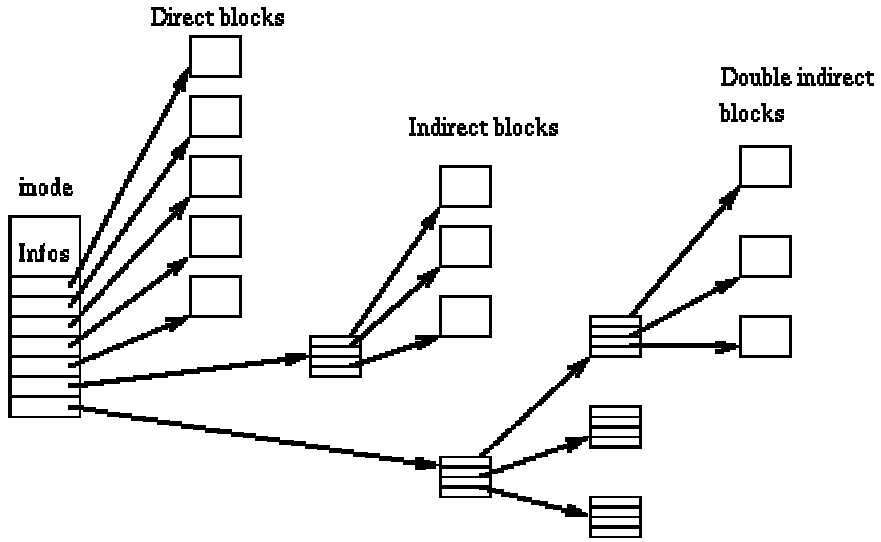
Linux 001 文件系统部分代码分析（I）初稿

Rayx 整理（rayx@zju.edu.cn　qq:33771755　bbs:rayx(tsinghua,zju)）

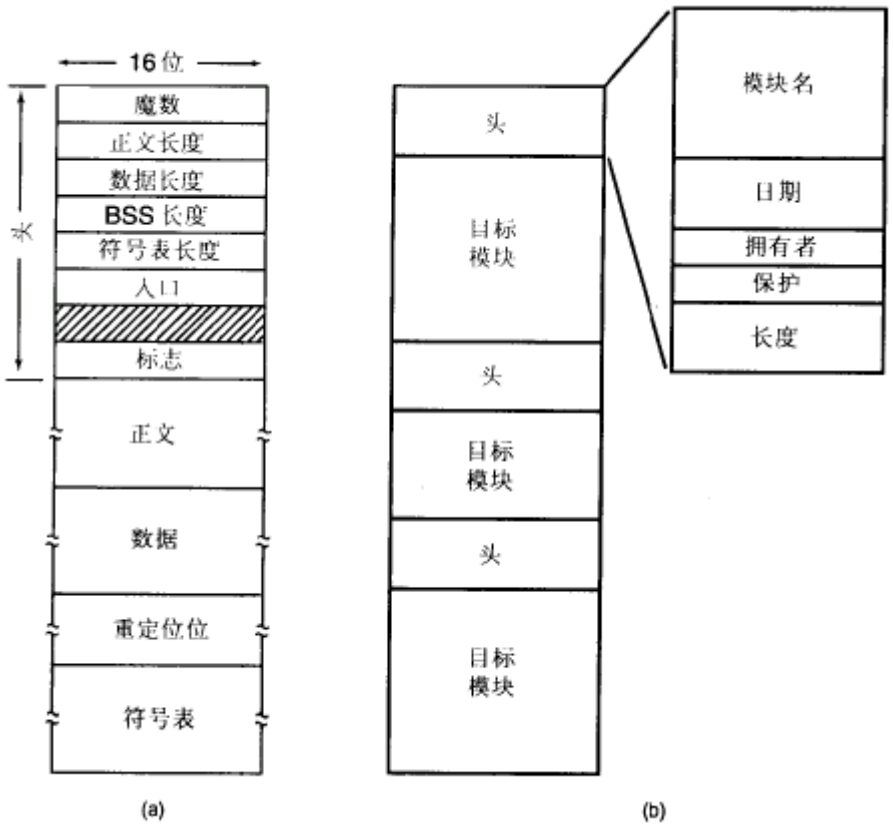
文件系统是操作系统中重要的一部分， 001 代码中文件系统相关的部分如下所示；

bitmap.c	inode 和 block 的 bitmaps	<100 ＊
block_dev.c	块设备读写。	<100＊
buffer.c	buff 的分配和等待队列管理。250 行＊	
Char_dev.c	字符设备读写＊	
Exec.c	exec.c 相关的函数，大概有 300 行	
Fcntl.c	文件控制	70 行
File_dev.c	文件读写	85 行
Inode.c	对 inode 的操作	288 行＊
Ioctl.c	sys_ioctl 函数	<40
Namei.c	文件查找和权限控制和对文件的其他操作	679 行
Open.c	文件操作 chown,open 等	<188 行
Pipe.c	pipe 读写分配	93 行
Read_write.c	系统的读写调用函数	<99 行＊
Stat.c	节点信息管理	53 行＊
Super.c	超节点管理	105 行＊
Truncate.c	缩减超节点？？	58 行
Tty_ioctl.c	tty 设备控制	167 行

操作系统课本上面的话我就不罗索了，下面是文件系统的结构图



下面是两种文件的结构，a 是可执文件，b 是存档文件。（摘自操作系统设计与实现　by Tanenbaum）



ecec.c

下面分析 exec()的实现过程：
exex()的功能就是执行用户输入的指令，包括以下步骤：
1、 查找指令对应的文件。（比如用户输入 “ls”，则首先找出”ls”对应的文件的 I 节点）；
2、 权限校验。根据文件的头和当前用户的环境决定用户是否有权利执行该文件。
3、 看看文件是不是可执行的。（就是判定 magic 魔数）。
4、 将命令行参数和系统参数复制到内存。
5、 将要执行的文件的数据区，代码区等和当前任务绑定。
6、 读入数据区和代码区的数据到内存。
7、 结束，等着系统调用。
Exec.c 其他函数可以参考我对函数的注释，这里就不罗列了。

Fcntl.c

向一个打开的文件请求操作。
Posix 请求参数表：
F_DUPFD 复制文件描述符
F_GETFD 获取 close_on_exec 标志
F_SETFD 设置 close_on_exec 标志

F_SETFL 设置文件状态标志
F_GETLK 获取文件锁状态
F_SETLK 对文件设置读写锁
F_SETLKW 对文件设置写锁
实际上，现在这个函数的主要功能就是给文件加锁。这个函数很简单，没有什么可以多说的。

File_dev.c

文件的读写 file_read, file_write
对于 file_read 来说，输入的是文件名，buf 指针，输出是成功表识，buf 里面是数据。
过程是：
首先定位到文件的物理块
把文件数据整块的读入到 buf 中去(bread 函数)
根据文件文件 f_pos 状况位移。
读完所有的块。
读完零头（put_fs_byte）。

file_write()函数
功能：将内存 i_node 的数据写道磁盘里面。Count 是要写的数目。
首先判定打开模式，是“ 追加 ”还是“ 写新 ”
然后生成新的磁盘块
然后过程和读相似。只不过
写完了还把时间保存了一把。
我对这个函数的正确性表示怀疑。

Ioctl.c

sys_ioctl（）
这个函数和目前的 sys_ioctl()差别和很大，只能实现文件模式的判定，不能实现真正的 control.
传入的参数根本没有用到。

Namei.c 内函数列表

Permission()
这个函数比较简单，就是看当前进程的用户符合文件用户的权限。可以看出 uid＝0 就是系统管理员。
这里有个 euid，这个东西叫有效 uid (gid)，一般和 uid(gid)相同，用户有时候被赋予其他权限的时候就改这个东东。

Match（int len,const char * name,struct dir_entry * de）

函数的功能就是注释里说的（/*
 * ok, we cannot use strncmp, as the name is not in our data space.
 * Thus we'll have to use match. No big problem. Match also makes
 * some sanity tests.
 *
 * NOTE! unlike strncmp, match returns 1 for success, 0 for failure.
 */）
程序是汇编写的，我懒得看，功能和下面的代码相同

```
    if (len != de->name_len)
46                      return 0; /*fails*/
47          if (!de->inode)
48                      return 0;
49          return !memcmp(name, de->name, len);
```

对目录的操作：
目录文件是一组目录入口的链表，它们包含以下信息：
inode
对应每个目录入口的 inode。它被用来索引储存在数据块组的 Inode 表中的 inode 数组。 在图 9.3 中 file 文件的目录入口中有一个对 inode 号 11 的引用。
name length
以字节记数的目录入口长度。
name
目录入口的名称
Linux 文件名的格式与 Unix 类似,是一系列以"/"隔开的目录名并以文件名结尾。/home/rusling/.cshrc 中/home 和/rusling 都是目录名而文件名为.cshrc。象 Unix 系统一样，Linux 并不关心文件名格式本身，它可以由任意可打印字符组成。为了寻找文件系统中表示此文件的 inode，系统必须将文件名从目录名中分离出来。
我们所需要的第一个 inode 是根文件系统的 inode，它被存放在文件系统的超块中。为读取某个 inode, 我们必须在适当数据块组的 inode 表中进行搜寻。如果根 inode 号为 42 则我们需要数据块组 0 inode 表的第 42 个 inode。此根 inode 对应于一个目录，即根 inode 的 mode 域将它描叙成目录且其数据块包含目录入口。home 目录是许多目录的入口同时此目录给我们提供了大量描叙/home 目录的 inode。我们必须读取此目录以找到 rusling 目录入口，此入口又提供了许多描叙/home/rusling 目录的 inode。最后读取由/home/rusling 目录描叙的 inode 指向的目录入口以找出.cshrc 文件的 inode 号并从中取得包含在文件中信息的数据块。（参考无名的分析报告）

```
struct dir_entry{
    unsigned short inode;   /*这个数指向下一级目录的 inode*/
    char name[255];
}

static struct buffer_head * find_entry(struct m_inode * dir,
    const char * name, int namelen, struct dir_entry ** res_dir)
```

函数用来查找文件路径，返回的 catch buffer 里面有找到的 entry.
函数过程：

1、 输入参数检查。
2、 entries = dir->i_size / (sizeof (struct dir_entry)); 得到当前目录节点里面的目录数目。
3、 if (!(block = dir->i_zone[0]))
return NULL;
if (!(bh = bread(dir->i_dev,block)))
return NULL;
读入输入的 dir 的第一块信息（根节点信息）
4、 de = (struct dir_entry *) bh->b_data;
将根节点的路径信息放入 de 中
5、 然后就是循环，如果找到的 entry 还是一个目录，就继续向下找，否则就用 match 比较
大家可以看看这个函数，他是不是只能支持两级目录呀。

add_entry（）
函数的输入输出和 find_entry()相似，流程也相似。
在主循环体中不断定位到目录树的叶节点。（在中间如果相关的磁盘块没有分配就新建）
定位以后就是增加目录项的操作。

static struct m_inode * get_dir(const char * pathname)
1、将 pathname 根据”/”分割成一个一个路径的节点。
2、针对每个路径节点，调用 find_entry 函数。
要说明的是，在搜索过程中不断的检查用户的权限。

有了上面的函数，下面的函数就比较简单了
static struct m_inode * dir_namei(const char * pathname,
int * namelen, const char ** name)
这里 namelen 是返回值，还有一个 m_inode 的返回值和 name 返回。
通过 get_dit()得到文件的路径，
然后分析路径长度，用 namelen 返回。
Name 里面存放的是绝对路径的字符串

struct m_inode * namei(const char * pathname)
调用 dir_namei 得到文件的 I_node,
然后对文件的时间进行修改。

int open_namei(const char * pathname, int flag, int mode,
struct m_inode ** res_inode)
打开 pathname 指定文件
1、 首先检查文件打开的模式。
2、 dir_namei 得到文件的路径（I_node）和路径长度
3、 find_entry()找到实际对应的磁盘文件，写 buffer。
4、 如果 3 不成功，就看是不是以新建方式打开文件，然后检查权限
new_inode 产生新节点，
add_entry 将节点加入指定文件路径
释放临时资源。

5、其他错误处理（权限不足，）

int sys_mkdir(const char * pathname, int mode)
就是生成新路径的系统调用。
1、 权限检查。
2、 dir_namei 得到要生成的路径的上层路径（如/usr/rayx/zzz/ded,上层路径在 zzz）
3、 路径长度和用户权限检查。
4、 find_entry 得到 I_node.
5、 生成新的节点，写入文件名。
6、 add_entry 将新的路径节点加入
7、 设置时间，属主信息。

要说明的是，每个路径的开始两个路径项是“ . ”和“ .. ”
static int empty_dir(struct m_inode * inode)
看看指定路径是否为空。
这里输入的 inode 是文件目录项。
1、 得到目录长度。
2、 目录长度小与 2 或者 zone[0]里面没东西，这个目录有问题。
3、 如果目录的前两项不是”.”和”..”也是错的。
4、 然后指定路径里面的每个文件进行检查

int sys_rmdir(const char * name)
rmdir 的系统调用
1、 查权限。
2、 搜索 find_entry 得到指定的目录 inode
3、 各种检查（权限，非空？，是不是文件…）
4、 de->inode = 0;删除目录下文件
5、 除掉连接，设置时间等等

int sys_unlink(const char * name)
找到指定的链接文件把 inode->i_nlinks——。
搜索过程参照前面的注释。

int sys_link(const char * oldname, const char * newname)
1、 找到 oldname（被连接文件）和 newname
2、 为 newname 建立新节点
3、 de->inode = oldinode->i_num;建立引用。
4、 对 oldname 的引用++,改时间 etc

Open.c

int sys_utime(char * filename, struct utimbuf * times)
函数设置文件的最后修改时间，然后用 iput(inode)保存修改。

sys_access（const char * filename, int mode）
1、定位指定文件（namei）
2、权限检查。

int sys_chdir(const char * filename)
int sys_chroot(const char * filename)
int sys_chmod(const char * filename,int mode)
int sys_chown(const char * filename,int uid,int gid)
以上函数都是找到 filename 指定的 inode 然后进行相关操作。

int sys_open(const char * filename,int flag,int mode)
mode: 可以新建 open，也可以只读 open,写 open
和刚才的函数不同的是，open 函数要对当前 task（current）的文件指针进行改动
同时 tty 在 001 里面也是作为一种文件打开的。

```
int sys_close(unsigned int fd)
{
	struct file * filp;

	if (fd >= NR_OPEN)
		return -EINVAL;
	current->close_on_exec &= ~(1<<fd);
	if (!(filp = current->filp[fd]))
		return -EINVAL;
	current->filp[fd] = NULL;
	if (filp->f_count == 0)
		panic("Close: file count is 0");
	if (--filp->f_count)
		return (0);
	iput(filp->f_inode);
	return (0);
}
```

这个函数写的很清楚，可以逆推一下 open 都作了什么。

Pipe.c

int read_pipe(struct m_inode * inode, char * buf, int count)
管道读，从 inode 里面读 count 个到 buf 里面去
是空就等，非空就读
int write_pipe(struct m_inode * inode, char * buf, int count)
写操作，程序写的很清晰。

int sys_pipe(unsigned long * fildes)

新建管道
函数实际上是调用 inode=get_pipe_inode()建立一个新管道.
首先在 current(当前进程)的文件打开表里面找到两个没有用到的项，用他们初始化 f[]和 fd[];
get_pipe_inode()返回一个 m_inode,并且将它置空。
然后 f[0]->f_inode = f[1]->f_inode = inode;
将两个文件描述符指向同一个内存 inode
f[0]->f_mode = 1; /* read */
f[1]->f_mode = 2; /* write */

truncate.c

static void free_ind(int dev,int block)
static void free_dind(int dev,int block)
这两个函数比较简单
设一个循环，读 block 里面的数据，如果没有读到东西就 free_block(dev,*p);
static void free_dind(int dev,int block)
是针对二级索引的

void truncate(struct m_inode * inode)
这个函数是从顶层开始
对一级索引使用 free_ind
二级使用 free_dind

Tty_ioctl.c

这是对 tty 设备的操作。
主函数是一个大 case 分枝，针对不同输入使用不同函数。
里面大部分是空着的。