操作系统实验班大作业

ramfs 的一种扩展 – myfs

蒋捷 / 1200012708 & 兰兆千 / 1100012458 & 邢曜鹏 / 1200012835 &

赵万荣 / 1200012808 & 周昊宇 / 1200012823 （音序）

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_ADD
R;

// 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。