1019文件系统+inode

1. close关闭fd之后文件内部没有数据
2. 1,2 stdout & stderr 的区别

```c
int main()
{
    close(1);
    int fd = open("log.txt",O_WRONLY|O_CREAT|O_TRUNC,0666);
    if(fd<0)
    {
        perror("open");
        return 0;
    }

    printf("hello world: %d\n", fd); //stdout -> 1 -> log.txt -> 缓冲区

    // const char* msg = "hello world\n";
    // write(fd, msg, strlen(msg));

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

问题 输出 调试控制台 终端 端口

```
yufc@VM-12-12-centos:~/bit/1019$ make
gcc -o myfile myfile.c
yufc@VM-12-12-centos:~/bit/1019$ ./myfile
yufc@VM-12-12-centos:~/bit/1019$ cat log.txt
yufc@VM-12-12-centos:~/bit/1019$
```

这一份代码，为什么没有成功重定向到log.txt的原因我们已经可以解释了

因为磁盘文件是全缓冲

所以字符串现在在缓冲区（stdout的缓冲区）里面
我们需要fflush一下才能出来！

但不是说进程退出会刷新吗？
因为我们提前close了fd
但是，fd关闭了，数据便无法刷新了！

上面我们一直用的都是标准输出

那么什么是标准错误呢？

```c
//标准错误演示
int main()
{
    //C
    printf("hello printf\n");
    fprintf(stdout, "hello fprintf\n");
    perror("hello perror");//stderr
    //系统调用
    const char* s1 = "hello write 1\n";
    write(1,s1,strlen(s1));
    const char* s2 = "hello write 2\n";
    write(2,s2,strlen(s2));
    //C++
    std::cout<<"hello cout"<<std::endl;
    std::cerr<<"hello cerr"<<std::endl;
    return 0;
}
```

```
问题 输出 调试控制台 终端 端口
yufc@VM-12-12-centos:~/bit/1019$ make
g++ -o myfile myfile.cc
yufc@VM-12-12-centos:~/bit/1019$ ./myfile
hello printf
hello fprintf
hello perror: Success
hello write 1
hello write 2
hello cout
hello cerr
yufc@VM-12-12-centos:~/bit/1019$
```

这个代码输出这个，肯定是没问题的，接下来，我们重定向一下！

```c
//标准错误演示
int main()
{
    //C
    printf("hello printf\n");
    fprintf(stdout, "hello fprintf\n");
    perror("hello perror");//stderr
    //系统调用
    const char* s1 = "hello write 1\n";
    write(1,s1,strlen(s1));
    const char* s2 = "hello write 2\n";
    write(2,s2,strlen(s2));
    //C++
    std::cout<<"hello cout"<<std::endl;
    std::cerr<<"hello cerr"<<std::endl;
    return 0;
}
```

```
问题 输出 调试控制台 终端 端口
yufc@VM-12-12-centos:~/bit/1019$ ./myfile > log.txt
hello perror: Success
hello write 2
hello cerr
yufc@VM-12-12-centos:~/bit/1019$ cat log.txt
hello write 1
hello printf
hello fprintf
hello cout
yufc@VM-12-12-centos:~/bit/1019$
```

此时，我们发现！
重定向过后
只有往1打的过去了
往2打的还在显示器上！

1，2都是显示器文件，但是他们两个是不同的显示器文件！
我们可以认为，同一个显示器文件，被打开了两次！

一般而言，如果程序运行有可能有问题的话，建议使用stderr来打印！
如果是常规打印，建议用stdout打印

区分之后，我们可以这么写程序
我们可以把正确的和错误的分开打印
通过以下这种方式：

```
yufc@VM-12-12-centos:~/bit/1019$ ./myfile > ok.txt 2>err.txt
yufc@VM-12-12-centos:~/bit/1019$ cat ok.txt
hello write 1
hello printf
hello fprintf
hello cout
yufc@VM-12-12-centos:~/bit/1019$ cat err.txt
hello perror: Success
hello write 2
hello cerr
yufc@VM-12-12-centos:~/bit/1019$
```

这个命令的意思
2>err.txt
可以理解成，把fd为2的内容放到err.txt里面去

那如果我想把错误的和正确的都放一起呢？

```
yufc@VM-12-12-centos:~/bit/1019$ ./myfile > log.txt 2>&1
yufc@VM-12-12-centos:~/bit/1019$ cat log.txt
hello perror: Success
hello write 1
hello write 2
hello printf
hello fprintf
hello cout
hello cerr
yufc@VM-12-12-centos:~/bit/1019$
```

此时这个命令可以理解成
先把1的完成重定向到log.txt
然后把1的地址拷贝给2

所以2的不往显示器放了
往1的位置去放
也就是log.txt

还有一种常用的用法

cat < log.txt 其实就是把log.txt的内容重定向给cat
等价于
cat log.txt 这个我们讲过了

所以
cat < log.txt > back.txt (cat 后面的 < 可以不写)
相当于把log.txt的内容交给cat
那么此时log.txt的内容准备往显示器打印
但是此时再次重定向到back.txt
所以最终就是
log.txt的内容完成一次拷贝到back.txt上！

但是这样是不行的

```
yufc@VM-12-12-centos:~/bit/1019$ log.txt > back.txt
bash: log.txt: command not found
yufc@VM-12-12-centos:~/bit/1019$
```

```
yufc@VM-12-12-centos:~/bit/1019$ rm back.txt
yufc@VM-12-12-centos:~/bit/1019$ cat log.txt > back.txt
yufc@VM-12-12-centos:~/bit/1019$ cat back.txt
hello perror: Success
hello write 1
hello write 2
hello printf
hello fprintf
hello cout
hello cerr
yufc@VM-12-12-centos:~/bit/1019$
```

perror向2号文件描述符写入！

C语言还有一个东西
叫做　　errno.h

里面有个变量叫做errno

```
       device file, the O_TRUNC flag is ignored.  Otherwise the effect of O_TRUNC is unspecified.

       Some of these optional flags can be altered using fcntl(2) after the file has been opened.
这个是open系统调用的手册
       creat() is equivalent to open() with flags equal to O_CREAT|O_WRONLY|O_TRUNC.

RETURN VALUE
       open() and creat() return the new file descriptor, or -1 if an error occurred (in which case,  errno
       is set appropriately).

ERRORS
       EACCES The  requested  access  to the file is not allowed, or search permission is denied for one of
              the directories in the path prefix of pathname, or the file  did  not  exist  yet  and  write
              access to the parent directory is not allowed.  (See also path_resolution(7).)

       EDQUOT Where  O_CREAT  is specified, the file does not exist, and the user's quota of disk blocks or
              inodes on the file system has been exhausted.

       EEXIST pathname already exists and O_CREAT and O_EXCL were used.

                                          your accessible address space.


                                   o complete an open of a slow device (e.g., a FIFO; see  fifo(7)),  the
```

问题　输出　调试控制台　终端　端口

```
yufc@VM-12-12-centos:~/bit/1019$ log.txt > back.txt
bash: log.txt: command not found
yufc@VM-12-12-centos:~/bit/1019$ man 2 open
yufc@VM-12-12-centos:~/bit/1019$ cat log.txt
hello perror: Success
hello write 1
hello write 2
hello printf
hello fprintf
hello cout
hello cerr
yufc@VM-12-12-centos:~/bit/1019$
```

errno变量的值，控制的是这个

那么还有一个函数
strerror
这个函数的结果
即使perror函数结果
冒号后面的那部分！

学习文件系统要掌握的背景知识：

1. 我们以前学习的都是被打开的文件，那们有没有没有被打开的文件？当然存在，在磁盘里

2. 我们学习磁盘级别的文件，我们侧重点在哪里呢？

单个文件的角度 --- 这个文件在哪里？这个文件多大？这个文件的其他属性是什么？
站在系统的角度 --- 一共有多少个文件？各自属性在哪里？如何快速找到？我还可以存储多少个文件？如何快速找到制定的文件？

3. 如何进行对磁盘文件进行分门别类的存储，又来支持更好的存取？

所以，我们先要了解磁盘

内存 --- 是一种掉电易失存储介质，即断电，数据就没了

磁盘 --- 永久性存储介质 --- SSD，U盘，flash卡，光盘，磁带

磁盘是一个外设 + 还是我们计算机中唯一个机械设备 --- 慢！

我们可以搜一下磁盘寻道道过程

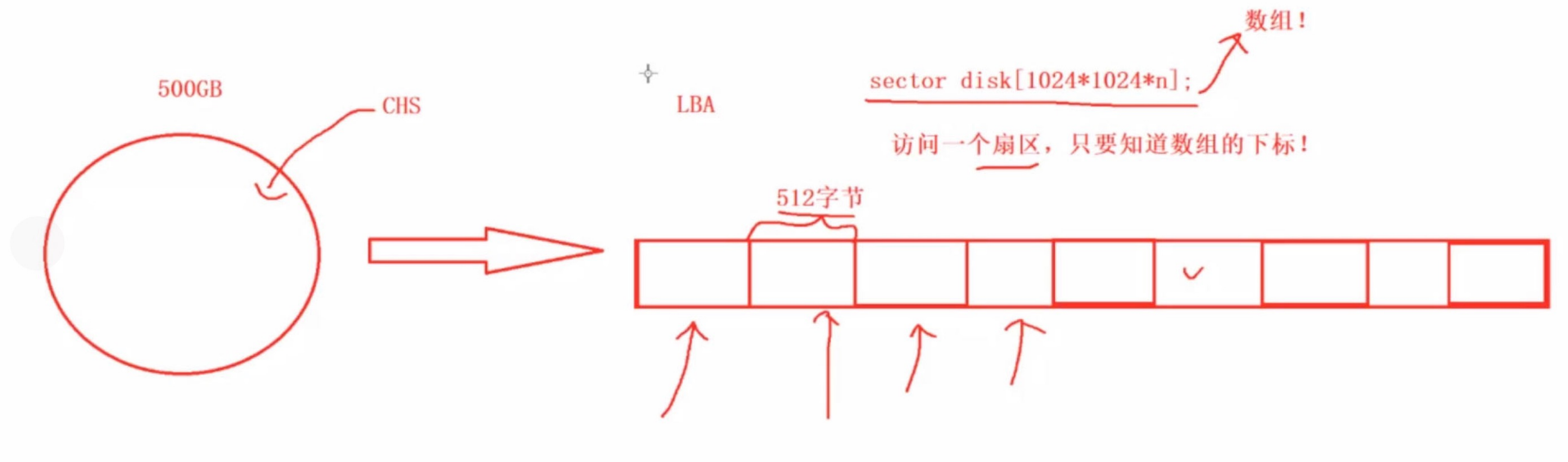
蛋哥对磁盘结构的讲解
没有记笔记
具体可以看蛋哥的板书

CHS寻址
1. 在哪一个面上（确定哪一个磁头）
2. 在哪一个磁道上（柱面）上
3. 在哪一个扇区上

一个扇区，512字节
有些是4kb，但是共识是512字节

磁盘的抽象结构

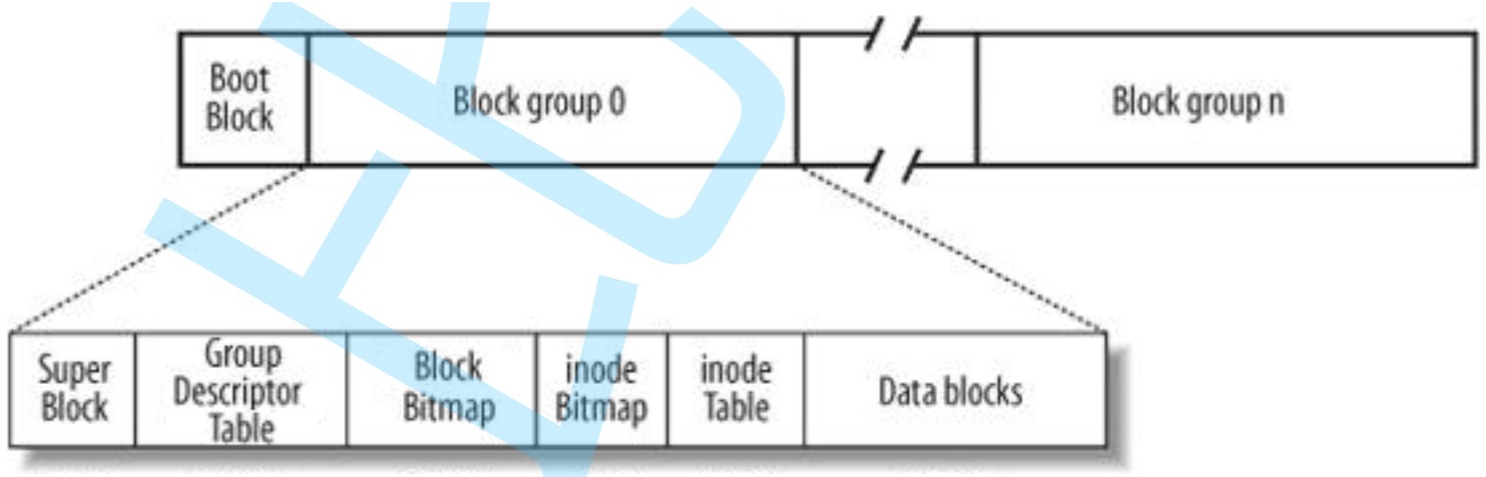
可以把，圆形的磁盘物理结构
抽象成线性结构



这个叫做LBA的寻址方式

我们把LBA转化成CHS就可以找到地址了！

一个磁盘很大！
我们可以对他们进行分区！
分区之后
还是很大，继续分区
就到了我们课件上面的样子



Linux ext2文件系统，上图为磁盘文件系统图（内核内存映像肯定有所不同），磁盘是典型的块设备，硬盘分区被划分为一个个的block。一个block的大小是由格式化的时候确定的，并且不可以更改。例如mke2fs的-b选项可以设定block大小为1024、2048或4096字节。而上图中启动块（Boot Block）的大小是确定的，

- Block Group：ext2文件系统会根据分区的大小划分为数个Block Group。而每个Block Group都有着相同的结构组成。政府管理各区的例子
- 超级块（Super Block）：存放文件系统本身的结构信息。记录的信息主要有：bolck 和 inode的总量，未使用的block和inode的数量，一个block和inode的大小，最近一次挂载的时间，最近一次写入数据的时间，最近一次检验磁盘的时间等其他文件系统的相关信息。Super Block的信息被破坏，可以说整个文件系统结构就被破坏了
- GDT，Group Descriptor Table：块组描述符，描述块组属性信息，有兴趣的同学可以在了解一下
- 块位图（Block Bitmap）：Block Bitmap中记录着Data Block中哪个数据块已经被占用，哪个数据块没有被占用
- inode位图（inode Bitmap）：每个bit表示一个inode是否空闲可用。
- i节点表:存放文件属性 如 文件大小，所有者，最近修改时间等
- 数据区：存放文件内容

将属性和数据分开存放的想法看起来很简单，但实际上是如何工作的呢？我们通过touch一个新文件来看看如何工作。

- 超级块（Super Block）：存放文件系统本身的结构信息。记录的信息主要有：bolck 和 inode的总量，未使用的block和inode的数量，一个block和inode的大小，最近一次挂载的时间，最近一次写入数据的时间，最近一次检验磁盘的时间等其他文件系统的相关信息。Super Block的信息被破坏，可以说整个文件系统结构就被破坏了

Super Block记录的是整个大分区的重要信息：哪些是否被占用等
但为什么在每个组都有SuperBlock呢

因为避免因为物理原因，SuperBlock被破坏
所以每个组都备份一个

一个重要概念：
虽然磁盘的基本单位是扇区（512字节）
但是操作系统（文件系统）和磁盘进行IO的基本单位是：4kb

为什么？
1. 太小了，有可能会导致多次的IO，进而导致效率降低
2. 如果OS使用和磁盘一样的大小，万一磁盘基本大小变了的话，OS的源代码要不要改呢？　　所以硬件和软件（OS）进行解耦

DataBlock：

可以理解成多个4kb的集合

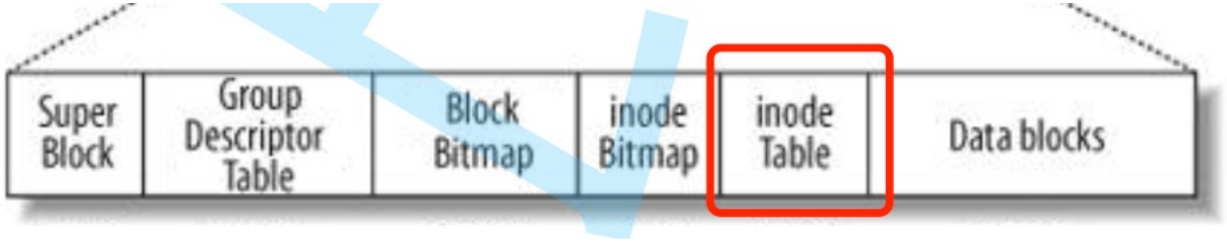
文件 = 内容 + 属性

Linux在磁盘上存储文件的时候，将内容和属性是分开存储的！
DataBlock存的都是内容！

```
yufc@VM-12-12-centos:~$ ls -li
total 20
788529 drwxrwxr-x 14 yufc yufc 4096 Feb  6 23:23 bit
660503 drwxrwxr-x  2 yufc yufc 4096 Jan 10 21:20 coding_draft
661238 drwxrwxr-x  3 yufc yufc 4096 Jan 26 16:49 linux_kernel
660018 drwxrwxr-x  3 yufc yufc 4096 Jan  9 23:35 project
659938 drwxrwxr-x  3 yufc yufc 4096 Jan 20 00:48 SYSU
yufc@VM-12-12-centos:~$
```

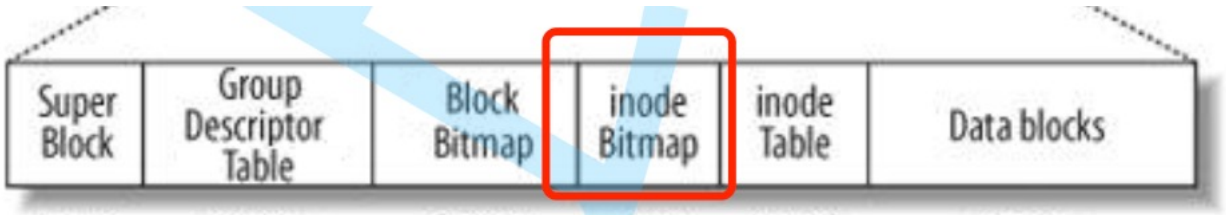
前面这些是inode

一般而言：
一个文件，一个inode，一个inode编号

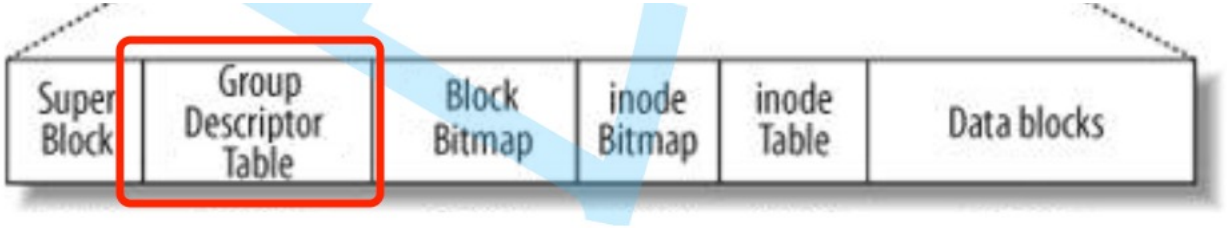


Inode是一个大小为128字节的空间，保存的是对应文件的属性
Inode Table，保存了所有文件的inode空间的集合，因为需要标识唯一性，每一个inode块都要有一个inode编号

BlockBitMap：假设有10000+个blocks，10000+个比特位：比特位和特定的block是一一对应的！其中比特位为1，代表该block被占用，否则表示可用！



inodeBitMap：假设有10000+个inode节点，就有10000+个比特位，比特位和特定的inode是一一对应的
其中bitmap中比特位为1，表示该inode被占用，否则表示可用！



GDT：
快组描述符
这个块组多大，已经使用了多少？
有多少个inode？
已经占用了多少，还剩多少？
一个多少block？
……

我们讲的块组分割成为上面的内容，宾切写入相关的管理数据 -> 每一个块组都这么干 -> 整个分区就被写入了文件系统信息

这个过程叫做格式化！

一个文件“只”对应一个inode属性节点，inode编号

一个文件只能对应一个block吗？不一定了！

那么可以引出下面的问题：

1. 怎么判断哪些block属于同一个文件？

2. 找到文件，只要找到文件对应的inode，就能找到文件的属性集合，课室，文件的内容呢？