**操作系统实验班大作业**

ramfs的一种扩展 – myfs

**周昊宇 / 1200012823**

# 概要

作为本小组的完成项目之一，我们在ramfs的基础上实现了一个比ramfs功能更全面的文件系统，编写成了内核模块，能够实现完整文件系统所支持的各种功能。

*以下两节内容与README.md一致。*

# 改进之处

1. 增加了很多选项，可以自定义更多参数
2. ramfs会无限制地占用内存，myfs增加了设置大小限制的选项
3. 原本的ramfs不支持statfs（df查看不到文件系统的大小和已用空间等信息），myfs增加了该项支持
4. ramfs删除inode或取消挂载时，由于页面和inode是脏的，会有内存泄露，myfs通过强行驱逐脏页减少了内存泄露

# 使用方法

你可以直接**用root权限**运行 test.sh （测试脚本），会自动完成make、安装、挂载、测试、结果展示和最终清理。

否则手动步骤如下：

* **使用root权限** make
* **使用root权限** 运行 install.sh
* **使用root权限** 运行 loadramdisk.sh 或 mount -t myfs -o <选项> /dev/null <挂载点>
  + 选项是逗号分割的 键=值 列表，比如mode=0777,size=10000000
  + 支持的选项：
    - mode（根目录权限位，值是8进制）
    - size（文件系统大小，值是16进制，单位字节）
    - filemsz（文件最大大小，值是16进制，单位字节）
    - blksz（块大小，值是16进制，单位字节，通常不能修改）
* 建议将挂载点所有者设为当前用户，然后便可以在挂载点内测试了
* df -h -t myfs 可以看到文件系统的大小、已用空间。
* viewlog.sh 可以看到myfs的日志。

# 具体实现

## 实现statfs接口

设置super\_operations.statfs为自己实现的statfs函数。

static int myfs\_statfs(struct dentry \*dentry, struct kstatfs \*buf)

{

struct super\_block \*sb = dentry->d\_sb;

buf->f\_type = sb->s\_magic;

buf->f\_bsize = sb->s\_blocksize;

buf->f\_namelen = NAME\_MAX;

buf->f\_blocks = MYFS\_INFO(sb)->fs\_max\_size / sb->s\_blocksize;

buf->f\_bavail = buf->f\_bfree = buf->f\_blocks - atomic\_long\_read(&MYFS\_INFO(sb)->used\_blocks);

if (buf->f\_bavail < 0)

buf->f\_bavail = buf->f\_bfree = 0;

printk("myfs: statfs - maxblks = %llu, freeblks = %llu\n", buf->f\_blocks, buf->f\_bfree);

return 0;

}

**【myfs\_statfs函数】**

## 跟踪块的分配和释放以计算占用的块数量

分配块发生在address\_space\_operations.write\_begin和write\_end中。

这里的块数量使用原子操作改动以免出现同步问题。

if (set\_page\_dirty(page))

{

// 如果是第一次设为Dirty，则修改文件系统的已用块数

struct super\_block \*sb = inode->i\_sb;

atomic\_long\_inc(&MYFS\_INFO(sb)->used\_blocks);

printk("myfs: write\_end[%pD] - set to dirty\n", file);

}

**【write\_end中的跟踪】**

释放块发生在dir\_inode\_operations.unlink中。

// drop\_nlink

WARN\_ON(inode->i\_nlink == 0);

inode->\_\_i\_nlink--;

if (!inode->i\_nlink)

{

atomic\_long\_inc(&inode->i\_sb->s\_remove\_count);

// 减掉文件系统的页面计数

atomic\_long\_sub(inode->i\_mapping->nrpages, &MYFS\_INFO(sb)->used\_blocks);

**【unlink中的跟踪】**

## 检查文件系统是否已满，防止文件进一步写入

文件写入发生在address\_space\_operations.write\_begin之后，通过检查文件系统fs\_info里空余块数量来决定是否允许，若空间不足则返回错误，阻止写入。

long maxblks = MYFS\_INFO(sb)->fs\_max\_size / sb->s\_blocksize;

long usedblks = atomic\_long\_read(&MYFS\_INFO(sb)->used\_blocks);

pgoff\_t index;

printk("myfs: write\_begin - maxblks = %ld, usedblks = %ld\n",

maxblks, usedblks);

if (usedblks >= maxblks)

{

printk("myfs: write\_begin[%pD] - insufficent space\n", file);

return -ENOMEM;

}

**【write\_begin中的检查】**

## 删除文件时强行驱赶Dirty页面

删除文件是dir\_inode\_operations.unlink，其中若引用计数为0，则调用linux里内存管理的删页函数，这个函数可以无视页面Dirty与否强行驱逐。

// 如果link计数为0，则完全删除该文件在内存中对应的所有Dirty页

truncate\_inode\_pages(inode->i\_mapping, 0);

printk("myfs: unlink[somefile under %pD] - final delete\n", dentry);

}

**【unlink中的删页】**

## 卸除文件系统时暴力删除所有页面

卸载文件系统即删除内存中的超级块，是file\_system\_type.kill\_sb，其中除了free掉之前分配的fs\_info以外，增加了来自内核vfs实现中的删除inode函数（invalidate\_inodes），这个函数可以无视inode和页面Dirty与否强行删除。

然而这个定义在fs/inode.c里的函数并没有对外公开（没有导出符号），正常情况下无法在自定义内核模块中直接调用，因此项目中使用了一个hack，即在make时从/proc/kallsyms里读取这个函数的内核态地址，然后通过宏赋给代码里定义的函数指针，并强行调用。

// 这是个Hack，用于从内核中找出私有函数的地址

int (\*invalidate\_inodes) (struct super\_block \*, bool) = (int (\*)(struct super\_block \*, bool)) INVALIDATE\_NODES\_ADDR;

// 回收超级块

static void myfs\_kill\_sb(struct super\_block \*sb)

{

kfree(sb->s\_fs\_info);

if (invalidate\_inodes(sb, 1)) // 强制毁灭所有inode及其dirty页

printk("myfs: Eliminate all inodes finished partially!\n");

kill\_litter\_super(sb);

}

**【myfs\_kill\_sb函数】**

## 文件系统额外信息

通过自定义super\_block里的fs\_info实现文件系统大小统计和参数存储。

struct myfs\_fs\_info {

unsigned long fs\_max\_size; // 文件系统总大小限制，默认是MAX\_FS\_SIZE

unsigned long file\_max\_size; // 文件大小限制，默认是MAX\_LFS\_FILESIZE

unsigned long block\_size; // 默认是PAGE\_CACHE\_SIZE

umode\_t root\_mode; // 文件系统根目录权限位，默认是DEFAULT\_MODE

     atomic\_long\_t used\_blocks; // 文件系统已用页面数

};

**【myfs.h部分内容】**

## 参数解析

通过将fill\_sb的data传入解析函数，使用match\_token等函数来得到fs\_info。

# 经验收获

## 内核编程主要靠读代码

内核编程和普通编程不同，案例稀少，在网上很难找到自己所需的答案，这时读代码技能便变得非常重要，以致本次大作业中完成该项目的时间中多半是在读内核源码。

由于内核源码注释稀少，文档也难得一见，我们读代码时遭遇了非常大的困难。而且由于内核版本更替，经常出现所用函数不存在的情况，导致代码经常需要重写。

再者，通常的IDE都不能完善支持内核编程，因此寻找函数、结构体、宏等的定义就变得十分麻烦。有一个网站<http://lxr.free-electrons.com/source>可以搜索符号，但是网速很慢，而且有些“小众”符号仍然无法直接搜索。

## 不必重复造轮

通常在内核编程中遇到的需求都是比较常用的，因此往往有人已经在内核中实现了。虽然搜索功能近乎鸡肋，我们很难根据自己的功能描述找到所需函数，但是内核中文件的命名风格都是比较好的，通过在源码树中不断缩小搜索范围，往往能找到自己要的函数，比如上文提到的暴力删除函数。

## 输出调试大法好

由于调试工具难用，简单粗暴的输出调试便成了最好的选择。特别是内核设计者们提供的printk具有各种神奇的格式化符号，比如%pD可以直接输出dentry或file的路径，大大方便了我们的调试工作。