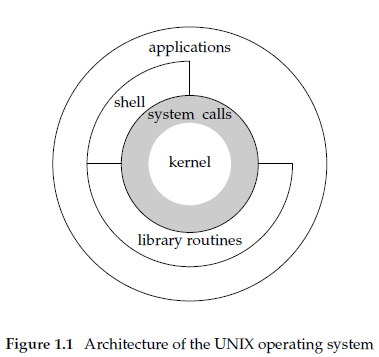
#### Overview（系统调用包裹整个kernel）

This book describes the programming interface to the Unix system—the system call interface and many of the functions provided in the standard C library.（安装了相关模块的Unix系统都会提供的系统调用）

（先熟练掌握这些，然后再研究驱动以及系统移植相关的🡪买一本嵌入式系统开发教程，了解自定制文件系统相关的东西）

Like most operating systems, Unix provides numerous services to the programs that are running — open a file, read a file, start a new program, allocate a region of memory, get the current time-of-day, and so on. This has been termed the system call interface. Additionally, the standard C library provides numerous functions that are used by almost every C program (format a variable's value for output, compare two strings, etc.).



The shell is a special application that provides an interface for running other applications.当初在学习汇编（有时间再看一遍汇编，多重复才是学习的关键）的时候接触过系统调用，可以理解为给OS输入参数，让OS内核去执行并返回结果（把参数放到某个特定区域，比如某个寄存器，然后使用转移指令，代码跳转到另一个地方执行，另一个地方的代码便是系统区，最后再到指定区域取回返回值）。

目录也是一种文件，其内容是文件的索引（每种文件都有文件类型属性，比如普通文件，链接文件，目录文件，有些命令会检查文件的类型属性，以决定是否执行下一步）；A pathname that begins with a slash is called an absolute pathname; otherwise, it's called a relative pathname. Relative pathnames refer to files relative to the current directory.

1. return返回函数值，是关键字； exit 是一个函数；

2. return是语言级别的，它表示了调用堆栈的返回；而exit是系统调用级别的，它表示了一个进程的结束；

3. return是函数的退出(返回)；exit是进程的退出；

4. return是C语言提供的，exit是操作系统提供的（或者函数库中给出的）；

5. return用于结束一个函数的执行，将函数的执行信息传出个其他调用函数使用；exit函数是退出应用程序，删除进程使用的内存空间，并将应用程序的一个状态返回给OS，这个状态标识了应用程序的一些运行信息，这个信息和机器和操作系统有关，一般是 0 为正常退出，非0 为非正常退出；

6. 非主函数中调用return和exit效果很明显，但是在main函数中调用return和exit的现象就很模糊，多数情况下现象都是一致的；

7.在C++中还会涉及到构造函数和析构函数的行为；

###### 一切皆文件（open，close，read，write）

**File descriptors** are normally small non-negative integers that the kernel uses to identify the files accessed by a process. Whenever it opens an existing file or creates a new file, the kernel returns a file descriptor that we use when we want to read or write the file.

**Standard Input, Standard Output, and Standard Error**

By convention, all shells open three descriptors whenever a new program is run: standard input, standard output, and standard error. If nothing special is done, as in the simple command

ls

then all three are connected to the terminal. Most shells provide a way to redirect any or all of these three descriptors to any file. For example：

ls > file.list（其他有输出的命令也可以这样做，这是shell的行为，不是命令本身有对>的解析，如果file.list文件不存在，那么shell会默认创建它，这也是shell的行为）

executes the ls command with its standard output redirected to the file named file.list.（printf应该不用打开标准输出并关闭，而是直接依文件描述符往标志IO文件中输出，scanf也是同理，标准输入输出文件描述符一般是固定的即0、1和2这些：一切皆文件\_&\_Linux驱动开发\_&\_Linux编程思维）

The constants STDIN\_FILENO and STDOUT\_FILENO are defined in <unistd.h> and specify the file descriptors for standard input and standard output. These values are 0 and 1, respectively, as required by POSIX.1, but we'll use the names for readability

stdin采用行缓冲的方式即当输入输出过程遇到换行符\n（并自动补上EOF）或者当分配缓冲区已满时，才开始执行 I\O 操作（通知读取缓冲区的函数可以执行读取操作了，比如修改某个标志变量，读取它的函数阻塞判断这个标志标量）；一般涉及终端的读写操作如stdin与stdout使用这种缓冲方式；

终端输入回显是如何操作的？ read函数返回看“文件通知”，所以read阻塞与否取决于文件属性行为，如果文件是无论如何，当下立即通知，则read是非阻塞的，如果文件偏移量到达末尾或者可读字节数达到read参数要求后才通知，那么read是阻塞的，修改文件阻塞属性可以通过open函数参数指定



EOF是end of file的意思，它是一个符号常量（宏），一般是一个负值；当读到文件末尾时会返回这个值，而实际上是没有文件结束符这个东西的，所谓的CTRL + Z（Windows，ASCII码是4）或者CTRL + D（Unix，ASCII码是26）只是告诉终端输入结束了，也可通过配置修改为输入其他键告诉输入结束了；

Unix systems in the shell conventionally use control-D to tell an application that an end of input (file) has been reached, but the control-D is not stored in the file. In C, EOF is purposely made -1 to indicate that it is not a valid character. Standard I/O returns EOF when an end-of-file condition is detected — not a special character.操作系统有办法通过其他方式判断文件是否到末尾了，比如文件大小，而不必要在文件末尾加上一个特殊字符（有一些老系统会这么做）；

对于输入型函数，返回EOF意思是发生错误；

We'll see how the kernel uses the user ID to check whether we have the appropriate permissions to perform certain operations. We call the user whose user ID is 0 either root or the superuser. The entry in the password file normally has a login name of root, and we refer to the special privileges of this user as superuser privileges.

一般文件在存储的时候，所有者属性写的是ID（用户ID和组ID），如果ls命令要显示所有者名字，组名字这些，那么程序要去读取/etc/passwd文件内容进行对应，然后输出，文件属性当中不会存储名字（字符串），否则会带来很多麻烦（更多的存储空间，在进行权限check的时候比对时间更长）；

The OS kernel has a data structure called a process control block for each process running which has data about that process. This can be looked up by the process id (PID) and （this）included a table of signal actions and pending signals.（长句翻译）。

先这么理解：传送信号给某个线程是内核负责的（比如系统调用的方式传送信号），然后再立即理解下面的话，When a signal is sent to a process the OS kernel will look up that process's process control block and examines the signal action table to locate the action for the particular signal being sent. If the signal action value is SIG\_IGN then the new signal is forgotten about by the kernel. If the signal action value is SIG\_DFL then the kernel looks up the default signal handling action for that signal in another table and preforms that action. If the values are anything else then that is assumed to be a function address within the process that the signal is being sent to which should be called. The values for SIG\_IGN and SIG\_DFL are numbers cast to function pointers whose values are not valid addresses within a process's address space (such as 0 and 1, which are both in page 0, which is never mapped into a process).

If a signal handling function were registered by the process (the signal action value was neither SIG\_IGN or SIG\_DFL) then an entry in the pending signal table is made for that signal and that process is marked as ready to RUN (it may have been waiting on something, like data to become available for a call to read, waiting for a signal, or several other things).

Now the next time that the process is run the OS kernel will first add some data to the stack and changes the instruction pointer for that process so that it looks almost like the process itself has just called the signal handler. This is not entirely correct and actually deviates enough from what actually happens that I'll talk about it more in a little bit.

The signal handler function can do whatever it does (it is part of the process that it was called on behalf of, so it was written with knowledge about what that program should do with that signal). When the signal handler returns then the regular code for the process begins executing again. (again, not accurate, but more on that next)

Ok, the above should have given you a pretty good idea of how signals are delivered to a process. I think that this pretty good idea version is needed before you can grasp the full idea, which includes some more complicated stuff.

Very often the OS kernel needs to know when a signal handler returns. This is because signal handlers take an argument (which may require stack space), you can block the same signal from being delivered twice during the execution of the signal handler, and/or have system calls restarted after a signal is delivered. To accomplish this a little bit more than stack and instruction pointer changes.

What has to happen is that the kernel needs to make the process tell it that it has finished executing the signal handler function. This may be done by mapping a section of RAM into the process's address space which contains code to make this system call and making the return address for the signal handler function (the top value on the stack when this function started running) be the address of this code. I think that this is how it is done in Linux (at least newer versions). Another way to accomplish this (I don't know if this is done, but it could be) would be do make the return address for the signal handler function be an invalid address (such as NULL) which would cause an interrupt on most systems, which would give the OS kernel control again. It doesn't matter a whole lot how this happens, but the kernel has to get control again to fix up the stack and know that the signal handler has completed.

WHILE LOOKING INTO ANOTHER QUESTION I LEARNED

that the Linux kernel does map a page into the process for this, but that the actual system call for registering signal handlers (what sigaction calls ) takes a parameter sa\_restore parameter, which is an address that should be used as the return address from the signal handler, and the kernel just makes sure that it is put there. The code at this address issues the I'm done system call (sigreturn)and the kernel knows that the signal handler has finished.

signal generation

I'm mostly assuming that you know how signals are generated in the first place. The OS can generate them on behalf of a process due to something happening, like a timer expiring, a child process dying, accessing memory that it should not be accessing, or issuing an instruction that it should not (either an instruction that does not exist or one that is privileged), or many other things. The timer case is functionally a little different from the others because it may occur when the process is not running, and so is more like the signals sent with the kill system call. For the non-timer related signals sent on behalf of the current process these are generated when an interrupt occurs because the current process is doing something wrong. This interrupt gives the kernel control (just like a system call) and the kernel generates the signal to be delivered to the current process.

###### system call

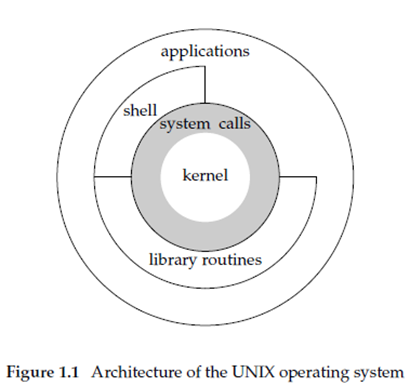
The system call interface's definition is in the C language, no matter which implementation technique is actually used on any given system to invoke a system call.（远古操作系统接口是汇编的形式）

The technique used on UNIX systems is for each system call to have a function of the same name in the standard C library. The user process calls this function, using the standard C calling sequence. This function then invokes the appropriate kernel service, using whatever technique is required on the system. For example, the function may put one or more of the C arguments into general registers and then execute some machine instruction that generates a software interrupt in the kernel. For our purposes, we can consider the system calls to be C functions. From our(programmer) perspective in this text, both system calls and library functions appear as normal C functions.

printf的底层实现用到了系统调用write函数；

The UNIX System, in contrast, provides a single system call that returns the number of seconds since the Epoch: midnight, January 1, 1970, Coordinated Universal Time. Any interpretation of this value, such as converting it to a human-readable time and date using the local time zone, is left to the user process. The standard C library provides routines to handle most cases. These library routines handle such details as the various algorithms for daylight saving time.

An application can either make a system call or call a library routine. Also realize that many library routines invoke a system call.



标准输入输出（the standard I/O functions）一般就是系统调用（unbuffered I/O functions）的封装扩展；

1. Regular file. The most common type of file, which contains data of some form. There is no distinction to the UNIX kernel whether this data is text or binary. Any interpretation of the contents of a regular file is left to the application processing the file.

2. Directory file. A file that contains the names of other files and pointers to information on these files. Any process that has read permission for a directory file can read the contents of the directory, but only the kernel can write directly to a directory file. Processes must use the functions described in this chapter to make changes to a directory.

3. Block special file. A type of file providing buffered I/O access in fixed-size units to devices such as disk drives.

4. Character special file. A type of file providing unbuffered I/O access in variable-sized units to devices. All devices on a system are either block special files or character special files.

5. FIFO. A type of file used for communication between processes. It’s sometimes called a named pipe.

6. Socket. A type of file used for network communication between processes. A socket can also be used for non-network communication between processes on a single host.

7. Symbolic link. A type of file that points to another file.

S\_ISREG() regular file

S\_ISDIR() directory file

S\_ISCHR() character special file

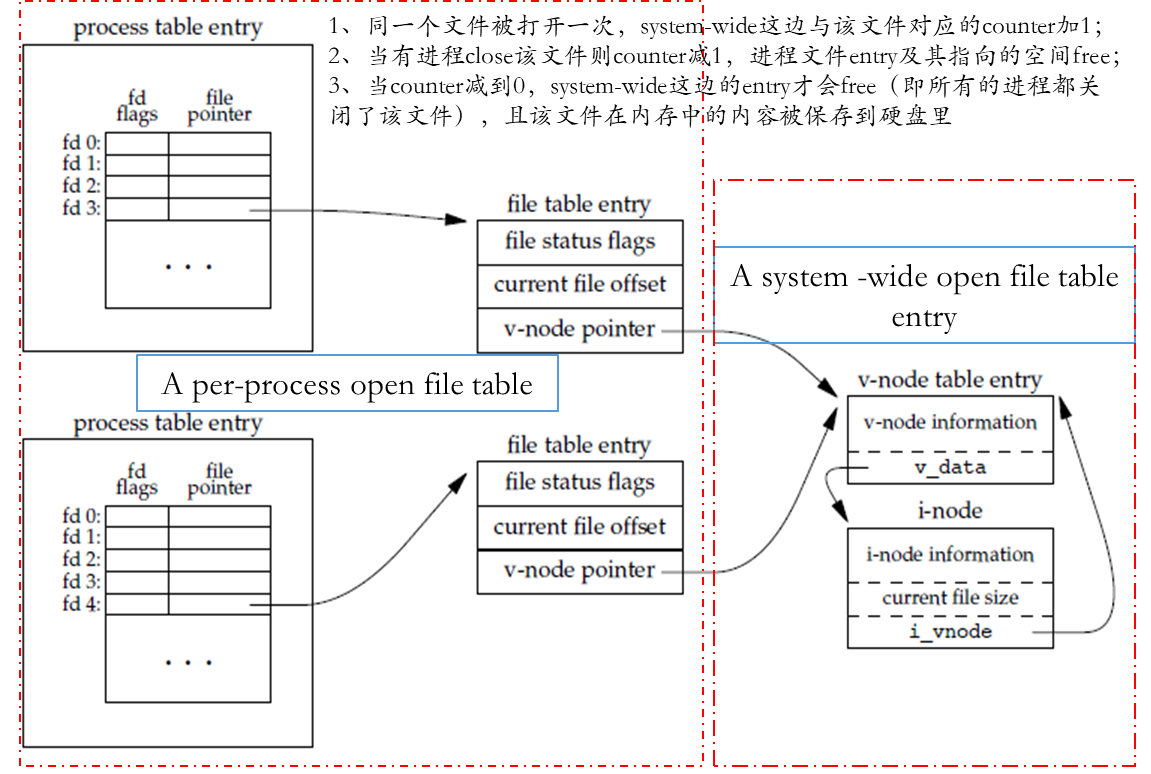
S\_ISBLK() block special file

S\_ISFIFO() pipe or FIFO

S\_ISLNK() symbolic link

S\_ISSOCK() socket

“那些文件操作函数以及相关概念对应这些文件类型是否有意义，比如socket文件类型有没有current file offset这一说，open函数能否打开socket（应该不能，因为open函数需要输入文件路径参数字符串；而其实在创建socket时已经返回了一个文件描述符）”；



The i-node contains all the information about the file: the file type, the file's access permission bits, the size of the file, pointers to the file's data blocks, and so on. Most of the information in the stat structure is obtained from the i-node. Only two items of interest are stored in the directory entry: the filename and the i-node number.

进程有各种ID属性（5种：），而文件有owner属性（2种：所有者以及所属组）；set the effective user ID of the process to be the owner of the file；是to be不是to，是将effective user ID设置成文件所有者，而不是将effective user ID的内容置到文件所有者；

进程对应的是５种ID属性，文件对应的是２种owner属性（文件≠进程）；一个进程的real user ID和real group ID从password文件中读取（password文件就是定义了在该系统上注册的所有用户的信息，比如每一个用户的密码，名称，登录之后可以取得的shell等等）；然后effective user ID，effective group ID和supplementary group ID也是进程的属性，用来判断进程的操作权限，一开始登录时前两者的值从real user ID和real group ID而来；通常情况下它们也一直相等；

学到现在，已知有一种情况，它们会不相等：**文件是可执行的**，然后文件的控制结构块相关标志置位，可以使得文件在执行时（变成进程），该进程的effective user ID和effective group ID的值从文件（变成进程对应的同一个文件）的2个owner属性值而来（鉴权用的effective ID**来源**），从而让进程的权限改变，因为在执行相关权限操作（主要是文件操作）时，用来进行鉴权的是effective user ID，effective group ID以及supplementary group ID；如果在该文件的创建者是超级管理，那么在相关位置位的情况下，文件在执行时可以获得超级权限，该进程代码定义的（超级管理者需要对该进程代码所作的操作负责）所有操作是具备超级权限的（鉴权用的是超级管理者ID）；比如修改用户密码的进程（每个用户可以修改自己的密码，但不能访问password文件）；

其他用户也可以通过这种方式进行类似处理（前提是需要对自己创建的可执行文件里面定义的操作负责）；

清晰一个概念，对于不同用户登录时，Linux上的所有文件都是“可见的”，比如每一个用户都可以看到/etc/password（通过cd和ls等操作），Linux不会根据仅展现当前用户相关的文件，而隐藏其他所有用户的文件；Linux的做法是可以通过文件权限不让某个用户查看某个文件夹文件从而达到隐藏的目的，但不是自己之前理解的那种全部的，屏蔽式隐藏（概念一定要清晰过来）；

再举个例子加深进程权限的理解，cat进程real user ID和real group ID从password文件中取得，但是其对应的可执行文件相关位没有置位，所以它的鉴权用的ID属性和real ID一样（好好嚼一嚼第3、4章，不着急，但也绝不浪费时间）；

进程打开文件的前提是拥有包含它的各层级所有目录的执行权限；Note that read permission for a directory and execute permission for a directory mean different things. Read permission lets us read the directory, obtaining a list of all the filenames in the directory. Execute permission lets us pass through the directory when it is a component of a pathname that we are trying to access. (We need to search the directory to look for a specific filename.)

The user ID of a new file is set to the effective user ID of the process.是不是就说明超级管理者创建的进程文件被其他用户执行之后，创建的新文件owner就是超级管理者了？？？当然不是，这个进程文件里面根本没有创建文件的操作（所以说超级管理者需要对自己创建的相关位置位的进程文件里面定义的内容负责）；

但是以上情况会造成进程拥有者切换（非当前登录用户），在某些情况下，进程需要切换回自己的用户权限即real ID，Unix提供了这种切换的相关函数，使得进程在进行文件操作时用real ID进行鉴权（在进程种调用这几个函数）；

已打开的文件（文件描述符），未打开的文件（文件路径名）；

struct stat {

mode\_t st\_mode; /\* file type（一共有7种） & mode (permissions) \*//&/

ino\_t st\_ino; /\* i-node number (serial number) \*/

dev\_t st\_dev; /\* device number (file system) \*/

dev\_t st\_rdev; /\* device number for special files \*/

nlink\_t st\_nlink; /\* number of links \*/

uid\_t st\_uid; /\* user ID of owner \*//&/

gid\_t st\_gid; /\* group ID of owner \*//&/

off\_t st\_size; /\* size in bytes, for regular files \*//&/

struct timespec st\_atim; /\* time of last access \*//&/

struct timespec st\_mtim; /\* time of last modification \*//&/

struct timespec st\_ctim; /\* time of last file status change \*//&/

blksize\_t st\_blksize; /\* best I/O block size \*/

blkcnt\_t st\_blocks; /\* number of disk blocks allocated \*/

};

1、st\_mode是通过按位编码的方式记录文件类型和文件权限（可读可写可执行…）的，可通过设置按位操作的宏判断文件类型，文件权限这些；S\_ISREG(buf.st\_mode)等于1则是regular file，类似的还有另外七种宏；

这个i-node是什么东西？（文件系统是怎么组织并记录文件的？）；

2、These two bits in the file's mode word (**st\_mode**) are called the set-user-ID bit and the set-group-ID bit.这两位就是决定是否将可执行文件的effective ID变成文件的owner（可以通过宏S\_ISUID and S\_ISGID查看这两位的情况）；

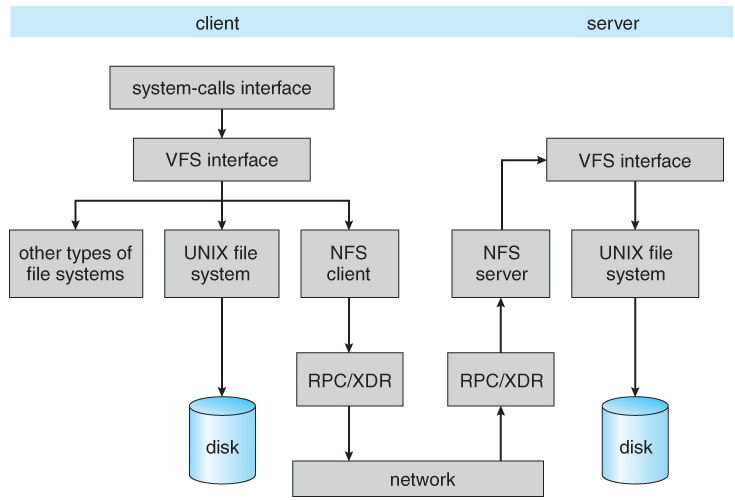
3、文件权限可读可写可执行与修改文件的stat有什么关系？对文件有可写权限是否意味着可以修改文件的stat？文件的stat是应该不是和文件内容一起保存在磁盘中；

4、先不管关于3的疑惑，留个flag；函数stat，fstat，lstat和fstatat的作用是把文件的stat放到参数buf中，这本身不需要任何文件权限（但如果涉及到路径名为参数的，则需要文件所在目录的执行权限，即搜索权限；文件描述符则不需要，因为描述符代表文件已经在进程中的其他地方打开了，进程已经具备对目录文件的执行权限）；改变参数buf的内容不等于改变文件的stat；至于Unix如何维护文件的stat，先不管（应该不是和文件内容放在一起，而是集成在文件系统数据架构中）；

5、修改文件的stat应该是通过系统调用；

6、所有7大类型的文件都有上述的stat，只不过各个类型的stat在一些成员上有差异；

7、挂载本质上是挂载一个“file system”，Linux的通用函数（如open，read，write以及close）可以识别常见的file system；



Schematic view of the NFS architecture（Network File System）

文件权限也encode在st\_mode中，包括以及涉及的宏有

S\_IRUSR user-read

S\_IWUSR user-write

S\_IXUSR user-execute

S\_IRGRP group-read

S\_IWGRP group-write

S\_IXGRP group-execute

S\_IROTH other-read

S\_IWOTH other-write

S\_IXOTH other-execute

###### 文件权限规则如下

1. The first rule is that whenever we want to open any type of file by name, we must have execute permission in each directory mentioned in the name, including the current directory（每个层级的目录文件都需要）, if it is implied. This is why the execute permission bit for a directory is often called the search bit.如果已经在当前目录下，并是直接写文件名的形式，那么只需要当前目录的执行权限即可（上面层级的执行权限可无视，当然如果在当前目录下仍然以绝对路径的方式，那么还是需要上面层级目录文件的执行权限，相对于非当前目录的其他相对目录也是从一开始相对的那个目录算起，一般那个目录是用文件描述符指定的，即它已经被进程打开了）；
   1. Read permission lets us read the directory, obtaining a list of all the filenames in the directory. Execute permission lets us pass through the directory when it is a component of a pathname that we are trying to access. (We need to search the directory to look for a specific filename.)即我们需要对目录下的文件进行读写或执行的操作（access）；
2. 当进程使用open函数打开文件时，open函数指定的参数（如O\_EDONLY，O\_RDWR，O\_WRONLY，O\_TRUNC）与文件的权限相关；delete文件，create文件也是如此：
   1. The read permission for a file determines whether we can open an existing file for reading: the O\_RDONLY and O\_RDWR flags for the open function.
   2. The write permission for a file determines whether we can open an existing file for writing: the O\_WRONLY and O\_RDWR flags for the open function.
   3. We must have write permission for a file to specify the O\_TRUNC flag in the open function.
   4. We cannot create a new file in a directory unless we have write permission and execute permission in the directory.
   5. To delete an existing file, we need write permission and execute permission in the directory containing the file. We do not need read permission or write permission for the file itself.
   6. Execute permission for a file must be on if we want to execute the file using any of the seven exec functions (Section 8.10). The file also has to be a regular file.

###### access（open，delete，create等）文件时鉴权规则如下（内核执行access时按顺序判断以下4步）：

1. If the effective user ID of the process is 0 (the superuser), access is allowed. This gives the superuser free rein throughout the entire file system.（直接bypass）
2. If the effective user ID of the process equals the owner ID of the file (i.e., the process owns the file), access is allowed if the appropriate user access permission bit is set. Otherwise, permission is denied.（文件的user权限组）
3. If the effective group ID of the process or one of the supplementary group IDs of the process equals the group ID of the file, access is allowed if the appropriate group access permission bit is set. Otherwise, permission is denied.（文件的group权限组）
4. If the appropriate other access permission bit is set, access is allowed.Otherwise, permission is denied.（文件的other权限组）

###### 新文件（包括目录文件）的ownership（涉及到文件stat结构st\_mode成员的修改\*）

The user ID of a new file is set to the effective user ID of the process.然后在Linux中，group ID由以下两者之一：

1. The group ID of a new file can be the effective group ID of the process.
2. The group ID of a new file can be the group ID of the directory in which the file is being created.（set-group-ID bit要置位）

access函数和faccessat函数用来检测如果以real ID（user和group）鉴权，进程是否有相关权限（检测步骤和上面的４步一样，只不过effective ID换成real ID）；在某些情况下会用到（暂时想不到什么情况）；

faccessat通过指定标志还可以检查effective ID的鉴权；

int access(const char \*pathname, int mode);

int faccessat(int fd, const char \*pathname, int mode, int flag);

mode Description

R\_OK test for read permission

W\_OK test for write permission

X\_OK test for execute permission

###### 新文件（包括目录文件）的permission（涉及到stat结构st\_mode成员的修改）

umask函数和open函数（参数指定）；

###### 已存在文件的stat修改（同样也是st\_mode的ownership以及permission）

To change the permission bits of a file, the effective user ID of the process must be equal to the owner ID of the file, or the process must have superuser permissions.（effective group ID不行，这和是否对文件有读写执行没有关系，它也不会经过4步鉴权，只判断effective ID和owner ID）；

进程改变permission涉及到的函数是chmod，fchmod以及fchmodat；包括修改set-user-ID bit以及set-group-ID bit（就是决定可执行文件变成进程时，effective ID来源的两个控制位）

###### 文件系统

“文件系统的实现是关于硬盘的数据管理机制（应该是一个卷/分区经过格式化后，再设置一个文件系统，然后挂载到某个目录下，Linux用户便可以访问了）”

The logical file system deals with all of the meta data associated with a file ( UID, GID, mode, dates, etc ), i.e. everything about the file except the data itself. This level manages the directory structure and the mapping of file names to file control blocks, FCBs, which contain all of the meta data as well as block number information for finding the data on the disk.

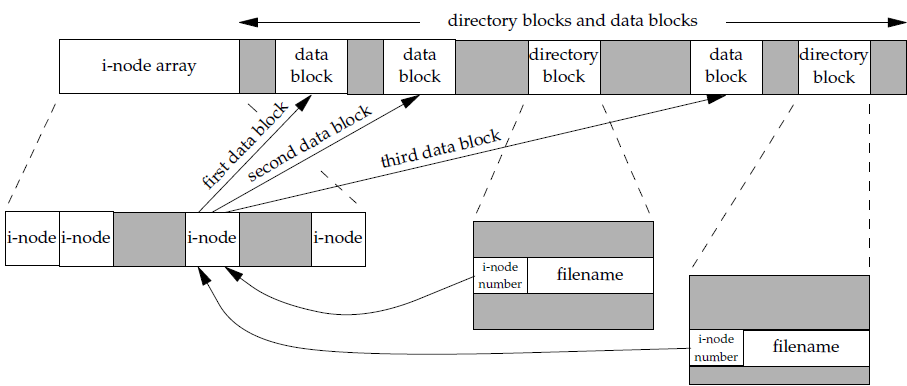
Linux系统中每个文件对应一个inode，在一个文件系统中，所有文件的inode存储在inode区里，每个inode占用固定大小的空间（文件系统是管理一片硬盘存储区的管理机制）；

File systems store several important data structures on the disk（上面提到的那些进程表项是存储在内存中）:

A boot-control block, ( per volume ) a.k.a. the boot block in UNIX or the partition boot sector in Windows contains information about how to boot the system off of this disk. This will generally be the first sector of the volume if there is a bootable system loaded on that volume, or the block will be left vacant otherwise.

A volume control block, ( per volume ) a.k.a. the master file table in UNIX or the superblock in Windows, which contains information such as the partition table, number of blocks on each filesystem, and pointers to free blocks and free FCB blocks.

A directory structure ( per file system ), containing file names and pointers to corresponding FCBs. UNIX uses inode numbers, and NTFS uses a master file table.



The File Control Block, FCB, ( per file ) containing details about ownership, size, permissions, dates, etc. UNIX stores this information in inodes, and NTFS in the master file table as a relational database structure.

每个inode在硬盘所占的空间存储有file permissions，file dates（create，access，write），file owner，file size以及file data blocks or pointers to file data blocks等等这些；

There are also several key data structures stored in memory:

An in-memory mount table.

An in-memory directory cache of recently accessed directory information.

A system-wide open file table, containing a copy of the FCB for every currently open file in the system, as well as some other related information.

A per-process open file table, containing a pointer to the system open file table as well as some other information. ( For example the current file position pointer may be either here or in the system file table, depending on the implementation and whether the file is being shared or not. )

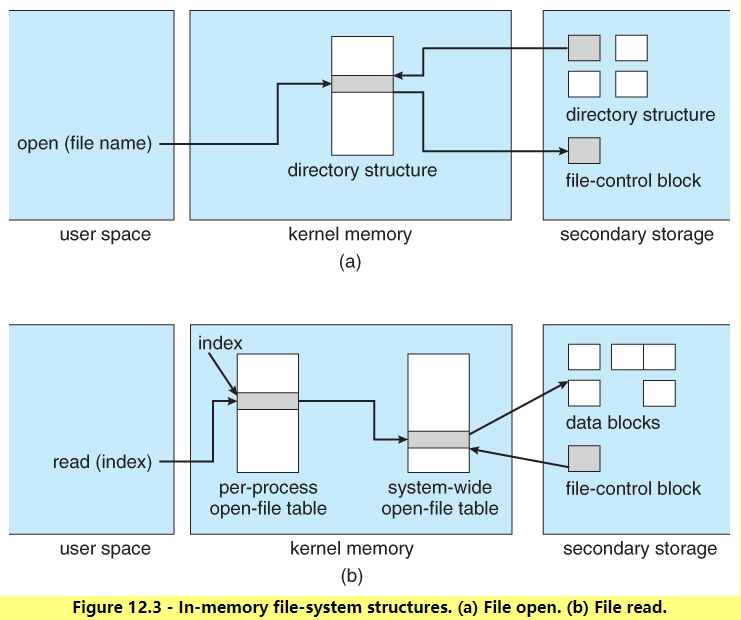
Figure 12.3 illustrates some of the interactions of file system components when files are created and/or used:

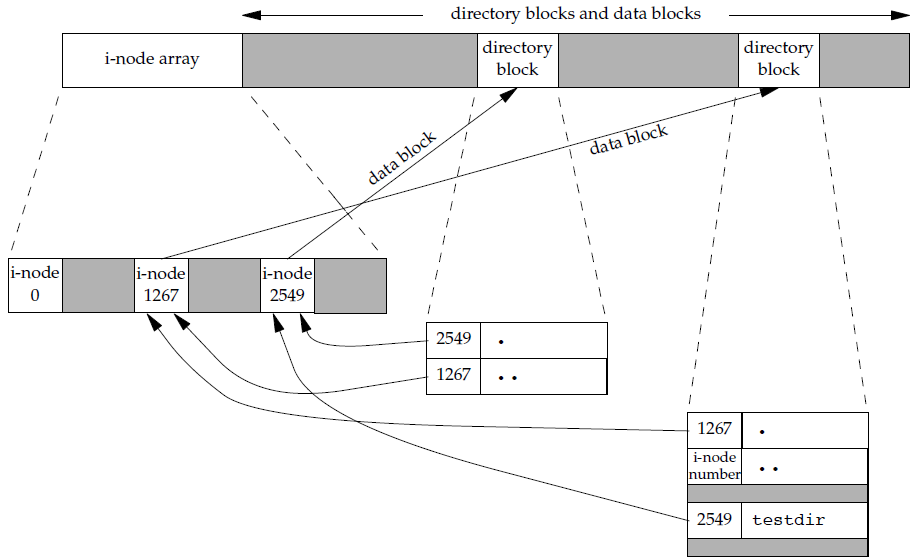
When a new file is created, a new FCB is allocated and filled out with important information regarding the new file. The appropriate directory is modified with the new file name and FCB information.

When a file is accessed during a program, the open( ) system call reads in the FCB information from disk, and stores it in the system-wide open file table. An entry is added to the per-process open file table referencing the system-wide table, and an index into the per-process table is returned by the open( ) system call. UNIX refers to this index as a file descriptor, and Windows refers to it as a file handle.（文件描述符是进程内文件entry的索引，不是系统内文件entry的索引，这里结合上面那张图看很容易理解）

If another process already has a file open when a new request comes in for the same file, and it is sharable, then a counter in the system-wide table is incremented and the per-process table is adjusted to point to the existing entry in the system-wide table.（另一个进程打开一个其他进程已经打开的文件，系统会再创建…还是看上面那张图，很容易理解）

When a file is closed, the per-process table entry is freed, and the counter in the system-wide table is decremented. If that counter reaches zero, then the system wide table is also freed. Any data currently stored in memory cache for this file is written out to disk if necessary.





目录文件的data block就是directory block（应该不是，这张图的directory block和上一张图的directory block应该不是同一个概念，需要再细看系统调取文件inode的过程是怎样的？它是如何从directory block当中索引entry的？）；directory block是如何组织的？

#### 其他

Linux终端打开后，shell程序也默认打开；终端提供输入和输出的方式，shell是处理输入输出的程序；

#### C语言标准IO库

系统调用IO让用户自己决定读取的buf大小；标准IO库关注“流（stream）”的概念：打开文件=建立一个文件流（字符流，一个字符可能是一个字节或者多个字节）；

stream's orientation determines whether the characters that are read and written are single byte or multibyte. Initially, when a stream is created, it has no orientation. If a multibyte I/O function (see <wchar.h>) is used on a stream without orientation, the stream's orientation is set to wide oriented. If a byte I/O function is used on a stream without orientation, the stream's orientation is set to byte oriented. Only two functions can change the orientation once set. The freopen function (discussed shortly) will clear a stream's orientation; the fwide function can be used to set a stream's orientation.

3个流已经被定义好了（利用默认已经打开的3个文件描述符）：标准输出流、标准输入流和标准错误输出流；这3个流对应stdin，stdout以及stderr（3个都是FILE\*类型），定义在头文件stdio.h中；

unbuffered IO用到的buf是程序自己管理的数据区，而buffered IO（标准IO库）帮程序管理buf，程序从这个buf里面读取数据到自己的数据区；

"that requires data to be requested from the kernel"的data to be requested

为什么需要buffer（缓冲）？（先忽略kernel buffer）

从缓冲区里读数据显然要比直接从硬盘中读数据要快，因为buffer就在内存中；

往缓冲区中写数据显然要比直接往硬盘里写数据要快，因为buffer就在内存中；

（当涉及到多次读写小规模数据时，有buffer显然更有优势：直接往buffer里面读写）

学习类似于看“绝命毒师”，看一个小片段顺便把整块内容看完（带着思考，结合新学的知识，悟出新理解或者更清晰认识其本质，记得更牢，用的更熟练，前提是：先全面而细致地看完第一遍）

**问题1**：不同orientation的文件IO函数决定了文件流的不同orientation，而文件流的orientation只能通过fwide函数和freopen函数修改，那么byte orientation的IO函数操作了文件流之后，multibyte orientation的IO函数能否可以操作同一个文件流？orientation中的multibyte指的不是固定的多字节，而是任意的多字节？？？

**问题2**：buffering与具体的IO操作函数的联系？比如当开始调用fgetc函数时，buffering是如何发生的？

问题2的解答：全buffer好理解，所有的读写都是写到buffer当中，而实际的IO操作发生与否取决于buffer是否满了，比如buffer满了，管理buffer的标准库函数将数据写到磁盘中（调用write），如果没有满，则进程写的字节写到buffet中放着不动；

如果是line buffer，然后输入一行，那么实际的IO操作会跟着发生，但第一次输入若干个字符（没有加上换行符，则字节留在buffer中，实际的IO没有发生），第二次输入…第n次输入若干个字符并加上newline character，此时实际IO发生（line buffer不是说每次输入都加newline character，可以在最后一次输入才加newline character，但是对于terminal device，每次输入都加上了newline character，这样等于无buffer？？？）；

Second, whenever input is requested through the standard I/O library from either (a) an unbuffered stream or (b) a line-buffered stream (that requires data to be requested from the kernel), all line-buffered output streams are flushed. The reason for the qualifier on (b) is that the requested data may already be in the buffer, which doesn't require data to be read from the kernel. Obviously, any input from an unbuffered stream, item (a), requires data to be obtained from the kernel.这段话的意思是文件B（或其他地方）从文件A中请求输入（以只读的方式打开文件A），文件A的流对应的是输出，此时文件A的output stream flush；

当具体使用标准IO函数时，比如fgets，fgetc，fputc，不用考虑buffer的问题（从buffer中按IO参数要求读取即可）；