第 4 章　文件 I/O

[单元概述]

在对文件进行访问的过程中,最常见的 I/O 操作就是读写文件。在 Linux 中读写文件之前需要打开文件,文件读写完成之后还应关闭。另外还可以在读写文件的过程中进行读写位置的定位。Linux 中这些基本文件 I/O 操作使用系统调用完成。如系统调用 open()用于打开一个已有的文件或创建一个新文件;read()用于从文件中读取数据;write()用于向文件中写入数据;lseek()用于在文件中定位读写指针;close()用于关闭文件。这些访问文件的操作称之为基本文件 I/O。

在本章我们要编写一个复制文件的项目。该项目类似于 Linux 中的 cp 命令,用户可以在命令行中指定要被复制的源文件名和复制的目标文件名,该项目可以完成将源文件复制成目标文件。该项目主要利用 Linux 中标准文件 I/O 系统调用完成。

[教学重点与难点]

重点:文件系统、文件类型、文件描述符、文件基本 I/O 操作。

难点:访问文件的内核数据结构。

4.1　文件系统简介

【基本知识】文件系统

用户在使用计算机的过程中,大量的信息都是以文件的形式保存在计算机的存储设备上(如硬盘或者 CDROM)。操作系统的一个重要任务就是管理文件。对于操作系统来说,不仅要把众多的文件存放在存储设备上,还要对这些文件进行管理,同时对用户访问文件提供服务,所以操作系统不仅要将大量的文件存储在存储设备上,还要知道存储设备上都存有哪些文件,这些文件的属性是什么,存放在什么位置等等。此外,操作系统在管理文件时还需要一些数据结构来记录这些信息。而文件系统正是操作系统中管理文件的那部分系统软件及管理文件所需的各种数据结构。

用户以“按名存取”的方式访问文件。由于文件是系统中的重要资源,因此用户不能直接操作文件,而是通过给出文件名向操作系统发出请求,操作系统根据文件名找到文件,完成相应的操作并将结果返回给用户。

东软电子出版社

在命令行接口，当用户想查看某目录下的文件信息时，可以通过执行“ls”命令向操作系统发出请求，操作系统会将执行的结果显示给用户。如果用户想查看某文件的内容，则可以执行“cat”命令向操作系统发出请求并得到相应的服务。图形用户界面的操作也是类似的。

对于程序员要开发一个访问文件的程序，则需要在程序中调用操作系统提供的与文件相关的系统调用。例如在读文件时需要调用系统调用 read()，程序执行到系统调用 read()时，由操作系统完成文件数据的读出，并将结果返回给用户程序，用户程序可以对读到的结果进行处理。

本节主要介绍 Linux/UNIX 操作系统文件系统管理涉及到的一些概念。

为了实现对文件的管理，存储设备上除了存放文件，操作系统要在上面建立一些用于管理文件的数据结构。这些数据结构可以理解为一些表格。操作系统将存储设备上存储的文件的相关信息记录在这些表格里，比如这些文件的类型、长度、存取时间、存取权限、存储位置等。操作系统可以通过这些表格中记录的信息来管理文件。这些数据结构和文件一样，都保存在存储设备上。

一般来说，文件系统这个概念应该指的是操作系统中管理文件的那一部分系统软件、它们管理的对象(即文件)以及管理用到的数据结构。有时文件系统也指存储设备上的文件及用于管理的数据结构，或者仅仅指的是存储设备上存放的文件，这要从具体的上下文中区分。

不同的操作系统管理文件的方法各不相同，它们用于管理文件的数据结构和对存储设备的使用方式也不同。在操作系统使用一个新的存储设备时，首先要对其进行“格式化”。所谓的“格式化”就是操作系统按照自己的方式在存储设备上建立文件系统。操作系统对存储设备的空间进行划分，并在其上建立初始的数据结构。这个过程完成后，文件系统就建立好了，操作系统就可以在上面存取并管理文件了。

目前有许多不同的文件系统类型，常见的有 DOS/Windows 采用的 FAT 和 NTFS 文件系统、Linux 采用的 ext2/ext3 文件系统、UNIX 采用的 UFS 文件系统等等。

通常操作系统不仅使用自己的文件系统类型，还支持多种其他不同类型的文件系统。例如，在 UNIX 中，我们也可以存取一个 Windows 的 FAT 类型的硬盘中的文件。对多种文件系统类型的支持方便了用户在不同系统之间共享文件。

4.1.1　UNIX/Linux 文件系统概述

UNIX 系统及类 UNIX 系统采用称为 UFS(UNIX File System)的文件系统，一个简单的 UNIX 文件系统分为四个部分：引导块、超级块、索引节点表及数据区。

引导块(boot block)：在每个分区的第一个块上，其中包含有用于引导该分区内操作系统的引导程序。

超级块(super block)：在引导块之后，由若干个块(如磁盘块)组成，存放了该 UFS 的一些重要参数，例如该文件系统的块总数、空闲块数、索引节点总数、空闲索引节点总数等等。

索引节点表(inodes)：位于超级块和数据区之间，由若干个块组成，其中包含了很多索引节点 inodes(Index nodes)。在 UFS 中，使用一个称为索引节点 inodes 的结构来保存每一个文件的属性信息。一个 UFS 的第三部分即是该文件系统中索引节点的集合，每一个保存在该文件系统中的文件，在这一部分都对应有一个索引节点保存该文件的信息。

数据区(data blocks)：在索引节点表之后，是一个分区除了前面三部分之外的剩余部分，也

是存储文件具体内容的区域，占据文件系统的绝大部分空间。

UNIX 文件系统结构如图 4.1 所示。

图 4.1 UNIX 文件系统结构

UNIX 操作系统对 UFS 文件系统的使用大致是这样：当用户需要将一个文件保存到该文件系统上时，UNIX 系统需要为这个文件分配相应的空闲磁盘空间和一个空闲索引节点。UNIX 系统根据超级块中关于空闲块的参数从数据区找出足够的空闲磁盘块用来存放文件具体内容，并根据超级块中关于空闲索引节点的参数找出一个空闲索引节点用来存储该文件的属性信息，属性信息中包括文件类型、存取权限、存取时间、文件长度以及存放的磁盘块位置等。

当以后用户需要存取该文件时，UNIX 根据文件名在索引节点表中找到该文件的索引节点，从中读出其属性信息，进行存取权限的验证，并根据索引节点中记录的文件存储位置到数据区找到相应磁盘块对文件内容进行存取。

Linux 最早的版本是基于 Minix 文件系统的。后来 Linux 引入了扩展文件系统（Ext FS）。这相当于扩展文件系统第一版，其性能不能令人满意。Linux 在 1994 年引入了第二扩展文件系统（second Extended Filesystem，Ext2）。它除了包含几个新的特点外，还相当高效和强健，已成为广泛使用的 Linux 文件系统。

一个 Ext2 分区中的第一个块是引导块。其余部分分成块组（block group），其结构图如图 4.2 所示。

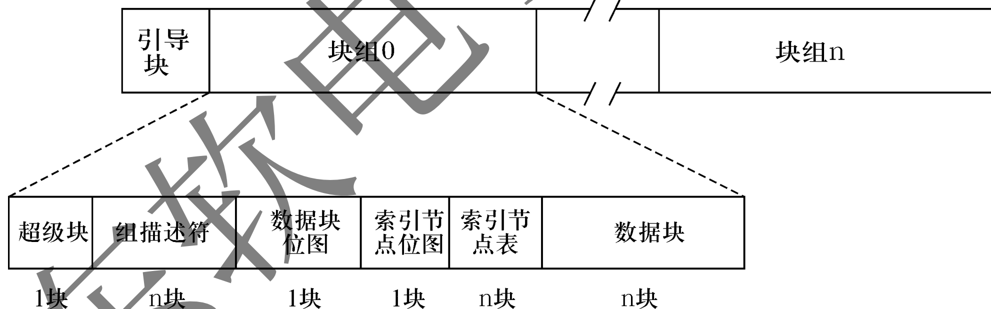

图 4.2 Ext2 文件系统的结构

由于内核尽可能地把属于一个文件的数据块存放在同一块组中，所以块组减少了文件的碎片。块组中的每个块包含下列信息之一：

- 文件系统超级块的一个拷贝；
- 一组块组描述符的拷贝；
- 一个数据块位图；
- 一组索引节点；
- 一个索引节点位图；
- 属于文件的一大块数据；即一个数据块。

东软电子出版社

4.1.2　VFS 虚拟文件系统

【基本知识】虚拟文件系统

VFS 是虚拟文件系统(Virtual File System)的简称。虚拟文件系统 VFS 是建立在各种具体文件系统之上的一个抽象层。VFS 的目的是为了使用户可以以统一的方式访问不同的文件系统。VFS 可以实现透明地访问本地的或者是网络的存储设备,而用户感觉不到它们之间的差别。VFS 定义了一组统一的接口来访问具体文件系统的文件,如 open、read、write、close、lseek 等。有了 VFS,一个操作系统就可以方便地支持多种不同的文件系统。同样增加对一个新的文件系统的支持就变得很容易了。用户在通过 VFS 访问文件时也可以使用统一的接口,而不必关心具体文件系统的类型。

4.1.3　索引节点 inode

【基本知识】索引节点

UNIX/Linux 中,每个文件分为两部分:索引节点 inode 及文件数据块部分。其中索引节点 inode 用于记录文件的各种属性信息,而文件的具体内容存放在文件的数据块中。inode 中主要包含以下信息,如表 4.1 所示。

表 4.1　索引节点 inode

字段	类型	描述
i_mode	_ _u16	文件类型和访问权限
i_uid	_ _u16	拥有者标识符
i_size	_ _u32	以字节为单位的文件长度
i_atime	_ _u32	最后一次访问文件的时间
i_ctime	_ _u32	索引节点最后改变的时间
i_mtime	_ _u32	文件内容最后改变的时间
i_dtime	_ _u32	文件删除的时间
i_gid	_ _u16	组标识符
i_blocks	_ _u32	文件的数据块数
i_flags	_ _u32	文件标志
i_block	_ _u32	指向数据块的指针

可以看到,除了文件名以外,文件的主要属性信息都存放在 inode 节点中。文件系统中所有文件的 inode 都存放于文件系统的索引节点表。而目录文件中保存着文件名与索引节点的对应关系。

4.1.4　文件的类型

【基本知识】文件类型

在 Linux 中,文件的类型分为以下七种:

华软电子出版社

- 普通文件(rugular file);
- 目录文件(directory);
- 字符设备文件(character device);
- 块设备文件(block device);
- FIFO 文件(fifo);
- 符号链接文件(symbolic link);
- Socket 套接字文件(socket)。

1. 普通文件(regular file)

在 Linux 中,一般文件都属于普通文件,也称为常规文件,如文本文件、二进制文件、图形文件等等。

/etc/profile 文件就是一个普通文件,它的长格式信息如下:

```
-rw-r--r--  1  root  root  1029  2009-05-11  /etc/profile
```

2. 目录文件(directory file)

在 Linux 中,目录也是一个文件,目录文件有自己的索引节点以及文件内容。目录文件中保存的内容是该目录下的文件名及对应的索引节点编号。每一个目录中都包含有两个文件"."和"..",其中"."代表本级目录,".."代表上级目录。

Linux 系统采取"按名存取"的方式访问文件。用户访问一个文件时,需要给出包含路径的文件名,系统一级一级地搜索每一级目录文件的内容,直到找到该文件名并获得其索引节点编号,最后根据索引节点中记录的信息找到该文件并对其访问。

例如,要访问文件"/usr/include/fcntl. h",系统首先读出根目录"/"的目录文件内容(根目录一般使用 2 号索引节点),在其中查找"usr"并得到"usr"的索引节点,接下来根据"usr"索引节点中记录的信息将"/usr"目录文件内容读出,并在其中查找"include"及其索引节点。同理,根据"include"索引节点读出"include"目录文件内容,在其中再查找"fcntl. h"。这样最终在"include"目录文件中找到了"fcntl. h"及其索引节点,这样系统就可以根据其索引节点的信息对该文件进行访问了。

/home 目录的长格式信息如下:

```
drwxr-xr-x   5  root root   4096  11-14 10:32  /home
```

3. 设备文件(device file)

在 UNIX/Linux 中,设备也作为一个文件来看待。这样,对设备的输入输出操作就和对文件的读写操作统一起来。设备分为字符设备和块设备。字符设备以字符为单位输入输出数据,键盘、打印机等就是典型的字符设备。块设备则是以块为单位进行数据的输入输出。磁盘是一个典型的块设备,一般一个磁盘块是 512 字节(或者其倍数),系统对磁盘进行读写操作时,每次读写至少是一个磁盘块。

在 Linux 中,每一个设备都对应着一个设备文件。设备文件也分为字符设备文件和块设备文件。这些设备文件都存放在"/dev"目录下。

如/dev/ttyS0 是第一个串口,是一个字符设备;/dev/sda1 是第一个磁盘分区,是一个块设备文件。

```
crw-rw----  1  root  uucp  4, 64   11-14 10:44  /dev/ttyS0
brw-r-----   1  root  disk  8, 1    11-01 10:28  /dev/sda1
```

东软电子出版社

4. 符号链接文件(symbolic link file)

在 Linux 中,为了访问文件方便,可以给一个文件创建一个符号链接。符号链接也是一个文件,它不同于所链接的目标文件,拥有自己的索引节点和文件内容。符号链接文件的内容是其所链接的目标文件的路径名。

当对一个符号链接文件进行操作时,一般情况下,系统会将操作转移到其所链接的目标文件上。例如系统中有一个文件 f1,并且给 f1 文件建立了一个符号链接文件 sf1。sf1 指向其链接的目标文件 f1。当我们对 sf1 文件执行 cat 命令查看其内容时,看到的是其链接的目标文件 f1 的内容,而不是 sf1 自己的内容。

如/dev/cdrom 指向光盘驱动器,就是一个符号链接文件:

```
lrwxrwxrwx  1  root  root  3  11-01 10:28  /dev/cdrom->hdc
```

5. 管道文件(fifo file)

管道是 Linux 中进程间通信的一种机制。管道就是进程之间按照先进先出 FIFO 方式传递数据的一种文件。管道分为无名管道和命名管道。无名管道用于有亲缘关系的进程之间,如一个父进程创建了一个管道,然后创建子进程。其子进程可以从父进程那里继承来管道,这样父子进程通过管道进行数据传递。无名管道是一种临时文件,而命名管道则是有文件名的,如/dev/initctl/:

```
prw-------   1  root  root  0  11-01 10:30  /dev/initctl
```

6. 套接字文件(socket file)

套接字是 Linux 中进程间进行网络通信的一种机制,如/dev/gpmctl 和/dev/log 都是套接字文件:

```
srwxrwxrwx  1  root  root  0  11-01 10:29  /dev/gpmctl
srw-rw-rw-  1  root  root  0  11-01 10:29  /dev/log
```

4.1.5　文件的访问权限

【基本知识】文件访问权限

UNIX/Linux 系统是一个多用户的操作系统,它允许多个用户同时登录到系统中。因此对文件设置访问权限是多用户系统中文件管理的一个重要内容。

在 Linux 中,用户被分为三类:文件的主人 owner、组成员 group 和其他用户 other。用户对文件的基本访问权限也分为三种:读(read)、写(write)和执行(execute)。这样一个文件拥有九种基本权限,如表 4.2 所示。

表 4.2　基本文件访问权限

文件主人			组成员			其他用户		
读 read	写 write	执行 execute	读 read	写 write	执行 execute	读 read	写 write	执行 execute

除此之外,文件还有 setuid、setgid 和 sticky 三个特殊权限(在此暂不对这三个特殊权限进行介绍)。

在 Linux 的文件系统中,文件的索引节点 inode 里面有一个 st_mode 字段记录文件的类型及权限。该字共 16bits,其中高 4 位用于表示文件类型,低 12 为表示文件权限。

在 Linux 系统中定义了文件类型的二进制码。

S_IFMT	0170000	文件类型字段的二进制掩码
S_IFSOCK	0140000	socket 文件
S_IFLNK	0120000	符号链接文件
S_IFREG	0100000	常规文件(普通文件)
S_IFBLK	0060000	块设备文件
S_IFDIR	0040000	目录文件
S_IFCHR	0020000	字符设备文件
S_IFIFO	0010000	fifo 文件

st_mode 字段低 12 位中,第 11bit 表示 SUID 权限,第 10bit 表示 SGID 权限,第 9bit 表示 sticky 权限,第 8～6bits 表示文件主人对文件的读、写、执行权限;第 5～3bits 表示同组用户对文件的读写执行权限;第 2～0bits 表示其他用户对文件的读写执行权限。

在 Linux 系统中定义了文件类型的二进制码。

S_ISUID	0004000	SUID 权限
S_ISGID	0002000	SGID 权限
S_ISVTX	0001000	sticky 权限
S_IRWXU	00700	文件主人权限掩码
S_IRUSR	00400	主人有读权限
S_IWUSR	00200	主人有写权限
S_IXUSR	00100	主人有执行权限
S_IRWXG	00070	同组用户权限掩码
S_IRGRP	00040	同组用户有读权限
S_IWGRP	00020	同组用户有写权限
S_IXGRP	00010	同组用户有执行权限
S_IRWXO	00007	其他用户权限掩码
S_IROTH	00004	其他用户有读权限
S_IWOTH	00002	其他用户有写权限
S_IXOTH	00001	其他用户有执行权限

Linux 系统还定义了下列判断文件类型的宏,如表 4.3 所示。

表 4.3 判断文件类型的宏

宏	描述
S_ISREG(m)	判断是否为常规文件
S_ISDIR(m)	判断是否为目录文件
S_ISCHR(m)	判断是否为字符设备文件
S_ISBLK(m)	判断是否为块设备文件
S_ISFIFO(m)	判断是否为 fifo 文件
S_ISLNK(m)	判断是否为符号链接文件
S_ISSOCK(m)	判断是否为 Socket 文件

Linux 中文件类型及权限字段 st_mode 结构如图 4.3 所示。

东软电子出版社

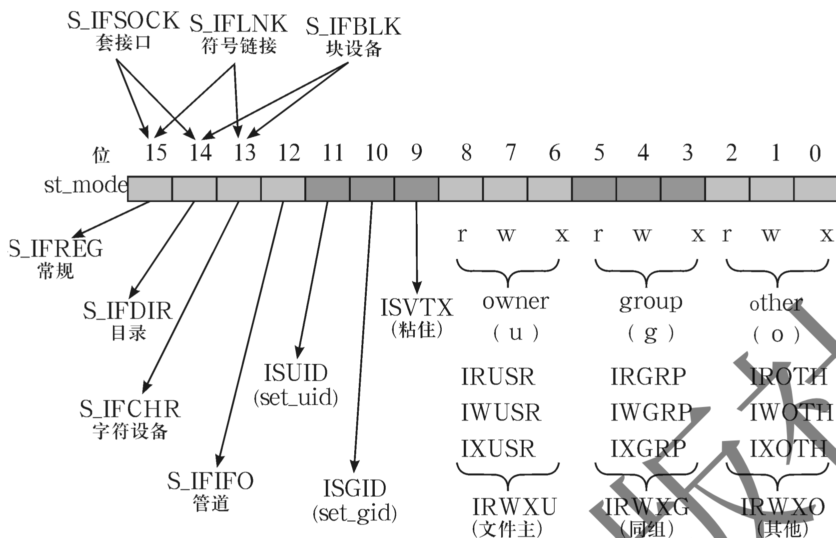

图 4.3　文件的类型与权限——st_mode 字段

4.2　访问文件的内核数据结构

当要访问一个文件时，系统需要在内存中建立相应数据结构来记录访问文件需要的各种信息，这些数据结构包括：内存 inode 表、文件表、用户文件描述符表等，如图 4.4 所示。

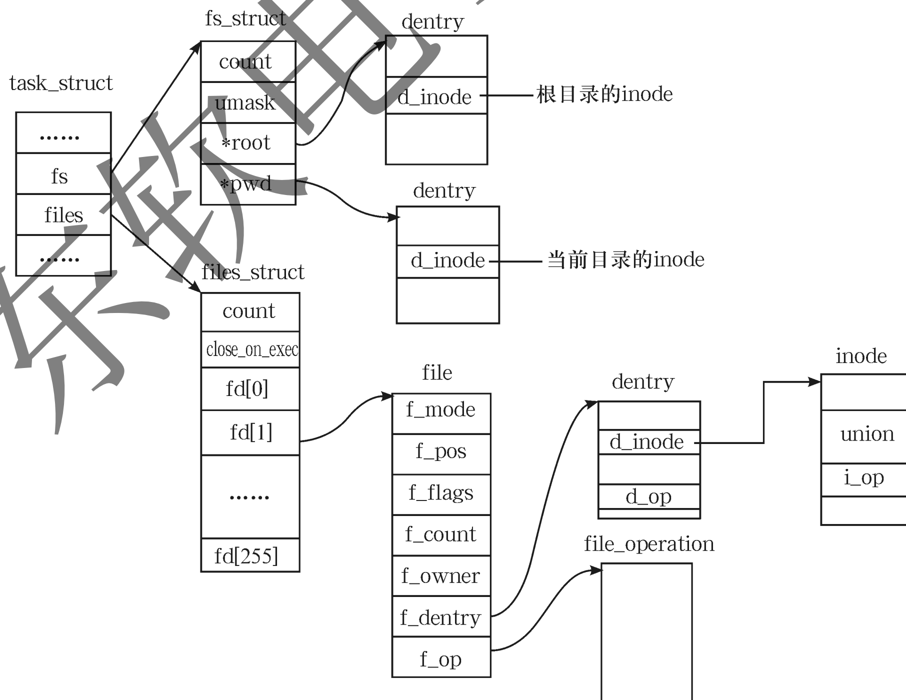

图 4.4　Linux 进程对文件的访问

东软电子出版社

用户通过文件名向系统发出访问文件的请求，系统根据文件名在磁盘文件系统的 inode 表中查找到该文件的 inode，同时将该 inode 复制到内存的 inode 表中，以便提高访问 inode 的速度。内存 inode 表是用户用到的文件 inode 在内存的拷贝。内存中 inode 节点会比磁盘 inode 节点多一些信息，如内存 inode 节点是否被修改过的标记、内存 inode 节点引用计数等。

系统还要建立一个文件表，用于记录系统中所有打开的文件。每当用户打开一个文件时，都要在该表中登记一个表项，用于记录该文件的打开方式、读写指针、指向内存 inode 节点的指针等。注意：如果用户两次执行 open()打开同一个文件，也会在文件表中登记两个表项。

对于每一个用户（进程），系统也要建立一个用户打开文件描述符表来记录每个用户当前打开的所有文件。用户文件描述符表中保存着指向文件表中对应表项的指针。

用户打开文件描述符表中每个表项的索引就是每个文件的文件描述符。当打开文件成功时，系统将该文件在用户打开文件描述符表中的索引返回给用户。用户以后对该文件访问时都要通过该文件描述符。

每一个进程在建立好时，都默认打开三个文件：标准输入设备文件、标准输出设备文件、标准错误输出设备文件。这三个文件登记在用户打开文件描述符表中的前三项。因此这三个文件的文件描述符分别是 0、1 和 2。Linux 系统中定义了这三个文件描述符：

```
#define  STDIN_FILENO      0        //标准输入设备文件
#define  STDOUT_FILENO     1        //标准输出设备文件
#define  STDERR_FILENO     2        //标准错误输出设备文件
```

4.3 文件基本 I/O 操作

4.3.1 打开/创建文件 open/creat

使用系统调用 open()可以打开文件并得到一个文件描述符，该文件描述符用于对打开的文件进行访问，如表 4.4 所示。

表 4.4 系统调用 open/creat

项目	描述
头文件	#include <sys/types.h> #include <sys/stat.h> #include <fcntl.h>

原型	int open(const char *pathname, int flags);
	int open(const char *pathname, int flags, mode_t mode);
	int creat(const char *pathname, mode_t mode);

功能	按照 flags 方式打开 pahtname 指定的文件，或者创建权限为 mode 的新文件 pathname
参数	pathname：要打开或者创建的文件名（包含路径） flags：打开方式（如表 1） mode：当要创建一个新文件时（flags 中包含 O_CREAT），mode 指定新创建文件的权限
返回值	int 型结果：当打开文件成功时，返回一个文件描述符；当打开文件失败时，返回－1，并且 errno 为错误码

东软电子出版社

1. 文件描述符

【基本知识】文件描述符

成功地打开文件时,会得到一个文件描述符。文件描述符是唯一标识用户打开该文件的非负整数。这个整数其实是该文件在用户打开文件描述符表中表项的索引。

每个进程都有一个用户文件描述符表,用来记录该进程已经打开的文件。当一个进程通过 open()打开一个文件时,系统根据 pathname 找到该文件的 inode,校验该文件的属性信息是否可以打开。如果可以打开,则在该进程的用户文件描述符表中查找最小可用表项,并将该表项的索引作为文件描述符返回。所以 open()函数总是得到一个最小可用的文件描述符。

2. 打开方式 flags

当用户使用 open()打开一个文件时,可以通过参数 flags 指定不同的打开方式。首先,打开方式 flags 必须取只读方式(O_RDONLY)、只写方式(O_WRONLY)和读写方式(O_RDWR)三者之一。除此之外,flags 还可以再增加其他的打开方式,如追加方式(O_APPEND)等等。打开方式 flags 取值如表 4.5 所示。

表 4.5　打开文件方式 flags

flags	含义	备注
O_RDONLY	只读方式	三者必选其一,且只能选其一
O_WRONLY	只写方式	
O_RDWR	读写方式	
O_CREAT	如果 pathname 指定的文件不存在就创建该文件。	
O_EXCL	与 O_CREAT 一起使用时,当 pathname 指定的文件已经存在,则 open()函数失败并返回一个错误	
O_NOCTTY	如果 pathname 是终端设备,则不会把该终端设备当成进程控制终端	
O_TRUNC	如果 pathname 指定的文件是已经存在的普通文件(regular file),并且打开的方式是可写的(如 O_RDWR 或者 O_WRONLY),就将文件的长度截为 0。对于 FIFO 文件或者终端设备文件,该方式将被忽略。	
O_APPEND	以追加的方式打开文件。在每一次调用 write()写文件之前,文件指针将被自动置到文件末尾,该方式保证对文件的写都是在文件末尾进行追加。	
O_NONBLOCK	以不可阻断的方式打开文件。无论有无数据读取或等待,都会立即返回进程中。	同 O_NDELAY
O_SYNC	文件以同步 I/O 的方式打开。这样每一次对文件的写操作之后进程都会阻塞直到数据物理地写到存储设备上。	
O_NOFOLLOW	如果 pathname 是一个符号链接文件,则 open()将执行失败。这是 FreeBSD 中的一个功能,后来加入到 Linux 2.1.126 中	
O_DIRECTORY	如果 pathname 不是一个目录,则 open()将执行失败。	

东软电子出版社

表 4.6 是文件打开方式 flags 几种常见的取值。

表 4.6 文件打开方式举例

flags 取值	含义
O_RDONLY	只读方式打开
O_RDWR|O_CREAT	如果文件存在就以读写方式打开,否则就创建文件
O_RDWR|O_CREAT|O_EXCL	如果文件不存在就创建,否则就返回错误
O_RDWR|O_CREAT|O_TRUNC	如果文件存在,则以读写方式打开,并将文件清空;否则创建该文件
O_WRONLY|O_APPEND	以只写方式打开文件,并且数据以追加的方式每次写入文件末尾
O_WRONLY|O_SYNC	以只写方式打开文件,并且每次写文件后等待数据真正写入磁盘后再返回进程

3. 新文件的权限 modes

系统调用 open()也可以用于创建新文件(当 flags 中设置了 O_CREAT 时)。此时 open()需要第三个参数 modes 设置新建文件的访问权限。如果 open()函数的 flags 中不包含 O_CREAT,则 modes 将会被忽略。

新建文件的最终权限与文件模式掩码有关。文件模式掩码 umask 与新建文件和目录的权限有关,八进制形式如 0022。如果使用 touch 命令新建文件,则新建文件的权限为“0666 &～umask”(新建文件默认不设置执行权限);而使用 mkdir 命令新建目录时,则新建目录的权限为“0777 &～umask”;如果使用系统调用函数 open 或 mkdir 创建新文件或目录时,新文件或目录的权限将是“权限参数 &～umask”。

文件模式掩码 umask 可以使用 umask 命令及 umask 函数修改。需要注意的是,umask 命令修改的是当前 Shell 进程的文件模式掩码,umask 函数修改的是当前执行进程的文件模式掩码。

open 的参数 modes 的取值既可以是八进制,如 0644,也可以是以下符号常量(宏),如表 4.7所示。

表 4.7 文件访问权限

符号常量(宏)	八进制权限值	权限含义
S_IRWXU	00700	文件主人有读(R)写(W)执行(X)权限
S_IRUSR	00400	文件主人有读(R)权限
S_IWUSR	00200	文件主人有写(W)权限
S_IXUSR	00100	文件主人有执行(X)权限
S_IRWXG	00070	用户组有读(R)写(W)执行(X)权限
S_IRGRP	00040	用户组有读(R)权限
S_IWGRP	00020	用户组有写(W)权限
S_IXGRP	00010	用户组有执行(X)权限
S_IRWXO	00007	其他用户有读(R)写(W)执行(X)权限
S_IROTH	00004	其他用户有读(R)权限
S_IWOTH	00002	其他用户有写(W)权限
S_IXOTH	00001	其他用户有执行(X)权限

东软电子出版社

例如：如果想要创建一个权限为“主人可以读写，组及其他用户只能读”的文件，则 modes 应该为“S_IRUSR| S_IWUSR| S_IRGRP| S_IROTH”或者为 0644。可以看出，以八进制表示文件权限比较简练；而以符号常量方式表示文件权限比较直观。

【知识验证】打开并得到文件描述符

下面给出打开文件的几个应用实例。

下列代码以只读的方式打开/etc/passwd 文件：

```
int fd = open( "/etc/passwd", O_RDONLY);
if ( fd = = -1)
    perror("open");
```

如果要以读写的方式打开“/home/mydata”，且如果该文件不存在就创建它，权限为“主人可以读写，组及其他用户只能读”。成功打开文件后，内存中三个相关数据结构的内容如图 4.5 所示。

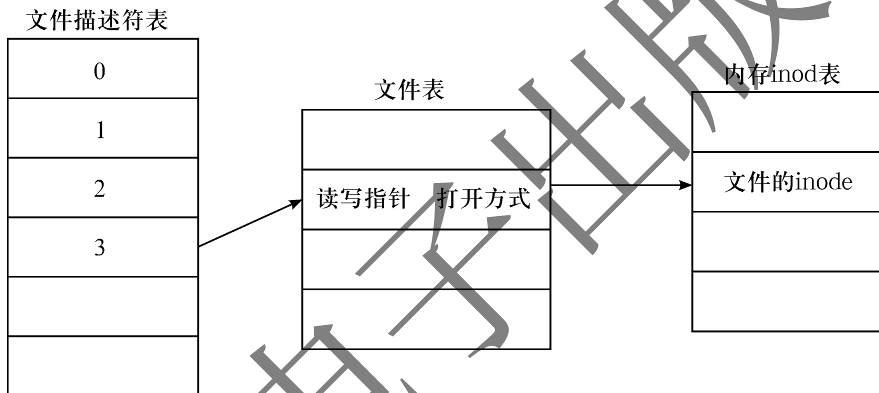

图 4.5　执行后内存数据结构的内容

```
int fd = open( "/home/mydata", O_RDWR|O_CREAT, S_IRUSR| S_IWUSR| S_IRGRP| S_IROTH);
if ( fd = = -1)
      perror( "open");
```

如果要创建文件“/home/mydata”用于读写，权限为“主人可以读写，组及其他用户只能读”，但如果该文件已经存在就返回错误。

```
int fd = open( "/home/mydata", O_RDWR|O_CREAT|O_EXCL, S_IRUSR| S_IWUSR| S_IRGRP| S_IROTH);
if ( fd = = -1)
     perror( "open");
```

【知识验证】打开文件并得到文件描述符

假设当前目录下有名为“f1”、“f2”、“f3”的三个文件，下面的代码打开这些文件，得到得到文件描述符。

(1)源程序(fdtest.c)。

```
1.#include <fcntl.h>
2.main()
3.{
4.    int fd1,fd2,fd3;
```

华软电子出版社

```
5.     fd1=open("f1",O_RDWR);
6.     fd2=open("f2",O_RDWR);
7.     fd3=open("f3",O_RDWR);
8.     printf("fd1= %d\nfd2= %d\nfd3= %d\n",fd1,fd2,fd3);
9.     close(fd1);
10.     close(fd2);
11.     close(fd3);
12.}
```

(2)编译。

```
#gcc  -o  fdtest   fdtest.c
```

(3)运行。

```
#./fdtest
```

(4)运行结果。

输出的结果为：

```
fd1=3
fd2=4
fd3=5
```

当三个 open()语句执行后，内存中数据结构的内容如图 4.6 所示。

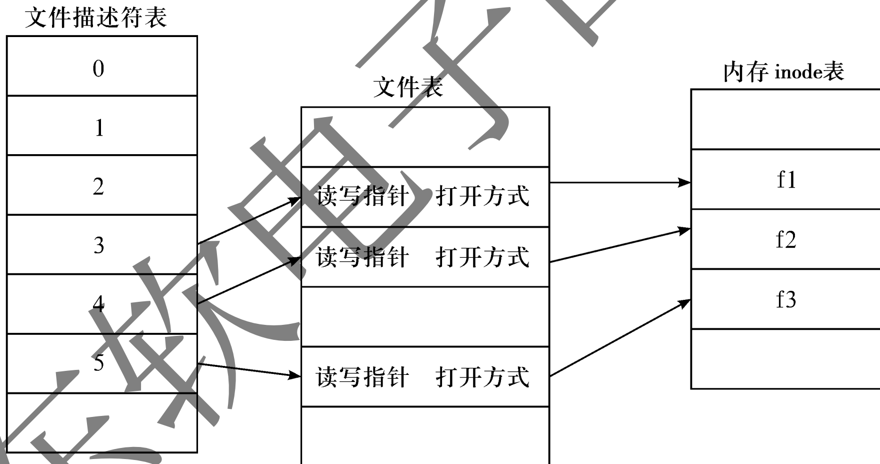

图 4.6 执行后内存数据结构的内容

东软电子出版社

对上述的程序如果作如下修改：

```
1.main()
2.{
3.     int fd1,fd2,fd3;
4.     fd1=open("f1",O_RDWR);
5.     fd2=open("f2",O_RDWR);
6.     printf("fd1= %d\nfd2= %d\n ",fd1,fd2);
7.     close(fd1);
8.     fd3=open("f3",O_RDWR);
```

```
9.    printf("fd3=%d\n",fd3);
10.    close(fd2);
11.    close(fd3);
12.}
```

则执行结果为：

```
fd1=3
fd2=4
fd3=3
```

由此可见，一个进程默认已经打开了标准输入设备文件、标准输出设备文件和标准错误输出设备文件。他们的文件描述符分别是0，1和2。进程中使用open()打开的文件得到的描述符从3开始，所以打开文件f1、f2、f3时得到的描述符是3、4和5。

open()每次打开文件总是得到一个最小的可用文件描述符。当进程打开f1、f2文件后得到的文件描述符是3和4。当执行close()关闭f1文件后，文件描述符3被释放变为可用。此时再执行open()打开f3文件时，得到的最小可用文件描述符是3。

4. 创建文件 creat()

系统调用creat()用于创建新文件。creat()函数与flags参数为O_CREAT|O_WRONLY|O_TRUN的open()函数作用相同。

4.3.2 读文件

当使用系统调用open()成功地打开或者新建一个文件后，可以得到一个文件描述符。通过该文件描述符可以对打开的文件进行访问，包括读和写。读文件使用系统调用read()。

表4.8 系统调用read

项目	描述
头文件	#include <unistd.h>
原型	ssize_t read(int fd, void *buf, size_t count);
功能	该函数从文件描述符为fd的文件中读取count个字节的数据并放入buf缓冲区中
参数	fd:要读文件的文件描述符。在执行read()之前，应该执行open/creat打开或创建该文件，并得到该文件的文件描述符 buf:指向接收数据缓冲区的指针，读出的数据将存放在该缓冲区中 count:要读取数据的字节数
返回值	整型结果:执行成功时，返回的是实际读回数据的字节数；当失败时，返回-1，并且errno为错误码

当read()执行成功时，read()返回的是实际读回数据的字节数，文件的读写指针会随之移动。此时会有以下几种情况：

● 返回值与count相等：实际读回的字节数与要读的字节数相等。读文件成功，并读回了想要字节数的数据。

● 返回值小于count：实际读回的字节数小于要读的字节数。一般为读写指针接近文件末尾，文件剩余数据不够count个字节，此次read将剩余数据读出，并将实际读出字节数返回。

华软电子出版社

● 返回值为 0：表明从文件末尾读（所以读回来 0 个字节的数据）。

当 read 读文件失败时，返回−1，并且 errno 根据错误的原因被设置为相应的错误码，如表 4.9 所示。

表 4.9 read 失败时的错误码

错误码	含义
EINTR	在数据读取之前，read 被信号中断
EAGAIN	选择了非阻塞 I/O 方式()但暂时没有数据可读
EIO	I/O 错误
EISDIR	fd 是一个目录，不能对目录执行 read
EBADF	fd 不是一个有效的文件描述符，或者 fd 不是为读打开的
EFAULT	buf 超出了可访问的地址空间范围

对于文件的读写，每个文件都有一个读写指针，指向该文件下一次读写的位置。当打开一个文件时，读写指针为 0，即指向该文件开始位置。每次执行读写操作后，如果成功，读写指针会自动移动。例如刚打开一个文件后，读写指针为 0，接下来执行 read()/write()从文件中读 10 字节/向文件中写 10 个字节，执行成功后读写指针为 10。

4.3.3 写文件

打开或者新建一个文件后，可以通过文件描述符对文件进行写操作，写文件使用系统调用 write()，如表 4.10 所示。

表 4.10 系统调用 write

项目	描述
头文件	#include <unistd.h>
原型	ssize_t write(int fd, const void *buf, size_t count);
功能	该函数将 buf 缓冲区中 count 个字节的数据写入文件描述符为 fd 的文件中
参数	fd：要写文件的文件描述符。在执行 write()之前，应该执行 open/creat 打开或创建该文件，并得到该文件的文件描述符 buf：指向要写数据缓冲区的指针，要写入文件的数据应先存放在该缓冲区中 count：要写入数据的字节数
返回值	整型结果：执行成功时，返回的是实际写入数据的字节数；当失败时，返回−1，并且 errno 为错误码

系统调用 write()返回一个整型结果。

当 write()执行成功时，write()返回的是实际写入数据的字节数。文件的读写指针会随之移动。如果返回值为 0，表明没有写入任何数据（实际写入字节数为 0）

当 write()写文件失败时，返回−1，并且 errno 根据错误的原因被设置为相应的错误码如

华软电子出版社

表 4.11 所示。

表 4.11　write 失败时的错误码

错误码	含义
ENOSPC	fd 指向的文件所在的设备没有足够的空间写入数据
EINTR	在数据写入之前,系统调用被信号中断
EINVAL	fd 指向的文件不能用于写
EIO	修改 inode 节点时发生了一个低级的 I/O 错误
EISDIR	fd 是一个目录,不能对目录执行 read
EBADF	fd 不是一个有效的文件描述符,或者 fd 不是为读打开的
EFAULT	buf 超出了可访问的地址空间范围

【知识验证】读写文件

打开文件 f1,将其所有内容读出并将数据写到屏幕上输出,每次从中读取 20 个字节的数据。

(1)源程序(filerw.c)。

```
1.  #include <fcntl.h>
2.  main()
3.  {
4.        int fd;
5.        int num;
6.        char buf[20];
7.        fd=open("f1",O_RDONLY);
8.        if(fd= = -1)
9.        {
10.           perror("open");
11.           exit(1);
12.       }
13.       while((num=read(fd,buf,20))>0)
14.       {
15.           if(write(1,buf,num)<num)
16.           printf("write 1 less than should\n");
17.       }
18.       close(fd);
19. }
```

在程序 filerw.c 中,以只读方式打开源文件 f1,如果打开成功,则进入一个循环。循环每次通过 read()从 f1 文件中读取 20 字节的数据,并写入屏幕设备文件(文件描述符为 1 的标准输出设备文件)中。当 read()返回的结果等于 0(读到 f1 文件末尾)或者等于-1(读 f1 文件失败)时循环结束。

在循环的过程中,如果向屏幕设备文件写入的字节数不等于从 f1 文件中读出的字节数时,给出相应提示。

华软电子出版社

循环结束后，关闭 f1 文件。

(2)编译。

```
#gcc  -o  filerw  filerw.c
```

(3)运行。

```
#./ filerw
```

4.3.4 文件定位

如前所述，每一个打开的文件都有一个读写指针指向下一次读写操作的位置。改变读写指针的位置可以使用系统调用 lseek()，如表 4.12 所示。

表 4.12 系统调用 lseek

项目	描述
头文件	#include <sys/types.h> #include <unistd.h>
原型	off_t lseek(int fd, off_t offset, int whence);
功能	该函数将文件描述符为 fd 的文件读写指针按照 whence 方式移动 offset 字节的偏移量
参数	fd:要改变读写指针位置的文件的文件描述符。在执行 lseek()之前，应该执行 open/creat 打开或创建该文件，并得到该文件的文件描述符 offset:要改变的读写指针的偏移量(字节数)，可以是正值、负值或 0 whence:移动读写指针的方式，可以为 SEEK_SET、SEEK_CUR 或 SEEK_END 之一
返回值	整型结果:执行成功时，返回的是定位后新的读写指针位置;当失败时，返回 -1，并且 errno 为错误码

whence 的取值如表 4.13 所示。

表 4.13 whence 的取值

whence 参数	读写指针移动方式	读写指针移动结果
SEEK_SET	从文件开始位置移动	即为 offset 的值
SEEK_CUR	从文件当前位置移动	当前读写指针的值+offset
SEEK_END	从文件末尾位置移动	文件长度+offset

系统调用 lseek()返回一个整型结果。当 lseek 执行成功时，该结果是定位后新的读写指针位置。否则当 lseek()执行失败时返回 -1，并在 errno 中设置相应的错误码，错误码如表 4.14 所示。

表 4.14 lseek 失败时的错误码

错误码	含义
EBADF	fd 不是一个有效的文件描述符
ESPIPE	fd 是一个管道，Socket 或者 FIFO 的文件描述符
EINVAL	whence 的值不正确

通过系统调用 lseek()对文件的读写指针进行定位，要保证移动后的指针位置是非负数。

东软电子出版社

例如一个长度为 100 字节的文件打开后,文件描述符为 fd,读写指针当前的位置在第 50 字节。下列操作都是正确的:

```
lseek( fd, 20, SEEK_SET);      //从文件开始位置移动 20 字节,移动后指针为 20
lseek( fd, 120, SEEK_SET);     //从文件开始位置移动 120 字节,移动后指针为 120
lseek( fd, 20, SEEK_CUR);      //从文件当前位置移动 20 字节,移动后指针为 70
lseek( fd, -20, SEEK_CUR);     //从文件当前位置移动-20 字节,移动后指针为 30
lseek( fd, 60, SEEK_CUR);      //从文件当前位置移动 60 字节,移动后指针为 110
lseek( fd, 20, SEEK_END);      //从文件末尾位置移动 20 字节,移动后指针为 120
lseek( fd, -20, SEEK_END);     //从文件末尾位置移动-20 字节,移动后指针为 80
```

而下列操作都是错误的:

```
lseek( fd, -20, SEEK_SET);  //从文件开始位置移动-20 字节
lseek( fd, -60, SEEK_CUR);  //从文件当前位置移动-60 字节
lseek( fd, -120, SEEK_END);  //从文件末尾位置移动-120 字节
```

也就是说可以将文件的读写指针移动到超过文件末尾再往后的位置,但不能将指针移动到文件开始再往前的位置。

通过 lseek()移动了读写指针后,下一次的读写操作就会从新的指针位置开始进行。

在 UNIX/Linux 中,允许将读写指针定位到超过文件长度的位置。如上面的举例,一个文件长度为 100 字节,可以将读写指针移动到第 120 字节处。此时对文件的读写操作将如下处理:

- 读文件 read(fd, buf, 20);从文件当前位置读 20 个字节到 buf 中,返回结果为 0,即当前指针已经是(超过)文件末尾,实际读回的字节数为 0 个。
- 写文件 write(fd, buf, 20),将 buf 中 20 个字节的数据写入文件当前位置。返回结果 20,即实际写入字节数 20(从文件第 120 字节处写入),写入后文件长度为 140,其中文件第 100 字节至第 120 字节由 0 填充。

对于读写指针已经超过文件长度时执行写操作,UNIX/Linux 系统会先将文件的长度延长到当前位置,然后再写入。延长的部分用 0 来填充,成为空洞。在存储该文件时,空洞部分并不占据存储空间。

4.3.5　关闭文件

当一个打开的文件使用完之后,可以使用 close()关闭该文件,如表 4.15 所示。

表 4.15　系统调用 close

项目	描述
头文件	#include <unistd.h>
原型	int　close(int fd);
功能	该函数关闭文件描述符为 fd 的文件。
参数	fd:要关闭的文件的文件描述符
返回值	如果成功的关闭,返回 0;如果关闭文件失败,则返回-1,并且 errno 为错误码。

电子工业出版社

系统调用 close()返回一个整型结果。如果成功关闭，返回 0；如果关闭文件失败，则返回 −1，同时 errno 中设置错误码，如表 4.16 所示。

表 4.16　close 失败时的错误码

错误码	含义
EBADF	fd 不是一个有效的打开文件的文件描述符
EINTR	系统调用 close()被信号中断
EIO	发生了一个 I/O 错误

人们在编写程序时经常忘记检查 close()的返回值，这样做会有一些问题。很有可能在前面对文件执行 write()操作时出现错误，但在执行 close()才报告，如果不检查 close()的返回值，则可能漏掉这样的出错信息。不检查 close()的返回值有可能导致文件数据的丢失。

打开文件时，系统会根据文件名找到文件的索引节点，并将文件的相关信息填入系统用于管理打开文件的数据结构中；关闭文件时，系统执行相反的操作，将相关数据结构中关于该文件的登记项信息释放。

另外，在关闭文件时，系统会将先前执行写操作但数据仍在缓冲区中尚未写入文件的数据真正写入文件。如果要确保数据真正写入文件，也可调用 fsync()。

4.3.6　文件操作举例

【知识验证】文件基本 I/O 操作

编写一个程序，两次打开同一个文件进行读写操作，并观察结果。

(1)源程序(fileopenone.c)。

```
1. #include <unistd.h>
2. #include <fcntl.h>
3. main()
4. {
5.     int fd1,fd2;
6.     int num;
7.     char buf[20];
8.     fd1 = open("f1",O_RDWR|O_TRUNC);
9.     if(fd1==-1)
10.     {
11.         perror("open");
12.         exit(1);
13.     }
14.     printf("fd1 is %d \n",fd1);
15.     fd2 = open("f1",O_RDWR);
16.     if(fd2==-1)
17.     {
18.         perror("open");
```

东软电子出版社

```
19.         exit(1);
20.     }
21.     printf("fd2 is %d \n",fd2);
22.     num=write(fd1,"hello world!",12);
23.     printf("write num= %d bytes into f1\n",num);
24.     num=read(fd2, buf,20);
25.     buf[num]=0;
26.     printf("read   %d bytes from f1: %s \n ",num, buf);
27.     close(fd1);
28.     close(fd2);
29.}
```

(2)编译。

```
#gcc  -o  fileopenone   fileopenone.c
```

(3)运行。

```
#./fileopenone
```

(4)运行结果。

输出的结果为：

```
fd1 is 3
fd2 is 4
write num=12 bytes into f1
read  12 bytes from f1: hello world!
```

在程序 fileopenone.c 中，两次使用 open()打开文件 f1。虽然两次打开的是同一个文件，但由于这是两次独立的 open()操作，每一次 open()操作打开文件时，都会在系统的打开文件表中登记一个不同的表项，并在该进程的用户文件描述符表中也会登记一个不同的表项。因此两次 open()操作得到了两个不同的文件描述符(在此例中 fd1 为 3，fd2 为 4)。这两个文件描述符对应着文件表中的两个不同的表项。由于文件的读写指针保存在文件表表项中，因此两个文件描述符都拥有各自独立的文件读写指针。

两次 open()执行之后，两个文件描述符对应的读写指针都指向文件 f1 的开始位置。此时通过文件描述符 fd1 向文件 f1 中写入了 12 个字符“hello world!”，成功写入之后其读写指针为 12。而这时另一个文件描述符 fd2 对应的读写指针仍为 0，指向文件的开始位置。因此接下来通过 fd2 从文件中读数据时，会将刚才写入的字符串读出。

【知识验证】文件基本 I/O 操作

编写一个父子进程访问同一个文件的程序。

(1)源程序(filefork.c)。

```
1. #include <fcntl.h>
2. #include <unistd.h>
3. main()
4. {
5.     int fd;
6.     int num;
```

华东电子出版社

```
7.     pid_t pid;
8.     char buf[20];
9.     fd=open("f1",O_RDWR|O_TRUNC|O_CREAT,644);
10.     if(fd==-1)
11.     {
12.         perror("open");
13.         exit(1);
14.     }
15.     write(fd,"helloworld",10);
16.     lseek(fd,0,SEEK_SET);
17.     if((pid=fork())<0)
18.     {
19.         perror("fork");
20.     }
21.     else if(pid==0)
22.     {
23.         num=read(fd,buf,5);
24.         buf[num]=0;
25.         printf("i am child,my pid=%d, read num=%d bytes from f1(fd=%d):%s\n",
getpid(),num,fd,buf);
26.     }
27.     else
28.     {
29.         num=read(fd,buf,5);
30.         buf[num]=0;
31.         printf("i am parent,my pid=%d, read num=%d bytes from f1(fd=%d):%s\n",
getpid(),num,fd,buf);
32.         wait();
33.     }
34.     close(fd);
35. }
```

(2)编译。

```
#gcc -o filefork filefork.c
```

(3)运行。

```
#./filefork
```

(4)运行结果。

```
i am child,my pid=1935, read num=5 bytes from f1(fd=3):hello
i am parent,my pid=1934, read num=5 bytes from f1(fd=3):world
```

在程序 filefork.c 中,父进程通过 open()创建了 f1 文件,得到文件描述符 fd=3,并向文件中写入 10 个字符“helloworld”。写入后通过 lseek()将文件读写指针定位于文件开始位置。

当父进程通过 fork()创建了子进程时,子进程从父进程那里继承来了用户文件描述符表,也就是说父子进程此时拥有内容一样的用户文件描述符表。在子进程中,也有一个文件描述符

东软电子出版社

3，它与父进程的文件描述符 3 同样指向文件表中的同一个表项。由于文件的读写指针保存在文件表的表项中，所以父子进程不仅拥有同样的文件描述符，它们的文件描述符也对应着同一个文件读写指针。当父进程先从文件中读取了 5 个字符“hello”后，读写指针改变为 5，此时子进程在从文件中读数据时，由于使用的是和父进程相同的读写指针，因此将接下来的 5 个字符“world”读了出来。

此处涉及的父子进程相关知识将在第 7 章进行详细介绍。

4.4　文件访问的同步

系统调用 sync、fsync 和 fdatasync 用于文件访问时的同步如表 4.17 和表 4.18 所示。

表 4.17　系统调用 sync

项目	描述
头文件	＃include ＜unistd. h＞
原型	void sync(void)；
功能	将缓冲区中的内容写入磁盘
参数	无
返回值	无

表 4.18　系统调用 fsync/ fdatasync

项目	描述
头文件	＃include ＜unistd. h＞
原型	int fsync(int fd)； int fdatasync(int fd)；
功能	将指定文件的内容写入磁盘
参数	fd：要同步的文件的文件描述符
返回值	如果成功，返回 0；如果失败，则返回－1，并且 errno 为错误码

当用户调用 write()向文件中写数据时，操作系统并不是立刻将数据真正写入磁盘，而是将数据先写入了磁盘缓冲区中，待事后再将磁盘缓冲区中的内容更新到磁盘上。这样做可以提高读写磁盘的效率，但是会出现磁盘文件与缓冲区中暂时不一致的情况。如果用户程序执行了 write()后，数据还没有更新到磁盘上时系统崩溃，则会造成数据丢失。

sync()可以强制将缓冲区中所有未更新到磁盘的数据立刻更新到磁盘上。fsync()只更新缓冲区中指定文件的内容，包括该文件的数据部分和索引节点中改变了但尚未更新的内容。fdatasync()则只更新缓冲区中指定文件的数据部分。

华软电子出版社

4.5　项目:文件复制命令的实现

4.5.1　项目分析与设计

在最终的“Linux 网络传输系统”项目中,我们要实现将服务器端收到的数据保存到文件中这个功能。这个功能其实就是将接收数据写入一个文件中,将涉及 Linux 系统中基本文件 I/O 操作。在本节,我们通过开发一个复制文件的程序来掌握这些技能。

在 UNIX 中,基本文件的访问过程分为如下步骤:

(1)打开文件/创建新文件。

(2)访问文件(包括定位读写指针、读文件、写文件等操作)。

(3)最后关闭文件。

打开/创建文件使用系统调用 open()/creat();访问文件的操作包括读文件 read()、写文件 write()、定位文件读写指针 lseek()等;关闭文件使用系统调用 close()。

“文件复制命令的实现”项目将通过打开源文件、创建目标文件、从源文件中读出数据写入目标文件、关闭源文件和目标文件这样的过程实现文件的复制,这将用到上述的系统调用。

本项目中只是实现文件复制最基本的功能,其他复杂的功能读者可以在学习后续章节后自行实现。

复制文件就是将一个指定的源文件复制出一个与其一样的目标文件。源文件名和目标文件名由用户执行程序时在命令行以参数的形式给出。程序首先要打开源文件,并创建目标文件。然后从源文件中将数据读出,并写入目标文件中。最后分别关闭源文件和目标文件。

在读写源文件和目标文件时,需要定义一个缓冲区。将从源文件读出的数据存放在该缓冲区中,然后再将缓冲区的数据写入目标文件。由于首先定义的缓冲区可能小于文件的大小,因此可能需要循环重复多次,才能将源文件中的数据全部写入目标文件。

本项目程序流程图如图 4.7 所示。

东软电子出版社

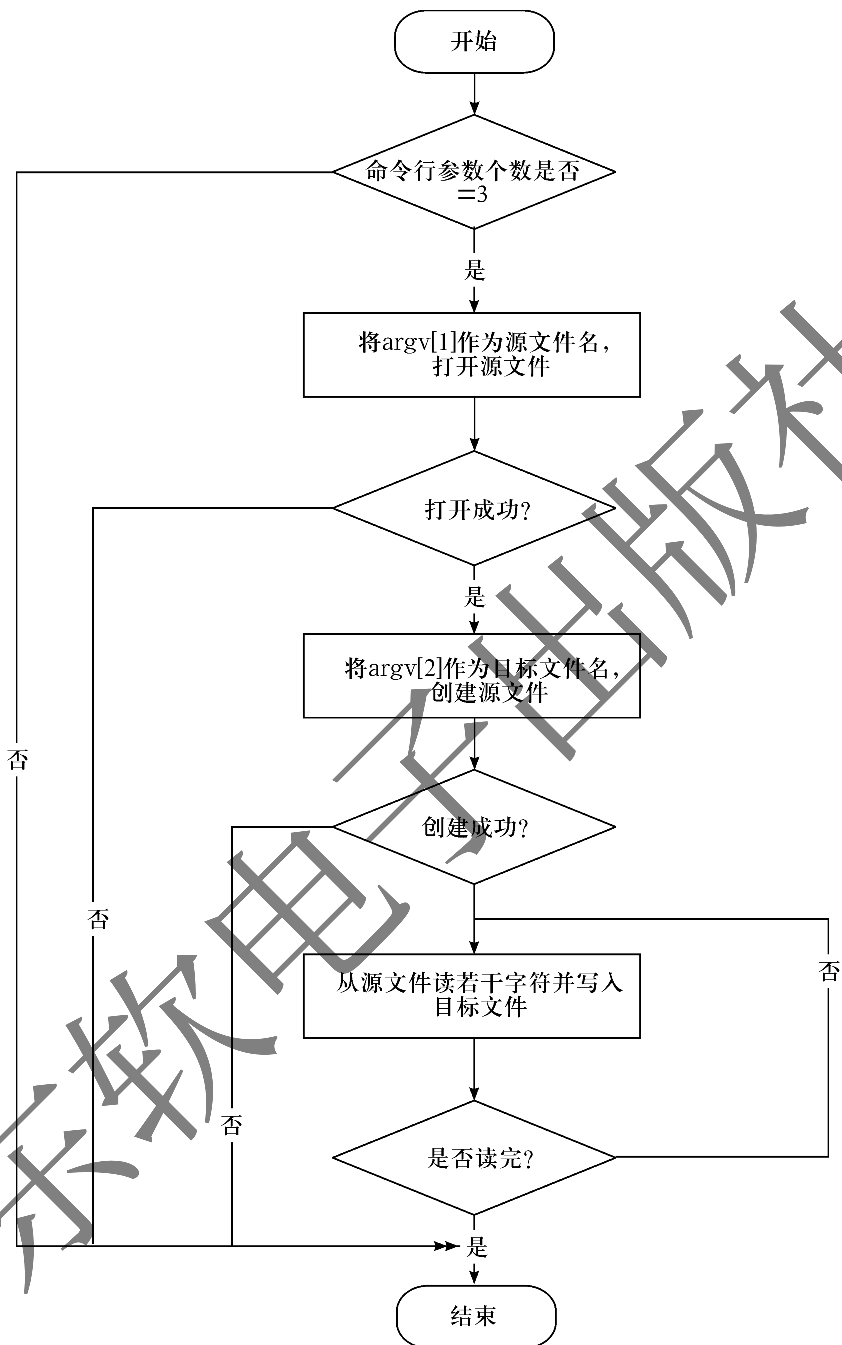

图 4.7　程序流程图

4.5.2　项目实施

本项目源代码(mycp.c)如下。

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

东软电子出版社

```
2. #include <sys/stat.h>
3. #include <fcntl.h>
4. #define BUFSIZE 512
5. void copy(char *from, char *to)
6. {
7.     int fromfd = -1, tofd = -1;
8.     ssize_t nread;
9.     char buf[BUFSIZE];
10.     if((fromfd = open(from, O_RDONLY))==-1)
11.     {
12.         perror("open");
13.         exit(1);
14.     }
15.     if((tofd=open(to, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR))==-1)
16.     {
17.         perror("open to"); exit(1);
18.     }
19.     nread = read(fromfd, buf, sizeof(buf));
20.     while (nread > 0)
21.     {
22.         if (write(tofd, buf, nread) != nread)
23.             printf("write %s error\n",to);
24.     nread = read(fromfd, buf, sizeof(buf));
25.     }
26.     if (nread == -1)
27.     printf("write %s error\n",to);
28.     close(fromfd);
29.     close(tofd);
30.     return;
31. }
32. main(int argc, char **argv)
33. {
34.     if(argc!=3)
35.     {
36.         printf("Usage:program  fromfilename  tofilename\n");
37.         exit(1);
```

东软电子出版社

```
38.     }
39.     copy(argv[1],argv[2]);
40. }
```

程序 mycp.c 实现复制文件的功能。该程序执行时需要带两个参数:源文件和目标文件名。因此在 main()函数中包含两个参数 argc 和 argv。当参数个数不等于 3 时(argc 不等于 3)给出程序执行的提示并返回。否则调用函数 copy 完成复制文件功能。

程序运行时将源文件名(argv[1])和目标文件名(argv[2])作为参数传入函数 copy()中。在 copy()中,首先以只读方式打开源文件,然后以只写方式创建目标文件,如果目标文件已经存在则清空。接下来进入一个循环,不断从源文件读出 512 字节数据写入目标文件,直至源文件数据全部写入目标文件。

4.5.3　项目编译与运行

(1)编译。

```
#gcc -o mycp  mycp.c
```

(2)运行。

如果当前目录下有 f1 文件,先要将其复制为 f2 文件,则程序如下执行:

```
#./mycp  f1  f2
```

该项目每次从源文件中读取 512 字节数据并写入目标文件。每次读写的字节数对项目的效率有何影响?读者可以将缓冲区的大小进行修改,然后观察执行情况加以分析。

[教学效果测评]

一、选择题

1. 文件系统保存在磁盘的(　　)。

A. 引导块　B. 超级块　C. i 节点块　D. 数据块

2. linux 文件系统的根目录的 i 节点号为(　　)。

A. 0　B. 1　C. 2　D. 3

3. 文件描述符的数据类型为(　　)。

A. char　B. int　C. double　D. float

4. 设置文件偏移量的系统调用是(　　)。

A. truncate　B. sync　C. lseek　D. creat

5. 下面哪个不是 lseek 第三个参数的取值(　　)?

A. SEEK_SET　B. SEEK_CUR　C. SEEK_NOW　D. SEEK_END

6. sync 系统调用的功能是(　　)。

A. 刷新所有缓存到磁盘　B. 刷新缓存中某个文件的所有信息到磁盘

C. 刷新缓存中某个文件的数据到磁盘　D. 刷新缓存中某个文件的属性信息到磁盘

7. fsync 系统调用的功能是(　　)。

A. 刷新所有缓存到磁盘　B. 刷新缓存中某个文件的所有信息到磁盘

C. 刷新缓存中某个文件的数据到磁盘　D. 刷新缓存中某个文件的属性信息到磁盘

东软电子出版社

8. fdatasync 系统调用的功能是(　　)。

A. 刷新所有缓存到磁盘　　B. 刷新缓存中某个文件的所有信息到磁盘

C. 刷新缓存中某个文件的数据到磁盘　　D. 刷新缓存中某个文件的属性信息到磁盘

二、填空题

1. Linux 系统下,表示标准输入、标准输出和标准错误输出的文件描述符(符号表示)为__________、__________和__________,它们的值分别为__________、__________、__________。

2. 数字 635 表示的权限使用字母方式表示为__________,使用符号方式表示为__________。

3. 系统调用 open 的功能是__________。

4. 使用 open 打开文件时有三个标志必须要选择其一,这三个标志是__________、__________、__________。

5. 文件偏移量代表________________。

6. 将文件偏移量设置为当前偏移处之前的 4 个字节的位置,使用 lseek(fd,__________,__________)。

7. 设置打开文件标志__________可以截短文件为 0,使用系统调用__________可以截短或加长文件。

8. 如果 umask 设为 022,则创建一个新文件的权限(数字表示)为__________,创建一个新目录的权限(数字表示)为__________。

9. 如果 umask 设为 024,则创建一个新文件的权限(数字表示)为__________,创建一个新目录的权限(数字表示)为__________。

三、简答题

1. Linux 文件类型主要有哪七类?

2. 简述文件、i 节点、文件名、目录之间的关系。

3. 什么是文件描述符?

4. 使用符号方式表示 rwxrwxrwx 权限。

5. 写出 open 以下六种打开标志:只读、只写、读写、追加、文件不存在创建、截短为 0。

6. 读程序,写出执行结果,并解释得到该结果的原因。

```
main()
{
    int fd1,fd2;
    fd 1= open("/etc/passwd", O_RDONLY);
    fd 2= open("/etc/passwd", O_RDWR);
    printf("fd1 =  %d   fd2= %d\n",fd1,fd2);
    close(fd);
    close(fd2);
}
```

华软电子出版社

四、编程题

1. 向文件 f1 中写入"hello world!",然后再将 f1 中的内容读出并显示在屏幕上。(注意必要的错误判断)

2. 向文件 f2 中写入"aabbccddee",然后将偏移量移到绝对偏移为 4 的位置处,读 6 个字符,并将结果显示在屏幕上。

3. 向文件 f3 中写入"aabbccddeeffgghh",然后将文件截短至 8 个字节,然后将截短后的文件内容读出并显示在屏幕上。

4. 在程序中将 umask 改至 044,创建文件 f4。

5. 实现"cat 文件名"显示文件内容。

6. 实现"cp　原文件　目标文件"。

东软电子出版社