

《操作系统实验》Lab7 实验报告

朱恬骅

09300240004 计算机科学与技术

1 实验目标

1. 学习关于文件系统的知识
2. 实现一个模仿文件系统的程序

2 实验要求

实现和设计一个简单的文件系统，实现下列功能：

1. 创建/删除文件：touch/rm
2. 文件夹相关操作：mkdir/rmdir and cd
3. 显示文件夹内容：ls

3 实验原理

3.1 文件系统

MINIX 文件系统与标准UNIX 的文件系统基本相同。它由6 个部分组成。对于一个360K 的软盘，其各部分的分布见图1所示。

图中，引导块是计算机加电启动时可由ROM BIOS自动读入的执行代码和数据。但并非所有盘都用于作为引导设备，所以对于不用于引导的盘片，这一盘块中可以不含代码。但任何盘片必须含有引导块，以保持MINIX 文件系统格式的统一。超级块用于存放盘设备上文件系统结构的信息，并说明各部分的大小。

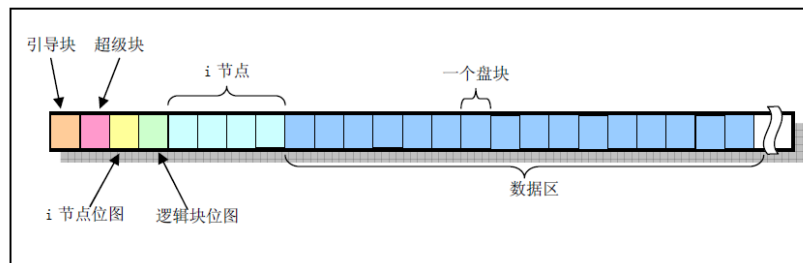


Figure 1: 文件系统示意图

3.2 Superblock 超级块

逻辑块位图最多使用8块缓冲块（`s_zmap[8]`），而每块缓冲块可代表8192个盘块，因此，MINIX 文件系统1.0 所支持的最大块设备容量（长度）是64MB。

i节点位图用于说明i节点是否被使用，每个比特位代表一个i节点。对于1K大小的盘块来讲，一个盘块就可表示8191个i节点的使用情况。逻辑块位图用于描述盘上的每个数据盘块的使用情况，每个比特位代表盘上数据区中的一个数据盘块。因此，逻辑块位图的第一个比特位代表盘上数据区中第一个数据盘块。当一个数据盘块被占用时，则逻辑块位图中相应比特位被置位。

3.3 i节点

盘上的i节点部分存放着文件系统中文件（或目录）的索引节点，每个文件（或目录）都有一个i节点。每个i节点结构中存放着对应文件的相关信息，如文件宿主的id(uid)、文件所属组id(gid)、文件长度和访问修改时间等。整个结构共使用32个字节。

`i_mode` 字段用来保存文件的类型和访问权限属性。其比特位15-12用于保存文件类型，位11-9保存执行文件时设置的信息，位8-0表示文件的访问权限。

文件中的数据是放在磁盘块的数据区中的，而一个文件名则通过对应的i节点与这些数据磁盘块相联系，这些盘块的号码就存放在i节点的逻辑块数组*i_zone*中。其中，*i_zone*数组用于存放i节点对应文件的盘块号。*i_zone*[0]到*i_zone*[6]用于存放文件开始的7个磁盘块号，称为直接块。若文件长度小于等于7K字节，则根据其i节点可以很快就找到它所使用的盘块。若文件大一些时，就需要用到一次间接块了（*i_zone*[7]），这个盘块中存放着附加的盘块号。对于MINIX 文件系统它可以存放512个盘块

	字段名称	数据类型	说明
出现在盘上和内存中的字段	s_ninodes	short	i 节点数
	s_nzones	short	逻辑块数 (或称为区块数)
	s_imap_blocks	short	i 节点位图所占块数
	s_zmap_blocks	short	逻辑块位图所占块数
	s_firstdatazone	short	第一个逻辑块号
	s_log_zone_size	short	Log ₂ (数据块数/逻辑块)
	s_max_size	long	最大文件长度
	s_magic	short	文件系统幻数
仅在内存中使用的字段	s_imap[8]	buffer_head *	i 节点位图在高速缓冲块指针数组
	s_zmap[8]	buffer_head *	逻辑块位图在高速缓冲块指针数组
	s_dev	short	超级块所在设备号
	s_isup	m_inode *	被安装文件系统根目录 i 节点
	s_imount	m_inode *	该文件系统被安装到的 i 节点
	s_time	long	修改时间
	s_wait	task_struct *	等待本超级块的进程指针
	s_lock	char	锁定标志
	s_rd_only	char	只读标志
	s_dirt	char	已被修改(脏)标志

Figure 2: Superblock的结构

号，因此可以寻址512 个盘块。若文件还要大，则需要使用二次间接盘块（i_zone[8]）。二次间接块的一级盘块的作用类似与一次间接盘块，因此使用二次间接盘块可以寻址512*512 个盘块。

4 设计和实现

大致思路如实验原理中所示，但是有所不同，即文件系统没有引导块。超级块和i节点的定义也略有区别，如下所示。

4.1 简化的超级块和i节点

由于实验的要求比较简单，我们不可能操作太大的文件，因而对超级块和i节点作了变动，具体如下注释描述。rawfs.h文件定义了文件系统的一些基本常量和基本结构，并声明了相关操作的函数。rawfs.cpp提供了这些操作的实现。

```
typedef struct inode {
    bool directory; // 是否为目录
    bool readable;  // 是否可读，总是为true
```

	字段名称	数据类型	说明
共 32 字节	i_mode	short	文件的类型和属性 (rwx 位)
	i_uid	short	文件宿主的用户 id
	i_size	long	文件长度 (字节)
	i_mtime	long	修改时间 (从 1970.1.1:0 时算起, 秒)
	i_gid	char	文件宿主的组 id
	i_nlinks	char	链接数 (有多少个文件目录项指向该 i 节点)
	i_zone[9]	short	文件所占用的盘上逻辑块号数组。其中: zone[0]~zone[6]是直接块号; zone[7]是一次间接块号; zone[8]是二次 (双重) 间接块号。 注: zone 是区的意思, 可译成区块或逻辑块。

Figure 3: inode的结构

```

bool writable; // 是否可写, 总是为 true
bool deletable; // 是否可删除, 总是为 true
bool link; // 是否是链接, 总是为 false
unsigned long size; // 文件大小
unsigned short zones [ZONE_NUM]; // 文件所占的磁盘
    块, 所有的磁盘块都是直接指向
unsigned short id; // 在文件系统表中的位
    置 inode
unsigned short devId; // 驱动器, 总是为 ID0
} inode;

typedef struct superblock {
    unsigned short nInodes; // 的个数 inode
    unsigned short nInodeBlocks; // 块个数 inode
    unsigned short nLogicalBlocks; // 逻辑块个数
    unsigned short nLBBMPBlocks; // 占用空间的位图
    unsigned short firstDataBlock; // 第一数据块编号
} superblock;

```

4.2 位图结构和相关操作

```

typedef struct bitmap {
    unsigned short nBlocks;

```

```

        unsigned char *data;
    } bitmap;

void bitmap_constr(bitmap& b, unsigned short nBlocks/*
    = 1*/);
void bitmap_set(bitmap& b, unsigned short id, unsigned
    short size /* = 1 */);
void bitmap_unset(bitmap& b, unsigned short id,
    unsigned short size /* = 1 */);
unsigned short bitmap_findFreeSpace(bitmap& b, unsigned
    short nBlocks);
void bitmap_load(FILE* fp, bitmap& dst);
void bitmap_dump(FILE* fp, bitmap& src);

```

4.3 文件系统API的实现

在fs.h和fs.cpp中，具体实现了这个文件系统上的一些操作。

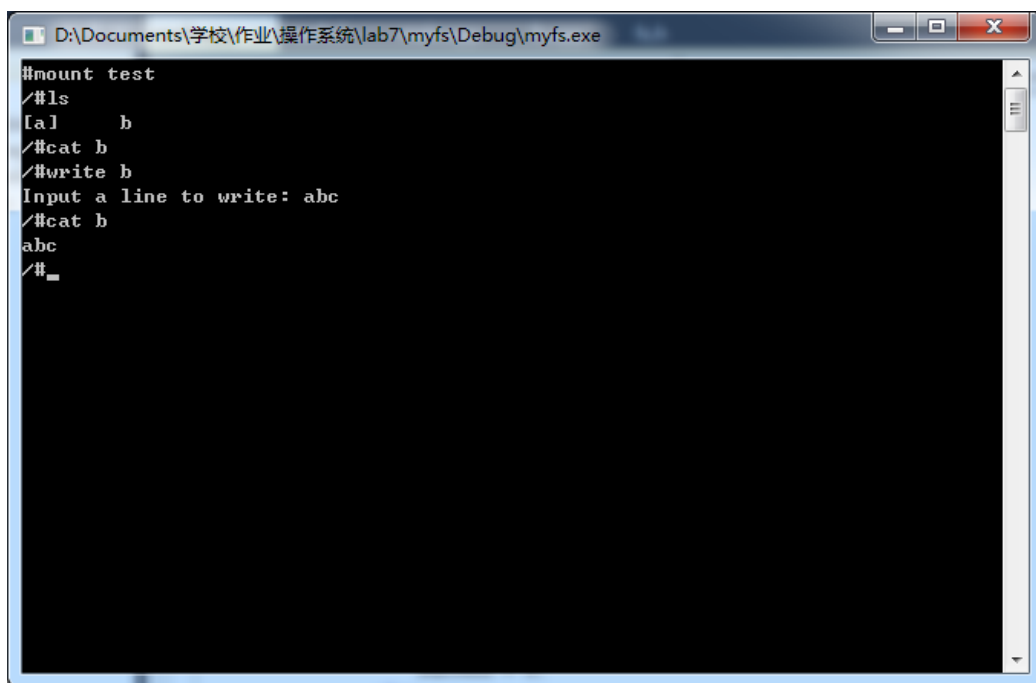
```

int makeFS(std::string name, int size);
int mountFS(std::string name);
void saveFS();
int unmountFS();
int exists(std::string fullpath);
int touch(std::string fullpath);
int write(std::string fullpath, std::string value);
int cat(std::string fullpath);
int rm(std::string fullpath);
int mkdir(std::string fullpath);
int rmdir(std::string fullpath);
inode* locate(std::string fullpath, int traceback);
int ls(std::string fullpath);
int lsx(std::string fullpath);

```

4.4 命令解析器

命令解析器对输入的字符串按空格进行分割，然后判断命令，执行相应的函数，并对出错情况进行判断、输出。main函数调用doCommand函数对分割好的命令及其参数进行执行，根据doCommand的返回值判断是否有出



```
#mount test
/#ls
[a] b
/#cat b
/#write b
Input a line to write: abc
/#cat b
abc
/#_
```

Figure 4: 运行时截图

错；doCommand则判断参数数量，调用具体的实现函数执行，并将实现函数的返回值返回给main函数。

4.5 运行截图

屏幕输出如图4所示。

这就完成了实验。

5 实验收获

1. 学习了Linux中关于文件系统的知识

2011年12月17日