文件系统

文件存储

首先了解如下文件存储相关概念： inode、 dentry、数据存储、文件系统。

inode

其本质为结构体，存储文件的属性信息。如：权限、类型、大小、时间、用户、盘块位置 ，也叫作文件属性管理结构，大多数的 inode 都存储在磁盘上。

少量常用、近期使用的 inode 会被缓存到内存中。

dentry

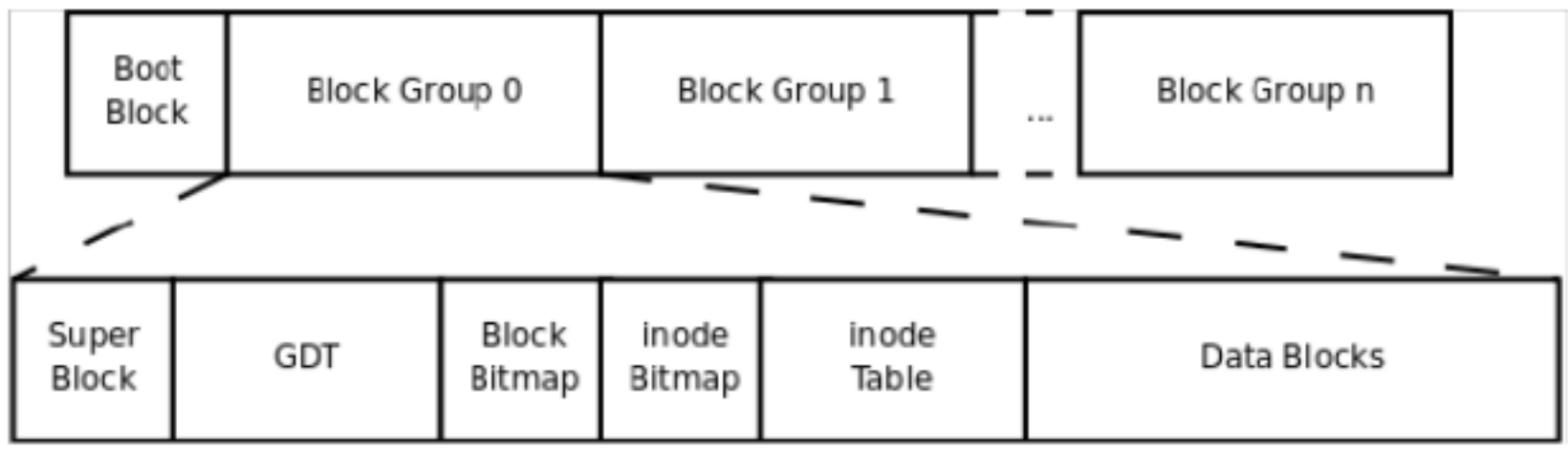
目录项， 其本质依然是结构体， 重要成员变量有两个 {文件名， inode，...}，而文件内容 (data)保存在磁盘盘块中。

文件系统

文件系统是，一组规则，规定对文件的存储及读取的一般方法。文件系统在磁盘格式化过程中指定。

常见的文件系统有： fat32 ntfs exfat ext2 、ext3 、ext4

ext2 文件系统



ext2 文件系统

我们知道，一个磁盘可以划分成多个分区，每个分区必须先用格式化工具（例如某种 mkfs 命令）格式化成某种格式的文件系统，然后才能存储文件，格式化的过程会在磁盘上写一些管理存储布局的信息。下图是一个磁盘分区格式化成 ext2 文件系统后的存储布局。

文件系统中存储的最小单位是块 （Block），一个块究竟多大是在格式化时确定的，例如 mke2fs 的 -b 选项可以设定块大小为 1024、2048 或 4096 字节。而上图中 启动块 （BootBlock）的大小是确定的，就是 1KB，启动块是由 PC 标准规定的， 用来存储磁盘分区信息和启动信息， 任何文件系统都不能使用启动块。 启动块之后才是 ext2 文件系统的开始， ext2 文件系统将整个分区划成若干个同样大小的块组（ Block Group），每个块组都由以下部分组成。

超级块 （Super Block）描述整个分区的文件系统信息，例如块大小 (1k、 2k、 4K,，)、文件系统版本号、上次 mount 的时间等等。超级块在每个块组的开头都有一份拷贝。

块组描述符表 （GDT，Group Descriptor Table）由很多块组描述符组成，整个分区分成多少个块组就对应有多少

个块组描述符。 每个块组描述符 （Group Descriptor ）存储一个块组的描述信息， 例如在这个块组中从哪里开始是 inode 表，从哪里开始是数据块，空闲的 inode 和数据块还有多少个等等。和超级块类似，块组描述符表在每个块组的开头也都有一份拷贝，这些信息是非常重要的，一旦超级块意外损坏就会丢失整个分区的数据，一旦块组描述符意外损坏就会丢失整个块组的数据，因此它们都有多份拷贝。通常内核只用到第 0 个块组中的拷贝，当执行 e2fsck 检查文件系统一致性时，第 0 个块组中的超级块和块组描述符表就会拷贝到其它块组，这样当第 0 个块组的开头意外损坏时就可以用其它拷贝来恢复，从而减少损失。

块位图 （Block Bitmap ）一个块组中的块是这样利用的： 数据块存储所有文件的数据， 比如某个分区的块大小是 1024 字节，某个文件是 2049 字节，那么就需要三个数据块来存，即使第三个块只存了一个字节也需要占用一个整块； 超级块、块组描述符表、块位图、 **inode** 位图、 **inode** 表这几部分存储该块组的描述信息。 那么如何知道哪些块已经用来存储文件数据或其它描述信息， 哪些块仍然空闲可用呢？块位图就是用来描述整个块组中哪些块已用哪些块空闲的，它本身占一个块，其中的每个 bit 代表本块组中的一个块，这个 bit 为 1 表示该块已用，这个 bit 为 0 表示该块空闲可用。

为什么用 df 命令统计整个磁盘的已用空间非常快呢？因为只需要查看每个块组的块位图即可， 而不需要搜遍整个分区。相反，用 du 命令查看一个较大目录的已用空间就非常慢，因为不可避免地要搜遍整个目录的所有文件。

与此相联系的另一个问题是： 在格式化一个分区时究竟会划出多少个块组呢？主要的限制在于 块位图本身必须只占一个块 。用 mke2fs 格式化时默认块大小是 1024 字节，可以用 -b 参数指定块大小， 现在设块大小指定为 b 字节，那么一个块可以有 8b 个 bit，这样大小的一个块位图就可以表示 8b 个块的占用情况，因此一个块组最多可以有 8b 个块，如果整个分区有 s 个块，那么就可以有 s/(8b) 个块组。格式化时可以用 -g 参数指定一个块组有多少个块，但是通常不需要手动指定， mke2fs 工具会计算出最优的数值。

inode 位图 （inode Bitmap ）和块位图类似，本身占一个块，其中每个 bit 表示一个 inode 是否空闲可用。

inode 表（inode Table）我们知道，一个文件除了数据需要存储之外，一些描述信息也需要存储，例如文件类型（常规、目录、符号链接等） ，权限，文件大小，创建 / 修改 / 访问时间等，也就是 ls -l 命令看到的那些信息，这些信息存在 inode 中而不是数据块中。每个文件都有一个 inode，一个块组中的所有 inode 组成了 inode 表。 inode 表占多少个块在格式化时就要决定并写入块组描述符中， mke2fs 格式化工具的默认策略是一个块组有多少个 8KB 就分配多少个 inode。由于数据块占了整个块组的绝大部分，也可以近似认为数据块有多少个 8KB 就分配多少个 inode，换句话说，如果平均每个文件的大小是 8KB，当分区存满的时候 inode 表会得到比较充分的利用，数据块也不浪费。如果这个分区存的都是很大的文件 （比如电影），则数据块用完的时候 inode 会有一些浪费， 如果这个分区存的都是很小的文件 （比如源代码），则有可能数据块还没用完 inode 就已经用完了，数据块可能有很大的浪费。 如果用户在格式化时能够对这个分区以后要存储的文件大小做一个预测， 也可以用 mke2fs 的 -i 参数手动指定每多少个字节分配一个 inode。

inode 表中每个 **inode** 的大小： **ext2**、**ext3** 中 **128** 字节， **ext4** 中 **256** 字节 。

数据块（ Data Block）根据不同的文件类型有以下几种情况：

对于常规文件，文件的数据存储在数据块中。

对于目录，该目录下的所有文件名和目录名存储在数据块中，注意文件名保存在它所在目录的数据块中，除文件名之外， ls -l 命令看到的其它信息都保存在该文件的 inode 中。注意这个概念：目录也是一种文件，是一种特殊类型的文件。

对于符号链接，如果目标路径名较短则直接保存在 inode 中以便更快地查找，如果目标路径名较长则分配一个数据块来保存。

设备文件、 FIFO和 socket 等特殊文件没有数据块，设备文件的主设备号和次设备号保存在 inode 中。

数据块寻址：

如果一个文件有多个数据块，这些数据块很可能不是连续存放的。这些数据块通过 inode 中的索引项 Block 来找到。

这样的索引项一共有 15 个, Block[0]--Block[14] , 每个索引项占 4 字节。前 12 个索引项都表示块的编号, 如 Block[0] 保存 27，表示第 27 个块是该文件的数据块。

如果块的大小是 1KB，这种方法可以表示从 0-12KB 的文件。 如果剩下的三个索引项也这么用, 那就只能表示最大 15KB 的文件了。这是远远不够的。

剩下的 3 个索引项 Block[12]、Block[13]、 Block[14] 都是间接索引。假设块的大小为 b，那么一个间接寻址块中可以存放 b/4 个索引项，指向 b/4 个数据块。

如： b=1KB，那么 Block[0]--Block[12] 都用上，最大可以表示 1024/4 + 12 = 268K 大小的文件。虽然大很多，但仍然不够用。

于是有了二级间接寻址块 Block[13]，三级间接寻址块 Block[14]。这样 1K 的块最大可以表示 16.06GB 大小的文件。

stat 命令中： “ 块 ” -->物理块 (512B)的个数；“ IO 块 ” -->逻辑块的大小

文件操作

stat 函数

获取文件属性, (从 inode 结构体中获取)

int stat(const char *path, struct stat *buf); 成返回 0；失败返回 -1 设置 errno 为恰当值。

参数 1：文件名

参数 2：inode 结构体指针 (传出参数)

文件属性将通过传出参数返回给调用者。

练习：使用 stat 函数查看文件属性 【stat.c】

lstat 函数

int lstat(const char *path, struct stat *buf); 成返回 0；失败返回 -1 设置 errno 为恰当值。

练习：给定文件名，判断文件类型。 【get_file_type.c 】

文件类型判断方法： st_mode 取高 4 位。但应使用宏函数：

S_ISREG(m) is it a regular file?

S_ISDIR(m) directory?

S_ISCHR(m) character device?

S_ISBLK(m) block device?

S_ISFIFO(m) FIFO (named pipe)?

S_ISLNK(m) symbolic link? (Not in POSIX.1-1996.)

S_ISSOCK(m) socket? (Not in POSIX.1-1996.)

穿透符号链接： stat：会； lstat ：不会

特殊权限位：

包含三个二进制位。依次是：设置组　　ID 位 setGID；设置用户　ID 位 setID；黏住位　sticky

黏住位

早起计算机内存紧，只有精要的常用的程序可以常驻物理内存，剩下的要暂存磁盘中。当内存不够用的时候会将该部分程序存回磁盘，腾出内存空间。若文件设置了黏住位，那么即使在内存比较吃紧的情况下，也不会将该文件回存到磁盘上。由于现阶段操作系统的虚拟内存管理分页算法完善。该功能已经被废弃。

但我们仍然 可以对目录设置黏住位　。被设置了该位的目录，其内部文件只有：

①超级管理员

②该目录所有者

③该文件的所有者

以上三种用户有权限做删除、修改操作。其他用户可以读、创建但不能随意删除。

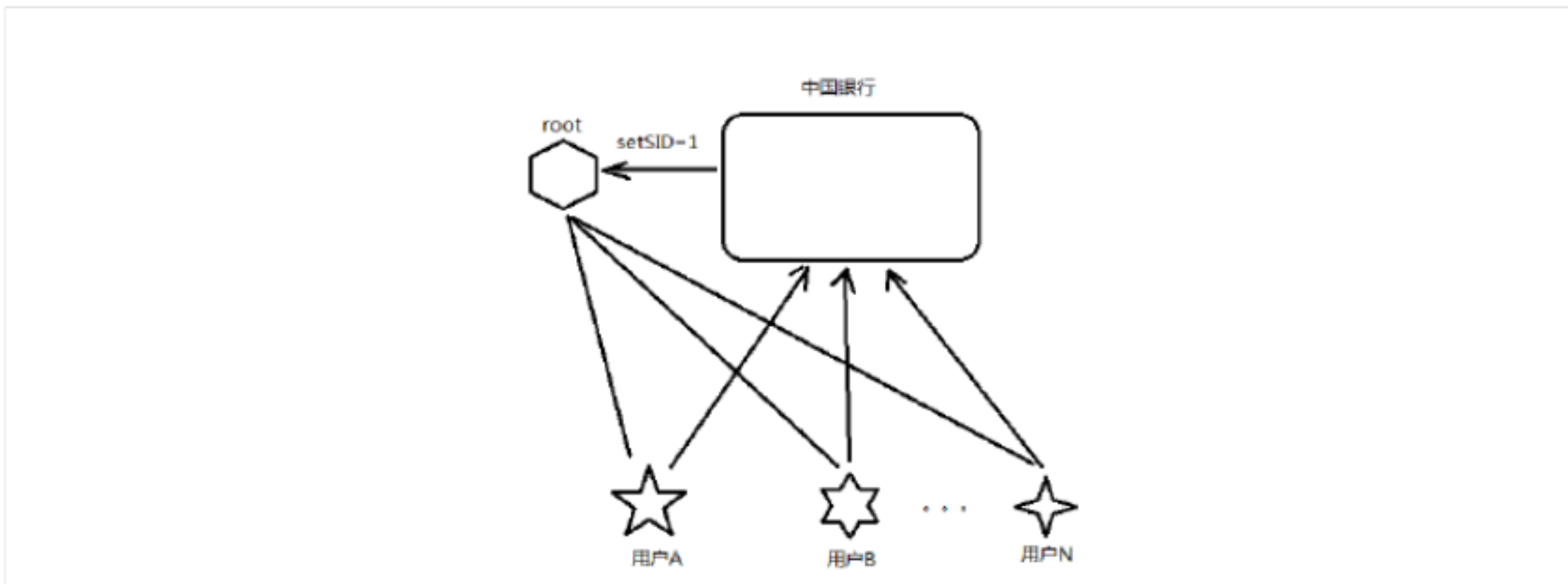
setUID位

进程有两个　ID：EID(有效用户　ID)，表示进程履行哪个用户的权限。

UID(实际用户　ID)，表示进程实际属于哪个用户。

多数情况下，　EID和 UID 相同。但是，当文件的　setID 被设置后两个　ID 则有可能不一样。

例如：当进程执行一个　root 用户的文件，若该文件的　setID 位被设置为　1，那么执行该文件时，进程的　UID 不变。 EID 变为 root，表示进程开始履行　root 用户权限。



setGID 位于 setID 相类似。

access函数

测试指定文件是否存在　/拥有某种权限。

int access(const char *pathname, int mode);　成功 /具备该权限：　0；失败 /不具备　-1 设置 errno 为相应值。

参数 2：R_OK、W_OK、X_OK

通常使用 access函数来测试某个文件是否存在。 F_OK

chmod 函数

修改文件的访问权限

int chmod(const char *path, mode_t mode); 成功： 0；失败： -1 设置 errno 为相应值

int fchmod(int fd, mode_t mode);

truncate 函数

截断文件长度成指定长度。常用来拓展文件大小，代替 lseek。

int truncate(const char *path, off_t length); 成功： 0；失败： -1 设置 errno 为相应值

int ftruncate(int fd, off_t length);

link 函数

思考，为什么目录项要游离于 inode 之外，画蛇添足般的将文件名单独存储呢？？这样的存储方式有什么样的好处呢？

其目的是为了实现文件共享。 Linux 允许多个目录项共享一个 inode，即共享盘块 (data)。不同文件名，在人类眼中将它理解成两个文件，但是在内核眼里是同一个文件。

link 函数，可以为已经存在的文件创建目录项 (硬链接)。

int link(const char *oldpath, const char *newpath); 成功： 0；失败： -1 设置 errno 为相应值

注意：由于两个参数可以使用 “ 相对 /绝对路径 +文件名 ” 的方式来指定，所以易出错。

如： link("../abc/a.c", "../ioc/b.c") 若 a.c， b.c 都对，但 abc， ioc 目录不存在也会失败。

mv 命令既是修改了目录项，而并不修改文件本身。

unlink 函数

删除一个文件的目录项；

int unlink(const char *pathname); 成功： 0；失败： -1 设置 errno 为相应值

练习：编程实现 mv 命令的改名操作 【imp_mv.c】

注意 Linux 下删除文件的机制：不断将 st_nlink -1，直至减到 0 为止。无目录项对应的文件，将会被操作系统择机释放。 (具体时间由系统内部调度算法决定)

因此，我们删除文件，从某种意义上说， 只是让文件具备了被释放的条件。

unlink 函数的特征： 清除文件时， 如果文件的硬链接数到 0 了，没有 dentry 对应，但该文件仍不会马上被释放。要等到所有打开该文件的进程关闭该文件，系统才会挑时间将该文件释放掉。 【unlink_exe.c】

隐式回收

当进程结束运行时，所有该进程打开的文件会被关闭，申请的内存空间会被释放。系统的这一特性称之为隐式

回收系统资源。

symlink 函数

创建一个符号链接

int symlink(const char *oldpath, const char *newpath);　　成功： 0；失败： -1 设置 errno 为相应值

readlink 函数

读取符号链接文件本身内容，得到链接所指向的文件名。

ssize_t readlink(const char *path, char *buf, size_t bufsiz);　成功：返回实际读到的字节数；失败：　-1 设置 errno 为相应值。

rename 函数

重命名一个文件。

int rename(const char *oldpath, const char *newpath);　成功： 0；失败： -1 设置 errno 为相应值

目录操作

工作目录：“ ./ ”代表当前目录，指的是进程当前的工作目录，默认是进程所执行的程序所在的目录位置。

getcwd 函数

获取进程当前工作目录　　(卷 3，标库函数)

char *getcwd(char *buf, size_t size);　成功： buf 中保存当前进程工作目录位置。失败返回　NULL。

chdir 函数

改变当前进程的工作目录

int chdir(const char *path);　成功： 0；失败： -1 设置 errno 为相应值

练习：获取及修改当前进程的工作目录，并打印至屏幕。　　【imp_cd.c】

文件、目录权限

注意： 目录文件也是 “ 文件 ”。其文件内容是该目录下所有子文件的目录项　dentry 。可以尝试用 vim 打开一个目录。

	r	w	x
文件	文件的内容可以被查看 cat 、 more、 less ，	内容可以被修改 vi 、 > ，	可以运行产生一个进程 ./ 文件名
目录	目录可以被浏览 ls 、 tree ，	创建、删除、修改文件 mv、 touch 、 mkdir ，	可以被打开、进入 cd

目录设置黏住位：若有　w 权限，创建不变，删除、修改只能由　root 、目录所有者、文件所有者操作。

opendir 函数

根据传入的目录名打开一个目录　(库函数)　DIR *　类似于　FILE *

DIR *opendir(const char *name);　成功返回指向该目录结构体指针，失败返回　NULL

参数支持相对路径、绝对路径两种方式：例如：打开当前目录：①　getcwd() , opendir()　②　opendir(".");

closedir函数

关闭打开的目录

int closedir(DIR *dirp);　成功： 0；失败： -1 设置 errno 为相应值

readdir 函数

读取目录　(库函数)

struct dirent *readdir(DIR *dirp);　成功返回目录项结构体指针；失败返回　NULL 设置 errno 为相应值

需注意返回值，读取数据结束时也返回　NULL 值，所以应借助　errno 进一步加以区分。

struct 结构体：

```
struct dirent {
ino_t          d_ino;inode 编号
off_t          d_off;
unsigned short d_reclen;      文件名有效长度
unsigned char  d_type;        类型 (vim 打开看到的类似 @*/ 等)
char           d_name[256]; 文件名
};
```

其成员变量重点记忆两个：　d_ino 、d_name 。实际应用中只使用到　d_name 。

练习 1：实现简单的 ls 功能。　【imp_ls.c】

练习 2：实现 ls 不打印隐藏文件。每　5 个文件换一个行显示。　【imp_ls2.c】

拓展 1：实现 ls -a -l 功能。

拓展 2：统计目录及其子目录中的普通文件的个数

rewinddir 函数

回卷目录读写位置至起始。

void rewinddir(DIR *dirp);　返回值：无。

telldir/seekdir 函数

获取目录读写位置

long telldir(DIR *dirp); 成功：与 dirp 相关的目录当前读写位置。失败 -1，设置 errno

修改目录读写位置

void seekdir(DIR *dirp, long loc); 返回值：无

参数 loc 一般由 telldir 函数的返回值来决定。

递归遍历目录

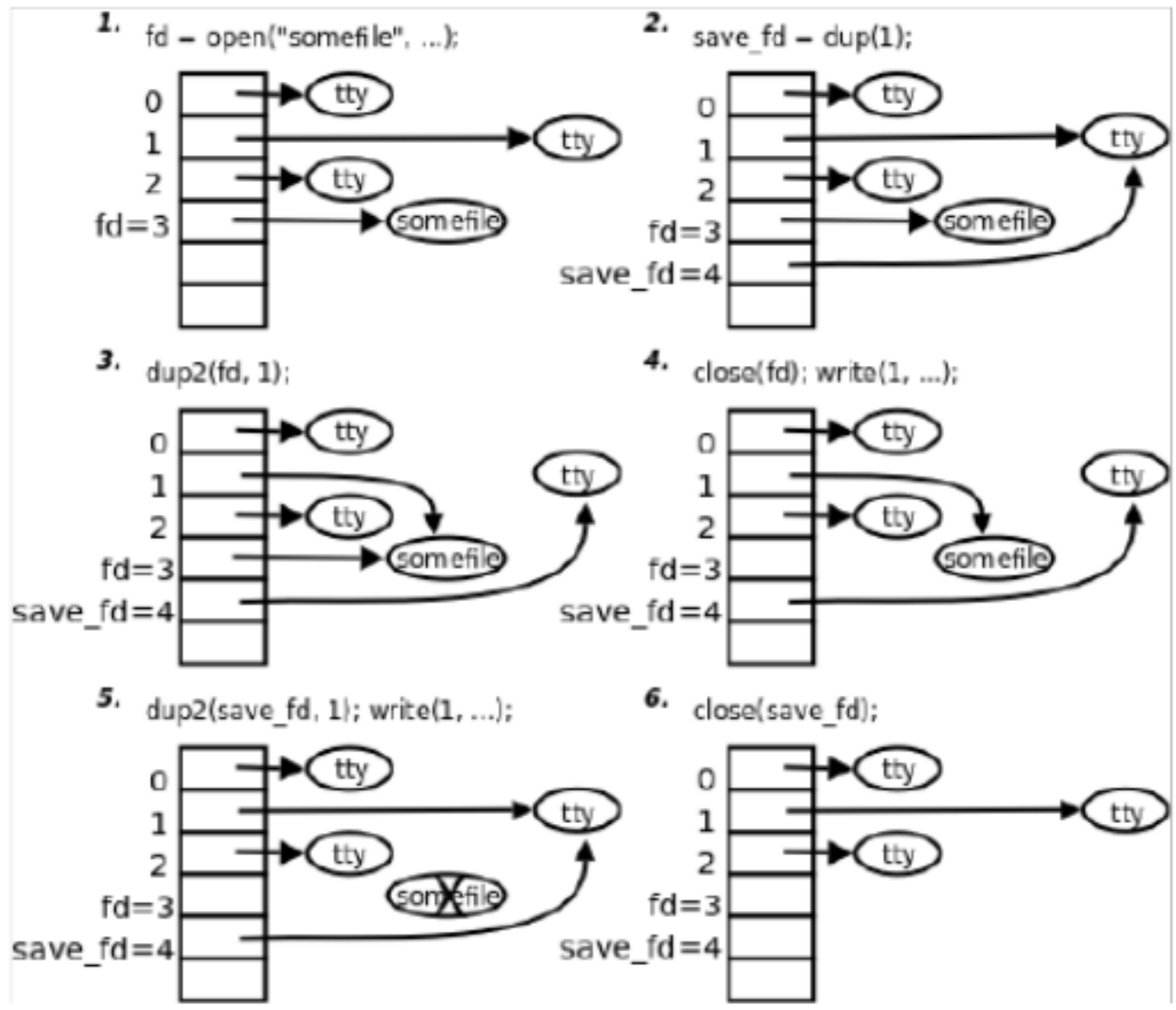
查询指定目录，递归列出目录中文件，同时显示文件大小。 【ls_R.c】

重定向

dup 和 dup2 函数

int dup(int oldfd); 成功：返回一个新文件描述符；失败： -1 设置 errno 为相应值

int dup2(int oldfd, int newfd);



重定向示

记忆方法两种：

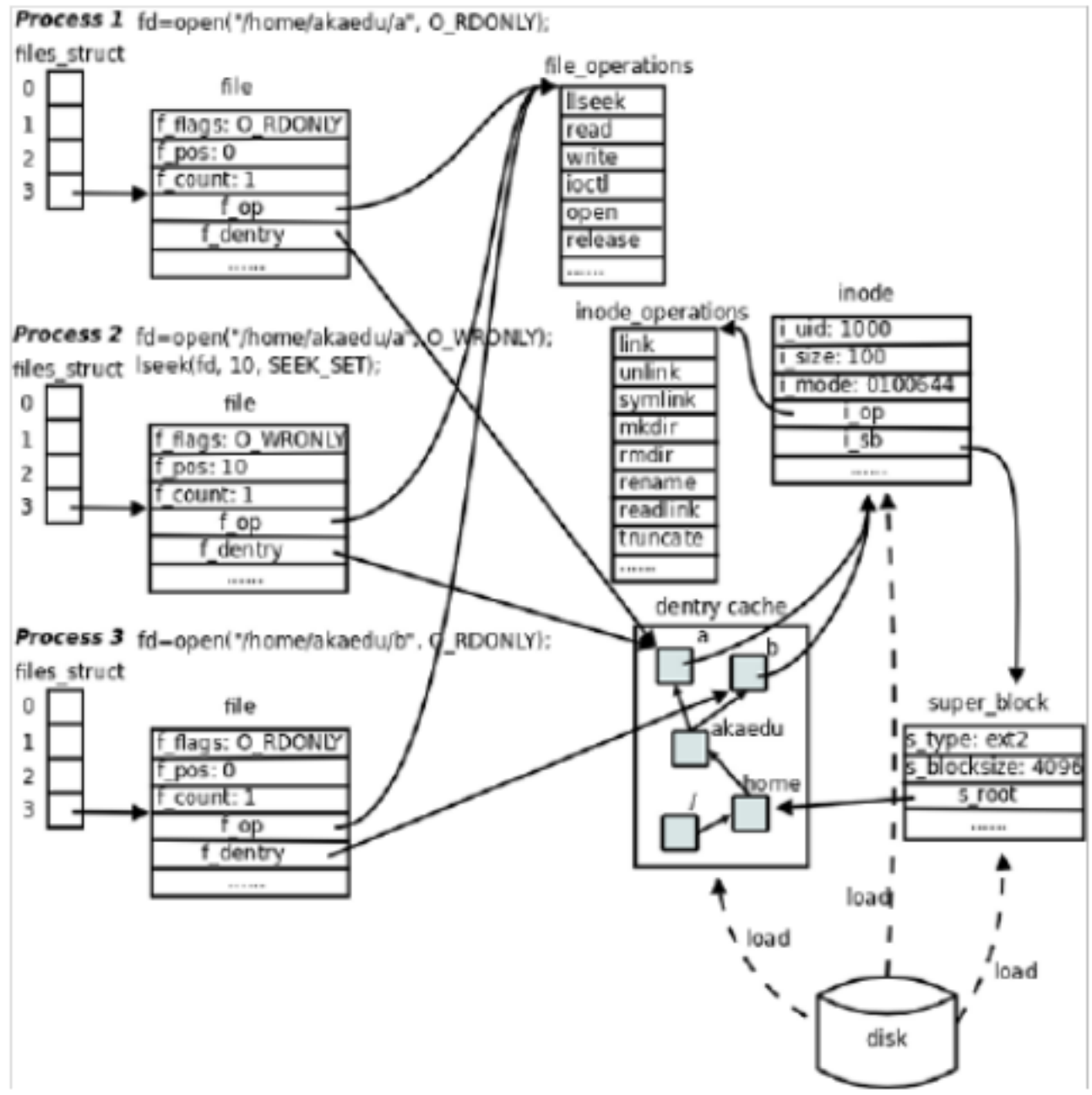
1. 文件描述符的本质角度理解记忆。
2. 从函数原型及使用角度，反向记忆。

练习：借助 dup 函数编写 mycat 程序，实现 cat file1 > file2 命令相似功能。 【mycat.c】

VFS虚拟文件系统

Linux 支持各种各样的文件系统格式，如 ext2、ext3、reiserfs、FAT、NTFS、iso9660 等等，不同的磁盘分区、光盘或其它存储设备都有不同的文件系统格式，然而这些文件系统都可以 mount 到某个目录下，使我们看到一个统一的目录树，各种文件系统上的目录和文件我们用 ls 命令看起来是一样的，读写操作用起来也都是一样的，这是怎么做到的呢？ Linux 内核在各种不同的文件系统格式之上做了一个抽象层，使得文件、目录、读写访问等概念成为抽

象层的概念，因此各种文件系统看起来用起来都一样，这个抽象层称为虚拟文件系统（VFS，Virtual Filesystem）。



VFS虚拟文件系统