到进程组号为 pid 绝对值的所有进程。

signumber 为要发送的信号。

(3) signal()系统调用：

头文件：#include <signal.h>

函数原型：void(*signal(int signumber,void(*func)(int)))(int);

函数功能：signal 的功能是捕捉信号,给信号指定处理函数。

函数参数：

signumber 是要捕捉的信号；

func（）是指定给 signumber 的处理函数。func()可以是下面 3 种类型：

(1) 用户定义函数。

(2) SIG_IGN，告诉系统忽略这个信号。

(3) SIG_DFL，告诉系统对这个信号的处理采用系统默认的处理。

(4) waitpid()系统调用：

头文件：#include <sys/wait.h>

函数原型：pid_t waitpid(pid_t pid, int *status, int options);

函数功能：waitpid 与 wait 在调用上的区别是：waitpid 等待由参数 pid 指定的子进程退出。函数参数：pid<-1 时，将等待任何一个组 ID 等于 pid 绝对值的进程；

pid=0 时，等待任何一个组 ID 和调用者的组 ID 相同的进程；

pid>0 时，则等待指定的进程（其进程 ID=pid）；

pid=-1 时，与 wait 的调用功能相似，等待任何一个子进程退出。

(5) exit()系统调用：

头文件：#include <stdlib.h>

函数原型：void exit(int status);

函数功能：exit 的功能是正常终止进程自己。

函数参：status 是返回给父进程结束状态码。

(6) pause()系统调用：

头文件 #include <unistd.h>

函数原型 int pause(void);

函数功能：pasuse 的功能是将进程挂起直到有一个信号被接收处理。就是说，只有当一个信号处理函数被执行且返回之后 pause 才返回。

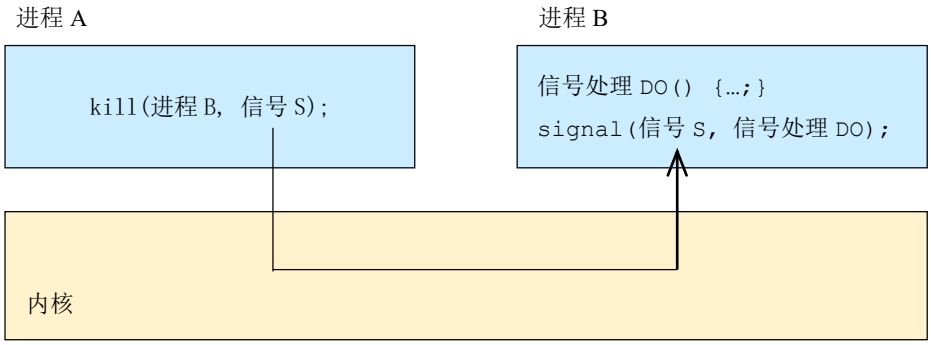


图 2.4 信号通信示意图

【任务 2】

编写一段程序，实现下列功能：创建两个子进程。用户按中断键(Ctrl-C),被父进程捕获。父进程捕获用户中断键后向子进程发信号，让两个子进程显示下列信息后终止：

```
Child process 1 is killed by parent!
Child process 2 is killed by parent!
```

父进程等待两个子进程终止后，自己输出下列信息后终止：

```
Parent process is killed!
```

(1)参考程序

使用 while 语句控制 fork()创建两个字进程 p1 和 p2。父进程使用系统调用 signal() 捕捉用户的中断键。若用户按下中断键 Ctrl-C，则用 kill()向 p1、p2 发自定义信号。然后用 wait()等待 p1、p2 结束。p1、p2 用 signal()捕捉父进程的自定义信号(信号 16 和 17)，捕到后显示信息，然后用 exit（）终止自己。代码如下：

```
//e203.c
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

void waiting();
void stop();
```

```
int wait_mark; //忙等待标志, 1--等待, 0--不等待

int main()
{
  int p1,p2;
  while((p1=fork())==-1); //创建子进程 p1
  if(p1>0)
  {
    while((p2=fork())==-1); //创建子进程 p2
    if(p2>0) //父进程
    {
      printf("parent\n");
      wait_mark=1; //置等待标志
      signal(SIGINT,stop); //捕捉'Ctrl-C'信号，调用信号处理函数 stop()
      waiting(); //忙等待
      kill(p1,16); //向 p1 发中断信号 16*/
      kill(p2,17); //向 p2 发中断信号 17*/
      wait(0); //同步, 等待 p1 结束
      wait(0); //同步, 等待 p2 结束
      printf("parent process is kill!\n");
      exit(0); //父进程结束自己
    }else{       //子进程 p2
      printf("p2\n");
      wait_mark=1; //置等待标志
      signal(17,stop);  //捕捉父进程信号 17，调用信号处理函数 stop()
      waiting(); //忙等待
      lockf(stdout,1,0); //锁住标准输出 stdout
      printf("child process 2 is killed by prent!\n");
      lockf(stdout,0,0); //解锁
      exit(0); //子进程 p2 结束自己
    }
  }else{ //子进程 p1
    printf("p1\n");
    wait_mark=1; //置等待标志
    signal(16,stop); //捕捉父进程信号 16，调用信号处理函数 stop()
    waiting(); //忙等待
    lockf(stdout,1,0); //锁住标准输出 stdout
    printf("child process 1 is killed by parent!\n");
    lockf(stdout,0,0); //解锁
    exit(0); //子进程 p2 结束自己
  }
  return 0;
}
```

```
void waiting() //忙等待函数
{
  while(wait_mark!=0);      //忙等待!!!
}

void stop()
{
  wait_mark=0; //清除忙等待标志
}
```

(2)上机操作

用 gedit 输入源代码。
编译、运行，观察屏幕，记录结果。

(3) 问题：为什么两个子进程没有显示预期的信息？

原因分析：SIGINT 信号会发给前台的每一个进程，因此，子进程 p1、p2 收到后会立即终止。这样就不会显示后面的信息了。

修改程序：只要让子进程忽略中断键 Ctrl-C 就可以了。所以在每个子进程的 wait_mark=1; 语句前添加一句：signal(SIGINT,SIG_IGN); /* 子进程忽略 ctrl-C 信号 */

重新编译、运行；观察、记录结果。
结果：两个子进程和父进程都显示了预期的信息。

(4) 思考与练习(课后完成)

阅读下列程序，上机测试，记录程序运行结果。回答下列问题：

(a)为什么进程一直保持在阻塞状态而无法退出？

(b)怎样修改程序，使进程不会一直阻塞下去？

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

int pid1,pid2;
void IntDelete()          /* 发送信号函数 */
{
   kill(pid1,16);         /* 发信号 16 */
   kill(pid2,17);         /* 发信号 17 */
}

void Int1()              /* 子进程 1 打印信息 */
{
   printf("child process 1 is killes by parent!\n");
   exit(0);
```

```
}

void Int2()              /* 子进程 2 打印信息 */
{
   printf("child process 2 is killes by parent!\n");
   exit(0);
}

int main()
{
   int exitpid;
   signal(SIGINT,SIG_IGN);       /* 忽略 SIGINT 信号 */
   signal(SIGQUIT,SIG_IGN);      /* 忽略 SIGQUIT 信号 */
   while((pid1=fork())==-1);     /* 创建子进程 pid1 */
   if(pid1==0)                   /* 在子进程 pid1 中 */
   {
      printf("p1\n");            /* pid1 打印它的名字 */
      signal(SIGUSR1,Int1); /* pid1 捕捉信号 SIGUSR1,转信号处理函数 Int1() */
      signal(16,SIG_IGN);        /* pid1 忽略信号 16 */
      pause();                   /* pid1 等待信号被处理 */
      exit(0);                   /* pid1 结束 */
   }else{
      while((pid2=fork())==-1);  /* 创建子进程 pid2 */
      if(pid2==0)                /* 在子进程 pid2 中 */
      {
         printf("p2\n");         /* pid2 打印它的名字 */
         signal(SIGUSR2,Int2); /*pid2 捕捉信号 SIGUSR2,转信号处理函数 Int2()*/
         signal(17,SIG_IGN);     /* pid2 忽略信号 17 */
         pause();                /* pid2 等待信号被处理 */
         exit(0);                /* pid2 结束 */
      }else{
         printf("parent\n");     /* 父进程打印它的名字 */
         signal(SIGINT,IntDelete); /*父进程捕捉 SIGINT 信号,转 IntDelete() */
         waitpid(-1,&exitpid,0); /* 父进程等待任何子进程结束 */
         printf("parent process is killed!\n"); /* 父进程打印信息 */
      }
   }
   return 0;
}
```

上机操作说明：假设程序名为 e800，程序运行后，键入 ps -A
看看有没有三个名为 e800 的进程。区分它们的 PID。
使用 kill 杀掉这三个进程

【实战任务】控制 FireFox 浏览器运行:创建进程，在该进程中打开 FireFox 浏览器，向 FireFox 发送信号使其停止、继续、终止。

提示：创建进程加载外部程序的系统调用是 exec 函数族的函数,用法范例：

```
execlp(“firefox", “firxfox", "-new-window", NULL);
```

2.2 进程间通信

1. 进程的管道通信

【预备知识】

(1)管道

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，是进程通信的一种方法。

管道是半双工的，数据只能向一个方向流动；如果要进行双向通信，则需要建立起两个管道；管道只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）；对于管道两端的进程而言，管道就是一个存在于内存中的文件。数据的读出和写入：一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。数据被读完之后就从管道中消失。

创建管道使用系统调用 pipe()。管道两端是固定了任务的，一端只能用于读，另一端只能用于写。管道两端可以分别用 fd[0]和 fd[1]来描述，如图 2.2 所示。

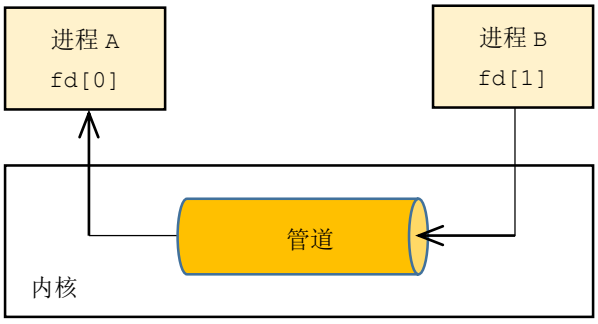


图 2.2　管道通信示意图

fd[0]为管道读端，fd[1]为管道写端。管道的读写使用文件系统调用 read()、write()。Linux 系统不保证管道读写的原子性。

(2) pipe（）系统调用

头文件：#include <unistd.h>

函数原型：int pipe(int fd[2]);

功能：创建管道。fd[0]为管道读端，fd[1]为管道写端。

(3) read（）系统调用

头文件：#include <unistd.h>

函数原型：ssize_t read(int fd, void* buf, size_t count);

功能：从参数 fd 所指的文件中读取 count 个字节到 buf 所指向的内存中。返回实际读取到的字节数，若返回值为 0，表示已到达文件尾或者无数据可读。若返回值为-1，表示出错。

(4) write（）系统调用

头文件：#include <unistd.h>

函数原型：ssize_t write(int fd, void* buf, size_t count);

功能：把参数 buf 所指向的内存中的 count 个字节写入到 fd 所指的文件内。返回实际写入的字节数，若返回值为-1，表示出错。

(5) sleep（）函数

头文件：#include <unistd.h>

函数原型：unsigned int sleep(unsigned int seconds);

功能：使进程挂起 senconds 秒，直到指定的时间到或者收到信号。如果指定挂起的时间到了，该函数返回 0；如果被信号打断，则返回剩余挂起的时间数。

(6) sprintf（）函数

头文件：#include <stdio.h>

函数原型：int sprintf(char *str, const char *format, ...);

功能：格式化字符串 format，格式化的结果复制到 str 中。成功则返回 str 的长度;失败则返回-1。

【任务 3】

编写一段程序，实现进程的管道通信。使用系统调用 fork()创建两个子进程 p1 和 p2。使用系统调用 pipe()建立一条管道。两个子进程 p1 和 p2 分别向管道各写一句话：

child 1 is sending a message!

child 2 is sending a message!

父进程从管道中读出来自两个子进程的信息，显示在屏幕上。

(1)程序代码

```
//e204.c
#include<unistd.h>
#include <stdio.h>
#include <signal.h>

int pid1,pid2;

int main()
{
   int fd[2];
   char OutPipe[100], InPipe[100];

   pipe(fd);
   while((pid1=fork())==-1);
   if(pid1==0)
   {
      printf("p1\n");
      lockf(fd[1],1,0);
      sprintf(OutPipe,"Child 1 process is sending a message!");
      write(fd[1],OutPipe,50);
      sleep(1);
      lockf(fd[1],0,0);
      exit(0);
```

```
  }else{
    while((pid2=fork())==-1);
    if(pid2==0)
    {
      printf("p2\n");
      lockf(fd[1],1,0);
      sprintf(OutPipe,"Child 2 process is sending a message!");
      write(fd[1],OutPipe,50);
      sleep(1);
      lockf(fd[1],0,0);
      exit(0);
    }else{
      printf("parent\n");
      wait(0);
      read(fd[0],InPipe,50);
      printf("%s\n",InPipe);
      wait(0);
      read(fd[0],InPipe,50);
      printf("%s\n",InPipe);
      exit(0);
    }
  }

  return 0;
}
```

(2)上机操作

编辑、编译、运行、观察屏幕、记录结果。

2.消息的创建、发送和接收。

【预备知识】

(1) 消息通信

消息通信机制允许进程之间大批量交换数据。消息通信机制是以消息队列为基础的，消息队列是消息的链表。发送进程将消息挂入接收进程的消息队列，接收进程从消息队列中接收消息。消息队列有一个消息描述符。对消息队列的操作是通过描述符进行的。任何进程，只要有访问权并且知道描述符，就可以访问消息队列。

每个消息包括一个正长整型的类型字段，和一个非负长度的数据。进程读或写消息时，要给出消息的类型。若队列中使用的消息类型为0，则读取队列中的第一个消息。

(2) msgget()系统调用：

头文件 #include <sys/msg.h>

函数原型 int msgget(key_t key, int flag);

功能：创建消息队列，或返回与 key 对应的队列描述符。成功返回描述符，失败则返回-1。

参数：key 是通信双方约定的队列关键字，为长整型数。flag 是访问控制命令，它的低 9

位为访问权限（代表用户、组用户、其他用户的读、写、执行访问权），其它位为队列建立方式。

(3) msgsnd()系统调用：

头文件 #include <sys/msg.h>

函数原型 int msgsnd(int id, struct msgbuf *msgp,int size,int flag);

功能：发送一个消息。成功返回 0，失败返回-1。

参数：id 是队列描述符。msgp 是用户定义的缓冲区。size 是消息长度。flag 是操作行为，若(flag&IPC_NOWAIT)为真，调用进程立即返回；若(flag&IPC_NOWAIT)为假，调用进程睡眠，直到消息被发送出去或队列描述符被删除或收到中断信号为止。 缓冲区结构定义如下：struct msgbuf{ long mtype; char mtext[n]; };

(4) msgrcv()系统调用：

头文件 #include <sys/msg.h>

函数原型 int msgrcv(int id, struct msgbuf *msgp, int size,int type,int flag);

功能：接收一个消息。成功返回消息正文长度，失败返回-1。

参数：id 是队列描述符。msgp 是用户定义的缓冲区。size 是要接收的消息长度。type 是消息类型，若 type 为 0 则接收队列中的第一个消息，若 type 为正则接收类型为 type 的第一个消息。flag 是操作行为,若(flag&IPC_NOWAIT)为真，调用进程立即返回。若(flag&IPC_NOWAIT)为假，调用进程睡眠，直到接收到消息为止。

(5) msgctl()系统调用：

头文件 #include <sys/msg.h>

函数原型 int msgctl(int id, int cmd, struct msgid_ds *buf);

功能：查询消息队列描述符状态，或设置描述符状态，或删除描述符。成功返回 0，失败返回-1。

参数：id 是队列描述符。cmd 是命令类型，若 cmd 为 IPC_STAT，队列 id 的消息队列头结构读入 buf 中；若 cmd 为 IPC_SET,把 buf 所指向的信息复制到 id 的消息队列头结构中。若 cmd 为 IPC_RMID,删除 id 的消息队列。Buf 为消息队列头结构 msgid_ds 指针。

【任务 4】

使用系统调用 msgget()、msgsnd()、msgrcv()、msgctl()，编写消息发送和接收程序。要求消息的长度为 1KB。

(1) 参考程序

先定义消息结构，

```
struct msgbuf{
   long mtype;
   char mtext[n];
};
```

用这个结构定义消息缓冲全局变量 msg。定义消息队列描述符 msgqid。约定队列关键字为 75。

创建两个子进程 client 和 server。Client 使用 msgget()创建消息队列，使用 msgsnd()发送 10 条消息。Server 使用 msgget()获取消息队列描述符,然后用 msgrcv()

接收消息,完毕后删除队列描述符。为了清楚地显示 Client 发送的是哪条消息，每发送一条消息，打印消息号(消息类型)，Sever 每收到一条消息，也打印消息类型。

设计收发方式。Client 每发送一条，Sever 就接收一条。

```
/* 收发方式：Client()每发送一条消息，Server()就接收一条 */
/* 此方法不能保证一定能同步。对于不同速度的机器，如果没有其他耗时的进程，可以调整 sleep 的时间值而获得同步。*/
//msg.c
#include <stdio.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/ipc.h>

#define MSGKEY 75                          /* 通信双方约定的队列关键字*/

struct msgform                             /* 消息结构 */
{  long mtype;                             /* 消息类型 */
   char mtext[1030];                       /* 消息正文 */
}msg;
int msgqid;                                /* 消息队列描述符 */

void Client()
{  int i;  /* 局部变量 i，消息类型（表示第几条消息）*/
   msgqid=msgget(MSGKEY,0777); /*  创建消息队列，访问权限为 777 */
   for(i=10;i>=1;i--)
   {  msg.mtype=i;                         /* 指定消息类型 */
      printf("(client %d) sent.\n",i);  /* 打印消息类型 */
      msgsnd(msgqid,&msg,1024,0);
      /* 发送消息 msg 到 msgqid 消息队列,可以先把消息正文放到 msg.mtext 中*/
      sleep(1);  /* 等待接收。比较加上这一句和不加这一句的结果 */
   }
   exit(0);
}

void Server()
{  /* 获得关键字对应的消息队列描述符 */
   msgqid=msgget(MSGKEY,0777|IPC_CREAT);
   do {
      msgrcv(msgqid,&msg,1030,0,0); /* 从 msgqid 队列接收消息 msg */
      printf("(server  %d)received.\n",msg.mtype);     /*  打印消息类型
*/
   }while(msg.mtype!=1);              /* 消息类型为 1 时，释放队列*/
   msgctl(msgqid,IPC_RMID,0);         /* 删除消息队列 */
   exit(0);
```

```
}

void main()
{   int i;
   while((i=fork())==-1);              /* 创建子进程 */
   if(!i) Server();                    /* 子进程 Server */
   else
   {   while((i=fork())==-1);          /* 创建子进程 */
      if(!i) Client();                 /* 子进程 Client */
   }
   wait(0);                            /* 等待子进程结束 */
   wait(0);                            /* 等待子进程结束 */
}
```

(2)上机操作

编辑、编译、运行、观察屏幕、记录结果。

【实战任务】修改 msg.c 程序，Client 向 Server 发送消息"How much is it? "，Server 向 Client 响应消息"$2018."。

(3)课堂练习

修改上述程序，让 Client 向 Server 发送一个字符串“The message here is just a joke.”。Server 收到消息后打印出来。

5. 3 享存储的创建、附接和断接

【预备知识】

(1) 共享存储区

共享存储区机制直接通过共享虚拟存储空间进行通信。通信时，进程首先提出申请，系统为之分配存储空间并返回共享区标示符。这时，进程把它附加到自己的虚拟存储空间中。通信的进程对共享区的访问要互斥地进行。

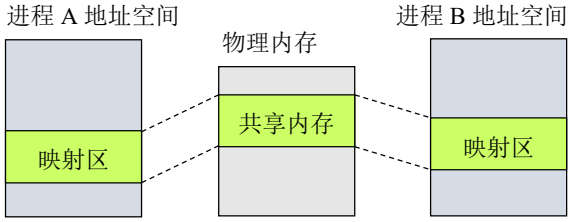


图 2.3 共享内存示意图

(2) shmget()系统调用：

头文件 #include <sys/shm.h>

函数原型 int shmget(key_t key, int size, int flag);

功能：申请一个共享存储区。成功返回共享内存标识符，失败则返回-1。

参数：key 是共享存储区关键字。size 是存储区大小。flag 访问权限和控制标志。

(3) shmat()系统调用：

头文件 #include <sys/shm.h>
函数原型 int shmat(int id, char *addr, int flag);
功能：将一个共享存储区附接到进程的虚地址空间。成功返回起始地址，失败则返回-1。
数：id 是共享存储区标识符。addr 是附接的虚地址。flag 访问权限和控制标志。

(4) shmdt()系统调用：
头文件 #include <sys/shm.h>
函数原型 int shmdt(char *addr);
功能：一个共享存储区与指定进程的断开。

(5) shmctl()系统调用：
头文件 #include <sys/shm.h>
函数原型 int shmctl(int id, int cmd, struct_ds* buf;
功能：共享存储区的控制操作。成功返回 0，失败则返回-1。
参数：id 是共享存储区标识符。cmd 为 IPC_STAT 共享存储的区的控制信息块读入 buf。flag 访问权限和控制标志。cmd 为 IPC_SET 则共享存储区的控制信息块读入 buf。cmd 为 IPC_RMID 则删除 shmid 指示的共享内存。flag 访问权限和控制标志。

【任务 5】
使用系统调用 shmget()、shmat()、shmdt()、shmctl()，编写两进程通过共享存储区进行通信的程序。

(1)参考程序
约定共享区关键字 75。创建两个子进程 client 和 server。Client 发送 10 条消息。Server 接收消息,完毕后删除共享区。

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

#define SHMKEY 75              /* 定义共享存储区关键词*/

int shmid,i;
int *addr;

void Client()
{  int i;
   shmid=shmget(SHMKEY,1024,0777);  /*  获取共享区,  长度 1024,关键词为 SHMKEY
*/
   addr=(int*)shmat(shmid,0,0);  /* 共享区的起始地址为 addr */
   for(i=9;i>=0;i--)
   { while(*addr!=-1);            /* 在这里做一个标号 A */
     printf("(client %d)sent\n",i);  /* 打印(client) sent */
```

```
    *addr=i;                          /* 把 i 赋给 addr 所指向的区域 */
  }
  exit(0);
}

void Server()
{  shmid=shmget(SHMKEY,1024,0777|IPC_CREAT);  /* 创建共享区 */
  addr=(int*)shmat(shmid,0,0);  /* 共享区的起始地址为 addr */
  do
  { *addr=-1;
    while(*addr==-1);             /* 等待发来信息，转到上面的标号 A; */
    printf("(server  %d)received!\n",*addr);    /*  服务进程使用共享区
*/
  }while(*addr);
  shmctl(shmid,IPC_RMID,0);
  exit(0);
}

int main()
{  int i;
  while((i=fork())==-1);
  if(!i) Server();
  else {
    while((i=fork())==-1);
    if(!i) Client();
  }
  wait(0);
  wait(0);

  return 1;
}
```

(2)上机操作

编辑、编译、运行、观察屏幕、记录结果。

2.3 调度算法

1. 处理器的调度

处理器的调度按层次分为三级:

(1) 高级调度: 又称为作业调度。

(2) 中级调度: 又称为平衡调度，概据主存资源决定主存中所能容纳的进程数，并根据进程的当前状态来决定辅助存储器和主存中的进程的对换。

(3) 低级调度: 又称进程调度/线程调度，根据某种原则决定就绪队列中的哪个进程/内核级线程获得处理器。

2. 调度机制的组成

调度机制由 3 个逻辑功能程序模块组成验证型:

(1) 队列管理程序

当一个进程/线程转换为就绪态时，其 PCB/TCB 会被更新以反映这种变化，队列管理程序将 PCB/TCB 指针放入等待 CPU 资源的进程/线程列表中，每当进程/线程移入就绪队列时， 可计算为此进程/线程分配 CPU 的优先级备用。

(2) 上下文切换程序

当调度程序把 CPU 从正在运行的进程/线程那里切换至另一个进程 /线程时，上下文切换程序将当前运行进程/线程的上下文信息保存至其 PCB/TCB 中，恢复选中进程/线程的上下文信息，从而使其占用处理器运行。

(3) 分派程序

当一个进程/线程让出 CPU 资源后，分派程序被激活，为了运行分派程序，需要将其上下文装入 CPU， 分派程序从就绪队列中选择进程/线程，而后完成从其自身到所选择的进程/线程间的又一次上下文切换，把 CPU 让给被选中的进程/线程。

3. 低级调度的基本类型

剥夺式: 又称抢先式。有两种常用的处理器剥夺原则：一是高优先级进程/线程可剥夺低优先级进程/线程；二是当运行进程/线程的时间片用完后被剥夺。

非剥夺式：又称非抢先式。一旦某个进程/线程开始运行后便不再让出处理器，除非此进程/线程运行结束，或主动放器处理器，或因某个事件而不能继续执行。

4. 作业调度和低级调度算法

(1) 先来先服务算法

非剥夺式

按照作业进入系统后备作业队列的先后顺序来挑选作业，先进入系统的作业将优先被挑选进入主存，创建用户进程，分配所需资源，然后移入就绪队列，每次调度时，先择最先进入此队列的进程/线程。

优点：易于实现。

缺点：效率不高，不利于短作业。

(2) 最短作业优先算法

非剥夺式。

以进入系统的作业所要求的 CPU 运行时间的长短为标准，总是选取预计计算时间最短的作业投入运行。

优点：易于实现

缺点：执行效率不高，主要弱点是：一，要预先知道作业所需要的 CPU 运行时间，很难精确估算，如果估值过低，系统可能提前终止此作；二，忽视作业的等待时间，有可能会使长作业处理饥饿状态；三，因缺少剥夺机制，对于分时、实时处理仍然不理想

(3) 最短剩余时间优先算法

剥夺式

当新进入的进程/线程的剩余时间比当前 CPU 中执行的进程的剩余时间还短，则会强行剥夺当前执行者的控制权

优点：短作业一进入系统，便可立即得到服务

缺点：可能会造成长作业发生饥饿现象

(4) 响应比最高者优先算法

非剥夺式

响应比＝作业周转时间 / 作业处理时间

＝ (作业等待时间 + 作业处理时间)/作业处理时间

每当调度作业时， 总是先计算响应比，将响应比最高者投入运行

优点：短作业容易得到较高的响应比，并且，如果长作业在系统中等待时间足够长，则长作业响应比比较大，即其优先级得以提高，从而可被选中执行，不会发生饥饿现象

缺点：每次计算各道作业的响应比会导致一定的时间开销，其性能比 SJF 算法略差

(5) 优先级调度算法

调度时，总是选择就绪队列中的优先级最高者投入运行。

分为静态优先级调度算法和动态优先级调度算法

a. 静态优先级调度算法：

非剥夺式

进程被选中调度后，直到它结束或出现等待事件而主动让出处理器，再高度另一个优先级高的进程/线程运行。

优点：实现简单

缺点：容易发生饥饿

b. 动态优先级调度算法：

正在运行的进程/线程随着占有 CPU 时间　的增加，逐渐降低其优先级，就绪队列中等待。CPU 的进程/线程随着等待时间的增加，逐渐提高其优先级，等待时间足够长的进程/线程会因其优先级不断提高而被调度运行

优点：克服了优先级低的进程出现饥饿的现象

(6) 轮转调度算法

调度程序生次把 CPU 分配给就绪队列首进程/线程使用规定的时间间隔，称为时间片，通常为 10ms~200ms，就绪队列中的每个进程/线程轮流地运行一个时间片，当时间片耗尽时，就强迫当前进/线程让出处理器，转而排列到就绪队列尾部，等候下一轮的调度

(7) 多级反馈队列调度算法

a. 由系统建立多个就绪队列,每个队列对应一个优先级,第一个队列优先级最后，其后的队列优先级逐个降低

b. 较高优先级队列的进程/线程给予的时间片较短,较低优先级队列给予的时间片较长，最后一个队列进程/线程按 FCFS 算法进行调度

c. 进程进入系统时，可以设置其优先级，将其放入指定优先级的队列中；也可以不设置优先级，进入系统中，将其放入优先级最高的第一个队列。

d. 调度时，首先从第一个队列中选取执行者，同一队列中的进程/线程按 FCFS 原则排队，只有在未选到时，才从较低一级的就绪队列中选取。仅当前面所有队列为空时，才会运行最后一个就绪队列

e. 如果在给定的时间片内完成，便可将进程从系统中撤离；否则，就移入下一个就绪队列的末尾，直至进入最低优先级就绪队列

优点：具有较好的性能，能满足各类应用的需要。

缺点：会导致饥饿问题，如一长作业进入系统，最终进入优先级最低的队列时，如果不断地有短作业进入系统，则会导致最低优先级队列中的长进程无法得到调度。

改良：对于低优先级的队列中等待时间足够长的进程提升其优先级。

【任务 1】

(1) 程序设计：设计一个调度算法，能够显示一个进程调度的方式，并且将其调度处理结果保存至文件中。

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

```
#include <unistd.h>

typedef struct pcb
{
    int pid;            /* Process ID */
    int prior;
    char    status[10];
    int time;
    struct  pcb *next;
} PCB;

FILE *fp=NULL;

int main()
{
    int iNumProc=0;
    int i = 0;

    PCB *stPCBHead=NULL;
    PCB *stPCB=NULL;
    PCB *stPCBFront=NULL;
    PCB *stPCBNode=NULL;

    char    filename[128];

    memset(filename, 0x00, sizeof(filename));

    sprintf(filename, "%s/program/schedule.txt", getenv("HOME"));
    if ( (fp=fopen(filename, "w")) == NULL)
    {
        printf("Create log file failed!\n");
        return -1;
    }

    /* Input the number of process */
    printf("Please input the number of process\n");
    scanf("%d", &iNumProc);

    /* Process number must great than 0 */
    while ( iNumProc <=0 )
    {
        printf("Ther number of process cann't be 0! \n");
        printf("Please input the number of process again\n");
        scanf("%d", &iNumProc);
```

```
    }

    for(i=0; i<iNumProc; i++)
    {
        /* Create process's node */
        stPCBNode=(PCB *)malloc(sizeof(PCB));
        if ( stPCBNode == NULL )
        {
            printf("malloc memory failed!\n");
            return -1;
        }
        memset(stPCBNode, 0x00, sizeof(PCB));

        /* Init process's information */
        stPCBNode->pid=i;       /* Process ID */
        printf("Please input ther prior of process[%d]\n", i);
        scanf("%d", &stPCBNode->prior); /* Process's prior */

        while ( stPCBNode->prior < 0 )
        {
            printf("Process's prior must great than 0\n");
            printf("Please input ther run time of process[%d]\n", i);
            scanf("%d", &stPCBNode->prior);
        }

        printf("Please input ther run time of process[%d][%d]\n", i, __LINE__);
        scanf("%d", &stPCBNode->time);

        while ( stPCBNode->time < 0 )
        {
            printf("Process's run time must great than 0\n");
            printf("Please input ther run time of process[%d]\n", i);
            scanf("%d", &stPCBNode->time);
        }

        strcpy(stPCBNode->status, "ready"); /* Init the process's status ready
*/
        stPCBNode->next = NULL;

        /* Link the new process's node */
        if ( stPCBHead == NULL )
        {
            stPCBHead = stPCBNode;
            continue;
```

```
        }

        stPCB=stPCBHead;
        stPCBFront=stPCBHead;

        while ( stPCB != NULL && stPCB->prior >= stPCBNode->prior )
        {
            stPCBFront = stPCB;
            stPCB=stPCB->next;
        }

        if ( stPCB == stPCBHead )
        {
            stPCBNode->next = stPCBHead;
            stPCBHead=stPCBNode;
        }
        else if ( stPCB == NULL )
        {
            stPCBFront->next = stPCBNode;       /* Add to the tail of the link
*/
        }
        else                    /* insert into the link */
        {
            stPCBNode->next = stPCB;
            stPCBFront->next=stPCBNode;
        }

        printf("Create the process [%d] success\n", i);
    }
    fprintf(fp, "Create %d processes success\n", iNumProc);


    printf("In the begin of schedule, these process's queque\n");
    fprintf(fp, "Before Schedule\n");
    printProc(stPCBHead);

    sleep(1);

    /* Now Schedual */
    printf("Begin schedule\n");
    fprintf(fp, "Begin schedule\n");
    stPCB=stPCBHead;
    while ( iNumProc > 0 )
    {
```

```
        /* schedule from first process */
        if ( strcmp(stPCBHead->status, "ready") == 0 )
            strcpy(stPCBHead->status, "running");
        printProc(stPCBHead);

        stPCBHead->time--;
        stPCBHead->prior--;

        /* Update the the level of proccess which in wait status */
        for (stPCB=stPCBHead; stPCB != NULL; stPCB=stPCB->next)
        {
            if ( strcmp(stPCB->status, "ready") == 0 )
            {
                stPCB->prior++;
            }
        }

        /* sort the schedule again */
        sort(&stPCBHead, &iNumProc); /* Sort the link and delete which the
process's time is 0 */
    }

    return 0;
}

int printProc(PCB *pstPCBHead)
{
    PCB *pstPCB=pstPCBHead;
    printf("-----------------------------------------------------------\n");
    printf("| Process's ID    | Process's prior| Process's Stauts |The time
left|Current Process Addr|Next Process Addr|\n");
    printf("-----------------------------------------------------------\n");


    fprintf(fp, "-----------------------------------------------------------\n");
    fprintf(fp, "| Process's ID   | Process's prior| Process's Stauts |The time
left|Current Process Addr|Next Process Addr|\n");
   fprintf(fp, -----------------------------------------------------------\n");
    while ( pstPCB != NULL )
    {
        sleep(1);

        printf("|%16d|%16d|%18s|%13d|%20d|%17d|\n", pstPCB->pid,
            pstPCB->prior, pstPCB->status, pstPCB->time, pstPCB, pstPCB->next);
```

```
        printf("-------------------------------------------------\n");

        fprintf(fp, "|%16d|%16d|%18s|%13d|%20d|%17d|\n", pstPCB->pid,
            pstPCB->prior, pstPCB->status, pstPCB->time, pstPCB, pstPCB->next);
        fprintf(fp, "-----------------------------------------\n");
        pstPCB=pstPCB->next;

    }

    printf("\n\n");
    fprintf(fp, "\n\n");

    return 0;
}

int sort(PCB **pstPCBHead, int *iNumProc)
{
    PCB *pstPCB=NULL;
    PCB *pstPCBFront=NULL;
    int flag=0;

    pstPCB=(*pstPCBHead)->next;
    pstPCBFront=(*pstPCBHead);

    if ( (*pstPCBHead)->time == 0 )
    {
        printf("Proccess [%d] run end!\n", pstPCBFront->pid);
        (*pstPCBHead) = (*pstPCBHead)->next;
        free(pstPCBFront);
        pstPCBFront=NULL;
        (*iNumProc)--;
        return 0;
    }

    if ( (*iNumProc) <= 1 )
    {
        return 0;
    }


    while ( pstPCB != NULL && (*pstPCBHead)->prior <= pstPCB->prior )
    {
        pstPCBFront = pstPCB;
        pstPCB=pstPCB->next;
```

```
        flag=1;
    }

    if ( pstPCB == NULL && flag==1)
    {
        strcpy((*pstPCBHead)->status, "ready");
        pstPCBFront->next = (*pstPCBHead);
        *pstPCBHead = (*pstPCBHead)->next;
        pstPCBFront->next->next = NULL;
    }
    else if ( flag== 1)
    {
        strcpy((*pstPCBHead)->status, "ready");
        pstPCBFront->next = (*pstPCBHead);
        (*pstPCBHead)= (*pstPCBHead)->next;
        pstPCBFront->next->next=pstPCB;
    }

    return 0;
}
```

(2)上机操作

用 gedit 输入源代码。

编译、运行，观察屏幕，记录结果。

思考:请先运行此程序，查看运行结果，分析此调度属于哪种调度算法。描述此种调度算法的调度过程。

2.4 P、V 原语应用程序

1.生产者－消息者问题

有 n 个生产者和 m 个消费者,共用 k 个缓冲区。生产者将产品生产出来后，放入空缓冲区中，消息者从缓冲区中取出产品消费。当缓冲区满时，生产者不能把产品放入缓冲区，必须停止生产；当缓冲区空时，消息者不能从缓冲区中取得产品。如图 2.4 所示。

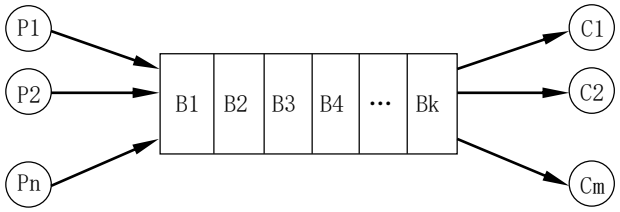


图 2.4 生产者-消费者共享缓冲区示意图

生产者和消费者必须要满足如下条件：

1) 消费者想要接收数据，有界缓冲区中至少有一个单元是满的。

2) 生产者想要发送数据，有界缓冲区中至少有一个单元是空的。

3) 各生产者和各消费者进程之间必须互斥使用缓冲区。

2.信号量

信号量 sem 是一整数，在 sem 大于零时可代表可供并发进程使用的资源实体数，但 sem 小于零时，则表示正在等待使用临界区的进程数。建立一个信号量必须经过说明所建信号量所代表的意义，和赋初值以及建立相应的数据结构以便指向那些等待使用该临界区的资源。

信号量只有三种操作：赋值，P、V 操作。

P 原语的操作的主要动作是：

1）sem 减 1；

2) 若 sem 减 1 后仍大于或等于零，则进程继续执行

3) 若 sem 减 1 后小于零，则该进程被阻塞后与该信号相对应的队列中，然后转进程调度

V 原语操作的主要动作是：

1）sem 加 1；

2）若相加结果大于零，则进程继续执行

3）若相加结果小于零，则从该信号的等待队列中唤醒一等待队列中一等待进程，然后再返回原进程继续执行或转调度进程。

私用信号量与公用信号量的区别：

公用信号量：用于整组并发进程互斥时，称为公用信号量

私用信号量：用于制约进程与被制约进程的信号量，制约进程发送到被制约进程执行所需要的消息。

3.Linux 下信号量的实现

Linux 中所有信号量函数接口提供的是通用接口，即信号量标识引用的是一组由独立信号量组成的信号量集合，而不是单个的信号量。如果我们只需对单个信号量进行操作，那么只需指定由一个信号量组成的集合；但如果我们需要用一个操作请求多个信号量时，就要指定由多个信号量组成的集合。

每一个信号量集合有一个类型为 semid_ds 的数据结构与之相连，该结构含有 IPC 许可权限、各个信号量本身以及它们的访问信号。

```
struct  semid_ds
{
    struct  ipc_perm *sem_per;/* 操作许可权限数据结构指针 */
    struct sem *sem_base;     /* 指向信号量集合的指针                    */
    unsigned short int sem_nsems;  /* 集合中的信号量个数                 */
    time_t sem_otime;        /* 最后一次操作的时间                       */
    time_t sem_ctime;       /* 最后一次改变此结构的时间                   */
}
struct  sem
{
    unsigned short semval;  /* 信号量的当前值                           */
    pid_t sempid;           /* 最后操作该信号量的进程 ID */
    unsigned short semncnt;  /* 等待对该信号量执行 P 操作的进程数 */
    unsigned short semzcnt;   /* 等待 semval 为 0 的进程数              */
}
```

(1) 信号量的初始化

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

功能：创建或获得与 key 关键字关联的信号量集合

参数：key --是 IPC 的关键字，它有一个特殊值 IPC_PRIVATE， 这表示总是创建一个私有信号量集合

nsems--指明信号量集合中信号量的个数。

semflg--若 key 不为 IPC_PRIVATE 时，semflg 的标志位可为以下值：

IPC_CREAT:若单独设置此标志，当系统中不存在相同的 key 值的信号量集合时，则创建一个新的信号量集合，否则返回已存在的信号量集合标识。

IPC_EXCL：当与 IPC_CREAT 同时设置时，若系统中已存在与相同 key 值的信号量集合时，则返回错误。

semget 调用成功返回一个非负整数，即与 key 相连的信号量集合 id

(2) 信号量的控制

int semctl(int semid, int semnum, int cmd, [union semum arg])

功能：对 semid 指点的信号量集合中的编号为 semnum 的信号量执行 cmd 说明操作，

参数：semid—由 semget 返回的 id，指明信号量集合的 id

semnum--指明一个信号量集合中的一个特定的信号量,它指明这个信号量的编号。

arg--是一个类型为 semun 的联合体，若需要时，应用程序必须这样定义它

```
union semun
{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
} arg;
```

cmd—semctl 所能执行的操作，如表 2.1 所示。

表 2.1 信号量控制常量

命令	描述	注
SETVAL	设置单个信号量的值	arg.val
GETALL	返回信号量集合中所有信号量的值	arg.array
SETALL	设置信号量集合中所有信号量的值	arg.array
IPC_STAT	放置与信号量集合相连的 semid_ds 结构当前值于 arg.buf 指定的缓冲区	arg.buf
IPC_SET	用 arg.buf 指定结构值替代与信号量集合相连的 semid_ds 结构值	arg.buf
GETVAL	返回单个信号量的值	
GETPID	返回最后一个操作该信号量集合的进程 ID	
GETNCNT	返回 semncnt 之值	
GETZCNT	返回 semzcnt 之值	
IPC_RMID	删除指定的信号量集合	

其中 arg 是一个类型为 semun 的联合体,若需要时，应用程序必须这样定义它

```
union semun
{
    int val;
    struct semid_ds *buf;
```

```
    unsigned short *array;
} arg;
```

其中

val　用于 SETVAL 命令，指明要设置的信号量的值

buf　用于 IPC_STAT/IPC_SET 命令，表示存放信号量集合数据结构的缓冲。

array 用于 GETALL/SETALL 命令，存放所获得的或要设置的信号量集合中所有信号量的值。

(3) 信号量操作

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

功能: 对指定的信号量集合进行操作，既可以是针对单个信号量，也可以针对整个信号量集合

参数: semid--合法的信号量标识

sops--指向一个类型为 sembuf 的结构体数组的指针,该数组的每一个元素给出关于一个信号量的操作信息，这些信息用 sembuf 结构对象表示

```
struct sembuf
{
    unsigned short int sem_num;        /* 信号量编号          */
    short int sem_op;                  /* 信号量操作          */
    short int sem_flg;                 /* 操作标志            */
}
```

sem_num 给出信号量的编号,sem_op 给出操作类型,sem_op<0 时,减少一个信号量的值，减少的值为 abs(sem_op)，当 sem_od 为-1 时，相当于 P 操作；sem_op>0 时，增加一个信号量的值，所增加的值为 sem_op，当 sem_op 为 1 时，相当于 V 操作。当 sem_op==0 时，等待信号量变为 0，这对应于等待信号量控制的所有资源均可用，如果信号量已经为 0，调用立即返回，否则当没有设置 IPC_NOWAIT 时，调用被阻塞。

sem_flg--用于对操作进行适当的控制，可以指定如下两个控制标志：

IPC_NOWAIT:如果设置，当指定的操作不能完成时，进程将不等待而是立即返回，

IPC_UNDO:如果设置，当进程退出时，执行信号量解除操作。

生产者与消费者

(4) 单缓冲区生产者-消费者问题

```
semaphore  empty=1, full=0;
process producter()                    process consumer()
{                                      {
    while(true)                            while(true)
```

```
    {
        produce();
        P(empty);
        append to buffer;
        V(full);
    }
}
```

```
    {
        P(full);
        take from buffer;
        V(empty);
        consume();
    }
}
```

(5) m 个生产者 n 个消费者共享 k 件产品缓冲区的问题

```
semaphore empty=k; full=0;
semaphore mutex; mutex1=1;
process producter()
{
    while(true)
    {
        produce();
        P(empty);
        P(mutex);
        append to buffer;
        in=(in+1) % k;
        V(mutex);
        V(full);
    }
}
```

```
process consumer()
{
    while(true)
    {
        P(full);
        P(mutex);
        take from buffer;
        out=(out+1) % k;
        V(mutex);
        V(empty);
        consume();
    }
}
```

【任务 1】

(1) 思考：在上述单缓冲区生产者－消费者模型中，消费者进程 consumer()的 V(empty)和 consume()能否互换位置，并说明理由。

答案：不能互换位置，因为这会引起效率问题。当消费者将产品从缓冲区中取出后（即 P(full)），即可释放缓冲区(V(empty))，以让生产者可以继续往缓冲区中放入产品，而不必等到消费者消费(consume())完后，再释放缓冲区，否则，因为消费者消费速度太慢，将会导致有可用的缓冲区时，生产者仍不能生产。

(2) 思考：在 m 个生产者和 n 个消费者共享 k 个缓冲区时，能否将生产者的 P(empty)和 P(mutex)互换位置，并说明理由。能否将生产者的 V(mutex)和 V(full)互换位置，并说明理由

答案：不能将生产者的 P(empty)和 P(mutex)互换位置。否则，将会导致生产者将 buffer 锁住(P(mutex))后，执行 P(empty)时，因没有足够的 empty 缓冲区，而导致生产者进程阻塞，并且其它生产者与消费者会因缓冲区锁住而无法继续执行发生阻塞，导致死锁。从效率上讲，不能将 V(mutex)和 V(full)互换位置，因为首先执行 V(mutex)解锁操作后，将会唤醒与此缓冲区相关的等待队列进程中的一个。

【任务 2】

（1） 程序设计

用信号量及共享存储实现 m 个生产者 n 个消费者共享 k 个缓冲区的程序

```
/* 文件名: muti_producer.c
 * 功能:   生产者
 * 模拟多个生产者时，只要将该文件编译后的可执行程序在多个终端分别执行即可
 */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <errno.h>

#define  SEMNUM  3
#define  SHMSIZE 900
typedef union semun
{
 int val;
 struct semid_ds *buf;
 short int *array;
} SEMUN;

int main()
{
 int semid = 0;
 int shmid = 0;
 int key = 0;
 int num_of_production = 0;      /* Num Of Production */
 int pos=0;
 char *AppendFirstAddr=NULL;
 int i=0;

 char buffer[SHMSIZE];
 char *shmaddr=NULL;
```

```
short int array[SEMNUM];
memset(buffer, 0x00, sizeof(buffer));
memset(array, 0x00, sizeof(array));

printf("<---------------------------------------->\n");
printf("The producer's ID is [%d]\n", getpid());

if ( (key=GetKey()) < 0 )
{
    printf("Get Key Failed\n");
    return -1;
}
printf("key[%d]\n", key);

if ( GetSemCollection(key, &semid) < 0)
{
    printf("Get Sem Failed[%d]\n", errno);
    return -1;
}
printf("semid[%d]\n", semid);

if ( GetShm(key, &shmid, &shmaddr) < 0 )
{
    printf("Get Shm Failed[%d]\n", errno);
    return -1;
}
printf("shmid[%d], shmaddr[%d]\n", shmid, shmaddr);
printf("<---------------------------------------->\n");

/* Init the shm's first sizeof(int) unit to 0 */
memcpy(&pos, shmaddr, sizeof(int));
AppendFirstAddr=shmaddr+2*sizeof(int);
printf("AppendFirstAddr[%s] pos[%d]\n", AppendFirstAddr, pos);

while(1)
{
    memset(buffer, 0x00, sizeof(buffer));
    printf("\n");
    /* Produce Production */

    if ( semctl(semid, 0, GETALL, array) < 0 )
    {
        printf("Get Sem Val Failed\n");
        return -1;
```

```
    }

    printf("The resource of the system's list:\n");
    for ( i =0; i < SEMNUM; i++)
    {
        if ( i==0 )
        {
            printf("empty position=[%d]\n",  array[i]);
        }
        else if ( i==1 )
        {
            printf("full position=[%d]\n",  array[i]);
        }
        else if ( i==2 )
        {
            printf("mutex=[%d]\n", array[i]);
        }
    }
    printf("\n");

    if ( Produce(buffer) < 0 )
    {
        printf("Get Shm Failed[%d]\n", errno);
        return -1;
    }

    if ( strncmp(buffer, "quit", 4)  == 0 )
    {
        printf("All producer's Sem and shm will be over\n");
        break;
    }

    num_of_production = strlen(buffer);
    printf("---->Producer[%d] produce [%d] productions:[%s]<----\n\n",
        getpid(), num_of_production, buffer);

    printf("Producer[%d]  Requiring  [%d]  shop's  position  to  put  the
production\n", getpid(), num_of_production);

    /* Get Empty Shared Memory */
    if ( P(semid, "empty", num_of_production) < 0 )
    {
        printf("P full Operation Failed\n");
        return -1;
```

```
    }
    printf("Producer [%d] Get [%d] position of the shop\n", getpid(),
num_of_production);

    printf("Producer[%d] Requring the right(mutex) to put the production
to the position... \n", getpid());

    /* Mutex */
    if ( P(semid, "mutex", 1) < 0 )
    {
        printf("P mutex Operation Failed\n");
        return -1;
    }

    printf("Producer[%d] Get the right(mutex) to put the production to
right position\n", getpid());

    memcpy(&pos, shmaddr, sizeof(int));
    printf("----->pos[%d]<------\n", pos);
    if ( Append(AppendFirstAddr, buffer, pos) < 0 )
    {
        printf("Append Production To Shared Memory Failed\n");
        return -1;
    }

    printf("Producer [%d] put the production [%s] to the position of the
shop\n", getpid(), buffer);
    printf("(The production of the shop is )Shared Memory Content[%s]\n",
AppendFirstAddr);

    pos = (pos+ num_of_production) % SHMSIZE;
    memcpy(shmaddr, &pos, sizeof(int));

    V(semid, "mutex", 1);
    printf("Producer [%d] release the operation right(mutex) of the
position of the shop\n", getpid());

    V(semid, "full", num_of_production);
    printf("Now Consumer can take [%d] more of production[%s] to
consume\n", num_of_production, buffer);
 }

 semctl(semid, 0, IPC_RMID);
 shmdt(shmaddr);
```

```
 if ( shmctl(shmid, IPC_RMID, NULL) < 0 )
 {
     printf("RM Shm Failed[%d]\n", errno);
     return -1;
 }

 return 0;
}

int GetKey()
{
 char    filepath[128];
 char    cmd[128];
 int key  = 0;

 memset(filepath, 0x00, sizeof(filepath));
 memset(cmd, 0x00, sizeof(cmd));

 sprintf(filepath, "%s/key.ini", getenv("HOME"));

 sprintf(cmd, "touch %s", filepath);

 if ( system(cmd) < 0 )
 {
     printf("Create Key File Failed\n");
     return -1;
 }

 if ( (key=ftok(filepath, 20)) < 0 )
 {
     printf("Create Key Failed\n");
     return -1;
 }

 return key;
}

int GetSemCollection(int key, int *semid)
{
 int lsemid=0;
 SEMUN   unsem;
 short int array[3];
 if ( (lsemid=semget(key, SEMNUM, IPC_CREAT|0666)) < 0 )
 {
```

```
    printf("Get Sem ID Failed\n");
    return -1;
 }

 array[0] = SHMSIZE;          /* Empty Buffer Size */
 array[1] = 0;                /* Full Buffer Size  */
 array[2] = 1;                /* For Mutex         */

 unsem.array=array;
 if ( semctl(lsemid, 0, SETALL, unsem) < 0 )
 {
    printf("Init Sem Failed\n");
    return -1;
 }

 *semid=lsemid;

 return 0;
}

int GetShm(int key, int *shmid, void **shmaddr)
{
 int lshmid = 0;
 int shmsize=SHMSIZE + 2*sizeof(int);
 char *lshmaddr = NULL;

 /* shmsize=SHMSIZE+2*sizeof(int);*/
 if ( (lshmid=shmget(key, shmsize, IPC_CREAT | 0666)) < 0 )
 {
    printf("Get Shared Memory Failed\n");
    return -1;
 }

 if ( (lshmaddr=(char *)shmat(lshmid, 0, 0)) == (char *) -1 )
 {
    printf("Attach Shared Memory Failed, PID[%d]\n", getpid());
    return -1;
 }

 *shmid = lshmid;
 (char *)*shmaddr = (char *)lshmaddr;

 return 0;
}
```

```
int Produce(char *buf)
{
 if ( buf == NULL )
 {
     printf("Produce Buffer Couldn't Be NULL\n");
     return -1;
 }

 printf("Producer[%d] Produce Production(Please Input):\n");
 gets(buf);

 return 0;
}

int P(const int semid, char *PType, int len)
{
 struct sembuf   stSembuf;
 memset(&stSembuf, 0x00, sizeof(stSembuf));

 if ( PType == NULL )
 {
     printf("PType Cann't Be NULL\n");
     return -1;
 }

 if ( strcmp(PType, "empty") == 0 )
 {
     stSembuf.sem_num = 0;
 }
 else if ( strcmp(PType, "full") == 0 )
 {
     stSembuf.sem_num = 1;
 }
 else if ( strcmp(PType, "mutex") == 0 )
 {
     stSembuf.sem_num = 2;
 }
 else
 {
     printf("Sem Type Wrong\n");
     return -1;
 }
```

```
 stSembuf.sem_op = (-1)*len;
 stSembuf.sem_flg = SEM_UNDO;

 if ( semop(semid, &stSembuf, 1) < 0)
 {
     printf("P [%s] Operation Failed\n", PType);
     return -1;
 }

 return 0;
}

int V(const int semid, char *PType, int len)
{
 struct sembuf   stSembuf;
 memset(&stSembuf, 0x00, sizeof(stSembuf));

 if ( PType == NULL )
 {
     printf("PType Cann't Be NULL\n");
     return -1;
 }

 if ( strcmp(PType, "empty") == 0 )
 {
     stSembuf.sem_num = 0;
 }
 else if ( strcmp(PType, "full") == 0 )
 {
     stSembuf.sem_num = 1;
 }
 else if ( strcmp(PType, "mutex") == 0 )
 {
     stSembuf.sem_num = 2;
 }
 else
 {
     printf("Sem Type Wrong\n");
     return -1;
 }

 stSembuf.sem_op = len;
 stSembuf.sem_flg = SEM_UNDO;
```

```
 if ( semop(semid, &stSembuf, 1) < 0)
 {
     printf("P [%s] Operation Failed\n", PType);
     return -1;
 }

 return 0;
}

int Append(void *AppendFirstAddr, char *buffer, int pos)
{
 char *tmp=(char *)AppendFirstAddr;
 if ( AppendFirstAddr == NULL || buffer == NULL )
 {
     printf("Parameter cann't be NULL[%d]\n", __LINE__);
     return -1;
 }

 printf("---%d--[%d]---\n", pos, __LINE__);
 sprintf(tmp+pos, "%s", buffer);

 return 0;
}

/* 文件名: muti_consumer.c
 * 功能:   消费者
 * 模拟多个消费者时,只要将该文件编译后的可执行程序在多个终端分别执行即可
 */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <errno.h>

#define  SEMNUM  3
#define  SHMSIZE 900
typedef union semun
{
 int val;
 struct semid_ds *buf;
 short int *array;
} SEMUN;
```

```
int main()
{
 int semid = 0;
 int shmid = 0;
 int key = 0;
 int num_to_buy = 0;     /* Num Of Production */
 int pos=0;
 int i=0;
 short int array[SEMNUM];

 char buffer[SHMSIZE];
 char *shmaddr=NULL;
 char *ConsumeFirstAddr=NULL;
 memset(buffer, 0x00, sizeof(buffer));
 memset(array, 0x00, sizeof(array));

 printf("Consumer's ID is[%d]\n", getpid());

 if ( (key=GetKey()) < 0 )
 {
     printf("Get Key Failed\n");
     return -1;
 }

 printf("key[%d]\n", key);

 if ( GetSemCollection(key, &semid) < 0)
 {
     printf("Get Sem Failed[%d]\n", errno);
     return -1;
 }
 printf("semid[%d]\n", semid);

 if ( GetShm(key, &shmid, &shmaddr) < 0 )
 {
     printf("Get Shm Failed[%d]\n", errno);
     return -1;
 }
 printf("shmid[%d] shmaddr[%d]\n", shmid, shmaddr);

 memcpy(&pos, shmaddr+sizeof(int), sizeof(int));
 ConsumeFirstAddr=shmaddr+2*sizeof(int);
 printf("=======pos[%d],          ConsumeFirstAddr[%s]===\n",          pos,
ConsumeFirstAddr);
```

```
while(1)
{
    printf("\n\n");
    memset(buffer, 0x00, sizeof(buffer));

    if ( semctl(semid, 0, GETALL, array) < 0 )
    {
        printf("Get Sem Val Failed\n");
        return -1;
    }

    for ( i=0; i < SEMNUM; i++)
    {
        if ( i==0 )
        {
            printf("empty=[%d]\n", array[i]);
        }
        else if ( i==1 )
        {
            printf("full=[%d]\n", array[i]);
        }
        else if ( i==2 )
        {
            printf("mutex=[%d]\n", array[i]);
        }
    }

    printf("Please input the num of production you want to buy:");
    scanf("%d", &num_to_buy);
    if ( num_to_buy<0 || num_to_buy>900)
    {
        printf("The num input is wrong,it must between 0 and 100\ n");
        continue;
    }

    /* Get Empty Shared Memory */
    printf("Consumer[%d] Now Requiring [%d] production to buy\n",
        getpid(), num_to_buy);
    if ( P(semid, "full", num_to_buy) < 0 )
    {
        printf("P full Operation Failed\n");
        return -1;
    }
```

```
    printf("Consumer[%d] can buy [%d] productions \n", num_to_buy);
    printf("Consumer[%d] waiting for buying the productions\n",
        getpid());
    /* Mutex */
    if ( P(semid, "mutex", 1) < 0 )
    {
        printf("P mutex Operation Failed\n");
        return -1;
    }
    printf("Now it's turn of the consumer[%d] to buy the productions\n" ,
getpid());
    memcpy(&pos, shmaddr+sizeof(int), sizeof(int));

    printf("----->pos[%d]       num_to_buy[%d]   shmaddr[%s]\n",   pos,
num_to_buy, shmaddr+2*sizeof(int));
    if ( Buy(buffer, ConsumeFirstAddr, pos, num_to_buy ) < 0 )
    {
        printf("Buy Production Failed\n");
        return -1;
    }

    printf("Consumer [%d] Buy Production[%s] shmaddr[%s]\n", getpid(),
buffer, shmaddr+2*sizeof(int));

    pos = (pos + num_to_buy) % SHMSIZE;
    memcpy(shmaddr+sizeof(int), &pos,sizeof(int));

    V(semid, "mutex", 1);
    printf("Consumer [%d] leave the shop\n", getpid());
    V(semid, "empty", num_to_buy);
   printf("Now the shop has [%d] more position to put the productions\n",
num_to_buy);
 }

 return 0;
}

int GetKey()
{
 char    filepath[128];
 char    cmd[128];
 int key  = 0;

 memset(filepath, 0x00, sizeof(filepath));
```

```
 memset(cmd, 0x00, sizeof(cmd));

 sprintf(filepath, "%s/key.ini", getenv("HOME"));

/*
 sprintf(cmd, "touch %s", filepath);

 if ( system(cmd) < 0 )
 {
     printf("Create Key File Failed\n");
     return -1;
 }
*/

 if ( (key=ftok(filepath, 20)) < 0 )
 {
     printf("Create Key Failed\n");
     return -1;
 }

 return key;
}

int GetSemCollection(int key, int *semid)
{
 int lsemid=0;
 SEMUN   unsem;
 short int array[3];
 if ( (lsemid=semget(key, SEMNUM, IPC_CREAT|0666)) < 0 )
 {
     printf("Get Sem ID Failed\n");
     return -1;
 }

#if 0
 array[0] = SHMSIZE;         /* Empty Buffer Size */
 array[1] = 0;               /* Full Buffer Size  */
 array[2] = 1;               /* For Mutex         */

 unsem.array=array;
 if ( semctl(lsemid, 0, SETALL, unsem) < 0 )
 {
     printf("Init Sem Failed\n");
     return -1;
```

```
 }
#endif

 *semid=lsemid;

 return 0;
}

int GetShm(int key, int *shmid, void **shmaddr)
{
 int lshmid = 0;
 char *lshmaddr = NULL;

 if ( (lshmid=shmget(key, SHMSIZE+2*sizeof(int), IPC_CREAT | 0666)) < 0 )
 {
     printf("Get Shared Memory Failed\n");
     return -1;
 }

 if ( (lshmaddr=(char *)shmat(lshmid, 0, 0)) == (char *) -1 )
 {
     printf("Attach Shared Memory Failed, PID[%d]\n", getpid());
     return -1;
 }

 *shmid=lshmid;
 (char *)*shmaddr=(char *)lshmaddr;

 return 0;
}

int Produce(char *buf)
{
 if ( buf == NULL )
 {
     printf("Produce Buffer Couldn't Be NULL\n");
     return -1;
 }

 printf("Please Produce Production:\n");
 gets(buf);
 printf("\nProducer PID[%d] Has Produce Something [%s]\n", buf);

 return 0;
```

```
}

int P(const int semid, char *PType, int len)
{
 struct sembuf   stSembuf;
 memset(&stSembuf, 0x00, sizeof(stSembuf));

 if ( PType == NULL )
 {
     printf("PType Cann't Be NULL\n");
     return -1;
 }

 if ( strcmp(PType, "empty") == 0 )
 {
     stSembuf.sem_num = 0;
 }
 else if ( strcmp(PType, "full") == 0 )
 {
     stSembuf.sem_num = 1;
 }
 else if ( strcmp(PType, "mutex") == 0 )
 {
     stSembuf.sem_num = 2;
 }
 else
 {
     printf("Sem Type Wrong\n");
     return -1;
 }

 stSembuf.sem_op = (-1)*len;
 stSembuf.sem_flg = SEM_UNDO;

 if ( semop(semid, &stSembuf, 1) < 0)
 {
     printf("P [%s] Operation Failed\n", PType);
     return -1;
 }

 return 0;
}

int V(const int semid, char *PType, int len)
```

```
{
 struct sembuf   stSembuf;
 memset(&stSembuf, 0x00, sizeof(stSembuf));

 if ( PType == NULL )
 {
     printf("PType Cann't Be NULL\n");
     return -1;
 }

 if ( strcmp(PType, "empty") == 0 )
 {
     stSembuf.sem_num = 0;
 }
 else if ( strcmp(PType, "full") == 0 )
 {
     stSembuf.sem_num = 1;
 }
 else if ( strcmp(PType, "mutex") == 0 )
 {
     stSembuf.sem_num = 2;
 }
 else
 {
     printf("Sem Type Wrong\n");
     return -1;
 }

 stSembuf.sem_op = len;
 stSembuf.sem_flg = SEM_UNDO;

 if ( semop(semid, &stSembuf, 1) < 0)
 {
     printf("P [%s] Operation Failed\n", PType);
     return -1;
 }

 return 0;
}

int Buy(void *outbuffer, void *ConsumeFirstAddr, int pos, int num)
{
 int i = 0;
 char *tmp=(char *)ConsumeFirstAddr;
```

```
    if ( outbuffer == NULL )
    {
        printf("Parameter 'outbuffer' cann't be NULL\n");
        return -1;
    }

    memcpy(outbuffer, tmp+pos, num);

    for(i=0; i<num; i++)
    {
        tmp[pos+i]='*';
    }

    return 0;
}
```

（2） 上机操作

用 gedit 输入源代码。

编译、链接程序，生成可执行文件。

打开 5 个终端，其中 3 个终端执行 muti_producer，2 个终端执行 muti_consumer。

按程序提示操作

观察程序的结果

3 存储管理

3.1 页面置换算法

1 预备知识

页面置换算法也叫页面替换算法。

1.1 请求页式虚拟内存管理页面置换原理

我们已经知道，进程运行前须全部装入内存。

采用虚拟内存管理技术，进程不必全部装入内存，而只要装入一部分就可以运行。这是怎么实现的呢？

(1) 分页管理原理

把内存物理空间划分为长度相等的块，每一块叫做一个**页框（也叫页帧）**，用同样的块大小划分进程逻辑空间，每一块叫做一**页**（图 3.1）。进程装入内存时，以页为单位装入。由于页长等于页框长，一页恰好可以放到一个页框中。

进程放入内存时，可以不连续存放。那么系统怎么知道某个页框是属于哪个进程的页呢？

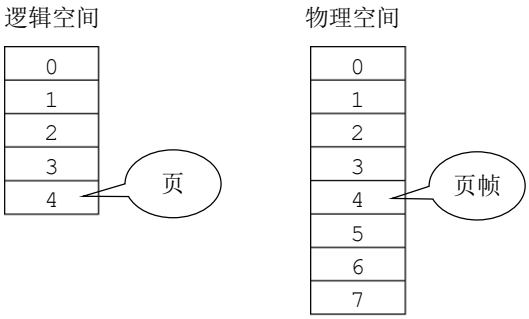


图 3.1　页与页框

(2) 页表

某页装入内存时，页与页框的对应关系记录在**页表**中。CPU 访问某一页时，只要查页表就可以知道对应的页框，然后到该页框中取指令或操作数(图 3.2)。

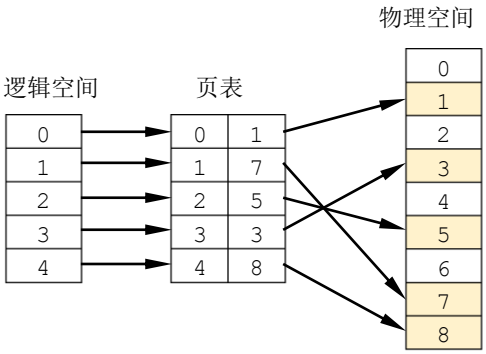


图 3.2　页表

(3) 虚拟存储管理

思想：进程不是一次性地全部装入内存，而只装入将要执行和调用的部分。其他部分则留在磁盘上，在需要的时候再动态装入。

请求分页虚拟存储管理思想：进程执行时，只装入将要访问的一些页，其它页放在磁盘上。当需要访问那些其它页时，再从磁盘装入内存（图 3.3）。

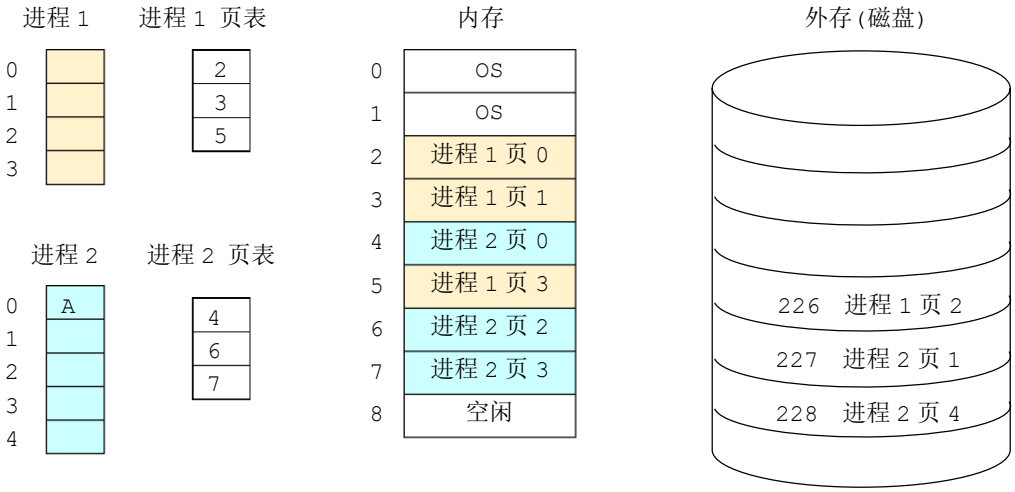


图 3.3 请求分页虚拟存储管理

请求分页虚拟存储管理过程：进程访问某一页时，系统先查页表，若页表中有该页，则可以访问相应的页框。若页表中找不到该页，说明该页不在内存中而在磁盘上，这时，会产生**缺页中断**，然后由缺页中断处理程序从磁盘调入相应页，并修改页表。调入新页时，会有两种情况：

情况 1，有足够内存。直接调入新页即可。

情况 2，内存不够。这时，需要移走某些页，以腾出空间，然后调入新页到腾出的空间。这个过程称之为**页面置换（页面替换）**。当要置换一页时，移走那一页合适呢？这取决于采用什么置换算法。常用的置换算法有先进先出算法（FIFO）、最佳页面置换算法（OPT）、最近最久未使用算法(LRU)

1.2 页面置换算法

(1) 先进先出页面置换算法（FIFO）：置换最早调入内存的页。

(2) 最佳页面置换算法（OPT）：置换将来被访问距当前最远的页。

(3) 最近最久未使用页面置换算法（LRU）：置换过去被访问距当前最远的页。

(4) 时钟页面置换算法（CLOCK）：扫描循环页面队列，遇到未引用页则置换。

1.3 缺页率

(1) 进程运行时，所要访问的页的序列称为**引用串**，又叫**访问串**。

(2) 缺页率＝(缺页数/访问串长度)×100%

(3) 命中率＝(1－缺页率)×100%

【例】设操作系统分配给进程 P 共 3 个页框，P 的访问串为 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2。

① FIFO 置换过程

访问串	7	0	1	2	0	3	0	4	2	3	0	3	2
页框 A	7	7	7	2	2	2	2	4	4	4	0	0	0
页框 B		0	0	0	0	3	3	3	2	2	2	2	2
页框 C			1	1	1	1	0	0	0	3	3	3	3
缺　页	×	×	×	×		×	×	×	×	×	×		

缺页数：10
缺页率：10/13=83.3%

② OPT 置换过程

访问串	7	0	1	2	0	3	0	4	2	3	0	3	2
页框 A	7	7	7	2	2	2	2	2	2	2	2	2	2
页框 B		0	0	0	0	0	0	4	4	4	0	0	0
页框 C			1	1	1	3	3	3	3	3	3	3	3
缺　页	×	×	×	×		×		×			×		

缺页数：7
缺页率：(7/13)×100%=53.8%

③ LRU 置换过程

访问串	7	0	1	2	0	3	0	4	2	3	0	3	2
页框 A	7	7	7	2	2	2	2	4	4	4	0	0	0
页框 B		0	0	0	0	0	0	0	0	3	3	3	3
页框 C			1	1	1	3	3	3	2	2	2	2	2
缺　页	×	×	×	×		×		×	×	×	×		

缺页数：9
缺页率：(9/13)×100%=69.2%

④ CLOCK 置换过程(注：下表中黄色块代表扫描指针指向块)

访问串	7	0	1	2	0	3	0	4	2	3	0	3	2
页帧 A	7(1)	7(1)	7(1)	2(1)	2(1)	2(1)	2(1)	4(1)	4(1)	4(1)	0(1)	0(1)	0(1)
页帧 B		0(1)	0(1)	0(0)	0(1)	0(1)	0(1)	0(0)	2(1)	2(1)	2(0)	2(0)	2(1)
页帧 C			1(1)	1(0)	1(0)	3(1)	3(1)	3(0)	3(0)	3(1)	3(0)	3(1)	3(1)
缺　页	×	×	×	×		×		×	×		×		

缺页数：8
缺页率：(8/13)×100%=61.5%

2 任务 1：FIFO 算法

2.1 任务描述

设计算法模拟程序：从键盘输入访问串。计算 FIFO 算法在不同内存页框数时的缺页数和缺页率。

2.2 程序设计

(1) 算法设计

FIFO 算法的原理是置换最早调入内存的页。实际算法设计是建立一个忙页框链，每访问一页，判断是否在此链中，若不在，则加入到链中。缺页时，若需替换，淘汰忙页框链首的页面。

定义页表结构：

```
struct pl_type{
   int pn; //页号
   int fn; //页框号
};
```

定义页框链结构：
```
struct fl_type{
   int pn; //页号
   int fn; //页框号
   struct fl_type *next;//链接指针
};
```
定义页表：`pl_type pl[512];`

准备页框链结点空间：`fl_type fl[512];`

定义空闲页框链指针：`*free_head;`

定义忙页框链指针：`*busy_head,*busy_tail;`

定义访问串：`int page[512];`

进程访问某页时，先用页号 page[i]查页表 pl[]，若找不到，则摘取空闲页框链首结点作为忙页框链尾结点，再把页号 page[i]放入该结点 pn 成员，把该结点 fn 值填入页表。摘取空闲页框时，若无空闲页框（即 free_head 为 NULL），则先摘取忙页框链首结点并把它加入空闲页框链，页表中相应的 fn 成员改为无效。下面以例子来说明。

设：分配给进程的页框数为 3，访问串为 3,0,1,0,2,0,4,2。

- 访问串 page[]={3,0,1,0,2,0,4,2};
- 初始化的空页表如图 3.4 所示。初始化代码为：

```
for(i=0;i<8;i++){    //为了描述简单，这里假定页表长为 8
   pl[i].pn=i;       //页号
   pl[i].fn=INVALID; //页框号为-1，（开始时，-1 表示页还未装入到页框）
}
```

	p1[0]	p1[1]	p1[2]	p1[3]	p1[4]	p1[5]	p1[6]	p1[7]
.pn	0	1	2	3	4	5	6	7
.fn	-1	-1	-1	-1	-1	-1	-1	-1

图 3.4　空页表

- 初始化的空闲页框链如图 3.5 所示。分给进程的页框数为 3，初始化代码为：

```
for(i=1;i<3;i++){
    fl[i-1].next=&fl[i]; //建立 fl[i-1]和 fl[i]间的链接
    fl[i-1].fn=i-1;
}
fl[3-1].next=NULL;  //链表末尾为空指针
fl[3-1].fn=3-1;     //末尾结点的页框号
free_head=&fl[0];   //空页框队列的头指针指向 fl[0]
```

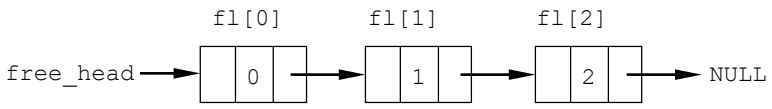


图 3.5　空闲页框链

- 开始时，忙页框链为空（图 3.6）。

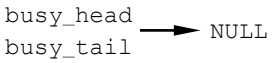


图 3.6　忙页框链

◆ 进程访问页号 3（即 page[0]），访问后的状态如图 3.7 所示。空闲页框分配步骤如下：

(a)查页表：if(pl[page[0]].fn==-1)为真，发生缺页，缺页数加 1。

(b)保存空闲页框链第 2 个结点：p=free_head->next;

(c)摘取空闲页框头结点：free_head->next=NULL;

(d)把页号 3 调入摘取的结点：free_head->pn=page[0];

(e)修改页表，记录对应的页框号：pl[page[0]].fn=free_head->fn;

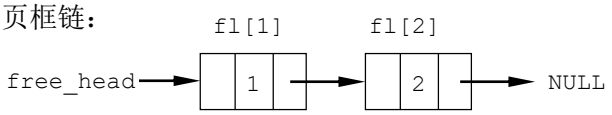
(f)摘取的结点加入忙页框链：busy_head=busy_tail=free_head;

(g)空闲页框链被摘走一个结点，头指针前移：free_head=p;

页表：

	p1[0]	p1[1]	p1[2]	p1[3]	p1[4]	p1[5]	p1[6]	p1[7]
.pn	0	1	2	3	4	5	6	7
.fn	-1	-1	-1	0	-1	-1	-1	-1

空闲页框链：



忙页框链：

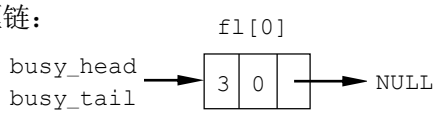


图 3.7　访问页号 3 后的页表和页框链

◆ 进程访问页号 3,0,1 后，页表和页框链状态如图 3.8 所示。

页表：

	p1[0]	p1[1]	p1[2]	p1[3]	p1[4]	p1[5]	p1[6]	p1[7]
.pn	0	1	2	3	4	5	6	7
.fn	1	2	-1	0	-1	-1	-1	-1

空闲页框链：　free_head → NULL

忙页框链：

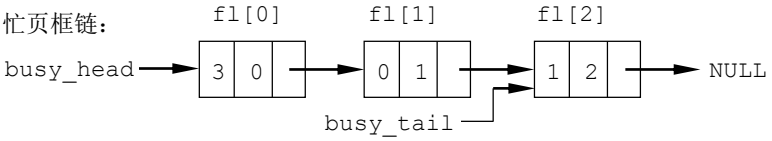


图 3.8 访问页号 3,0,1 后的页表和页框链

◆ 接下来访问页号 0。查页表，由于 0 页已分配了页框（即已在内存中），可直接访问。页表和页框链不发生变化。

◆ 在接下来访问页号 2。查页表，该页未分配页框，发生缺页。这时，无空闲页框（即 free_head 为 NULL）。须淘汰忙页框链首结点，淘汰步骤为：

(a)保存忙页框链第二个结点：　p=busy_head->next;

(b)修改页表，使相应页框号无效：　pl[busy_head->pn].fn=-1;

(c)摘取忙页框链头结点，加入空闲页框链：free_head=busy_head;
　　free_head->next=NULL;

(d)忙页框链被摘走一个结点，头指针前移：busy_head=p;

现在，有空闲页框了。接下来按前面所述的空闲页框分配步骤进行页框分配，再调入新页。访问后，页表和页框链的状态如图 3.9 所示。

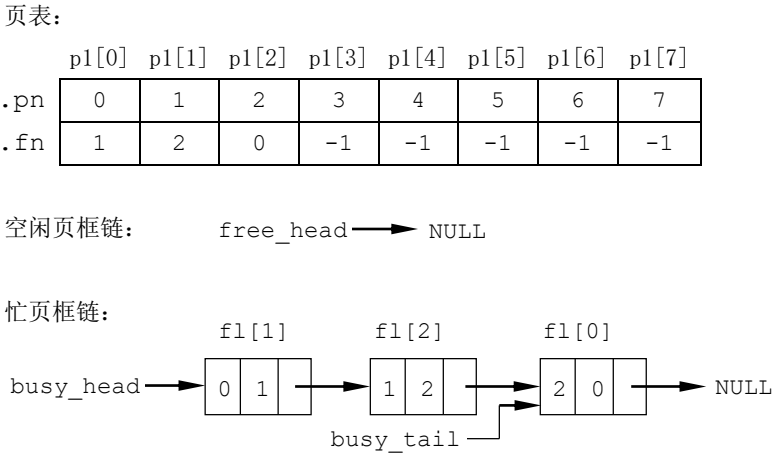


图 3.9 访问页号 3,0,1,0,2 后的页表和页框链

(2) 编码

```
#include "stdio.h"

#ifndef TRUE
#define TRUE 1
#define FALSE 0
#endif

#ifndef NULL
#define NULL 0
#endif

#define INVALID    -1    //-1 表示缺页
#define MAX_INT    ~(1<<sizeof(int)*8-1) //最大正整数

////页表结构类型
typedef struct pl_type {
    int pn; //页号
    int fn; //页框号
}pl_type;

//页框链结构类型
typedef struct fl_type {
    int pn; //页号
    int fn; //页框号
    struct fl_type *next; //链接指针
}fl_type;

//结构变量
```

```
pl_type pl[512]; //页表
fl_type fl[512],*free_head,*busy_head,*busy_tail; //页框

int page[512];  //访问串（访问的页号序列）
int total_papges; //访问串长度
int diseffect;  //缺页数

//初始化函数
//形参 frame_number 为分配给用户进程的内存页框数
void initialize(int frame_number)
{
    int i;
    diseffect=0; //页故障数初始化为 0

    //建立空页表
    for(i=0;i<512;i++) {
        pl[i].pn=i;       //页号
        pl[i].fn=INVALID; //页面号为空，（开始时，页还未装入到页面）
        pl[i].time=-1;    //时间为-1
    }

    //建立空闲页面链
    for(i=1;i<frame_number;i++) {
        fl[i-1].next=&fl[i]; //建立 fl[i-1]和 fl[i]间的链接
        fl[i-1].fn=100+i-1;//页面号由系统分配，例如从 100 开始，fl[i-1].fn=1000+i-1;
    }
    fl[frame_number-1].next=NULL;  //链表末尾为空指针
    fl[frame_number-1].fn=100+frame_number-1; //末尾结点的页面号
    free_head=&fl[0];  //空页面队列的头指针指向 fl[0]
}

//FIFO 函数：计算 FIFO 算法下的缺页率
void FIFO(int frame_number)
{
    int i;
    fl_type *p;

    initialize(frame_number);  //初始化页表和页面链
    busy_head=busy_tail=NULL; //忙页框链初始化为空
    for(i=0; i<total_pages; i++) {
        if(pl[page[i]].fn==INVALID) { //用页号 page[i]查页表，若为 INVALID 则页故障
            diseffect++; //记录页故障次数
            if(free_head==NULL){ //若无空闲页面,下面 5 句淘汰一个忙页面
                p=busy_head->next;  //保存忙页框链第 2 个结点
```

```
                pl[busy_head->pn].fn=INVALID;  //修改忙页框链头结点在页表中的页面号为未装入
                free_head=busy_head; //淘汰忙页框链头结点，加入空闲页面链首
                free_head->next=NULL; //断开忙页框链头结点
                busy_head=p; //忙页框链头指针前移
            }
            p=free_head->next; //保存空闲页面链第 2 个结点。
            free_head->next=NULL; //断开空闲页面链头结点（即摘取空闲页面头结点）
            free_head->pn=page[i]; //新页装入摘取的空闲页面结点
            pl[page[i]].fn=free_head->fn; //修改页表：记录装入的页面号
            if(busy_tail==NULL) { //若忙页框链为空
                busy_head=busy_tail=free_head; //忙页框链头指针指向摘取的空闲页面结点
            } else {
                busy_tail->next=free_head; //摘取的空闲页面结点加入忙页框链尾
                busy_tail=free_head; // 忙页框链尾指针前移
            }
            free_head=p; //空闲页面链头指针前移
        }
    }
    //打印结果
    printf("FIFO: %02d 次(%.02f%%)  ", diseffect, 100.0*diseffect/total_pages);
}

//FIFO 函数 2：计算 FIFO 算法下的缺页率，忙页框链另一种处理方法
void FIFO2(int frame_number)
{
    int i;
    fl_type *p;

    initialize(frame_number);  //初始化页表和页面链
    busy_head=free_head;
    for(i=0; i<total_pages; i++) {
        if(pl[page[i]].fn==INVALID) { //查页表，若缺页
            diseffect++; //记录缺页数
            if(free_head==NULL){ //若无空闲页面，下面 3 句置换忙页框链头结点
                pl[busy_head->pn].fn=INVALID; //淘汰忙页框链头结点
                busy_head->pn=page[i]; //忙页框链头结点装入新页
                pl[page[i]].fn=busy_head->fn; //修改页表
                p=busy_head;  //下面 5 句把忙页框链头结点移动到尾部
                busy_head=busy_head->next;
                busy_tail->next=p;
                busy_tail=busy_tail->next;
                busy_tail->next=NULL;
            }else{ //若有空闲页面，下面 3 句直接装入新页
```

```
                free_head->pn=page[i]; //空闲页面链头结点装入新页
                pl[page[i]].fn=free_head->fn; //修改页表
                free_head=free_head->next; //空闲页面链头指针前移
                if(free_head) busy_tail=free_head; //记录忙页框链尾结点
            }
        }
    }
    //打印结果
    printf("FIFO: %02d次(%.02f%%)  ", diseffect, 100.0*diseffect/total_pages);
}

//输入访问串
void Input_reference_string(void)
{
    int i;
    printf("输入的访问串，页号间以空格分开。例如: 7 0 2 6\n");
    printf("请输入访问串: ");
    for(i=0;i<512;i++) {
        if(scanf("%d", &page[i])!=1) break;
        if(getchar()==0x0a && i>0) break;
    }
    total_pages=i+1;  //访问串大小
    printf("访问串大小：%d\n访问串为: ", total_pages);
    for(i=0;i<total_pages;i++) printf("%d ", page[i]);
    printf("\n\n");
}

int main()
{
    int frames;

    Input_reference_string();
    printf("缺页数(缺页率)为\n");
    for(frames=2;frames<=total_pages;frames++){ //从2个页面到total_pages个页面
        printf(" 分配%2d个页面时，", frames);
        FIFO2(frames);
        OPT2(frames); //OPT2(frames);
        LRU2(frames);
        printf("\n");
    }
    return 0;
}
```

2.3 Belady 现象

一般情况下，分配给进程的页框数越多，缺页率越小。但 FIFO 算法情况下，有时给进

程的页框数越多，缺页率反而越大，这种现象称之为 Belady 线现象。比如，上面的访问串“1,2,3,4,1,2,5,1,2,3,4,5”在 FIFO 算法下会出现 Belady 现象。

3 任务 2：OPT 算法

3.1 任务描述

设计算法模拟程序：从键盘输入访问串。计算 OPT 算法在不同内存页框数时的缺页数和缺页率。

3.2 程序设计

(1) 算法设计

OPT 算法的原理是置换将来被访问距当前最远的页。实际算法设计是给每页设置一个距离 dist。缺页时，若需替换，从当前页开始向后遍历所有页，计算距离。淘汰距离最大的页面。程序如下：

(2) 编码

实验者自己编写代码。

4 任务 3：LRU 算法

4.1 任务描述

设计算法模拟程序：从键盘输入访问串。计算 LRU 算法在不同内存页框数时的缺页数和缺页率。

4.2 程序设计

(1) 算法设计

LRU 算法的原理是置换过去被访问距当前最远的页。LRU 的实现有几种方法，如计时法、计数法、队列法。这里采用计时法。给每页设一个访问时间 time，初始值都为-1。每访问一页，把它的 time 置为当前时间。缺页时，若需替换，淘汰 time 值最小的页面，淘汰后把该 time 置为-1。程序修改如下：

- 在页表结构中增加访问时间成员 time。页表结构变为：

```
struct pl_type{  //页表结构
   int pn;       //页号
   int fn;       //页框号
   int time;     //访问时间
};
```

- 初始化函数中增加 time 的初始化。

  在 initialize()中找到语句 pl[i].fn=INVALID; 在其下面添加：

```
pl[i].time=-1;
```

- 添加 LRU()函数代码

(2) 编码

实验者自己编写代码。

5 任务 4： CLOCK 算法（选做）

5.1 任务描述

编写 CLOCK 页面替换算法实现程序。装入内存的页框用链接指针组织成循环队列。为该队列设置一个扫描指针，用于记录上次替换后下一页框的位置。每页的页表项设一个引用位。每访问一页，其”引用位”赋值为 1。淘汰时，扫描循环队列，若引用位为 0 则淘汰该页；若引用位为 1 则改为 0，继续扫描下一页。

5.2 程序设计

实验者自己设计并编码。

4 设备管理

4.1 设备驱动程序

1 预备知识

用户空间和内核空间

当你开发设备驱动时，需要理解“用户空间”和内核空间之间的

区别。

（1）内核空间 ：Linux 操作系统，特别是它的内核，用一种简单而有效的方法管理机器的硬件，给用户提供一个简捷而统一的编程接口。同样的，内核，特别是它的设备驱动程序，是连接最终用户/程序员和硬件的一坐桥或者说是接口。任何子程序或者函数只要是内核的一部分（例如：模块，和设备驱动），那它也就是内核空间的一部分。

（2）用户空间. 最终用户的应用程序，像 UNIX 的 shell 或者其它的 GUI 的程序(例如，gedit),都是用户空间的一部分。很显然，这些应用程序需要和系统的硬件进行交互。但是，他们不是直接进行，而是通过内核支持的函数进行。

它们的关系可以通过 4.1 图表示。

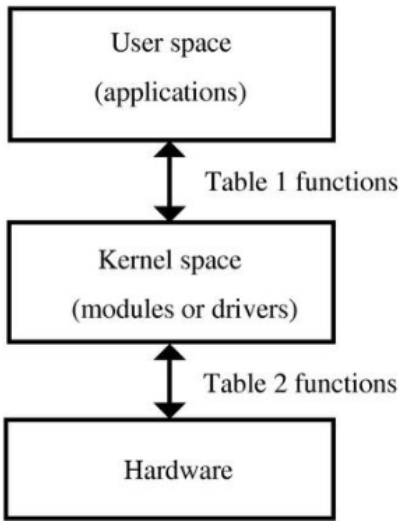


图 4.1 应用程序驻留在用户空间，模块和设备

（3）内核空间和硬件设备之间的接口函数

内核在用户空间提供了很多子程序或者函数，它们允许用户应用程序员和硬件进行交互。通常，在 Linux 系统中，这种交互是通过函数或者子程序进行的以便文件的读和写操作。这是因为从用户的视角看，UNIX 的设备就是一个个文件。

从另一方面看，在 Linux 内核空间同样提供了很多函数或者子程序以在底层直接地对硬件进行操作，并且允许从内核向用户空间传递信息。

通常，用户空间的每个函数（用于使用设备或者文件的），在内核空间中都有一个对应的功能相似并且可将内核的信息向用户传递的函数。这种关系可从图 4.2 看出来。目前这个表是空的，在后面每个表项都会填入对应的函数。

在内核空间同样有可以控制设备或者在内核和硬件之间交换信息的函数。

Events	User functions	Kernel functions
Load module	insmod	module_init()
Open device		
Read device		
Write device		
Close device		
Remove module	rmmod	module_exit()

图 4.2 用户空间与内核空间接口函数

2 程序设计

实现一个字符设备驱动程序 chardev.c，该设备的功能是维护聊天会话信息。

实现一个用户应用程序 chat.c,该程序可以向 chardev.c 读或写信息，从而实现多个用户之间的聊天功能。

（1）chardev.c

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/cdev.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/vmalloc.h>
#include <linux/slab.h>

#define MAX_DEVICES 10
#define MAJOR_NUM  101
#define MINOR_NUM 0
#define BUFFER_SIZE  1048576

struct _device_data{
    struct cdev  chardev;
    unsigned char *buffer;
    int npos;
}*mydata[MAX_DEVICES];

static ssize_t device_read(struct file *filp,
    char __user* buff, size_t len, loff_t* offset)
{
    int nlen=len;
    struct _device_data* pdb=filp->private_data;
    if(nlen>pdb->npos-*offset)
        nlen=pdb->npos-*offset;
    if(copy_to_user(buff, (pdb->buffer)+*offset, nlen))
```

```
        return -EFAULT;
    *offset += nlen;
    return nlen;
}

static ssize_t device_write(struct file *filp,
    const char __user* buff, size_t len, loff_t* offset)
{
    int nlen=len;
    struct _device_data* pdb=filp->private_data;
    if(nlen>BUFFER_SIZE-pdb->npos)
        nlen=BUFFER_SIZE-pdb->npos;
    if(nlen==0)
        return -ENOMEM;
    if(copy_from_user(&pdb->buffer[pdb->npos], buff, nlen))
        return -EFAULT;
    pdb->npos += nlen;
    return nlen;
}

static ssize_t device_open(struct inode *inode, struct file * filp)
{
    int nminor=iminor(inode);
    if(!mydata[nminor]->buffer)
        mydata[nminor]->buffer=(unsigned                          char
*)vmalloc(BUFFER_SIZE);
    if(!mydata[nminor]->buffer)
        return -ENOMEM;
    filp->private_data=mydata[nminor];
    if((filp->f_flags&O_ACCMODE)==O_WRONLY)
        mydata[nminor]->npos=0;
    return 0;
}

static ssize_t device_release(struct inode* inode,
    struct file* filp)
{
    return 0;
}

struct file_operations fops = {
    .owner=THIS_MODULE,
    .read=device_read,
    .write=device_write,
```

```
    .open=device_open,
    .release=device_release,
};

static int device_init(void)
{
    int i, ndev, ret;
    printk(KERN_INFO "Loading " KBUILD_MODNAME "...\n");
    for(i=0; i<MAX_DEVICES; ++i) {
        mydata[i] = (struct _device_data*)kmalloc(sizeof(*mydata[0]), GFP_KERNEL);
        if(!mydata[i]){
            printk(KERN_EMERG "Can't allocate memory to mydata\n");
            return -ENOMEM;
        }
        mydata[i]->buffer=NULL;
        mydata[i]->npos=0;
        cdev_init(&mydata[i]->chardev, &fops);
        mydata[i]->chardev.owner=THIS_MODULE;
        ndev=MKDEV(MAJOR_NUM, MINOR_NUM+i);
        ret=cdev_add(&mydata[i]->chardev, ndev,1);
        if(ret){
            printk(KERN_EMERG "Can't register device[%d]!\n", i);
            return -1;
        }
    }
    return 0;
}

static void device_exit(void)
{
    int i;
    printk(KERN_INFO "Unloading " KBUILD_MODNAME "...\n");
    for(i=0; i<MAX_DEVICES; ++i){
        cdev_del(&mydata[i]->chardev);
        if(mydata[i]->buffer)
            vfree(mydata[i]->buffer);
        kfree(mydata[i]);
    }
}


module_exit(device_exit);
module_init(device_init);
MODULE_LICENSE("GPL");
```

（3）Makefile 文件

```
# Makefile for chardev driver
obj-m := chardev.o
modules:
    make  -C  /usr/src/linux-headers-3.11.0-12-generic  M=/home/administrator  modules
```

（4）使用 make 编译得到 chardev.ko

（5）测试字符设备的 shell 程序 do.sh

```
#!/bin/sh
insmod ./chardev.ko
mknod /dev/chardev0 c 101 0
mknod /dev/chardev1 c 101 1
mknod /dev/chardev2 c 101 2
mknod /dev/chardev3 c 101 3
mknod /dev/chardev4 c 101 4
mknod /dev/chardev5 c 101 5
mknod /dev/chardev6 c 101 6
mknod /dev/chardev7 c 101 7
mknod /dev/chardev8 c 101 8
mknod /dev/chardev9 c 101 9
chmod 666 /dev/chardev*
```

（6）显示设备内容

```
ls -l  > /dev/chardev0
ls -a  > /dev/chardev1
cat /dev/chardev0
cat /dev/chardev1
```

(7) 写设备的应用程序 test.c

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

int main()
{
    int fd;
    int ch;
    fd=open("/dev/chardev0", O_RDWR|O_APPEND);
    if(fd==-1){
```

```
        fprintf(stderr, "can't openfile chardev0\n");
        exit(0);
    }
    fprintf(stderr, "请输入字符, 输入'q'结束\n");
    while((ch=getchar()) != 'q'){
        write(fd, &ch, 1);
    }
    close(fd);
    return 0;
}
```

（8）自己设计聊天程序

两个进程向设备读或写信息，从而实现聊天功能。

4.2 SPOOLing 模拟程序

1.SPOOLing(Simultaneous Peripheral Operation On-Line)

SPOOLing 技术是用一类物理设备模拟另一类物理设备的技术，是使独占型设备变成共享设备的一种技术，为缓和 CPU 的高速性与 I/O 设备低速性之间的矛盾而引入脱机输入、脱机输出技术，它是在多通道技术和多道程序设计基础上产生的。

2. SPOOLing 技术的实现

SPOOLing 系统主要由三部分组成：预输入程序、缓输出程序、井管理程序。如图 4.3 所示。

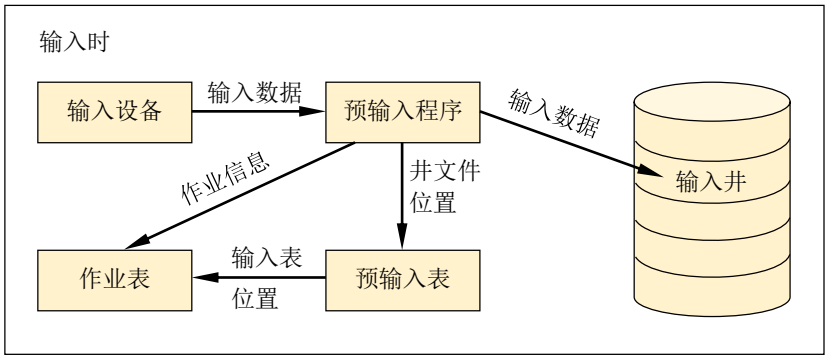


图 4.3 SPOOLing 设备组成

系统中存在作业表(JCB):

JCB1	JCB2	JCB3	…JCBn

作业表中的内容有：

作业名	作业状态	预输入表位置	缓输出表位置	其它信息

作业的状态有:

a. 输入状态 b. 收容状态 c. 执行状态 d. 完成状态

预输入表和缓输出表的内容有：

设备类	文件名	信息长度	存放位置

输入型虚拟设备的实现:

1. 启动预输入程序

2. 查询作业表及输入井,是否能容纳新作业,如果允许,继续往下面的步骤执行,否则报错

3.启动输入设备, 置输入井中的作业状态为输入状态, 读取作业说明书 ,将作业信息及预输入表的位置存入作业表中。

在输入井中寻找空闲块,把从设备中读入的信息以文件形式保存到输入井中,并登记井文件的位置到预输入表中。

置作业状态为收容状态。

当作业执行时,从输入井中取数据处理时,将状态作业改为执行状态,并释放井空间

输出型虚拟设备的实现，如图 4.4 所示。

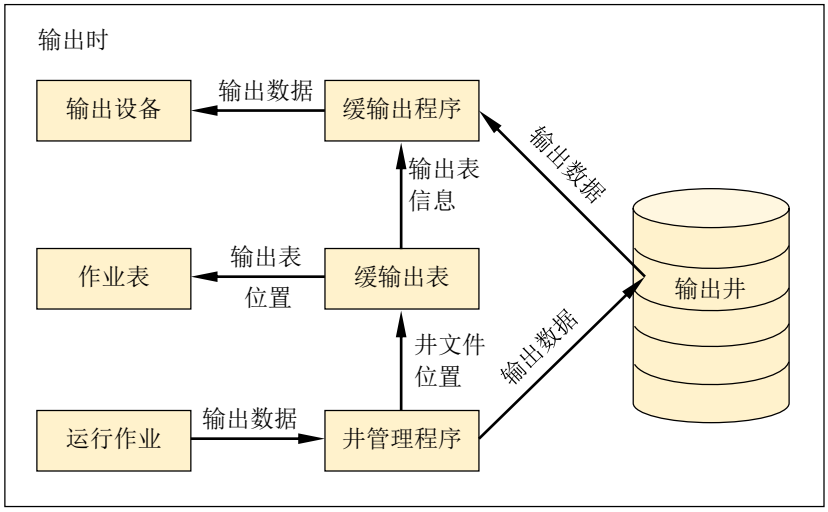


图 4.4　SPooling 结构示意图

1.　运行的作业需要输出时,井管理程序将输出的信息输出到输出井中，　将缓输出表的位置登记到作业表中,并登记输出井的位置到缓输出表中。

2.　当设备空闲时,缓输出程序查找缓输出表，　将输出井中的数据读取入缓冲区中,送到输出设备中输出.(数据也可能进行格式化后再从输出设备中输出)。

3.理解预输入程序、缓输出程序、井管理程序的作用

预输入程序：用于从外设中读取数据，并将数据写入输入井中，登记作业表和预输入表等相关信息缓输出表：从作业表和缓输出表中读取相关信息，将数据从输出井中取出数据，送至外设输出井管理程序：用于从输入井中取出预输入数据，交给作业进行处理，或者从输入井中取出数据后，经过格式化等处理后，将数据写入输出井。

4.画出利用 SPOOLing 技术的流程图

请自行绘制。

5.编写 SPOOLing 模拟程序。

要求设计一个 SPOOLing 输出进程和两个请求输出的用户进程，以及一个 SPOOLing 输出服务进程。当请求输出的用户进程希望输出一系列信息时，调用输出服务进程，由输出服务进程将该信息送入输出井，待遇到一个输出结束标志时，表示进程该次输出文件结束。之后，申请一个输出请求块，等待 SPOOLing 进程进行输出。输出请求块用来记录请求输出的用户进程的名称、信息在输出井中的位置、待输出信息的长度等。SPOOLing 输出进程工作时，根据请求块记录的各进程待输出的信息，将其实际输出到打印机或显示器。并且要求 SPOOLing 进程和用户进程可以并发执行。

5 文件系统

5.1 模拟文件系统设计

1 预备知识

(1) 文件系统

文件系统实现了信息的长期存取，它在一个有名字的对象中保存信息，这个对象称作**文件**。每个文件都有一个名字，称为**文件名**。用户访问文件是通过文件名来访问的，这叫做**按名存取**。用户对文件的操作形式有建立、打开、关闭、删除、读、写、控制等操作。

文件系统是操作系统中负责管理和存取文件信息的软件机构，它负责文件的建立，撤消，存入，读写，修改和复制，还负责完成对文件的按名存取和进行存取控制，以及向用户提供使用文件系统的接口。

系统中的每个文件包括文件属性和文件数据两部分。**文件属性**包括文件的基本信息（如文件名、文件标识符、文件大小、文件类型、文件主）、地址信息（如在磁盘上的起始地址、分配大小）、存取控制信息（如访问权限）、使用信息(如创建时间、修改时间)。文件系统为了能够快速准确地找到指定文件，把所有文件的属性信息组织成**目录**(即在磁盘上专门开辟一个区域，集中存放文件属性)。访问文件时，系统先用文件名查找目录，目录中找到指定文件名后，就可以得到该文件的属性信息，从而可以对文件进行进一步操作。为了能够把文件属性存放在目录里，系统为文件属性定义了一个数据结构，称为**文件控制块**。一般而言，每个目录条目就是一个文件控制块。为了进一步加快目录的查找速度，不少成功的操作系统都把文件名和其它属性分开，把这些除文件名以外的属性存放在一个叫做**索引结点**（即 **inode 结点**）的数据结构里。这样一来，文件目录的每个目录项仅由两部分构成，一个是文件名,另一个是 inode 指针。目录项缩小了，它所占的磁盘空间也就小了，查找目录自然就快了。

目录的结构形式有单级目录结构、二级目录、树型目录。**单级目录**结构是最简单的目录结构。整个系统当中只建立一张目录表，每个文件分配一个目录项。**两级目录**结构是在系统中建立一个**主文件目录 MFD**（Master File Directory），为每个用户建立一个单独的**用户文件目录 UFD**（User FileDirectory），主文件目录里面存放每个用户的目录项；该目录项中包含用户名和指向该用户目录的指针。图 5.1 是一个两级目录结构的示意。

为了避免每次文件操作都到磁盘上去查找文件目录,在(首次)操作文件之前先要打开文件。**打开文件**就是把文件目录项内容从磁盘读入内存,以后访问该文件就不需要在磁盘上搜索目录了，在内存中可以快速定位到文件的目录项。因为多个用户可以打开同一个文件，所以系统设立两种表来管理文件的打开，一是**系统打开文件表**,二是**用户打开文件表**。整个系统只有一张系统打开文件表，它记录文件目录项、文件号、共享计数、修改标志等信息。每个用户各有一张用户打开文件表，该表记录文件的描述符、打开方式、读写指针、指向系统打开文件表的指针等信息。用户打开文件表的地址保存在进程的 PCB 中。

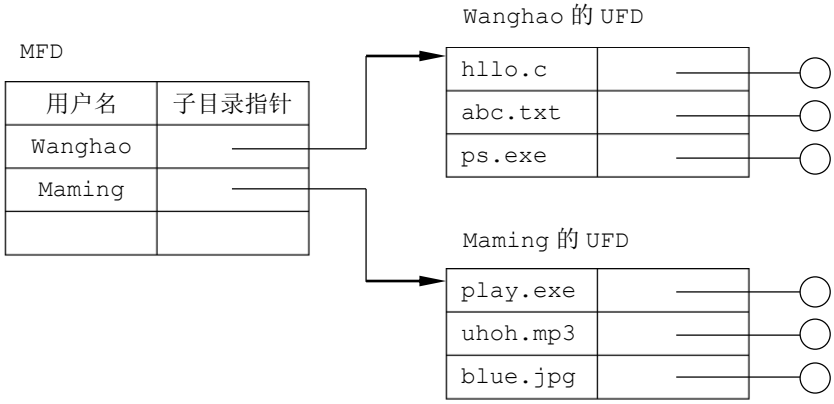


图 5.1 两级目录结构

文件最终要存储在物理磁盘上，文件在磁盘上的存放方法称为**文件的物理结构**。常见的物理结构有连续结构、链接结构、索引结构。存放文件时，把文件分割成小块，称为逻辑块，把磁盘也分为小块，称为物理块。存放时以块为单位存放。采用**连续结构**存放时，文件信息存放到连续物理块中；采用**链接结构**存放时，每个物理块设有一个指针指向下一个物理块，形成一个链接队列；采用**索引结构**存放时，每个文件建立一张索引表，表中每一栏目指出文件信息所在的逻辑块号和与之对应的物理块号。

文件存放在磁盘上要占用磁盘空间，这会牵涉到文件存储空间的管理，包括空闲块的组织，空闲块的分配与空闲块的回收。常用的空闲块管理方法有空闲块链、位示图法。**空闲块链**把所有空闲块链接在一起，从链头分配空闲块，把回收的空闲块插入链尾。而**位示图**法从内存中划出若干字节。每个比特对应一个物理块，如果该位为“0”，则表示所对应的块是空闲块；如果该位为“1”，则表示所对应的块已被分配出去。

(2) Linux 的EXT2 文件系统

在 Linux 当中，普通文件和目录文件保存在称为“块物理设备”的磁盘或者磁带等存储介质上。一套 Linux 系统支持若干个物理盘，每个物理盘可以定义一个或者多个文件系统。每个文件系统均由逻辑块的序列组成。一般来说，一个逻辑盘可以划分为几个用途各不相同的部分：引导块、超级块、inode 区以及数据区。

EXT2 是 Linux 自己的文件系统。它有几个重要的数据结构，一个是超级块，用来描述目录和文件在磁盘上的物理位置、文件大小和结构等信息；另一个是 inode。文件系统中的每个目录和文件均由一个 inode 描述。它包含：文件模式（类型和存取权限）、数据块位置等信息。一个文件系统除了重要的数据结构之外，还必须为用户提供有效的接口操作。比如 EXT2 提供的 OPEN/CLOSE 接口操作。

- **EXT2 的组描述符**

EXT2 文件系统将它所占用的逻辑分区划分成块，如图 5.2 所示。

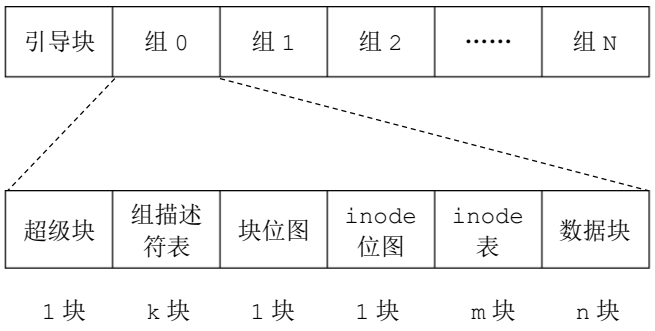


图 5.2 EXT2 文件系统的结构

超级块和组描述符:每个组中保存着关于文件系统的备份信息(比如超级块和所有组描述符)。在某个组的超级块或者 inode 受损时，可以用来恢复系统。

块位图（block bitmap）:记录本组内各个数据块的使用情况，其中每一位对应于一个数据块，0 表示空闲；非 0 表示已经分配。

inode 位图（inode bitmap）:记录 inode 表中 inode 的使用情况。

inode 表(inode table):保存本组所有的 inode。EXT2 文件系统使用 inode 来描述文件。一个 inode 对应于一个文件，而目录属于一种特殊的文件。每个 inode 对应于一个唯一的 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文件描述信息。

数据块（data block）:真正的文件数据区。在 EXT2 文件系统当中所有的数据块长度是一样的，数据块是系统为文件分配存储空间的单位。

- **EXT2 的超级块**

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。超级块对于文件系统的维护至关重要。它的数据结构位于 include/Linux/ext2_fs.h 中。 一般，只有块组号

为 0 的超级块才读入内存，其他块组的超级块仅仅作为备份。

- **EXT2 的 inode**

inode 是 EXT2 的基本组成部分。文件系统的每个文件（目录被看成文件）由一个 inode 描述。属于同一块组的 inode 保存在同一个 inode 表中，与该组的 inode 位图一一对应。Linux 关于 inode 数据结构的描述位于 include/Linux/ext2_fs.h 中。

(3) Windows的FAT文件系统

有关 FAT 文件系统的知识请参考微软白皮书“FAT32 File System Specification”，请参考 http://msdn.microsoft.com/zh-cn/windows/hardware/gg463080

http://wenku.baidu.com/view/bf355e31b90d6c85ec3ac621.html

【任务】

设计一个二级文件系统。要求如下，

- 目录结构要求：系统至少有两级目录，第一级对应于用户帐号，第二级对应于用户帐号下的文件或目录。列出的目录信息至少包括文件名、物理地址、保护码和文件长度。

- 文件操作要求：设置文件存取权限，提供对文件的读写保护。
- 功能要求：以命令形式提供下列功能给用户使用。登录 login、注销 logout、退出文件系统 halt、目录操作（创建目录 mkdir、改变目录 chdir、列目录 dir）、文件操作（创建文件 create、打开文件（open、读文件 read、写文件 write、关闭文件 close、删除文件 delete）等功能。（可以不考虑文件共享、文件系统安全、管道文件与设备文件等特殊内容）。

【设计提示】

（1）设计步骤

- 确定文件系统功能

即确定实验任务指定功能的细节。包括系统与用户的交互界面。

- 确定文件系统结构

确定文件系统结构主要是确定一些数据结构，如目录结构、文件的物理结构、用户信息结构等。时可以借鉴一些流行的操作系统文件系统结构，根据自己的设想做适当修改。

- 确定系统常量

要确定文件系统最多能存放多少数据，最多能存放多少个文件，数据块大小多少等系统常量。

- 确定磁盘布局

根据文件系统结构和系统常量，划分磁盘空间，确定系统数据存放地址。

- 定义数据结构

依据前面确定的文件系统结构而定。

- 设计底层支持函数

设计磁盘操作、结点操作、目录操作等函数。

- 格式化磁盘

这部本牵涉到许多磁盘驱动程序、设备管理方面的底层技术，这些技术不是本实　验的核

心内容。为了让设计工作与硬件无关，可以在磁盘上开辟一个区域，存放本文件系统的内容，这个专用区域可以用一个文件来模拟，假设这个文件取名为 filsys。格式化磁盘就是创建 filsys 文件，然后把文件系统的系统信息写入 filsys。

- 安装文件系统

安装的目的是向系统注册并把文件系统参数传给系统。模拟设计要做的主要工作是从磁盘中把文件系统信息读出来赋给系统内存一些结构变量。

- 用户注册

设计用户注册函数，等待用户注册，记录注册信息。

- 设计命令循环

在命令循环里等待用户输入命令，执行这些命令

- 完善系统功能

设计创建目录、列目录、改变目录、创建文件、打开文件等等函数，逐步完善系统功能。

(2)借鉴 Linux EXT2 进行设计

■ 磁盘结构设计

假定磁盘块大小为 512B，允许存放 512 个数据块。

设置一个启动块，一个超级块，32 个结点块（即，inode 块）。

根据上述参数，计算得到磁盘总容量为:（512+1+1+32）×512B=279552B

磁盘地址安排表 5.1 所示。

表 5.1 磁盘地址安排

地址	信息说明	备注
0x00000（0）	引导块	1 个引导块
0x00200（512）	超级块	1 个超级块
0x00400（1024）	索引节点 0	32 个 节点块
0x00420（1056）	索引节点 1	
0x00440（1088）	索引节点 2	
……	……	
0x04400（17408）	主目录 (block0)	512 个 数据块
0x04600（17920）	etc 目录 (block1)	
0x04800（18432）	passwd 区(block2)	
0x04A00（18944）	空闲	
0x04C00（19456）	空闲	
0x04E00（19968）	空闲	
0x05000（20480）	空闲	
0x05200（20992）	空闲	
. . .	. . .	
0x39c00（236544）	空闲	

■ 数据结构

与系统有关的数据结构

```
//超级块结构
struct filsys{
   unsigned short s_isize;  // i 结点块块数
   unsigned long s_fsize;  // 数据块块数
   unsigned int s_nfree;  // 空闲块块数
   unsigned short s_pfree;  // 空闲块指针
   unsigned int s_free[NICFREE];  // 空闲块堆栈
```

```
   unsigned int s_ninode;  // 空闲 i 结点数
   unsigned short s_pinode;  // 空闲 i 结点结点指针
   unsigned int s_inode[NICINOD];  // 空闲 i 结点数组
   unsigned int s_rinode;  // 铭记 i 结点
   char s_fmod;  // 超级块修改标志
};

//索引结点结构
struct inode{
   struct inode * i_forw;
   struct inode * i_back;
   char i_flag;
   unsigned int i_ino; // 磁盘 i 结点标志
   unsigned int i_count; // 引用计数
   unsigned short di_number; // 关联文件数，当为 0 时，则删除该文件
   unsigned short di_mode; // 存取权限
   unsigned short di_uid;
   unsigned short di_gid;
   unsigned int di_size;  // 文件大小
   unsigned int di_addr[NADDR] ;  // 物理块号
};
```

与目录有关的数据结构

```
//磁盘 i 结点结构
struct dinode{
   unsigned short di_numbcr;  // 关联文件数
   unsigned short di_mode;  // 存取权限
   unsigned short di_uid;
   unsigned short di_gid;
   unsigned long di_size;  // 文件大小
   unsigned int di_addr[NADDR] ;  // 物理块号
};

//目录项
struct direct{
```

```
   char d_name[DIRSIZ];
   unsigned int d_ino;
};
//目录
struct dir{
   struct direct direct[DIRNUM];
   int size;  // 当前目录大小
};
```

与用户有关的数据结构

```
//用户
struct user{
   unsigned short u_default_mode;
   unsigned short u_uid;
   unsigned short u_gid;
   unsigned short u_ofile[NOFILE];  // 用户打开文件表
};
//口令
struct pwd{
   unsigned short p_uid;
   unsigned short p_gid;
   char password[PWDSIZ];
};
```

与文件操作有关的数据结构

```
//系统打开文件表
struct file{
    char f_flag;  // 文件操作标志
    unsigned int f_count;  // 引用计数
    struct inode  * f_inode;  // 指向内存 i 节点
    unsigned long f_off;  // 读-写指针
};
```

(3)借鉴 Windows FAT 进行设计

- **系统的磁盘存储结构**

模仿 FAT16 设计文件系统，以 16 位表示一个簇。每簇大小 CLUSTER_SIZE=4KB

空间分配如表 5.2 所示。

文件系统的磁盘空间 256MB (总共占用 65536 簇)

根据磁盘空间大小和每簇大小可以计算出 FAT 空间大小：

FAT 空间大小=256MB/4KB*2B=128KB(占用 32 簇)

表 5.2 磁盘空间分配

0 簇	存放根目录项及用户表
1～32 簇	存放 FAT
33～65534 簇	用于存储文件及目录项
65535 簇	空闲不使用，作为尾簇标记

每个文件目录项的长度为 32 字节。

文件、目录的名称最长 8 字节。

用户信息项（USER_INFO）的长度为 32 字节。

用户名、密码的长度不超过 12 字节。

■ **文件目录项结构**

```
struct DirItemType{
  BYTE  szFileName[8] ;          // 文件名
  BYTE  szFileExt[3] ;           // 扩展名
  BYTE  bAttribute ;             // 0:只读，1：可写， 2：目录
  unsigned short  nStartClu ;    // 起始簇
  UINT  nFileSize ;              // 文件大小（字节）
  BYTE  szUserName[12] ;         // 所属用户
  BYTE  bReserve[2] ;            // 保留
};
```

■ **用户信息结构**

```
struct UserInfoType{
  BYTE  szUserName[12] ;  // 用户名
  BYTE  szUserPwd[12] ;   // 密码
  BYTE  bReserve[8] ;     // 保留
};
```

2 程序设计

二级文件系统在一个 Windows 文件中实现，盘块的读写可利用可函数或者利用 WindowsAPI。

自行选用文件系统结构设计二级文件系统。

5.2 文件删除与恢复

【实验目的】

1、 掌握 Windows FAT32 文件系统的文件删除前、后系统数据发生的变化。

2、 掌握文件完全恢复的必要条件，并编程实现。

【实验原理】

1、FAT32 分区在删除文件后 FDT（包括其标准 FDT 和所有拓展 FDT）的第一个字节会被置为 0xE5。

2、FAT32 分区删除文件后该文件的 FAT0 和 FAT1 的簇链信息数据将被清零。

3、FAT32 分区删除文件后，如果文件分配为连续的簇才可能被恢复。

4、FAT32 分区删除后的文件起始簇在 FAT 中位置的数据为 00000000，而且连续文件原有大小所占簇数个 FAT 单元的数据全是 00000000 的话文件很可能可以完全恢复。

【操作验证】

（一）实验本实验软件操作：

1、使用本实验软件，打开一个物理磁盘，查看其分区结构，选中分区类型为 FAT32 的分区，然后“右击”鼠标，会弹出一个快捷菜单，如图 5.3 所示。

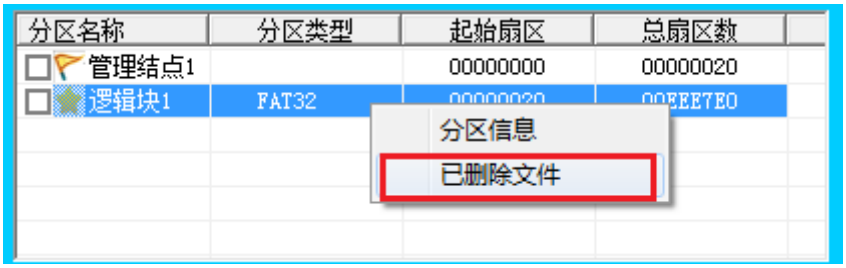


图 5.3 查看分区结构

2、选中快捷菜单中的“已删除文件”，即可查看根目录下所有已经删除的文件，选择某个已删除文件，右击，选择“簇链”选项卡，查看文件的簇链信息，如图 5.4 所示：

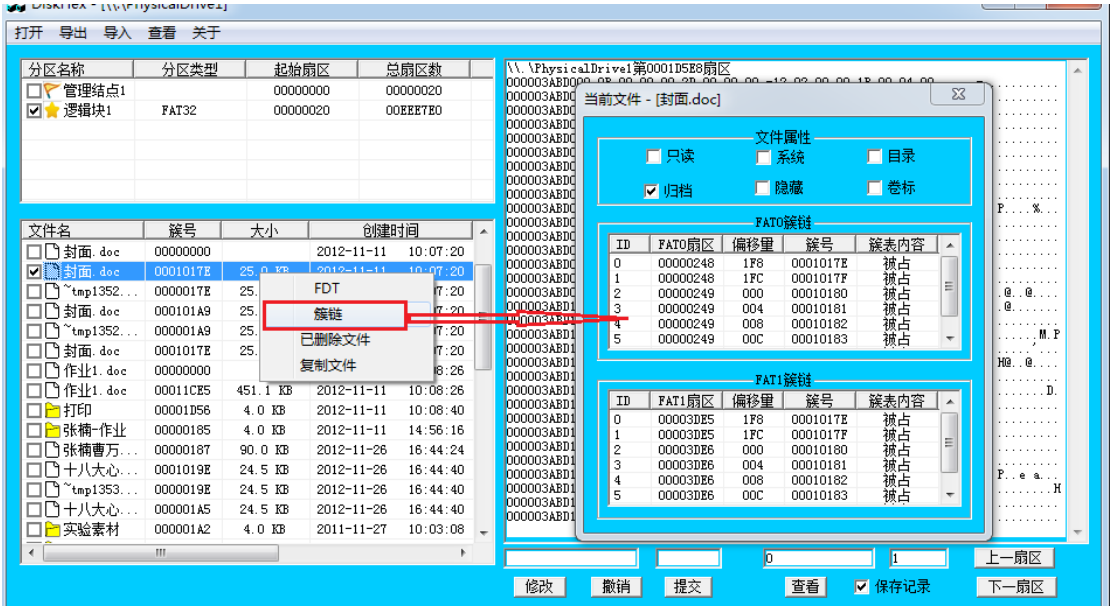


图 5.4　点击“簇链”

3、确定文件的簇链是否被占用的情况，点击簇链列表项，查看所点击的簇所在 FAT 表中是否为 00000000，如是，则表示文件的簇链没有被占，否则表示已被占。也可以看弹出的子对话框中的“簇表内容”项的情况。

4、不管文件数据的簇链有没有被占，右击该文件，选择“复制文件”选项卡，选择要保存到的目录，然后保存文件（复制文件时，都是假设文件的簇链是连续的，从文件 FDT 中获得文件的长度计算出文件的簇链长度）。

5、查看保存的文件的内容，看是不是要恢复的文件，判断是否恢复成功。如果未成功分析原因。

（二）手动实现文件的删除与恢复

1、在一个文件类型为 FAT32 的分区下创建一个文件，本实验已 U 盘为例，在 U 盘根目录下创建一个文件为：H:\file.txt 内容为 This is the test file! 循环 N 次。

在文件删除前查看文件的 FDT 与簇链。

文件删除前 FDT 如下所示：

\\.\PhysicalDrive1 第 00001E1A 扇区

……

0000003C3580 46 49 4C 45 20 20 20 20 -54 58 54 20 18 4E E4 9D FILE   TXT .N..

0000003C3590 A4 42 A4 42 00 00 EC 9D -A4 42 1E 02 BE 36 00 00 .B.B.....B...6..

……

文件删除前的簇链（FAT0 中）如下所示：

\\.\PhysicalDrive1 第 0000002A 扇区

000000005400 01 02 00 00 02 02 00 00 -03 02 00 00 04 02 00 00 ....

```
000000005410 05 02 00 00 06 02 00 00 -07 02 00 00 08 02 00 00  ................
000000005420 09 02 00 00 0A 02 00 00 -0B 02 00 00 0C 02 00 00  ................
000000005430 0D 02 00 00 0E 02 00 00 -0F 02 00 00 10 02 00 00  ................
000000005440 11 02 00 00 12 02 00 00 -13 02 00 00 FF FF FF 0F  ................
000000005450 15 02 00 00 16 02 00 00 -17 02 00 00 18 02 00 00  ................
000000005460 19 02 00 00 1A 02 00 00 -FF FF FF 0F FF FF FF 0F  ................
000000005470 FF FF FF 0F 7B 21 00 00 -1F 02 00 00 20 02 00 00  ....{!..... ...
000000005480 21 02 00 00 FF FF FF 0F -00 00 00 00 00 00 00 00  !...............
```

FAT1 中的簇链如下：

\\.\PhysicalDrive1 第 00000F23 扇区

```
0000001E4600 01 02 00 00 02 02 00 00 -03 02 00 00 04 02 00 00  ................
0000001E4610 05 02 00 00 06 02 00 00 -07 02 00 00 08 02 00 00  ................
0000001E4620 09 02 00 00 0A 02 00 00 -0B 02 00 00 0C 02 00 00  ................
0000001E4630 0D 02 00 00 0E 02 00 00 -0F 02 00 00 10 02 00 00  ................
0000001E4640 11 02 00 00 12 02 00 00 -13 02 00 00 FF FF FF 0F  ................
0000001E4650 15 02 00 00 16 02 00 00 -17 02 00 00 18 02 00 00  ................
0000001E4660 19 02 00 00 1A 02 00 00 -FF FF FF 0F FF FF FF 0F  ................
0000001E4670 FF FF FF 0F 7B 21 00 00 -1F 02 00 00 20 02 00 00  ....{!.........
0000001E4680 21 02 00 00 FF FF FF 0F -00 00 00 00 00 00 00 00  ................
```

2、获得文件删除前的 FDT 与簇链后。直接将文件删除，查看删除后文件的 FDT 与簇链。删除后文件的 FDT 与簇链还是在原来的位置，比较两者之间的变化。

删除后的 FDT 如下所示：

\\.\PhysicalDrive1 第 00001E1A 扇区

……

```
0000003C3580 E5 49 4C 45 20 20 20 20 -54 58 54 20 18 4E E4 9D  .ILE    TXT .N..
0000003C3590 A4 42 A4 42 00 00 EC 9D -A4 42 1E 02 BE 36 00 00  .B.B.....B...6..
```

……

删除后文件簇链（FAT0 中）所在位置数据如下所示：

\\.\PhysicalDrive1 第 0000002A 扇区

```
000000005400 01 02 00 00 02 02 00 00 -03 02 00 00 04 02 00 00  ................
000000005410 05 02 00 00 06 02 00 00 -07 02 00 00 08 02 00 00  ................
000000005420 09 02 00 00 0A 02 00 00 -0B 02 00 00 0C 02 00 00  ................
000000005430 0D 02 00 00 0E 02 00 00 -0F 02 00 00 10 02 00 00  ................
000000005440 11 02 00 00 12 02 00 00 -13 02 00 00 FF FF FF 0F  ................
000000005450 15 02 00 00 16 02 00 00 -17 02 00 00 18 02 00 00  ................
000000005460 19 02 00 00 1A 02 00 00 -FF FF FF 0F FF FF FF 0F  ................
000000005470 FF FF FF 0F 7B 21 00 00 -00 00 00 00 00 00 00 00  ....{!.........
000000005480 00 00 00 00 00 00 00 00 -00 00 00 00 00 00 00 00  ................
```

FTA1 中的簇链如下所示：

\\.\PhysicalDrive1 第 00000F23 扇区

```
0000001E4600 01 02 00 00 02 02 00 00 -03 02 00 00 04 02 00 00  ................
0000001E4610 05 02 00 00 06 02 00 00 -07 02 00 00 08 02 00 00  ................
0000001E4620 09 02 00 00 0A 02 00 00 -0B 02 00 00 0C 02 00 00  ................
0000001E4630 0D 02 00 00 0E 02 00 00 -0F 02 00 00 10 02 00 00  ................
0000001E4640 11 02 00 00 12 02 00 00 -13 02 00 00 FF FF FF 0F  ................
0000001E4650 15 02 00 00 16 02 00 00 -17 02 00 00 18 02 00 00  ................
0000001E4660 19 02 00 00 1A 02 00 00 -FF FF FF 0F FF FF FF 0F  ................
0000001E4670 FF FF FF 0F 7B 21 00 00 -00 00 00 00 00 00 00 00  ....{!..........
0000001E4680 00 00 00 00 00 00 00 00 -00 00 00 00 00 00 00 00  ................
```

3、实现文件的恢复，对于上面删除的文件 file.txt 实现恢复的话，就比较简单了，因为文件的簇链并没有被占用，恢复的步骤为：

①将 FDT 中被修改的部分修改过来（删除时只将 FDT 的第 1 个字节修该成了 0xE5,所以此时只要将 0XE5 修改成原来的字节即可恢复 FDT）；

②恢复文件的簇链，只要将原来获得的未删除的文件的簇链的数据写入到在 FAT0 与 FAT1 对应的位置即可。

如果文件删除后它的簇链被其他文件占用了，且不知道文件未被删除前的簇链，则恢复的方法是：找到已被删除的文件的 FDT，通过 FDT 获得文件的起始簇号和文件的长度，通过文件长度计算文件占用的簇数,此时对于簇链是连续的文件则可以通过起始簇号获得其他的簇号，然后定位到位置数据存放的扇区地址获得文件的数据从而实现恢复。

【程序阅读】

1、FAT32 中由于删除后文件目录项并没有被删除，只是把第一个字节置为 0xE5（已删除标志），所以扫描 FAT32 已删除文件和扫描 FAT32 文件方式一样，只不过是把文件目录项 FDT 首字节不是 0xE5 的过滤掉（和扫描未删除文件相反）。流程如下图 5.5 所示。

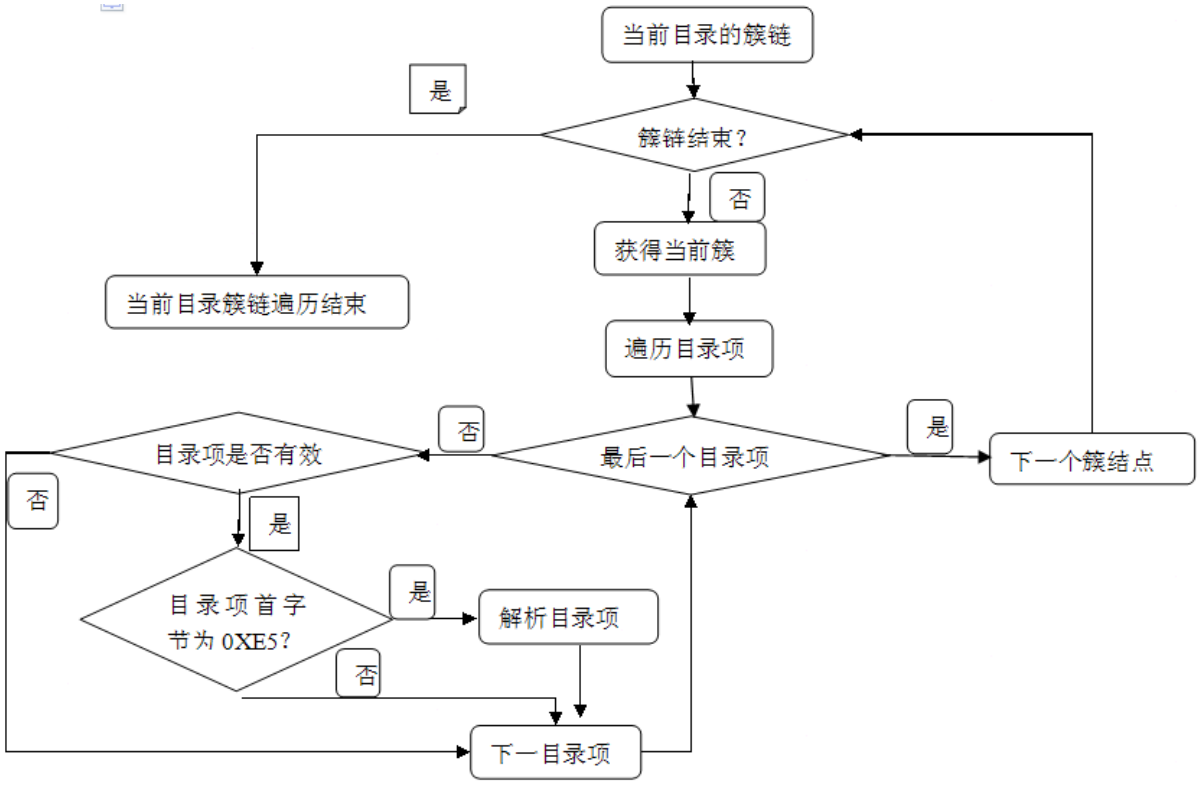


图 5.5 已删除文件扫描流程图

2、此过程的实现可以在文件 DealFAT.cpp 中，使用到的主要函数有：

```
//获得目录下所有已经删除的文件
void getAllDelFiles(FATbpbNode bpb ,FATNode fat ,CString diskName, DirNode dir);
//处理每簇中所有扇区中已经删除的文件
void dealPerDelClus(FATbpbNode bpb ,int FDTAddr,CString diskName, DirNode &dir);
///获得已经删除的文件的簇链 FAT0
void GetDelFATClust(FATbpbNode bpb ,CString diskName ,int cluster ,
    int phyAddr ,ULONGLONG fileSize ,FATNode &fat );
///获得已经删除的文件的簇链 FAT1
void  getDelFat1List(FATbpbNode bpb ,CString diskName ,int cluster ,
    int phyAddr ,ULONGLONG fileSize ,FATNode &fat1);
```

【问题思考】

1、你认为 FAT32 的文件恢复可以恢复到本分区吗？请阐述理由。

附录　Linux 系统调用

以下是 Linux 系统调用的一个列表，包含了大部分常用系统调用和由系统调用派生出的的函数。其中有一些函数的作用完全相同，只是参数不同。（有点像 C++函数重载，但是 Linux 核心是用 C 语言写的，所以只能取成不同的函数名）。还有一些函数已经过时，被新的更好的函数所代替了（gcc 在链接这些函数时会发出警告），但因为兼容的原因还保留着，这些函数在前面标上"*"号以示区别。

各系统调用的使用方法，可以通过 man 命令获得。

一、进程控制：

fork	创建一个新进程
clone	按指定条件创建子进程
execve	运行可执行文件
exit	中止进程
_exit	立即中止当前进程
getdtablesize	进程所能打开的最大文件数
getpgid	获取指定进程组标识号
setpgid	设置指定进程组标志号
getpgrp	获取当前进程组标识号
setpgrp	设置当前进程组标志号
getpid	获取进程标识号
getppid	获取父进程标识号
getpriority	获取调度优先级
setpriority	设置调度优先级
modify_ldt	读写进程的本地描述表
nanosleep	使进程睡眠指定的时间
nice	改变分时进程的优先级
pause	挂起进程，等待信号
personality	设置进程运行域
prctl	对进程进行特定操作
ptrace	进程跟踪
sched_get_priority_max	取得静态优先级的上限
sched_get_priority_min	取得静态优先级的下限
sched_getparam	取得进程的调度参数
sched_getscheduler	取得指定进程的调度策略
sched_rr_get_interval	取得按 RR 算法调度的实时进程的时间片长度
sched_setparam	设置进程的调度参数
sched_setscheduler	设置指定进程的调度策略和参数
sched_yield	进程主动让出处理器,并将自己等候调度队列队尾
vfork	创建一个子进程，以供执行新程序，常与 execve 等同时使用
wait	等待子进程终止

wait3	参见 wait
waitpid	等待指定子进程终止
wait4	参见 waitpid
capget	获取进程权限
capset	设置进程权限
getsid	获取会晤标识号
setsid	设置会晤标识号

二、文件系统控制

1、文件读写操作

fcntl	文件控制
open	打开文件
creat	创建新文件
close	关闭文件描述字
read	读文件
write	写文件
readv	从文件读入数据到缓冲数组中
writev	将缓冲数组里的数据写入文件
pread	对文件随机读
pwrite	对文件随机写
lseek	移动文件指针
_llseek	在 64 位地址空间里移动文件指针
dup	复制已打开的文件描述字
dup2	按指定条件复制文件描述字
flock	文件加/解锁
poll	I/O 多路转换
truncate	截断文件
ftruncate	参见 truncate
umask	设置文件权限掩码
fsync	把文件在内存中的部分写回磁盘

2、文件系统操作

access	确定文件的可存取性
chdir	改变当前工作目录
fchdir	参见 chdir
chmod	改变文件方式
fchmod	参见 chmod
chown	改变文件的属主或用户组
fchown	参见 chown
lchown	参见 chown
chroot	改变根目录
stat	取文件状态信息
lstat	参见 stat

fstat	参见 stat
statfs	取文件系统信息
fstatfs	参见 statfs
readdir	读取目录项
getdents	读取目录项
mkdir	创建目录
mknod	创建索引结点
rmdir	删除目录
rename	文件改名
link	创建链接
symlink	创建符号链接
unlink	删除链接
readlink	读符号链接的值
mount	安装文件系统
umount	卸下文件系统
ustat	取文件系统信息
utime	改变文件的访问修改时间
utimes	参见 utime
quotactl	控制磁盘配额

三、系统控制

ioctl	I/O 总控制函数
_sysctl	读/写系统参数
acct	启用或禁止进程记账
getrlimit	获取系统资源上限
setrlimit	设置系统资源上限
getrusage	获取系统资源使用情况
uselib	选择要使用的二进制函数库
ioperm	设置端口 I/O 权限
iopl	改变进程 I/O 权限级别
outb	低级端口操作
reboot	重新启动
swapon	打开交换文件和设备
swapoff	关闭交换文件和设备
bdflush	控制 bdflush 守护进程
sysfs	取核心支持的文件系统类型
sysinfo	取得系统信息
adjtimex	调整系统时钟
alarm	设置进程的闹钟
getitimer	获取计时器值
setitimer	设置计时器值
gettimeofday	取时间和时区
settimeofday	设置时间和时区

stime	设置系统日期和时间
time	取得系统时间
times	取进程运行时间
uname	获取当前 UNIX 系统的名称、版本和主机等信息
vhangup	挂起当前终端
nfsservctl	对 NFS 守护进程进行控制
vm86	进入模拟 8086 模式
create_module	创建可装载的模块项
delete_module	删除可装载的模块项
init_module	初始化模块
query_module	查询模块信息
*get_kernel_syms	取得核心符号,已被 query_module 代替

四、内存管理

brk	改变数据段空间的分配
sbrk	参见 brk
mlock	内存页面加锁
munlock	内存页面解锁
mlockall	调用进程所有内存页面加锁
munlockall	调用进程所有内存页面解锁
mmap	映射虚拟内存页
munmap	去除内存页映射
mremap	重新映射虚拟内存地址
msync	将映射内存中的数据写回磁盘
mprotect	设置内存映像保护
getpagesize	获取页面大小
sync	将内存缓冲区数据写回硬盘
cacheflush	将指定缓冲区中的内容写回磁盘

五、网络管理

getdomainname	取域名
setdomainname	设置域名
gethostid	获取主机标识号
sethostid	设置主机标识号
gethostname	获取本主机名称
sethostname	设置主机名称

六、socket 控制

socketcall	socket 系统调用
socket	建立 socket
bind	绑定 socket 到端口
connect	连接远程主机
accept	响应 socket 连接请求

send	通过 socket 发送信息
sendto	发送 UDP 信息
sendmsg	参见 send
recv	通过 socket 接收信息
recvfrom	接收 UDP 信息
recvmsg	参见 recv
listen	监听 socket 端口
select	对多路同步 I/O 进行轮询
shutdown	关闭 socket 上的连接
getsockname	取得本地 socket 名字
getpeername	获取通信对方的 socket 名字
getsockopt	取端口设置
setsockopt	设置端口参数
sendfile	在文件或端口间传输数据
socketpair	创建一对已联接的无名 socket

七、用户管理

getuid	获取用户标识号
setuid	设置用户标志号
getgid	获取组标识号
setgid	设置组标志号
getegid	获取有效组标识号
setegid	设置有效组标识号
geteuid	获取有效用户标识号
seteuid	设置有效用户标识号
setregid	分别设置真实和有效的的组标识号
setreuid	分别设置真实和有效的用户标识号
getresgid	分别获取真实的,有效的和保存过的组标识号
setresgid	分别设置真实的,有效的和保存过的组标识号
getresuid	分别获取真实的,有效的和保存过的用户标识号
setresuid	分别设置真实的,有效的和保存过的用户标识号
setfsgid	设置文件系统检查时使用的组标识号
setfsuid	设置文件系统检查时使用的用户标识号
getgroups	获取后补组标志清单
setgroups	设置后补组标志清单

八、进程间通信

ipc	进程间通信总控制调用

1、信号

sigaction	设置对指定信号的处理方法
sigprocmask	根据参数对信号集中的信号执行阻塞/解除阻塞等操作
sigpending	为指定的被阻塞信号设置队列

sigsuspend	挂起进程等待特定信号
signal	参见 signal
kill	向进程或进程组发信号
*sigblock	向被阻塞信号掩码中添加信号,已被 sigprocmask 代替
*siggetmask	取得现有阻塞信号掩码,已被 sigprocmask 代替
*sigsetmask	用给定信号掩码替换现有阻塞信号掩码,已被 sigprocmask 代替
*sigmask	将给定的信号转化为掩码,已被 sigprocmask 代替
*sigpause	作用同 sigsuspend,已被 sigsuspend 代替
sigvec	为兼容 BSD 而设的信号处理函数,作用类似 sigaction
ssetmask	ANSI C 的信号处理函数,作用类似 sigaction

2、消息

msgctl	消息控制操作
msgget	获取消息队列
msgsnd	发消息
msgrcv	取消息

3、管道

pipe	创建管道

4、信号量

semctl	信号量控制
semget	获取一组信号量
semop	信号量操作

5、共享内存

shmctl	控制共享内存
shmget	获取共享内存
shmat	连接共享内存
shmdt	拆卸共享内存

参考文献

[1] 费翔林．Linux 操作系统实验教程． 北京：高等教育出版社，2009．

[2] 罗宇．Linux 操作系统实验教程．北京：电子工业出版社，2009．

[3] 徐虹． 操作系统实验指导—基于 Linux 内核(第 2 版)． 北京：清华大学出版社，2009．

[4] 张尧学．计算机操作系统教程（第 3 版）习题解答与实验指导．北京：清华大学出版社，2006．