Lab4: Cloud File System

AndroidOS@PKU

2017 年 1 月 13 日

1 前言

随着智能手机和平板电脑的普及，这些基于闪存的固定容量的设备与我们的生活越来越密切。然而我们都知道相比磁盘而言，闪存价格要高昂的多，而且现在的设备闪存的大小基本都是固定的,如32G、64G。人们为了购买容量较大的产品不得不花费额外的高昂费用。在这样的背景下，现在的消费者所购买的容量大部分都是来存储各种媒体文件(如图片、视频、音乐等)。而实际上，对于硬盘上的数据而言，CPU每次使用的其实只有很小的一部分，其余的都是以存储备份的形式保留在我们的设备上。从某种意义上讲，这其实是一种浪费。

在这个Lab中,我们要解决这一问题。我们要修改一个安卓文件系统,使得它可以将原来设备的部分数据存储在设备上,而其他不经常使用的文件动态的存储在更便宜的云盘上。这样我们就可以牺牲一定的速度来达到理论上的无限存储空间。具体来说，我们可以把本地文件系统考虑成一个缓存,当它的使用率超过一个额度W时,一段时间内没有使用的旧文件会被驱逐并存储在云盘中。如果用户要访问一个存储在云盘中的文件,它将被取回(必要时要将其他文件驱逐)。需要注意的是，我们只能把数据上传到我们的云服务器上，这样我们才能管理我们所有的文件。因此，我们希望在设备上存储所有文件的元数据（比如目录文件和inodes），因为他们消耗空

间相对较少,并且能让用户快速浏览文件系统的内容。也就是说，类似于磁盘和内存的交换系统，我们将实现的是一个闪存和远程磁盘的交换系统。

2 Lab简介

由于Android使用的是Linux操作系统的内核，所以在文件系统方面两者的实现几乎是一致的。为了能识别众多复杂的文件系统，Linux引入了虚拟文件系统机制（VFS），也就是介于具体文件系统和用户函数之间的一层接口。虽然Linux是用C语言实现，但是为了能够处理多种文件系统它还是引入了面向对象的一些机制。具体来讲，VFS内部本身定义了像read、write这些函数指针。注意，这些只是函数指针，如果文件系统支持这些操作，他们就会把自己具体的函数体赋值给这些指针，像EXT4、FAT32这些具体的文件系统在被挂载上去的时候会将自己的一系列函数对应到VFS内部定义的地址中，这样实际在调用的时候就可以通过使用VFS内部的这些函数指针，来对应到具体的操作了。

在这个lab中，我们要在VFS中添加一种新的监控机制，这种监控机制可以让我们实时监控我们的文件系统内的磁盘使用率，并在必要时把一些不需要的文件上传到网盘。我们将逐步实现这个功能。

值得一提的是，现代Android默认的文件系统是YAFFS2 flash-optimized日志结构文件系统。在这个lab中，为了简便起见我们将会选择修改Ext2文件系统来实现云文件系统，所以我们要在我们的模拟设备上挂载一个额外的存储设备，附录中有详细的说明指南。另外，在完成这个lab的时候，如果对一个功能或者函数不熟悉，你可以到内核的其他代码中看看其相关的实现和调用，相信对你的工作会很有帮助，

3 Part A 系统自启动线程

首先，我们要在在linux/fs.h中的super_operation中添加一种新的VFS superblock操作：

```
1        int evict_fs(struct super_block *super);
```

这个操作以一个文件系统的super_block为参数，如果该文件系统支持这个操作，那么我们可以通过VFS启动它。

然后，你需要创建一个叫做kfs_evictd的内核监控线程，这个线程在系统启动时自动启动，周期性地依次遍历这个操作系统中所有的文件系统，如果文件系统支持监控操作，那么就要调用evict_fs()函数。请在fs/evictd.c文件中实现这件工作。一旦开始运行,监视线程应该每分钟运行一次,扫描每个挂载的文件系统并调用evict_fs()。

提示:

1.系统自启注册函数为fs_initcall()。

2.线程启动函数为kthread_run()。

4 Part B 挂载文件系统

附录中介绍了如何初始化一个SD卡镜像，和如何格式化成标准EXT2文件系统并mount到文件系统中，请仔细阅读并完成相关操作，之后再进行如下的工作。

在这个lab中，我们希望使用EXT2的原生文件结构组织，也就是说，我们不能修改inode和super_block的结构体。但是对mount指令的一些基本的扩展是可以做的，因为mount中生成的一些结构是保存在内存中，与硬盘无关。在这部分中，我们希望对mount这个指令进行修改，使得当我们挂载一个标准的EXT2文件系统的时候可以使用-o选项来输入更多的参数。具体要求如下：

I. srv=server name or ip:port：云服务器程序所在的IP地址和端口，如果没有这个参数，mount命令应该返回错误。

II. wh=num：the high watermark。应该是一个从0到100的数字。其缺省值为95。详细定义请见Part D。

III. wl=num：the low watermark。应该是一个0到wh的数字。其缺省值为85。详细定义请见Part D。

IV. evict=num：the eviction target。应该是一个0到wl的数字。其缺省值为70。详细定义请见Part D。

另外，我们需要在/fs/ext2/目录下建立ext2_evicted.c文件，这个文件中将保存我们云服务器有关操作。在这个文件中声明一个具体的监控函数:

```
1        int ext2_evict_fs(struct super_block *super);
```

并在mount的时候将这个函数注册到vfs的evict_fs()上，这样这个函数就可以被Part A中的线程调用了。这个函数的具体实现我们将在下一部分详细说明。

提示:

1.与ext2文件系统的mount指令相关的操作都在/fs/ext2/super.c中。

2.ext2_setup_super(); 和 parse_options();两个函数提供了基本的参数解析，请自行学习并修改有关结构体和函数。

5 Part C 实现监控程序

在这一部分，我们将对ext2_evict_fs();进行一个大体的初步实现。我们需要完成如下的两个任务：1.遍历当前文件系统中的所有文件。2.查看当前文件系统的磁盘使用率。

具体的来说，我们将在ext2_evict_fs();函数中实现对当前文件系统中的所有文件的遍历。需要注意，虽然VFS中的inode自己本身有一个链表，但是这个链表并不代表着所有的文件。请回想文件块缓冲机制，只有磁盘上的一部分块会加载到操作系统中，因此我们需要通过读取磁盘信息完成对文件系统文件的遍历。EXT2的空闲块和inode的管理方式都是bitmap的形式，我们可以通过对bitmap的访问来找到对应的inode号，从而访问所有的文件。另外，为了方便我们接下来的策略的实现，我们在遍历文件的时候要求保留扫描的指针，也就是说扫描指针应该在重启前后能够保持一致。

在Part B中，我们提到过我们希望使用EXT2的原生文件结构组织，所以不能修改super_block的组织。

为了在不改变EXT2原生结构的基础上存储一些metadata，最基础的想法是把这些数据保存到数据块上。在这里我们介绍Extended Attributes(xattr)机制。这个机制是EXT2引入进行安全性检查的模块，其保存了文件的一些安全信息，并且没有文件系统的结构体进行修改。在这里，我们把扫描指针存储为root目录的一个扩展属性'clockhand'，用来记录指针指向的inode值(如果找不到该扩展属性，默认值为0。

另外，我们还应该单独实现一个函数:

```
1       int ext2_get_usage(struct super_block *super);
```

这个函数可以返回当前ETX2文件系统的块使用率。

提示:

1.你可以在 /fs/ext2/ialloc.c 中的ext2_new_inode函数中找到关于bitmap的类似操作。

2.有关XATTR的操作可以在/ext2/xattr.c 和 /ext2/xattr.h中找到有关的实现。

6 Part D 驱逐策略:时钟算法

在Part B中，我们引入了三个参数:the high watermark(wh)、the low watermark(wl)、the eviction target(evict)。这三个参数分别表示分别表示三个阈值。

我们已经在在fs/Ext2/ext2_evict.c中实现一个Ext2驱逐策略的核心函数的原型

```
1       int ext2_evict_fs(struct super_block *super);
```

接下来我们把这个函数进行进一步的完善。我们应该在Ext2文件系统上扫描文件并实现Clock (Second Chance)算法。具体要求如下：

1. 如果文件系统的使用率大于the low watermark(wl), 操作系统应该调用具体文件系统过的evict_fs(), 这个函数应该从当前指针(clock hand)指向的位置开始扫描inodes序列, 跳过目录inode和已经打开的inode并记录最后一次扫描时间,使用Clock (Second Chance)算法找到需要驱逐的文件的inode，并调用驱逐程序接口驱逐这个文件(驱逐程序将在Part F中完成，这里可以先调用一个空函数)。重复这个过程, 直到磁盘使用量低于the eviction target(evict)。

2. 在实现Clock算法时，我们需要添加一个名为scantime的扩展属性，记录文件的最后扫描时间。如果找不到某文件的scantime,创建一个初始值为0的scantime。

3. 除了在系统启动线程上被启动外，如果Ext2文件系统在分配一个新的磁盘块时，FS利用率在分配后高于the high watermark(wh)，那么将直接在分配前调用ext2_evict_fs()函数。

注：虽然这里evict_fs()函数和ext2_evict_fs()函数最后执行的函数体是同一个，但是其所代表的层次和意义是不一样的，请仔细思考两者之间层次的差别。

7 Part E 服务器代码编写

在正式实现驱逐和取回函数之前，你需要写一个运行在Android模拟器外的云端服务器程序clfs_server.c。这个程序用于存储并管理从你的文件系统中驱逐出来的文件。你的服务器程序只能使用标准的libC API(不能使用外部库)。它监听来自端口8888的TCP连接请求。在接收一个新的连接时,它应该产生一个新的线程并提供相应的服务。请求格式如下:

```
1       struct clfs_req{
2               enum {
3                       CLFS_PUT,
4                       CLFS_GET,
5                       CLFS_RM
6               } type;
7               int inode;
```

```
        int size;
    };
```

分别为请求的类型，inode号和文件的大小。注意，这里我们有CLFS_RM命令格式，但是我们在这个lab并不做要求，有余力的同学可以自己实现相应的删除请求。

在受到操作请求之后，你的服务器应该回复一个状态码，然后进行相关操作，状态格式如下：

```
    enum clfs_status {
        CLFS_OK = 0,            /*Success*/
        CLFS_INVAL = EINVAL,    /*Invalid address*/
        CLFS_ACCESS = EACCESS,  /*Could not read/write file*/
        CLFS_ERROR              /*Other errors*/
    }
```

当服务器收到CLFS_PUT请求并成功响应后时,服务器应该等待接收一个unsigned char data[size]数据块，在接收完数据之后应该返回一个状态码表示是否发送成功。如果请求类型是CLFS_GET，服务器应该进行相关检查并回复状态码，如果没有错误,服务器应该随后返回被请求的unsigned char data[size]数据块。如果是CLFS_RM,那么服务器应该回复状态码并删除相应的文件。注意这个程序是在服务器用户态下运行的，所以你可以使用相关的系统调用。我们强烈建议你在运行安卓模拟器的主机上运行云端服务器，这样你可以在模拟器内通过IP地址10.0.2.2连接到主机。云服务器接受的文件命名应该为[inodenum].dat，并储存在clfs_store文件夹中，这个文件夹应该与clfs_server.o在同一个目录下。

8 Part F 驱逐和取回操作

本部分将实现扩展的Ext2文件系统的两个操作，因而也是这次lab中比较复杂的部分。这些操作与云端服务器进行通讯，并向操作系统提供接口，为了把文件驱逐到云服务器并在需要时取回，我们需要在fs/ext2/ext2_evict.c中实现以下两个函数。与服务器程序不同，这些程序运行在内核态下，所以

在进行直接的读写操作时要注意相关的安全问题。

```
/* Evict the file identified by i_node to the cloud server,
* freeing its disk blocks and removing any page cache pages.
* The call should return when the file is evicted. Besides
* the file data pointers, no other metadata, e.g., access time,
* size, etc. should be changed. Appropriate errors should
* be returned. In particular, the operation should fail if the
* inode currently maps to an open file. Lock the inode
* appropriately to prevent a file open operation on it while
* it is being evicted.
*/
int ext2_evict(struct inode *i_node);

/* Fetch the file specified by i_node from the cloud server.
* The function should allocate space for the file on the local
* filesystem. No other metadata of the file should be changed.
* Lock the inode appropriately to prevent concurrent fetch
* operations on the same inode, and return appropriate errors.
*/
int ext2_fetch(struct inode *i_node);
```

ext2_evict()和ext2_fetch()函数分别以两个inode为输入，通过内核socket编程将inode对应的文件块发送到服务器上。同时我们需要添加一个名为evicted的扩展属性，如果文件被驱逐则值为1，否则为0。如果找不到某个文件的这个属性,那么默认其值为0，并将这个属性添加到该文件。任何分配给被驱逐文件的空间（包括直接和间接块）必须被释放,并释放对该文件的页面缓存。注意文件的inode应始终保留在移动设备上，并且其大小也要保存，当文件被驱逐,它的数据块会存储到云盘并从本地文件系统中删除。也就是说，在操作系统眼里这个文件还是这么大，但是其inode中指向相关块的指针已经失效了。

文件size不变且块指针失效对未修改的操作系统来说是一件可怕的事情，因为这意味这文件的丢失。因此我们需要修改 Ext2 在 VFS 中的注册函数 file_operations.open() 来检查一个要打开的文件是否已经被驱逐。如果文件被驱逐了，那么在打开的时候要通过调用 ext2_fetch()函数从云服务器取回它。注意我们没有实现删除的有关操作(为了减少工作量)，但是我

们在与服务器通讯时保留这个命令。

注：

1. 在与服务器通讯的时候，我们可以使用系统调用中实现的socket接口，注意相关的安全权限的检查。

2. 注意我们的fetch和evict函数以inode为参数。我们不能使用file，是因为file是一个与进程相关的结构体，而我们关心的是磁盘上的文件，也就是说这些文件可能并没有加载到VFS中。同理，我们也不能通过inode找到相应的dentry结构体，这也是违法的，因为dentry也是只存在于操作系统中，而且一个inode可能会有多个dentry与其对应。

提示:

1. 内核态的socket可到网上查找相关资料，注意指针安全性的考虑。另外，由于模拟器版本和内核参数的原因，可能会导致部分socket创建失败。如果发生有关错误，可以到ipv4的driver中关闭相关的安全检查。

2. 本部分文件的读写需要使用内存页对文件块的映射，请自行学习文件系统中address_space的有关机制，你可能需要了解read_mapping_page()，set_size()等等函数的功能和实现。

9 Part G 测试

在实现了两个函数接口之后，我们Part A中的线程应该就正常的工作了，同时我们可以进行相关的测试，理论上我们可以在这个文件系统中写入无穷大的文件。

我们将为你提供client_test.sh和host_test.sh在模拟器和主机上测试你的文件系统的实现。你将要提交如下文件：

I. diff文件, 新增加的文件的源代码，云服务器源代码。

II. client_test评分程序在模拟器中运行的输出。

III.评分程序完成后的内核日志(adb dmesg命令的输出)。

IV.测试完成后最终的磁盘映像文件CFS.img。

V. host_test评分程序在云服务器主机上运行的输出。

10 附录：挂载一个SD卡

在这个Lab中，我们将挂载一个额外的文件系统，在修改并挂载我们的Ext2文件系统之前，我们需要修改一些内核编译的配置文件。安卓内核提供了一些内置的支持，使用

```
make menuconfig
```

命令调出内核配置工具。在这个工具里，选择File systems菜单里的Second extended fs support 和 Ext2 extended attributes，请不要选择其他与Ext2相关的选项。将内核配置文件保存到 arch/x86/configs/goldfish_ext2_defconfig，并使用它来配置编译你的内核。

接下来，我们要挂载一个文件系统：

1. 生成SD卡镜像。

在使用模拟器之前，你可以使用AVD setting(android avd)并选择 "SD-Card"选项来初始化生成一个大小为10M的SD卡，这个镜像会保存在

```
~/.android/avd/<name_of_your_avd>.avd/sdcard.img
```

或者，你可以使用Android SDK包（需要下载）中tools/目录下的 mksdcard命令，mksdcard命令的具体指令为

```
mksdcard -l CFS 10M CFS.img
```

不管你使用何种方法产生了自己的SD卡镜像，在使用emulator时需要在命令行中加入新的选项指令

```
-sdcard <img_name.img>
```

最后，SD镜像会以如下安卓设备的方式出现:

```
/dev/block/mmcblk0
```

2. 下载busybox工具包

Android的内置内核并没有处理Ext2文件系统和硬盘相关的命令，为了使用这些命令，我们需要使用第三方工具包busybox，你可以在这里下载安

卓中可执行的busybox二进制文件，选择合适的版本并用adb命令传输到模拟器中

```
adb pull busybox /data
```

使用chmod命令使busybox可读可写可执行。

3.格式化SD卡

接下来，我们需要在附加的SD卡上挂在我们的文件系统。如果SD卡已经自动挂载，到Setting的 Storage&USB 目录下卸载它，接下来执行如下命令。

```
Run adb shell
cd /data
./busybox fdisk /dev/block/mmcblk0
```

这时我们会进入到fdisk中，使用如下命令：

```
o (create a new partition table)
n (add a new primary partition 1)
  (start at cylinder 1, end at ylinder 320)
  (This partition should contain 10232 blocks whose size is 512byte)
t (maksure partition type is set to Linux(type number: 83))
v (verification)
w (wirte)
```

这时，你应该能够看到 /dev/block/mmcblk0p1 分区，我们要在这个分区上建立Ext2操作系统，使用命令 ./busybox mke2fs -b 1024 -L CFS /dev/block/mmcblk0p1 可以将该分区格式化为每块大小为1K的Ext2文件系统。

4.挂载

使用mount指令把这个文件系统挂载在/mnt/sdcard上，

```
rm -rf /mnt/sdcard
mkdir /mnt/sdcard
mount -t ext2 /dev/block/mmcblk0p1 /mnt/sdcard
```

你可以使用没有参数的mount指令查看挂载情况。