Luffy Li (李壮)

嵌入式学习笔记（四）

Linux应用编程和网络编程

[一、Linux中的文件IO 1](#_Toc26344972)

[1.1应用编程框架介绍 1](#_Toc26344973)

[1.1.1.什么是应用编程 1](#_Toc26344974)

[1.1.2.课程思路 1](#_Toc26344975)

[1.1.3.什么是文件IO 1](#_Toc26344976)

[1.2 文件操作的主要接口API 1](#_Toc26344977)

[1.2.1.什么是操作系统API 1](#_Toc26344978)

[1.2.2.文件操作的一般步骤 1](#_Toc26344979)

[1.2.3.重要概念：文件描述符 2](#_Toc26344980)

[1.3一个简单的文件读写实例 2](#_Toc26344981)

[1.3.1.打开文件与关闭文件 2](#_Toc26344982)

[1.3.2.实时查询man手册 2](#_Toc26344983)

[1.3.3.读取文件内容 2](#_Toc26344984)

[1.3.4.向文件中写入 2](#_Toc26344985)

[1.4 open函数的flag详解 3](#_Toc26344986)

[1.4.1.读写权限：O\_RDWR、O\_RDONLY、O\_WRONLY 3](#_Toc26344987)

[1.4.2. 打开存在并有内容的文件时：O\_APPEND、O\_TRUNC 3](#_Toc26344988)

[1.4.3.打开不存在的文件时：O\_CREAT、O\_EXCL 3](#_Toc26344989)

[1.4.4. O\_NONBLOCK 3](#_Toc26344990)

[1.4.5. O\_SYNC 4](#_Toc26344991)

[1.5 文件读写的一些细节 4](#_Toc26344992)

[1.5.1. exit、\_exit、\_Exit退出进程 4](#_Toc26344993)

[1.5.2. errno和perror 4](#_Toc26344994)

[1.5.3. read和write的count 4](#_Toc26344995)

[1.5.4.文件IO效率和标准IO 5](#_Toc26344996)

[1.6 linux系统如何管理文件 5](#_Toc26344997)

[1.6.1.硬盘中的静态文件和inode（i节点） 5](#_Toc26344998)

[1.6.2.内存中被打开的文件和vnode（v节点） 7](#_Toc26344999)

[1.6.3.文件与流的概念 7](#_Toc26345000)

[1.7 lseek详解 7](#_Toc26345001)

[1.7.1. lseek函数介绍 7](#_Toc26345002)

[1.7.2.用lseek计算文件长度 7](#_Toc26345003)

[1.7.3.用lseek构建空洞文件 8](#_Toc26345004)

[1.8 多次打开同一文件与O\_APPEND 9](#_Toc26345005)

[1.8.1.重复打开同一文件读取 9](#_Toc26345006)

[1.8.2.重复打开同一文件写入 9](#_Toc26345007)

[1.8.3.加O\_APPEND解决覆盖问题 10](#_Toc26345008)

[1.8.4. O\_APPEND的实现原理和其原子操作性说明 10](#_Toc26345009)

[1.9 文件共享的实现方式 10](#_Toc26345010)

[1.9.1.什么是文件共享 10](#_Toc26345011)

[1.9.2.文件共享的3种实现方式 10](#_Toc26345012)

[1.9.3.再论文件描述符 11](#_Toc26345013)

[1.10 文件描述符的复制 12](#_Toc26345014)

[1.10.1. dup和dup2函数介绍 12](#_Toc26345015)

[1.10.2.使用dup进行文件描述符复制 12](#_Toc26345016)

[1.10.3.使用dup的缺陷分析 12](#_Toc26345017)

[1.10.4.实验：标准输出stdout重定位 12](#_Toc26345018)

[1.10.5.使用dup2进行文件描述符复制 13](#_Toc26345019)

[1.10.6. dup2共享文件交叉写入测试 13](#_Toc26345020)

[1.10.7.命令行中重定位命令 > 14](#_Toc26345021)

[1.11 fcntl函数介绍 14](#_Toc26345022)

[1.11.1. fcntl的原型和作用 14](#_Toc26345023)

[1.11.2. fcntl的常用cmd 14](#_Toc26345024)

[1.11.3.使用fcntl模拟dup2 15](#_Toc26345025)

[1.12 标准IO库介绍 15](#_Toc26345026)

[1.12.1.标准IO和文件IO有什么区别 15](#_Toc26345027)

[1.12.2.常用标准IO函数介绍 16](#_Toc26345028)

[1.12.3.一个简单的标准IO读写文件实例 16](#_Toc26345029)

[二、文件属性 16](#_Toc26345030)

[2.1 linux中各种文件类型 16](#_Toc26345031)

[2.1.1.普通文件（- regular file） 16](#_Toc26345032)

[2.1.3.字符设备文件（c character） 17](#_Toc26345033)

[2.1.4.块设备文件（b block） 17](#_Toc26345034)

[2.1.5.管道文件（p pipe） 17](#_Toc26345035)

[2.1.6.套接字文件（s socket） 17](#_Toc26345036)

[2.1.7.符号链接文件（l link） 17](#_Toc26345037)

[2.2 常用文件属性获取 18](#_Toc26345038)

[2.2.1. stat、fstat、lstat函数简介 18](#_Toc26345039)

[2.2.2. struct stat结构体简介 18](#_Toc26345040)

[2.2.3.写个程序来查看一些常见属性信息 18](#_Toc26345041)

[2.3 stat函数的应用案例 18](#_Toc26345042)

[2.3.1.用代码判断文件类型 19](#_Toc26345043)

[2.3.2.用代码判断文件权限设置 19](#_Toc26345044)

# 一、Linux中的文件IO

## 1.1应用编程框架介绍

### 1.1.1.什么是应用编程

(1)整个嵌入式linux核心课程包括5个点，按照学习顺序依次是：裸机、C高级、uboot和系统移植、linux应用编程和网络编程、驱动。

(2)典型的嵌入式产品就是基于嵌入式linux操作系统来工作的。典型的嵌入式产品的研发过程就是；第一步让linux系统在硬件上跑起来（系统移植工作），第二步基于linux系统来开发应用程序实现产品功能。

(3)基于linux去做应用编程，其实就是通过调用linux的系统API来实现应用需要完成的任务。

### 1.1.2.课程思路

(1)通过本课程9个小课程的学习，学会如何使用linux系统提供的API（和C库函数）来实现一定的功能，通过学习对应用层的编程有所掌握来配合后面驱动的学习。

(2)如果希望深入学习linux应用尤其是网络编程知识，可以去看一些专门介绍这一块的书。

### 1.1.3.什么是文件IO

IO就是input/output，输入/输出。文件IO的意思就是读写文件。

## 1.2 文件操作的主要接口API

### 1.2.1.什么是操作系统API

(1)API是一些函数，这些函数是由linux系统提供支持的，由应用层程序来调用。

(2)应用层程序通过调用API来调用操作系统中的各种功能，来干活。

(3)学习一个操作系统，其实就是学习使用这个操作系统的API。

(4)今天我们要使用linux系统来读写文件，手段就是学习linux系统API中和文件IO有关的几个：open、close、write、read、lseek

### 1.2.2.文件操作的一般步骤

(1)在linux系统中要操作一个文件，一般是先open打开一个文件，得到一个文件描述符（fd，即file descriptor），然后对文件进行读写操作（或其他操作），最后close关闭文件即可

(2)强调一点：我们对文件进行操作时，一定要先打开文件，打开成功后才能去操作（如果打开本身失败，后面就不用操作了）；最后读写完成之后一定要close关闭文件，否则可能会造成文件损坏。

(3)文件平时是存在块设备中的文件系统中的，我们把这种文件叫静态文件。当我们去open打开一个文件时，linux内核做的操作包括：内核在进程中建立了一个打开文件的数据结构，记录下我们打开的这个文件；内核在内存中申请一段内存，并且将静态文件的内容从块设备中读取到内存中特定地址管理存放（叫动态文件）。

(4)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的，而并不是针对静态文件的。当我们对动态文件进行读写后，此时内存中的动态文件和块设备中的静态文件就不同步了，当我们close关闭动态文件时，close内部内核将内存中的动态文件的内容去更新（同步）块设备中的静态文件。

(5)常见的一些现象：

第一个：打开一个大文件时比较慢。因为CPU要将大文件从块设备中读取到内存中去。

第二个：我们写了一半的文件，如果没有点保存直接关机/断电，重启后文件内容丢失。因为读写文件的目标文件是存放在内存中的动态文件。

(6)为什么要这么设计？

因为块设备本身有读写限制（回忆NnadFlash、SD等块设备的读写特征），本身对块设备进行操作非常不灵活。而内存可以按字节为单位来操作，而且可以随机操作（内存就叫RAM，random），很灵活。所以内核设计文件操作时就这么设计了。

### 1.2.3.重要概念：文件描述符

(1)文件描述符其实实质是一个int型数字，这个数字在一个进程中表示一个特定的含义，当我们open打开一个文件时，操作系统在内存中构建了一些数据结构来表示这个动态文件，然后返回给应用程序一个数字作为文件描述符，这个数字就和我们内存中维护这个动态文件的这些数据结构挂钩绑定上了，以后我们应用程序如果要操作这一个动态文件，只需要用这个文件描述符进行区分。

(2)文件描述符就是用来区分一个程序打开的多个文件的。

(3)文件描述符的作用域就是当前进程，出了当前进程这个文件描述符就没有意义了

## 1.3一个简单的文件读写实例

### 1.3.1.打开文件与关闭文件

(1)linux中的文件描述符fd的合法范围是0或者一个正整数数，不可能是一个负数。

(2)open返回的fd程序必须记录好，以后向这个文件的所有操作都要靠这个fd去对应这个文件，最后关闭文件时也需要fd去指定关闭这个文件。如果在我们关闭文件前fd丢掉了那就惨了，这个文件没法关闭了也没法读写了。

### 1.3.2.实时查询man手册

(1)当我们写应用程序时，很多API原型都不可能记得，所以要实时查询，用man手册

(2)man 1 xx查linux shell命令，man 2 xxx查API， man 3 xxx查库函数

### 1.3.3.读取文件内容

(1)ssize\_t read(int fd, void \*buf, size\_t count);

fd表示要读取哪个文件，fd一般由前面的open返回得到

buf是应用程序自己提供的一段内存缓冲区，用来存储读出的内容

count是我们要读取的字节数

返回值ssize\_t类型是linux内核用typedef重定义的一个类型（其实就是int），返回值表示成功读取的字节数。为什么要这样搞，为的是创建平台无关类型，方便移植。

### 1.3.4.向文件中写入

(1)写入用write系统调用，write的原型和理解方法和read相似

(2)注意const在buf前面的作用，结合C语言高级专题中的输入型参数和输出型参数一节来理解。

(3)注意buf的指针类型为void，结合C语言高级专题中void类型含义的讲解

## 1.4 open函数的flag详解

### 1.4.1.读写权限：O\_RDWR、O\_RDONLY、O\_WRONLY

(1)linux中文件有读写权限，我们在open打开文件时也可以附带一定的权限说明（譬如O\_RDONLY就表示以只读方式打开，O\_WRONLY表示以只写方式打开，O\_RDWR表示以可读可写方式打开）

(2)当我们附带了权限后，打开的文件就只能按照这种权限来操作。

### 1.4.2. 打开存在并有内容的文件时：O\_APPEND、O\_TRUNC

(1)思考一个问题：当我们打开一个已经存在并且内部有内容的文件再写入时会怎么样？

可能结果1：新内容会替代原来的内容（原来的内容就不见了，丢了）（O\_TRUNC）

可能结果2：新内容添加在前面，原来的内容继续在后面

可能结果3：新内容附加在后面，原来的内容还在前面（O\_APPEND）

可能结果4：不读不写的时候，原来的文件中的内容保持不变

(2)O\_TRUNC属性去打开文件时，如果这个文件中本来是有内容的，则原来的内容会被丢弃。这就对应上面的结果1

(3)O\_APPEND属性去打开文件时，如果这个文件中本来是有内容的，则新写入的内容会接续到原来内容的后面，对应结果3

(4)默认不使用O\_APPEND和O\_TRUNC属性时就是结果4

### 1.4.3.打开不存在的文件时：O\_CREAT、O\_EXCL

(1)思考：当我们去打开一个并不存在的文件时会怎样？当我们open打开一个文件时如果这个文件名不存在则会打开文件错误。

(2)vi或者windows下的notepad++，都可以直接打开一个尚未存在的文件。

(3)open的flag O\_CREAT就是为了应对这种打开一个并不存在的文件的。O\_CREAT就表示我们当前打开的文件并不存在，我们是要去创建并且打开它。

(4)思考：当我们open使用了O\_CREAT，但是文件已经存在的情况下会怎样？经过实验验证发现结果是报错。

(5)结论：open中加入O\_CREAT后，不管原来这个文件存在与否都能打开成功，如果原来这个文件不存在则创建一个空的新文件，如果原来这个文件存在则会重新创建这个文件，原来的内容会被消除掉（有点类似于先删除原来的文件再创建一个新的）

(6)这样可能带来一个问题？我们本来是想去创建一个新文件的，但是把文件名搞错了弄成了一个老文件名，结果老文件就被意外修改了。我们希望的效果是：如果我CREAT要创建的是一个已经存在的名字的文件，则给我报错，不要去创建。

(7)这个效果就要靠O\_EXCL标志和O\_CREAT标志来结合使用。当这连个标志一起的时候，则没有文件时创建文件，有这个文件时会报错提醒我们。

(8)open函数在使用O\_CREAT标志去创建文件时，可以使用第三个参数mode来指定要创建的文件的权限。mode使用4个数字来指定权限的，其中后面三个很重要，对应我们要创建的这个文件的权限标志。譬如一般创建一个可读可写不可执行的文件就用0666

fd = open(“a.txt”, O\_RDWR | O\_CREAT |O\_EXCL, 0666 );

### 1.4.4. O\_NONBLOCK

(1)阻塞与非阻塞。如果一个函数是阻塞式的，则我们调用这个函数时当前进程有可能被卡住（阻塞住，实质是这个函数内部要完成的事情条件不具备，当前没法做，要等待条件成熟），函数被阻塞住了就不能立刻返回；如果一个函数是非阻塞式的那么我们调用这个函数后一定会立即返回，但是函数有没有完成任务不一定。

(2)阻塞和非阻塞是两种不同的设计思路，并没有好坏。总的来说，阻塞式的结果有保障但是时间没保障；非阻塞式的时间有保障但是结果没保障。

(3)操作系统提供的API和由API封装而成的库函数，有很多本身就是被设计为阻塞式或者非阻塞式的，所以我们应用程序调用这些函数的时候心里得非常清楚。

(4)我们打开一个文件（设备文件）默认就是阻塞式的，如果你希望以非阻塞的方式打开文件，则flag中要加O\_NONBLOCK标志。

(5)只用于设备文件，而不用于普通文件。

### 1.4.5. O\_SYNC

(1)write阻塞等待底层完成写入才返回到应用层。

(2)无O\_SYNC时write只是将内容写入底层缓冲区即可返回，然后底层（操作系统中负责实现open、write这些操作的那些代码，也包含OS中读写硬盘等底层硬件的代码）在合适的时候会将buf中的内容一次性的同步到硬盘中。这种设计是为了提升硬件操作的性能和效率，提升硬件寿命；但是有时候我们希望硬件不要等待，直接将我们的内容写入硬盘中，这时候就可以用O\_SYNC标志。

## 1.5 文件读写的一些细节

### 1.5.1. exit、\_exit、\_Exit退出进程

(1)当我们程序在前面步骤操作失败导致后面的操作都没有可能进行下去时，应该在前面的错误监测中结束整个程序，不应该继续让程序运行下去了。

(2)我们如何退出程序？

第一种：在main用return，一般原则是程序正常终止return 0，如果程序异常终止则return -1。

第二种：正式终止进程（程序）应该使用exit或者\_exit或者\_Exit之一。

### 1.5.2. errno和perror

(1)errno就是error number，意思就是错误号码。linux系统中对各种常见错误做了个编号，当函数执行错误时，函数会返回一个特定的errno编号来告诉我们这个函数到底哪里错了。

(2)errno是由OS来维护的一个全局变量，任何OS内部函数都可以通过设置errno来告诉上层调用者究竟刚才发生了一个什么错误。

(3)errno本身实质是一个int类型的数字，每个数字编号对应一种错误。当我们只看errno时只能得到一个错误编号数字（譬如-37），不适应于人看。

(4)linux系统提供了一个函数perror（意思print error），perror函数内部会读取errno并且将这个不好认的数字直接给转成对应的错误信息字符串，然后print打印出来。

### 1.5.3. read和write的count

(1)count和返回值的关系。count参数表示我们想要写或者读的字节数，返回值表示实际完成的要写或者读的字节数。实现的有可能等于想要读写的，也有可能小于（说明没完成任务）

(2)count再和阻塞非阻塞结合起来，就会更加复杂。如果一个函数是阻塞式的，则我们要读取30个，结果暂时只有20个时就会被阻塞住，等待剩余的10个可以读。

(3)有时候我们写正式程序时，我们要读取或者写入的是一个很庞大的文件（譬如文件有2MB），我们不可能把count设置为2\*1024\*1024，而应该去把count设置为一个合适的数字（譬如2048、4096），然后通过多次读取来实现全部读完。

### 1.5.4.文件IO效率和标准IO

(1)文件IO就指的是我们当前在讲的open、close、write、read等API函数构成的一套用来读写文件的体系，这套体系可以很好的完成文件读写，但是效率并不是最高的。

(2)应用层C语言库函数提供了一些用来做文件读写的函数列表，叫标准IO。标准IO由一系列的C库函数构成（fopen、fclose、fwrite、fread），这些标准IO函数其实是由文件IO封装而来的（fopen内部其实调用的还是open，fwrite内部还是通过write来完成文件写入的）。标准IO加了封装之后主要是为了在应用层添加一个缓冲机制，这样我们通过fwrite写入的内容不是直接进入内核中的buf，而是先进入应用层标准IO库自己维护的buf中，然后标准IO库自己根据操作系统单次write的最佳count来选择好的时机来完成write到内核中的buf（内核中的buf再根据硬盘的特性来选择好的时机去最终写入硬盘中）。



## 1.6 linux系统如何管理文件

### 1.6.1.硬盘中的静态文件和inode（i节点）

https://blog.csdn.net/wwwlyj123321/article/details/100298377

(1)文件平时都在存放在硬盘中的，硬盘中存储的文件以一种固定的形式存放的，我们叫静态文件。

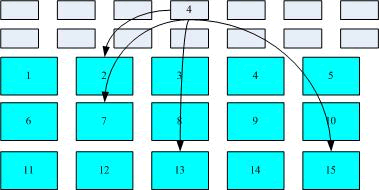
(2)一块硬盘中可以分为两大区域：一个是硬盘内容管理表项，另一个是真正存储内容的区域。操作系统访问硬盘时是先去读取硬盘内容管理表，从中找到我们要访问的那个文件的扇区级别的信息，然后再用这个信息去查询真正存储内容的区域，最后得到我们要的文件。

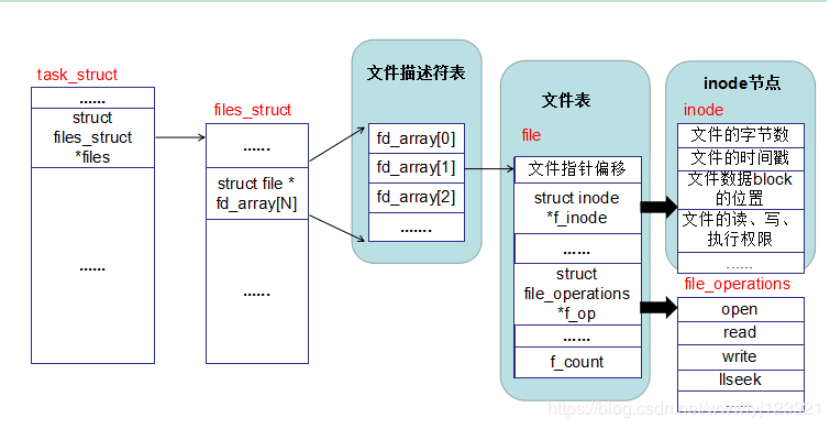
(3)操作系统最初拿到的信息是文件名，最终得到的是文件内容。第一步就是去查询硬盘内容管理表，这个管理表中以文件为单位记录了各个文件的各种信息，每一个文件有一个信息列表index node（我们叫inode，i节点，其实质是一个结构体，这个结构体有很多元素，每个元素记录了这个文件的一些信息，其中就包括文件名、文件在硬盘上对应的扇区号、块号那些东西……）

强调：硬盘管理的时候是以文件为单位的，每个文件一个inode，每个inode有一个数字编号，对应一个结构体，结构体中记录了各种信息。

(4)联系平时实践，大家格式化硬盘（U盘）时发现有：快速格式化和底层格式化。快速格式化非常快，格式化一个32GB的U盘只要1秒钟，普通格式化格式化速度慢。这两个的差异？其实快速格式化就是只删除了U盘中的硬盘内容管理表（其实就是inode），真正存储的内容没有动。这种格式化的内容是有可能被找回的。

如图所示：文件系统先格式化出 inode 和 block 块，假设某文件的权限和属性信息存放到 inode 4 号位置，这个 inode 记录了实际存储文件数据的 block 号有 4 个，分别为 2、7、13、15，由此，操作系统就能快速地找到文件数据的存储位置。





### 1.6.2.内存中被打开的文件和vnode（v节点）

(1)一个程序的运行就是一个进程，我们在程序中打开的文件就属于某个进程。每个进程都有一个数据结构用来记录这个进程的所有信息（叫进程信息表），表中有一个指针会指向一个文件管理表，文件管理表中记录了当前进程打开的所有文件及其相关信息。文件管理表中用来索引各个打开的文件的index就是文件描述符fd，我们最终找到的就是一个已经被打开的文件的管理结构体vnode，virtual node.

(2)一个vnode中就记录了一个被打开的文件的各种信息，而且我们只要知道这个文件的fd，就可以很容易的找到这个文件的vnode进而对这个文件进行各种操作。

### 1.6.3.文件与流的概念

(1)流（stream）对应自然界的水流。文件操作中，文件类似是一个大包裹，里面装了一堆字符，但是文件被读出/写入时都只能一个字符一个字符的进行，而不能一股脑儿的读写，那么一个文件中N多的个字符被挨个一次读出/写入时，这些字符就构成了一个字符流。

(2)流这个概念是动态的，不是静态的。

(3)编程中提到流这个概念，一般都是IO相关的。所以经常叫IO流。文件操作时就构成了一个IO流。

## 1.7 lseek详解

### 1.7.1. lseek函数介绍

(1)文件指针：当我们要对一个文件进行读写时，一定需要先打开这个文件，所以我们读写的所有文件都是动态文件。动态文件在内存中的形态就是文件流的形式。

(2)文件流很长，里面有很多个字节。那我们当前正在操作的是哪个位置？GUI模式下的软件用光标来标识这个当前正在操作的位置，这是给人看的。

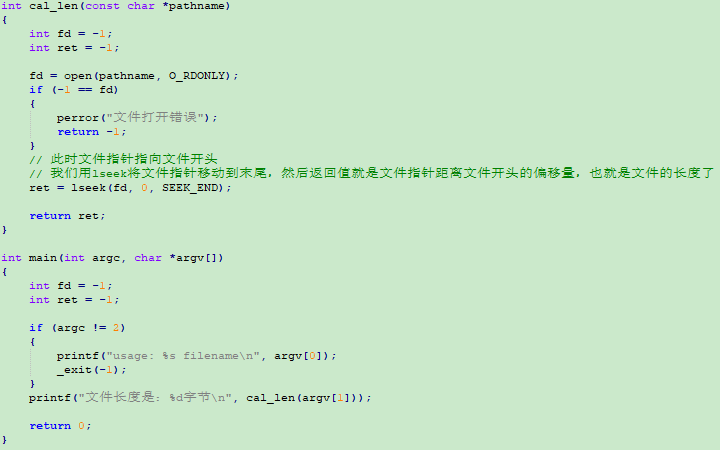
(3)在动态文件中，我们会通过文件指针来表征这个正在操作的位置。所谓文件指针，就是我们文件管理表这个结构体里面的一个指针。所以文件指针其实是vnode中的一个元素。这个指针表示当前我们正在操作文件流的哪个位置。这个指针不能被直接访问，linux系统用lseek函数来访问这个文件指针。

(4)当我们打开一个空文件时，默认情况下文件指针指向文件流的开始。所以这时候去write时写入就是从文件开头开始的。write和read函数本身自带移动文件指针的功能，所以当我write了n个字节后，文件指针会自动向后移动n位。如果需要人为的随意更改文件指针，那就只能通过lseek函数了

(5)read和write函数都是从当前文件指针处开始操作的，所以当我们用lseek显式的将文件指针移动后，那么再去read/write时就是从移动过后的位置开始的。

(6)回顾前面一节中我们从空文件，先write写了12字节，然后read时是空的（但是此时我们打开文件后发现12字节确实写进来了）。

### 1.7.2.用lseek计算文件长度



### 1.7.3.用lseek构建空洞文件

(1)空洞文件就是这个文件中有一段是空的。

(2)普通文件中间是不能有空的，因为我们write时文件指针是依次从前到后去移动的，不可能绕过前面直接到后面。

(3)我们打开一个文件后，用lseek往后跳过一段，再write写入一段，就会构成一个空洞文件。

(4)空洞文件方法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，如果从头开始依次构建时间很长。有一种思路就是将文件分为多段，然后多线程来操作每个线程负责其中一段的写入。

int main(int argc, char \*argv[])

{

int fd = -1;

char buf[100] = {0};

char writebuf[20] = "abcd";

int ret = -1;

fd = open("123.txt", O\_RDWR | O\_CREAT);

if (-1 == fd)

{

perror("文件打开错误");

\_exit(-1);

}

else

{

printf("文件打开成功，fd = %d.\n", fd);

}

ret = lseek(fd, 10, SEEK\_SET);

printf("lseek, ret = %d.\n", ret);

#if 1

ret = write(fd, writebuf, strlen(writebuf));

if (-1 == ret)

{

perror("write失败");

\_exit(-1);

}

else

{

printf("write成功，写入了%d个字符\n", ret);

}

#endif

#if 1

lseek(fd, 0, SEEK\_SET);

ret = read(fd, buf, 20);

if (-1 == ret)

{

printf("read失败\n");

\_exit(-1);

}

else

{

printf("实际读取了%d字节.\n", ret);

printf("文件内容是：[%s].\n", buf);

}

#endif

close(fd);

\_exit(0);

}

## 1.8 多次打开同一文件与O\_APPEND

### 1.8.1.重复打开同一文件读取

(1)一个进程中两次打开同一个文件，然后分别读取，看结果会怎么样

(2)结果无非2种情况：一种是fd1和fd2分别读，第二种是接续读。经过实验验证，证明了结果是fd1和fd2分别读。

(3)分别读说明：我们使用open两次打开同一个文件时，fd1和fd2所对应的文件指针是不同的2个独立的指针。文件指针是包含在动态文件的文件管理表中的，所以可以看出linux系统的进程中不同fd对应的是不同的独立的文件管理表。

### 1.8.2.重复打开同一文件写入

(1)一个进程中2个打开同一个文件，得到fd1和fd2.然后看是分别写还是接续写？

(2)正常情况下我们有时候需要分别写，有时候又需要接续写，所以这两种本身是没有好坏之分的。关键看用户需求

(3)默认情况下应该是：分别写（实验验证过的）

### 1.8.3.加O\_APPEND解决覆盖问题

有时候我们希望接续写而不是分别写？办法就是在open时加O\_APPEND标志即可

### 1.8.4. O\_APPEND的实现原理和其原子操作性说明

(1)O\_APPEND为什么能够将分别写改为接续写？关键的核心的东西是文件指针。分别写的内部原理就是2个fd拥有不同的文件指针，并且彼此只考虑自己的位移。但是O\_APPEND标志可以让write和read函数内部多做一件事情，就是移动自己的文件指针的同时也去把别人的文件指针同时移动。（也就是说即使加了O\_APPEND，fd1和fd2还是各自拥有一个独立的文件指针，但是这两个文件指针关联起来了，一个动了会通知另一个跟着动）

(2)O\_APPEND对文件指针的影响，对文件的读写是原子的。

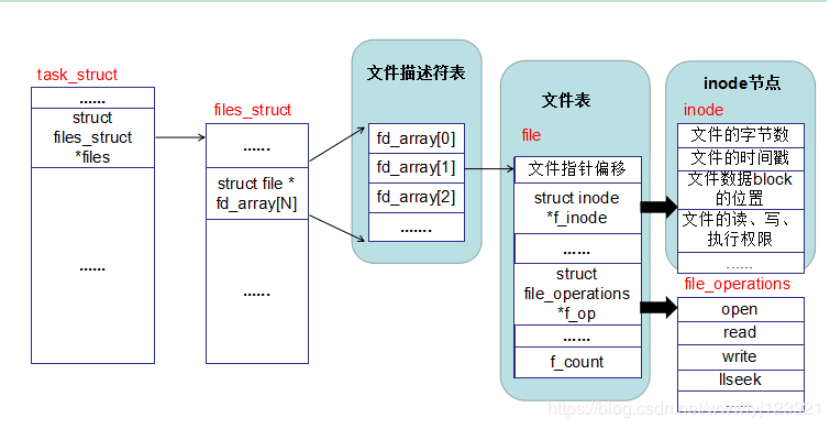
(3)原子操作的含义是：整个操作一旦开始是不会被打断的，必须直到操作结束其他代码才能得以调度运行，这就叫原子操作。每种操作系统中都有一些机制来实现原子操作，以保证那些需要原子操作的任务可以运行。

## 1.9 文件共享的实现方式

### 1.9.1.什么是文件共享

(1)文件共享就是同一个文件（同一个文件指的是同一个inode，同一个pathname）被多个独立的读写体（几乎可以理解为多个文件描述符）去同时（一个打开尚未关闭的同时另一个去操作）操作。

(2)文件共享的意义有很多：譬如我们可以通过文件共享来实现多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。



### 1.9.2.文件共享的3种实现方式

(1)文件共享的核心就是怎么弄出来多个文件描述符指向同一个文件。

(2)常见的有3种文件共享的情况：第一种是同一个进程中多次使用open打开同一个文件，第二种是在不同进程中去分别使用open打开同一个文件（这时候因为两个fd在不同的进程中，所以两个fd的数字可以相同也可以不同），第三种情况是后面要学的，linux系统提供了dup和dup2两个API来让进程复制文件描述符。

(3)我们分析文件共享时的核心关注点在于：分别写/读还是接续写/读

### 1.9.3.再论文件描述符

https://www.2cto.com/kf/201712/709410.html

【我们知道在Linux下一切皆文件，因此我们需要一个东西对这些文件进行管理，此时就需要文件描述符来管理了。文件描述符简称fd，对于内核而言，所有打开的文件都要通过文件描述符来引用。文件描述符是一个递增的非负整数，一旦当我们打开或者创建一个新的文件的时候，内核向进程返回一个文件描述符。

文件描述符一般有以下三个性质：

每个进程都具有自己的一个递增的文件描述符，如果我们关闭了一个文件描述符所占用的正整数，则这个正整数有可能被其它文件描述符所占用。 单个进程能同时打开的文件描述符数量受到limit设置所限制，可以用ulimit -a查看最大文件描述符个数。进程最大打开文件数目默认是1024个。 根据规定，所有的shell启动新的程序的时候，总是将0、1、2这三个数字的文件描述符打开为标准输入、标准输出，标准错误】

(1)文件描述符的本质是一个数字，这个数字本质上是进程表中文件描述符表的一个表项，进程通过文件描述符作为index去索引查表得到文件表指针，再间接访问得到这个文件对应的文件表。

(2)文件描述符这个数字是open系统调用内部由操作系统自动分配的，操作系统分配这个fd时也不是随意分配，也是遵照一定的规律的，我们现在就要研究这个规律。

(3)操作系统规定，fd从0开始依次增加。fd也是有最大限制的，在linux的早期版本中（0.11）fd最大是20，所以当时一个进程最多允许打开20个文件。linux中文件描述符表是个数组（不是链表），所以这个文件描述符表其实就是一个数组，fd是index，文件表指针是value

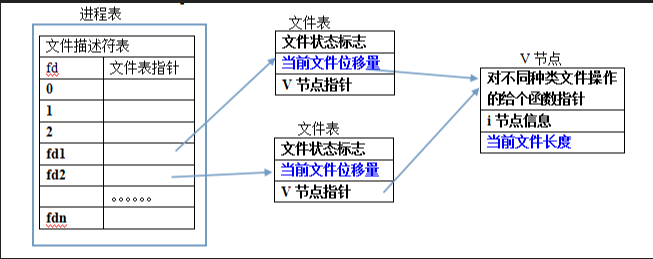
(4)当我们去open时，内核会从文件描述符表中挑选一个最小的未被使用的数字给我们返回。也就是说如果之前fd已经占满了0-9，那么我们下次open得到的一定是10.（但是如果上一个fd得到的是9，下一个不一定是10，这是因为可能前面更小的一个fd已经被close释放掉了）

(5)fd中0、1、2已经默认被系统占用了，因此用户进程得到的最小的fd就是3了。

(6)linux内核占用了0、1、2这三个fd是有用的，当我们运行一个程序得到一个进程时，内部就默认已经打开了3个文件，这三个文件对应的fd就是0、1、2。这三个文件分别叫stdin、stdout、stderr。也就是标准输入、标准输出、标准错误。

(7)标准输入一般对应的是键盘（可以理解为：0这个fd对应的是键盘的设备文件），标准输出一般是LCD显示器（可以理解为：1对应LCD的设备文件）

(8)printf函数其实就是默认输出到标准输出stdout上了。stdio中还有一个函数叫fpirntf，这个函数就可以指定输出到哪个文件描述符中。



## 1.10 文件描述符的复制

### 1.10.1. dup和dup2函数介绍

int dup(int oldfd);

int dup2(int oldfd, int newfd);

dup()和dup2()函数是用来复制一个文件描述符，可以实现文件共享。由于返回新的文件描述符与原来的文件描述符对应同一个文件表，所以它们共享同一个当前文件的偏移量，因此在利用新的文件描述符向文件中写入数据的时候不会出现数据覆盖的问题。所以它们经常被用来重定向到进程的标准输入、标准输出、标准出错。

oldfd：被复制的文件描述符

newfd：在dup2中指定新的文件描述符

### 1.10.2.使用dup进行文件描述符复制

(1)dup系统调用对fd进行复制，会返回一个新的文件描述符（譬如原来的fd是3，返回的就是4）

(2)dup系统调用有一个特点，就是自己不能指定复制后得到的fd的数字是多少，而是由操作系统内部自动分配的，分配的原则遵守fd分配的原则。

(3)dup返回的fd和原来的oldfd都指向oldfd打开的那个动态文件，操作这两个fd实际操作的都是oldfd打开的那个文件。实际上构成了文件共享。

(4)dup返回的fd和原来的oldfd同时向一个文件写入时，结果是分别写还是接续写？验证结果为接续写。

### 1.10.3.使用dup的缺陷分析

dup并不能指定分配的新的文件描述符的数字，dup2系统调用修复了这个缺陷，所以平时项目中实际使用时根据具体情况来决定用dup还是dup2.

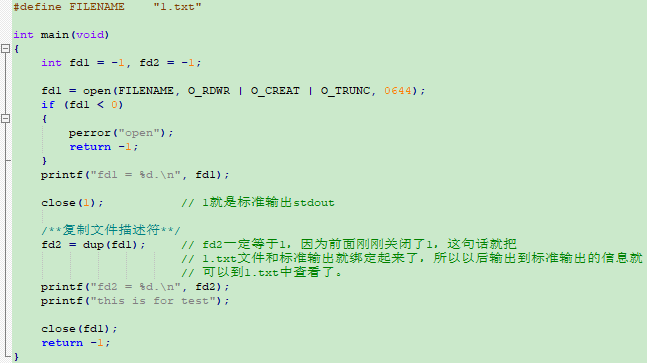
### 1.10.4.实验：标准输出stdout重定位

(1)之前课程讲过0、1、2这三个fd被标准输入、输出、错误通道占用。而且我们可以关闭这三个

(2)我们可以close(1)关闭标准输出，关闭后我们printf输出到标准输出的内容就看不到了

(3)然后我们可以使用dup重新分配得到1这个fd，这时候就把oldfd打开的这个文件和我们1这个标准输出通道给绑定起来了。这就叫标准输出的重定位。

(4)可以看出，我们可以使用close和dup配合进行文件的重定位。



### 1.10.5.使用dup2进行文件描述符复制

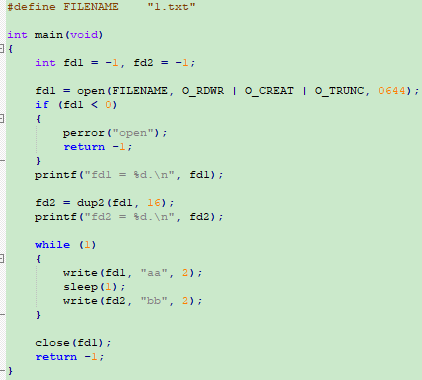
(1)dup2和dup的作用是一样的，都是复制一个新的文件描述符。但是dup2允许用户指定新的文件描述符的数字。

(2)使用方法看man手册函数原型即可。

### 1.10.6. dup2共享文件交叉写入测试

(1)dup2复制的文件描述符，和原来的文件描述符虽然数字不一样，但是这连个指向同一个打开的文件

(2)交叉写入的时候，结果是接续写（实验证明的）。



### 1.10.7.命令行中重定位命令 >

(1)linux中的shell命令执行后，打印结果都是默认进入stdout的（本质上是因为这些命令譬如ls、pwd等都是调用printf进行打印的），所以我们可以在linux的终端shell中直接看到命令执行的结果。

(2)能否想办法把ls、pwd等命令的输出给重定位到一个文件中（譬如2.txt）去，实际上linux终端支持一个重定位的符号>很简单可以做到这点。

(3)这个>的实现原理，其实就是利用open+close+dup，open打开一个文件2.txt，然后close关闭stdout，然后dup将1和2.txt文件关联起来即可。

## 1.11 fcntl函数介绍

### 1.11.1. fcntl的原型和作用

int fcntl(int fd, int cmd, long arg);

(1)fcntl函数是一个多功能文件管理的工具箱，接收2个参数+1个变参。第一个参数是fd表示要操作哪个文件，第二个参数是cmd表示要进行哪个命令操作。变参是用来传递参数的，要配合cmd来使用。

(2)cmd的样子类似于F\_XXX，不同的cmd具有不同的功能。学习时没必要去把所有的cmd的含义都弄清楚（也记不住），只需要弄明白一个作为案例，搞清楚它怎么看怎么用就行了，其他的是类似的。其他的当我们在使用中碰到了一个fcntl的不认识的cmd时再去查man手册即可。

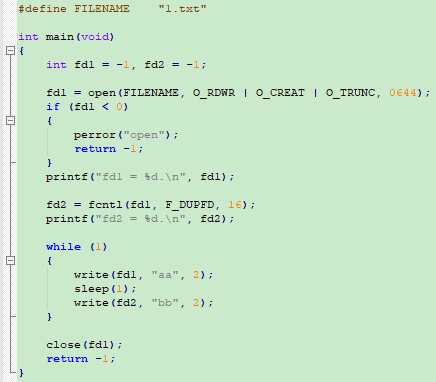
### 1.11.2. fcntl的常用cmd

F\_DUPFD这个cmd的作用是复制文件描述符（作用类似于dup和dup2），这个命令的功能是从可用的fd数字列表中找一个比arg大或者和arg一样大的数字作为oldfd的一个复制的fd，和dup2有点像但是不同。dup2返回的就是我们指定的那个newfd否则就会出错，但是F\_DUPFD命令返回的是>=arg的最小的那一个数字。

实际上调用dup(oldfd)；  
等效于  
        fcntl(oldfd, F\_DUPFD, 0);

而调用dup2(oldfd, newfd)；  
等效于  
        close(oldfd)；  
        fcntl(oldfd, F\_DUPFD, newfd)；

### 1.11.3.使用fcntl模拟dup2



该程序运行结果会是1.txt中aabbaabb…接续写。fd 3和fd 16指向同一份文件表。

## 1.12 标准IO库介绍

### 1.12.1.标准IO和文件IO有什么区别

(1)看起来使用时都是函数，但是：标准IO是C库函数，而文件IO是linux系统的API

(2)C语言库函数是由API封装而来的。库函数内部也是通过调用API来完成操作的，但是库函数因为多了一层封装，所以比API要更加好用一些。

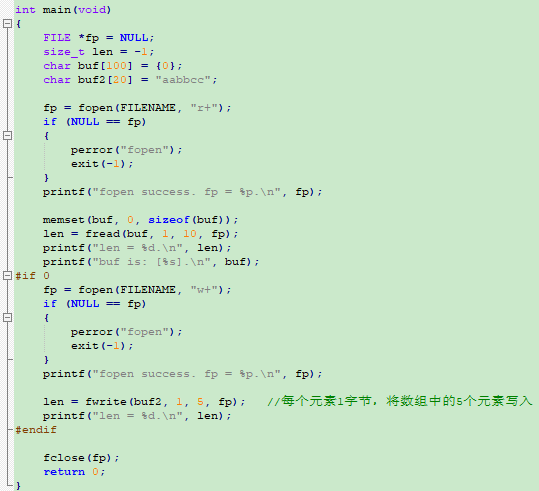
(3)库函数比API还有一个优势就是：API在不同的操作系统之间是不能通用的，但是C库函数在不同操作系统中几乎是一样的。所以C库函数具有可移植性而API不具有可移植性。

(4)性能上和易用性上看，C库函数一般要好一些。譬如IO，文件IO是不带缓存的，而标准IO是带缓存的，因此标准IO比文件IO性能要更高。

### 1.12.2.常用标准IO函数介绍

常见的标准IO库函数有：fopen、fclose、fwrite、fread、ffulsh（让写入硬盘的数据直接刷入而非在缓冲区等待运转）、fseek

### 1.12.3.一个简单的标准IO读写文件实例



# 二、文件属性

## 2.1 linux中各种文件类型

### 2.1.1.普通文件（- regular file）

(1)文本文件。文件中的内容是由文本构成的，文本指的是ASCII码字符。文件里的内容本质上都是数字（不管什么文件内容本质上都是数字，因为计算机中本身就只有1和0），而文本文件中的数字本身应该被理解为这个数字对应的ASCII码。常见的.c文件, .h文件 .txt文件等都是文本文件。文本文件的好处就是可以被人轻松读懂和编写。所以说文本文件天生就是为人类发明的。

(2)二进制文件。二进制文件中存储的本质上也是数字，只不过这些数字并不是文字的编码数字，而是就是真正的数字。常见的可执行程序文件（gcc编译生成的a.out，arm-linux-gcc编译连接生成的.bin）都是二进制文件。

(3)对比：从本质上来看（就是刨除文件属性和内容的理解）文本文件和二进制文件并没有任何区别。都是一个文件里面存放了数字。区别是理解方式不同，如果把这些数字就当作数字处理则就是二进制文件，如果把这些数字按照某种编码格式去解码成文本字符，则就是文本文件。

(4)我们如何知道一个文件是文本文件还是二进制文件？在linux系统层面是不区分这两个的（譬如之前学过的open、read、write等方法操作文件文件和二进制文件时一点区别都没有），所以我们无法从文件本身准确知道文件属于哪种，我们只能本来就知道这个文件的类型然后用这种类型的用法去用他。有时候会用一些后缀名来人为的标记文件的类型。

(5)使用文本文件时，常规用法就是用文本文件编辑器去打开它、编辑它。常见的文本文件编辑器如vim、gedit、notepad++、SourceInsight等，我们用这些文本文件编辑器去打开文件的时候，编辑器会read读出文件二进制数字内容，然后按照编码格式去解码将其还原成文字展现给我们。如果用文本文件编辑器去打开一个二进制文件会如何？这时候编辑器就以为这个二进制文件还是文本文件然后试图去将其解码成文字，但是解码过程很多数字并不对应有意义的文字所以成了乱码。

(6)反过来用二进制阅读工具去读取文本文件会怎么样？得出的就是文本文字所对应的二进制的编码。

2.1.2.目录文件（d directory）

(1)目录就是文件夹，文件夹在linux中也是一种文件，不过是特殊文件。用vi打开一个文件夹就能看到，文件夹其实也是一种特殊文件，里面存的内容包括这个文件的路径，还有文件夹里面的文件列表。

(2)但是文件夹这种文件比较特殊，本身并不适合用普通文件的方式来读写。linux中是使用特殊的一些API来专门读写文件夹的。

### 2.1.3.字符设备文件（c character）

### 2.1.4.块设备文件（b block）

(1)设备文件对应的是硬件设备，也就是说这个文件虽然在文件系统中存在，但是并不是真正存在于硬盘上的一个文件，而是文件系统虚拟制造出来的（叫虚拟文件系统，如/dev /sys /proc等）

(2)虚拟文件系统中的文件大多数不能或者说不用直接读写的，而是用一些特殊的API产生或者使用的，具体在驱动阶段会详解。

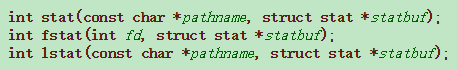
### 2.1.5.管道文件（p pipe）

### 2.1.6.套接字文件（s socket）

### 2.1.7.符号链接文件（l link）

## 2.2 常用文件属性获取

### 2.2.1. stat、fstat、lstat函数简介



(1)每个文件中都附带了这个文件的一些属性（属性信息是存在于文件本身中的，但是它不像文件的内容一样可以被vi打开看到，属性信息只能被专用的API打开看到）

(2)文件属性信息查看的API有三个：stat、fstat、lstat，三个作用一样，参数不同，细节略有不同。

(3)linux命令行下还可以去用stat命令去查看文件属性信息，实际上stat命令内部就是使用stat系统调用来实现的。

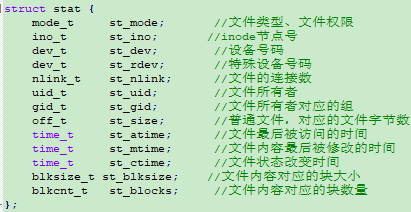
(4)stat这个API的作用就是让内核将我们要查找属性的文件的属性信息结构体的值放入我们传递给stat函数的buf中，当stat这个API调用从内核返回的时候buf中就被填充了文件的正确的属性信息，然后我们通过查看buf这种结构体变量的元素就可以得知这个文件的各种属性了。

(5)fstat和stat的区别是：stat是从文件名出发得到文件属性信息结构体，而fstat是从一个已经打开的文件fd出发得到一个文件的属性信息。所以用的时候如果文件没有打开（我们并不想打开文件操作而只是希望得到文件属性）那就用stat，如果文件已经被打开了然后要属性那就用fstat效率会更高（stat是从磁盘去读取静态文件的，而fstat是从内存读取动态文件的）。

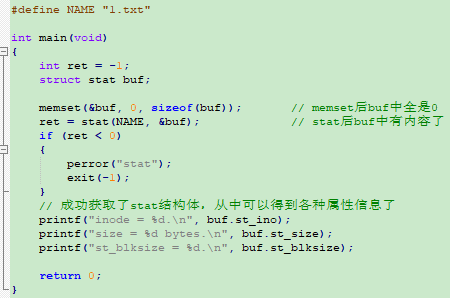
(6)lstat和stat/fstat的差别在于：对于符号链接文件，stat和fstat查阅的是符号链接文件指向的文件的属性，而lstat查阅的是符号链接文件本身的属性。

### 2.2.2. struct stat结构体简介

struct stat是内核定义的一个结构体，在<sys/stat.h>中声明，所以我们可以用。这个结构体中的所有元素加起来就是我们的文件属性信息。



### 2.2.3.写个程序来查看一些常见属性信息



## 2.3 stat函数的应用案例

### 2.3.1.用代码判断文件类型

(1)文件类型就是-、d、l …

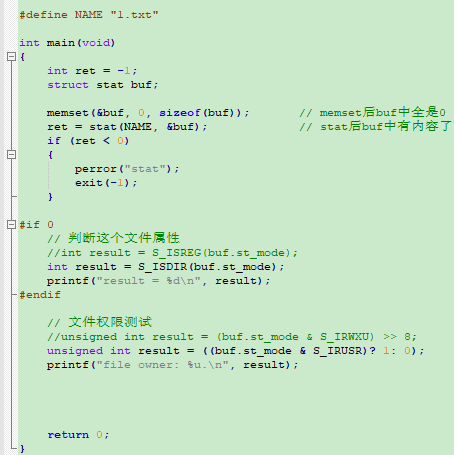
(2)文件属性中的文件类型标志在struct stat结构体的元素中，这个元素其实是一个按位来定义的一个位标志（有点类似于ARM CPU的CPSR寄存器的模式位定义）。这个东西有很多个标志位共同构成，记录了很多信息，如果要查找时按位&操作就知道结果了，但是因为这些位定义不容易记住，因此linux系统给大家事先定义好了很多宏来进行相应操作。

(3)譬如S\_ISREG宏返回值是1表示这个文件是一个普通文件，如果文件不是普通文件则返回值是0。

### 2.3.2.用代码判断文件权限设置

(1)st\_mode中除了记录了文件类型之外，还记录了一个重要信息：文件权限。

(2)linux并没有给文件权限测试提供宏操作（只给文件类型提供了宏操作），而只是提供了位掩码，所以我们只能用位掩码来自己判断是否具有相应权限。



## 2.4 文件权限管理1

### 2.4.1 st\_mode中记录的文件权限位

(1)st\_mode本质上是一个32位的数（类型就是unsinged int），这个数里的每一个位表示一个含义。

(2)文件类型和文件的权限都记录在st\_mode中。我们用的时候使用专门的掩码去取出相应的位即可得知相应的信息。

### 2.4.2. ls -l打印出的权限列表

(1)123456789一共9位，3个一组。第一组三个表示文件的属主（owner、user）对该文件的可读、可写、可执行权限；第2组3个位表示文件的属主所在的组（group）对该文件的权限；第3组3个位表示其他用户（others）对该文件的权限。

(2)属主就是这个文件属于谁，一般来说文件创建时属主就是创建这个文件的那个用户。但是我们一个文件创建之后还可以用chown命令去修改一个文件的属主，还可以用chgrp命令去修改一个文件所在的组。

### 2.4.3.文件操作时的权限检查规则

(1)一个程序a.out被执行，a.out中试图去操作一个文件1.txt，这时候如何判定a.out是否具有对1.txt的某种操作权限呢？

(2)判定方法是：首先1.txt具有9个权限位，规定了3种人（user、group、others）对该文件的操作权限。所以我们判定1.txt是否能被a.out来操作，关键先搞清楚a.out对1.txt到底算哪种人。准确的说是看a.out被谁执行，也就是当前程序（进程）是哪个用户的进程。

(3)刚才上面说的是我的分析，到底对不对还得验证。

## 2. 5文件权限管理2

### 2.5.1. access函数检查权限设置

(1)文本权限管控其实蛮复杂，一般很难很容易的确定对一个文件是否具有某种权限。设计优秀的软件应该是：在操作某个文件之前先判断当前是否有权限做这个操作，如果有再做如果没有则提供错误信息给用户。

(2)access函数可以测试得到当前执行程序的那个用户在当前那个环境下对目标文件是否具有某种操作权限。



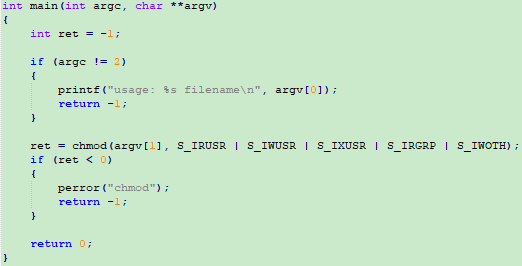
mode: F\_OK, R\_OK, W\_OK, and X\_OK

### 2.5.2. chmod/fchmod与权限修改

(1)chmod是一个linux命令，用来修改文件的各种权限属性。chmod命令只有root用户才有权利去执行修改。

(2)chmod命令其实内部是用linux的一个叫chmod的API实现的。

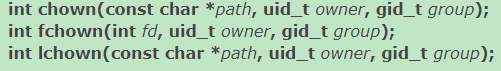




### 2.5.3. chown/fchown/lchown与属主修改

(1)linux中有个chown命令来修改文件属主

(2)chown命令是用chown API实现的



### 2.5.4. umask与文件权限掩码

(1)文件掩码是linux系统中维护的一个全局设置，umask的作用是用来设定我们系统中新创建的文件的默认权限的。

(2)umask命令就是用umask API实现的

## 2.6 读取目录文件

### 2.6.1. opendir与readdir函数

man 3

1. opendir打开一个目录后得到一个DIR类型的指针给readdir使用

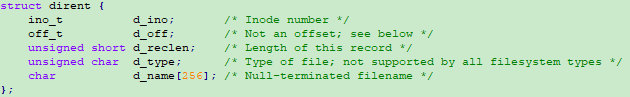




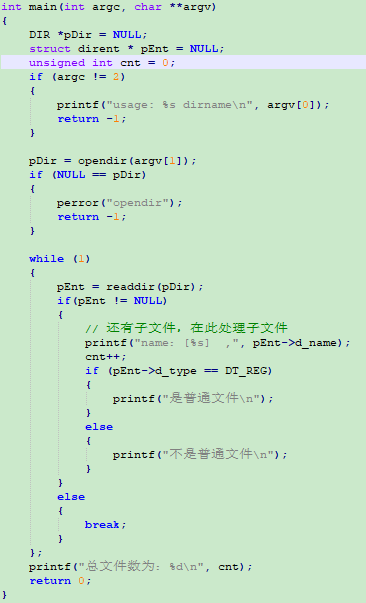
(2)readdir函数调用一次就会返回一个struct dirent类型的指针，这个指针指向一个结构体变量，这个结构体变量里面记录了一个目录项（所谓目录项就是目录中的一个子文件）。

(3)readdir调用一次只能读出一个目录项，要想读出目录中所有的目录项必须多次调用readdir函数。readdir函数内部户记住哪个目录项已经被读过了哪个还没读，所以多次调用后不会重复返回已经返回过的目录项。当readdir函数返回NULL时就表示目录中所有的目录项已经读完了。

### 2.6.2.dirent结构体



### 2.6.3.读取目录实战演练



### 2.6.4.可重入函数介绍

(1)有些函数是可重入的有些是不可重入的，具体概念可以去百度。

(2)readdir函数和我们前面接触的一些函数是不同的，首先readdir函数直接返回了一个结构体变量指针，因为readdir内部申请了内存并且给我们返回了地址。多次调用readdir其实readir内部并不会重复申请内存而是使用第一次调用readdir时分配的那个内存。这个设计方法是readdir不可重入的关键。所以不可重入函数被一个进程修改后会影响所有调用这个函数的进程。

(3)readdir在多次调用时是有关联的，这个关联也标明readdir函数是不可重入的。

(4)库函数中有一些函数当年刚开始提供时都是不可重入的，后来意识到这种方式不安全，所以重新封装了C库，提供了对应的可重复版本（一般是不可重入版本函数名\_r）

# 三、获取系统信息

## 3.1 关于时间的概念

### 3.1.1. GMT时间

(1)GMT是格林尼治时间，也就是格林尼治地区的当地之间。

(2)GMT时间的意义？用格林尼治的当地时间作为全球国际时间，用以描述全球性的事件的时间，方便大家记忆。

(3)一般为了方便，一个国家都统一使用一个当地时间。

### 3.1.2. UTC时间

(1)GMT时间是以前使用的，近些年来越来越多的使用UTC时间。

(2)关于北京时间，可以参考：<http://www.cnblogs.com/qiuyi21/archive/2008/03/04/1089456.html>

UTC ＋ 时区差 ＝ 本地时间

### 3.1.3.计算机中与时间有关的部件

(1)点时间和段时间。段时间=点时间-点时间

(2)定时器和实时时钟。定时器（timer）定的时间就是段时间，实时时钟（RTC）就是和点时间有关的一个器件。

## 3.2 Linux中的时间

### 3.2.1. jiffies的引入

(1)jiffies是Linux内核中的一个全局变量，这个变量用来记录以内核的节拍时间为单位时间长度的一个数值。

(2)内核配置的时候定义了一个节拍时间，实际上Linux内核的调度系统工作时就是以这个节拍时间为时间片的。

(3)jiffies变量开机时有一个基准值，然后内核每过一个节拍时间jiffies就会加1，然后到了系统的任意一个时间我们当前时间就被jiffies这个变量所标注。

### 3.2.2. Linux系统如何记录时间

(1)内核在开机启动的时候会读取RTC硬件获取一个时间作为初始基准时间，这个基准时间对应一个jiffies值（这个基准时间换算成jiffies值的方法是：用这个时间减去1970-01-01 00:00:00 +0000(UTC)，然后把这个时间段换算成jiffies数值），这个jiffies值作为我们开机时的基准jiffies值存在。然后系统运行时每个时钟节拍的末尾都会给jiffies这个全局变量加1，因此操作系统就使用jiffies这个全局变量记录了下来当前的时间。当我们需要当前时间点时，就用jiffies这个时间点去计算（计算方法就是先把这个jiffies值对应的时间段算出来，然后加上1970-01-01 00:00:00 +0000(UTC)即可得到这个时间点）

(2)其实操作系统只在开机时读一次RTC，整个系统运行过程中RTC是无作用的。RTC的真正作用其实是在OS的2次开机之间进行时间的保存。

(3)理解时一定要点时间和段时间结合起来理解。jiffies这个变量记录的其实是段时间（其实就是当前时间和1970-01-01 00:00:00 +0000(UTC)这个时间的差值）

(4)一个时间节拍的时间取决于操作系统的配置，现代Linux系统一般是10ms或者1ms。这个时间其实就是调度时间，在内核中用HZ来记录和表示。如果HZ定义成1000那么时钟节拍就是1/HZ，也就是1ms。这些在学习驱动时会用到。

### 3.2.3. Linux中时间相关的系统调用

(1)常用的时间相关的API和C库函数有9个： time/ctime/localtime/gmtime/mktime/asctime/strftime/gettimeofday/settimeofday

(2)time系统调用返回当前时间以秒为单位的距离1970-01-01 00:00:00 +0000(UTC)过去的秒数。这个time内部就是用jiffies换算得到的秒数。其他函数基本都是围绕着time来工作的。

(3)gmtime和localtime会把time得到的秒数变成一个struct tm结构体表示的时间。区别是gmtime得到的是国际时间，而localtime得到的是本地（指的是你运行localtime函数的程序所在的计算机所设置的时区对应的本地时间）时间。mktime用来完成相反方向的转换（struct tm到time\_t）

(4)如果从struct tm出发想得到字符串格式的时间，可以用asctime或者strftime都可以。（如果从time\_t出发想得到字符串格式的时间用ctime即可）

(5)gettimeofday返回的时间是由struct timeval和struct timezone这两个结构体来共同表示的，其中timeval表示时间，而timezone表示时区。settimeofday是用来设置当前的时间和时区的。

(6)总结：不管用哪个系统调用，最终得到的时间本质上都是一个时间（这个时间最终都是从kernel中记录的jiffies中计算得来的），只不过不同的函数返回的时间的格式不同，精度不同。