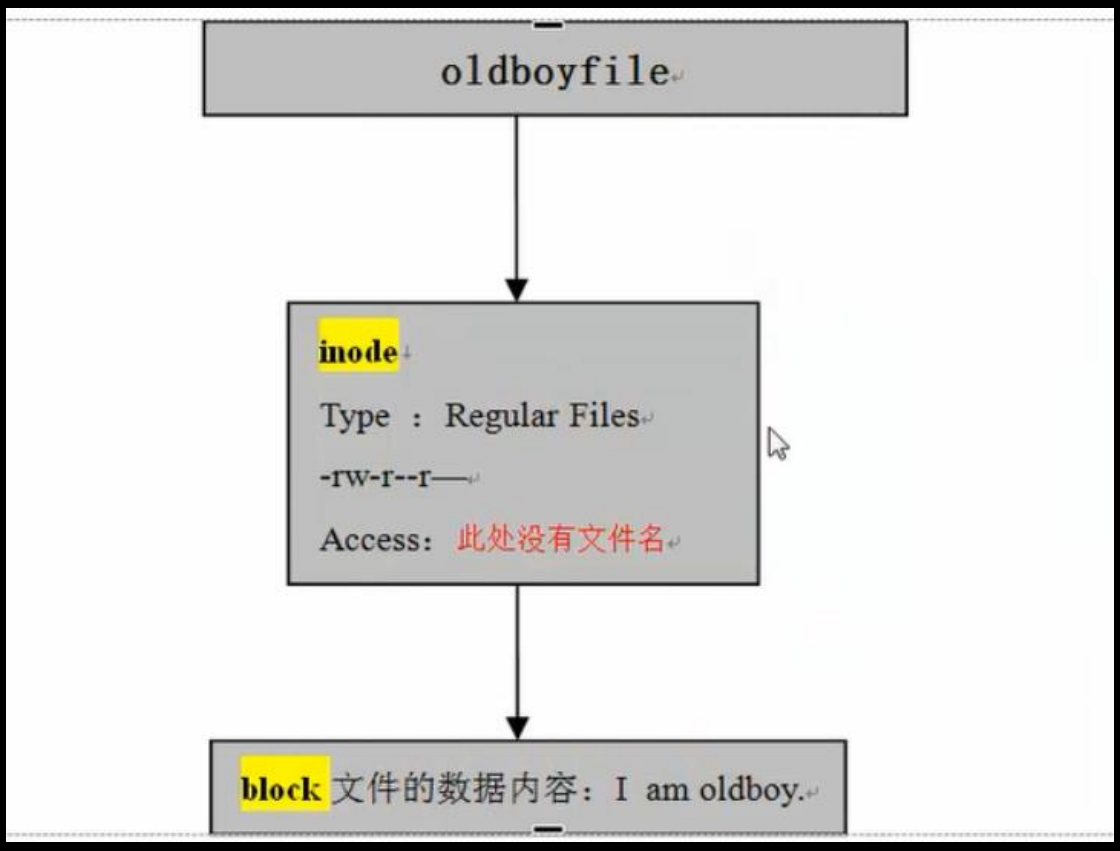
# Linux 学习

## 文件系统

硬盘分区，格式化、创建文件系统
被格式化的磁盘分为两部分：第一部分是 Inode 第二部分是 block
block 是用来存储实际数据用的，例如：照片、视频等普通文件数据
inode 是用来存储这些数据的属性的（也就是 ls-l 的结果）
inode 包含的属性信息有文件大小、属主、归属的用户组、读写权限、问价类型、修改时间，还有指向文件实体指针的功能（inode 节点----block 的对应关系），但是**唯独不包含文件名**




访问一个文件【**通过文件名找到 inode---->block** 】


# Inode

文件储存:

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

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）

文件数据都储存在"块"中，那么很显然，我们还必须找到一个地方储存文件的元信息，比如文件的创建者、文件的创建日期、文件的大小等等。这种储存文件元信息的区域就叫做 inode，中文译名为"索引节点"

每一个文件都有对应的 inode，里面包含了与该文件有关的一些信息

具体来说有以下内容:

* 文件的字节数
* 文件拥有者的 User ID
* 文件的 Group ID
* 文件的读、写、执行权限
* 文件的时间戳，共有三个：ctime 指 inode 上一次变动的时间，mtime 指文件内容上一次变动的时间，atime 指文件上一次打开的时间。
* 硬链接数，即有多少文件名指向这个 inode
* 文件数据 block 的位置

可以用 stat 命令，查看某个文件的 inode 信息:

```
stat example.txt
```

inode 也会消耗硬盘空间，所以硬盘格式化的时候，操作系统自动将硬盘分成两个区域。一个是数据区，存放文件数据；另一个是 inode 区（inode table），存放 inode 所包含的信息。
每个 inode 节点的大小，一般是 128 字节或 256 字节

## inode 号码

每个 inode 都有一个号码，操作系统用 inode 号码来识别不同的文件。
Unix/linux 系统内部不使用文件名，而使用 inode 号码来识别文件。对于系统来说，文件名只是 inode 号码便于识别的别称或者绰号。

表面上，用户通过文件名，打开文件。实际上，**系统内部这个过程分成三步：**
**1. 首先，系统找到这个文件名对应的 inode 号码；**
**2. 其次，通过 inode 号码，获取 inode 信息；**
**3. 最后，根据 inode 信息，找到文件数据所在的 block，读出数据。**

使用 ls -i 命令，可以看到文件名对应的 inode 号码：

```
    ls -i example.txt
```

## 目录文件

Unix/Linux 系统中，目录（directory）也是一种文件。打开目录，实际上就是打开目录文件。

目录文件的结构非常简单，就是一系列目录项（dirent）的列表。每个目录项，由两部分组成：所包含文件的文件名，以及该文件名对应的 inode 号码。

## 硬链接

一般情况下，文件名和 inode 号码是"一一对应"关系，每个 inode 号码对应一个文件名。
但是，Unix/Linux 系统允许，**多个文件名指向同一个 inode 号码**。
这意味着，可以用不同的文件名访问同样的内容；对文件内容进行修改，会影响到所有文件名；但是，删除一个文件名，不影响另一个文件名的访问。这种情况就被称为"硬链接"（hard link）。
创建硬链接：

```
    ln 源文件 目标文件
```

运行上面这条命令以后，源文件与目标文件的 inode 号码相同，都指向同一个 inode。inode 信息中有一项叫做"链接数"，记录指向该 inode 的文件名总数，这时就会增加 1。

创建目录时，默认会生成两个目录项："."和".."。前者的 inode 号码就是当前目录

的 inode 号码，等同于当前目录的"硬链接"；后者的 inode 号码就是当前目录的父目录的 inode 号码，等同于父目录的"硬链接"。所以，任何一个目录的"硬链接"总数，总是等于 2（某一目录的目录名和该目录的当前目录名）加上它的子目录总数（含隐藏目录）。（因为 inode 信息中有一项叫做"链接数"，记录指向该 inode 的文件名总数）

```
#include <unistd.h>
int link(const char *oldpath, const char *newpath);
(Return: 0 if success; -1 if failure)
```

新的文件名，两个文件名同文件

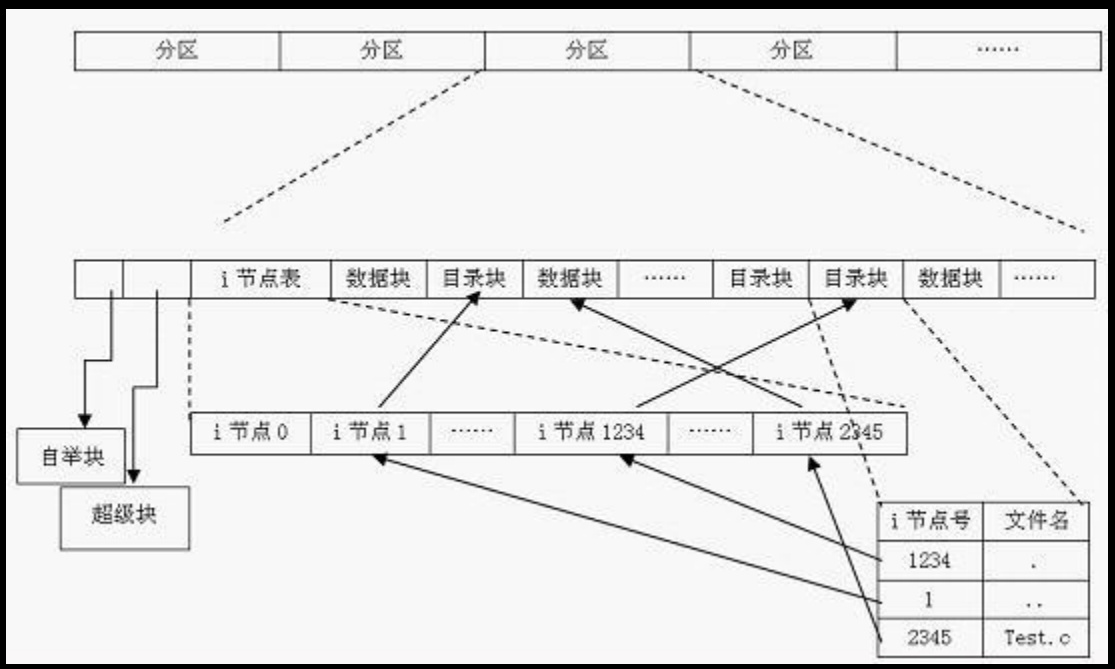
软链接

除了硬链接以外，还有一种特殊情况。

文件 A 和文件 B 的 inode 号码虽然不一样，但是文件 A 的内容是文件 B 的路径。读取文件 A 时，系统会自动将访问者导向文件 B。因此，无论打开哪一个文件，最终读取的都是文件 B。这时，文件 A 就称为文件 B 的"软链接"（soft link）或者"符号链接（symbolic link）

```
#include <unistd.h>
int symlink(const char *oldpath, const char *newpath);
(Return: 0 if success; -1 if failure)
```

这意味着，文件 A 依赖于文件 B 而存在，如果删除了文件 B，打开文件 A 就会报错："No such file or directory"。这是软链接与硬链接最大的不同：文件 A 指向文件 B 的文件名，而不是文件 B 的 inode 号码，文件 B 的 inode"链接数"不会因此发生变化。




## 7 种文件类型

1. 普通文件类型
   Linux 中最多的一种文件类型，包括 纯文本文件(ASCII)；二进制文件(binary)；数据格式的文件(data)；各种压缩文件.第一个属性为 [-]
2. 目录文件
   就是目录， 能用 # cd 命令进入的。第一个属性为 [d]，例如 [drwxrwxrwx]
3. 块设备文件
   就是存储数据以供系统存取的接口设备，简单而言就是**硬盘**。例如一号硬盘的代码是 /dev/hda1 等文件。第一个属性为 [b]
4. 字符设备
   即串行端口的接口设备，例如键盘、鼠标等等。第一个属性为 [c]
5. 套接字文件
   这类文件通常用在网络数据连接。可以启动一个程序来监听客户端的要求，客户端就可以通过套接字来进行数据通信。第一个属性为 [s]，最常在 /var/run 目录中看到这种文件类型
6. 管道文件
   FIFO 也是一种特殊的文件类型，它主要的目的是，解决多个程序同时存取一个文件所造成的错误。FIFO 是 first-in-first-out(先进先出)的缩写。第一个属性为 [p]
7. 链接文件
   类似 Windows 下面的快捷方式。第一个属性为 [l]，例如 [lrwxrwxrwx]

普通
目录
块
字符
套接字
管道
链接

**普通木块接管了字节**

## 查看文件信息

方式有三:
1. Ls
2. File
3. stat

```
ls -lhi
```

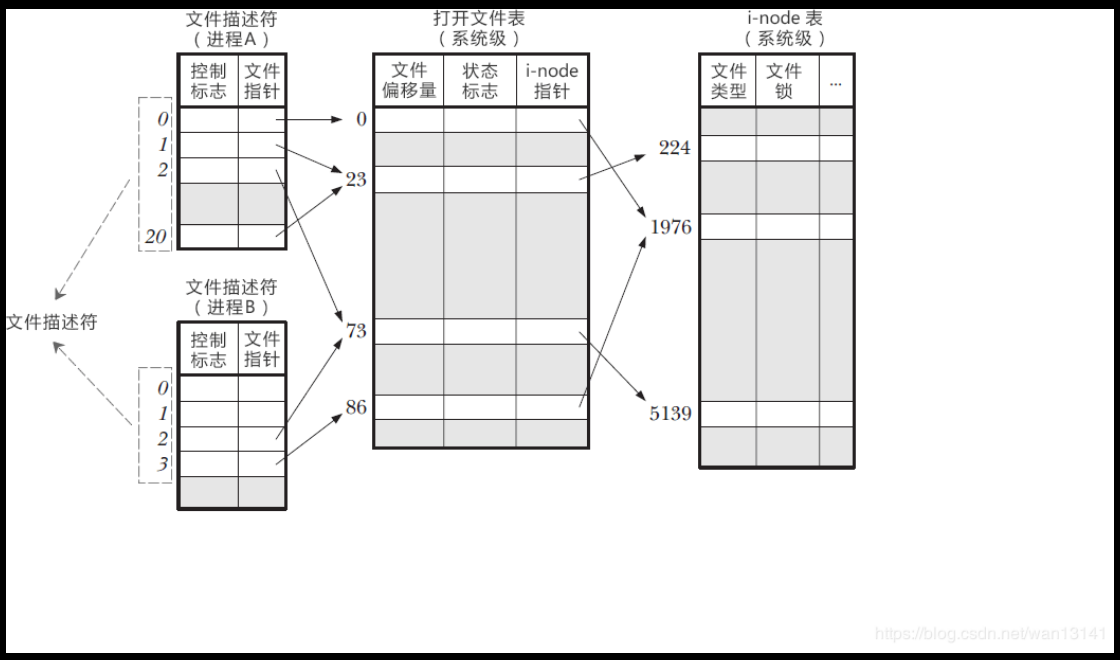
```
ls -lhi
total 4.0K
1445264 -rw-rw-r-- 1 ubuntu ubuntu 79 Aug 13 20:29 ReadMe.md
```


## 文件描述符

Linux 中一切皆文件

为了表示和区分已经打开的文件，Linux 会给每个文件分配一个编号（一个 ID），这个编号就是一个整数，被称为文件描述符（File Descriptor）

一个 Linux 进程启动后，会在内核空间中创建一个 PCB 控制块，PCB 内部有一个**文件描述符表**（File descriptor table），记录着当前进程所有可用的文件描述符，也即当前进程所有打开的文件。



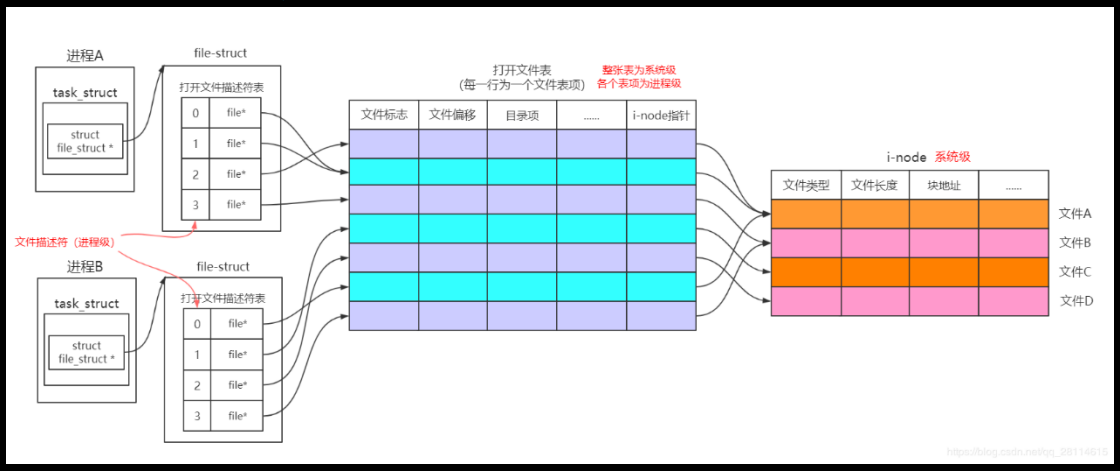

文件描述符是 linux 内核为了**高效管理已被打开的文件所创建的索引**，所有的 IO 操作系统调用都是使用文件描述符。

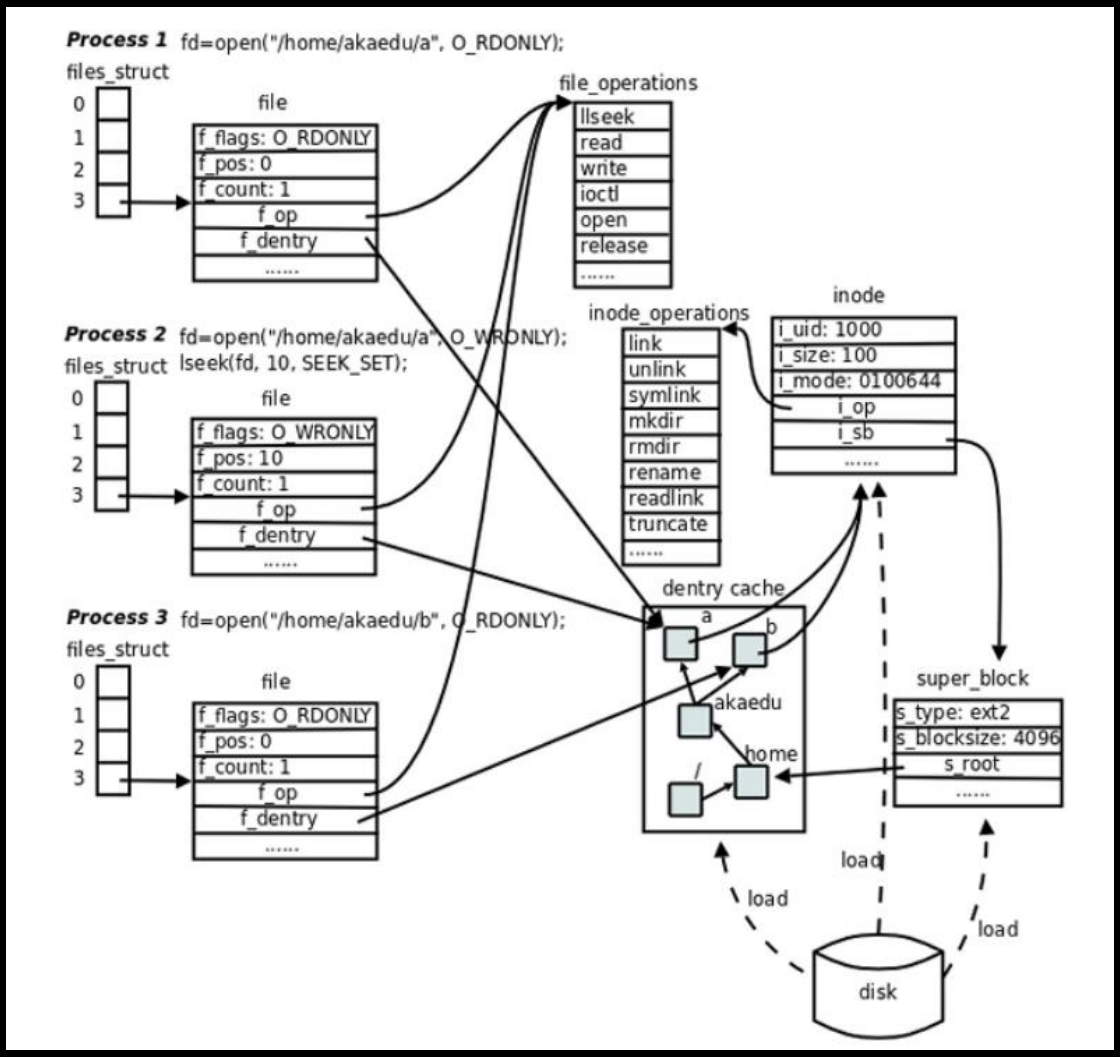
Edwin Xu

每打开一个文件都会创建文件描述符，并将文件指针指向这个文件描述符，文件描述符由非负整数表示，系统默认的 3 个文件描述符是 0，1，2，即标准输入、标准输出、标准错误输出。

此时打开一个文件即从 3 开始，写入到文件描述符表中。每个进程在 PCB（Process Control Block）即进程控制块中都保存着一份文件描述符表。

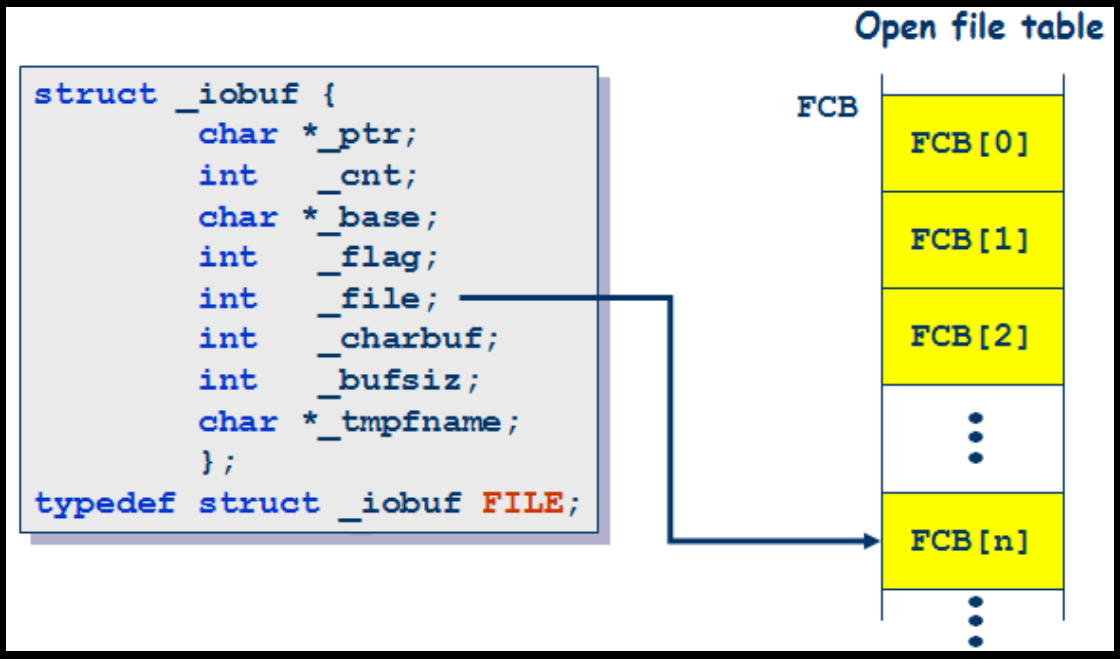
1. 每个进程对应一张打开文件描述符表，这是进程级数据结构，也就是每一个进程都各自有这样一个数据结构；
2. 内核维持一张打开文件表，文件表由多个文件表项组成，这是系统级数据结构，也就是说这样的数据结构是针对于整个内核而言的，每个进程都可共享的；
3. 每个打开的文件对应一个 i 节点（i-node）数据结构（Linux 下只有 i 节点没有 v 节点），由于这是每一个打开的文件与之对应的，因此这也是一个系统级数据结构，存在于内核中，非进程所独有。






## 文件指针 FILE*

文件指针变量标示符，一般形式为大写，可以看出是系统定义的一个结构体，该结构体中含有一系列文件名，文件状态，当前位置信息，文件描述符 fd，还有缓冲区等。C 程序用不同的 FILE 结构管理每个文件。程序员可以使用文件，但是不需要知道 FILE 结构的细节。实际上，FILE 结构是间接地操作系统的文件控制块(FCB)来实现对文件的操作的：




文件是存放在物理磁盘上的，包括文件控制块(FCB)和数据块。文件控制块通常包括文件权限、日期（创建、读取、修改）、拥有者、文件大小、数据块信息。数据块用来存储实际的内容。当打开一个文件时，程序会将物理磁盘上的文件数据块读入到内存，然后通过文件指针的移动读取内存中的文件数据。

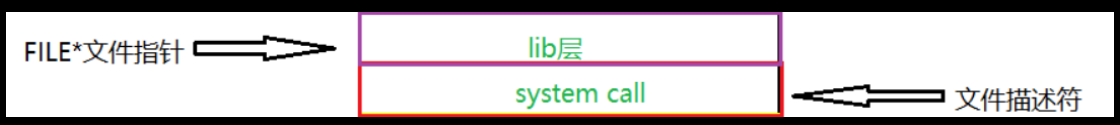
```
FILE *fopen( char *file, char *open_mode ); //打开文件，读文件到内存，返回文件信息结构指针

FILE *fpt = fopen("a.txt","r");
```

```
typedef struct{
    short level;/*缓冲区“满/空”的程度*/
    unsigned flags;/*文件状态标志字*/
    char fd;
    unsigned char hold;
    short bsize;/*缓冲区大小*/
    unsigned char *buffer;/*数据缓冲区的位置*/
    unsigned char *curp;/*当前读写位置指针*/
    unsigned istemp;
    short token;
}FILE;
```

FILE *里面存放的都是 open 打开的原始 fd 和一些文件信息，譬如当前文件偏移、文件大小、读写状态等等。

## Fd vs. FILE*




1. FILE*是库函数，使用 fopen、fclose、fread、fwirte 对文件进行操作时
2. Fd 是系统调用层，调用的接口为 write/read、close/open，它们的返回值为 fd。
3. FILE*指向的结构体里面包含 fd

## 硬链接与软链接的区别

由于 linux 下的文件是通过索引节点（Inode）来识别文件，硬链接可以认为是一个指针，指向文件索引节点的指针，系统并不为它重新分配 inode。每添加一个一个硬链接，文件的链接数就加 1。

```
ln file2 file2hard
```

尽管硬链接节省空间，也是 Linux 系统整合文件系统的传统方式，但是存在一下不足之处：

*（1）不可以在不同文件系统的文件间建立链接

*（2）只有超级用户才可以为目录创建硬链接。

软链接（符号链接）：

软链接克服了硬链接的不足，没有任何文件系统的限制，任何用户可以创建指向目录的符号链接。因而现在更为广泛使用，它具有更大的灵活性，甚至可以跨越不同机器、不同网络对文件进行链接。

给 ln 命令加上- s 选项，则建立软链接。

```
Ln -s file softfile
```

建立软链接就是建立了一个新文件。当访问链接文件时，系统就会发现他是个链接文件，它读取链接文件找到真正要访问的文件。保存了其代表的文件的绝对路径，是另外一种文件，在硬盘上有独立的区块，访问时替换自身路径。

缺点：

因为链接文件包含有原文件的路径信息，所以当原文件从一个目录下移到其他目录中，再访问链接文件，系统就找不到了，而硬链接就没有这个缺陷，你想怎么移就怎么移；还有它要系统分配额外的空间用于建立新的索引节点和保存原文件的路径。

## Linux 管道 pipe 的实现原理

管道是进程间通信的主要手段之一。一个管道实际上就是个只存在于内存中的文件，

对这个文件的操作要通过两个已经打开文件进行，它们分别代表管道的两端。
管道是一种特殊的文件，它**不属于某一种文件系统，而是一种独立的文件系统，有其自己的数据结构**。
根据管道的适用范围将其分为：**无名管道和命名管道**。

### 无名管道

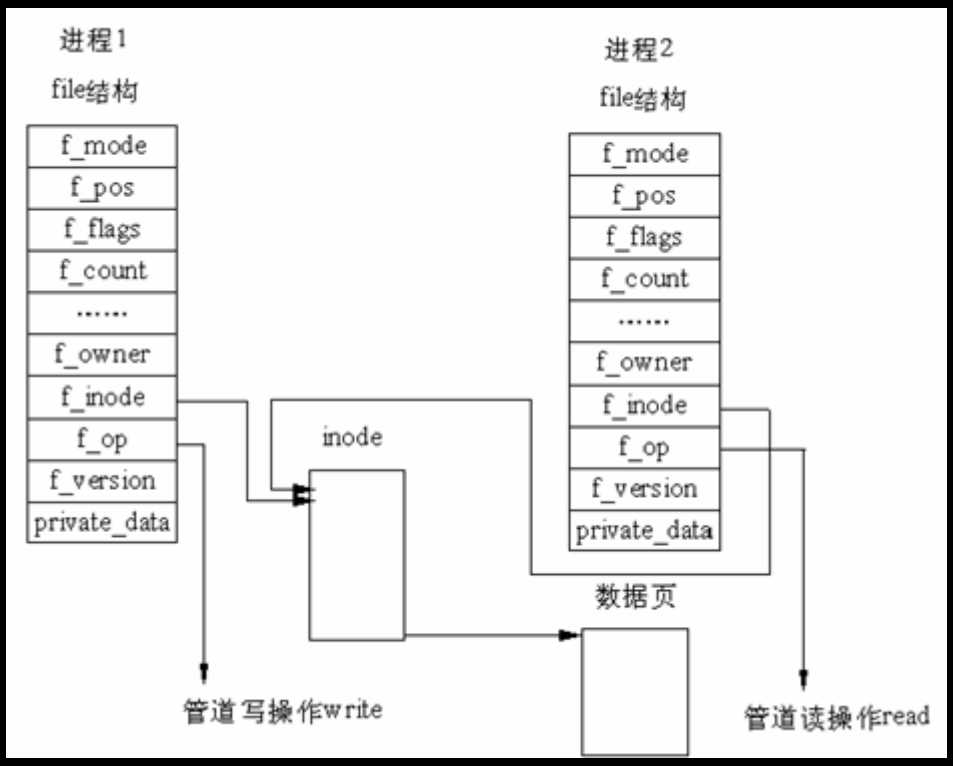
主要用于**父进程与子进程**之间，或者两个兄弟进程之间。在 linux 系统中可以通过系统调用建立起一个**单向**的通信管道，且这种关系只能由父进程来建立。

### 命名管道

命名管道是**建立在实际的磁盘介质或文件系统（而不是只存在于内存中）上有自己名字的文件，任何进程可以在任何时间通过文件名或路径名与该文件建立联系**。为了实现命名管道，引入了一种新的文件类型——FIFO 文件（遵循先进先出的原则）。
实现一个命名管道实际上就是实现一个 FIFO 文件。命名管道一旦建立，之后它的读、写以及关闭操作都与普通管道完全相同。虽然 FIFO 文件的 inode 节点在磁盘上，但是仅是一个节点而已，文件的数据还是存在于内存缓冲页面中，和普通管道相同。

在 Linux 中，管道的实现并没有使用专门的数据结构，而是**借助了文件系统的 file 结构和 VFS 的索引节点 inode**。
通过将**两个 file 结构指向同一个临时的 VFS 索引节点，而这个 VFS 索引节点又指向一个物理页面而实现**的。

如下图有两个 file 数据结构，但它们定义文件操作例程地址是不同的，其中一个是向管道中写入数据的例程地址，
而另一个是从管道中读出数据的例程地址。
这样，**用户程序的系统调用仍然是通常的文件操作**，而内核却利用这种抽象机制实现了管道这一特殊操作。




读写操作

管道实现的源代码在 **fs/pipe.c（系统调用）**中，在 pipe.c 中有很多函数，其中有两个函数比较重要，

即管道读函数 **pipe_read()**和管道写函数 **pipe_wrtie()**。

管道写函数通过将字节复制到 VFS 索引节点指向的物理内存而写入数据，而管道读函数则通过复制物理内存中的字节而读出数据。

当然，内核必须利用一定的机制同步对管道的访问，为此，**内核使用了锁、等待队列和信号**。

当写进程向管道中写入时，它利用**标准的库函数 write()**，系统根据库函数传递的文件描述符，可找到该文件的 file 结构。

file 结构中指定了用来进行写操作的函数（即写入函数）地址，于是，内核调用该函数完成写操作。(前提：内存没有被读程序锁定)

不过锁已经被占用了，加入等待队列

## OS

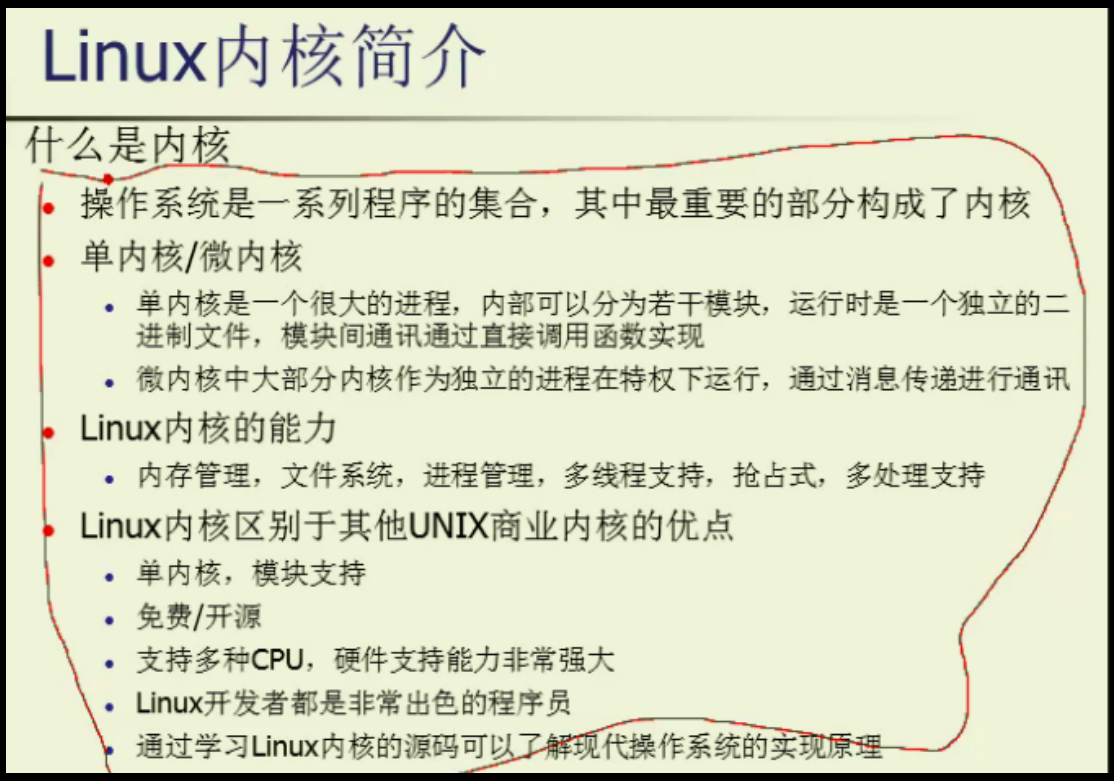
什么是操作系统?

操作 OperationSystem 是计算机最基础的底层软件，和硬件直接交互，管理这计算机的多方面，包括作业管理，进程管理，内存管理，文件管理，

OS 是计算机系统最基础的系统软件， 管理软硬件资源、控制程序执行，改善人机界

面，合理组织计算机工作流程，为用户使用计算机提供良好运行环境

什么是 OS 内核?
内核是操作系统最基本的部分。它是为众多应用程序提供对计算机硬件的安全访问的一部分软件，这种访问是有限的，并且内核决定一个程序在什么时候对某部分硬件操作多长时间。内核的分类可分为单内核和双内核以及微内核。严格地说，内核并不是计算机系统中必要的组成部分。
内核的基本功能
支撑功能：中断处理、时钟管理、原语操作
资源管理功能：进程管理、存贮管理、设备管理




## 字符设备和块设备的区别

在 LINUX 里面，设备类型分为：字符设备、块设备以及网络设备
Character Device Drive 又被称为字符设备或裸设备 raw devices; Block Device Driver 通常成为块设备。而 Block Device Driver 是以固定大小长度来传送转移资料 ；Character Device Driver 是以不定长度的字元传送资料 。且所连接的 Devices 也有所不同，Block Device 大致是可以随机存取(Random Access)资料的设备，如硬碟机或光碟机；而 Character Device 刚好相反，依循先後顺序存取资料的设备，如印表机 、终端机等皆是。

**字符设备按照字符流的方式被有序访问**，像**串口和键盘**就都属于字符设备。如果一个

硬件设备是以字符流的方式被访问的话，那就应该将它归于**字符设备**；反过来，如果一个设备是随机（无序的）访问的，那么它就属于块设备。

区别：
1. 字符设备只能以字节为最小单位访问，而块设备**以块为单位访问**，例如 512 字节，1024 字节等
2. 块设备可以随机访问，但是字符设备不可以
3. 字符和块没有访问量大小的限制，块也可以以字节为单位来访问

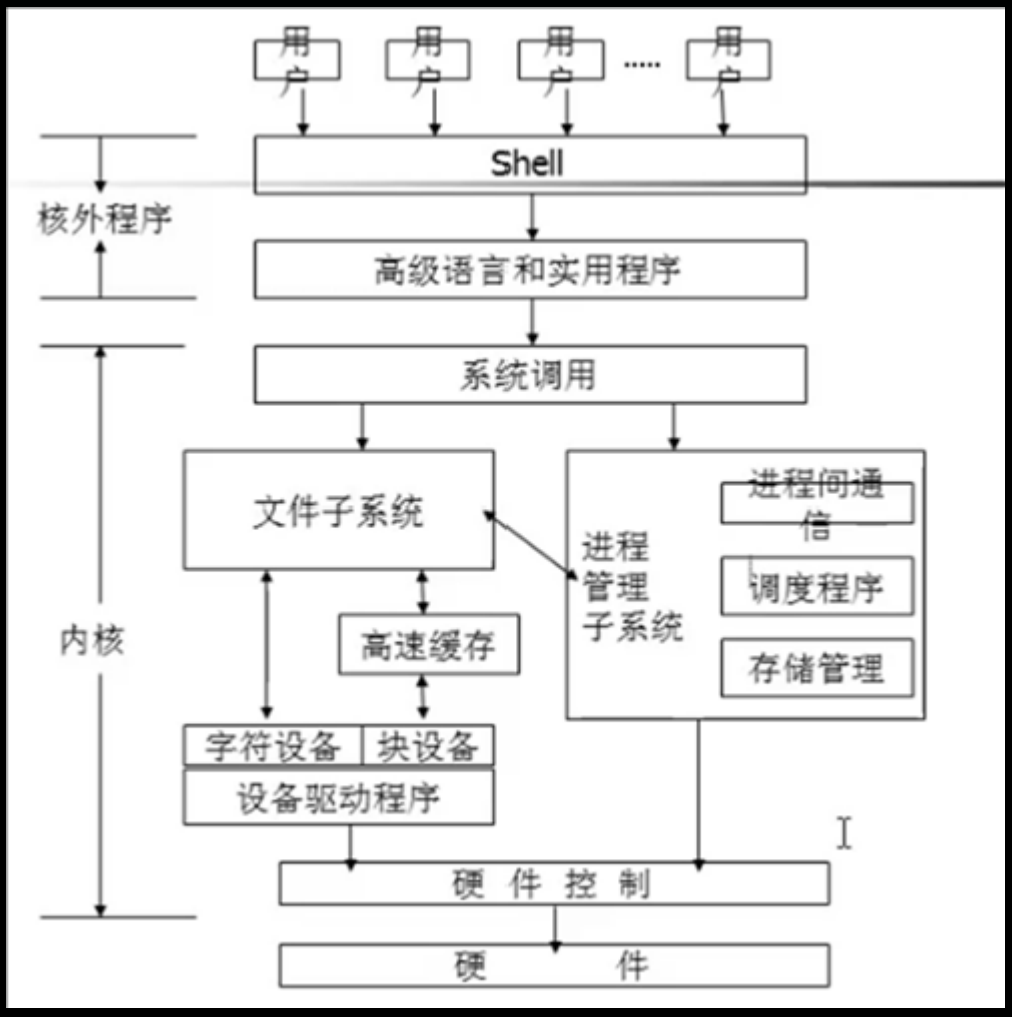
## Shell

用户和操作系统的接口

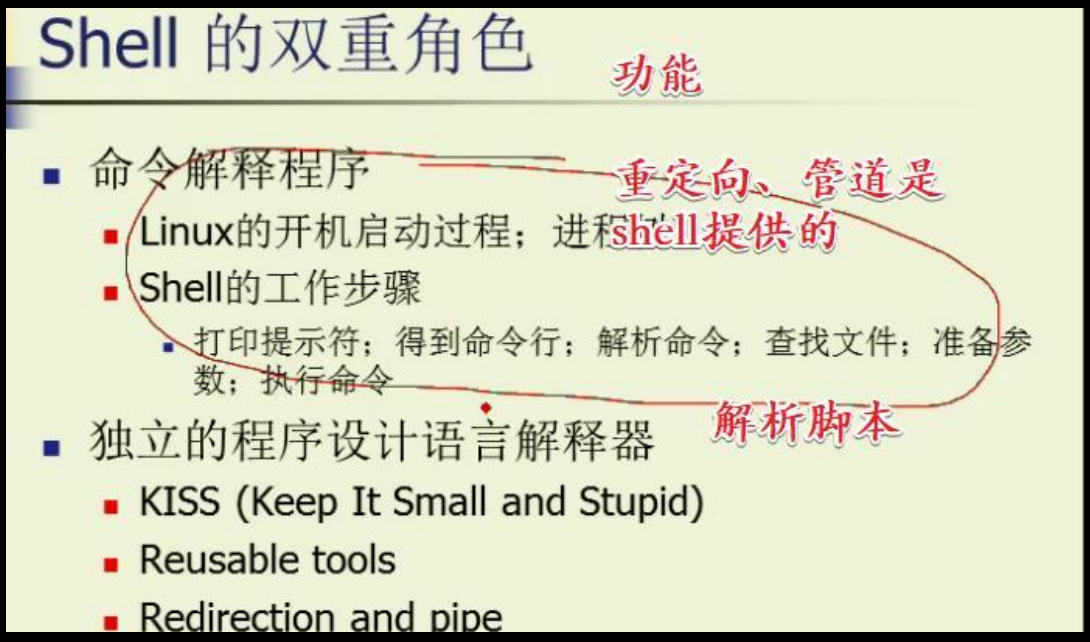
一个广义的解释就是在用户与操作系统之间，提供一个工具或接口给用户来操作计算机系统；用户在 shell 中通过输入命令行，按下回车键，shell 执行命令后就能返回结果，达到操作计算机的效果。

Shell 即一个用户态程序，用户通过 shell 可以调用内核提供的系统调用
用户使用 Linux 的桥梁
Shell 是指一种应用程序，这个应用程序提供了一个界面，用户通过这个界面访问操作系统内核的服务。
Shell **既是一种命令语言，又是一种程序设计语言**。

Shell 的位置：




Shell 的功能： （命令式语言，程序设计式语言）




## chmod

chmod [-cfvR] [--help] [--version] mode file...

Edwin Xu

1. u 表示该文件的拥有者，g 表示与该文件的拥有者属于同一个群体(group)者，o 表示其他以外的人，a 表示这三者皆是。
2. + 表示增加权限、- 表示取消权限、= 表示唯一设定权限。
3. r 表示可读取，w 表示可写入，x 表示可执行，X 表示只有当该文件是个子目录或者该文件已经被设定过为可执行。

1. -c : 若该文件权限确实已经更改，才显示其更改动作
2. -f : 若该文件权限无法被更改也不要显示错误讯息
3. -v : 显示权限变更的详细资料
4. -R : 对目前目录下的所有文件与子目录进行相同的权限变更(即以递回的方式逐个变更)
5. --help : 显示辅助说明
6. --version : 显示版本

将文件 file1.txt 设为所有人皆可读取:

```
chmod ugo+r file1.txt
or:
chmod a+r file1.txt
a 表示所有人
```

将文件 file1.txt 与 file2.txt 设为该文件拥有者，与其所属同一个群体者可写入，但其他以外的人则不可写入 :

```
chmod ug+w,o-w file1.txt file2.txt
```

将 ex1.py 设定为只有该文件拥有者可以执行 :

```
chmod u+x ex1.py
```

将目前目录下的所有文件与子目录皆设为任何人可读取 :

```
chmod -R a+r *
```

此外 chmod 也可以用数字来表示权限如 :

```
chmod 777 file
```

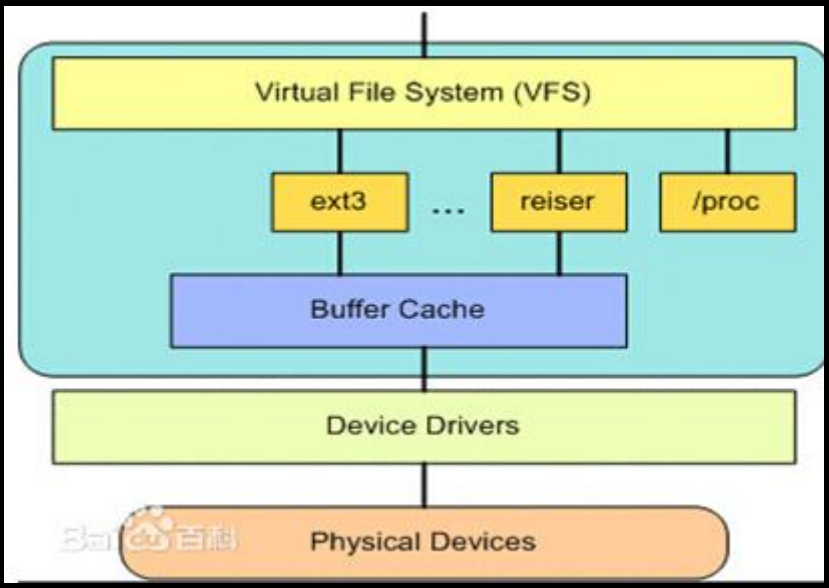
其中 a,b,c 各为一个数字，分别表示 User、Group、及 Other 的权限。

## VFS

Linux 系统中存在很多的文件系统，例如常见的 ext2,ext3,ext4,sysfs,rootfs,proc...很多很多。。。我们知道每个文件系统是独立的，有自己的组织方法，操作方法。那么对于用户来说，不可能所有的文件系统都了解，那么怎么做到让用户透明的去处理文件呢？例如：我想写文件，那就直接 read 就 OK，不管你是什么文件系统，具体怎么去读！OK，这里就需要引入虚拟文件系统 VFS。

VFS 作为中间一层！用户直接和 VFS 打交道。例如 read，write，那么映射到 VFS 中就是 sys_read，sys_write，那么 VFS 可以根据你操作的是哪个“实际文件系统”(哪

个分区)来进行不同的实际的操作!




Linux 虚拟文件系统四大对象:

**1）超级块(super block)**

**2）索引节点(inode)**

**3）目录项(dentry)**

**4）文件对象(file)**

超级块: 一个超级块对应一个文件系统(已经安装的文件系统类型如 ext2，此处是实际的文件系统哦，不是 VFS)

一个超级块对于一个独立的文件系统。保存文件系统的类型、大小、状态等等。

```
1. struct super_block {
2. 746         struct list_head       s_list;        /* Keep this first */
3. 747         kdev_t                 s_dev;
4. 748         unsigned long          s_blocksize;
5. 749         unsigned char          s_blocksize_bits;
```

索引节点 inode: 保存的其实是实际的数据的一些信息，这些信息称为“元数据”(也就是对文件属性的描述)。例如: 文件大小，设备标识符，用户标识符，用户组标识符，文件模式，扩展属性，文件读取或修改的时间戳，链接数量，指向存储该内容的磁盘区块的指针，文件分类等等。

inode 有两种，一种是 VFS 的 inode，一种是具体文件系统的 inode。前者在内存中，后者在磁盘中。所以每次其实是将磁盘中的 inode 调进填充内存中的 inode，这样才是算使用了磁盘文件 inode。

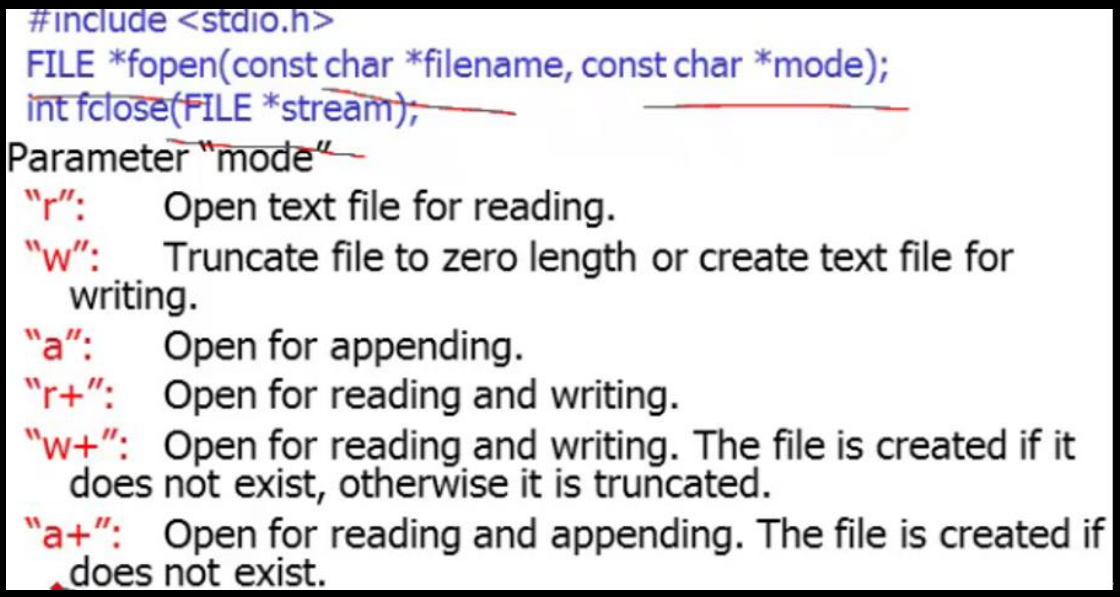
inode 和文件的关系:当创建一个文件的时候,就给文件分配了一个 inode。一个 inode 只对应一个实际文件，一个文件也会只有一个 inode。inodes 最大数量就是文件的最大数量。

目录项：目录项是**描述文件的逻辑属性**，**只存在于内存中**，并没有实际对应的磁盘上的描述，更确切的说是存在于内存的目录项缓存，为了提高查找性能而设计。注意不管是文件夹还是最终的文件，都是属于目录项，所有的目录项在一起构成一颗庞大的目录树。例如：open 一个文件/home/xxx/yyy.txt，那么/、home、xxx、yyy.txt 都是一个目录项，VFS 在查找的时候，根据一层一层的目录项找到对应的每个目录项的 inode，那么沿着目录项进行操作就可以找到最终的文件。

```
67 struct dentry {
68        atomic_t d_count;
69        unsigned int d_flags;
70        struct inode  * d_inode;       /* Where the name belongs to - NULL is negative
   */
71        struct dentry * d_parent;      /* parent directory */
72        struct list_head d_hash;       /* lookup hash list */
73        struct list_head d_lru;        /* d_count = 0 LRU list */
74        struct list_head d_child;      /* child of parent list */
75        struct list_head d_subdirs;    /* our children */
76        struct list_head d_alias;      /* inode alias list */
77        int d_mounted;
78        struct qstr d_name;
79        unsigned long d_time;          /* used by d_revalidate */
80        struct dentry_operations  *d_op;
81        struct super_block * d_sb;     /* The root of the dentry tree */
82        unsigned long d_vfs_flags;
83        void * d_fsdata;              /* fs-specific data */
84        unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */
85 }
```

文件对象：注意文件对象描述的是进程已经打开的文件。因为一个文件可以被多个进程打开，所以一个文件可以存在多个文件对象。但是由于文件是唯一的，那么 inode 就是唯一的，目录项也是定的！

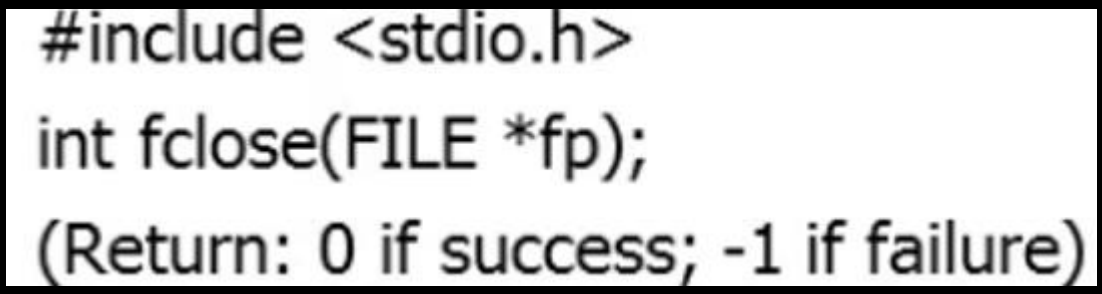
## 文件流



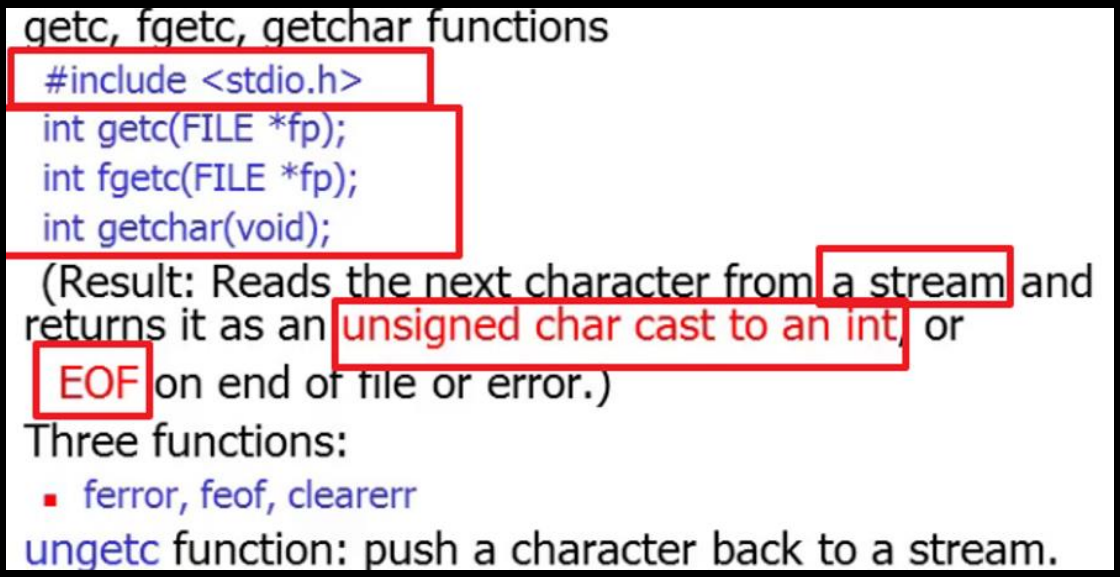

**Fopen(const char* name, const char *mode)**

1. R: 读
2. W: 覆盖写，无则创建
3. A: 追加写
4. R+: 读，覆盖写
5. W+: 读，覆盖写，无则创建
6. A+: 读，追加写，无则创建

Int fclose(FILE *fp)




字符输入：




```
int fgetc(FILE *stream); //传入流
char *fgets(char *s, int size, FILE *stream);
int getc(FILE *stream);
int getchar(void);
char *gets(char *s);
```

1. fgetc() ：从读取文件流中读取下一个字符，并将其以无符号数返回，或者返回文件末尾标志或错误标志。
2. getc() ：和 fgetc 差不多，只是在读取的时候会检查文件流。
3. getchar() ：和 getc(stdin)等效。
4. gets() ：从 stdin 中读取一行字符串，直到一行结束或是到文件末尾，字符串以’\0’结尾。但是不会检查是否有溢出。
5. fgets() ：读取少于 size 长度的字符，直到新的一行开始或是文件结束，最后会在读取的字符串最后一个字符后加一个结束字符’\0’

Getc\fgetc:功能相同，都是库函数，getc 执行稍快，因为在编译前先替换为系统调用了。

Getc 不能带副作用，否则可能造成错误。

## Linux 文件权限

我们看一下：0777 代表什么意义：

1.先看 777 的含义，linux 下文件权限分为所属用户权限、所属组权限和其他人权限

2.RWX 对应的权限是： r:4 w:2 x:1 所以 777 就代表 rwxrwxrwx

3.0777 最前面的 0，是一个叫 suid 和 guid 和 sticky 的东西，设置 suid 就是把 0 变为 4，设置 guid 就把 0 变为 2，设置 sticky 就是把 0 变为 1

同样第一位换成二进制也分成三部分

abc

a - setuid 位，如果该位为 1，则表示设置 setuid 对应 4
b - setgid 位，如果该位为 1，则表示设置 setgid 对应 2
c - sticky 位，如果该位为 1，则表示设置 sticky 对应 1

1. setuid: 设置使文件在执行阶段具有文件所有者的权限. 典型的文件是 /usr/bin/passwd. **如果一般用户执行该文件，则在执行过程中，该文件可以获得 root 权限**，从而可以更改用户的密码.
2. setgid: 该权限只对目录有效. 目录被设置该位后，**任何用户在此目录下创建的文件都具有和该目录所属的组相同的组**.
3. sticky bit: 该位可以理解为**防删除位**. 一个文件是否可以被某用户删除，主要取决于该文件所属的组是否对该用户具有写权限.如果没有写权限，则这个目录下的所有文件都不能被删除，同时也不能添加新的文件. 如果**希望用户能够添加文件但同时不能删除文件**,则可以对文件使用 **sticky bit 位. 设置该位后，就算用户对目录具有写权限，也不能删除该文件**. （让用户可以写，但是不能删，就是防删的）

## chmod/chown 函数

Change permissions of a file

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
(Return: 0 if success; -1 if failure)
```

参数fd，已经打开的文件

Change ownership of a file

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

可以改用户和用户组

```
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
(Return: 0 if success; -1 if failure)
```

fd打开的文件

软链接

# umask function

- 为进程设置文件存取权限屏蔽字，并返回以前的值

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t mask);
```

## 目录函数

```
#include <unistd.h>
int chdir(const char *path);
int fchdir(int fd);
(Return: 0 if success; -1 if failure)
```

change dir

unistd.h系统调用

带f的函数名都是使用fd

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode);
(Return: 0 if success; -1 if failure)
```

## 删除一个空目录

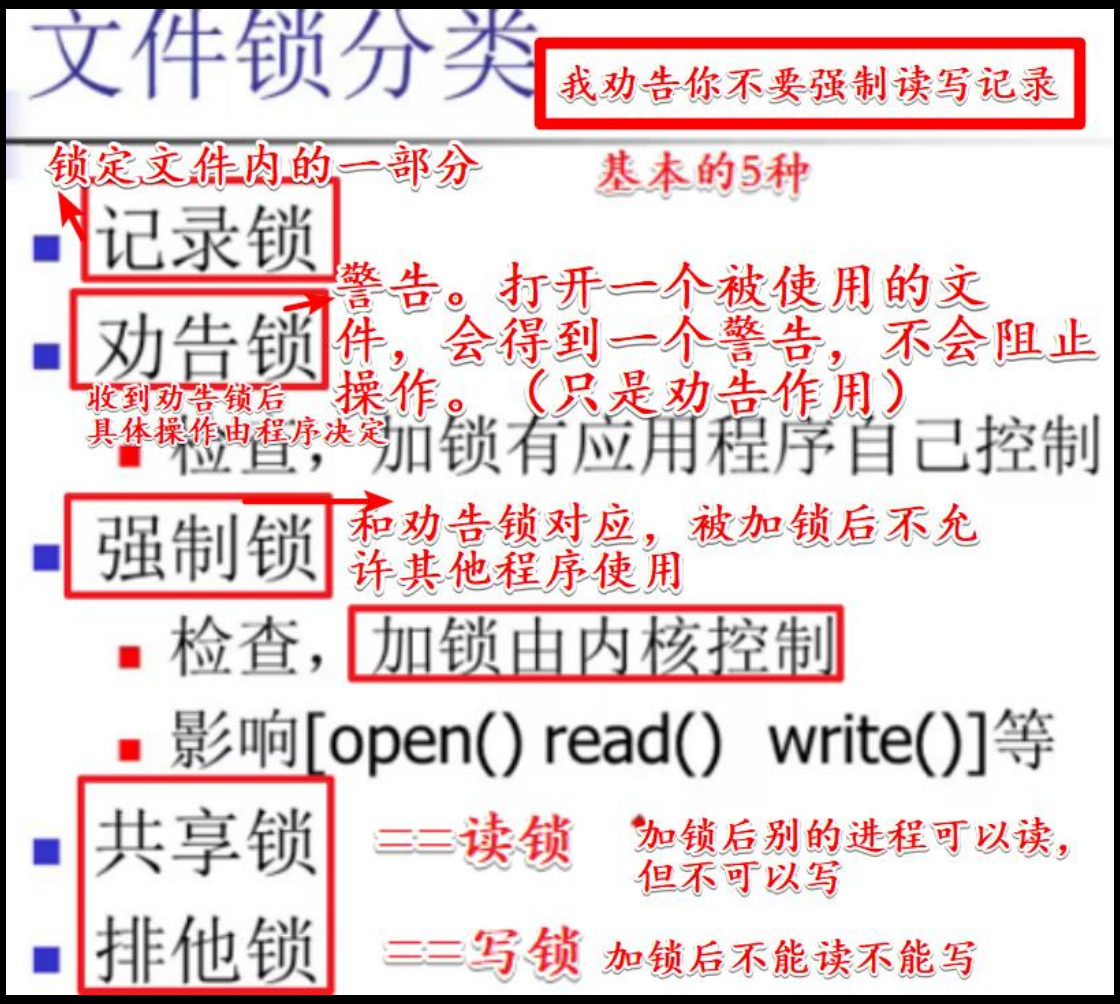
```
#include <unistd.h>
int rmdir(const char *pathname);
(Return: 0 if success; -1 if failure)
```

目录的打开、关闭、读、定位

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

DIR *opendir(const char *name);
int closedir(DIR *dir);
struct dirent *readdir(DIR *dir);
off_t telldir(DIR *dir);
void seekdir(DIR *dir, off_t offset);
```

文件锁




记录

劝告
强制锁
共享锁
排他锁
(排他锁只是写锁，加锁后不然其他程序读写；而强制锁是加锁后不让操作，包括读写)

## Umask

umask 命令用来设置限制新建文件权限的掩码。当新文件被创建时，其最初的权限由文件创建掩码决定。用户每次注册进入系统时，umask 命令都被执行， 并自动设置掩码 mode 来限制新文件的权限。用户可以通过再次执行 umask 命令来改变默认值，新的权限将会把旧的覆盖掉。

```
umask(选项)(参数)
-p: 输出的权限掩码可直接作为指令来执行;
-S: 以符号方式输出权限掩码。
```

```
umask u=, g=w, o=rwx
```

执行该命令以后，对于下面创建的新文件，其文件主的权限未做任何改变，而组用户没有写权限，其他用户的所有权限都被取消

操作符“=”在 umask 命令和 chmod 命令中的作用恰恰相反。在 chmod 命令中，利用它来设置指定的权限，而其余权限则被删除；但是在 umask 命令中，它**将在原有权限的基础上删除指定的权限**。

系统默认的掩码是 0022。 umask -S 显示

```
ubuntu@VM-0-13-ubuntu:~$ umask
0002
```

可以看到 umask 值为 0002，其中第一个 0 与特殊权限有关，可以暂时不用理会，后三位 002 则与普通权限(rwx)有关，其中 002 中第一个 0 与用户(user)权限有关，表示从用户权限减 0，也就是权限不变，所以文件的创建者的权限是默认权限(rw)，第二个 0 与组权限(group)有关，表示从组的权限减 0，所以群组的权限也保持默认权限(rw)，最后一位 2 则与系统中其他用户（others）的权限有关，由于 w=2，所以需要从其他用户默认权限（rw）减去 2，也就是去掉写（w）权限，则其他人的权限为 rw - w = r，则创建文件的最终默认权限为 -rw-rw-r-- 。同理，目录的默认权限为 drwxrwxrwx ，则 d rwx rwx rwx - 002 = (d rwx rwx rwx) - (- --- --- -w-) = d rwx rwx r-x，所以用户创建目录的默认访问权限为 drwxrwxr-x

## 读写文件：系统调用 vs. 库函数

write 系统调用

**size_t write(int fildes, const void *buf, size_t nbytes);**

fildes: 文件描述符，标识了要写入的目标文件。例如:fildes 的值为 1，就像标准输出写数据，也就是在显示屏上显示数据；如果为 2 ，则想标注错误写数据。

*buf: 待写入的文件，是一个字符串指针。

nbytes: 要写入的字符数。

read 系统调用

**size_t read(int fildes, void *buf, size_t nbytes);**

size_t 返回成功读取的字符数，它可能会小于请求的字节数。

open 系统调用

系统调用 open 的作用是打开一个文件，并返回这个文件的描述符。

int open(const char *path, int oflags);

int open(const char *path, int oflags, mode_t mode );

oflags: 指出要打开文件的访问模式。

1. O_APPEDN: 把写入数据追加在文件的末尾。
2. O_TRUNC: 把文件长度设为零，丢弃以后的内容。
3. O_CREAT: 如果需要，就按参数 mode 中给出的访问模式创建文件。
4. O_EXCL: 与 O_CREAT 一起调用，确保调用者创建出文件。使用这个模式可防止两个程序同时创建一个文件，如果文件已经存在，open 调用将失败。

mode:

当使用哦、O_CREAT 标志的 open 来创建文件时，我们必须使用三个参数格式的 open 调用。第三个参数 mode 是几个标志按位 OR 后得到的。他们是:

S_IRUSR: 读权限，文件属主。

S_IWUSR:写权限，文件属主。

S_ IXUSR:执行权限，文件属主。

S_IRGRP:读权限，文件所属组。

S_IWGRP:写权限，文件所属组。

```
open（"myfile", O_CREAT）
```

close 系统调用

```
int close(int fildes);
```

终止一个文件描述符 fildes 以其文件之间的关联。文件描述符被释放，并能够重新使用。

close 成功返回 1，出错返回-1.

ioctl 系统调用

提供了一个用于控制设备及其描述符行为和配置底层服务的接口。

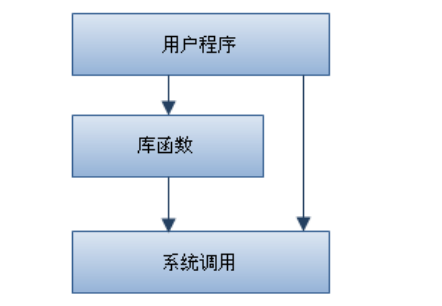
```
#include<unistd.h>
int ioctl(int fildes, int cmd,,,,,,);
```

## 库函数

在输入、输出操作中，直接使用系统调用效率会非常底。具体原因有二：

1. 系统调用会影响系统性能。与函数调用相比，系统调用的开销大。因为在执行系统调用的时候，要切换到内核代码区执行，然后再返回用户代码。这必然就需要大量的时间开支。一种解决办法是：尽量减少系统调用的次数，让每次系统调用完成尽可能多的 任务。例如每次系统调用写入大量的字符而不是单个字符。
2. 硬件会对系统调用一次能读写的数据块做一定的限制。

为了提高文件访问操作的效率，并且使得文件操作变得更方便，Linux 发行版提供了一系列的标准函数库。




标准 I/O 库及其头文件<stdio.h>为底层 I/O 系统调用提供了一个通用的接口。

在许多方面，使用标准 I/O 库和使用底层文件描述符类似。需要先打开一个文件，已建立一个文件访问路径（也就是系统调用中的文件描述符）

在标准 I/O 库中，与文件描述符对应的叫**流（stream）**，它被实现为指向结构 **FILE** 的指针。

在启动程序时，有三个文件流是自动打开的。他们是：

1. **stdin：标准输入**
2. **stdout：标准输出**
3. **stderr：标准错误输出**

fopen 函数

fopen 函数类似于系统调用中的 open 函数。和 open 一样，它返回文件的标识符，只是这里叫做流（stream），在库函数里实现为一个指向文件的指针。

```
#include<stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

*mode: 打开的方式

1. r 以只读方式打开文件，该文件必须存在。
2. r+ 以可读写方式打开文件，该文件必须存在。
3. rb+ 读写打开一个二进制文件，允许读数据。
4. rw+ 读写打开一个文本文件，允许读和写。
5. w 打开只写文件，若文件存在则文件长度清为 0，即该文件内容会消失。若文件不存在则建立该文件
6. w+ 打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。

成功是返回一个非空的 FILE *指针。失败返回 NULL

fread/fwrite 函数

fread 函数从文件流中读取数据，对应于系统调用中的 read；fwrite 函数从文件流中写数据，对应于系统调用中的 write

```
#include<stdio.h>
size_t  fread(void *ptr, size_t size, size_t nitems, FILE *stream);
```

*ptr 要读取数据的缓冲区，也就是要存放读取数据的地方。

size：指定每个数据记录的长度。

nitems：  计数，给出要传输的记录个数。

```
size_t  fwrite(const coid *ptr, size_t size , size_t nitimes, FILE *stream);
```

fclose 函数

```
int fclose(FILE *stream);
```

fflush 函数

fflush 函数的作用是把文件流中所有未写出的数据全部写出。 处于效率考虑，在使用库函数的时候会使用数据缓冲区，当缓冲区满的时候才进行写操作。使用 fflush 函数可以将缓冲区的数据全部写出，而不关心缓冲区是否满。fclose 的执行隐含调用了 fflush 函数，所以不必再 fclose 执行之前调用 fflush。

```
int fflush(FILE *stream);
```

## unistd.h

unistd.h 不是 c 语言的东西，是 linux/unix 的**系统调用**，包含了许多 U N I X 系统服务的函数原型,例如 r e a d,

w r i t e 和 getpid 函数。

## More

more 功能类似 cat ，cat 命令是整个文件的内容从上到下显示在屏幕上。 more 会以一页一页的显示方便使用者逐页阅读，而最基本的指令就是按空白键（space）就往下一页显示，按 b 键就会往回（back）一页显示，而且还有搜寻字串的功能 。more 命令从前向后读取文件，因此在启动时就加载整个文件。

```
more [-dlfpcsu ] [-num ] [+/ pattern] [+ linenum] [file ... ]
```

**命令参数：**

+n　　从第 n 行开始显示

-n　　定义屏幕大小为 n 行

+/pattern 在每个档案显示前搜寻该字串（pattern），然后从该字串前两行之后开始显示

-c　　从顶部清屏，然后显示

-d　　提示“Press space to continue，’q’ to quit（按空格键继续，按 q 键退出）”，禁用响铃功能

-l　　忽略 Ctrl+l（换页）字符

-p　　通过清除窗口而不是滚屏来对文件进行换页，与-c 选项相似

-s　　把连续的多个空行显示为一行

-u　　把文件内容中的下画线去掉

**常用操作命令：**

Enter　　向下 n 行，需要定义。默认为 1 行

Ctrl+F　　向下滚动一屏

空格键　向下滚动一屏

Ctrl+B　返回上一屏

=　　输出当前行的行号

: f　　输出文件名和当前行的行号

V　　调用 vi 编辑器

!命令　调用 Shell，并执行命令

q　　退出 more

## less

less 工具也是对文件或其它输出进行分页显示的工具，应该说是 linux 正统查看文件内容的工具，功能极其强大。less 的用法比起 more 更加的有弹性。 在 more 的

时候，我们并没有办法向前面翻， 只能往后面看，但若使用了 less 时，就可以使用 [pageup] [pagedown] 等按 键的功能来往前往后翻看文件，更容易用来查看一个文件的内容！除此之外，在 less 里头可以拥有更多的搜索功能，不止可以向下搜，也可以向上搜。

```
less [参数]  文件
```

**命令参数：**

-b <缓冲区大小> 设置缓冲区的大小
-e  当文件显示结束后，自动离开
-f  强迫打开特殊文件，例如外围设备代号、目录和二进制文件
-g  只标志最后搜索的关键词
-i  忽略搜索时的大小写
-m  显示类似 more 命令的百分比
-N  显示每行的行号
-o <文件名> 将 less 输出的内容在指定文件中保存起来
-Q  不使用警告音
-s  显示连续空行为一行
-S  行过长时间将超出部分舍弃
-x <数字> 将“tab”键显示为规定的数字空格
/字符串：向下搜索“字符串”的功能
?字符串：向上搜索“字符串”的功能
n：重复前一个搜索（与 / 或 ? 有关）
N：反向重复前一个搜索（与 / 或 ? 有关）
b  向后翻一页
d  向后翻半页
h  显示帮助界面
Q  退出 less 命令
u  向前滚动半页
y  向前滚动一行
空格键 滚动一行
回车键 滚动一页
[pagedown]:  向下翻动一页
[pageup]:  向上翻动一页

## 内核栈

进程的堆栈

内核在创建进程的时候，在创建 task_struct 的同时，会为进程创建相应的堆栈。**每个进程会有两个栈：一个用户栈，存在于用户空间，一个内核栈，存在于内核空间。**

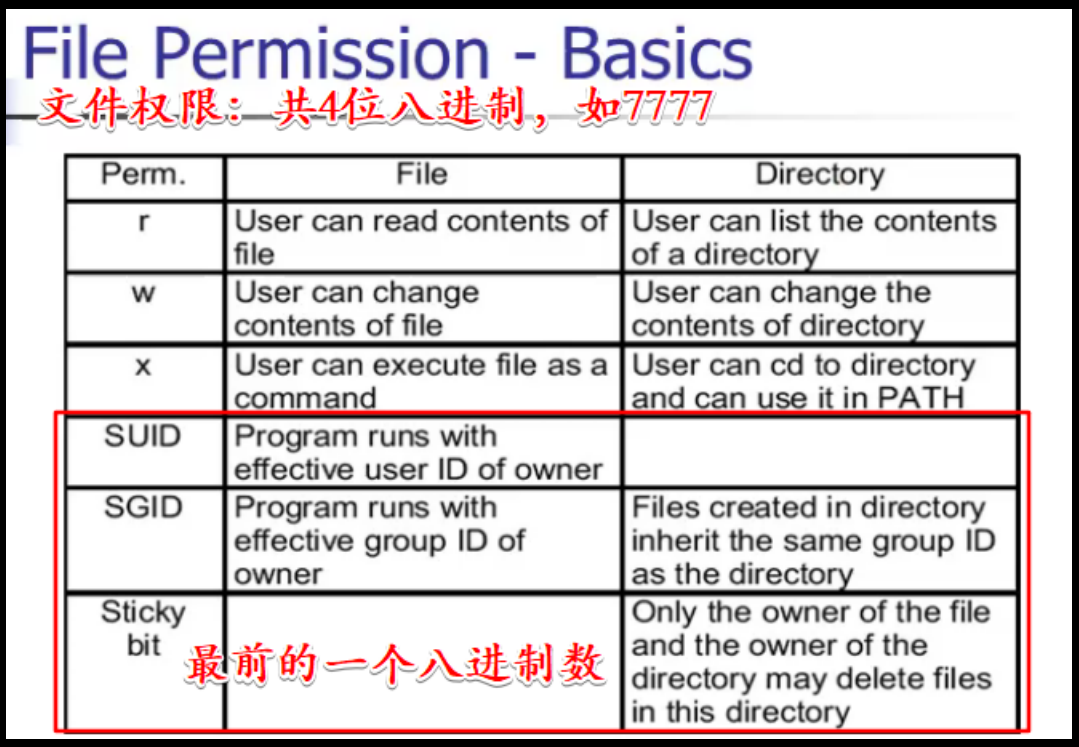
1. 当进程在用户空间运行时，cpu 堆栈指针寄存器里面的内容是用户堆栈地址，使用用户栈；
2. 当进程在内核空间时，cpu 堆栈指针寄存器里面的内容是内核栈空间地址，使用内核栈。

进程用户栈和内核栈的切换

当进程因为中断或者系统调用而陷入内核态之行时，进程所使用的堆栈也要从用户栈转到内核栈。

进程陷入内核态后，先把用户态堆栈的地址保存在内核栈之中，然后设置堆栈指针寄存器的内容为内核栈的地址，这样就完成了用户栈向内核栈的转换；当进程从内核态恢复到用户态之行时，在内核态之行的最后将保存在内核栈里面的用户栈的地址恢复到堆栈指针寄存器即可。这样就实现了内核栈和用户栈的互转。

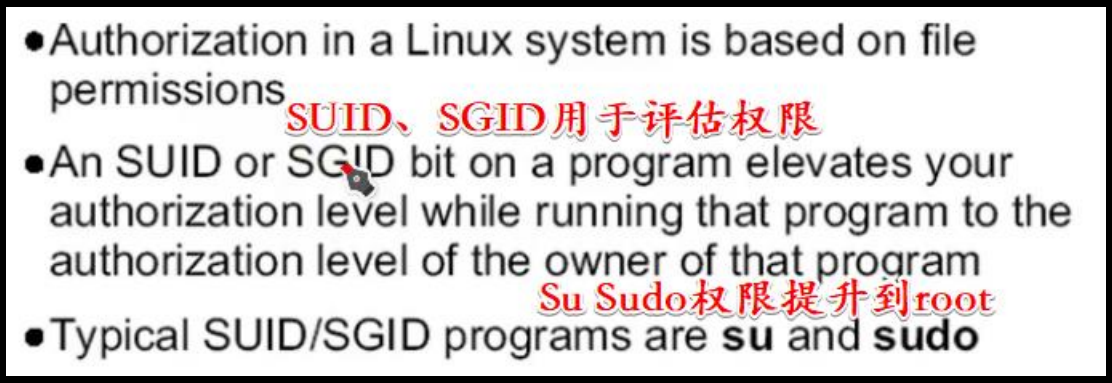
## 文件权限



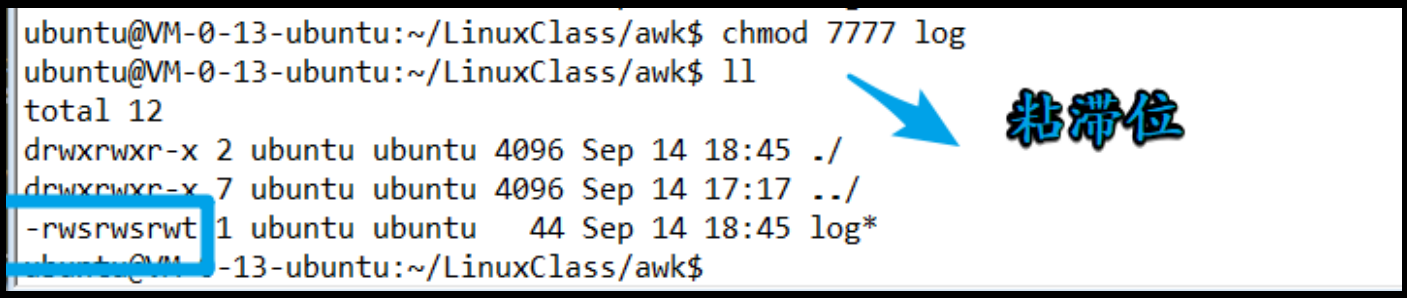

低 3 八进制：分别是文件所有者、群组用户、其他用户

Sticky Bit：粘滞位

在目录上加粘滞位，在目录下，只有文件/目录的所有者才能删除文件




Sudo 的 owner 是 root，所在组也是 root




粘滞位（Stickybit），又称粘着位，是 Unix 文件系统权限的一个旗标。最常见的用法在目录上设置粘滞位，

也只能针对目录设置，对于文件无效。则设置了粘滞位后，只有目录内文件的所有者或者 root 才可以删除或移动

该文件。如果不为目录设置粘滞位，任何具有该目录写和执行权限的用户都可以删除和移动其中的文件。实际应用中，粘滞位一般用于/tmp 目录，以防止普通用户删除或移动其他用户的文件。

粘滞位权限都是针对其他用户（ other）设置，使用chmod 命令设置目录权限时，

**/dev/vda1**

```
ubuntu@VM-16-11-ubuntu:/dev$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            885M     0  885M   0% /dev
tmpfs           184M  6.4M  177M   4% /run
/dev/vda1        50G  8.2G   39G  18% /
tmpfs           917M   24K  917M   1% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           917M     0  917M   0% /sys/fs/cgroup
tmpfs           184M     0  184M   0% /run/user/500
```

这个可是系统盘（root）

xvda 是使用 Xen DomainU 所虚拟出来之主磁碟

挂载在/

## 清理内存

echo 3 > /proc/sys/vm/drop_caches

？？？

## Swapfile 扩充内存(虚存)

2G 内存，不够用

```
ubuntu@VM-16-11-ubuntu:/$ free -m
              total        used        free      shared  buff/cache   available
Mem:           1833        1497         125           6         209         196
Swap:             0           0           0
ubuntu@VM-16-11-ubuntu:/$ free -h
              total        used        free      shared  buff/cache   available
Mem:           1.8G        1.5G        124M        6.4M        209M        194M
Swap:            0B          0B          0B
```

解决方案：

配置**交换空间（也就是虚拟内存）**

Swap 概念

Swap 分区（也称交换分区）是硬盘上的一个区域，被指定为操作系统可以临时存储数据的地方，这些数据不能再保存在 RAM 中。基本上，这使您能够增加服务器在工作“内存”中保留的信息量，但有一些注意事项，主要是当 RAM 中没有足够的空间容纳正在使用的应用程序数据时，将使用硬盘驱动器上的交换空间。 写入磁盘的信息将比保存在 RAM 中的信息慢得多，但是操作系统更愿意将应用程序数据保存在内存中，并使用交换旧数据。 总的来说，当系统的 RAM 耗尽时，将交换空间作为回落空间可能是

一个很好的安全网，可防止非 SSD 存储系统出现内存不足的情况。

检查系统信息

在开始之前，我们可以检查系统是否已经有一些可用的交换空间，可能有多个交换文件或交换分区，但通常应该是足够的。我们可以通过如下的命令来查看系统是否有交换分区：

```
sudo swapon --show
```

如果没有任何结果或者没有任何显示，说明系统当前没有可用的交换空间。Free -h 命令用来查看空闲的内存空间，其中包括交换分区的空间。

检查硬盘驱动器分区上的可用空间

为 swap 分配空间的最常见方式是使用专门用于具体某个任务的单独分， 但是，改变分区方案并不是一定可行的，我们只是可以轻松地创建驻留在现有分区上的交换文件。

在开始之前，我们应该通过输入以下命令来检查当前磁盘的使用情况：

```
df -h
```

创建 swap 文件

如果磁盘上并没有空闲的分区可以用，可以选择手动创建一个 swapfile 来充当交换空间。

据说交换空间的大小一般是内存的两倍，我现在只有 4g 的内存空间，于是当然要创建一个 8G 大小的交换空间了。 使用以下命令创建 swapfile

```
sudo fallocate -l 8G /swapfile
```

经过测试，OpenSuSE 系统要使用以下命令才能成功创建 swapfile

```
sudo dd if=/dev/zero of=/swapfile count=4096 bs=1MiB
```

```
ubuntu@VM-16-11-ubuntu:/$ ls -l swapfile
-rw-r--r-- 1 root root 4294967296 Apr 25 22:49 swapfile
```

修改 swapfile 权限

```
sudo chmod 600 /swapfile
```

激活交换空间

```
sudo mkswap /swapfile
sudo swapon /swapfile
```

```
sudo swapon /swapfile
Setting up swapspace version 1, size = 4 GiB (4294963200 bytes)
no label, UUID=05d543c5-de89-47cc-b2d9-ab56487777d8
ubuntu@VM-16-11-ubuntu:/$ sudo swapon /swapfile
ubuntu@VM-16-11-ubuntu:/$

ubuntu@VM-16-11-ubuntu:/$ sudo swapon --show
NAME      TYPE SIZE USED PRIO
/swapfile file   4G   0B   -2
ubuntu@VM-16-11-ubuntu:/$
```

成功开启了swapfile

```
ubuntu@VM-16-11-ubuntu:/$ free -h
               total        used        free      shared  buff/cache   available
Mem:           1.8G        1.5G         94M        6.4M        252M        206M
Swap:          4.0G          0B        4.0G
ubuntu@VM-16-11-ubuntu:/$
```

添加到 fstab

这样**每次开机系统就会自动吧 swapfile 挂载为交换空间**。 首先请自行备份 fstab 文件。 然后把以下配置添加到 fstab 文件末尾。

```
/swapfile none swap sw 0 0
```

或者直接使用以下命令:

```
sudo cp /etc/fstab /etc/fstab.bak
echo '/swapfile none swap sw 0 0' | sudo tee -a /etc/fstab
```

## curl

curl 是常用的命令行工具，用来请求 Web 服务器。它的名字就是客户端（client）的 URL 工具的意思。

它的功能非常强大，命令行参数多达几十种。如果熟练的话，完全可以取代 Postman 这一类的图形界面工具。

https://www.ruanyifeng.com/blog/2019/09/curl-reference.html

# 面试题

## CPU 100%怎么排查

1. 确认是程序性能问题导致，还是系统硬件瓶颈？
2. 确认引发 CPU 飙升的进程，进程 PID？
3. 确认引发飙升进程下哪个线程 CPU 占用率较高？

   Top 命令

```
top - 17:15:02 up 332 days, 13:56,  1 user,  load average: 0.03, 0.16, 0.17
Tasks:  85 total,   1 running,  84 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.3 us,  1.3 sy,  0.0 ni, 97.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1882892 total,   100212 free,   871008 used,   911672 buff/cache
KiB Swap:        0 total,        0 free,        0 used.   801184 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
  500 polkitd   20   0  540736   7184    364 S  0.3  0.4  10:31.13 polkitd
  502 dbus      20   0   60392   1508    764 S  0.3  0.1  45:42.25 dbus-daemon
  506 root      20   0   26868   1640    788 S  0.3  0.1  18:13.21 systemd-logind
 1229 mysql     20   0 1154748 291852   2352 S  0.3 15.5 176:13.87 mysqld
24040 root      20   0 2661664 383412  14324 S  0.3 20.4  35:01.60 java
    1 root      20   0   51672   2632   1384 S  0.0  0.1 102:53.63 systemd
    2 root      20   0       0      0      0 S  0.0  0.0   0:01.83 kthreadd
    3 root      20   0       0      0      0 S  0.0  0.0   7:18.96 ksoftirqd/0
```

4. jstack 打印进程下全部的线程堆栈信息，查找 CPU 占用较高的线程的堆栈详情信息？
5. 根据第四步的堆栈信息，确定出现问题的代码行号，对比程序，修改优化程序。

## No Space on this device

创建文件、目录时如果出现，表明没有可用空间

Why?

什么原因导致的？ 回忆 VFS，

1. 磁盘空间确实不足了
2. 如果 iNode 不足了，也会无法创建文件

排查，看是那种情况：

Df -h 看磁盘空间

```
ubuntu@VM-0-13-ubuntu:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            885M  4.0K  885M   1% /dev
tmpfs           184M   25M  160M  14% /run
/dev/vda1        50G   29G   19G  62% /
tmpfs           917M   24K  917M   1% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           917M     0  917M   0% /sys/fs/cgroup
tmpfs           184M     0  184M   0% /run/user/500
```

Df -i 看 iNode 使用情况

```
ubuntu@VM-0-13-ubuntu:~$ df -i
Filesystem      Inodes   IUsed   IFree IUse% Mounted on
udev            226506     400  226106    1% /dev
tmpfs           234688    1803  232885    1% /run
/dev/vda1      3276800 3276800       0  100% /
tmpfs           234688       7  234681    1% /dev/shm
tmpfs           234688       6  234682    1% /run/lock
tmpfs           234688      18  234670    1% /sys/fs/cgroup
tmpfs           234688      10  234678    1% /run/user/500
ubuntu@VM-0-13-ubuntu:~$
```

发现/dev/vda1i(系统盘) 的 iNode 已经使用完了

```
crw-rw----  1 root tty       7, 135 Feb 18  2020 vcsa7
brw-rw----  1 root disk    252,   0 Feb 18  2020 vda
brw-rw----  1 root disk    252,   1 Feb 18  2020 vda1
drwxr-xr-x  2 root root          60 Feb 14  2020 vfio/
```

什么会导致 iNode 使用完:

1. 文件过多
2. 僵尸进程

于是需要清理 iNode

解决方法: 删除无用的文件, 释放 inode。

```
root@VM-0-13-ubuntu:/var/lib/jenkins/workspace# ll
total 28
drwxr-xr-x  6 jenkins jenkins 4096 Aug 23 15:40  ./
drwxr-xr-x 22 jenkins jenkins 4096 Sep  1 04:06  ../
drwxr-xr-x  9 root    root    4096 Aug  2 18:02  OASIS_master/
drwxr-xr-x  8 root    root    4096 Aug 23 15:35 'OASIS_master@2'/
drwxr-xr-x  2 root    root    4096 Aug 23 15:35 'OASIS_master@2@tmp'/
drwxr-xr-x  2 root    root    4096 Aug 23 23:30 'OASIS_master@tmp'/
-rw-r--r--  1 root    root      26 Aug 23 15:35  workspaces.txt
root@VM-0-13-ubuntu:/var/lib/jenkins/workspace# rm -rf OASIS_master@*
root@VM-0-13-ubuntu:/var/lib/jenkins/workspace# ll
total 16
drwxr-xr-x  3 jenkins jenkins 4096 Nov 17 23:42 ./
drwxr-xr-x 22 jenkins jenkins 4096 Sep  1 04:06 ../
drwxr-xr-x  9 root    root    4096 Aug  2 18:02 OASIS_master/
-rw-r--r--  1 root    root      26 Aug 23 15:35 workspaces.txt
root@VM-0-13-ubuntu:/var/lib/jenkins/workspace# df -i
Filesystem      Inodes   IUsed  IFree IUse% Mounted on
udev            226506     400 226106    1% /dev
tmpfs           234688    1796 232892    1% /run
/dev/vda1      3276800 3276450    350  100% /
tmpfs           234688       7 234681    1% /dev/shm
tmpfs           234688       6 234682    1% /run/lock
tmpfs           234688      18 234670    1% /sys/fs/cgroup
tmpfs           234688      10 234678    1% /run/user/500
```

删除一些文件后 iNode 就有空闲的了