[一文件 2](#_Toc163305767)

[一、 文件IO 2](#_Toc163305768)

[1. 定义 2](#_Toc163305769)

[2. IO操作流程 2](#_Toc163305770)

[3. 文件描述符 2](#_Toc163305771)

[4. OPEN 3](#_Toc163305772)

[5. Write 5](#_Toc163305773)

[6. read 5](#_Toc163305774)

[7. Close 5](#_Toc163305775)

# 入门

## 驱动编程

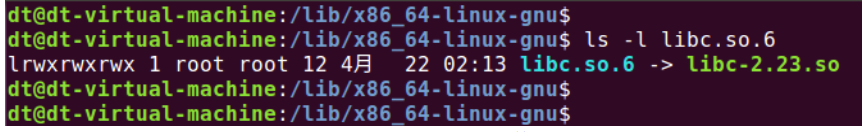
Linux 驱动编程指的是基于内核驱动框架开发驱动程序， 驱动开发工程师通过调用 Linux 内核提供的接口完成设备驱动的注册， 驱动程序负责底层硬件操作相关逻辑

## **应用编程**

而 Linux 应用编程（系统编程）则指的是基于 Linux 操作系统的应用编程，在应用程序中通过调  
用系统调用 API 完成应用程序的功能和逻辑， 应用程序运行于操作系统之上。通常在操作系统下有两种不  
同的状态：内核态和用户态，应用程序运行在用户态、而内核则运行在内核态。

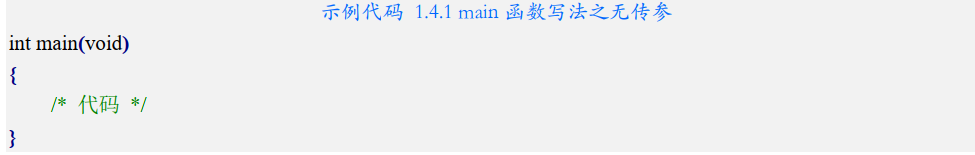
## 确定 **Linux** 系统的 **glibc** 版本

/lib/x86\_64-linux-gnu 目录下，

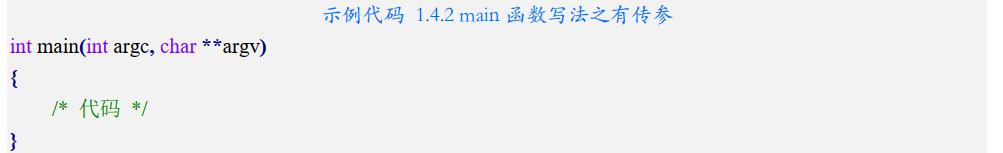


## Main函数

main 函数的形参一般会有两种写法，如果执行应用程序无需传参



如果在执行应用程序的时候需要向应用程序传递参数



argc 形参表示传入参数的个数，包括应用程序自身路径和程序名，并且这些参数都是作为字符串的形式传递给 main 函数:

./hello 112233  
argv[0]等于"./hello"  
argv[1]等于"112233"

# 文件 I/O 基础

一个通用的 IO 模型通常包括打开文件、读写文件、关闭文  
件这些基本操作

### 定义



### IO操作流程

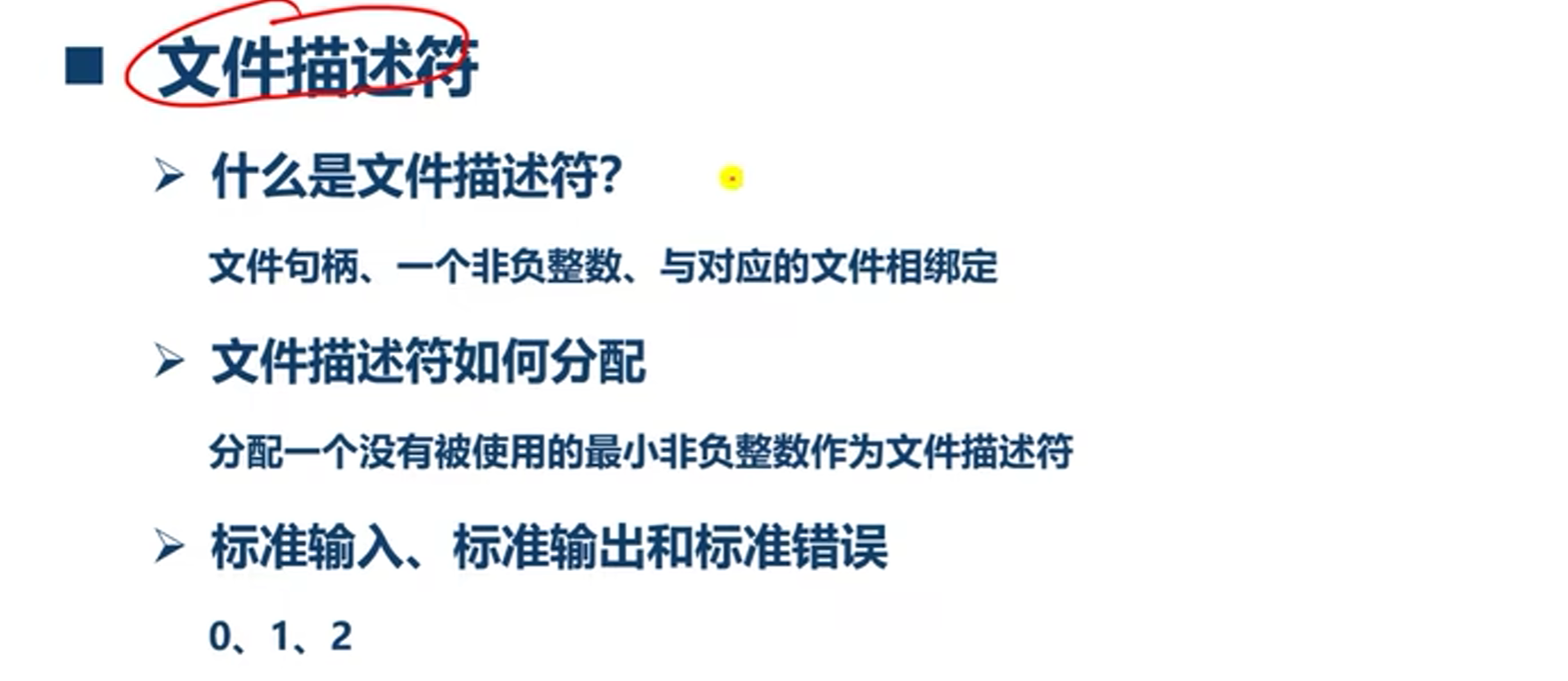
1打开文件

2读写文件

3关闭文件

### 文件描述符

文件描述符数字最大值为 1023（0~1023） 。 每一个被打开的文件在同一个进程中都有一个唯一的文件描述符，不会重复，如果文件被关闭后，它对应的文件描述符将会被释放，那么这个文件描述符将可以再次分配给其它打开的文件、与对应的文件绑定起来。



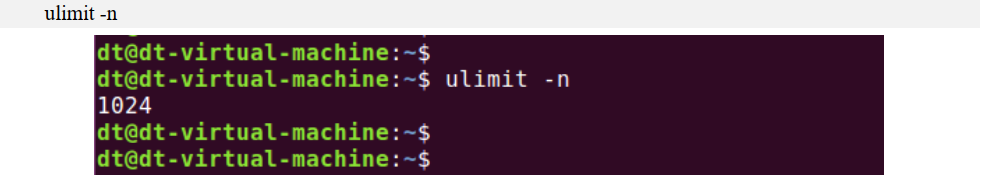
### ulimit 命令来查看进程可打开的最大文件数

### **man**

在 Linux 系统下，可以通过 man 命令（也叫 man 手册） 来查看某一个 Linux 系统调用的帮助信息， man  
命令可以将该系统调用的详细信息显示出来

Man man 查看man的用发

man 2 open #查看 open 函数的帮助信息

man 命令后面跟着两个参数，数字 2 表示系统调用， man 命令除了可以查看系统调用的帮助信息外，还可以查看 Linux 命令（对应数字 1）以及标准 C 库函数（对应数字 3）所对应的帮助信息；最后一个参数 open 表示需要查看的系统调用函数名 

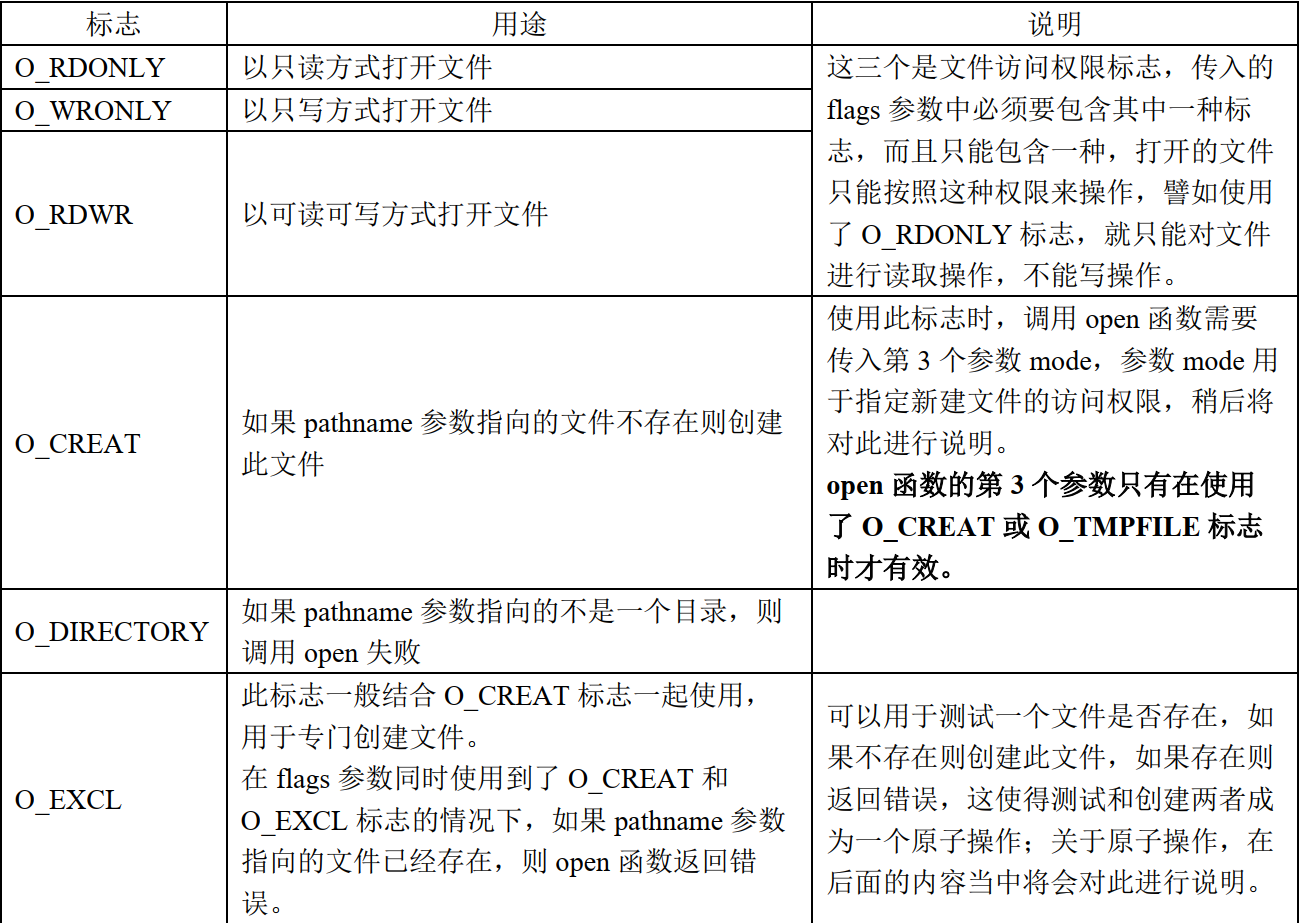
### OPEN

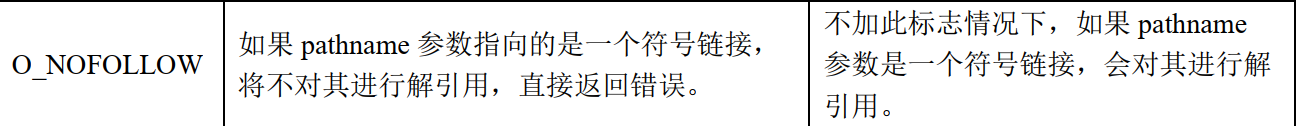
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
int open(const char \*pathname, int flags);int open(const char \*pathname, int flags, mode\_t mode);

第三个参数 mode 需要在第二个参数 flags 满足条件时才会有效，在应用程序中使用 open 函数时，需要包含 3 个头文件“ #include，<sys/types.h>”、“#include <sys/stat.h>”、“#include <fcntl.h>”

pathname： 字符串类型，用于标识需要打开或创建的文件

flags： 调用 open 函数时需要提供的标志， 包括文件访问模式标志以及其它文件相关标志，我们传入 flags 参数时既可以单独使用某一个标志，也可以通过位或运算（|） 将多个标志进行组合





mode： 此参数用于指定新建文件的访问权限，只有当 flags 参数中包含 O\_CREAT 或 O\_TMPFILE 标志时才有效（O\_TMPFILE 标志用于创建一个临时文件）。



例子

使用 open 函数打开一个指定的文件（譬如/home/dengtao/hello），如果该文件不存在则创建该文件，创建该文件时，将文件权限设置如下

int fd = open("/home/dengtao/hello", O\_RDWR | O\_CREAT, S\_IRWXU | S\_IRGRP | S\_IROTH);  
if (-1 == fd)  
return fd;

O\_APPEND **和** O\_TRUNC **标志打开**

open 打开一个文件，默认的读写位置偏移量会处于文件头 ，O\_TRUNC **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**；**



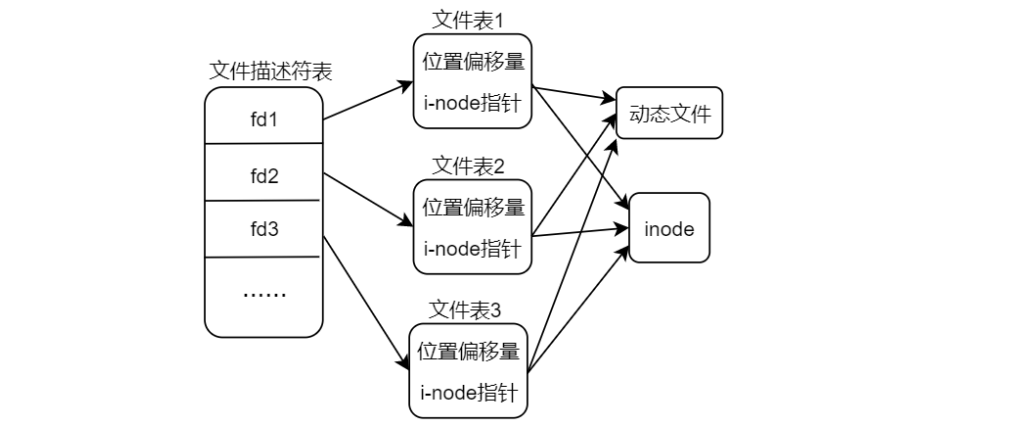
如果 open 函数携带了 O\_APPEND 标志， 调用 open 函数打开文件，  
当每次使用 write()函数对文件进行写操作时，都会自动把文件当前位置偏移量移动到文件末尾， 从文件末尾开始写入数据，也就是意味着每次写入数据都是从文件末尾开始，O\_APPEND标志并不会影响读文件，当读取文件时， O\_APPEND 标志并不会影响读位置偏移量， 即使使用了 O\_APPEND标志，读文件位置偏移量默认情况下依然是文件头

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

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。

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。

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。



多次打开同一文件进行读操作会用到 O\_APPEND 标志，如果不属于那么会分别写数据到文件，存在覆盖问题，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，将分别写更改为接续写O\_APPEND就行

使用，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

### Write



Ssize\_t 如果成功将返回写入的字节数，如果此数字小于 count 参数，这不  
是错误，譬如磁盘空间已满，可能会发生这种情况；如果写入出错，则返回-1。

### read



Ssize\_t为实际读取到的长度，发生错误的时ssize\_t为-1；

### Close

使用 open write read clos函数需要先包含 unistd.h 头文件。

除了使用 close 函数显式关闭文件之外，在 Linux 系统中，当一个进程终止时，内核会自动关闭它打开的所有文件，也就是说在我们的程序中打开了文件， 如果程序终止退出时没有关闭打开的文件，那么内核会自动将程序中打开的文件关闭。

### Sleek

当调用 read()或 write()函数对文件进行读写操作时，就会从当前读写位置  
偏移量开始进行数据读写。当打开文件时，会将读写偏移量设置为指向文件开始位置处，以后每次调用 read()、 write()将自动对其进行调整，

首先调用 lseek 函数需要包含<sys/types.h>和<unistd.h>两个头文件。

off\_t lseek(int fd, off\_t offset, int whence);

offset： 偏移量，以字节为单位。

whence： 用于定义参数 offset 偏移量对应的参考值

SEEK\_SET：读写偏移量将指向 offset 字节位置处

SEEK\_CUR：读写偏移量将指向当前位置偏移量 + offset 字节位置处，offset 可以为正、也可以为负

SEEK\_END：读写偏移量将指向文件末尾 + offset 字节位置处，同样 offset 可以为正、也可以为负

返回值： 成功将返回从文件头部开始算起的位置偏移量

# 文件IO入门

### stat 显示文件或文件系统状态



### 文件打开时的状态

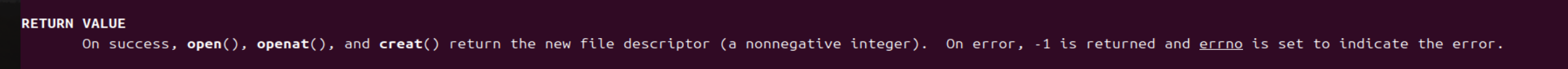
当我们调用 open 函数去打开文件的时候，内核会申请一段内存（一段缓冲区） ，并且将静态文件的数据内容从磁盘这些存储设备中读取到内存中进行管理、 缓存，打开文件后，以后对这个动态文件（缓存）的读写操作，并不是针对磁盘中存放的静态文件

。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

### 返回错误处理与 errno

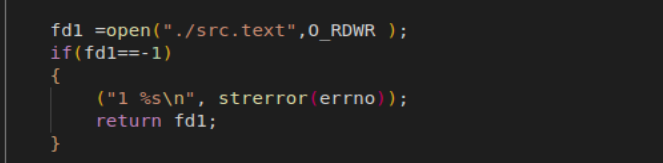
函数执行失败后，会调用 return 退出程序，但是对于我们来说，我们并不知道为什么会出错，什么原因导致此函数执行失败， 因为执行出错之后它们的返回值都是-1，当函数执行发生错误的时候， 操作系统会将这个错误所对应的编号  
赋值给 errno 变量， 每一个进程（程序）都维护了自己的 errno 变量， 它是程序中的全局变量， 该变量用于存储就近发生的函数执行错误编号，也就意味着下一次的错误码会覆盖上一次的错误码。所以由此可知道，当程序中调用函数发生错误的时候，操作系统内部会通过设置程序的 errno 变量来告知调用者究竟发生了什么错误！

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，对于open 可以通过 man 2 open 找到



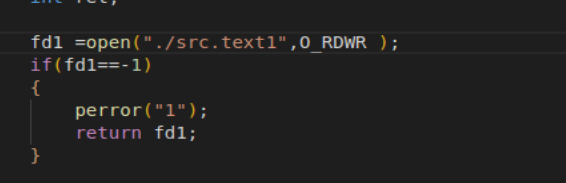
从图中部分描述文字可知，当函数返回错误时会设置 errno，当然这里是以 open 函数为例，其它的系统调用也可以这样查找。

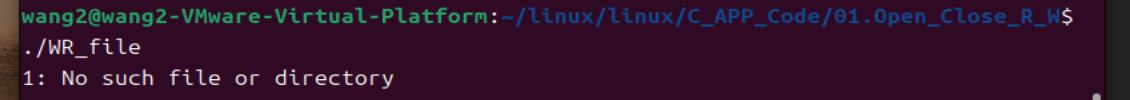
在我们的程序当中如何去获取系统所维护的这个errno变量呢？ 只需要在我们程序当中包含<errno.h>头文件即可，你可以直接认为此变量就是在<errno.h>头文件中的申明的，errno 仅仅只是一个错误编号，C 库函数strerror()， 该函数可以将对应的 errno 转换成适合我们查看的字符串信息



### 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





### \_exit()、\_Exit()、exit()函数

void \_exit(int status);

void \_Exit(int status);

void exit(int status);

exit()和\_Exit()两者等价，用法作用是一样的，需要注意的是这 2 个函数都是系统调用

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。

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

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()**。**

### 空洞文件

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，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。

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。

### Dup与dup2 于复制文件描述符

#include <unistd.h>  
int dup(int oldfd);

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

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，通过复制文件描述符可以实现接续写

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

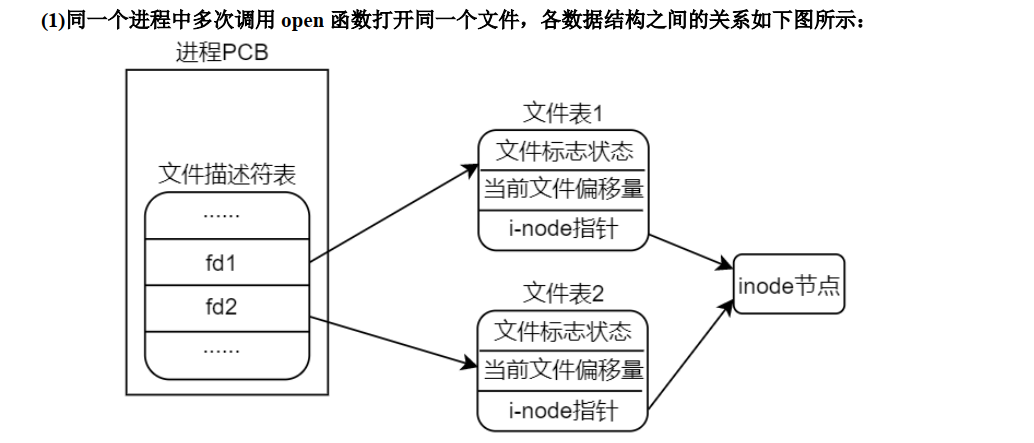
int dup2(int oldfd, int newfd);

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）

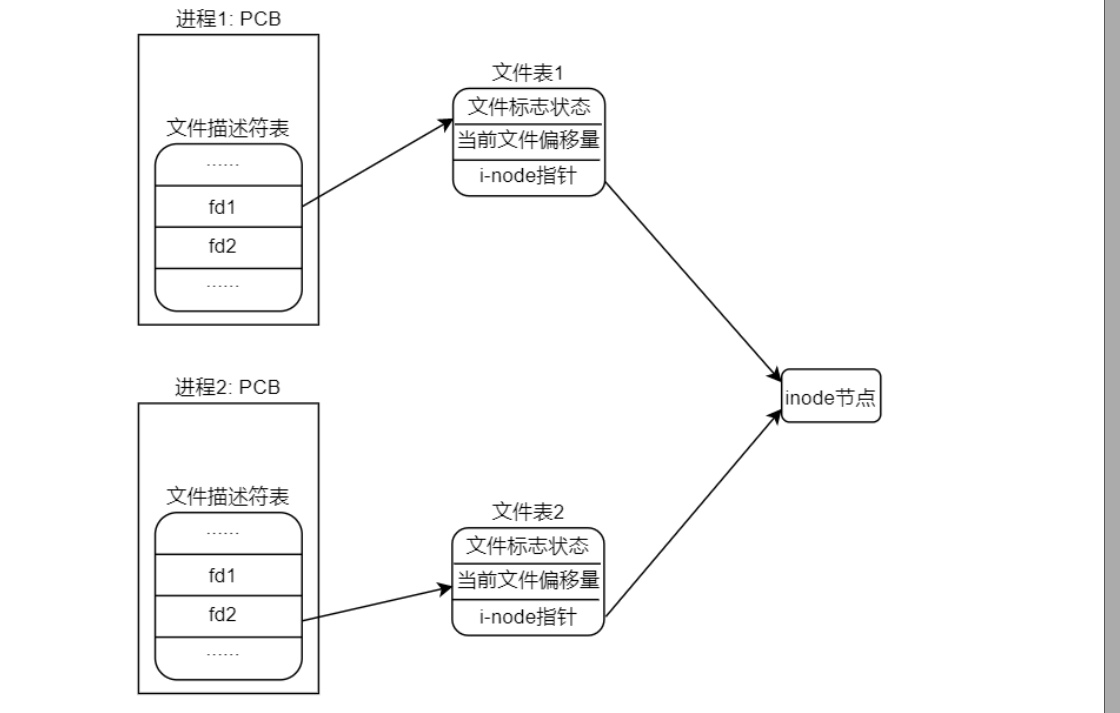
### 文件共享

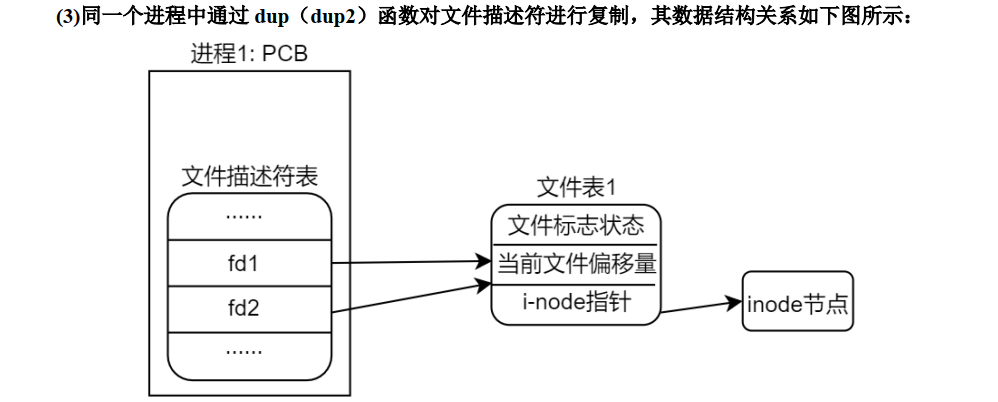
所谓文件共享指的是同一个文件（譬如磁盘上的同一个文件，对应同一个 inode） 被  
多个独立的读写体同时进行 IO 操作，同时进行 IO 操作指的是一个读写体操作文件尚未调用 close 关闭的情况下，另一个读写体去操作文件，譬如多次调用 open 函数重复打开同一个文件得到多个不同的文件描述符、使用 dup()或 dup2()函数对文件描述符进行复制以得到多个不同的文件描述符就是文件共享

常见的三种文件共享的实现方式



(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





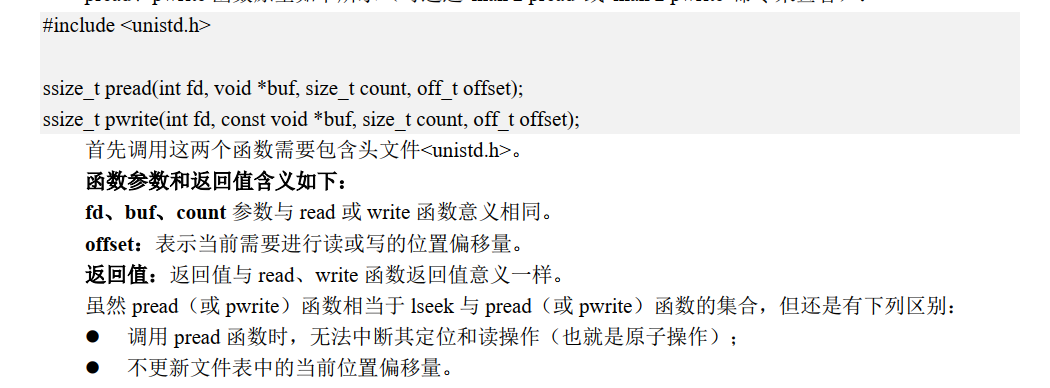
对于文件共享，存在着竞争冒险

### **原子操作与竞争冒险**

所谓原子操作， 是有多步操作组成的一个操作

### **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。



### Printf行缓冲

printf()输出的字符串写入到了标准输出的 stdio 缓冲区中，只有输出换行符时（不考虑缓冲区填满的情况） 才会将这一行数据刷入到内核缓冲区，也就是写入标准输出文件（终端设备）

使标准输出变成无缓冲模式



void main(void)

{

if(setvbuf(stdout,NULL,\_IONBF,0))

{

perror("setvbuf eer");

exit(0);

}

printf("HElLOW WORD1!");

printf("HElLOW WORD2!\n");

printf("HElLOW WORD3!");

}

或者使用fflush(stdout);

void main(void)

{

printf("HElLOW WORD1!");

printf("HElLOW WORD2!\n");

printf("HElLOW WORD3!");

fflush(stdout);

}