lab 5: 虚拟文件系统

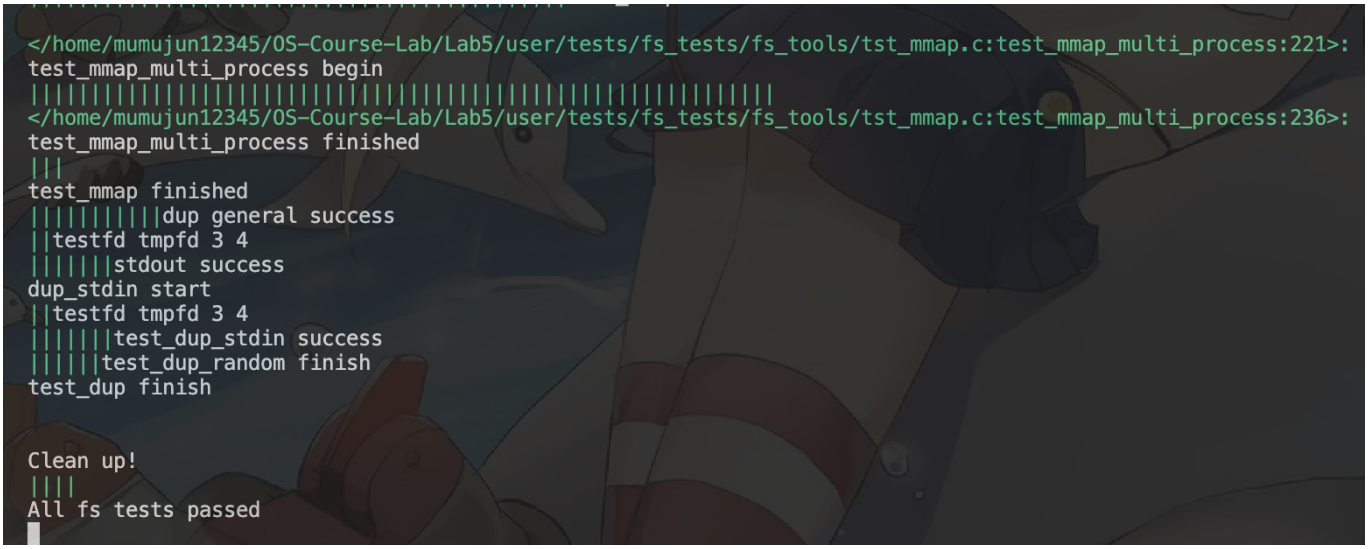
共同阅读：IPADS OS Course Lab Manual

目录

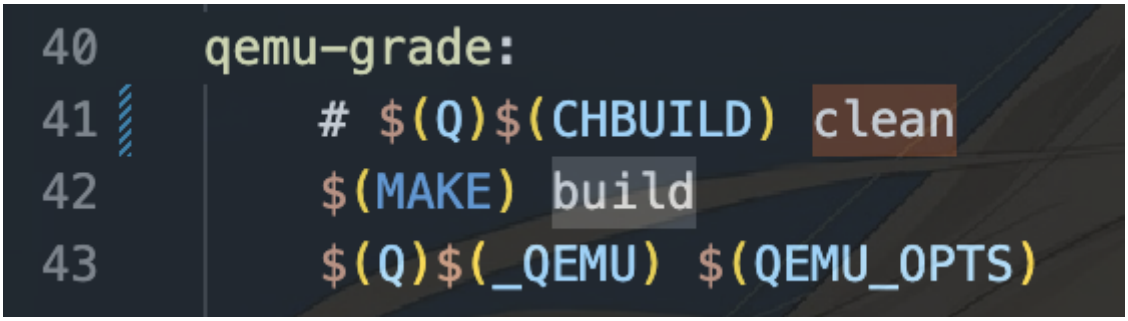
- 尽可能减少时间损耗
- Posix适配
- FSM
- FS_Base

减少时间

在你成功进行了所有实验后，执行`make qemu`后出现以下类似场景后说明你已经成功了。



为了获得更快的执行速度，你可以进行以下操作：回到实验根目录，修改文件 `Scripts/extras/lab5.mk` 中的Line 41:（也就是qemu-grade）处。注释掉clean所在的行。



接着，你需要对脚本`Scripts/extras/lab5/grader.sh`进行修改。主要修改的地方如下：

注释掉Line 25，36，49，61. 修改Line 27，38，51，63行后-t的时间为你成功时需要的最大时间。这里笔者采用了240s。如果你不想注释掉Line 25，36，49，61，则需要按照make clean后再make qemu进行计时。在没进行注释的情况下，笔者需要设置时间为840s。（毕竟是qemu套amd64套qemu套aarch64）

```
# make distclean &> /dev/null
make build &> /dev/null
${SCRIPTS}/capturer.py -f ${LABDIR}/scores-part1.json -t 240 make qemu-grade 2> /dev/null
```

最后，修改总体超时时间，修改的地方在lab5目录下。`Makefile`

```
LAB := 5
TIMEOUT := 960

include $(CURDIR)/../Scripts/lab.mk

```

Posix适配

> 练习一：阅读 `user/chcore-libc/libchcore/porting/overrides/src/chcore-port/file.c` 的 `chcore_openat` 函数，分析 ChCore 是如何处理 `openat` 系统调用的，关注 IPC 的调用过程以及 IPC 请求的内容。

阅读源码我们可以发现，在进行`openat`系统调用的过程中，我们总共经历了两次ipc调用。接下来我们详细分析该系统调用的执行过程。

1. 传入参数分别是路径描述符(dirfd, 可用于判断路径是否存在),路径名称, flags, 以及一些文件的属性（文件类型和权限等）信息mode。
2. 首先为我们打开的路径分配一个文件描述符，这将会被用于构建从虚拟文件系统的文件标识符真实文件系统的文件标识符的映射。`alloc_fd()`
3. 接着，利用传入的路径标识符和需要查找的当前目录下的文件名生成完整的路径。`generate_full_path()`
4. 接着进行**第一次IPC调用**，目的是为了根据我们目前的虚拟文件完整路径，通过ipc来获取真实文件系统内对应文件的路径信息和挂载信息。`parse_full_path()`。进行ipc通信时，利用文件系统请求(`struct fsm_request *`)创建ipc通信信息，请求操作为`FSM_REQ_PARSE_PATH`，生成注册回调函数后调度进行ipc请求。
5. 接着，利用上面ipc请求返回的挂载信息，准备进行**第二次IPC调用**。本次IPC调用才是真正进行文件打开操作的系统调用。利用挂载信息填充进行的ipc请求操作(`FS_REQ_OPEN`)和进行操作的目标文件在真实文件系统的路径后，进行ipc调用打开文件。

这就是chcore虚拟文件系统进行`chcore_openat`的两次ipc调用流程。

补充：

- 查看`struct fs_request`的结构来了解虚拟文件系统中包含的各种基础文件操作方式，查看`struct fsm_request`的结构来了解文件系统中的相关结构：`user/chcore-libc/libchcore/porting/overrides/include/chcore-internal/fs_defs.h`.
- 这里仍然遵循Lab4中进行的client-server式ipc通信模式。

- 实际上是进行了如下的对接，这样可以用于适配各类文件系统：

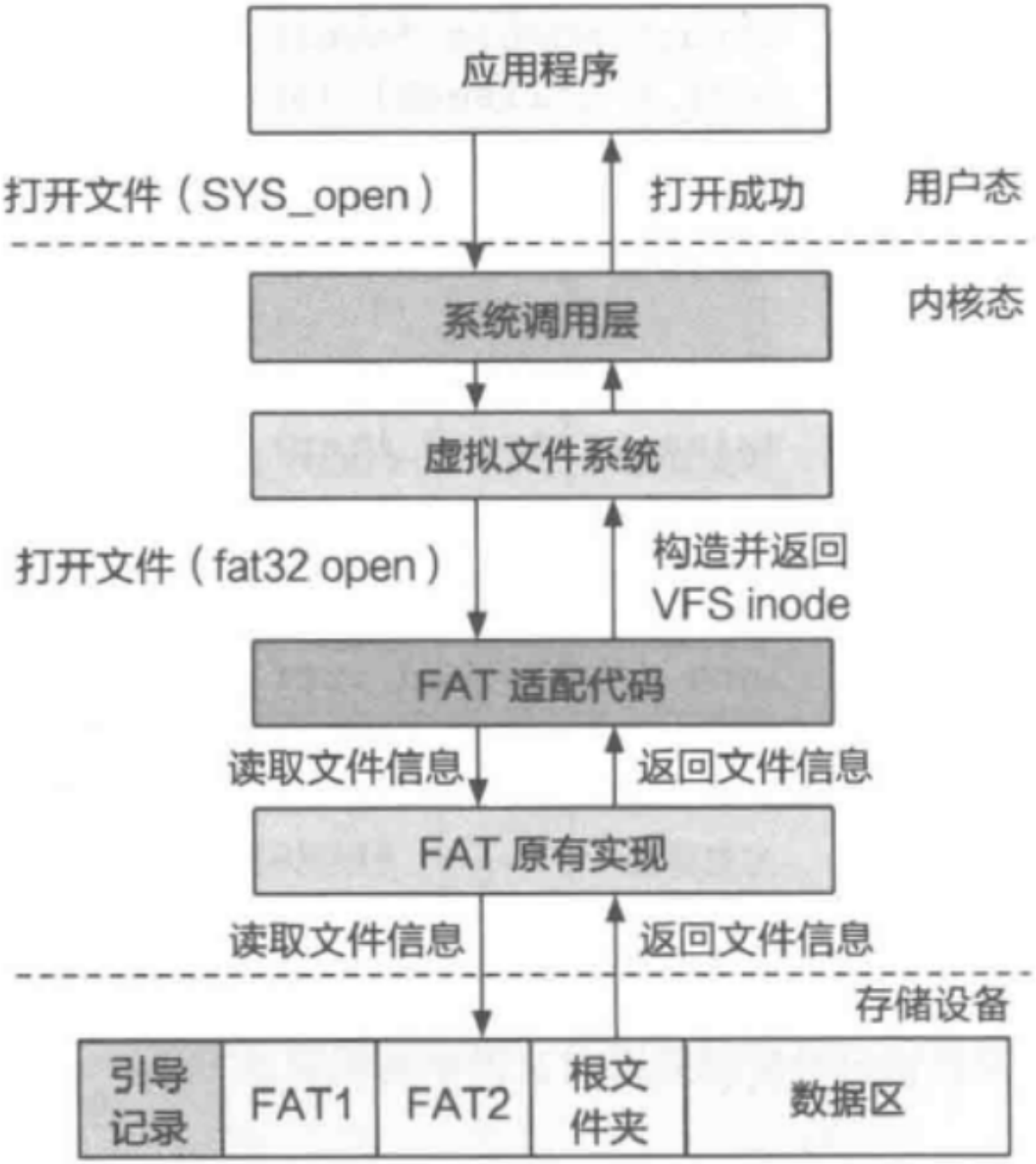


图 11.19 适配代码将 FAT32 的原有实现与 VFS 进行对接

- **未来会补充read和write等相关系统调用。**

FSM

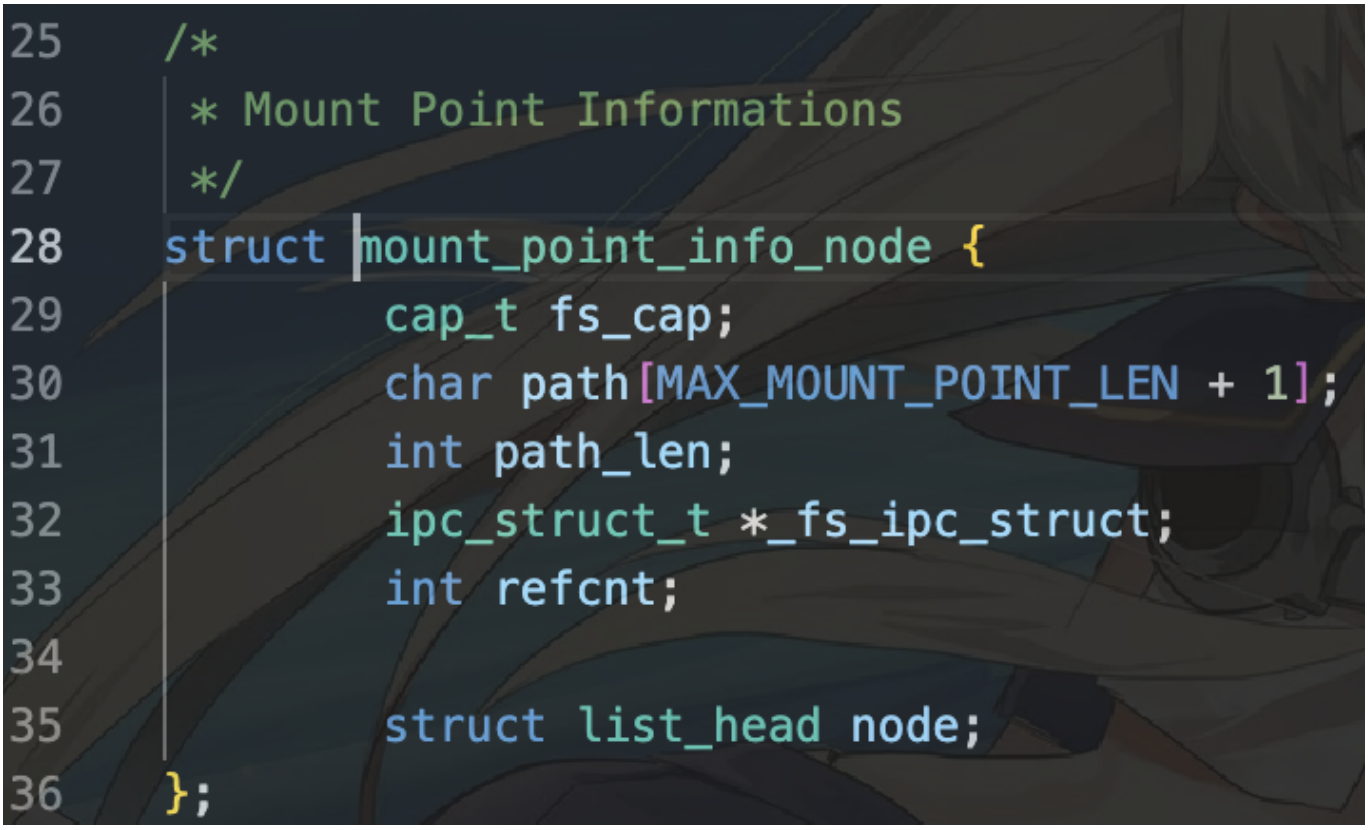
> 练习二： 实现 `user/system-services/system-servers/fsm/fsm.c` 的 `fsm_mount_fs` 函数。

综合我们的手册和注释来看，该函数中主要进行以下的工作：

1. 进行文件挂载，判断需要挂载的是文件是本地二进制文件还是储存设备内存储的文件。

2. 设定挂载点，创建挂载信息并加入表中。
3. 准备进行ipc通信，注册client，申请ipc通信。

这里我们仅需要完成的就是注册ipc进行申请ipc通信一块。 首先，注册ipc客户端需要利用服务端线程的能力组中，调用`ipc_register_client`参数获得对应的ipc通信结构，填充到挂载点对应的ipc通信结构中。我们来查看需要使用的挂载点`mp_node`，具有如下结构：(`user/system-services/system-servers/fsm/mount_info.h`)



```
25  /*
26   * Mount Point Informations
27   */
28  struct mount_point_info_node {
29          cap_t fs_cap;
30          char path[MAX_MOUNT_POINT_LEN + 1];
31          int path_len;
32          ipc_struct_t *_fs_ipc_struct;
33          int refcnt;
34
35          struct list_head node;
36  };
```

我们可以发现我们需要填充的就是其中的`ipc_struct_t *_fs_ipc_struct`字段。该字段需要通过将服务端能力组信息传入函数`ipc_register_client`中获取。我们来回顾lab4中的说法：

> 2. IPC客户端线程调用 `ipc_register_client` （定义在 `user/chcore-libc/musl-libc/src/chcore-port/ipc.c` 中）来申请建立IPC连接。
>   - 该函数仅有一个参数，即IPC服务器的主线程在客户端进程cap_group中的capability。该函数会首先通过系统调用申请一块物理内存作为和服务器的共享内存（即图中的Shared Memory）。
>   - 随后调用 `sys_register_client` 系统调用。该系统调用实现在 `kernel/ipc/connection.c` 当中，该系统调用会将刚才申请的物理内存映射到客户端的虚拟地址空间中，然后调用 `create_connection` 创建并初始化一个 `struct ipc_connection` 类型的内核对象，该内核对象中的shm字段会记录共享内存相关的信息（包括大小，分别在客户端进程和服务器进程当中的虚拟地址和capability）。
>   - 之后会设置注册回调线程的栈地址、入口地址和第一个参数，并切换到注册回调线程运行。

再结合当前`fsm_mount_fs`中设定挂载操作点的函数`set_mount_point`(位于`user/system-services/system-servers/fsm/mount_info.c`)中，我们发现我们的服务端线程能力组就是`fs_cap`。接着不要忘记处理返回值为0，以及增加挂载函数。

> Ans:

```
147         /* Lab 5 TODO Begin (Part 1) */
148         /* HINT: fsm has the ability to request page_cache syncing and mount and
149          * unmount request to the corresponding filesystem. Register an ipc client f
150
151         /* mp_node->_fs_ipc_struct = ipc_register_client(...) */
152         mp_node -> _fs_ipc_struct = ipc_register_client(fs_cap);
153         /* Increment the fs_num */
154         fs_num++;
155         /* Set the correct return value */
156         ret = 0;
```

这样就完成了第一部分。

> 练习三： 实现 user/system-services/system-servers/fsm/fsm.c 的 IPC 请求处理函数。

根据提示，我们需要完成DEFINE_SERVER_HANDLER下的PARSE_PATH部分。我们根据注释一步步完成。

Step 1: 加锁

获取挂载点信息前，我们需要为我们的挂载点加上读者锁。因此一行可以完成：

```
pthread_rwlock_rdlock(&mount_point_infos_rwlock);
```

Step 2: 获取挂载点信息

首先我们先进行很重要的概念区分。查看struct fsm_request如下：

```
236 struct fsm_request {
237         /* Request Type */
238         enum fsm_req_type req;
239
240         /* Arguments */
241         // Means `path to parse` in normal cases. `device_name` for
242         // FSM_REQ_MOUNT/UMOUNT
243         char path[FS_REQ_PATH_BUF_LEN];
244         int path_len;
245
246         /* Arguments or Response */
247         // As arguements when FSM_REQ_MOUNT/UMOUNT, as reponse when
248         // FSM_REQ_PARSE_PATH
249         char mount_path[FS_REQ_PATH_BUF_LEN];
250         int mount_path_len;
251
252         /* Response */
253         int mount_id;
254         int new_cap_flag;
255 };
```

这里的path_len和我们所理解的**路径长度**有所不同。这里，无论是挂载点路径，还是路径的len，都指的是**盘径深度**而非真实长度。例如：`/`的len为0，`/dir/`的len则为1。以示区分，我们将路径的**字符串长度**称为长度，**路径深度**称为深度。

接下来我们需要获取挂载点的信息，调用`get_mount_point`函数，根据我们文件管理系统传来的路径及其**长度**。这样我们就有：

```
mpinfo = get_mount_point(fsm_req -> path, strlen(fsm_req -> path));
```

Step 3: 客户端能力组

接下来我们需要读取我们的挂载点id，也就是客户端（其他文件系统）的能力组。不要忘记我们的client_cap_table是全局变量，需要加上互斥锁。因此我们有：

```
pthread_mutex_lock(&fsm_client_cap_table_lock);
mount_id = fsm_get_client_cap(client_badge, mpinfo -> fs_cap);
```

接下来我们需要完成这部分获取能力组的函数，该函数位于`user/system-services/system-servers/fsm/fsm_client_cap.c`。这是一个ipc通信过程。

Step 3.1: 获取能力组节点

通过注释我们可以知道，我们需要首先获取的是能力组对应的节点。在该文件中我们又有一个全局变量：

```
struct list_head fsm_client_cap_table;
```

这样很明显我们就需要从这个链表中获取对应的能力组节点。回顾上次Lab4我们获取能力组的方式，我们需要迭代寻找在该链表中寻找属于我们进程的节点，因此我们需要用到`list.h`中定义的`for_each_in_list`函数。 然后，判断是否属于我们的client（其实就是对应的一个文件系统，一个虚拟文件系统通过挂载来适配不同的文件系统）的能力组，通过`client_badge`进行判断。最后，判断在该节点中哪一个capability是我们对应需要的（根据`struct fsm_client_cap_node`注释我们可以发现，他会收集对应的client的所有cap（一个文件系统可能会有多个挂载到虚拟文件系统的挂载点）。我们还需要迭代他的能力组表来获得我们需要的对应的节点。

```
/*
 * FSM will record all the caps of fs those are sended to some client.
 * Such information is recorded in the following structure.
 */
struct fsm_client_cap_node {
        badge_t client_badge;

        cap_t cap_table[16];
        int cap_num;

        struct list_head node;
};
```

这样我们需要走两次for循环来得到我们的cap，也就是我们的挂载点id。

```
        struct fsm_client_cap_node * node = NULL;
        for_each_in_list(node, struct fsm_client_cap_node, node,
&fsm_client_cap_table)
        {
                if(node -> client_badge == client_badge)
                {
                        for (int i = 0; i < node->cap_num; i++)
                                if (node->cap_table[i] == cap)
                                        return i;
                }
        }
        return -1;
```

默认在没有记录的情况下返回-1。我们还是按照他说的来，避免犯错。

Step 4: 对应能力组不存在

接下来需要处理挂载点能力组还没有创建的情况。首先我们先知道什么情况下挂载点不存在对应的能力组。我们已经知道了在-1时挂载点没有记录，因此我们需要首先通过ipc通信（因为现在我们在虚拟文件系统下）来给文件系统设定好挂载点信息。我们于是有：

```
                /* if mount_id is not present, we first register the cap
set the
                 * cap and get mount_id */
                if(mount_id == -1)
                {
                        // printf("[INFO_H] set with cap: %d", mpinfo ->
fs_cap);
                        mount_id = fsm_set_client_cap(client_badge, mpinfo
-> fs_cap);
                        // printf("[INFO_H] set mount_id: %d\n",
mount_id);
```

```
                        ret_with_cap = true;    // 需要返回新的能力组。
                        ipc_set_msg_return_cap_num(ipc_msg, 1);
                        ipc_set_msg_cap(ipc_msg, 0, mpinfo->fs_cap);
                }
```

Step 4.1: 设立能力组

再次前往user/system-services/system-servers/fsm/fsm_client_cap.c，和上方3.1寻找能力组也是类似的。我们首先先要进行的是遍历寻找到对应的文件系统下的能力组，然后在进行挂载点的设定。设定增加了对应文件系统能力组的挂载点的数目，并且返回对应的挂载点（也就是下标啦）。

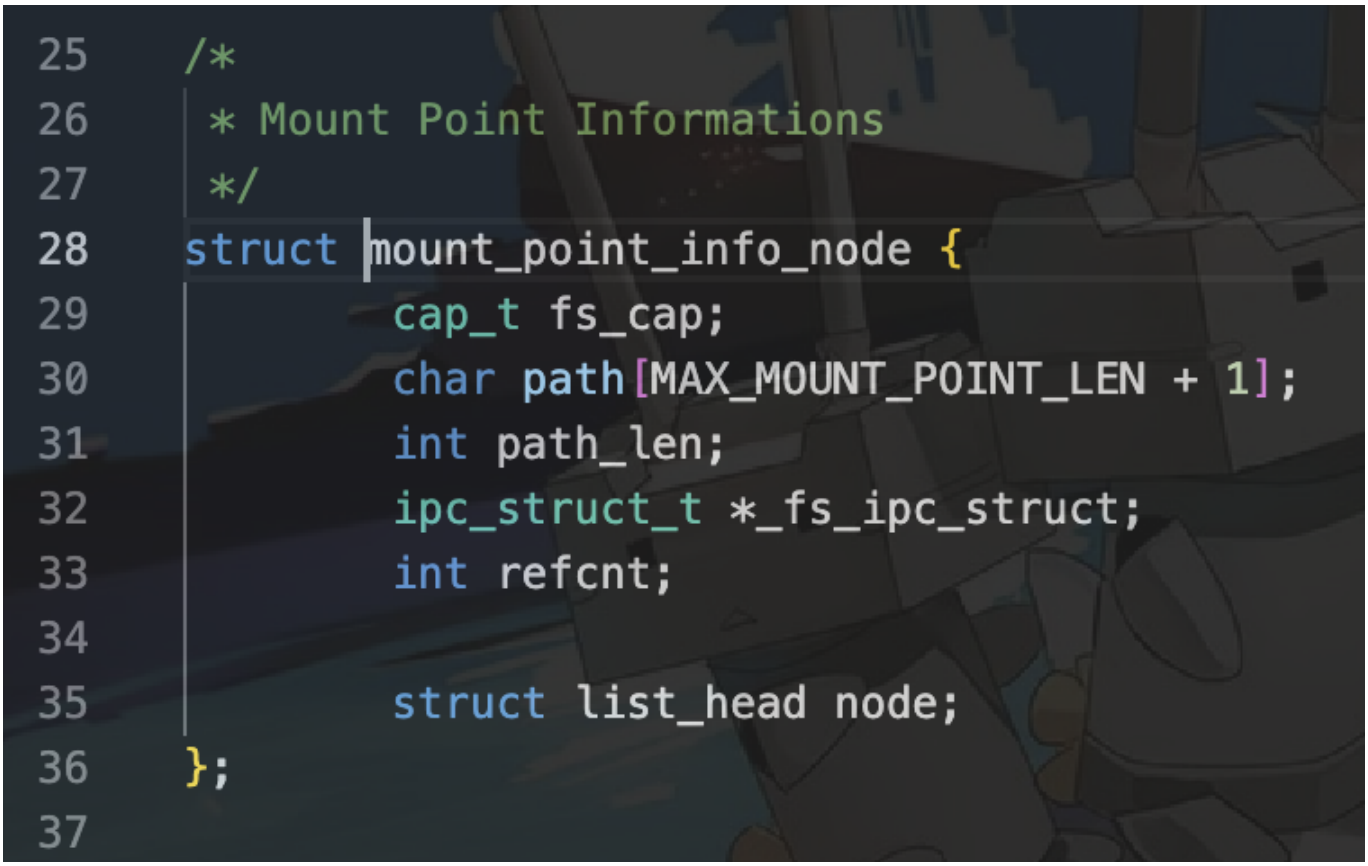
```
        struct fsm_client_cap_node * node = NULL;
        for_each_in_list(node, struct fsm_client_cap_node, node,
&fsm_client_cap_table)
        {
                if(node -> client_badge == client_badge)
                {
                        node->cap_table[node->cap_num] = cap;
                        node -> cap_num ++;
                        return node -> cap_num - 1;
                }
        }
        // 接后半部分
```

但是也有可能甚至我们的文件系统也没有进行记录。如果出现了这样的情况我们需要分配能力节点给该文件系统然后创造对应的文件系统下的能力组中的对应能力并将其返回。因此我们就有：

```
        // 续前半部分
        node = (struct fsm_client_cap_node *) malloc (sizeof(struct
fsm_client_cap_node)); // 首先创建挂载文件系统对应节点。
        if(!node) return -1;    // means no memory.
        node -> client_badge = client_badge;    // 给该节点填充badge表明对象。
        memset(node -> cap_table, 0, sizeof(node -> cap_table));        //
初始化
        node -> cap_table[0] = cap;         // 新的能力组的新的能力！
        node -> cap_num = 1;
        list_append(&node -> node, &fsm_client_cap_table); // 不要忘记讲节点
加入到虚拟文件系统的能力组表中！
        return 0;   // 第一个（0）节点。
```

Step 5: 设定我们的挂载信息

将挂载点，挂载路径，挂载路径**深度**进行设置，也就是通过我们的挂载点信息mpinfo来进行设定。



```c
/*
 * Mount Point Informations
 */
struct mount_point_info_node {
        cap_t fs_cap;
        char path[MAX_MOUNT_POINT_LEN + 1];
        int path_len;
        ipc_struct_t *_fs_ipc_struct;
        int refcnt;

        struct list_head node;
};

```

我们的mpinfo记录了：

- 文件系统在虚拟文件系统中的能力值
- 挂载路径
- 挂载深度
- ipc通信结构
- 引用次数
- 挂载点在挂载表中的节点索引

因此这里我们逐个填充字段即可。

```
                fsm_req -> mount_id = mount_id;
                fsm_req -> mount_path_len = mpinfo -> path_len;
                strcpy(fsm_req -> mount_path, mpinfo -> path);
```

还记得我们上一步中的返回了新的能力组吗？如果我们出现了新的能力组，我们还需要进行标记，需要让调用的主线程知道新注册了一个能力组并将其返回给我们的调用者。因此我们有：

```
fsm_req -> new_cap_flag = ret_with_cap;
```

Step 6: 解锁

完成了上述的操作后，我们需要解锁，以供其他新的线程访问。我们有：

```
                pthread_mutex_unlock(&fsm_client_cap_table_lock);
                pthread_rwlock_unlock(&mount_point_infos_rwlock);
```

这是我们的第二部分的内容。

Answer:

```
                /* Lab 5 TODO Begin (Part 1) */

                /* HINT: MountInfo is the info node that records each
mount
                 * point and actual path*/
                /* It also contains a ipc_client that delegates the actual
                 * filesystem. PARSE_PATH is the actual vfs layer that
does the
                 * path traversing */
                /* e.g. /mnt/ -> /dev/sda1 /mnt/1 -> /dev/sda2 */
                /* You should use the get_mount_point function to get
                 * mount_info. for example get_mount_info will return
                 * mount_info(/mnt/1/123) node that represents
/dev/sda2.*/
                /* You should use get_mount_info to get the mount_info and
set
                 * the fsm_req ipc_msg with mount_id*/

                /* lock the mount_info with rdlock */
                pthread_rwlock_rdlock(&mount_point_infos_rwlock);
                /* mpinfo = get_mount_point(..., ...) */
                // printf("[INFO_H] fsm_req -> path: %s, length: %ld\n",
fsm_req -> path, strlen(fsm_req -> path));
                mpinfo = get_mount_point(fsm_req -> path, strlen(fsm_req -
> path));
                // printf("[INFO_H] mpinfo -> path: %s, length: %ld\n",
mpinfo -> path, strlen(mpinfo -> path));
                /* lock the client_cap_table with mutex */
                pthread_mutex_lock(&fsm_client_cap_table_lock);
                /* mount_id = fsm_get_client_cap(...) */
                // printf("[INFO_H] get with cap: %d, conn_cap: %d\n",
mpinfo -> fs_cap, mpinfo -> _fs_ipc_struct -> conn_cap);
                mount_id = fsm_get_client_cap(client_badge, mpinfo ->
fs_cap);
                // printf("[INFO_H] get mount_id: %d\n", mount_id);
                /* if mount_id is not present, we first register the cap
set the
                 * cap and get mount_id */
                if(mount_id == -1)
                {
                        // printf("[INFO_H] set with cap: %d", mpinfo ->
fs_cap);
                        mount_id = fsm_set_client_cap(client_badge, mpinfo
-> fs_cap);
```

```
                        // printf("[INFO_H] set mount_id: %d\n",
mount_id);
                        ret_with_cap = true;    // 需要返回新的能力组。
                        ipc_set_msg_return_cap_num(ipc_msg, 1);
                        ipc_set_msg_cap(ipc_msg, 0, mpinfo->fs_cap);
                }
                /* set the mount_id, mount_path, mount_path_len in the
fsm_req
                 */
                fsm_req -> mount_id = mount_id;
                fsm_req -> mount_path_len = mpinfo -> path_len;
                strcpy(fsm_req -> mount_path, mpinfo -> path);
                /* Specifically if we register a new fs_cap in the
cap_table, we
                 * should let the caller know with a fsm_req->new_cap_flag
and
                 * then return fs_cap (noted above from mount_id) to the
                 * caller*/
                fsm_req -> new_cap_flag = ret_with_cap;
                /* Before returning to the caller , unlock the
client_cap_table
                 * and mount_info_table */
                pthread_mutex_unlock(&fsm_client_cap_table_lock);
                pthread_rwlock_unlock(&mount_point_infos_rwlock);
                // UNUSED(mpinfo);

                // UNUSED(mount_id);
                /* Lab 5 TODO End (Part 1) */
```

FS_BASE

有一说一这是整个实验中**最恶心最难受的部分了**。

Vnode

首先我们先来完成对vnode的管理，分配。在虚拟文件系统中，vnode储存着对应挂载文件系统上的信息，Chcore采用红黑树的方式进行管理。这样提供了统一的管理方式，使得挂载任意类型的文件系统成为可能。

```
struct fs_vnode {
        ino_t vnode_id; /* identifier */
        struct rb_node node; /* rbtree node */

        enum fs_vnode_type type; /* regular or directory */
        int refcnt; /* reference count */
        off_t size; /* file size or directory entry number */
        struct page_cache_entity_of_inode *page_cache;
        cap_t pmo_cap; /* fmap fault is handled by this */
        void *private;
        // private中将会储存特定信息。
```

```
        pthread_rwlock_t rwlock; /* vnode rwlock */
};
```

下面针对user/system-services/system-servers/fs_base/fs_vnode.c进行补充。

首先我们先注意一下红黑树的相关函数接口。位于user/chcore-libc/libchcore/porting/overrides/include/chcore/container/rbtree.h中。

分配vnode

接下来完成的是函数：

```
struct fs_vnode *alloc_fs_vnode(ino_t id, enum fs_vnode_type type, off_t
size, void *private);
```

首先，我们先进行节点的分配。我们有：

```
        struct fs_vnode *ret = (struct fs_vnode *)malloc(sizeof(*ret));
        if (ret == NULL)
                return NULL;    // 分配空间不足就返回空。
```

接下来根据结构填充对应字段。

```
        ret->vnode_id = id;
        ret->type = type;
        ret->size = size;
        ret->private = private;
```

由于这是新创立的节点，因此引用次数需要为1，并且要标记对应物理内存pmo_cap为-1，用于缺页异常处理（参考lab2）。

```
        ret->refcnt = 1;
        ret->pmo_cap = -1;
```

接着注意到我们的字段里有一个page_cache。这是为了配合读写缓存和脏页处理等。虽然本次试验并没有用到page_cache，但是我们还是为其加上页缓存处理：

```
            // 可以不写入代码中。
            if (using_page_cache)
                ret->page_cache = new_page_cache_entity_of_inode(ret-
>vnode_id, ret);
```

最后不要忘记我们还有一个锁字段。我们需要进行初始化：

```
        pthread_rwlock_init(&ret->rwlock, NULL);
        return ret;
```

从id获得vnode

接下来我们需要完成从id到实体节点的转换，也就是利用inode的id来查找到对应的vnode。这里需要使用到红黑树的查找功能。我们来看一下在红黑树中，进行查找主要是通过比较函数的方式进行，我们需要用到的函数接口为：

```
/*
 * rb_search - search for a node matching @key in rbtree
 *
 * @this: rbtree to be searched
 * @key: pointer to user-specific key. We use void* to implement
generality
 * @cmp: user-provided compare function, comparing @key and rb_node in
rbtree.
 * Users should implement their logic to extract key from rb_node pointer.
 *
 * return:
 *     pointer of targeted node if @key exists in rbtree.
 *     NULL if no matching node is found.
 */
struct rb_node *rb_search(struct rb_root *this, const void *key,
comp_key_func cmp);
```

因此我们需要传入红黑树的根节点，当前的vnode_id和一个比较函数。比较函数在`user/system-services/system-servers/fs_base/fs_vnode.c`中也有定义：

```
__attribute__((unused)) static int comp_vnode_key(const void *key, const
struct rb_node *node)
{
        struct fs_vnode *vnode = rb_entry(node, struct fs_vnode, node);
        ino_t vnode_id = *(ino_t *)key;

        if (vnode_id < vnode->vnode_id)
                return -1;
        else if (vnode_id > vnode->vnode_id)
                return 1;
        else
                return 0;
}
```

这样我们只需要完成如下的部分。需要注意的是我们不是返回红黑树节点，而是节点所反映的vnode节点。vnode节点中有定义红黑树节点成员，因此再使用一次entry函数即可返回我们的vnode节点。

```
struct fs_vnode *get_fs_vnode_by_id(ino_t vnode_id)
{
        /* Lab 5 TODO Begin (Part 2) */
        /* Use the rb_xxx api */
        struct rb_node *node =
                rb_search(fs_vnode_list, &vnode_id, comp_vnode_key);
        if (node == NULL)
                return NULL;
        return rb_entry(node, struct fs_vnode, node);
        /* Lab 5 TODO End (Part 2) */
}
```

增加/减少引用次数

在实验手册中已经提到，`private node`中记录了inode被引用的次数。因此我们进行增加引用次数时，只需要进行增加即可。 首先，我们将其强制转换成`fs_vnode`，然后将其成员refcnt减去一即可。需要注意的是这里是**减少inode**的引用次数，而一个vnode可能会指代多个不同的文件系统的inode（被引用），因此其自身的refcnt与其指向的inode的refcnt不是相同的。

```
/* refcnt for vnode */
int inc_ref_fs_vnode(void *private)
{
        /* Lab 5 TODO Begin (Part 2) */
        /* Private is a fs_vnode */
        // UNUSED(private);
        ((struct fs_vnode *)private)->refcnt++;
        return 0;
        /* Lab 5 TODO End (Part 2) */
}
```

接下来是减少引用次数，需要注意的是引用次数不能为负数（常识嘛），因此当引用次数变为0时需要将它从红黑树中删除，并且关闭该文件。这里我们可以调用前面写好的`pop_free_fs_vnode`函数，该函数实现了将其从红黑树删除，并且判断是否有必要释放物理内存资源。

```
int dec_ref_fs_vnode(void *private)
{
        /* Lab 5 TODO Begin (Part 2) */
        /* Private is a fs_vnode Decrement its refcnt */
        // UNUSED(private);
        int ret;
        struct fs_vnode * node = (struct fs_vnode *)private;
        node->refcnt--;
        if (node->refcnt == 0) {
                ret = server_ops.close(
```

```
                        node->private, (node->type == FS_NODE_DIR), true);
                if (ret)
                        return ret;

                pop_free_fs_vnode(node);
        }
        return 0;
        /* Lab 5 TODO End (Part 2) */
}
```

Server_entry

接下来我们需要完成的是user/system-services/system-servers/fs_base/fs_wrapper.c的部分。越到后面指导手册能够给予我们的有效信息就越少了，因此我们一步一步慢慢来了解为什么。 首先，我们了解到该文件其实就是处理有关vnode和inode的文件标识符映射问题。每个进程都拥有一个文件标识符fd，在这里储存在libc中。我们需要完成将每个进程的fd转换成对应的文件系统侧的fid，并且维护好每个索引的vnode对象。

对于多线程任务，我们都需要用锁来保证临界区的互斥访问。这里我们可以看到采用了以下两个锁，分别是自旋锁和读写锁。

```
pthread_spinlock_t server_entry_mapping_lock;
pthread_rwlock_t fs_wrapper_meta_rwlock;
```

全局变量记录了一个映射列表。

```
struct list_head server_entry_mapping;
```

在进行任务之前，我们首先阅读已经存在的下面的函数代码：

```
void fs_wrapper_clear_server_entry(badge_t client_badge, int fid)
{
        struct server_entry_node *private_iter;

        /* Check if client_badge already involved */
        pthread_spin_lock(&server_entry_mapping_lock);
        for_each_in_list (private_iter,
                          struct server_entry_node,
                          node,
                          &server_entry_mapping) {
                if (private_iter->client_badge == client_badge) {
                        for (int i = 0; i < MAX_SERVER_ENTRY_NUM; i++) {
                                if (private_iter->fd_to_fid[i] == fid) {
                                        private_iter->fd_to_fid[i] = -1;
                                }
                        }
```

```
                        pthread_spin_unlock(&server_entry_mapping_lock);
                        return;
                }
        }
        pthread_spin_unlock(&server_entry_mapping_lock);
}
```

这是用于清楚对应的fd->fid映射的一段代码。我们发现：

- 首先，我们声明了一个用于迭代的映射节点`private_iter`。
- 接着，我们的映射表位于临界区域，会有多个进程访问。因此我们需要加上自旋锁。
- 然后采用我们很熟悉的链表迭代循环来寻找到记录了对应的挂载点(client_badge)的映射节点。这里使用client_badge是因为VFS虚拟文件系统和FS文件系统是通过ipc来进行协同的。
- 将其设为-1表示该映射已经被清空。fd和fid文件描述符在POSIX系统内都应当为非负数有效。
- 最后解锁。

了解了这些点后我们就可以开始我们的操作了。

获取server_entry

首先我们还需要了解一下我们为什么需要获得这样的一个映射。我们的VFS是这样管理不同的FS的，为用户提供了一个统一的接口来管理和访问不同的文件系统。

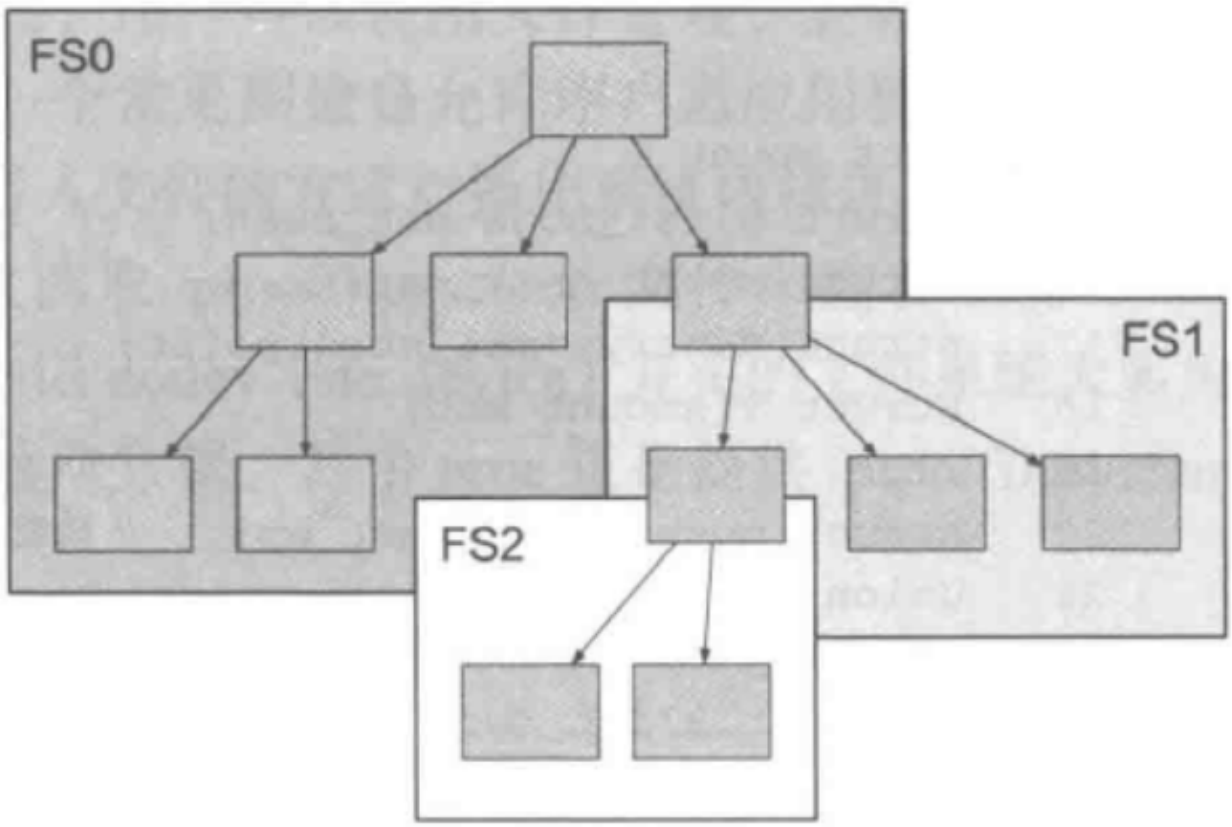


图 11.20　文件系统的挂载

这样我们在进行文件访问时，需要进行从虚拟文件系统的标识符fd到真实文件系统标识符fid的转换，然后再通过该信息来逐级访问，获取挂载点，最终得到真实磁盘文件。同样访问的方式，挂载点信息等转换信息可以通过缓存的方式储存来提高访问速度。此处不涉及缓存设计因此按下不表。

首先我们完成`fs_wrapper_get_server_entry`函数。

- 当我们需要获取的fd正好就是根fd时，也就是不需要转换的情况下，我们不需要进行转换。比如在访问根目录时。这里采用`AT_FDROOT`进行指代。

```
struct server_entry_node *n;
if (fd == AT_FDROOT)
                return AT_FDROOT;
```

- 除此之外，我们需要记得fd是一个**非负数**并且不能超过应该拥有的最大值。这里的最大值定义在`user/system-services/system-servers/fs_base/fs_wrapper_defs.h`中。（阅读头文件是一个好习惯），因此我们可以:

```
if (fd < 0 || fd >= MAX_SERVER_ENTRY_PER_CLIENT)
                return -1;
```

- 接下来根据我们清除映射的方式来访问我们的映射表，直到找到对应的client_badge对应的节点即可。不要忘记这是临界区域需要加上自旋锁，涉及到对全局记录表的访问。如果没有我们则返回-1。

```
        pthread_spin_lock(&server_entry_mapping_lock);
        for_each_in_list (n, struct server_entry_node, node,
&server_entry_mapping)
        {
                if (n->client_badge == client_badge)
                {
                        pthread_spin_unlock(&server_entry_mapping_lock);
                        return n->fd_to_fid[fd];
                }
        }
        pthread_spin_unlock(&server_entry_mapping_lock);
        return -1;
```

这样我们就完成了第一部分。

设置server_entry

当然也会出现映射不存在的情况。那么我们就需要进行添加映射的操作。添加映射后我们需要返回0表示正常添加，否则返回其他的错误代码（errno）。

首先我们先判断我们是否存在对应的文件系统(client badge).如果存在我们直接添加对应的映射。不要忘了自旋锁的存在。

```
        struct server_entry_node *private_iter;
        int ret = 0;

        // 检查我们的fd是否符合标准。
        if(fd < 0 || fd >= MAX_SERVER_ENTRY_PER_CLIENT)
                return -EFAULT;

        // 寻找。
        pthread_spin_lock(&server_entry_mapping_lock);
        for_each_in_list (private_iter,
                          struct server_entry_node,
                          node,
                          &server_entry_mapping) {
                if (private_iter->client_badge == client_badge) {
                        private_iter->fd_to_fid[fd] = fid;
                        pthread_spin_unlock(&server_entry_mapping_lock);
                        return ret;
                }
        }
```

如果不存在，我们还需要分配新的节点加入到节点表中，然后再添加分配。这里需要用到list_add或list_append操作。我们这里采用append。

```
        // 新的节点。
        struct server_entry_node *n = (struct server_entry_node
*)malloc(sizeof(*n));
        n->client_badge = client_badge;

        // 初始化所有映射。
        for (int i = 0; i < MAX_SERVER_ENTRY_PER_CLIENT; i++)
                n->fd_to_fid[i] = -1;

        // 添加映射。
        n->fd_to_fid[fd] = fid;

        // 加入表节点。
        list_append(&n->node, &server_entry_mapping);

        // 不忘初心。
        pthread_spin_unlock(&server_entry_mapping_lock);
        return ret;
```

这样这两个函数我们就完成了。也就完成了server_entry的所有任务。

fs_wrapper_ops

苦命活部分QAQ 注意这里的手册部分还没有更新。我们在这里需要完成的应该是：

> 练习6 实现 `user/system-services/system-servers/fs_base/fs_wrapper_ops.c` 中的 `fs_wrapper_open`、`fs_wrapper_close`、`__fs_wrapper_read_core`、`__fs_wrapper_write_core`、`fs_wrapper_lseek`函数。

开发者应该已经修复啦www

测试点设计的好像也有点问题，但是是完全可以覆盖我们的本次任务的测试的，所以我也不管啦

fs_wrapper_open

为了完成open，你需要先完全理解以下的内容。 open(2) close(2) fstatat(3p)

你也可以在你的类posix系统下输入以下指令来查看(前提是你安装了manual)。

```
man 2 open
man 2 close
man fstatat
```

首先我们根据注释一步步进行。我们首先需要检查我们的flags是否是正确的。对照测试文件`user/tests/fs_tests/fs_tools/tst_open.c`我们先来看一下我们需要注意传进来的哪些flag（有点面向测试编程了，这是一个不好的习惯，但是没办法了，我是骗分大王hhh）

我们可以发现对于我们的open操作主要涉及到以下的几个打开方式，对于文件权限设计我们主要设计以下的权限设计：

```
# 文件打开方式
O_CREAT, O_RDWR, O_APPEND
# 文件权限设置
S_IRUSR, S_IWUSR, S_IXUSR
```

- 我们先获取fr中给我们提供的信息。

```
        int new_fd = fr -> open.new_fd;
        char * path = fr -> open.pathname;
        mode_t mode = fr -> open.mode;
        // int fid = fr -> open.fid;    // 未使用变量会遭到严格的检查，因此我们不获取fid信息。
        int flags = fr -> open.flags;
```

接着我们进行简单的检查，主要是检查三个错误码：-EEXIST, -EISDIR, -ENOTDIR。对于是否存在，我们需要用到检测符号连接的函数`fstatat`，因此我们有：

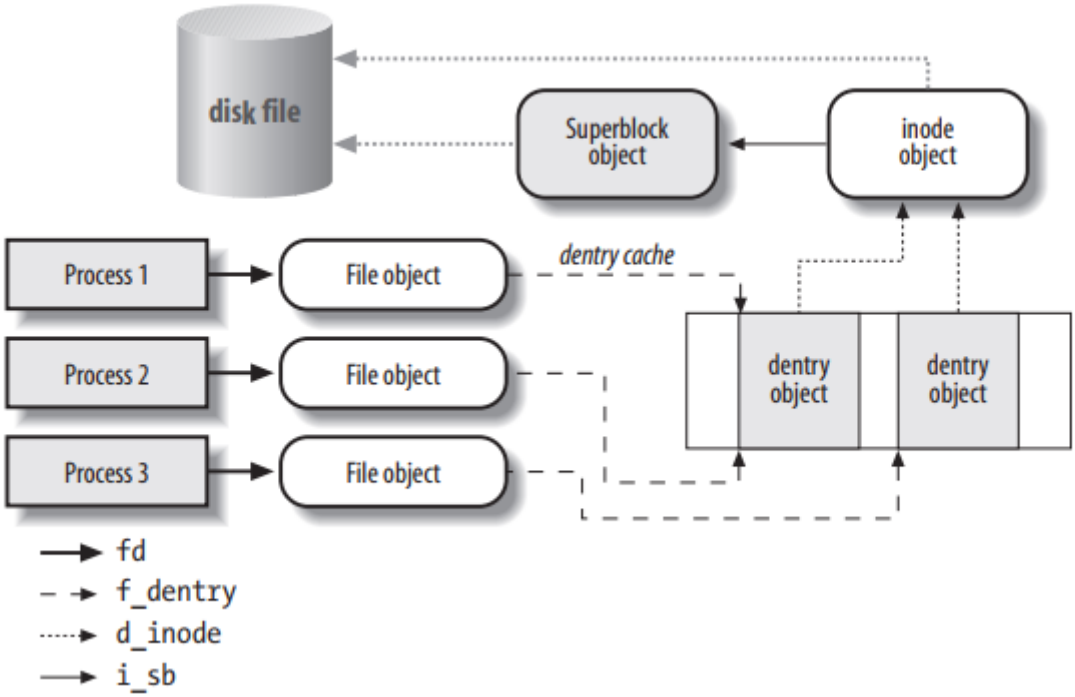
```
        // 检查是否创造存在的文件。通过fstatat获取。其实还有很多检查要做。但是测试文件里没有进行测试，不管啦!
        if((flags & O_CREAT) && (flags & O_EXCL))
```

```
        {
                struct stat status;
                // 检测文件是否是存在。如果存在，返回符号连接信息。否则返回错误信息提
示文件不存在。
                if(server_ops.fstatat(path, &status, AT_SYMLINK_NOFOLLOW)
== 0)
                        return -EEXIST;
        }
        // 检查文件类型。
        if((flags & (O_WRONLY | O_RDWR)) && S_ISDIR(mode))
                return -EISDIR;
        if((flags & O_DIRECTORY && !S_ISDIR(mode)))
        {
                return -ENOTDIR;
        }
```

- 接下来就可以打开我们的文件啦。我们需要进行打开操作，首先注意到server_ops.open的具体实现在user/system-services/system-servers/tmpfs/tmpfs.c中。我们可以发现在该函数中对传入的后四个参数进行了填充。因此我们可以先声明这几个参数，再将这几个参数的地址传入进行填充。（C语言是不存在引用的，因此这是取地址而非引用）。我们有：

```
        // 实现中可以看出需要填写四个字段，因此我们需要创造出他们，最后填写。
        ino_t vnode_id;
        off_t vnode_size;
        int vnode_type;
        void *private;
        int ret = server_ops.open(
                path,
                flags,
                mode,
                &vnode_id,
                &vnode_size,
                &vnode_type,
                &private
        );
        if(ret != 0)
                return -EINVAL;
```

函数其实在底部调用了openat函数，主要是通过文件对象来访问目录项，随后获得目录项对象。我们可以参考这个图：



- 接着根据返回的vnode_id来查找是否存在vnode节点。如果存在则增加该vnode节点的引用数，同时关闭vnode节点（但是不关闭真实文件），如果没有我们就创建一个新的vnode节点，将其加入到我们的vnode红黑树中。为我们后面建立映射做好准备。

```
        /* Check if the vnode_id is in rb tree.*/
        struct fs_vnode * vnode = get_fs_vnode_by_id(vnode_id);
        /* If not, create a new vnode and insert it into the tree. */
        if(vnode == NULL)
        {
                vnode = alloc_fs_vnode(vnode_id, vnode_type, vnode_size,
private);
                push_fs_vnode(vnode);
        }
        /* If yes, then close the newly opened vnode and increment the
refcnt of
         * present vnode */
        else
        {
                inc_ref_fs_vnode(vnode);
                server_ops.close(private, (vnode_type == FS_NODE_DIR),
false);
        }
```

- 最后我们将建立起真正的映射，然后返回我们客户端的fd，也就是我们新创建的fd。我们需要调用我们完成过的函数。

```
        // 创造新节点后，需要将虚拟系统fd映射到文件系统的fid上。fid需要进行分配。
        int entry_index = alloc_entry();        // 分配对应的需要插入新节点的表
项下标;
        fr->open.fid = entry_index;
```

```
        off_t offset = 0;
        // offset 文件游标设定:
        if((flags & O_APPEND) && S_ISREG(mode))
                offset = vnode_size;
        // 将下标指派给vnode并设定。这里需要创造新的字符串防止引用相同的字符串对象。
        assign_entry(server_entrys[entry_index], flags, offset, 1, (void
*)strdup(path), vnode);
        // 设定server_entry, 建立映射。
        fs_wrapper_set_server_entry(client_badge, new_fd, entry_index);
        /* Return the client fd */
        return new_fd;
```

这样我们就完成了我们的open函数。

fs_wrapper_close

接下来我们来完成close函数。我们可以按照注释来一步步进行，但是需要小心注释又细微地表达错误。

- 获取当前fd指向的server_entry。这里不要使用client_badge来获取fid，我们需要修改的是表项的对应引用次数以及server表项而非vnode。

```
struct server_entry * entry = server_entrys[fr->close.fd];
```

- 接着将表项的引用递减。当表项引用为0时，vnode就不需要了，因此清楚表项内容并且删除。

```
        /* If refcnt is 0, free the server_entry and decrement the vnode
         * refcnt*/
        if(entry -> refcnt == 0)
        {
                dec_ref_fs_vnode(entry -> vnode);
                fs_wrapper_clear_server_entry(client_badge, fr->close.fd);
                free_entry(fr->close.fd);
        }
        return 0;
```

这样就完成了close部分。

读写核心部分

这两个部分大同小异，都是首先判断是否是符合权限的情况，再进行调用读取/写入即可。请注意不要在函数内更新文件偏移，外部已经更新好了。

```
static int __fs_wrapper_read_core(struct server_entry *server_entry, void
*buf, size_t size, off_t offset)
{
        /* Lab 5 TODO Begin (Part 4)*/
```

```
        /* Use server_ops to read the file into buf. */
        /* Do check the boundary of the file and file permission correctly
Check
         * Posix Standard for further references. */
        // 判断文件是否可读。
        if(server_entry -> flags & O_WRONLY)
                return -EBADF;
        struct fs_vnode * vnode = server_entry -> vnode;
        // 读。
        ssize_t off = server_ops.read(vnode -> private, offset, size,
buf);
        /* You also should update the offset of the server_entry offset */
        // 返回读后的偏移。
        return off;
        /* Lab 5 TODO End (Part 4)*/
}
static int __fs_wrapper_write_core(struct server_entry *server_entry, void
*buf, size_t size, off_t offset)
{
        /* Lab 5 TODO Begin (Part 4)*/
        /* Use server_ops to write the file from buf. */
        /* Do check the boundary of the file and file permission correctly
Check
         * Posix Standard for further references. */
        // 判断文件是否可写。
        if((server_entry -> flags) & O_RDONLY)
                return -EBADF;
        struct fs_vnode * vnode = server_entry -> vnode;
        // 写。
        ssize_t off = server_ops.write(vnode -> private, offset, size,
buf);
        /* You also should update the offset of the server_entry offset */
        // 返回写后偏移。
        return off;
        /* Lab 5 TODO End (Part 4)*/
}
```

这样我们就完成了读写核心部分。

lseek

首先我们需要参考标准手册。你也可以在你的类posix系统中输入`man lseek`来查看。lseek(2)

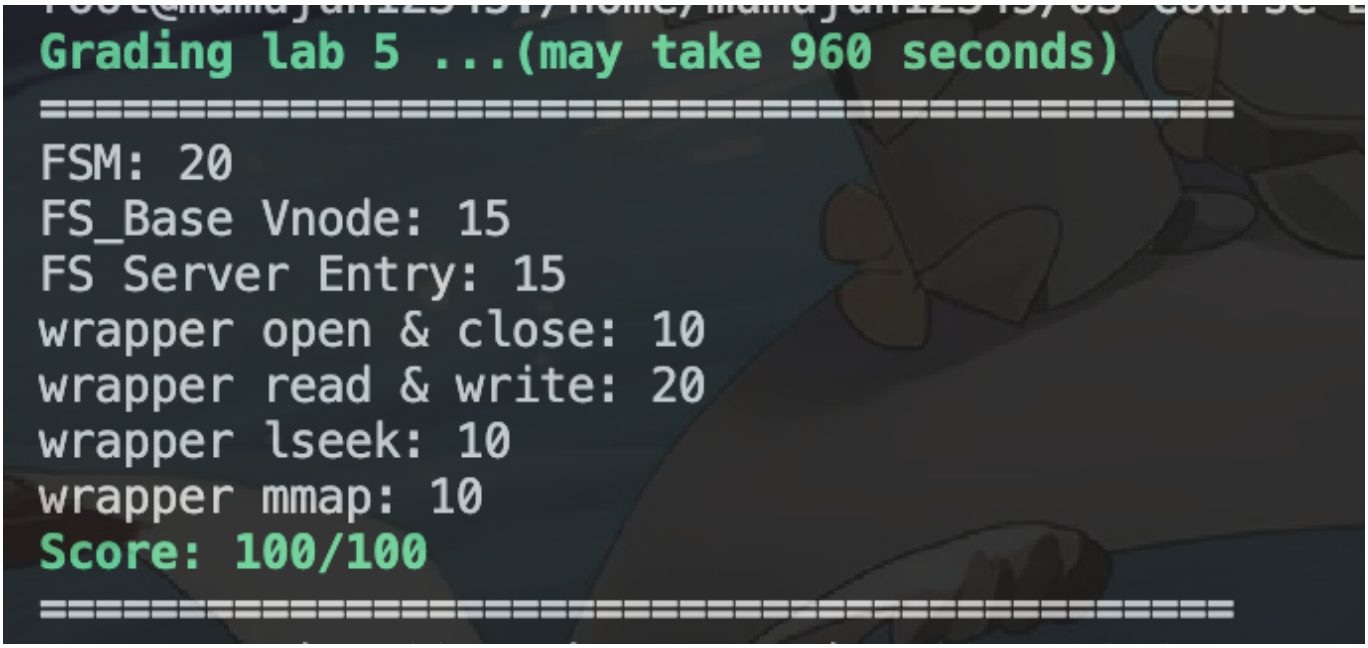
lseek是用来调整文件游标的。lseek重新调整已经打开的文件标识符fd关联的文件的游标offset，通过whence来选择不同的调整方式：

- SEEK_SET: 将游标设定到指定下标。
- SEEK_CUR: 将游标设定到当前偏移+指定偏移。
- SEEK_END: 将游标设定到当前文件末尾+指定偏移。

当出现不合法操作：比如将游标设定成了负值，或者whence不是我们需要的whence时，将其返回错误代码-EINVAL。

这样我们就有：

```
int fs_wrapper_lseek(ipc_msg_t *ipc_msg, struct fs_request *fr)
{
        /* Lab 5 TODO Begin (Part 4)*/
        /* Check the posix standard. Adjust the server_entry content.*/
        off_t offset = fr -> lseek.offset;
        int fd = fr -> lseek.fd;
        int whence = fr -> lseek.whence;
        // 根据whence选定调整方式。判断是否合法。
        switch (whence)
        {
                case SEEK_SET:
                {
                        if(offset < 0) return -EINVAL;
                        server_entrys[fd] -> offset = offset;
                        break;
                }
                case SEEK_CUR:
                {
                        if(server_entrys[fd] -> offset + offset < 0)
                                return -EINVAL;
                        server_entrys[fd] -> offset += offset;
                        break;
                }
                case SEEK_END:
                {
                        if(server_entrys[fd] -> vnode -> size + offset <
0)
                                return -EINVAL;
                        server_entrys[fd] -> offset = server_entrys[fd] ->
vnode -> size + offset;
                        break;
                }
                default:
                        return -EINVAL;
        }
        // 不要忘记设定fr的返回值。
        fr -> lseek.ret = server_entrys[fd] -> offset;
        return 0;
        /* Lab 5 TODO End (Part 4)*/
}
```


Grading lab 5 ...(may take 960 seconds)
==========
FSM: 20
FS_Base Vnode: 15
FS Server Entry: 15
wrapper open & close: 10
wrapper read & write: 20
wrapper lseek: 10
wrapper mmap: 10
Score: 100/100
==========