OSLAB 2 实验报告

161220049 黄奕诚

一、实验整体描述

按时完成要求的实现及相应测试，共用 15 小时左右。

目前文件系统有如下功能特性——

- 有三大类的文件系统：procfs、devfs 和 kvfs，并且有讲义中提到的/proc/[pid]显示线程的基本信息（pid、runnable 及寄存器现场，且每次 schedule 时会更新）、/proc/cpuinfo 和/proc/meminfo（因缺少相应接口、故只存有一行字符串）；同时有/dev/null 用于丢弃不必要的输入流、/dev/zero 提供 null 串、/dev/random 提供随机数（这里用 srand 与 rand 实现）；此外还有 key-value 文件系统，用户可以在该文件系统的文件内进行读写。
- 支持多个文件系统挂载到同一路径上，用链表采取栈的数据结构维护了一个路径上的所有文件系统，如此可以灵活地 mount 与 unmount 而不会丢失先前的文件系统。
- 支持解析出文件系统后，按对应文件系统分批处理 open 与 access。而后再按照不同的 inode 执行操作，一定程度上体现了面向对象编程的思想。

该文件系统有如下归约——

- 不允许多个进程或线程同时写一个文件，即若一个 inode 已经以可写的方式打开为某个 file，则其它进程或线程不可再写该文件。
- /proc 下的文件由线程的状态所决定，所以不可写。我是用特殊的方式在线程调度时更新线程文件内容的（暴力写）。/dev 和/zero 下的所有文件默认为既可读又可写。
- 目前只支持以上提到的三种文件系统，不支持非以上三种文件系统的挂载。
- 目前只支持用户以 O_RDONLY, O_WRONLY 和 O_RDWR 三种 flags 打开文件。
- 目前每个文件的内容仅是内存里的字符串，不支持磁盘读写。

二、设计思路

此次实验工程量比 oslab1 大一些，但是难度不大，多数是体力活。我的总体实现思路与讲义后面提供的思路大致相同，只是改动了几个内部函数的原型。

我以文件系统初始化及操作、文件系统挂载、文件初始化及读写操作、设备（特殊文件）读写、线程文件管理的顺序来阐述此文件系统的设计思路。为保证思路阐述的清晰性，我省去函数的具体定义，将相关功能的函数原型或引用附图辅助说明。

1.文件系统初始化及操作

在操作系统初始化阶段的 vfs_init()中，有如图所示的各阶段初始化工作。

```
static void vfs_init(){
    spinlock_init();
    path_init();
    rand_init();
    pool_init();
    oop_func_init();
    fs_init();
}
```

spinlock_init 是自旋锁的初始化，因为目前文件读取是在内存中进行的，不通过磁盘读取，所以自旋锁满足基本的并发实现条件。path_init 初始化三种文件系统的挂载路径，rand_init 提供 srand 函数以供/dev/random 设备使用，srand 的参数为 uptime()，调用了 amdev 的 API。pool_init 初始化了 fd 池和 file 池，它们维护了当前可用的 fd 或 file。oop_func_init 初始化各种文件系统的操作函数，继承于一个已经定义的操作函数类。最后，fs_init 在 nanos 中生成三种文件系统 kvfs、devfs 和 procfs，并初始化它们的一些文件，比如/dev/null。

对于文件系统的操作，我封装成如下三个操作，其中前者对文件系统的各个成员进行初始

```
/* fs's operations*/
static void fsops_init(struct filesystem *fs, const char *name);
static inode_t *fsops_lookup(struct filesystem *fs, const char *path);
static int fsops_close(inode_t *inode);
```

化，而文件系统的结构体定义如下所示，它维护了文件系统的名称、操作、类型、下属文件及同挂载路径的下一个文件系统。lookup 是在 open 或 access 执行后，解析 path 为具体的某个下属 inode，然后返回，若不存在则返回空指针。

```
struct filesystem{
    char name[MAX_FS_NAME_LEN];
    fsops_t *ops;
    int fs_type;
    inode_t *inodes[MAX_INODE_NUM];
    filesystem_t *next_fs_under_same_path;
};
```

```
struct mount_path{
    char name[MAX_PATH_LEN];
    filesystem_t *fs;
};
```

2.文件系统挂载

如上右图所示，我维护了每种文件系统的挂载路径，将相同类型的文件系统挂载到同一个路径上，每个路径维护了它的名称（如“/proc“）以及当前所指向的文件系统结点。当新挂载（mount）一个文件系统时，则将其插入到链表的头部，并将对应 mount_path 的 fs 设置为它。当 unmount 一个文件系统时，则将 mount_path 的 fs 指向其下一个文件系统，并将原 fs 删去即可。相当于维护了一个栈。因代码较为简短，贴出如下所示。

```
static int vfs_mount(const char *path, filesystem_t *fs){
    kmt->spin_lock(&fs_lock);
    if (strcmp(path, procfs_path.name) == 0 && fs->fs_type == PROCFS)
        mount_new_fs(&procfs_path, fs);
    else if (strcmp(path, devfs_path.name) == 0 && fs->fs_type == DEVFS)
        mount_new_fs(&devfs_path, fs);
    else if (strcmp(path, kvfs_path.name) == 0 && fs->fs_type == KVFS)
        mount_new_fs(&kvfs_path, fs);
    kmt->spin_unlock(&fs_lock);
    return 0;
}
```

其中 mount_new_fs 是链表的插入操作。vfs_unmount 同理，先确定挂载路径的类型，并进行链表的删除操作。

3.文件初始化及读写操作

我对每种文件系统都初始化了一些必要的文件，如 devfs 的 null、zero 和 random，用于满足实验要求；kvfs 的 a.txt 用于文件读写测试；而 procfs 的各个文件则在 kmt.c 中进行创建、修改及删除（分别对应 thread_create、schedule 和 thread_teardown）。我将基本的文件创建封装为一个接口——

```
static void create_inodes(filesystem_t *fs, char can_read, char can_write, char *inode_name, char *content, mount_path_t *path);
```

在 fs_init 中我便能干净利落地初始化如下文件——

```
static void create_procinodes(filesystem_t *fs) {
    TRACE_ENTRY;
    create_inodes(fs, 1, 0, "/cpuinfo", "This is the cpuinfo file.\n", &procfs_path);
    create_inodes(fs, 1, 0, "/meminfo", "This is the meminfo file.\n", &procfs_path);
    TRACE_EXIT;
}

static void create_kvinodes(filesystem_t *fs){
    create_inodes(fs, 1, 1, "a.txt", NULL, &kvfs_path);
}

static void create_devinodes(filesystem_t *fs){
    create_inodes(fs, 1, 1, "/null", NULL, &devfs_path);
    create_inodes(fs, 1, 1, "/zero", NULL, &devfs_path);
    create_inodes(fs, 1, 1, "/random", NULL, &devfs_path);
}
```

对于文件读写（read、write 和 lseek），我先根据参数的 fd 对整个 file_pool 进行遍历搜索，寻找到一个 fd 与其相同的 file。又因为我维护了 file 的对应实际文件 inode，于是满足了 fileops_read 的参数要求，剩下的工作交给它来做。

```
static ssize_t vfs_read(int fd, void *buf, size_t nbyte){
    kmt->spin_lock(&fs_lock);
    int file_index = search_for_file_index(fd);
    if (file_index == -1){
        kmt->spin_unlock(&fs_lock);
        return -1;
    }
    ssize_t ret = file_pool[file_index]->ops->read(file_pool[file_index]->f_inode, file_pool[file_index], buf, nbyte);
    kmt->spin_unlock(&fs_lock);
    return ret;
}


static ssize_t vfs_write(int fd, void *buf, size_t nbyte){
    kmt->spin_lock(&fs_lock);
    int file_index = search_for_file_index(fd);
    if (file_index == -1){
        kmt->spin_unlock(&fs_lock);
        return -1;
    }
    ssize_t ret = file_pool[file_index]->ops->write(file_pool[file_index]->f_inode, file_pool[file_index], buf, nbyte);
    kmt->spin_unlock(&fs_lock);
    return ret;
}
```

对于一个文件的 read 操作，思路很常规，维护好它的 offset，保证读取的长度以及 offset 不超过文件的大小，对特殊文件（设备）进行特判。如此即可。

而写文件的思路大致相同，考虑到我的每个 inode 以及 file 维护了其文件内容，而它实则是一个静态字符串（字符数组），有最大的容量，只要保证写进去的字符串长度不超过容量即可。目前我只实现了覆盖写，不过加个 append 其实也很容易（只是实验没要求，逃）

lseek 就更常规了，没啥好讲的 orz。

4.设备（特殊文件）读写

/dev 文件系统下我维护了三个设备：/dev/null、/dev/zero 和/dev/random，其中/dev/null 被读时直接返回 EOF（-1），/dev/zero 被读时直接把 buf 的要求大小的字节填满 0（null），/dev/random 被读时，每个 buf 的字节都对应了一个随机数（我用 rand、srand）实现的伪随机。这三个文件都是可写的，只是写进入的 buf 被直接丢掉了。我的 srand 包含在了 vfs_init()里面，调用了 amdev 的时间接口，并封装成 uptime()。这样每次 make run 之和生成的随机数不会全一样。

三、测试用例阐释

我按照不同的模块封装了多种测试，分为 procfs_test()、devfs_test()、kvfs_test()、mount_test()、multiopen_test()、multithread_test()。下面分别进行阐述。

1.PROCFS 文件系统测试

```
static void procfs_test(){
  TestLog("procfs_test begins...");
  int i;
  for (i = 0; i < 5; i++){
    kmt->create(&fs_test_thread[i], fs_test_func, (void *)i);
    TestLog("Thread %d created.", i);
  }
  kmt->teardown(&fs_test_thread[2]);
  TestLog("Thread 2 deleted.");
  print_proc_inodes();
  int fd = vfs->open("/proc/1/status", O_RDONLY);
  if (fd == -1){
    panic("open failed.\n");
  }
  assert(vfs->access("/proc/1/status", W_OK) == 0);
  assert(vfs->access("/proc/1/status", R_OK) == 0);
  char buf[200];
  if (vfs->read(fd, buf, sizeof(buf) - 1) == -1)
    panic("read failed");
  printf("buf------\n%s\n", buf);
  vfs->close(fd);
  assert(vfs->access("/proc/1/status", F_OK) == 0);
  assert(vfs->access("/proc/1/status", R_OK) == 0);
  assert(vfs->access("/proc/1/status", W_OK) == 0);
  assert(vfs->access("/proc/1/status", X_OK) == -1);
  TestLog("procfs_test passed.");
}
```

首先我创建了五个线程，然后删去了第三个线程，通过 printf_proc_inodes 打印此时 procfs 里的文件信息，进行检查。随后我以只读方式打开某个线程所对应的文件，并测试返回值。对该线程文件的内容进行读取，肉眼检查信息是否与理想的一致。因为一个线程的信息包含多种，用 assert 不是很方便。故采取更便捷的打印方法。而 access 的检验可以用 assert。如此可以检测 open、read、close 和 access 的实现。

2.DEVFS 文件系统测试

```
static void devfs_test(){
  TestLog("devfs_test begins...");
  int test = 5, ret;
  char buf[10];
        while(test--){
    ret = random_number(1, 10);
    assert(ret >= 1 && ret <= 10);
    printf("%d ", ret);
  }
  printf("\n");
  int fd = vfs->open("/dev/null", O_RDONLY);
  assert(vfs->read(fd, buf, sizeof(buf)) == -1);
  vfs->close(fd);
  fd = vfs->open("/dev/zero", O_RDONLY);
  vfs->read(fd, buf, sizeof(buf));
  vfs->close(fd);
  for (int i = 0; i < 10; i++)
    assert(buf[i] == 0);
  TestLog("devfs_test passed.");
}
```

为了测试三个设备的 read 能否与理想的达到一致。我封装了一个 random_number 函数，它能够通过 open、read 我的/dev/random 得到一个在每字节都包含随机数的字符串，然后处理为 min 到 max 之间的一个随机数。对随机数的检测只能靠肉眼，但观察发现确实呈现逼真的随机性。随后检测/dev/null 是不是真的直接返回 EOF，以及 /dev/zero 是不是真的返回 null 串。然后发现真的是这样的。

3.KVFS 文件系统测试

其实这个测试跟 procfs 的很像，只是因为我给了 kvfs 下的文件写权，所以此处我也测试了 write 的情况，考虑到 procfs 没测 lseek，这里也测了一下。由于写入与读出的内容可以由自己把控，所以我 assert 也包含了读出的内容，以及写入后再读出的内容。代码与 procfs 测试很像，不再贴出。

4.挂载与卸载测试

```
static void mount_test(){
  TestLog("mount_test begins...");
  int fd;
  filesystem_t *fs = (filesystem_t *)pmm->alloc(sizeof(filesystem_t));
  if (!fs) panic("fs allocation failed");
  fs->fs_type = KVFS;
  fs->ops = &kvfs_ops;
  vfs->mount("/", fs);
  assert((fd = vfs->open("/a.txt", O_RDONLY)) == -1);
  vfs->close(fd);
  fs = (filesystem_t *)pmm->alloc(sizeof(filesystem_t));
  if (!fs) panic("fs allocation failed");
  fs->fs_type = KVFS;
  fs->ops = &kvfs_ops;
  vfs->mount("/", fs);
  assert((fd = vfs->open("/a.txt", O_RDONLY)) == -1);
  vfs->close(fd);
  vfs->unmount("/");
  assert((fd = vfs->open("/a.txt", O_RDONLY)) == -1);
  vfs->close(fd);
  vfs->unmount("/");
  assert((fd = vfs->open("/a.txt", O_RDONLY)) != -1);
  vfs->close(fd);
  TestLog("unmount_test passed.");
}
```

为了测试挂载路径文件系统链表实现得怎么样。我简单地初始化了两个 kvfs 类型的文件系统（它们不包含/a.txt）。因为此时 nanos 里已经有一个在 vfs_init()阶段初始化的 kvfs，我依次将它们挂到同一路径上，并测试此时"a.txt"能不能打开。果然，只有当它们两者都被从路径上卸载时，路径才能对应最初的 kvfs，此时"a.txt"能够被打开。

5.单文件被多次打开测试

```
static void multiopen_test(){
  TestLog("multiopen_test begins...");
  int fd1, fd2, fd3;
  fd1 = vfs->open("/a.txt", O_RDONLY);
  assert(fd1 != -1);
  fd2 = vfs->open("/a.txt", O_RDWR);
  assert(fd2 != -1);
  fd3 = vfs->open("/a.txt", O_WRONLY);
  assert(fd3 == -1);
  assert(vfs->close(fd1) == 0);
  assert(vfs->close(fd2) == 0);
  assert(vfs->close(fd3) == -1);
  TestLog("multiopen_test passed.");
}
```

这里主要测试一个文件被多次打开时，打开的参数是否满足文件的读写权限。以及当文件被关闭时，相应的进行检查。我规定了一个文件在以可写的 flags 打开后，其它线程或进程不可再写它。而读则不受限制。

6.多线程同时打开一个文件读取测试

```
static thread_t fs_test_thread2[2];

static void fs_test_func2(void *arg){
    int fd = vfs->open("/a.txt", O_RDONLY);
    char buf[100];
    vfs->read(fd, buf, sizeof(buf));
    vfs->close(fd);
    printf("buf: %s\n", buf);
    while (1);
}

static void multithread_test(){
  TestLog("multithread_test begins...");
  kmt->create(&fs_test_thread2[0], fs_test_func2, NULL);
  kmt->create(&fs_test_thread2[1], fs_test_func2, NULL);
  TestLog("multithread_test ends.");
}
```

在这里我创建了两个线程，读同一个文件，然后观察是否能够成功创建两个不同的 file，并分别维护 open_offset，进行相应的读操作。与此同时检测自旋锁的可靠性。

四、TODO

- 添加更多的 flags，以更多方式打开文件
- 将一个窗口移植到 nanos，可以在窗口中显示信息，做成一个真正的 CUI
- 实现分页机制、更好的管理内存
- ……

五、WARNING

我写完 oslab2 才发现一件严重的事情——之前在写完 oslab1 之后，我把代码 push 到 github 去，按习惯我把 Makefile 里面有关于 git 的内容删除，这样发布到网上比较干净，其它人 make 的时候不会给 tracer 增加额外的记录。 然而，当我开写 oslab2 的时候，从 github 上把 oslab1 的代码 pull 了下来继续写，发生了 Makefile 的 merge 冲突，当时我脑子不清楚，没有想到 tracer 方面，就采用了删去 tracer 的那个 Makefile 版本。。。。于是就在刚刚我把 oslab2 push 到 github 时，惊讶 commit 次数怎么那么少，然后意识到了这个问题。。而后补完了 git，不过肯定损失了许多代码修改记录。

这一点已经向蒋老师解释过了。他承诺不会因此产生分数的损失。

六、个人感想

操作系统真的很有意思！以后有时间我想好好地完善这一操作系统。

对于本学期的 minilab 和 oslab，我个人认为每个实验设计的质量都是比较高的，难度把握得适中，不会让人感觉无所适从，但都需要花点时间去读手册、debug。虽然实验数量貌似多了些，但对我个人的学习压力并没增大多少（毕竟不是拖到 ddl 的那种）。感谢蒋老师及两位助教设计的实验，我觉得真的学到了不少东西。

以及感觉自己有必要去读一读 linux 代码，来体会自己实现的操作系统是多么 naïve 了。