第三章 深入探究文件 I/O

经过上一章内容的学习，相信各位读者对 Linux 系统应用编程中的基础文件 I/O 操作有了一定的认识和理解了，能够独立完成一些简单地文件 I/O 编程问题，如果你的工作中仅仅只是涉及到一些简单文件读写操作相关的问题，其实上一章的知识内容已经够你使用了。

当然作为大部分读者来说，我相信你不会止步于此、还想学习更多的知识内容，那本章笔者将会同各位读者一起，来深入探究文件 I/O 中涉及到的一些问题、原理以及所对应的解决方法，譬如 Linux 系统下文件是如何进行管理的、调用函数返回错误该如何处理、open 函数的 O_APPEND、O_TRUNC 标志以及等相关问题。

好了，废话不多说，开始本章的学习吧，加油！

本章将会讨论如下主题内容。

- 对 Linux 下文件的管理方式进行简单介绍；
- 函数返回错误的处理；
- 退出程序 exit()、_Exit()、_exit()；
- 空洞文件的概念；
- open 函数的 O_APPEND 和 O_TRUNC 标志；
- 多次打开同一文件；
- 复制文件描述符；
- 文件共享介绍；
- 原子操作与竞争冒险；
- 系统调用 fcntl()和 ioctl()介绍；
- 截断文件；

1.1 Linux 系统如何管理文件

1.1.1 静态文件与 inode

文件在没有被打开的情况下一般都是存放在磁盘中的，譬如电脑硬盘、移动硬盘、U 盘等外部存储设备，文件存放在磁盘文件系统中，并且以一种固定的形式进行存放，我们把他们称为静态文件。

文件储存在硬盘上，硬盘的最小存储单位叫做“扇区”（Sector），每个扇区储存 512 字节（相当于 0.5KB），操作系统读取硬盘的时候，不会一个个扇区地读取，这样效率太低，而是一次性连续读取多个扇区，即一次性读取一个“块”（block）。这种由多个扇区组成的“块”，是文件存取的最小单位。“块”的大小，最常见的是 4KB，即连续八个 sector 组成一个 block。

所以由此可以知道，静态文件对应的数据都是存储在磁盘设备不同的“块”中，那么问题来了，我们在程序中调用 open 函数是如何找到对应文件的数据存储“块”的呢，难道仅仅通过指定的文件路径就可以实现？这里我们就来简单地聊一聊这内部实现的过程。

我们的磁盘在进行分区、格式化的时候会将其分为两个区域，一个是数据区，用于存储文件中的数据；另一个是 inode 区，用于存放 inode table（inode 表），inode table 中存放的是一个一个的 inode（也成为 inode 节点），不同的 inode 就可以表示不同的文件，每一个文件都必须对应一个 inode，inode 实质上是一个结构体，这个结构体中有很多的元素，不同的元素记录了文件了不同信息，譬如文件字节大小、文件所有者、文件对应的读/写/执行权限、文件时间戳（创建时间、更新时间等）、文件类型、文件数据存储的 block（块）位置等等信息，如图 3.1.1 中所示（这里需要注意的是，文件名并不是记录在 inode 中，这个问题后面章节内容再给大家讲）。

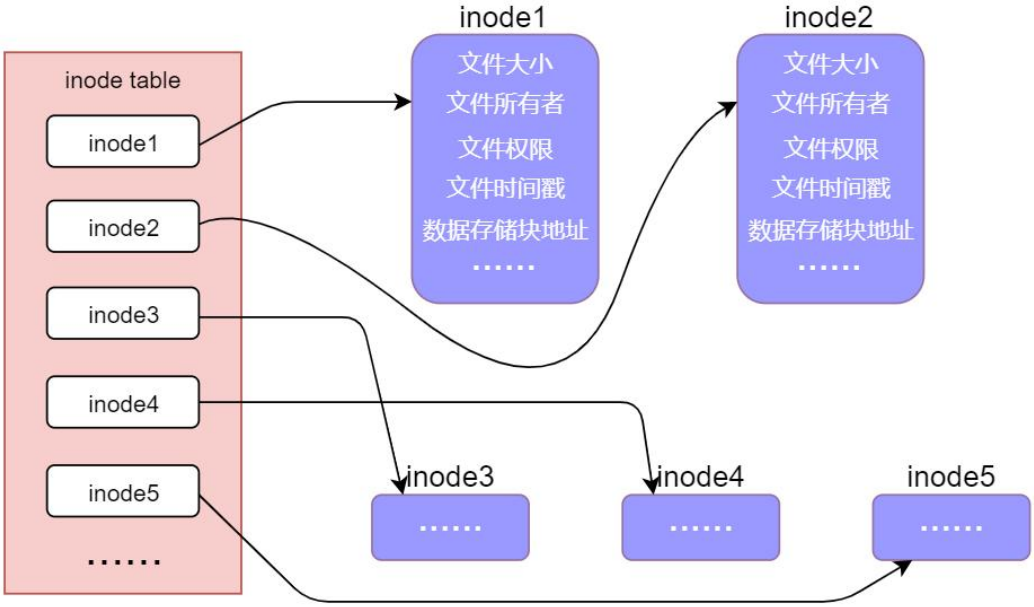

图 3.1.1 inode table 与 inode

所以由此可知，inode table 表本身也需要占用磁盘的存储空间。每一个文件都有唯一的一个 inode，每一个 inode 都有一个与之相对应的数字编号，通过这个数字编号就可以找到 inode table 中所对应的 inode。在 Linux 系统下，我们可以通过"ls -i"命令查看文件的 inode 编号，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$ ls -il
总用量 16
3701769 -rw-rw-r-- 1 dt dt 1583 1月    5 19:54 testApp_1.c
3701836 -rw-rw-r-- 1 dt dt  709 1月    5 20:07 testApp_2.c
3701854 -rw-rw-r-- 1 dt dt 1381 1月    5 20:23 testApp_3.c
3702154 -rw-rw-r-- 1 dt dt  800 1月    5 20:33 testApp_4.c
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
```

图 3.1.2 ls 查看文件的 inode 编号

上图中 ls 打印出来的信息中，每一行前面的一个数字就表示了对应文件的 inode 编号。除此之外，还可以使用 stat 命令查看，用法如下：

```
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
dt@dt-virtual-machine:~/vscode_ws/1_chapter$ stat testApp_1.c
  文件：'testApp_1.c'
  大小：1583            块：8          IO 块：4096   普通文件
设备：801h/2049d        Inode: 3701769     硬链接：1
权限：(0664/-rw-rw-r--)  Uid: ( 1000/      dt)   Gid: ( 1000/      dt)
最近访问：2021-01-05 20:07:30.925816725 +0800
最近更改：2021-01-05 19:54:20.012237341 +0800
最近改动：2021-01-05 19:54:20.012237341 +0800
创建时间：-
dt@dt-virtual-machine:~/vscode_ws/1_chapter$
```

图 3.1.3 stat 查看 inode 编号

由以上的介绍大家可以联系到实际操作中，譬如我们在 Windows 下进行 U 盘格式化的时候会有一个“快速格式化”选项，如下所示：

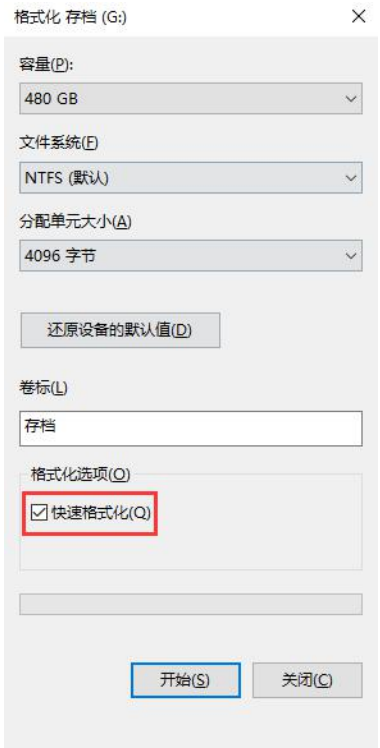

图 3.1.4 Windows 下格式化磁盘

如果勾选了“快速格式化”选项，在进行格式化操作的时候非常的快，而如果不勾选此选项，直接使用普通格式化方式，将会比较慢，那说明这两种格式化方式是存在差异的，其

实快速格式化只是删除了 U 盘中的 inode table 表，真正存储文件数据的区域并没有动，所以使用快速格式化的 U 盘，其中的数据是可以被找回来的。

通过以上介绍可知，打开一个文件，系统内部会将这个过程分为三步：

1) 系统找到这个文件名所对应的 inode 编号；
2) 通过 inode 编号从 inode table 中找到对应的 inode 结构体；
3) 根据 inode 结构体中记录的信息，确定文件数据所在的 block，并读出数据。

1.1.2 文件打开时的状态

当我们调用 open 函数去打开文件的时候，内核会申请一段内存（一段缓冲区），并且将静态文件的数据内容从磁盘这些存储设备中读取到内存中进行管理、缓存（也把内存中的这份文件数据叫做动态文件、内核缓冲区）。打开文件后，以后对这个文件的读写操作，都是针对内存中这一份动态文件进行相关的操作，而并不是针对磁盘中存放的静态文件。

当我们对动态文件进行读写操作后，此时内存中的动态文件和磁盘设备中的静态文件就不同步了，数据的同步工作由内核完成，内核会在之后将内存这份动态文件更新（同步）到磁盘设备中。由此我们也可以联系到实际操作中，譬如说：

- 打开一个大文件的时候会比较慢；
- 文档写了一半，没记得保存，此时电脑因为突然停电直接掉电关机了，当重启电脑后，打开编写的文档，发现之前写的内容已经丢失。

想必各位读者在工作当中都遇到过这种问题吧，通过上面的介绍，就解释了为什么会出现这种问题。好，我们再来说一下，为什么要这样设计？

因为磁盘、硬盘、U 盘等存储设备基本都是 Flash 块设备，因为块设备硬件本身有读写限制等特征，块设备是以一块一块为单位进行读写的（一个块包含多个扇区，而一个扇区包含多个字节），一个字节的改动也需要将该字节所在的 block 全部读取出来进行修改，修改完成之后再写入块设备中，所以导致对块设备的读写操作非常不灵活；而内存可以按字节为单位来操作，而且可以随机操作任意地址数据，非常地很灵活，所以对于操作系统来说，会先将磁盘中的静态文件读取到内存中进行缓存，读写操作都是针对这份动态文件，而不是直接去操作磁盘中的静态文件，不但操作不灵活，效率也会下降很多，因为内存的读写速率远比磁盘读写快得多。

在 Linux 系统中，内核会为每个进程（关于进程的概念，这是后面的内容，我们可以简单地理解为一个运行的程序就是一个进程，运行了多个程序那就是存在多个进程）设置一个专门的数据结构用于管理该进程，譬如用于记录进程的状态信息、运行特征等，我们把这个称为进程控制块（Process control block，缩写 PCB）。

PCB 数据结构体中有一个指针指向了文件描述符表（File descriptors），文件描述符表中的每一个元素索引到对应的文件表（File table），文件表也是一个数据结构体，其中记录了很多文件相关的信息，譬如文件状态标志、引用计数、当前文件的读写偏移量以及 i-node 指针（指向该文件对应的 inode）等，进程打开的所有文件对应的文件描述符都记录在文件描述符表中，每一个文件描述符都会指向一个对应的文件表，其示意图如下所示：

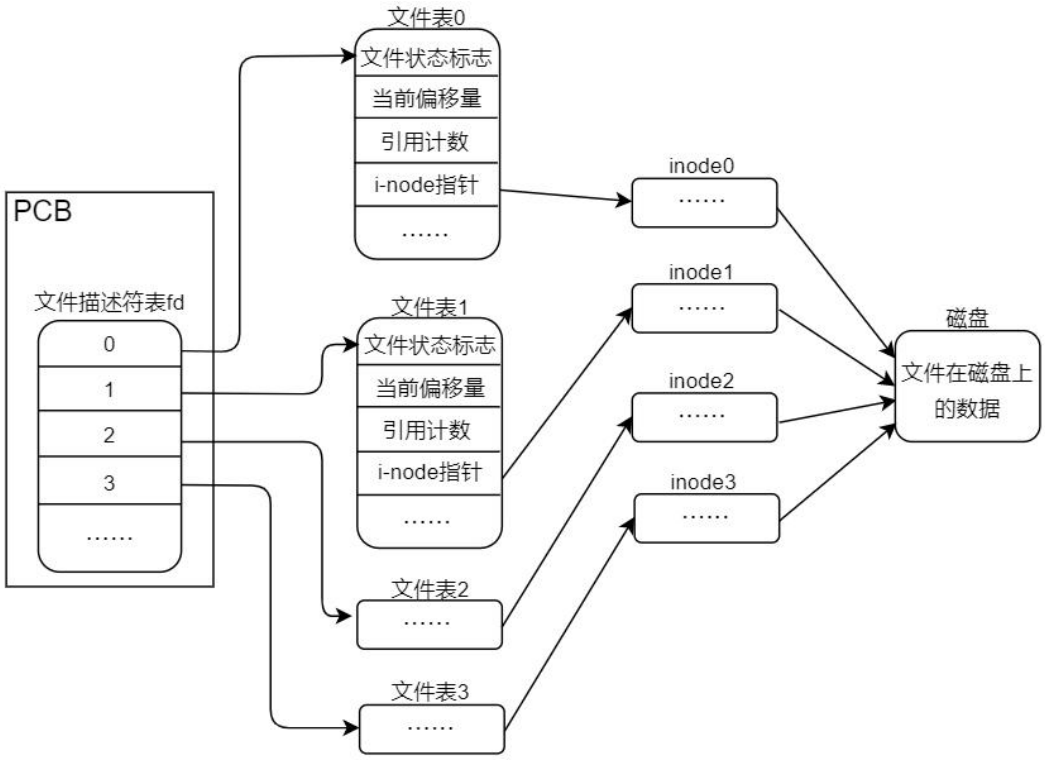

图 3.1.5 文件描述符表、文件表以及 inode 之间的关系

前面给大家介绍了 inode，inode 数据结构体中的元素会记录该文件的数据存储的 block（块），也就是说可以通过 inode 找到文件数据存在在磁盘设备中的那个位置，从而把文件数据读取出来。

以上就是本小节给大家介绍到所有内容了，上面给大家所介绍的内容后面的学习过程中还会用到，虽然这些理论知识对大家的编程并没有什么影响，但是会帮助大家理解文件 IO 背后隐藏的一些理论知识，其实这些理论知识还是非常浅薄的、只是一个大概的认识，其内部具体的实现是比较复杂的，当然这个不是我们学习 Linux 应用编程的重点，操作系统已经帮我们完成了这些具体的实现，我们要做的仅仅只是调用操作系统提供 API 函数来完成自己的工作。

好了，废话不多说，我们接着看下一小节内容。

1.2 返回错误处理与 errno

在上一章节中，笔者给大家编写了很多的示例代码，大家会发现这些示例代码会有一个共同的特点，那就是当判断函数执行失败后，会调用 return 退出程序，但是对于我们来说，我们并不知道为什么会出错，什么原因导致此函数执行失败，因为执行出错之后它们的返回值都是-1。

难道我们真的就不知道错误原因了吗？其实不然，在 Linux 系统下对常见的错误做了一个编号，每一个编号都代表着每一种不同的错误类型，当函数执行发生错误的时候，操作系统会将这个错误所对应的编号赋值给 errno 变量，每一个进程（程序）都维护了自己的 errno 变量，它是程序中的全局变量，该变量用于存储就近发生的函数执行错误编号，也就意味着下一次的错误码会覆盖上一次的错误码。所以由此可知道，当程序中调用函数发生错误的时候，操作系统内部会通过设置程序的 errno 变量来告知调用者究竟发生了什么错误！

errno 本质上是一个 int 类型的变量，用于存储错误编号，但是需要注意的是，并不是执行所有的系统调用或 C 库函数出错时，操作系统都会设置 errno，那我们如何确定一个函数

出错时系统是否会设置 errno 呢？其实这个通过 man 手册便可以查到，譬如以 open 函数为例，执行"man 2 open"打开 open 函数的帮助信息，找到函数返回值描述段，如下所示：

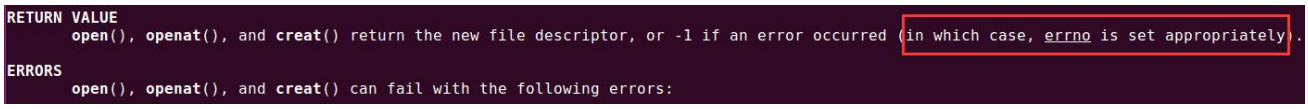

图 3.2.1 查看返回值描述信息

从图中红框部分描述文字可知，当函数返回错误时会设置 errno，当然这里是以 open 函数为例，其它的系统调用也可以这样查找，大家可以自己试试！

在我们的程序当中如何去获取系统所维护的这个 errno 变量呢？只需要在我们程序当中包含<errno.h>头文件即可，你可以直接认为此变量就是在<errno.h>头文件中的申明的，好，我们来测试下：

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

int main(void)
{
    printf("%d\n", errno);
    return 0;
}
```

以上的这段代码是不会报错的，大家可以自己试试！

1.2.1 strerror 函数

前面给大家说到了 errno 变量，但是 errno 仅仅只是一个错误编号，对于开发者来说，即使拿到了 errno 也不知道错误为何？还需要对比源码中对此编号的错误定义，可以说非常不友好，这里介绍一个 C 库函数 strerror()，该函数可以将对应的 errno 转换成适合我们查看的字符串信息，其函数原型如下所示（可通过"man 3 strerror"命令查看，注意此函数是 C 库函数，并不是系统调用）：

```
#include <string.h>

char *strerror(int errnum);
```

首先调用此函数需要包含头文件<string.h>。

函数参数和返回值如下：

errnum：错误编号 errno。

返回值：对应错误编号的字符串描述信息。

测试

接下来我们测试下，测试代码如下：

示例代码 3.2.1 strerror 测试代码

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

```
#include <string.h>

int main(void)
{
    int fd;

    /* 打开文件 */
    fd = open("./test_file", O_RDONLY);
    if (-1 == fd) {
        printf("Error: %s\n", strerror(errno));
        return -1;
    }

    close(fd);
    return 0;
}
```

编译源代码，在 Ubuntu 系统下运行测试下，在当前目录下并不存在 test_file 文件，测试打印结果如下：

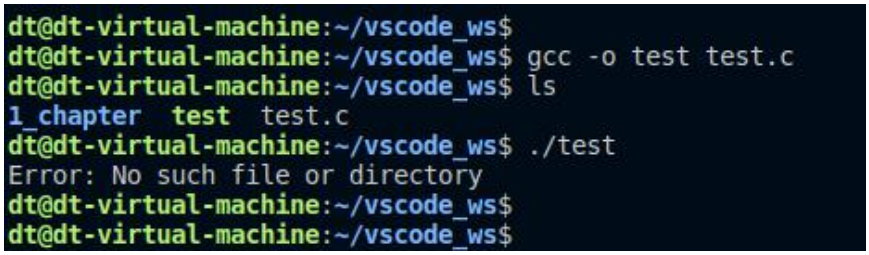

图 3.2.2 strerror 测试结果

从打印信息可以知道，strerror 返回的字符串是"No such file or directory"，所以从打印信息可知，我们就可以很直观的知道 open 函数执行的错误原因是文件不存在！

1.2.2 perror 函数

除了 strerror 函数之外，我们还可以使用 perror 函数来查看错误信息，一般用的最多的还是这个函数，调用此函数不需要传入 errno，函数内部会自己去获取 errno 变量的值，调用此函数会直接将错误提示字符串打印出来，而不是返回字符串，除此之外还可以在输出的错误提示字符串之前加入自己的打印信息，函数原型如下所示（可通过"man 3 perror"命令查看）：

```
#include <stdio.h>

void perror(const char *s);
```

需要包含<stdio.h>头文件。

函数参数和返回值含义如下：

s：在错误提示字符串信息之前，可加入自己的打印信息，也可不加，不加则传入空字符串即可。

返回值：void 无返回值。

测试

接下来我们进行测试，测试代码如下所示：

示例代码 3.2.2 perror 测试代码

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

int main(void)
{
    int fd;

    /* 打开文件 */
    fd = open("./test_file", O_RDONLY);
    if (-1 == fd) {
        perror("open error");
        return -1;
    }

    close(fd);
    return 0;
}
```

编译源代码，在 Ubuntu 系统下运行测试下，在当前目录下并不存在 test_file 文件，测试打印结果如下：

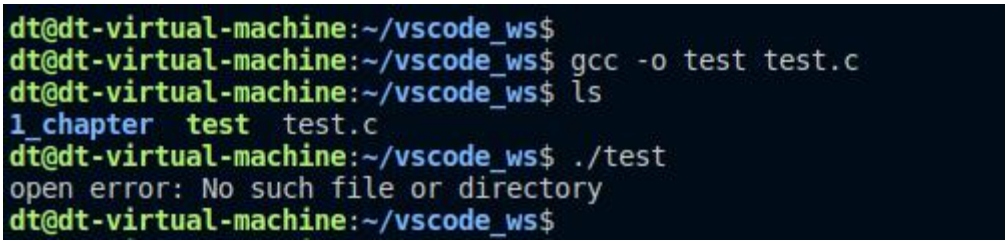

图 3.2.3 perror 测试结果

从打印信息可以知道，perror 函数打印出来的错误提示字符串是"No such file or directory"，跟 strerror 函数返回的字符串信息一样，"open error"便是我们附加的打印信息，而且从打印信息可知，perror 函数会在附加信息后面自动加入冒号和空格以区分。

以上给大家介绍了 strerror、perror 两个 C 库函数，都是用于查看函数执行错误时对应的提示信息，大家用哪个函数都可以，这里笔者推荐大家使用 perror，在实际的编程中这个函数用的还是比较多的，当然除了这两个之外，其它其它一些类似功能的函数，这里就不再给大家介绍了，意义不大！

1.3 exit、_exit、_Exit

当程序在执行某个函数出错的时候，如果此函数执行失败会导致后面的步骤不能在进行下去时，应该在出错时终止程序运行，不应该让程序继续运行下去，那么如何退出程序、终止程序运行呢？有过编程经验的读者都知道使用 return，一般原则程序执行正常退出 return 0，而执行函数出错退出 return -1，前面我们所编写的示例代码也是如此。

在 Linux 系统下，进程（程序）退出可以分为正常退出和异常退出，注意这里说的异常并不是执行函数出现了错误这种情况，异常往往更多的是一种不可预料的系统异常，可能是执行了某个函数时发生的、也有可能是收到了某种信号等，这里我们只讨论正常退出的情况。

在 Linux 系统下，进程正常退出除了可以使用 return 之外，还可以使用 exit()、_exit()以及_Exit()，下面我们分别介绍。

1.3.1 _exit()和_Exit()函数

main 函数中使用 return 后返回，return 执行后把控制权交给调用函数，结束该进程。调用_exit()函数会清除其使用的内存空间，并销毁其在内核中的各种数据结构，关闭进程的所有文件描述符，并结束进程、将控制权交给操作系统。_exit()函数原型如下所示：

```
#include <unistd.h>

void _exit(int status);
```

调用函数需要传入 status 状态标志，0 表示正常结束、若为其它值则表示程序执行过程中检测到有错误发生。使用示例如下：

示例代码 3.3.1 _exit()使用示例

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

int main(void)
{
    int fd;

    /* 打开文件 */
    fd = open("./test_file", O_RDONLY);
    if (-1 == fd) {
        perror("open error");
        _exit(-1);
    }

    close(fd);
    _exit(0);
}
```

用法很简单，大家可以自行测试！

_Exit()函数原型如下所示：

```
#include <stdlib.h>

void _Exit(int status);
```

_exit()和_Exit()两者等价，用法作用是一样的，这里就不再讲了，需要注意的是这 2 个函数都是系统调用。

1.3.2 exit()函数

exit()函数_exit()函数都是用来终止进程的，exit()是一个标准 C 库函数，而_exit()和_Exit()是系统调用。执行 exit()会执行一些清理工作，最后调用_exit()函数。exit()函数原型如下：

```
#include <stdlib.h>

void exit(int status);
```

该函数是一个标准 C 库函数，使用该函数需要包含头文件<stdlib.h>，该函数的用法和_exit()/_Exit()是一样的，这里就不再多说了。

本小节就给大家介绍了 3 中终止进程的方法：

- main 函数中运行 return；
- 调用 Linux 系统调用_exit()或_Exit()；
- 调用 C 标准库函数 exit()。

不管你用哪一种都可以结束进程，但还是推荐大家使用 exit()，其实关于 return、exit、_exit/_Exit()之间的区别笔者在上面只是给大家简单地描述了一下，甚至不太确定我的描述是否正确，因为笔者并不太多去关心其间的差异，对这些概念的描述会比较模糊、笼统，如果大家看不明白可以自己百度搜索相关的内容，当然对于初学者来说，不太建议大家去查找这些东西，至少对你现阶段来说，意义不是很大。好，本小节就介绍这么多，我们接着学习下一小节的内容。

1.4 空洞文件

1.4.1 概念

什么是空洞文件（hole file）？在上一章内容中，笔者给大家介绍了 lseek()系统调用，使用 lseek 可以修改文件的当前读写位置偏移量，此函数不但可以改变位置偏移量，并且还允许文件偏移量超出文件长度，这是什么意思呢？譬如有一个 test_file，该文件的大小是 4K（也就是 4096 个字节），如果通过 lseek 系统调用将该文件的读写偏移量移动到偏移文件头部 6000 个字节处，大家想一想会怎样？如果笔者没有提前告诉大家，大家觉得不能这样操作，但事实上 lseek 函数确实可以这样操作。

接下来使用 write()函数对文件进行写入操作，也就是说此时将是从偏移文件头部 6000 个字节处开始写入数据，也就意味着 4096~6000 字节之间出现了一个空洞，因为这部分空间并没有写入任何数据，所以形成了空洞，这部分区域就被称为文件空洞，那么相应的该文件也被称为空洞文件。

文件空洞部分实际上并不会占用任何物理空间，直到在某个时刻对空洞部分进行写入数据时才会为它分配对应的空间，但是空洞文件形成时，逻辑上该文件的大小是包含了空洞部分的大小的，这点需要注意。

那说了这么多，空洞文件有什么用呢？空洞文件对多线程共同操作文件是及其有用的，有时候我们创建一个很大的文件，如果单个线程从头开始依次构建该文件需要很长的时间，有一种思路就是将文件分为多段，然后使用多线程来操作，每个线程负责其中一段数据的写入；这个有点像我们现实生活当中施工队修路的感觉，比如说修建一条高速公路，单个施工队修筑会很慢，这个时候可以安排多个施工队，每一个施工队负责修建其中一段，最后将他们连接起来。

来看一下实际中空洞文件的两个应用场景：

- 在使用迅雷下载文件时，还未下载完成，就发现该文件已经占据了全部文件大小的空间，这也是空洞文件；下载时如果没有空洞文件，多线程下载时文件就只能从一

个地方写入，这就不能发挥多线程的作用了；如果有了空洞文件，可以从不同的地址同时写入，就达到了多线程的优势；

- 在创建虚拟机时，你给虚拟机分配了 100G 的磁盘空间，但其实系统安装完成之后，开始也不过只用了 3、4G 的磁盘空间，如果一开始就把 100G 分配出去，资源是很大的浪费。

关于空洞文件，这里就介绍这么多，上述描述当中多次提到了线程这个概念，关于线程这是后面的内容，这里先不给大家讲。

1.4.2 实验测试

这里我们进行相关的测试，新建一个文件把它做成空洞文件，示例代码如下所示：

示例代码 3.4.1 空洞文件测试代码

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

int main(void)
{
    int fd;
    int ret;
    char buffer[1024];
    int i;

    /* 打开文件 */
    fd = open("./hole_file", O_WRONLY | O_CREAT | O_EXCL,
              S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 将文件读写位置移动到偏移文件头 4096 个字节(4K)处 */
    ret = lseek(fd, 4096, SEEK_SET);
    if (-1 == ret) {
        perror("lseek error");
        goto err;
    }

    /* 初始化 buffer 为 0xFF */
    memset(buffer, 0xFF, sizeof(buffer));
```

```
    /* 循环写入 4 次，每次写入 1K */
    for (i = 0; i < 4; i++) {

        ret = write(fd, buffer, sizeof(buffer));
        if (-1 == ret) {
            perror("write error");
            goto err;
        }
    }

    ret = 0;
err:
    /* 关闭文件 */
    close(fd);
    exit(ret);
}
```

示例代码中，我们使用 open 函数新建了一个文件 hole_file，在 Linux 系统中，新建文件大小是 0，也就是没有任何数据写入，此时使用 lseek 函数将读写偏移量移动到 4K 字节处，再使用 write 函数写入数据 0xFF，每次写入 1K，一共写入 4 次，也就是写入了 4K 数据，也就意味着该文件前 4K 是文件空洞部分，而后 4K 数据才是真正写入的数据。

接下来进行编译测试，首先确保当前文件目录下不存在 hole_file 文件，测试结果如下：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
hole_file  testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -lh hole_file
-rw-r--r-- 1 dt dt 8.0K 1月    8 10:10 hole_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ du -h hole_file
4.0K    hole_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

示例代码 3.4.2 空洞文件测试结果

使用 ls 命令查看到空洞文件的大小是 8K，使用 ls 命令查看到的大小是文件的逻辑大小，自然是包括了空洞部分大小和真实数据部分大小；当使用 du 命令查看空洞文件时，其大小显示为 4K，du 命令查看到的大小是文件实际占用存储块的大小。

本小节内容就讲解完了，最后再向各位抛出一个问题：若使用 read 函数读取文件空洞部分，读取出来的将会是什么？关于这个问题大家可以先思考下，至于结果是什么，笔者这里便不给出答案了，大家可以自己动手编写代码进行测试以得出结论。

1.5 O_APPEND 和 O_TRUNC 标志

在上一章给大家讲解 open 函数的时候介绍了一些 open 函数的 flags 标志，譬如 O_RDONLY、O_WRONLY、O_CREAT、O_EXCL 等，本小节再给大家介绍两个标志，分别是 O_APPEND 和 O_TRUNC，接下来对这两个标志分别进行介绍。

1.5.1 O_TRUNC 标志

O_TRUNC 这个标志的作用非常简单，如果使用了这个标志，调用 open 函数打开文件的时候会将文件原本的内容全部丢弃，文件大小变为 0；这里我们直接测试即可！测试代码如下所示：

示例代码 3.5.1 O_TRUNC 标志测试

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

int main(void)
{
    int fd;

    /* 打开文件 */
    fd = open("./test_file", O_WRONLY | O_TRUNC);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 关闭文件 */
    close(fd);
    exit(0);
}
```

在当前目录下有一个文件 test_file，测试代码中使用了 O_TRUNC 标志打开该文件，代码中仅仅只是打开该文件，之后调用 close 关闭了文件，并没有对其进行读写操作，接下来编译运行来看看测试结果：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 16
-rw-rw-r-- 1 dt dt  318 1月   8 16:06 testApp.c
-rw-rw-r-- 1 dt dt 8760 1月   8 16:06 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 16
-rwxrwxr-x 1 dt dt 8760 1月   8 16:07 testApp
-rw-rw-r-- 1 dt dt  318 1月   8 16:06 testApp.c
-rw-rw-r-- 1 dt dt    0 1月   8 16:07 test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.5.1 O_TRUNC 测试结果

在测试之前 test_file 文件中是有数据的，文件大小为 8760 个字节，执行完测试程序后，再使用 ls 命令查看文件大小时发现 test_file 大小已经变成了 0，也就是说明文件之前的内容已经全部被丢弃了。这就是 O_TRUNC 标志的作用了，大家可以自己动手试试。

1.5.2 O_APPEND 标志

接下里聊一聊 O_APPEND 标志，如果 open 函数携带了 O_APPEND 标志，调用 open 函数打开文件，当每次使用 write()函数对文件进行写操作时，都会自动把文件当前位置偏移量移动到文件末尾，从文件末尾开始写入数据，也就是意味着每次写入数据都是从文件末尾开始。这里我们直接进行测试，测试代码如下所示：

示例代码 3.5.2 O_APPEND 标志测试

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

int main(void)
{
    char buffer[16];
    int fd;
    int ret;

    /* 打开文件 */
    fd = open("./test_file", O_RDWR | O_APPEND);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 初始化 buffer 中的数据 */
    memset(buffer, 0x55, sizeof(buffer));

    /* 写入数据: 写入 4 个字节数据 */
    ret = write(fd, buffer, 4);
    if (-1 == ret) {
        perror("write error");
        goto err;
    }

    /* 将 buffer 缓冲区中的数据全部清 0 */
    memset(buffer, 0x00, sizeof(buffer));
```

```
    /* 将位置偏移量移动到距离文件末尾 4 个字节处 */
    ret = lseek(fd, -4, SEEK_END);
    if (-1 == ret) {
        perror("lseek error");
        goto err;
    }

    /* 读取数据 */
    ret = read(fd, buffer, 4);
    if (-1 == ret) {
        perror("read error");
        goto err;
    }

    printf("0x%x 0x%x 0x%x 0x%x\n", buffer[0], buffer[1],
                buffer[2], buffer[3]);
    ret = 0;
err:
    /* 关闭文件 */
    close(fd);
    exit(ret);
}
```

测试代码中会去打开当前目录下的 test_file 文件，使用可读可写方式，并且使用了 O_APPEND 标志，前面笔者给大家提到过，open 打开一个文件，默认的读写位置偏移量会处于文件头，但测试代码中使用了 O_APPEND 标志，如果 O_APPEND 确实能生效的话，也就意味着调用 write 函数会从文件末尾开始写；代码中写入了 4 个字节数据，都是 0x55，之后,使用 lseek 函数将位置偏移量移动到距离文件末尾 4 个字节处，读取 4 个字节（也就是读取文件最后 4 个字节数据），之后将其打印出来，如果上面笔者的描述正确的话，打印出来的数据就是我们写入的数据，如果 O_APPEND 不能生效，则打印出来数据就不会是 0x55，接下来编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
0x55 0x55 0x55 0x55
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.5.2 O_APPEND 标志测试结果

从上面打印信息可知，读取出来的数据确实等于 0x55，说明 O_APPEND 标志确实有作用，当调用 write()函数写文件时，会自动把文件当前位置偏移量移动到文件末尾。

当然，本小节内容还并没有结束，这其中还涉及到一些细节问题需要大家注意，首先第一点，O_APPEND 标志并不会影响读文件，当读取文件时，O_APPEND 标志并不会影响读位置偏移量，即使使用了 O_APPEND 标志，读文件位置偏移量默认情况下依然是文件头，

关于这个问题大家可以自己进行测试，编程是一个实践性很强的工作，有什么不能理解的问题，可以自己编写程序进行测试。

大家可能会想到使用 lseek 函数来改变 write()时的写位置偏移量，其实这种做法并不会成功，这就是笔者给大家提的第二个细节，使用了 O_APPEND 标志，即使是通过 lseek 函数也是无法修改写文件时对应的位置偏移量（注意笔者这里说的是写文件，并不包括读），写入数据依然是从文件末尾开始，lseek 并不会该变写位置偏移量，这个问题测试方法很简单，也就是在 write 之前使用 lseek 修改位置偏移量，这里笔者就不再给大家测试了，我还是那句话，编程是一个实践性很强的工作，大家只需要把示例代码 3.5.2 进行简单地修改即可！

其实关于第二点细节原因很简单，当执行 write()函数时，检测到 open 函数携带了 O_APPEND 标志，所以在 write 函数内部会自动将写位置偏移量移动到文件末尾，当然这里也只是笔者的一个简单地猜测，至于是不是这样，笔者也无从考证。

到这里本小节的内容就暂时介绍完了，为什么说是“暂时”？因为后面的内容中还会聊到 O_APPEND 标志，最后笔者再给大家出一个小问题，大家可以自己动手测试。

- 当 open 函数同时携带了 O_APPEND 和 O_TRUNC 两个标志时会有什么作用？

1.6 多次打开同一个文件

大家看到这个小节标题可能会有疑问，同一个文件还能被多次打开？事实确实如此，同一个文件可以被多次打开，譬如在一个进程中多次打开同一个文件、在多个不同的进程中打开同一个文件，那么这些操作都是被允许的。本小节就来探讨下多次打开同一个文件会有一些什么现象以及相应的细节问题？

1.6.1 验证一些现象

- **一个进程内多次 open 打开同一个文件，那么会得到多个不同的文件描述符 fd，同理在关闭文件的时候也需要调用 close 依次关闭各个文件描述符。**

针对这个问题，我们编写测试代码进行测试，如下所示：

示例代码 3.6.1 多次打开同一个文件测试代码 1

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

int main(void)
{
    int fd1, fd2, fd3;
    int ret;

    /* 第一次打开文件 */
    fd1 = open("./test_file", O_RDWR);
    if (-1 == fd1) {
        perror("open error");
```

```
        exit(-1);
    }

    /* 第二次打开文件 */
    fd2 = open("./test_file", O_RDWR);
    if (-1 == fd2) {
        perror("open error");
        ret = -1;
        goto err1;
    }

    /* 第三次打开文件 */
    fd3 = open("./test_file", O_RDWR);
    if (-1 == fd3) {
        perror("open error");
        ret = -1;
        goto err2;
    }

    /* 打印出 3 个文件描述符 */
    printf("%d  %d  %d\n", fd1, fd2, fd3);

    close(fd3);
    ret = 0;
err2:
    close(fd2);

err1:
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}
```

上述示例代码中，通过 3 次调用 open 函数对 test_file 文件打开了 3 次，每一个调用传参一样，最后将 3 次得到的文件描述符打印出来，在当前目录下存在 test_file 文件，接下来编译测试，看看结果如何：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
6  7  8
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.6.1 打印文件描述符

从打印结果可知，三次调用 open 函数得到的文件描述符分别为 6、7、8，通过任何一个文件描述符对文件进行 IO 操作都是可以的，但是需要注意是，调用 open 函数打开文件使用的是什么权限，则返回的文件描述符就拥有什么权限，文件 IO 操作完成之后，在结束进程之前需要使用 close 关闭各个文件描述符。

在图 3.6.1 中，细心的读者可能会发现，调用 open 函数得到的最小文件描述符是 6，在上一章节内容中给大家提到过，程序中分配得到的最小文件描述符一般是 3，但这里竟然是 6！这是为何？其实这个问题跟 vscode 有关，说明 3、4、5 这 3 个文件描述符已经被 vscode 软件对应的进程所占用了，而当前这里执行 testApp 文件是在 vscode 软件提供的终端下进行的，所以 vscode 可以认为是 testApp 进程的父进程，相反，testApp 进程便是 vscode 进程的子进程，子进程会继承父进程的文件描述符。关于子进程和父进程这些都是后面的内容，这里暂时不给大家进行介绍，这是只是给大家简单地解释一下，免得大家误会！

其实可以直接在 Ubuntu 系统的 Terminal 终端执行 testApp，这时你会发现打印出来的文件描述符分别是 3、4、5，这里就不给大家演示了。

- **一个进程内多次 open 打开同一个文件，在内存中并不会存在多份动态文件。**

当调用 open 函数的时候，会将文件数据（文件内容）从磁盘等块设备读取到内存中，将文件数据在内存中进行维护，内存中的这份文件数据我们就把它称为动态文件！这是前面给大家介绍的内容，这里再简单地提一下。这里出现了一个问题：如果同一个文件被多次打开，那么该文件所对应的动态文件是否在内存中也存在多份？也就是说，多次打开同一个文件是否会将其文件数据多次拷贝到内存中进行维护？

关于这个问题，各位读者可以简单地思考一下，这里我们直接编写代码进行测试，测试代码如下所示：

示例代码 3.6.2 多次打开同一个文件测试代码 2

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

int main(void)
{
    char buffer[4];
    int fd1, fd2;
    int ret;

    /* 创建新文件 test_file 并打开 */
    fd1 = open("./test_file", O_RDWR | O_CREAT | O_EXCL,
                    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (-1 == fd1) {
        perror("open error");
        exit(-1);
    }
```

```
    /* 再次打开 test_file 文件 */
    fd2 = open("./test_file", O_RDWR);
    if (-1 == fd2) {
        perror("open error");
        ret = -1;
        goto err1;
    }

    /* 通过 fd1 文件描述符写入 4 个字节数据 */
    buffer[0] = 0x11;
    buffer[1] = 0x22;
    buffer[2] = 0x33;
    buffer[3] = 0x44;

    ret = write(fd1, buffer, 4);
    if (-1 == ret) {
        perror("write error");
        goto err2;
    }

    /* 将读写位置偏移量移动到文件头 */
    ret = lseek(fd2, 0, SEEK_SET);
    if (-1 == ret) {
        perror("lseek error");
        goto err2;
    }

    /* 读取数据 */
    memset(buffer, 0x00, sizeof(buffer));
    ret = read(fd2, buffer, 4);
    if (-1 == ret) {
        perror("read error");
        goto err2;
    }

    printf("0x%x 0x%x 0x%x 0x%x\n", buffer[0], buffer[1],
                    buffer[2], buffer[3]);

    ret = 0;
err2:
    close(fd2);

err1:
```

```
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}
```

当前目录下不存在 test_file 文件，上述代码中，第一次调用 open 函数新建并打开 test_file 文件，第二次调用 open 函数再次打开它，新建文件时，文件大小为 0；首先通过文件描述符 fd1 写入 4 个字节数据（0x11/0x22/0x33/0x44），从文件头开始写；然后再通过文件描述符 fd2 读取 4 个字节数据，也是从文件头开始读取。假如，内存中只有一份动态文件，那么读取得到的数据应该就是 0x11、0x22、0x33、0x44，如果存在多份动态文件，那么通过 fd2 读取的是与它对应的动态文件中的数据，那就不是 0x11、0x22、0x33、0x44，而是读取出 0 个字节数据，因为它的文件大小是 0。

接下来进行编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
0x11 0x22 0x33 0x44
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.6.2 测试结果 2

上图中打印显示读取出来的数据是 0x11/0x22/0x33/0x44，所以由此可知，即使多次打开同一个文件，内存中也只有一份动态文件。

- **一个进程内多次 open 打开同一个文件，不同文件描述符所对应的读写位置偏移量是相互独立的。**

同一个文件被多次打开，会得到多个不同的文件描述符，也就意味着会有多个不同的文件表，而文件读写偏移量信息就记录在文件表数据结构中，所以从这里可以推测不同的文件描述符所对应的读写偏移量是相互独立的，并没有关联在一起，并且文件表中 i-node 指针指向的都是同一个 inode，如下图所示：

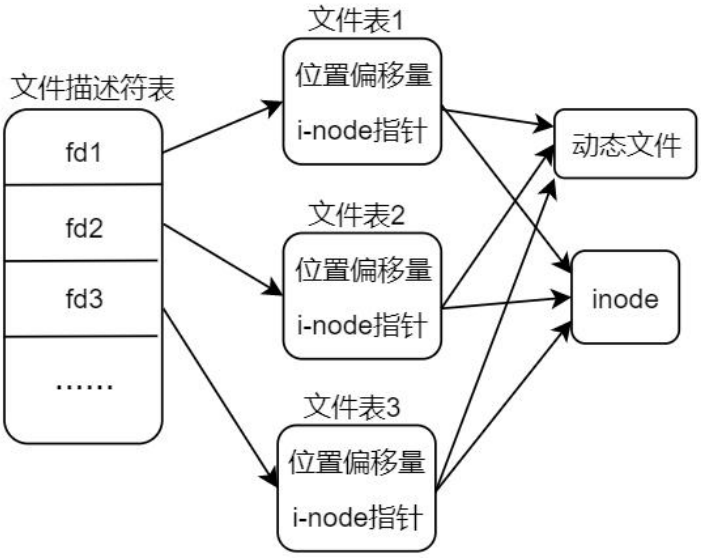

图 3.6.3 多次打开同一个文件--文件描述符表、文件表以及 inode 之间的关系

测试的方法很简单，只需在示例代码 3.6.2 中简单地修改即可，将 lseek 函数调用去掉，然后在编译测试，如果读出的数据依然是 0x11/0x22/0x33/0x44，则表示第三点结论成立，这里不再给大家演示。

Tips：多个不同的进程中调用 open()打开磁盘中的同一个文件，同样在内存中也只是维护了一份动态文件，多个进程间共享，它们有各自独立的文件读写位置偏移量。

动态文件何时被关闭呢？当文件的引用计数为 0 时，系统会自动将其关闭，同一个文件被打开多次，文件表中会记录该文件的引用计数，如图 3.1.5 所示，引用计数记录了当前文件被多少个文件描述符 fd 关联。

1.6.2 多次打开同一文件进行读操作与 O_APPEND 标志

重复打开同一个文件，进行写操作，譬如一个进程中两次调用 open 函数打开同一个文件，分别得到两个文件描述符 fd1 和 fd2，使用这两个文件描述符对文件进行写入操作，那么它们是分别写（各从各的位置偏移量开始写）还是接续写（一个写完，另一个接着后面写）？其实这个问题，3.6.1 小节中已经给出了答案，因为这两个文件描述符所对应的读写位置偏移量是相互独立的，所以是分别写，接下来我们还是编写代码进行测试，测试代码如下所示：

示例代码 3.6.3 多次打开同一个文件测试代码 3

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

int main(void)
{
    unsigned char buffer1[4], buffer2[4];
    int fd1, fd2;
    int ret;
```

```
int i;

/* 创建新文件 test_file 并打开 */
fd1 = open("./test_file", O_RDWR | O_CREAT | O_EXCL,
                S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
if (-1 == fd1) {
    perror("open error");
    exit(-1);
}

/* 再次打开 test_file 文件 */
fd2 = open("./test_file", O_RDWR);
if (-1 == fd2) {
    perror("open error");
    ret = -1;
    goto err1;
}

/* buffer 数据初始化 */
buffer1[0] = 0x11;
buffer1[1] = 0x22;
buffer1[2] = 0x33;
buffer1[3] = 0x44;

buffer2[0] = 0xAA;
buffer2[1] = 0xBB;
buffer2[2] = 0xCC;
buffer2[3] = 0xDD;

/* 循环写入数据 */
for (i = 0; i < 4; i++) {

    ret = write(fd1, buffer1, sizeof(buffer1));
    if (-1 == ret) {
        perror("write error");
        goto err2;
    }

    ret = write(fd2, buffer2, sizeof(buffer2));
    if (-1 == ret) {
        perror("write error");
        goto err2;
    }
}
```

```
    /* 将读写位置偏移量移动到文件头 */
    ret = lseek(fd1, 0, SEEK_SET);
    if (-1 == ret) {
        perror("lseek error");
        goto err2;
    }

    /* 读取数据 */
    for (i = 0; i < 8; i++) {

        ret = read(fd1, buffer1, sizeof(buffer1));
        if (-1 == ret) {
            perror("read error");
            goto err2;
        }

        printf("%x%x%x%x", buffer1[0], buffer1[1],
                buffer1[2], buffer1[3]);
    }

    printf("\n");
    ret = 0;
err2:
    close(fd2);

err1:
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}
```

示例代码 3.6.3 中，重复两次打开 test_file 文件，分别得到两个文件描述符 fd1、fd2；首先通过 fd1 写入 4 个字节数据（0x11、0x22、0x33、0x44）到文件中，接着再通过 fd2 写入 4 个字节数据（0xaa、0xbb、0xcc、0xdd）到文件中，循环写入 4 此；最后再将写入的数据读取出来，将其打印到终端。如果它们是分别写，那么读取出来的数据就应该是 aabbccdd……，因为通过 fd1 写入的数据被 fd2 写入的数据给覆盖了；如果它们是接续写，那么读取出来的数据应该是 11223344aabbccdd……，接下里我们编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
aabbccddaabbccddaabbccddaabbccddaabbccddaabbccddaabbccddaabbccdd
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.6.4 测试结果 3

从打印结果可知，它们确实是分别写。如果想要实现接续写，也就是当通过 fd1 写入完成之后，通过 fd2 写入的数据是接在 fd1 写入的数据之后，那么该怎么做呢？当然可以写入数据之前通过 lseek 函数将文件偏移量移动到文件末尾，如果是这样做，会存在一些问题，关于这个问题后面再给大家介绍；这里我们给大家介绍使用 O_APPEND 标志来解决这个问题，也就是将分别写更改为接续写。

前面给大家介绍了 open 函数的 O_APPEND 标志，当 open 函数使用 O_APPEND 标志，在使用 write 函数进行写入操作时，会自动将偏移量移动到文件末尾，也就是每次写入都是从文件末尾开始；这里结合本小节的内容，我们再来讨论 O_APPEND 标志，在多次打开同一个文件进行写操作时，使用 O_APPEND 标志会有什么样的效果，接下来进行测试：

示例代码 3.6.4 多次打开同一个文件测试代码 4

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

int main(void)
{
    unsigned char buffer1[4], buffer2[4];
    int fd1, fd2;
    int ret;
    int i;

    /* 创建新文件 test_file 并打开 */
    fd1 = open("./test_file", O_RDWR | O_CREAT | O_EXCL | O_APPEND,
                    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (-1 == fd1) {
        perror("open error");
        exit(-1);
    }

    /* 再次打开 test_file 文件 */
    fd2 = open("./test_file", O_RDWR | O_APPEND);
    if (-1 == fd2) {
        perror("open error");
        ret = -1;
        goto err1;
    }

    /* buffer 数据初始化 */
    buffer1[0] = 0x11;
    buffer1[1] = 0x22;
```

```
buffer1[2] = 0x33;
buffer1[3] = 0x44;

buffer2[0] = 0xAA;
buffer2[1] = 0xBB;
buffer2[2] = 0xCC;
buffer2[3] = 0xDD;

/* 循环写入数据 */
for (i = 0; i < 4; i++) {

        ret = write(fd1, buffer1, sizeof(buffer1));
        if (-1 == ret) {
                perror("write error");
                goto err2;
        }

        ret = write(fd2, buffer2, sizeof(buffer2));
        if (-1 == ret) {
                perror("write error");
                goto err2;
        }
}

/* 将读写位置偏移量移动到文件头 */
ret = lseek(fd1, 0, SEEK_SET);
if (-1 == ret) {
        perror("lseek error");
        goto err2;
}

/* 读取数据 */
for (i = 0; i < 8; i++) {

        ret = read(fd1, buffer1, sizeof(buffer1));
        if (-1 == ret) {
                perror("read error");
                goto err2;
        }

        printf("%x%x%x%x", buffer1[0], buffer1[1],
                        buffer1[2], buffer1[3]);
}
```

```
    printf("\n");
    ret = 0;
err2:
    close(fd2);

err1:
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}
```

示例代码 3.6.4 仅仅只是在示例代码 3.6.3 的基础上，open 函数添加了 O_APPEND 标志，其它内容并没有动过，接下来编译测试。

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
11223344aabbccdd11223344aabbccdd11223344aabbccdd11223344aabbccdd
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.6.5 测试结果 4

从打印出来的数据可知，加入了 O_APPEND 标志后，分别写已经变成了接续写。关于 O_APPEND 标志还涉及到一个原子操作的问题，后面再给大家介绍，本小节内容到此！

1.7 复制文件描述符

在 Linux 系统中，open 返回得到的文件描述符 fd 可以进行复制，复制成功之后可以得到一个新的文件描述符，使用新的文件描述符和旧的文件描述符都可以对文件进行 IO 操作，复制得到的文件描述符和旧的文件描述符拥有相同的权限，譬如使用旧的文件描述符对文件有读写权限，那么新的文件描述符同样也具有读写权限；在 Linux 系统下，可以使用 dup 或 dup2 这两个系统调用对文件描述符进行复制，本小节就给大家介绍这两个函数的用法以及它们之间的区别。

复制得到的文件描述符与旧的文件描述符都指向了同一个文件表，假设 fd1 为原文件描述符，fd2 为复制得到的文件描述符，如下图所示：

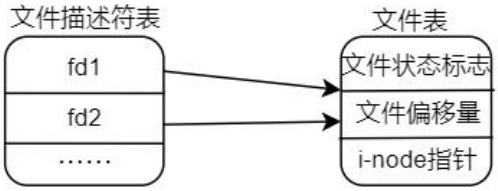

图 3.7.1 指向同一个文件表

因为复制得到的文件描述符与旧的文件描述符指向的是同一个文件表，所以可知，这两个文件描述符的属性是一样，譬如对文件的读写权限、文件状态标志、文件偏移量等，所以从这里也可知道“复制”的含义实则是复制文件表。同样，在使用完毕之后也需要使用 close 来关闭文件描述符。

1.7.1 dup 函数

dup 函数用于复制文件描述符，此函数原型如下所示（可通过"man 2 dup"命令查看）：

```
#include <unistd.h>

int dup(int oldfd);
```

首先使用此函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下：

oldfd：需要被复制的文件描述符。

返回值：成功时将返回一个新的文件描述符，由操作系统分配，分配置原则遵循文件描述符分配原则；如果复制失败将返回-1，并且会设置 errno 值。

测试

由前面的介绍可知，复制得到的文件描述符与原文件描述符都指向同一个文件表，所以它们的文件读写偏移量是一样的，那么是不是可以在不使用 O_APPEND 标志的情况下，通过文件描述符复制来实现接续写，接下来我们编写一个程序进行测试，测试代码如下所示：

示例代码 3.7.1 dup 函数测试代码

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

int main(void)
{
    unsigned char buffer1[4], buffer2[4];
    int fd1, fd2;
    int ret;
    int i;

    /* 创建新文件 test_file 并打开 */
    fd1 = open("./test_file", O_RDWR | O_CREAT | O_EXCL,
                    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (-1 == fd1) {
        perror("open error");
        exit(-1);
    }

    /* 复制文件描述符 */
    fd2 = dup(fd1);
    if (-1 == fd2) {
        perror("dup error");
        ret = -1;
        goto err1;
    }
```

```
printf("fd1: %d\nfd2: %d\n", fd1, fd2);

/* buffer 数据初始化 */
buffer1[0] = 0x11;
buffer1[1] = 0x22;
buffer1[2] = 0x33;
buffer1[3] = 0x44;

buffer2[0] = 0xAA;
buffer2[1] = 0xBB;
buffer2[2] = 0xCC;
buffer2[3] = 0xDD;

/* 循环写入数据 */
for (i = 0; i < 4; i++) {

        ret = write(fd1, buffer1, sizeof(buffer1));
        if (-1 == ret) {
                perror("write error");
                goto err2;
        }

        ret = write(fd2, buffer2, sizeof(buffer2));
        if (-1 == ret) {
                perror("write error");
                goto err2;
        }
}

/* 将读写位置偏移量移动到文件头 */
ret = lseek(fd1, 0, SEEK_SET);
if (-1 == ret) {
        perror("lseek error");
        goto err2;
}

/* 读取数据 */
for (i = 0; i < 8; i++) {

        ret = read(fd1, buffer1, sizeof(buffer1));
        if (-1 == ret) {
                perror("read error");
                goto err2;
```

```
        }

        printf("%x%x%x%x", buffer1[0], buffer1[1],
                buffer1[2], buffer1[3]);
    }

    printf("\n");
    ret = 0;
err2:
    close(fd2);

err1:
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}
```

测试代码中，我们使用了 dup 系统调用复制了文件描述符 fd1，得到另一个新的文件描述符 fd2，分别通过 fd1 和 fd2 对文件进行写操作，最后读取写入的数据来判断是分别写还是接续写，接下来编译测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
fd1: 6
fd2: 7
11223344aabbccdd11223344aabbccdd11223344aabbccdd11223344aabbccdd
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.7.2 dup 测试结果

由打印信息可知，fd1 等于 6，复制得到的新的文件描述符为 7（遵循 fd 分配原则），打印出来的数据显示为接续写，所以可知，通过复制文件描述符可以实现接续写。

1.7.2 dup2 函数

dup 系统调用分配的文件描述符是由系统分配的，遵循文件描述符分配原则，并不能自己指定一个文件描述符，这是 dup 系统调用的一个缺陷；而 dup2 系统调用修复了这个缺陷，可以手动指定文件描述符，而不需要遵循文件描述符分配原则，当然在实际的编程工作中，需要根据自己的情况来进行选择。

dup2 函数原型如下所示（可以通过"man 2 dup2"命令查看）：

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

同样使用该命令也需要包含<unistd.h>头文件。

函数参数和返回值含义如下：

oldfd：需要被复制的文件描述符。

newfd：指定一个文件描述符（需要指定一个当前进程没有使用到的文件描述符）。

返回值：成功时将返回一个新的文件描述符，也就是手动指定的文件描述符 newfd；如果复制失败将返回-1，并且会设置 errno 值。

测试

接下来编写一个简单地测试程序，如下所示：

示例代码 3.7.2 dup2 函数测试代码

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

int main(void)
{
    int fd1, fd2;
    int ret;

    /* 创建新文件 test_file 并打开 */
    fd1 = open("./test_file", O_RDWR | O_CREAT | O_EXCL,
                S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (-1 == fd1) {
        perror("open error");
        exit(-1);
    }

    /* 复制文件描述符 */
    fd2 = dup2(fd1, 100);
    if (-1 == fd2) {
        perror("dup error");
        ret = -1;
        goto err1;
    }

    printf("fd1: %d\nfd2: %d\n", fd1, fd2);
    ret = 0;

    close(fd2);
err1:
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}
```

测试代码使用 dup2 函数复制文件描述符 fd1，指定新的文件描述符为 100，复制成功之后将其打印出来，结果如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
fd1: 6
fd2: 100
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.7.3 dup2 函数测试结果

由打印信息可知，复制得到的文件描述符 fd2 等于 100，正是我们在 dup2 函数中指定的文件描述符。本小节的内容到这里结束了，最后再强调一点，文件描述符并不是只能复制一次，实际上可以对同一个文件描述符 fd 调用 dup 或 dup2 函数复制多次，得到多个不同的文件描述符。

1.8 文件共享

什么是文件共享？所谓文件共享指的是同一个文件（譬如磁盘上的同一个文件，对应同一个 inode）被多个独立的读写体同时进行 IO 操作。多个独立的读写体大家可以将其简单地理解为对应于同一个文件的多个不同的文件描述符，譬如多次打开同一个文件所得到的多个不同的 fd，或使用 dup()（或 dup2）函数复制得到的多个不同的 fd 等。

同时进行 IO 操作指的是一个读写体操作文件尚未调用 close 关闭的情况下，另一个读写体去操作文件，前面给大家编写的示例代码中就已经涉及到了文件共享的内容了，譬如 3.6 小节中编写的示例代码中，同一个文件对应两个不同的文件描述符 fd1 和 fd2，当使用 fd1 对文件进行写操作之后，并没有关闭 fd1，而此时使用 fd2 对文件再进行写操作，这其实就是一种文件共享。

文件共享的意义有很多，多用于多进程或多线程编程环境中，譬如我们可以通过文件共享的方式来实现多个线程同时操作同一个大文件，以减少文件读写时间、提升效率。

文件共享的核心是：如何制造出多个不同的文件描述符来指向同一个文件。其实方法在上面的内容中都已经给大家介绍过了，譬如多次调用 open 函数重复打开同一个文件得到多个不同的文件描述符、使用 dup()或 dup2()函数对文件描述符进行复制以得到多个不同的文件描述符。

常见的三种文件共享的实现方式

(1)同一个进程中多次调用 open 函数打开同一个文件，各数据结构之间的关系如下图所示：

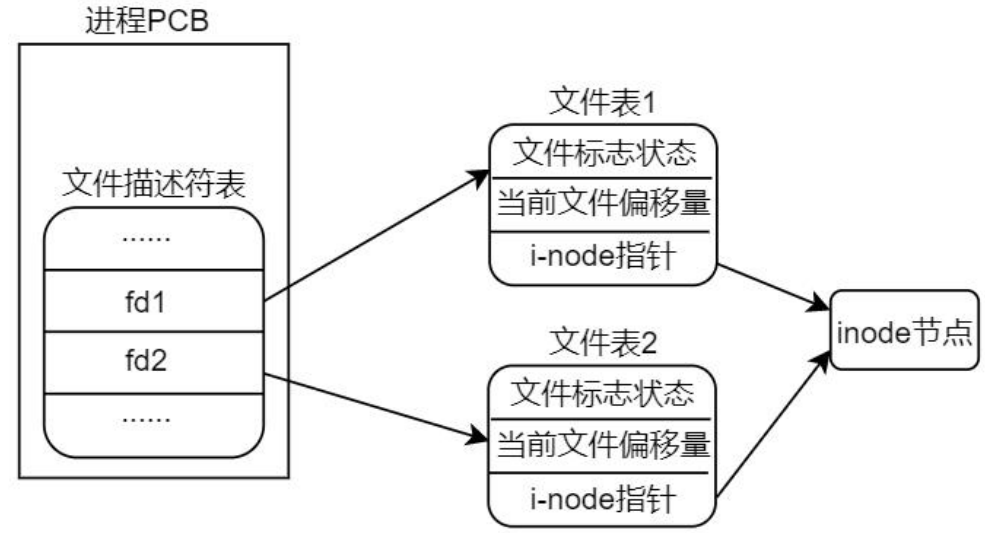

图 3.8.1 同一进程多次 open 打开同一文件各数据结构关系图

这种情况非常简单，多次调用 open 函数打开同一个文件会得到多个不同的文件描述符，并且多个文件描述符对应多个不同的文件表，所有的文件表都索引到了同一个 inode 节点，也就是磁盘上的同一个文件。

(2)不同进程中分别使用 open 函数打开同一个文件，其数据结构关系图如下所示：

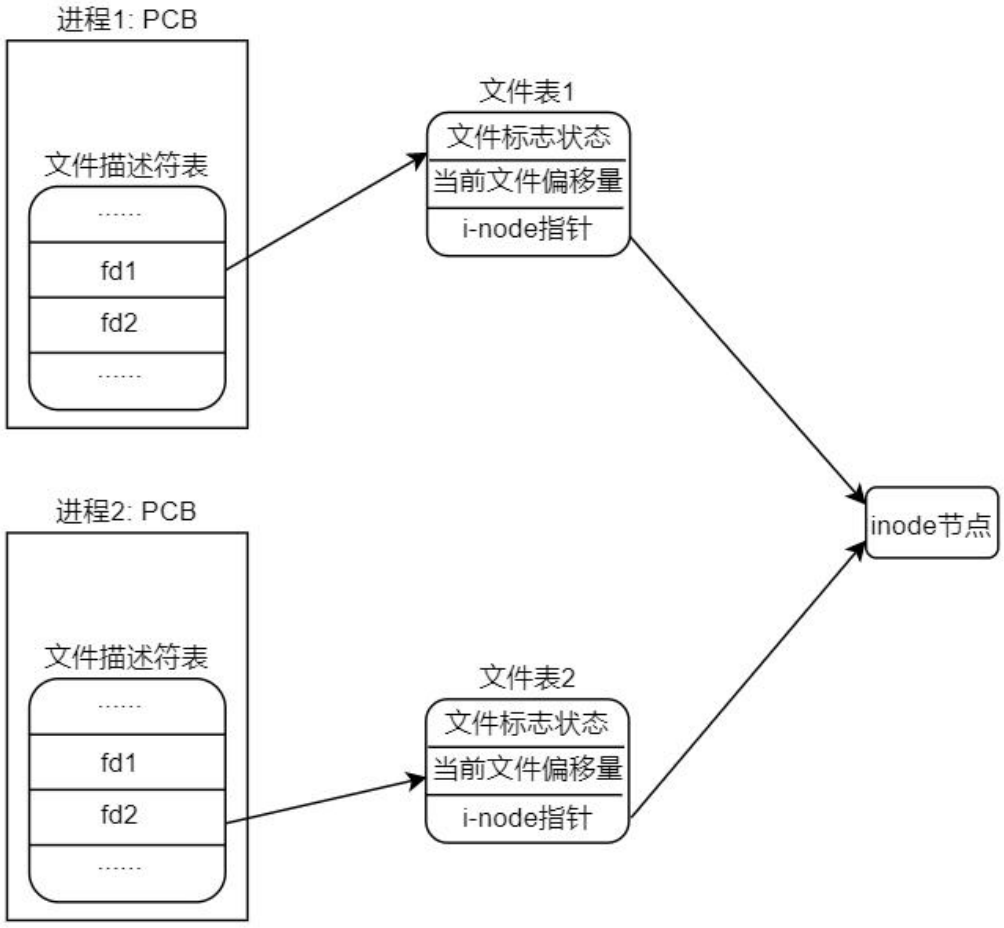

图 3.8.2 不同进程 open 打开同一文件数据结构关系图

进程 1 和进程 2 分别是运行在 Linux 系统上两个独立的进程（理解为两个独立的程序），在他们各自的程序中分别调用 open 函数打开同一个文件，进程 1 对应的文件描述符为 fd1，

进程 2 对应的文件描述符为 fd2，fd1 指向了进程 1 的文件表 1，fd2 指向了进程 2 的文件表 2；各自的文件表都索引到了同一个 inode 节点，从而实现共享文件。

(3)同一个进程中通过 dup（dup2）函数对文件描述符进行复制，其数据结构关系如下图所示：

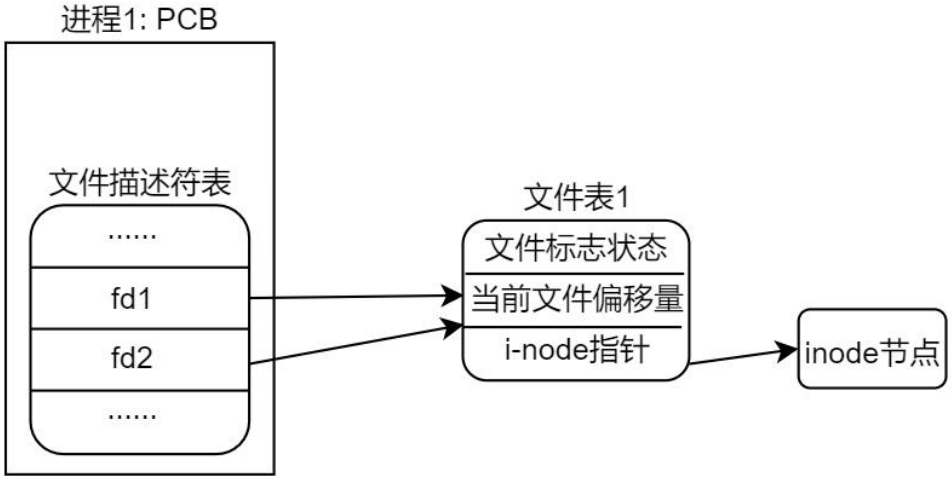

图 3.8.3 dup 复制文件描述符实现文件共享

这种方式上一小节已经给大家进行了详细讲解，这里不再重述！

对于文件共享，存在着竞争冒险，这个是需要大家关注的，下一小节将会向大家介绍。除此之外，我们还需要关心的是文件共享时，不同的读写体之间是分别写还是接续写，这些细节问题大家都要搞清楚。

1.9 原子操作与竞争冒险

Linux 是一个多任务、多进程操作系统，系统中往往运行着多个不同的进程、任务，多个不同的进程就有可能对同一个文件进行 IO 操作，此时该文件便是它们的共享资源，它们共同操作着同一份文件；操作系统级编程不同于大家以前接触的裸机编程，裸机程序中不存在进程、多任务这种概念，而在 Linux 系统中，我们必须要留意到多进程环境下可能会导致的竞争冒险。

1.9.1 竞争冒险简介

本小节给大家竞争冒险这个概念，如果学习过 Linux 驱动开发的读者对这些概念应该并不陌生，也就意味着竞争冒险不但存在于 Linux 应用层、也存在于 Linux 内核驱动层。

假设有两个独立的进程 A 和进程 B 都对同一个文件进行追加写操作（也就是在文件末尾写入数据），每一个进程都调用了 open 函数打开了该文件，但未使用 O_APPEND 标志，此时，各数据结构之间的关系如图 3.8.2 所示。每个进程都有它自己的进程控制块 PCB，有自己的文件表（意味着有自己独立的读写位置偏移量），但是共享同一个 inode 节点（也就是对应同一个文件）。假定此时进程 A 处于运行状态，B 未处于等待运行状态，进程 A 调用了 lseek 函数，它将进程 A 的该文件当前位置偏移量设置为 1500 字节处（假设这里是文件末尾），刚好此时进程 A 的时间片耗尽，然后内核切换到了进程 B，进程 B 执行 lseek 函数，也将其对该文件的当前位置偏移量设置为 1500 个字节处（文件末尾）。然后进程 B 调用 write 函数，写入了 100 个字节数据，那么此时在进程 B 中，该文件的当前位置偏移量已经移动到了 1600 字节处。B 进程时间片耗尽，内核又切换到了进程 A，使进程 A 恢复运行，当进程 A 调用 write 函数时，是从进程 A 的该文件当前位置偏移量（1500 字节处）开始写

入，此时文件 1500 字节处已经不再是文件末尾了，如果还从 1500 字节处写入就会覆盖进程 B 刚才写入到该文件中的数据。

其上述假设工作流程图如下图所示：

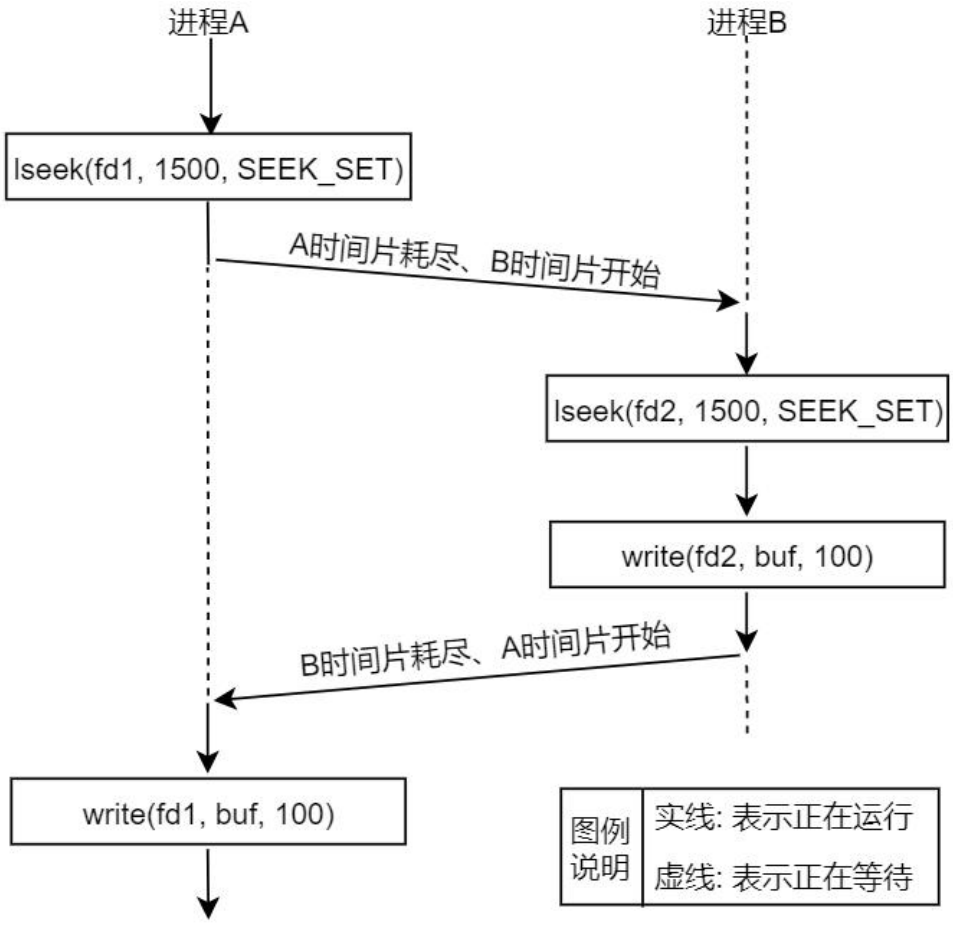

图 3.9.1 假设中的 AB 进程工作流程

以上给大家所描述的这样一种情形就属于竞争状态（也成为竞争冒险），操作共享资源的两个进程（或线程），其操作之后的所得到的结果往往是不可预期的，因为每个进程（或线程）去操作文件的顺序是不可预期的，即这些进程获得 CPU 使用权的先后顺序是不可预期的，完全由操作系统调配，这就是所谓的竞争状态。

既然存在竞争状态，那么该如何规避或消除这种状态呢？接下来给大家介绍原子操作。

1.9.2 原子操作

在上一章给大家介绍 open 函数的时候就提到过“原子操作”这个概念了，同样在 Linux 驱动编程中，也有这个概念，相信学习过 Linux 驱动编程开发的读者应该有印象。

从上一小节给大家提到的示例中可知，上述的问题出在逻辑操作“先定位到文件末尾，然后再写”，它使用了两个分开的函数调用，首先使用 lseek 函数将文件当前位置偏移量移动到文件末尾、然后在使用 write 函数将数据写入到文件。既然知道了问题所在，那么解决办法就是将这两个操作步骤合并成一个原子操作，所谓原子操作，是有多步操作组成的一个操作，原子操作要么一步也不执行，一旦执行，必须要执行完所有步骤，不可能只执行所有步骤中的一个子集。

(1)O_APPEND 实现原子操作

在上一小节给大家提到的示例中，进程 A 和进程 B 都对同一个文件进行追加写操作，导致进程 A 写入的数据覆盖了进程 B 写入的数据，解决办法就是将“先定位到文件末尾，然后写”这两个步骤组成一个原子操作即可，那如何使其变成一个原子操作呢？答案就是 O_APPEND 标志。

前面已经给大家多次提到过了 O_APPEND 标志，但是并没有给大家介绍 O_APPEND 的一个非常重要的作用，那就是实现原子操作。当 open 函数的 flags 参数中包含了 O_APPEND 标志，每次执行 write 写入操作时都会将文件当前写位置偏移量移动到文件末尾，然后再写入数据，这里“移动当前写位置偏移量到文件末尾、写入数据”这两个操作步骤就组成了一个原子操作，加入 O_APPEND 标志后，不管怎么写入数据都会是从文件末尾写，这样就不会导致出现“进程 A 写入的数据覆盖了进程 B 写入的数据”这种情况了。

(2)pread()和 pwrite()

pread()和 pwrite()都是系统调用，与 read()、write()函数的作用一样，用于读取和写入数据。区别在于，pread()和 pwrite()可用于实现原子操作，调用 pread 函数或 pwrite 函数可传入一个位置偏移量 offset 参数，用于指定文件当前读或写的位置偏移量，所以调用 pread 相当于调用 lseek 后再调用 read；同理，调用 pwrite 相当于调用 lseek 后再调用 write。所以可知，使用 pread 或 pwrite 函数不需要使用 lseek 来调整当前位置偏移量，并会将“移动当前位置偏移量、读或写”这两步操作组成一个原子操作。

pread、pwrite 函数原型如下所示（可通过"man 2 pread"或"man 2 pwrite"命令来查看）：

```
#include <unistd.h>

ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

首先调用这两个函数需要包含头文件<unistd.h>。

函数参数和返回值含义如下：

fd、buf、count 参数与 read 或 write 函数意义相同。

offset：表示当前需要进行读或写的位置偏移量。

返回值：返回值与 read、write 函数返回值意义一样。

虽然 pread（或 pwrite）函数相当于 lseek 与 pread（或 pwrite）函数的集合，但还是有下列区别：

- 调用 pread 函数时，无法中断其定位和读操作（也就是原子操作）；
- 不更新文件表中的当前位置偏移量。

关于第二点我们可以编写一个简单地代码进行测试，测试代码如下所示：

示例代码 3.9.1 pread 函数测试

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

int main(void)
{
    unsigned char buffer[100];
    int fd;
```

```
    int ret;

    /* 打开文件 test_file */
    fd = open("./test_file", O_RDWR);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }

    /* 使用 pread 函数读取数据(从偏移文件头 1024 字节处开始读取) */
    ret = pread(fd, buffer, sizeof(buffer), 1024);
    if (-1 == ret) {
        perror("pread error");
        goto err;
    }

    /* 获取当前位置偏移量 */
    ret = lseek(fd, 0, SEEK_CUR);
    if (-1 == ret) {
        perror("lseek error");
        goto err;
    }

    printf("Current Offset: %d\n", ret);
    ret = 0;
err:
    /* 关闭文件 */
    close(fd);
    exit(ret);
}
```

在当前目录下存在一个文件 test_file，上述代码中会打开 test_file 文件，然后直接使用 pread 函数读取 100 个字节数据，从偏移文件头部 1024 字节处，读取完成之后再使用 lseek 函数获取到文件当前位置偏移量，并将其打印出来。假如 pread 函数会改变文件表中记录的当前位置偏移量，则打印出来的数据应该是 1024 + 100 = 1124；如果不会改变文件表中记录的当前位置偏移量，则打印出来的数据应该是 0，接下来编译代码测试：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
Current Offset: 0
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.9.2 pread 函数测试结果

从上图中可知，打印出来的数据为 0，正如前面所介绍那样，pread 函数确实不会改变文件表中记录的当前位置偏移量；同理，pwrite 函数也是如此，大家可以把 pread 换成 pwrite 函数再次进行测试，不出意外，打印出来的数据依然是 0。

如果把 pread 函数换成 read（或 write）函数，那么打印出来的数据就是 100 了，因为读取了 100 个字节数据，相应的当前位置偏移量会向后移动 100 个字节。

(3)创建一个文件

前面给大家介绍 open 函数的 O_EXCL 标志的时候，也提到了原子操作，其中介绍到：O_EXCL 可以用于测试一个文件是否存在，如果不存在则创建此文件，如果存在则返回错误，这使得测试和创建两者成为一个原子操作。接下来给大家，创建文件中存在着的一个竞争状态。

假设有这么一个情况：进程 A 和进程 B 都要去打开同一个文件、并且此文件还不存在。进程 A 当前正在运行状态、进程 B 处于等待状态，进程 A 首先调用 open("./file", O_RDWR)函数尝试去打开文件，结果返回错误，也就是调用 open 失败；接着进程 A 时间片耗尽、进程 B 运行，同样进程 B 调用 open("./file", O_RDWR)尝试打开文件，结果也失败，接着进程 B 再次调用 open("./file", O_RDWR | O_CREAT, ...)创建此文件，这一次 open 执行成功，文件创建成功；接着进程 B 时间片耗尽、进程 A 继续运行，进程 A 也调用 open("./file", O_RDWR | O_CREAT, ...)创建文件，函数执行成功，如下图所示：

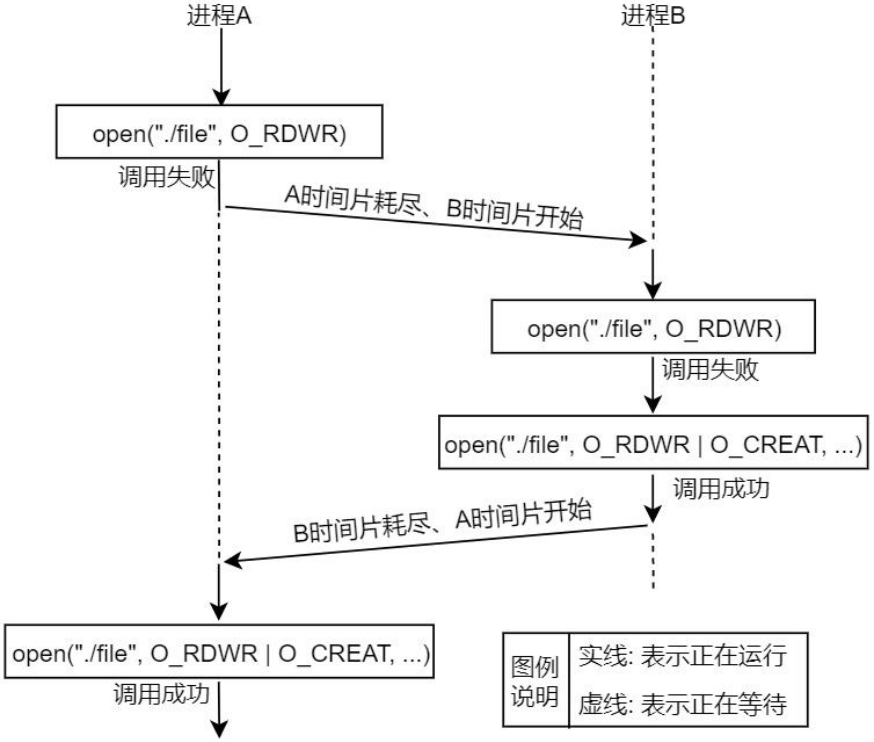

图 3.9.3 创建文件中存在的竞态

从上面的示例可知，进程 A 和进程 B 都会创建出同一个文件，同一个文件被创建两次这是不允许的，那如何规避这样的问题呢？那就是通过使用 O_EXCL 标志，当 open 函数中同时指定了 O_EXCL 和 O_CREAT 标志，如果要打开的文件已经存在，则 open 返回错误；如果指定的文件不存在，则创建这个文件，这里就提供了一种机制，保证进程是打开文件的创建者，将“判断文件是否存在、创建文件”这两个步骤合成为一个原子操作，有了原子操作，就保证不会出现图 3.9.3 中所示的情况。

1.10 fcntl 和 ioctl

本小节给大家介绍两个新的系统调用：fcntl()和 ioctl()。

1.10.1 fcntl 函数

fcntl()函数可以对一个已经打开的文件描述符执行一系列控制操作，譬如复制一个文件描述符（与 dup、dup2 作用相同）、获取/设置文件描述符标志、获取/设置文件状态标志等，类似于一个多功能文件描述符管理工具箱。fcntl()函数原型如下所示（可通过"man 2 fcntl"命令查看）：

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

int fcntl(int fd, int cmd, ... /* arg */ )
```

函数参数和返回值含义如下：

fd：文件描述符。

cmd：操作命令。此参数表示我们将要对 fd 进行什么操作，cmd 参数支持很多操作命令，大家可以打开 man 手册查看到这些操作命令的详细介绍，这些命令都是以 F_XXX 开头的，譬如 F_DUPFD、F_GETFD、F_SETFD 等，不同的 cmd 具有不同的作用，cmd 操作命令大致可以分为以下 5 种功能：

- 复制文件描述符（cmd=F_DUPFD 或 cmd=F_DUPFD_CLOEXEC）；
- 获取/设置文件描述符标志（cmd=F_GETFD 或 cmd=F_SETFD）；
- 获取/设置文件状态标志（cmd=F_GETFL 或 cmd=F_SETFL）；
- 获取/设置异步 IO 所有权（cmd=F_GETOWN 或 cmd=F_SETOWN）；
- 获取/设置记录锁（cmd=F_GETLK 或 cmd=F_SETLK）；

这里列举出来，并不需要全部学会每一个 cmd 的作用，因为有些内容并没有给大家提及到，譬如什么异步 IO、锁之类的概念，在后面的学习过程中，当学习到相关知识内容的时候再给大家介绍。

…：fcntl 函数是一个可变参函数，第三个参数需要根据不同的 cmd 来传入对应的实参，配合 cmd 来使用。

返回值：执行失败情况下，返回-1，并且会设置 errno；执行成功的情况下，其返回值与 cmd（操作命令）有关，譬如 cmd=F_DUPFD（复制文件描述符）将返回一个新的文件描述符、cmd=F_GETFD（获取文件描述符标志）将返回文件描述符标志、cmd=F_GETFL（获取文件状态标志）将返回文件状态标志等。

fcntl 使用示例

(1)复制文件描述符

前面给大家介绍了 dup 和 dup2，用于复制文件描述符，除此之外，我们还可以通过 fcntl 函数复制文件描述符，可用的 cmd 包括 F_DUPFD 和 F_DUPFD_CLOEXEC，这里就只介绍 F_DUPFD，F_DUPFD_CLOEXEC 暂时先不讲。

当 cmd=F_DUPFD 时，它的作用会根据 fd 复制出一个新的文件描述符，此时需要传入第三个参数，第三个参数用于指出新复制出的文件描述符是一个大于或等于该参数的可用文件描述符（没有使用的文件描述符）；如果第三个参数等于一个已经存在的文件描述符，则取一个大于该参数的可用文件描述符。

测试代码如下所示：

示例代码 3.10.1 fcntl 复制文件描述符

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

int main(void)
{
    int fd1, fd2;
    int ret;

    /* 打开文件 test_file */
    fd1 = open("./test_file", O_RDONLY);
    if (-1 == fd1) {
        perror("open error");
        exit(-1);
    }

    /* 使用 fcntl 函数复制一个文件描述符 */
    fd2 = fcntl(fd1, F_DUPFD, 0);
    if (-1 == fd2) {
        perror("fcntl error");
        ret = -1;
        goto err;
    }

    printf("fd1: %d\nfd2: %d\n", fd1, fd2);

    ret = 0;
    close(fd2);
err:
    /* 关闭文件 */
    close(fd1);
    exit(ret);
}
```

在当前目录下存在 test_file 文件，上述代码会打开此文件，得到文件描述符 fd1，之后再使用 fcntl 函数复制 fd1 得到新的文件描述符 fd2，并将 fd1 和 fd2 打印出来，接下来编译运行：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
fd1: 6
fd2: 7
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.10.1 fcntl 复制文件描述符测试结果

可知复制得到的文件描述符是 7，因为在执行 fcntl 函数时，传入的第三个参数是 0，也就时指定复制得到的新文件描述符必须要大于或等于 0，但是因为 0~6 都已经被占用了，所以分配得到的 fd 就是 7；如果传入的第三个参数是 100，那么 fd2 就会等于 100，大家可以自己动手测试。

(2)获取/设置文件状态标志

cmd=F_GETFL 可用于获取文件状态标志，cmd=F_SETFL 可用于设置文件状态标志。cmd=F_GETFL 时不需要传入第三个参数，返回值成功表示获取到的文件状态标志；cmd=F_SETFL 时，需要传入第三个参数，此参数表示需要设置的文件状态标志。

这些标志指的就是我们在调用 open 函数时传入的 flags 标志，可以指定一个或多个（通过位或 | 运算符组合），但是文件权限标志（O_RDONLY、O_WRONLY、O_RDWR）以及文件创建标志（O_CREAT、O_EXCL、O_NOCTTY、O_TRUNC）不能被设置、会被忽略；在 Linux 系统中，只有 O_APPEND、O_ASYNC、O_DIRECT、O_NOATIME 以及 O_NONBLOCK 这些标志可以被修改，这里面有些标志并没有给大家介绍过，后面我们在用到的时候再给大家介绍。所以对于一个已经打开的文件描述符，可以通过这种方式添加或移除标志。

测试代码如下：

示例代码 3.10.2 fcntl 读取/设置文件状态标志

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

int main(void)
{
    int fd;
    int ret;
    int flag;

    /* 打开文件 test_file */
    fd = open("./test_file", O_RDWR);
    if (-1 == fd) {
        perror("open error");
        exit(-1);
    }
```

```
    /* 获取文件状态标志 */
    flag = fcntl(fd, F_GETFL);
    if (-1 == flag) {
        perror("fcntl F_GETFL error");
        ret = -1;
        goto err;
    }

    printf("flags: 0x%x\n", flag);

    /* 设置文件状态标志,添加 O_APPEND 标志 */
    ret = fcntl(fd, F_SETFL, flag | O_APPEND);
    if (-1 == ret) {
        perror("fcntl F_SETFL error");
        goto err;
    }

    ret = 0;
err:
    /* 关闭文件 */
    close(fd);
    exit(ret);
}
```

上述代码会打开 test_file 文件，得到文件描述符 fd，之后调用 fcntl(fd, F_GETFL)来获取到当前文件状态标志 flag，并将其打印来；接着调用 fcntl(fd, F_SETFL, flag | O_APPEND)设置文件状态标志，在原标志的基础上添加 O_APPEND 标志。接下来编译测试:

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ gcc -o testApp testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
testApp  testApp.c  test_file
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ./testApp
flags: 0x8002
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.10.2 fcntl 测试结果 2

以上给大家介绍了 fcntl 函数的两种用法，除了这两种用法之外，还有其它多种不同的用法，这里暂时先不介绍了，后面学习到相应知识点的时候再给大家讲解。

1.10.2 ioctl 函数

ioctl()可以认为是一个文件 IO 操作的杂物箱，可以处理的事情非常杂、不统一，一般用于操作特殊文件或硬件外设，此函数将会在进阶篇中使用到，譬如可以通过 ioctl 获取 LCD 相关信息等，本小节只是给大家引出这个系统调用，暂时不会用到。此函数原型如下所示（可通过"man 2 ioctl"命令查看）：

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

```
int ioctl(int fd, unsigned long request, ...);
```

使用此函数需要包含头文件<sys/ioctl.h>。

函数参数和返回值含义如下：

fd：文件描述符。

request：此参数与具体要操作的对象有关，没有统一值，后面用到的时候再给大家介绍。

...：此函数是一个可变参函数，第三个参数需要根据 request 参数来决定，配合 request 来使用。

返回值：成功返回 0，失败返回-1。

关于 ioctl 函数就给大家介绍这么多，目的仅仅只是给大家引出这个系统调用，我们将会在第二篇<进阶篇>中给大家细说。

1.11 截断文件

使用系统调用 truncate()或 ftruncate()可将普通文件截断为指定字节长度，其函数原型如下所示：

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

int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

这两个函数的区别在于：ftruncate()使用文件描述符 fd 来指定目标文件，而 truncate()则直接使用文件路径 path 来指定目标文件，其功能一样。

这两个函数都可以对文件进行截断操作，将文件截断为参数 length 指定的字节长度，什么是截断？如果文件目前的大小大于参数 length 所指定的大小，则多余的数据将被丢失，类似于多余的部分被“砍”掉了；如果文件目前的大小小于参数 length 所指定的大小，则将其进行扩展，对扩展部分进行读取将得到空字节"\0"。

使用 ftruncate()函数进行文件截断操作之前，必须调用 open()函数打开该文件得到文件描述符，并且必须要具有可写权限，也就是调用 open()打开文件时需要指定 O_WRONLY 或 O_RDWR。

调用这两个函数并不会导致文件读写位置偏移量发生改变，所以截断之后一般需要重新设置文件当前的读写位置偏移量，以免由于之前所指向的位置已经不存在而发生错误（譬如文件长度变短了，文件当前所指向的读写位置已不存在）。

调用成功返回 0，失败将返回-1，并设置 errno 以指示错误原因。

使用示例

示例代码 3.11.1 演示了文件的截断操作，分别使用 ftruncate()和 truncate()将当前目录下的文件 file1 截断为长度 0、将文件 file2 截断为长度 1024 个字节。

示例代码 3.11.1 文件截断操作

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

```c
#include <fcntl.h>

int main(void)
{
    int fd;

    /* 打开 file1 文件 */
    if (0 > (fd = open("./file1", O_RDWR))) {
        perror("open error");
        exit(-1);
    }

    /* 使用 ftruncate 将 file1 文件截断为长度 0 字节 */
    if (0 > ftruncate(fd, 0)) {
        perror("ftruncate error");
        exit(-1);
    }

    /* 使用 truncate 将 file2 文件截断为长度 1024 字节 */
    if (0 > truncate("./file2", 1024)) {
        perror("truncate error");
        exit(-1);
    }

    /* 关闭 file1 退出程序 */
    close(fd);
    exit(0);
}
```

上述代码中，首先使用 open()函数打开文件 file1，得到文件描述符 fd，接着使用 ftruncate()系统调用将文件截断为 0 长度，传入 file1 文件对应的文件描述符；接着调用 truncate()系统调用将文件 file2 截断为 1024 字节长度，传入 file2 文件的相对路径。

接下来进行测试，在当前目录下准备两个文件 file1 和 file2，如下所示：

```
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls
file1  file2  testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
dt@dt-virtual-machine:~/vscode_ws/2_chapter$ ls -l
总用量 12
-rw-rw-r-- 1 dt dt 592 5月  28 18:25 file1
-rw-rw-r-- 1 dt dt 592 5月  28 18:25 file2
-rw-rw-r-- 1 dt dt 592 5月  28 18:25 testApp.c
dt@dt-virtual-machine:~/vscode_ws/2_chapter$
```

图 3.11.1 准备 file1 文件和 file2 文件

可以看到 file1 和 file2 文件此时均为 592 字节大小，接下来运行测试代码：

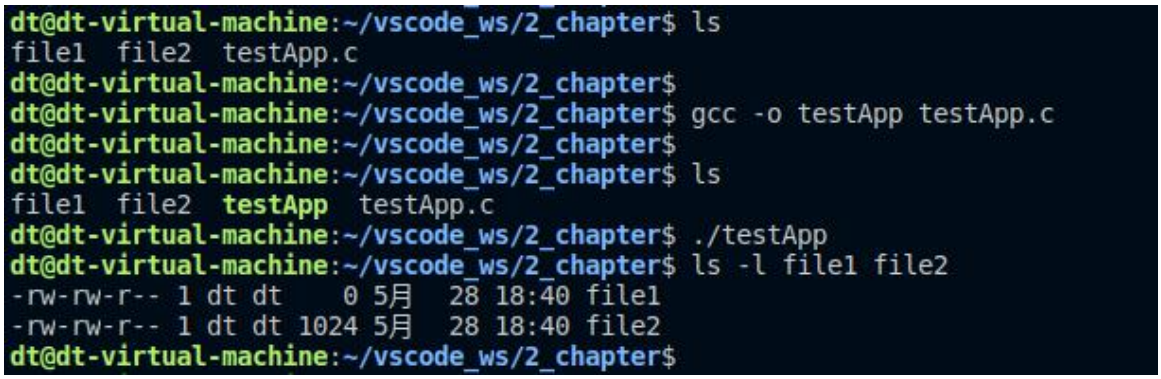

图 3.11.2 测试结果

程序运行之后，file1 文件大小变成了 0，而 file2 文件大小变成了 1024 字节，与测试代码想要实现的功能是一致的。