Android OS Practice Lab 4-基于 socket 通信的云文件系统

李炎 1400012951

Background And General Description：

随着网络技术日新月异的发展，很多场景下网络带宽已经不再是网络服务的瓶颈。这一现象也使得云文件系统的普及得到了可能。而另一方面，人们日常生活中使用的终端设备中，常用的只是一些媒体资料，或一些图像资料等，剩余的数据大多数是以备份的形式存放在存储介质上，某种程度上讲这其实是一种浪费。本 lab 以此为出发点，旨在搭建一个 Android 系统下的云文件系统，定期扫描设备，当文件系统的使用率达到算法容忍的下界后将一些不常用的文件驱逐出设备，为用户腾出更多的使用空间。

Part A：初始化并挂载文件系统

简便起见，本 lab 中我们开展实验的目标文件系统是 Ext2 文件系统。该文件系统以一个 SD 卡镜像为载体，挂载到系统的一个目录下。本部分按照 Write up 中所描述的一步一步开展即可。具体几个步骤如下：

1， 修改内核编译的配置选项，使其支持 Ext2 文件系统拓展，并重新编译内核。
2， 生成一个 SD 卡镜像。此步骤我使用了 Android SDK 包 tools 目录下的 mksdcard 工具。
3， 使用 busybox 工具格式化 SD 卡。
4， 将格式化完毕的文件系统挂载到/mnt/sdcard 目录下

Part B：云端服务器

为方便起见，本 lab 中所用到的云文件系统服务器我直接在虚拟机的宿主机上完成。这样虚拟机可以直接通过 IP 地址 10.0.0.2 直接与宿主机建立 TCP 连接。TCP 连接的握手信号定义如下：

```
struct clfs_req{
    enum{
        CLFS_PUT,
        CLFS_GET,
        CLFS_REM
    }type;
    int inode;
    int size;
};

enum clfs_status{
    CLFS_OK,
    CLFS_INVAL,
    CLFS_ACCESS,
    CLFS_ERROR
};
```

结构体clfs_req是client端发给server端的请求信号，枚举类enum定义了三种请求动作，分别是请求将文件驱逐到云服务器、将文件从云服务器取回、将文件从云服务器删除。Inode字段是一个文件的唯一标示，由内核中的 inode 结构体的 i_ino 字段唯一给出（后文还会提到）。size 是本次传输的数据长度。

结构体clfs_status是server端返回给client端的状态码，对应于server是否允许client建立并完成这次传输。

Server 端的服务器程序用 socket 编程实现。几个关键的函数有：

1，int open_listenfd(int port);

open_clientfd 函数接受一个指定监听端口的参数 port，并在该端口上建立一个监听文件描述符，具体代码如下：

```
int open_listenfd(int port){
    int listenfd, optval = 1;
    struct sockaddr_in serveraddr;

    if((listenfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return -1;

    if(setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
                (const void *)&optval, sizeof(int)) < 0)
        return -1;

    bzero((char *) &serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons((unsigned short)port);
    if(bind(listenfd, (struct sockaddr *)&serveraddr, sizeof(serveraddr)) < 0)
        return -1;

    if(listen(listenfd, LISTENQ) < 0)
        return -1;

    return listenfd;
}
```

2，void handle_req(void *args);

handle_req 函数接受一个连接文件描述符为参数，首先服务一个来自client端的握手信号，然后根据握手信号的情况进行判断以及存在性和安全性检查，最后根据不同的情况接受来自client 端发来的数据或者将云服务器的文件数据发送到 client 端。当从 client 端接收数据时，server 将文件数据写入到云文件系统的 clfs_store 文件夹下的 inode_id.dat 文件中，文件名前缀即为文件 inode 的唯一标示。注意到处理来自 client 端请求时主线程创建一个子线程来处理这个请求，因此为了不阻塞主线程，handle_req 中使用了

pthread_detach(pthread_self()) 具体代码如下：

```
void handle_req(void * args){
    int connfd;
    int ret;
    int circle;
    int type, inode, size;
    char req[REQLEN];
    char buf[MAXLEN];
    char filename[FILELEN];
    char temp[FILELEN];
    FILE *f;

    pthread_detach(pthread_self());
    connfd = (int)(*((int *)args));
    ret = recv(connfd, req, REQLEN, 0);
    if(ret > 0){
        sscanf(req, "%d%d%d", &type, &inode, &size);
        sprintf(filename, "./clfs_store/%d.dat", inode);
        switch(type){
            case 1://put
                circle = 0;
                printf("receiving evicted data\n");
                f = fopen(filename, "wt+");
                while((ret = recv(connfd, buf, MAXLEN, 0)) > 0){
                    printf("data #%d: %s\n", circle++, buf);
                    fputs(buf, f);
                }
                fclose(f);
                break;
            case 2://get
                if(!access(filename, 0)){
                    printf("file does not exist\n");
                    break;
                }
                f = fopen(filename, "wt+");
                while(size > MAXLEN){
                    fgets(buf, MAXLEN, f);
                    size -= MAXLEN;
                    send(connfd, buf, sizeof(buf), 0);
                }
                if(size != 0){
                    fgets(buf, size, f);
                    send(connfd, buf, sizeof(buf), 0);
                }
                break;
            case 3://remove
                if(!access(filename, 0)){
                    printf("file does not exist\n");
                    break;
                }
                if(!remove(filename)){
                    printf("remove file failed\n");
                }
                break;
        }
    }
    close(connfd);
}
```

3，int main(int argc, char **argv);

main 函数中首先调用 open_clientfd 创建一个在 8888 端口上的监听描述符，进入无限循环

不断接收来自 client 的数据请求。一旦接收到一个请求，就创建一个子线程处理这个连接，并进入下一轮循环继续处理别的连接请求。代码如下：

```
int main(){
    char buf[MAXLEN];
    int listenfd, connectfd;
    struct sockaddr_in client;
    socklen_t addrlen;
    pthread_t client_id;
    int create_ret;

    listenfd = open_listenfd(PORT);
    client_id = 0;
    while(1){
        printf("File server listening on port 8888...\n");
        ++client_id;
        addrlen = sizeof(client);
        connectfd = accept(listenfd, (struct sockaddr *)&client, &addrlen);
        create_ret = pthread_create(&client_id, NULL, (void *)handle_req, (void *)&connectfd);
        if(create_ret != 0)
            printf("Thread create failed!\n");
    }
    return 0;
}
```

Part C 驱逐机制

本部分拓展了 ext2 文件系统的文件操作，新增加了两个核心接口：

int ext2_evict(struct inode *i_node);

int ext2_fetch(struct inode *i_node);

本部分我将这两个函数的实现拆分为了几个模块，使程序有更强的结构性和易拓展性。具体如下：

1，头文件 ext2_evict.h

该头文件中定义了 ext2_evict 定义的所有的函数的返回码，具体如下：

```
#define MAXLEN 1024
#define MAXBLK 10232
#define EXT2SIZE 10*(1<<20)
#define EXT2_MAX_SIZE 0//7*(1<<20)

#define EVICT_WH
#define EVICT_HL
#define EVICT_TG

#define EVICT_REPEAT 0x1
#define EVICT_NOTEXIST 0x2
#define EVICT_NOSERVER 0x4
#define EVICT_NETERR  0x8

#define SERVERADDR "10.0.2.2"
#define SERVERPORT 8888
```

其中 EVICT_WH 等三个宏是决定调用驱逐函数的下界，EVICT_REPEAT 等四个宏定义了调用相关的函数遇到错误时的返回码，具体意义如下：

EVICT_REPEAT：驱逐一个已经被驱逐的文件

EVICT_NOTEXSIT：要驱逐的文件不存在

EVICT_NOSERVER：无法与文件服务器建立 TCP 连接（文件服务器未启动）

EVICT_NETERR：建立 TCP 连接中发生其他错误

头文件中向操作系统开放了四个接口：

```
int ext2_evict(struct inode *i_node);
int ext2_fetch(struct inode *i_node);
int ext2_evict_fs(struct super_block *super);
struct super_block *ext2_get_super_block(void);
unsigned long get_ext2_usage(struct super_block *super);
extern int clear_data_blocks(struct inode *i_node);
```

前两个接口前文已述。

函数 ext2_evict_fs 针对一个 super_block 进行扫描，获得超级块的利用率，以及在利用率超出下界时决定驱逐哪些 inode 对应的文件。该函数是整个云文件系统的接口，同时也作为 super_block 操作的一种添加到 super_operations 中。如下图：

```
struct super_operations {...
#ifdef CONFIG_QUOTA...
#endif
    int (*bdev_try_to_free_page)(struct super_block*, struct page*, gfp_t);
    int (*nr_cached_objects)(struct super_block *);
    void (*free_cached_objects)(struct super_block *, int);
    int (*evict_fs)(struct super_block *);
};
```

函数 ext2_get_super_block 获得我们挂载的 ext2 分区的超级块。通过遍历所有的 super_block 判断是否该 super_block（以下简称 sb）是 ext2 的 super_block。代码如下：

```
struct super_block *
ext2_get_super_block(void){
    struct super_block *sb;
    int cnt = 0;
    list_for_each_entry(sb, &super_blocks, s_list){
        if(sb)
            printk("[test] super block #%d name : %s\n", cnt++, sb->s_type->name);
        if(sb->s_op->evict_fs)
            return sb;
    }
    return NULL;
}
```

函数 get_ext2_usage 通过计算一个 sb 中空闲的磁盘块和已经使用的磁盘块数目之比获得当前 sb 的占用率，返回一个 0 到 100 之间的整数，代码如下：

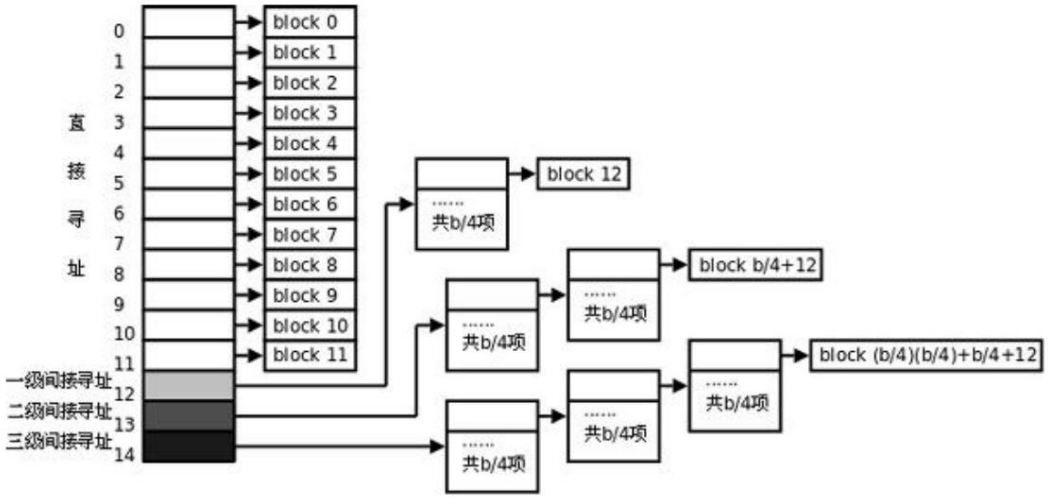
```
/**
 * Get use ratio of ext2 filesystem
 */
unsigned long get_ext2_usage(struct super_block *super){
    unsigned long unused;
    unsigned long total;
    unsigned long used;
    double ratio;

    total = MAXBLK;
    unused = ext2_count_free_blocks(super);
    used = total - unused;
    ratio = (used * 1.0) / (total * 1.0);

    return (unsigned long)(ratio*100);
}
```

函数 clear_data_blocks 接受一个 inode 指针作为参数，释放掉该 inode 所对应文件的所有磁盘块，但是不修改 inode 的任何元数据。为了调用相关函数的方便，该函数定义在文件 inode.c 中。实现过程主要分如下几个步骤：

Step 1，将所有缓存在 page cache 中的磁盘页清除，调用内核的 truncate_inode_pages 调用：

```
/* I think that this should clear out the page cache. */
truncate_inode_pages(&i_node->i_data, 0);
```

Step 2，linux 的 inode 的组织结构是这样的：一个 inode 对应 15 个磁盘块索引，前 12 个磁盘索引直接对应一个磁盘块。第 13 个索引指向一个一级索引表，该索引表索引到若干个磁盘块；第 14 个索引指向一个二级索引表，该索引表索引到若干一级索引表；第 15 个索引指向一个三级索引表，该索引表索引到若干二级索引表。如下图：



因此在释放一个 inode 对应的所有磁盘块时主要分四步，即分别释放各级索引表对应的磁盘块。

（1）直接索引的 12 个磁盘块内核提供了接口 ext2_free_data：

```
/*
 * Clear out the direct 12 blocks.
 * i_data + offsets[0] - should be 0
 * i_data + EXT2_NDIR_BLOCKS - end of direct blocks
 */
ext2_free_data(i_node, i_data + offsets[0], i_data + EXT2_NDIR_BLOCKS);
```

（2）间接索引的三个表项使用 ext2_free_branches。先分别检查该 inode 是否含有间接索引的 blocks，如果是则进行空间释放操作。代码如下：

```
/* Clear out the indirect blocks. */
if (i_data[EXT2_IND_BLOCK] != 0) {
    temp_data = i_data[EXT2_IND_BLOCK];
    i_data[EXT2_IND_BLOCK] = 0;
    if (temp_data != 0) {
        ext2_free_branches(i_node, &temp_data, &temp_data + 1,
                1);
    }
}
/* Clear out the double indirect blocks. */
if (i_data[EXT2_DIND_BLOCK] != 0) {
    temp_data = i_data[EXT2_DIND_BLOCK];
    i_data[EXT2_DIND_BLOCK] = 0;
    if (temp_data != 0) {
        ext2_free_branches(i_node, &temp_data, &temp_data + 1,
                2);
    }
}
/* Clear out the triple indirect blocks. */
if (i_data[EXT2_TIND_BLOCK] != 0) {
    temp_data = i_data[EXT2_TIND_BLOCK];
    i_data[EXT2_TIND_BLOCK] = 0;
    if (temp_data != 0) {
        ext2_free_branches(i_node, &temp_data, &temp_data + 1,
                3);
    }
}
```

以上即为几个自定义的函数，在接下来的操作中会用到。
下面来看驱逐机制的具体实现：

以 ext2_evict 为例，实现的过程中大致分为如下几步：
Step 1：检查该 inode 是否已经被打开。如果被打开则不进行驱逐（显然我们是不能将用户正在用手机浏览的图片不加判断地直接驱逐的）。函数 is_inode_open 实现如下，即遍历所有该 inode 对应的 dentry：

```
/**
 * Check if this inode is opened yet
 */
static int is_inode_open(struct inode *i_node){
    int open;
    struct dentry *curr_dentry;

    open = 0;
    hlist_for_each_entry(curr_dentry, &i_node->i_dentry, d_alias) {
        if (curr_dentry->d_count > 0) {
            printk("Dentry is open. %d\n", curr_dentry->d_count);
            open = 1;
        }
    }
    return open;
}
```

Step 2：检查该文件是否已经被驱逐，如果已经被驱逐则返回，否则标记该文件已经被驱逐。标记是通过 kernel 提供的拓展属性机制（xattr）实现的。拓展属性是不存在于 inode 中的另一种形式的元数据。为此我们设置一种叫做“evicted”的拓展属性，用来标记该文件是否已经被驱逐（这个属性在打开文件时会用到，如果发现即将打开的文件已经被驱逐，则调用 ext2_fetch 从云服务器取回数据）：

```
/* Judge evicted already? */
buffer_size = 1;
if(!i_node) return EVICT_NOTEXIST;
if(ext2_xattr_get(i_node, e_name_index, name, buffer, buffer_size) == -ENODATA){
    buffer[0] = 0;
    flags = XATTR_CREATE;
    ext2_xattr_set(i_node, e_name_index, name, buffer, buffer_size, flags);
}else{
    if(buffer[0] == 1){
        printk("already evicted\n");
        return EVICT_REPEAT;
    }
}
```

Step 3：由给定 inode 获取文件数据。主要分如下几个步骤：

（1）检查该 inode 对应的文件页是否在 page cache 中。调用内核接口 find_get_page，该函数接受 inode->mapping 和 index 两个参数，检查索引为 index 的 page 是否在缓存中。

（2）如果不在缓存中，则调用 page_cache_alloc_cold 分配一个 page，然后调用 add_to_page_cache_lru 将之添加到 page cache 中，最后调用 mpage_readpage 读取 page 所对应的文件数据。

（3）调用 kmap 将相应的 page 映射到当前的虚拟地址空间，即由一个物理地址直接得到对应当前进程的虚拟地址。

（4）创建一个长度为 file size 的字符数组，每读一页就使用 memcpy 将该页的内容附加到该字符数组的末尾。

（5）返回该字符数组，或根据不同的情况返回相应的错误码。

关键代码如下：

```
curr_page = find_get_page(mapping, index);
printk("before alloc curr_page = %p\n", curr_page);
if (!curr_page) {
    curr_page = page_cache_alloc_cold(mapping);
    printk("after alloc, curr_page = %p\n", curr_page);
    if (!curr_page) {
        printk("NULL page.\n");
        kfree(file_buffer);
        return NULL;
    } else {
        error = add_to_page_cache_lru(curr_page, mapping, index, GFP_KERNEL);
        if (error) {
            printk("error adding lru.\n");
            kfree(file_buffer);
            return NULL;
        }
        printk("added page to lru cache.\n");
        printk("curr_page = %p\n", curr_page);
        printk("Found page and preparing to read.\n");
        mpage_readpage(curr_page,
                ext2_get_block);
    }
}

printk("Found and read page.\n");
/*
 * Based on filemap.file_read_actor
 */
offset = 0;
kaddr = kmap(curr_page);
if (remaining < PAGE_SIZE) {
    memcpy((void *) curr_addr, kaddr, remaining);
    remaining = 0;
} else {
    memcpy((void *) curr_addr, kaddr + offset,
            PAGE_SIZE);
    remaining -= PAGE_SIZE;
}
kunmap(curr_page);
page_cache_release(curr_page);
curr_addr += PAGE_SIZE;
index++;
```

Step 4：创建 socket 并向 server 发出 send 请求，准备发送文件数据。用户空间创建 socket 的流程前文叙述 server 端编写时已经提及，内核空间的操作大同小异，只是为了内核的安全考虑做了一些附加的操作。值得一提的是，版本为 3.10 的 linux kernel 已经默认关闭了内核空间创建 socket 的开关，如果想无视之，必须在配置文件中更改相关配置选项。以发送为例，具体代码如下：

```c
/**
 * Send message to server specified by a socket object.
 */
static int ext2_send_msg(struct socket *sock, char *send_buf, int len){
    struct kvec vec;
    struct msghdr msg;
    int ret;

    vec.iov_base = send_buf;
    if(len == -1)
        vec.iov_len = strlen(send_buf);
    else
        vec.iov_len = len;

    memset(&msg, 0, sizeof(msg));

    /* Send message to the server */
    ret = kernel_sendmsg(sock, &msg, &vec, 1, len); /* send message */
    if(ret < 0){
        printk("client: kernel_sendmsg error!\n");
        return ret;
    }else{
        printk("client: ret = %d\n", ret);
    }
    printk("client sned ret: %d\n : ", ret);

    return ret;
}
```

Step 5：释放本地文件数据

直接调用前文所述的 ext2_free_data 接口，释放掉所有的 block。

Step 6：关闭 socket 文件描述符。

ext2_fetch 的实现类似，只是二者一个是读文件，另一个是写文件。因此只有相应 IO 操作有不同。我编写了 write_blocks 函数，用来处理 fetch 过程中向文件中写数据的操作。fetch 的实现分为如下步骤：

Step 1：检查“evicted”拓展属性。如果未被驱逐则直接返回。

Step 2：创建 socket，向 server 发起 TCP/IP 连接请求。

Step 3：从 server 接收到数据，并将之写入文件块中。首先根据 inode 元数据获得该文件大小，然后调用 ext2_get_block 获得相应的文件块，并将之映射到一个 buffer_head（以下简称 bh）。然后调用 sb_getblk 为该 bh 分配相应的磁盘块。bh 的 data 字段存储了该 bh 对应的数据。使用 memcpy 将从 server 收到的数据的相应段落拷贝到该缓冲区。然后通过 set_buffer_toupdate 设置该 bh 的 toupdate 标记，并调用 mark_buffer_dirty 将该缓冲区标记为 dirty。最后使用 sync_dirty_buffer 将已经改动的 bh 写回到磁盘。核心代码如下：

```
mutex_lock_nested(&i_node->i_mutex, I_MUTEX_QUOTA);
while (curr_addr < end_addr) {
    printk("executing write block loop.\n");
    tocopy = blocksize < ( end_addr - curr_addr) ? blocksize : (end_addr - curr_addr);

    tmp_bh.b_state = 0;
    ret_val = ext2_get_block(i_node, curr_block, &tmp_bh, 1);

    if (ret_val < 0) {
        mutex_unlock(&i_node->i_mutex);
        return ret_val;
    }

    bh = sb_getblk(sb, tmp_bh.b_blocknr);
    if (!bh) {
        ret_val = -EIO;
        mutex_unlock(&i_node->i_mutex);
        return ret_val;
    }

    lock_buffer(bh);

    memcpy(bh->b_data, file_data, tocopy);
    flush_dcache_page(bh->b_page);
    set_buffer_uptodate(bh);
    mark_buffer_dirty(bh);

    unlock_buffer(bh);
    sync_dirty_buffer(bh);
    brelse(bh);
    curr_addr += tocopy;
    curr_block++;
}
```

Step 4：修改 ext2 文件系统打开文件的基础操作，即先检查该 inode 对应的磁盘块是否被驱逐。核心代码如下：

```
/**
 * When open a file
 */
int ext2_open(struct inode *i_node, struct file *file){
    //if xattr evicted if 1 do ext2_fetch(i_node)
    int buffer[0];
    int buffer_size;
    const char *name = "evicted";
    int e_name_index = 0;
    int ret;
    int flags;

    buffer_size = 1;
    if(ext2_xattr_get(i_node, e_name_index, name, buffer, buffer_size) == -ENODATA){
        buffer[0] = 0;
        flags = XATTR_CREATE;
        ext2_xattr_set(i_node, e_name_index, name, buffer, buffer_size, flags);
    }else if(buffer[0] == 1){
        if(!(ret = ext2_fetch(i_node))){
            printk("fetch failed [#%d]\n", ret);
            goto fetch_fail;
        }
    }
    return dquot_file_open(i_node, file);
fetch_fail:
    return -ENODATA;
}
```

以上便是 evict 和 fetch 的主要实现流程。值得一提的是，在这个过程中要特别注意同步问题，即合适地使用内核相应的锁结构，在合适的场合进行上锁与解锁。

Part D 监视文件系统

前文所述的 evict_fs 即为监视文件系统的接口。在 evict_fs 文件系统中实现了 clockhand 算法。在一个给定的 super_block 中，一个 inode 可以根据一个 index 索引到。因此我将 clockhand 对应的整数值作为根目录 inode 的一个拓展属性“clockhand”来进行记录。如 writeup 中所说，在相应的情形下进行 evict 操作。此步骤中需要先修改 mount 的参数，以支持 srv，wl，wh，evict 等几个文件驱逐阈值。如下图：

```
/* Set evict mount options */
case Opt_srv:
    sbi->srv = htonl(in_aton(args[0].from));
    break;
case Opt_wh:
    if(match_int(&args[0], &option))
        return 0;
    sbi->wh = option;
    break;
case Opt_wl:
    if(match_int(&args[0], &option))
        return 0;
    sbi->wl = option;
    break;
case Opt_evict:
    if(match_int(&args[0], &option))
        return 0;
    sbi->evict = option;
```

具体 evict_fs 的实现参见附件中的代码。

在系统执行时需要保证一个守护进程一直在监视 ext2 分区的磁盘使用率并执行相应的驱逐操作。使用 kernel 的 kthread_run 可以启动一个线程。同时需要添加__init 关键字修饰，并用 late_initcall_sync 保证系统初始化时执行相关代码。(writeup 中说的是 fs_initcall，但是我没有尝试成功，查了资料似乎是初始化顺序的问题，于是改成了比较靠后的初始化顺序即 late_initcall_sync)。代码如下：

```
/**
 * A kernel monitor thread scanning ext2 filesystem
 * every minute.
 */
static int do_evictd(void* args){
    struct super_block *super = NULL;

    /* Get super block */
    super = ext2_get_super_block();
    if(super){
        printk("super block found\n");
        printk("[name] %s\n", super->s_type->name);
        printk("[size] %ld\n", super->s_blocksize);
    }
    else{
        printk("ext2 super block not found\n");
    }

    printk("kfs_evictd started\n");
    while(1){
        msleep(60000);
        printk("kfs_evictd working...\n");
        ext2_evict_fs(super);
    }
    return 0;
}

/**
 * Create a kernel monitor thread when booting.
 */
static __init int kfs_evictd(void){
    const char *name = "kfs_evictd";
    struct task_struct *task;
    task = kthread_run(do_evictd, NULL, name);

    if(task)
        printk("[kfs] create success!\n");
    else
        printk("[kfs] fail to create!\n");

    return 0;
}
late_initcall_sync(kfs_evictd);
```

Part E 测试 Cloud FS

参见附件中 output 文件

总结

本 lab 是本学期安卓操作系统实习的收官之作（加分项的没时间写了 QWQ）。这个 lab 中实现了一个真正意义上可以使用的云文件系统，用到了 linux kernel 中的许多机制，比如拓展属性，socket 通信，vfs 层以及更加底层的文件操作等。大大提高了阅读和编写工程代码的能力，以及查阅资料的能力。
总体来讲，Android 操统实习的这几个 Lab 让我对操作系统有了更深的认识，这门课是好课，只是不适合在一个课很多的学期选择。