Lab4 Cloud File System

qingyunqu

Part A 系统自启动线程

- 在 `include/linux/fs.h` 中的 `struct super_operations` 中添加： `int (*evict_fs)(struct super_block *super);`
- 在 `fs` 文件夹中新增文件 `evictd.c` ，并添加 `Makefile` ，并在其中实现内核自启动的监控线程
- 使用 `fs_initcall(init_evictd)` 和 `fs_exitcall(exit_evictd)` 实现系统自启动和结束
- 在 `init_evictd()` 中，调用`kthread_run() `生成新的内核线程
- 在 `exit_evictd()` 中，调用 `kthread_stop()` 结束线程
- 在 `do_evictd()` 中，循环判断是否接收到了结束信号，并调用 `iterate_supers()` 这个内置的遍历所有文件系统的函数
- 在 `try_evictd()` 中判断＝当前文件系统是否支持 `evict_fs` 操作，若支持便调用它
- 简化代码如下：

```
truct task_struct *task_evictd = NULL;
static void try_evictd(struct super_block *super,void *data)
{
        if(super->s_op->evict_fs != NULL)
                super->s_op->evict_fs(super);
}
static int do_evictd(void *data)
{
        while(!kthread_should_stop()){
                iterate_supers(try_evictd, NULL);
                ssleep(60);
        }
        return 0;
}
static int __init init_evictd(void)
{
        task_evictd = kthread_run(do_evictd,NULL,"kfs_evictd");
        return 0;
}
static void __exit exit_evictd(void)
{
        if(task_evictd){
                kthread_stop(task_evictd);
                task_evictd = NULL;
        }
}
fs_initcall(init_evictd);
fs_exitcall(exit_evictd);
```

Part B 挂载文件系统

- 首先按照WriteUp附录所说的将sdcard挂载上去
- 在 `fs/ext2/` 中新增文件 `ext2_evicted.c` 文件，并修改Makefile
- 在 `fs/ext2/super.c` 中的 `struct super_operations ext2_sops` 中添加 `.evict_fs = ext2_evict_fs`
- 根据 `super.c` 中已有的代码，修改其中的 `enum{}` 和 `match_table_t tockens` ，使其增加我们所需要的参数(Opt_xxx)
- 在 `parse_options()` 中添加对应的 `case xxx:` ，参数的解析:参数的解析由 `ext2_evicted.c` 中的 `clfs_parse_options()` 函数实现,只是简单的字符串处理,详情见代码
- 部分代码如下:

```
char server_ip[20] = "10.0.2.2" ;
unsigned short server_port = 8888 ;
int high_watermark = 95 ;
int low_watermark = 85 ;
int evict_target = 70 ;

static void set_wh(int wh){
        high_watermark = wh;
}
static void set_wl(int wl){
        low_watermark = wl;
}
static unsigned int atou(const char *str){
```

```
    unsigned res = 0;
    int i = 0;
    for (; str[i]; ++i){
        if (str[i] < 48 || str[i] > 57){
            break;
        }
        res = (res * 10) + str[i] - 48;
    }
    return res;
}
static void set_port(int port){
        server_port=(unsigned short)port;
}
static void set_srv(char *ip){
        int i=0;
        while(*ip != '\0'&&*ip!=':'){
                server_ip[i]=*ip;
                ip++;
                i++;
        }
        server_ip[i]='\0';
        printk("the server_ip is : %s\n",server_ip);
        if(*ip==':'){
                ip++;
                set_port(atou(ip));
        }
}
static void set_evict(int evict){
        evict_target = evict;
}
int clfs_parse_options(char *p){
        if(p[0]=='w'&&p[1]=='h'){
                set_wh(atou(p+3));
        }
        else if(p[0]=='w'&&p[1]=='l'){
                set_wl(atou(p+3));
        }
        if(p[0]=='s'&&p[1]=='r'&&p[2]=='v'){
                set_srv(p+4);
        }
        else if(p[0]=='e'){
                set_evict(atou(p+6));
        }
        return 0;
}
```

Part C 实现监控程序

- 在 `fs/ext2/ext2.h` 中添加 `extern int ext2_evict_fs(struct super_block *super);`
- 在 `ext2_evicted.c` 中添加 `ext2_evict_fs()` 函数，此时重新编译内核，在 `fs/evictd.c` 中实现的内核监控线程已经能成功调用该函数了
- `ext2_get_usage()` 的实现:通过获取文件系统的总block数和free的block数来计算总的usage

```
int ext2_get_usage(struct super_block *super){
        struct ext2_sb_info *sbi = EXT2_SB(super);
        struct ext2_super_block *es = sbi->s_es;
        unsigned long total = es->s_blocks_count;
        unsigned long used = total - ext2_count_free_blocks(super);
        return (100 * (used)) / total;
}
```

- 关于 `root` 目录的扩展属性 `clockhand` :目前暂未实现该属性，每次还是重头扫描

Part D 驱逐策略:时钟算法

- 首先是实现遍历 `inode` 的函数 `for_each_inode()` 通过传入函数指针的方式，使得对每个有实际意义的 `inode` 都调用 `ext2_judge_evict()` 来判断该 `inode` 是否为目录，是否正在被使用，以及当前文件系统的 `usage` 已经小于 `target_evict` ，符合被驱逐的条件就调用 `ext2_evict()` 来实际驱逐该 `inode` 所对应的文件
- 在 `clock_hand()` 中目前还是通过遍历每个 `inode` 来判断是否需要驱逐、是否能否驱逐
- 部分代码如下：

```
void ext2_judge_evict(struct inode *inode,void *arg)
{
```

```
        printk("ext2_judge_evict!\n");
        int value;
        int ret;
        if (inode->i_mode&S_IFDIR){
                printk("inode : %d is a dir\n",inode->i_ino);
                return;
        }
        if (inode_using(inode)){
                printk("inode : %d is in using\n",inode->i_ino);
                return;
        }
        ret = ext2_get_usage(inode->i_sb);
        if( ret < evict_target){
                printk("usage : %d < evict_target : %d\n",ret,evict_target);
                return;
        }
        ext2_evict(inode);
}
static int clock_hand(struct super_block *super)
{
        printk("clock_hand!\n");
        for_each_inode(super,ext2_judge_evict,NULL);
        return 0;
}
```

Part E 服务器代码编写

- 新建运行在宿主机上的代码 `clfs_server.c` ,在其中实现云服务器的功能
- 主要为 `socket` 的编程,监听本机的8888端口,并对不同的命令做出对应的动作
- 首先是接收一个命令,判断命令的类型:
- 如果是 `CLFS_PUT` 命令,发送一个 `CLFS_OK` 的状态字,接着接收来自客户端的数据,定义了最多接收的大小为 `BUFSIZE` ,存入缓冲区,接着打开或新建对应的文件,存入数据(注:数据存入的方式为 `APPEND`),再发送正常接收的状态字
- 如果是 `CLFS_GET` 命令,打开对应的文件,如果打开错误,就返回 `CLFS_INVAL` 的状态字;如果正常打开,就返回 `CLFS_OK` 的状态字,并读取接收到的 `size` 大小的内容并发送给客户端,此处没有考虑 `size` 过大的情况,一切由 `BUFSIZE` 限定
- 如果是 `CLFS_RM` 命令,打开失败同样返回 `CLFS_INVAL` 的状态字;打开正常便返回 `CLFS_OK` 的状态字,同时关闭打开文件,并删除文件
- 存在的问题:没有使用多线程,但是可以被循环连接
- 简化代码如下:

```
int main(){
        my_addr.sin_family=AF_INET;
        my_addr.sin_addr.s_addr=inet_addr("0.0.0.0");//allow connecting from all local address
        my_addr.sin_port=htons(8888);
    server_sockfd=socket(AF_INET,SOCK_STREAM,0);
    bind(server_sockfd,(struct sockaddr *)&my_addr,sizeof(struct sockaddr));

        listen(server_sockfd,5);

        while(1){  //connect
                client_sockfd=accept(server_sockfd,(struct sockaddr *)&remote_addr,&sin_size);

                while((len=recv(client_sockfd,buf,BUFSIZ,0))>0){  //request
                        struct clfs_req *req=(struct clfs_req *)buf;
                        int inode=req->inode;
                        enum clfs_type type=req->type;
                        char file[100];
                        sprintf(file,"/home/lyq/course_file/androidcourse_zyw/test_for_lab4/clfs_store/%d.dat\0",inode);
                        int size=req->size;
                        enum clfs_status state;
                        int fd;
                        if(type==CLFS_PUT){
                                state=CLFS_OK;
                                send(client_sockfd,(char *)&state,sizeof(enum clfs_status),0);
                                len=recv(client_sockfd,buf,size,0);
                                printf("CLFS_PUT : get %s\n",buf);
                                fd=open(file,O_RDWR|O_CREAT|O_APPEND,0666);//APPEND or TRUNC ?
                                if(fd<0){
                                        printf("CLFS_PUT error : can not open file\n");
                                        perror("ERROR");
                                        state=CLFS_ACCESS;
                                        send(client_sockfd,(char *)&state, sizeof(enum clfs_status),0);
                                }
                                else{
                                        write(fd,buf,len);
```

```
                                send(client_sockfd,(char *)&state, sizeof(enum clfs_status),0);
                        }
                        close(fd);
                        printf("CLFS_PUT : %d.dat\n",inode);
                }
                if(type == CLFS_GET){
                        state=CLFS_OK;
                        fd=open(file,O_RDONLY);
                        if(fd<0){
                                state=CLFS_INVAL;
                                send(client_sockfd,(char *)&state, sizeof(enum clfs_status),0);
                                printf("CLFS_GET error : no such file\n");
                        }
                        else{
                                send(client_sockfd,(char *)&state, sizeof(enum clfs_status),0);
                                len=read(fd,buf,size);
                                send(client_sockfd,buf,len,0);

                                close(fd);
                                printf("CLFS_GET : %d.dat\n",inode);
                        }
                }
                if(type==CLFS_RM){
                        state=CLFS_OK;
                        fd=open(file,O_RDONLY);
                        if(fd<0){
                                state=CLFS_INVAL;
                                send(client_sockfd,(char *)&state, sizeof(enum clfs_status),0);
                        }
                        else{
                                send(client_sockfd,(char *)&state, sizeof(enum clfs_status),0);
                                close(fd);
                                delete_file(file);
                                printf("CLFS_RM : %d.dat\n",inode);
                        }
                }
        }
    }
    return 0;
}
```

Part F 驱逐和取回操作

- （原本应该如此实现，但是由于未调通代码，注：我已经尝试了许多均未成功，故而我换了方法）首先在内核 `socket API` 的基础上,实现简单的 `socket` 函数: `socket_connect()` 用于建立连接, `socket_send()` 和 `socket_recv()` 用于发送和接收指定长度的报文, `socket_close()` 用于结束 `socket`，再在上述接口的基础上,实现简单的 `clfs_put()` 和 `clfs_get()` 函数,分别对应两个命令的具体实现,都是原理上的实现,鲁棒性不高
- 新的已调通的实现方案：通过 `syscalls.h` 中提供的 `sys_socket` 系列系统调用，直接通过标准的系统调用的（原本应该是用户态使用的）来实现 `clfs_put()` 和 `clfs_get()`，此处验证可行
- 然后实现了清除本地某文件的磁盘的函数 `rm_diskfile()` 用于删除本地文件,具体就是以 `O_TRUNC` 方式调用 `sys_open` 来清除文件内容而保留 `inode`,与此同时,为了解决 `inode` 信息在调用 `sys_open` 前后的变化,实现了 `sava_i_data()` 和 `recover_i_data()` 来保存和恢复重要的 `inode` 信息,与此同时,在调用 `ext2_evict()` 之前已经保证了本段代码是最后一个打开该文件的程序,故而只需要关闭文件描述符就可以将内存中文件缓存清除了
- `get_filename_by_inode()` 的实现:此功能暂时有BUG，未能调试通过
- 在 `ext2_evict()` 的实现中,,首先判断该 `inode` 对应的文件是否已经被 `evicted` ,如果是则返回-1,代表驱逐失败,否则继续,调用 `get_filename_by_inode()` 来获取文件完整的路径,然后调用 `clfs_put()` 来将文件发送给云端,接着清除本地的文件内容,然后修改或创建该文件的 `evicted` 属性
- 在 `ext2_fetch()` 的实现中,与上述差不多,就不再赘述
- Ext2文件系统本身的 `open()` 函数的修改:在 `fs/ext2/file.c` 中修改 `ext2_file_operations` 中 `.open` 指向的函数指针，对 `dquot_file_open()` 多进行一次包装，成为 `do_ext2_oppenfile()` ,将 `.open` 的函数指针指向该函数即可，该函数通过判断该 `inode` 是否已经 `evicted` 来决定是否执行 `evict_fetch()`
- 部分代码如下：

```
static int clfs_put(char *filename , unsigned long i_num, loff_t size)
{
        printk("clfs_put!\n");
        int sockfd;
        struct sockaddr_in remote_addr;
        //int sin_size;
        memset(&remote_addr,0,sizeof(remote_addr));

        remote_addr.sin_family=AF_INET;
        remote_addr.sin_addr.s_addr=in_aton(server_ip);
        remote_addr.sin_port=htons(server_port);

        if((sockfd=sys_socket(AF_INET,SOCK_STREAM,0))<0){
```

```
                printk("error : sys_socket()\n");
                return -1;
        }
        if(sys_connect(sockfd,(struct sockaddr *)&remote_addr,sizeof(struct sockaddr))<0){
                printk("error : sys_connect()\n");
                goto close_sockfd;
        }
        printk("connect to server success\n");

        int len=0;
        struct clfs_req req;
        enum clfs_status state;
        req.type=CLFS_PUT;
        req.inode=i_num;
        req.size=size;
        sys_send(sockfd,(char *)&req,sizeof(struct clfs_req),0);
        len = sys_recv(sockfd,(char*)&state,sizeof(enum clfs_status),0);

        if(state!=CLFS_OK){
                printk("CLFS_PUT request error\n");
                goto close_sockfd;
        }
        int fd;
        fd = sys_open(filename,O_RDONLY,0777);
        if(unlikely(fd<0)){
                printk("open file error!\n");
                goto close_sockfd;
        }
        len = sys_read(fd,buf,size);
        printk("read %d bytes from file\n",len);
        len = sys_send(sockfd,buf,len,0);
        len = sys_recv(sockfd,(char*)&state,sizeof(enum clfs_status),0);
        if(state!=CLFS_OK){
                printk("CLFS_PUT send error\n");
                goto close_fd;
        }
        sys_close(fd);
        sys_close(sockfd);
        return len;
close_fd:
        sys_close(fd);
close_sockfd:
        sys_close(sockfd);
        return -1;
}
static int clfs_get(char *filename, unsigned long i_num, loff_t size)
{
        printk("clfs_get!\n");
        int sockfd;
        struct sockaddr_in remote_addr;

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

        remote_addr.sin_family=AF_INET;
        remote_addr.sin_addr.s_addr=in_aton(server_ip);
        remote_addr.sin_port=htons(server_port);

        if((sockfd=sys_socket(AF_INET,SOCK_STREAM,0))<0){
                printk("error : sys_socket()\n");
                return -1;
        }
        if(sys_connect(sockfd,(struct sockaddr *)&remote_addr,sizeof(struct sockaddr))<0){
                printk("error : sys_connect()\n");
                goto close_sockfd;
        }
        printk("connect to server success\n");

        int len=0;
        struct clfs_req req;
        enum clfs_status state;
        req.type=CLFS_GET;
        req.inode=i_num;
        req.size=size;
        sys_send(sockfd,(char *)&req,sizeof(struct clfs_req),0);
        len = sys_recv(sockfd,(char*)&state,sizeof(enum clfs_status),0);

        if(state!=CLFS_OK){
```

```
                printk("CLFS_GET request error\n");
                goto close_sockfd;
        }
        len = sys_recv(sockfd,buf,size,0);
        printk("recv %d bytes from server\n");

        int fd;
        fd = sys_open(filename,O_WRONLY | O_TRUNC,0777);
        if(unlikely(fd<0)){
                printk("open file error!\n");
                goto close_sockfd;
        }
        sys_write(fd,buf,len);

        sys_close(fd);
        sys_close(sockfd);

        return len;
close_fd:
        sys_close(fd);
close_sockfd:
        sys_close(sockfd);

        return -1;
}
int ext2_evict ( struct inode * i_node )
{
        printk("ext2_evict!\n");
        unsigned long i_num;
        loff_t size;
        int ret;
        int evicted;
        mm_segment_t oldfs;

        i_num = i_node -> i_ino;
        size = i_node -> i_size;

        if(size == 0){
                printk("inode : %d 's size = 0 \n",i_node->i_ino);
                return -1;
        }

        evicted = inode_evicted(i_node);
        if(unlikely(evicted < 0)){
                return evicted;
        }
        if(evicted == 1){
                printk("inode : %d is evicted!\n");
                return -1;
        }

        //set_filename(i_node);

        save_i_data(i_node);

        ret = clfs_put(filename,i_num,size);
        if(unlikely(ret < 0)){
                printk("ext2_evict failed!\n");
        }

        rm_diskfile(filename);
        recover_i_data(i_node);

        evicted = 1;
        set_inode_evicted(i_node,evicted);

        return 0;
}
int ext2_fetch ( struct inode * i_node )
{
        unsigned long i_num;
        loff_t size;
        int ret;
        int evicted;

        i_num = i_node -> i_ino;
        size = i_node -> i_size;
```

```
        //set_filename(i_node);

        save_i_data(i_node);

        ret = clfs_get(filename,i_num, size);
        if(unlikely(ret < 0)){
                printk("clfs_get failed!\n");
        }

        recover_i_data(i_node);

        evicted = 0;
        set_inode_evicted(i_node,evicted);

        return 0;
}
```

存在的不足

- `clfs_server.c` 中没有实现多线程接受连接
- 传输过程中没有区分文本文件和Bin,并且没有实现单个大文件的分段传输,只是在原理上的实现传输,传输的鲁棒性有待提高(双方的)
- 对于通过 `inode` 获取文件完整路径名的函数还未调试通过，有待解决
- `clockhand` 属性和 `scantime` 属性等未添加， `second chance` 算法未完全实现
- 文件 `evict` 和 `fetch` 的过程中可能出现死锁的情况