LAB8：文件系统

PB18051183 白欣雨

实验目的

- 了解基本的文件系统系统调用的实现方法；
- 了解一个基于索引节点组织方式的Simple FS文件系统的设计与实现；
- 了解文件系统抽象层-VFS的设计与实现。

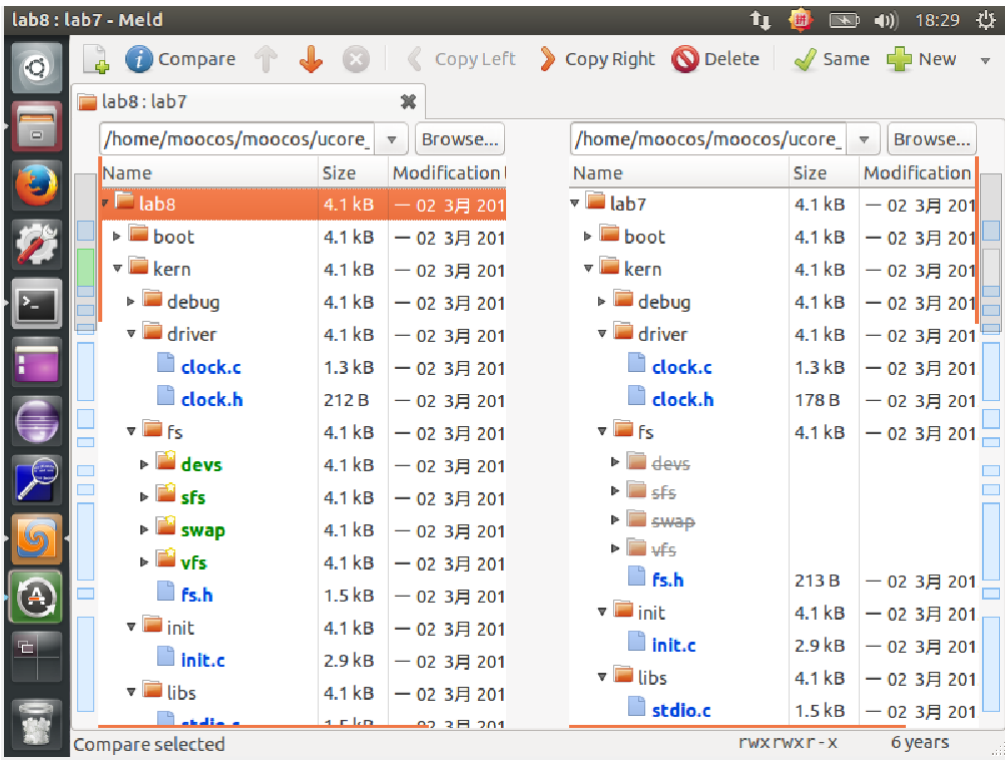
实验内容

实验七完成了在内核中的同步互斥实验。本次实验涉及的是文件系统，通过分析了解ucore文件系统的总体架构设计，完善读写文件操作，从新实现基于文件系统的执行程序机制（即改写do_execve），从而可以完成执行存储在磁盘上的文件和实现文件读写等功能。

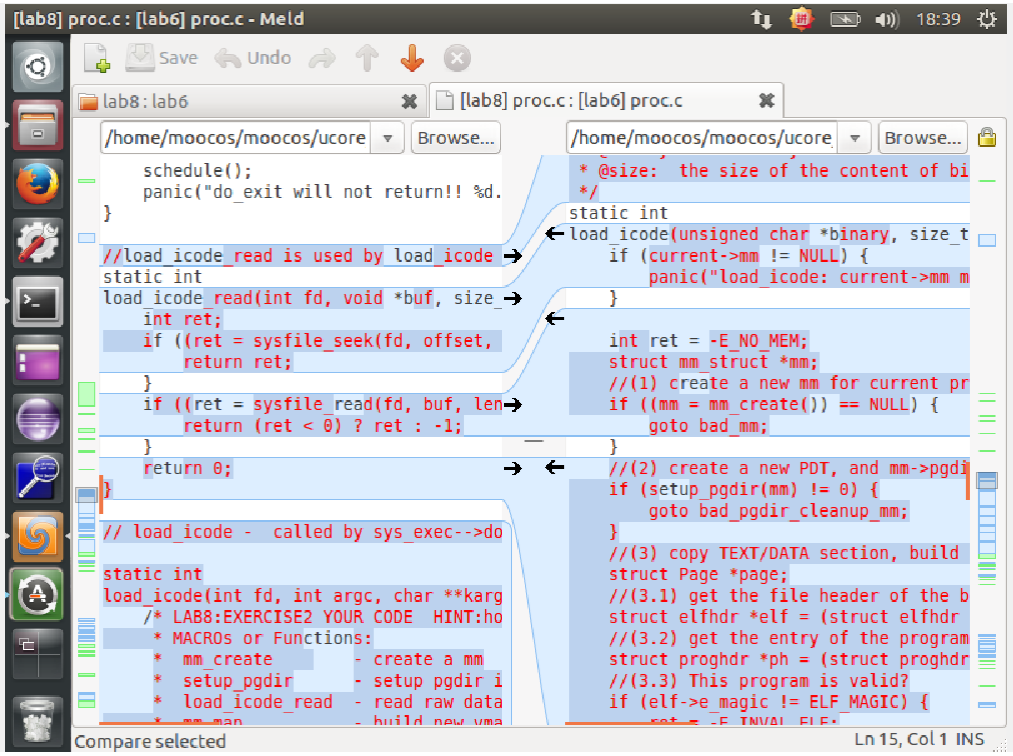
实验过程

练习0：填写已有实验

此练习需要把LAB6中的LAB1-LAB7缺省处补齐，本练习继续使用meld图形化方式进行合并，选择两个目录进行比较：



然后根据之前实验的内容，需要修改的文件有：proc.c 、default_pmm.c 、pmm.c 、swap_fifo.c 、vmm.c、trap.c、sched.c。以其中一个文件为例进行合并：



如图所示，点击箭头即可将需要修改补充的代码部分添加到相应部分，对其余文件作同样处理。除此以外，根据实验要求，我们还需要对部分的代码进行改进，但是经过分析并未发现要改进之处。

练习1：完成读文件操作的实现

算法分析

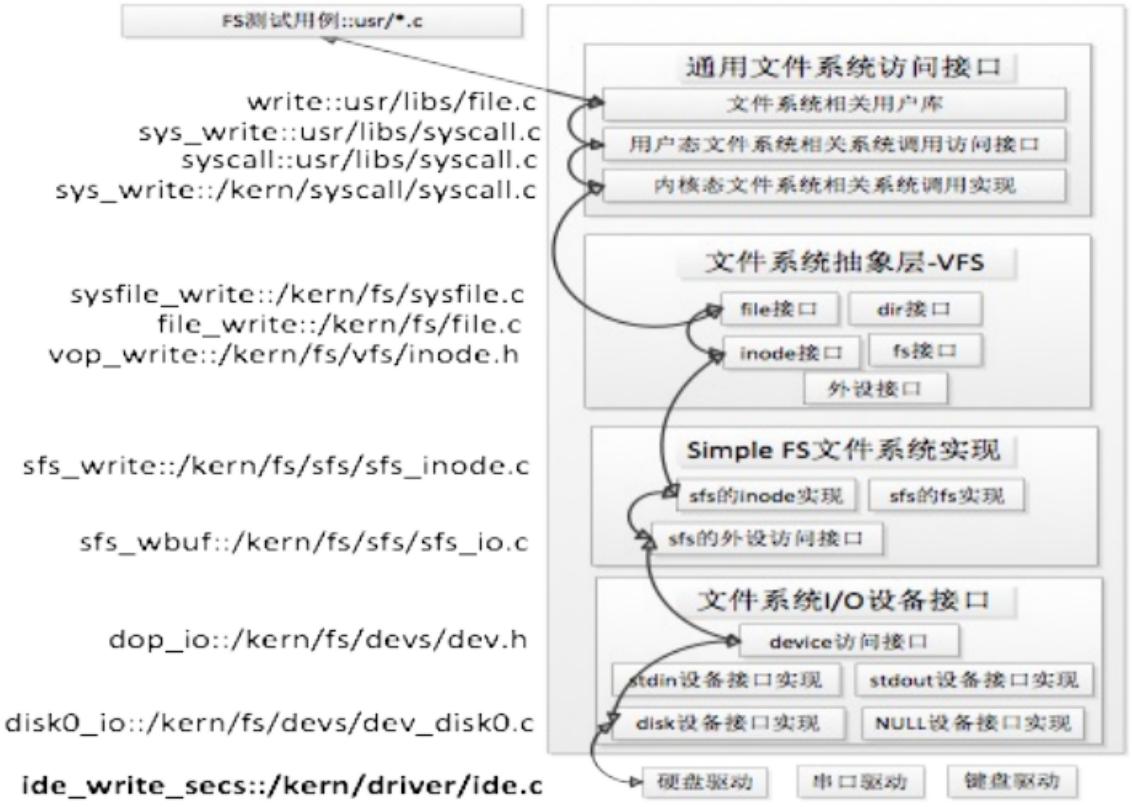
根据实验指导书的介绍，ucore的文件系统模型源于Havard的OS161的文件系统和Linux文件系统。但其实这二者都是源于传统的UNIX文件系统设计。UNIX提出了四个文件系统抽象概念：文件(file)、目录项(dentry)、索引节点(inode)和安装点(mount point)。下面对这四个概念进行介绍：

- 文件：UNIX文件中的内容可理解为是一有序字节buffer，文件都有一个方便应用程序识别的文件名称（也称文件路径名)。典型的文件操作有读、写、创建和删除等。
- 目录项：目录项不是目录，而是目录的组成部分。在UNIX中目录被看作一种特定的文件，而目录项是文件路径中的一部分。如一个文件路径名是"testestfile"，则包含的目录项为 : 根目录"T"，目录"test"和文件"testfile"，这三个都是目录项。一般而言，目录项包含目录项的名字（文件名或目录名）和目录项的索引节点（见下面的描述）位置。
- 索引节点：UNIX将文件的相关元数据信息（如访问控制权限、大小、拥有者、创建时间、数据内容等等信息）存储在一个单独的数据结构中，该结构被称为索引节点。
- 安装点：在UNIX中，文件系统被安装在一个特定的文件路径位置，这个位置就是安装点。所有的已安装文件系统都作为根文件系统树中的叶子出现在系统中。

ucore模仿了UNIX的文件系统设计，ucore的文件系统架构主要由四部分组成：

- 通用文件系统访问接口层：该层提供了一个从用户空间到文件系统的标准访问接口。这一层访问接口让应用程序能够通过一个简单的接口获得ucore内核的文件系统服务；
- 文件系统抽象层：向上提供一个一致的接口给内核其他部分（文件系统相关的系统调用实现模块和其他内核功能模块)访问。向下提供一个同样的抽象函数指针列表和数据结构屏蔽不同文件系统的实现细节；
- Simple FS文件系统层：一个基于索引方式的简单文件系统实例。向上通过各种具体函数实现以对应文件系统抽象层提出的抽象函数。向下访问外设接口；
- 外设接口层：向上提供device访问接口屏蔽不同硬件细节，向下实现访问各种具体设备驱动的接口，比如disk设备接口/串口设备接口/键盘设备接口等。

通过一张图可以理解上述四个部分的关系：



而从ucore操作系统不同的角度来看，ucore中的文件系统架构包含四类主要的数据结构，它们分别是：

- 超级块(SuperBlock)：它主要从文件系统的全局角度描述特定文件系统的全局信息。它的作用范围是整个OS空间。
- 索引节点(inode)：它主要从文件系统的单个文件的角度它描述了文件的各种属性和数据所在位置。它的作用范围是整
  个OS空间。
- 目录项(dentry)：它主要从文件系统的文件路径的角度描述了文件路径中的特定目录。它的作用范围是整个OS空间。
- 文件(ile)：它主要从进程的角度描述了一个进程在访问文件时需要了解的文件标识，文件读写的位置，文件引用情况等信息。它的作用范围是某一具体进程。

接下来分析总结打开一个文件的具体过程：例如某个应用程序需要操作文件，首先需要通过文件系统的通用文件系统访问接口层给用户空间提供的访问接口进入文件系统内部，接着由文件系统抽象层把访问请求转发给某个具体文件系统(比如Simple FS文件系统)，然后再由具体文件系统把应用程序的访问请求转化为对磁盘上的block的处理请求，并通过外设接口层交给磁盘驱动例程来完成具体的磁盘操作。

对于ucore，具体过程如下：

1. 以打开文件为例，首先用户会在进程中调用safe_ open()函数，然后依次调用如下函数open->sys_open->syscall引发系统调用然后进入内核态，然后会由sys_open内核函数处理系统调用，进一步调用到内核函数sysfile_open，然后将字符串"/test/estfile"拷贝到内核空间中的字符串path中，并进入到文件系统抽象层的处理流程完成进一步的打开文件操作中。
2. 在文件系统抽象层，系统会分配一个file数据结构的变量，这个变量其实是current->fs_struct->filemap[]中的一个空元素，即还没有被用来打开过文件，但是分配完了之后还不能找到对应对应的文件结点。所以系统在该层调用了vfs_open 函数通过调用vfs_lookup找到path对应文件的inode，然后调用vop_open函数打开文件。然后层层返回，通过执行语句file-node=node，就把当前进程的current->fs_struct->filemap[fd] (即file 所指变量)的成员变量node指针指向了代表文件的索引节点node，这时返回fd，最后完成打开文件的操作。
3. 在第2步中，调用了SFS文件系统层的vfs_lookup函数去寻找node, 这里在sfs_inode.c中我们能够知道vop_lookup = sfs_lookup。
4. sfs_lookup函数传入的三个参数，其中node是根目录所对应的inode节点；path是
   文件的绝对路径，而node_store是经过查找获得的file所对应的inode节点。函数以”/”为分割符，从左至右逐一分解path获得各个子目录和最终文件对应的inode节点。在本例中是分解出"test"子目录，并调用sfs_lookup_once函数获得"test"子目录对应的inode节点subnode，然后循环进一步调用sfs_lookup_once 查找以"test"子目录下的文件"testfile1"所对应的inode节点。当无法分解path后，就意味着找到了testfile1对应的inode节点，就可顺利返回了。
5. 而我们再进一步观察sfs_lookup_once函数，它调用sfs_dirent_search_nolock函数来查找与路径名匹配的目录项，如果找到目录项，则根据目录项中记录的inode所处的数据块索引值找到路径名对应的SFS磁盘inode，并读入SFS磁盘inode对的内容，创建SFS内存inode。

接下来，按照ucore文件系统架构的主要四个部分结合代码进行分析：

Simple FS文件系统

硬盘中的布局信息一般都储存在SFS中，本实验的SFS文件系统定义在kern/fs/sfs/sfs.h中：

```
struct sfs_fs {
    struct sfs_super super;                         /* on-disk superblock */
    struct device *dev;                             /* device mounted on */
    struct bitmap *freemap;                     /* blocks in use are mared 0 */
    bool super_dirty;                     /* true if super/freemap modified */
    void *sfs_buffer;                    /* buffer for non-block aligned io */
    semaphore_t fs_sem;                             /* semaphore for fs */
    semaphore_t io_sem;                             /* semaphore for io */
    semaphore_t mutex_sem;          /* semaphore for link/unlink and rename */
    list_entry_t inode_list;                        /* inode linked-list */
    list_entry_t *hash_list;                       /* inode hash linked-list */
};
```

根据代码的定义，前三项是三个结构体，对应的就是硬盘文件布局的全局信息，接下来我们一个一个分析：

- 超级块

  其定义：

  ```
  struct sfs_super {
      uint32_t magic;               /* magic number, should be SFS_MAGIC */
      uint32_t blocks;                                /* # of blocks in fs */
      uint32_t unused_blocks;                /* # of unused blocks in fs */
      char info[SFS_MAX_INFO_LEN + 1];           /* infomation for sfs  */
  };
  ```

  超级块是一个文件系统的全局角度描述特定文件系统的全局信息。根据注释知，结构体内定义了标识符magic、总块数blocks、空闲块数unused_blocks还有一些关于SFS的信息，通常是字符串。

- 根目录

  其定义：

  ```
  struct sfs_disk_inode {
      uint32_t size;                      /* size of the file (in bytes) */
      uint16_t type;                          /* one of SYS_TYPE_* above */
      uint16_t nlinks;                /* # of hard links to this file */
      uint32_t blocks;                                /* # of blocks */
      uint32_t direct[SFS_NDIRECT];                   /* direct blocks */
      uint32_t indirect;                              /* indirect blocks */
  };
  ```

  根据上述分析，inode是从文件系统的单个文件的角度它描述了文件的各种属性和数据所在位置，相当于一个索引，而root_dir 是一个根目录索引，根目录表示一开始访问这个文件系统可以看到的目录信息。主要关注direct和indirect，代表根目录下的直接索引和间接索引。

- 目录项

  其定义：

  ```
  struct sfs_disk_entry {
      uint32_t ino;                                   /* inode number */
      char name[SFS_MAX_FNAME_LEN + 1];               /* file name */
  };
  ```

  数组储存文件名，ino是该文件的iNode值。

除了硬盘文件的布局，SFS还需要传递上一层传过来的索引值inode，这个inode实现了硬盘上的实际的索引：

```
struct sfs_inode {
    struct sfs_disk_inode *din;                     /* on-disk inode */
    uint32_t ino;                                   /* inode number */
    bool dirty;                                     /* true if inode modified */
    int reclaim_count;                              /* kill inode if it hits
zero */
    semaphore_t sem;                                /* semaphore for din */
    list_entry_t inode_link;                        /* entry for linked-list in
sfs_fs */
    list_entry_t hash_link;                         /* entry for hash linked-
list in sfs_fs */
};
```

其中，sfs_disk_inode是SFS层的一个成员，它代表了这两个结构的上下级关系。

VFS虚拟文件系统

在VFS层中，我们需要对于虚拟的iNode，和下一层的SFS的iNode进行对接。文件系统抽象层是把不同文件系统的对外共性接口提取出来，这样一来通用文件系统访问接口层只需访问文件系统抽象层，而不需关心具体文件系统的实现细节和接口。

本实验的VFS的定义在kern/fs/vfs/vfs.h中：

```
struct fs {
    union {
        struct sfs_fs __sfs_info;
    } fs_info;                                      // filesystem-specific data
    enum {
        fs_type_sfs_info,
    } fs_type;                                      // filesystem type
    int (*fs_sync)(struct fs *fs);                  // Flush all dirty buffers to
disk
    struct inode *(*fs_get_root)(struct fs *fs);    // Return root inode of
filesystem.
    int (*fs_unmount)(struct fs *fs);               // Attempt unmount of
filesystem.
    void (*fs_cleanup)(struct fs *fs);              // Cleanup of filesystem.???
};
```

里面定义了很多函数指针，用于VFS操作。

而文件结构定义在kern/fs/file.h中：

```
struct file {
    enum {
        FD_NONE, FD_INIT, FD_OPENED, FD_CLOSED,
    } status;                       //文件的执行状态
    bool readable;                  //是否可读
    bool writable;                  //是否可写
    int fd;                         //在filemap中的索引值
    off_t pos;                      //当前位置
    struct inode *node;             //对应的inode指针
    int open_count;                 //打开文件次数
};
```

本层（VFS）的索引iNode定义，简要分析见注释：

```
struct inode {
    union {
        struct device __device_info;            //储存设备文件系统内存inode信息
        struct sfs_inode __sfs_inode_info;      //储存SFS文件系统内存inode信息
    } in_info;
    enum {
        inode_type_device_info = 0x1234,
        inode_type_sfs_inode_info,
    } in_type;                                  //所属文件类型
    int ref_count;
    int open_count;
    struct fs *in_fs;                           //抽象的文件系统
    const struct inode_ops *in_ops;             //抽象的inode操作
};
```

根据分析得，VFS层面的iNode包含了SFS和设备文件两种。

最后是inode操作函数的指针表：

```
/*
 * Abstract operations on a inode.
 *
 * These are used in the form VOP_FOO(inode, args), which are macros
 * that expands to inode->inode_ops->vop_foo(inode, args). The operations
 * "foo" are:
 *
 *    vop_open        - Called on open() of a file. Can be used to
 *                      reject illegal or undesired open modes. Note that
 *                      various operations can be performed without the
 *                      file actually being opened.
 *                      The inode need not look at O_CREAT, O_EXCL, or
 *                      O_TRUNC, as these are handled in the VFS layer.
 *
 *                      VOP_EACHOPEN should not be called directly from
 *                      above the VFS layer - use vfs_open() to open inodes.
 *                      This maintains the open count so VOP_LASTCLOSE can
 *                      be called at the right time.
 *
 *    vop_close       - To be called on *last* close() of a file.
 *
 *                      VOP_LASTCLOSE should not be called directly from
 *                      above the VFS layer - use vfs_close() to close
```

```
 *                      inodes opened with vfs_open().
 *
 *    vop_reclaim     - Called when inode is no longer in use. Note that
 *                      this may be substantially after vop_lastclose is
 *                      called.
 *
 ****************************************
 *
 *    vop_read        - Read data from file to uio, at offset specified
 *                      in the uio, updating uio_resid to reflect the
 *                      amount read, and updating uio_offset to match.
 *                      Not allowed on directories or symlinks.
 *
 *    vop_getdirentry - Read a single filename from a directory into a
 *                      uio, choosing what name based on the offset
 *                      field in the uio, and updating that field.
 *                      Unlike with I/O on regular files, the value of
 *                      the offset field is not interpreted outside
 *                      the filesystem and thus need not be a byte
 *                      count. However, the uio_resid field should be
 *                      handled in the normal fashion.
 *                      On non-directory objects, return ENOTDIR.
 *
 *    vop_write       - Write data from uio to file at offset specified
 *                      in the uio, updating uio_resid to reflect the
 *                      amount written, and updating uio_offset to match.
 *                      Not allowed on directories or symlinks.
 *
 *    vop_ioctl       - Perform ioctl operation OP on file using data
 *                      DATA. The interpretation of the data is specific
 *                      to each ioctl.
 *
 *    vop_fstat        -Return info about a file. The pointer is a
 *                      pointer to struct stat; see stat.h.
 *
 *    vop_gettype     - Return type of file. The values for file types
 *                      are in sfs.h.
 *
 *    vop_tryseek     - Check if seeking to the specified position within
 *                      the file is legal. (For instance, all seeks
 *                      are illegal on serial port devices, and seeks
 *                      past EOF on files whose sizes are fixed may be
 *                      as well.)
 *
 *    vop_fsync       - Force any dirty buffers associated with this file
 *                      to stable storage.
 *
 *    vop_truncate    - Forcibly set size of file to the length passed
 *                      in, discarding any excess blocks.
 *
 *    vop_namefile    - Compute pathname relative to filesystem root
 *                      of the file and copy to the specified io buffer.
 *                      Need not work on objects that are not
 *                      directories.
 *
 ****************************************
 *
 *    vop_creat       - Create a regular file named NAME in the passed
```

```
 *                      directory DIR. If boolean EXCL is true, fail if
 *                      the file already exists; otherwise, use the
 *                      existing file if there is one. Hand back the
 *                      inode for the file as per vop_lookup.
 *
 ****************************************
 *
 *    vop_lookup      - Parse PATHNAME relative to the passed directory
 *                      DIR, and hand back the inode for the file it
 *                      refers to. May destroy PATHNAME. Should increment
 *                      refcount on inode handed back.
 */
struct inode_ops {
    unsigned long vop_magic;
    int (*vop_open)(struct inode *node, uint32_t open_flags);
    int (*vop_close)(struct inode *node);
    int (*vop_read)(struct inode *node, struct iobuf *iob);
    int (*vop_write)(struct inode *node, struct iobuf *iob);
    int (*vop_fstat)(struct inode *node, struct stat *stat);
    int (*vop_fsync)(struct inode *node);
    int (*vop_namefile)(struct inode *node, struct iobuf *iob);
    int (*vop_getdirentry)(struct inode *node, struct iobuf *iob);
    int (*vop_reclaim)(struct inode *node);
    int (*vop_gettype)(struct inode *node, uint32_t *type_store);
    int (*vop_tryseek)(struct inode *node, off_t pos);
    int (*vop_truncate)(struct inode *node, off_t len);
    int (*vop_create)(struct inode *node, const char *name, bool excl, struct
inode **node_store);
    int (*vop_lookup)(struct inode *node, char *path, struct inode
**node_store);
    int (*vop_ioctl)(struct inode *node, int op, void *data);
};
```

函数功能见注释即可。

接下来的内容就是要编码实现了，见下一部分。

代码实现

有了前面的分析，接下来就是实现通用文件系统访问接口层，文件访问接口层的处理流程已经分析过，即首先用户会在进程中调用safe_open()函数，然后依次调用如下函数open->sys_open->syscall引发系统调用然后进入内核态，然后会由sys_open内核函数处理系统调用，进一步调用到内核函数sysfile_open，然后将字符串"/test/estfile"拷贝到内核空间中的字符串path中，并进入到文件系统抽象层的处理流程完成进一步的打开文件操作中。

下面简要分析VFS层处理的主要函数（处理过程详见上一部分），简要分析见注释：

```
/*
 * sfs_lookup - Parse path relative to the passed directory
 *              DIR, and hand back the inode for the file it
 *              refers to.
 */
static int
sfs_lookup(struct inode *node, char *path, struct inode **node_store) {
    struct sfs_fs *sfs = fsop_info(vop_fs(node), sfs);
    assert(*path != '\0' && *path != '/');  //分解path获得各个子目录与相应的节点
    vop_ref_inc(node);
```

```
    struct sfs_inode *sin = vop_info(node, sfs_inode);
    if (sin->din->type != SFS_TYPE_DIR) {
        vop_ref_dec(node);
        return -E_NOTDIR;
    }
    struct inode *subnode;
    int ret = sfs_lookup_once(sfs, sin, path, &subnode, NULL);
    //循环调用，以查找test子目录下文件对应的inode节点
    vop_ref_dec(node);
    if (ret != 0) { //是否查找到了
        return ret;
    }
    *node_store = subnode;
    return 0;
}
```

在此函数内调用了sfs_lookup_once()函数，为此我们在简要分析此函数，以便更好理解程序的功能：

```
*
 * sfs_lookup_once - find inode corresponding the file name in DIR's sin inode
 * @sfs:        sfs file system
 * @sin:        DIR sfs inode in memory
 * @name:       the file name in DIR
 * @node_store: the inode corresponding the file name in DIR
 * @slot:       the logical index of file entry
 */
static int
sfs_lookup_once(struct sfs_fs *sfs, struct sfs_inode *sin, const char *name,
struct inode **node_store, int *slot) {
    int ret;
    uint32_t ino;
    lock_sin(sin);
    {   // find the NO. of disk block and logical index of file entry
        ret = sfs_dirent_search_nolock(sfs, sin, name, &ino, slot, NULL);
    }
    unlock_sin(sin);
    if (ret == 0) {
        // load the content of inode with the the NO. of disk block
        ret = sfs_load_inode(sfs, node_store, ino);
    }
    return ret;
}
```

函数的主要功能在给出的注释中解释的很清楚了，不再具体分析。接下来就是要实现的读部分的函数，也是我们要补全的函数，首先看一下源代码：

```
static int
sfs_io_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, void *buf, off_t
offset, size_t *alenp, bool write) {
    struct sfs_disk_inode *din = sin->din;
    assert(din->type != SFS_TYPE_DIR);
    off_t endpos = offset + *alenp, blkoff;
    *alenp = 0;
    // calculate the Rd/Wr end position
    if (offset < 0 || offset >= SFS_MAX_FILE_SIZE || offset > endpos) {
        return -E_INVAL;
```

```
    }
    if (offset == endpos) {
        return 0;
    }
    if (endpos > SFS_MAX_FILE_SIZE) {
        endpos = SFS_MAX_FILE_SIZE;
    }
    if (!write) {
        if (offset >= din->size) {
            return 0;
        }
        if (endpos > din->size) {
            endpos = din->size;
        }
    }

    int (*sfs_buf_op)(struct sfs_fs *sfs, void *buf, size_t len, uint32_t blkno,
off_t offset);
    int (*sfs_block_op)(struct sfs_fs *sfs, void *buf, uint32_t blkno, uint32_t
nblks);
    if (write) {
        sfs_buf_op = sfs_wbuf, sfs_block_op = sfs_wblock;
    }
    else {
        sfs_buf_op = sfs_rbuf, sfs_block_op = sfs_rblock;
    }

    int ret = 0;
    size_t size, alen = 0;
    uint32_t ino;
    uint32_t blkno = offset / SFS_BLKSIZE;          // The NO. of Rd/Wr begin
block
    uint32_t nblks = endpos / SFS_BLKSIZE - blkno;  // The size of Rd/Wr blocks

  //LAB8:EXERCISE1 YOUR CODE HINT: call sfs_bmap_load_nolock, sfs_rbuf,
sfs_rblock,etc. read different kind of blocks in file
    /*
     * (1) If offset isn't aligned with the first block, Rd/Wr some content from
offset to the end of the first block
     *       NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
     *               Rd/Wr size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) :
(endpos - offset)
     * (2) Rd/Wr aligned blocks
     *       NOTICE: useful function: sfs_bmap_load_nolock, sfs_block_op
     * (3) If end position isn't aligned with the last block, Rd/Wr some content
from begin to the (endpos % SFS_BLKSIZE) of the last block
     *       NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
    */
out:
    *alenp = alen;
    if (offset + alen > sin->din->size) {
        sin->din->size = offset + alen;
        sin->dirty = 1;
    }
    return ret;
}
```

可能调用的函数、参数在注释部分都已经给出了提示，就是读三部分的数据，具体实现与分析见注释即可：

```
static int
sfs_io_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, void *buf, off_t
offset, size_t *alenp, bool write) {
    struct sfs_disk_inode *din = sin->din;
    assert(din->type != SFS_TYPE_DIR);
    off_t endpos = offset + *alenp, blkoff;
    *alenp = 0;
    // calculate the Rd/Wr end position
    if (offset < 0 || offset >= SFS_MAX_FILE_SIZE || offset > endpos) {
        return -E_INVAL;
    }
    if (offset == endpos) {
        return 0;
    }
    if (endpos > SFS_MAX_FILE_SIZE) {
        endpos = SFS_MAX_FILE_SIZE;
    }
    if (!write) {
        if (offset >= din->size) {
            return 0;
        }
        if (endpos > din->size) {
            endpos = din->size;
        }
    }

    int (*sfs_buf_op)(struct sfs_fs *sfs, void *buf, size_t len, uint32_t blkno,
off_t offset);
    int (*sfs_block_op)(struct sfs_fs *sfs, void *buf, uint32_t blkno, uint32_t
nblks);
    if (write) {
        sfs_buf_op = sfs_wbuf, sfs_block_op = sfs_wblock;
    }
    else {
        sfs_buf_op = sfs_rbuf, sfs_block_op = sfs_rblock;
    }

    int ret = 0;
    size_t size, alen = 0;
    uint32_t ino;
    uint32_t blkno = offset / SFS_BLKSIZE;          // The NO. of Rd/Wr begin
block
    uint32_t nblks = endpos / SFS_BLKSIZE - blkno;  // The size of Rd/Wr blocks
        //读取第一部分数据：
    if ((blkoff = offset % SFS_BLKSIZE) != 0) {
        //计算第一个数据块大小：
        size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
        //找到内存文件索引对应的block编号
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0) {
            goto out;
        }
```

```
        alen += size;
        if (nblks == 0) {
            goto out;
        }
        buf += size, blkno ++, nblks --;
    }

    //读取中间部分的数据，规定一个size，然后每次读都只读size的大小：
    size = SFS_BLKSIZE;
    while (nblks != 0) {
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        if ((ret = sfs_block_op(sfs, buf, ino, 1)) != 0) {
            goto out;
        }
        alen += size, buf += size, blkno ++, nblks --;
    }
    //读取第三部分的数据：
    if ((size = endpos % SFS_BLKSIZE) != 0) {
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        if ((ret = sfs_buf_op(sfs, buf, size, ino, 0)) != 0) {
            goto out;
        }
        alen += size;
    }
    //读取结束
out:
    *alenp = alen;
    if (offset + alen > sin->din->size) {
        sin->din->size = offset + alen;
        sin->dirty = 1;
    }
    return ret;
}
```

> - 请在实验报告中给出设计实现“UNIX的PIPE机制”的概要设计方案。

为了实现UNIX的PIPE机制，可以在磁盘上保留一部分空间作为pipe机制的缓冲区，接下来将说明如何完成对pipe机制的支持:

- 当某两个进程之间要求建立管道，设进程A的标准输出作为进程B的标准输入，那么可以在这两个进程的进程控制块上新增变量来记录进程的这种属性；并且同时生成一个临时的文件，并将其在进程A,B中打开;
- 当进程A使用标准输出进行write系统调用的时候，通过PCB中的变量可以知道，需要将这些标准输出的数据输出到先前提高的临时文件中去;
- 当进程B使用标准输入的时候进行read系统调用的时候，根据其PCB中的信息可以知道，需要从上述的临时文件中读取数据。

至此完成了对pipe机制的设计。

事实上，由于在真实的文件系统和用户之间还由一层虚拟文件系统，因此我们也可以不把数据缓冲在磁盘上，而是直接保存在内存中，然后完成一个根据虚拟文件系统的规范完成一个虚拟的pipe文件，然后进行输入输出的时候只要对这个文件进行操作即可。

练习2：完成基于文件系统的执行程序机制的实现

算法分析

根据指导书的要求，我们要修改proc.c中的相关函数，要修改的内容直接看proc.c文件中的注释提示即可。

代码实现

首先是alloc_proc函数，注释提示说我们要增加一些初始化的内容：

```
 //LAB8:EXERCISE2 YOUR CODE HINT:need add some code to init fs in proc_struct,
...
```

一个文件需要在VFS 中变为一个进程才能被执行，因此需要修改proc初始化内容，加上一句proc->filesp =NULL，完成初始化：

```
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        proc->state = PROC_UNINIT;
        proc->pid = -1;
        proc->runs = 0;
        proc->kstack = 0;
        proc->need_resched = 0;
        proc->parent = NULL;
        proc->mm = NULL;
        memset(&(proc->context), 0, sizeof(struct context));
        proc->tf = NULL;
        proc->cr3 = boot_cr3;
        proc->flags = 0;
        memset(proc->name, 0, PROC_NAME_LEN);
        proc->wait_state = 0;
        proc->cptr = proc->optr = proc->yptr = NULL;
        proc->rq = NULL;
        proc->run_link.prev = proc->run_link.next = NULL;
        proc->time_slice = 0;
        proc->lab6_run_pool.left = proc->lab6_run_pool.right = proc->
lab6_run_pool.parent = NULL;
        proc->lab6_stride = 0;
        proc->lab6_priority = 0;
        proc->filesp = NULL;
    }
    return proc;
}
```

然后是load_icode函数，我们需要完成这个函数的定义，我们先看一下函数内的注释，即给出的提示：

```
static int
load_icode(int fd, int argc, char **kargv) {
    /* LAB8:EXERCISE2 YOUR CODE  HINT:how to load the file with handler fd  in to
process's memory? how to setup argc/argv?
     * MACROs or Functions:
     *  mm_create        - create a mm
```

```
     *  setup_pgdir      - setup pgdir in mm
     *  load_icode_read  - read raw data content of program file
     *  mm_map           - build new vma
     *  pgdir_alloc_page - allocate new memory for  TEXT/DATA/BSS/stack parts
     *  lcr3             - update Page Directory Addr Register -- CR3
     */
    /* (1) create a new mm for current process
     * (2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT
     * (3) copy TEXT/DATA/BSS parts in binary to memory space of process
     *    (3.1) read raw data content in file and resolve elfhdr
     *    (3.2) read raw data content in file and resolve proghdr based on info
in elfhdr
     *    (3.3) call mm_map to build vma related to TEXT/DATA
     *    (3.4) callpgdir_alloc_page to allocate page for TEXT/DATA, read
contents in file
     *          and copy them into the new allocated pages
     *    (3.5) callpgdir_alloc_page to allocate pages for BSS, memset zero in
these pages
     * (4) call mm_map to setup user stack, and put parameters into user stack
     * (5) setup current process's mm, cr3, reset pgidr (using lcr3 MARCO)
     * (6) setup uargc and uargv in user stacks
     * (7) setup trapframe for user environment
     * (8) if up steps failed, you should cleanup the env.
     */
}
```

完整实现，过程见函数注释：

```
static int
load_icode(int fd, int argc, char **kargv) {
        //1.建立内存管理器：
        if (current->mm != NULL) {  // 判断当前进程的mm是否释放掉
            panic("load_icode: current->mm must be empty.\n");
        }

    int ret = -E_NO_MEM;
    struct mm_struct *mm;       //建立内存管理器

    if ((mm = mm_create()) == NULL) { // 为进程创建一个新的mm
        goto bad_mm;
    }

    //2.建立页项目：
    if ((ret = setup_pgdir(mm)) != 0) { // 进行页表项的设置
        goto bad_pgdir_cleanup_mm;
    }
    struct Page *page; //建立页表

    //3.从文件加载到程序内存
    struct elfhdr elf, *elfp = &elf;
    off_t offset = 0;
    load_icode_read(fd, (void *) elfp, sizeof(struct elfhdr), offset); // 从磁盘上
读取出elf文件头
    offset += sizeof(struct elfhdr);
    if (elfp->e_magic != ELF_MAGIC) { // 判断该ELF文件是否合法
        ret = -E_INVAL_ELF;
        goto bad_elf_cleanup_pgdir;
```

```
    }

    struct proghdr ph, *php = &ph;
    uint32_t vm_flags, perm;
    int i = 0;
    for (i = 0; i < elfp->e_phnum; i++) { // 根据elf-header中的信息，找到每一个
program header
        load_icode_read(fd, (void *) php, sizeof(struct proghdr), elfp->e_phoff
+ i * sizeof(struct proghdr)); // 读取program header

        if (php->p_type != ELF_PT_LOAD) {
            continue;
        }

        if (php->p_filesz > php->p_memsz) {
            ret = -E_INVAL_ELF;     //错误
            goto bad_cleanup_mmap;
        }

        if (php->p_filesz == 0) {
            continue;
        }

        vm_flags = 0, perm = PTE_U;     //建立虚拟与物理地址之间的映射
        if (php->p_flags & ELF_PF_X) vm_flags |= VM_EXEC;//对各个段的权限进行设置
        if (php->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;
        if (php->p_flags & ELF_PF_R) vm_flags |= VM_READ;
        if (vm_flags & VM_WRITE) perm |= PTE_W;
        // 将这些段的虚拟内存地址设置为合法的：
        if ((ret = mm_map(mm, php->p_va, php->p_memsz, vm_flags, NULL)) != 0) {
            goto bad_cleanup_mmap;
        }

        offset = php->p_offset;
        size_t off, size;
        uintptr_t start = php->p_va, la = ROUNDDOWN(start, PGSIZE);
        uintptr_t end = php->p_va + php->p_filesz; //计算数据段和代码终止地址
        ret = -E_NO_MEM;
        //将代码和数据复制
        end = php->p_va + php->p_memsz; //计算数据和代码的终止地址
        while (start < end) {
            // 为TEXT/DATA段逐页分配物理内存空间：
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
                goto bad_cleanup_mmap;
            }
            off = start - la;
            size = PGSIZE - off;
            la += PGSIZE;
            if (end < la) {
                size -= la - end;
            }
            //读取size大小的块
            load_icode_read(fd, page2kva(page) + off, size, offset);
            // 将磁盘上的TEXT/DATA段读入到分配好的内存空间中去
            start += size;
            offset += size;
        }
        //建立BBS段
```

```
        end = php->p_va + php->p_memsz; //计算数据和代码的终止地址
        if (start < la) { //若剩余的部分被BSS段占用，则进行清零初始化
            if (start == end) {
                continue;
            }
            off = start + PGSIZE - la, size = PGSIZE - off;
            if (end < la) {
                size -= la - end;
            }
            memset(page2kva(page) + off, 0, size); // init all BSS data with 0
            start += size;
            assert((end < la && start == end) || (end >= la && start == la));
        }

        while (start < end) {  //若BSS段还需要更多的内存空间，进一步进行分配
            //为BSS段分配新的物理内存页
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
                goto bad_cleanup_mmap;
            }
            off = start - la, size = PGSIZE - off, la += PGSIZE;
            if (end < la) {
                size -= la - end;
            }
            //同样，每次操作size大小
            memset(page2kva(page), 0, size); // 将分配到的空间清零初始化
            start += size;
        }
    }
    sysfile_close(fd); // 关闭传入的文件

    //4.建立虚拟内存的映射表
    vm_flags = VM_READ | VM_WRITE | VM_STACK;   //设置用户栈的权限
    // 将用户栈所在的虚拟内存区域设置为合法的：
    if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL))
!= 0) {
        goto bad_cleanup_mmap;
    }

    //5.设置用户栈
    mm_count_inc(mm);  // 切换到用户的内存空间
    current->mm = mm;
    current->cr3 = PADDR(mm->pgdir);
    lcr3(PADDR(mm->pgdir));

   //6.处理用户栈传入的参数
    uint32_t argv_size=0;
    uintptr_t stacktop = USTACKTOP - (argv_size/sizeof(long)+1)*sizeof(long);

    for (i = 0; i < argc; i ++) {       //确定传入给应用程序的参数占用多少空间
        argv_size += strnlen(kargv[i],EXEC_MAX_ARG_LEN + 1)+1;
    }
    char** uargv=(char **)(stacktop  - argc * sizeof(char *));

    argv_size = 0;
    for (i = 0; i < argc; i ++) {       //将参数取出来放到数组中
        uargv[i] = strcpy((char *)(stacktop + argv_size ), kargv[i]);
        argv_size +=  strnlen(kargv[i],EXEC_MAX_ARG_LEN + 1)+1;
    }
```

```
    stacktop = (uintptr_t)uargv - sizeof(int);   //计算用户栈顶
    *(int *)stacktop = argc;

    //7.设置进程的中断帧
    struct trapframe *tf = current->tf; //中断帧
    memset(tf, 0, sizeof(struct trapframe)); //初始化
    tf->tf_cs = USER_CS; //返回到用户态，使用用户态的数据段和代码段
    tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
    tf->tf_esp = stacktop; // 栈顶位置为先前计算过的栈顶位置
    tf->tf_eip = elfp->e_entry; // 将返回地址设置为用户程序的入口
    tf->tf_eflags = 0x2 | FL_IF;
    ret = 0;

    //8.错误处理
out:
    return ret;          //返回
bad_cleanup_mmap:       // 进行加载失败的一系列清理操作
    exit_mmap(mm);
bad_elf_cleanup_pgdir:
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
bad_mm:
    goto out;

}
```

与答案相比，大部分的思路一致，主要是根据注释的提示完成的，主要是少了很多判断语句，即缺少部分对参数合法化的判断，导致程序鲁棒性不是很好。除此以外，处理用户栈的参数部分，程序原本没用数组，后来看了答案才想起来可以用数组，然后就修改了一下，之前的代码太复杂了.....

至此，练习2的主要部分已经完成。

> - 请在实验报告中给出设计实现基于"UNIX的硬链接和软链接机制"的概要设计方案。

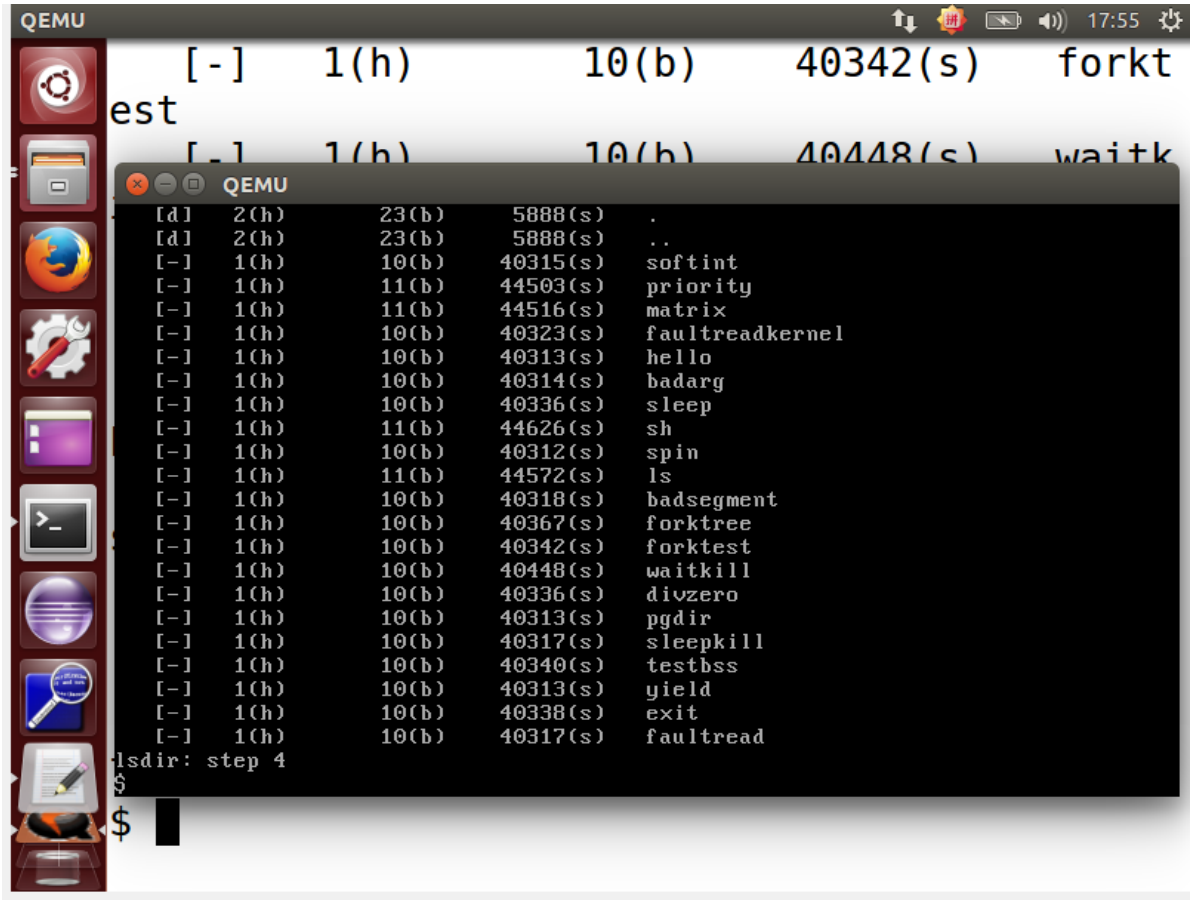
保存在磁盘上的inode信息均存在一个nlinks变量，用于表示当前文件的被链接的计数，因而支持实现硬链接和软链接机制;

- 如果在磁盘上创建一个文件A的软链接B，那么将B当成正常的文件创建inode，然后将TYPE域设置为链接，使用剩余的域中的一个，指向A的 inode位置，然后再额外使用一个位来标记当前的链接是软链接还是硬链接;
- 当访问到文件B (read，write等系统调用)，判断如果B是一个链接，则实际是将对B指向的文件A(已经知道了A的inode位置)进行操作;
- 当删除一个软链接B的时候，直接将其在磁盘上的inode 删掉即可;
  如果在磁盘上的文件A创建一个硬链接B，那么在按照软链接的方法创建完B之后，还需要将A中的被链接的计数加1;
- 访问硬链接的方式与访问软链接是一致的;
- 当删除一个硬链接B的时候，除了需要删除掉B的inode 之外，还需要将B指向的文件A的被链接计数减1，如果减到了0，则需要将A删除掉;

实验结果

下面对LAB8的代码进行验证。

按照指导书的要求运行make qemu，输入ls命令结果：



结果与指导书上的内容一致。完成LAB8的实验。

感想与总结

本次实验了解了文件系统的实现与操作流程，对其有了更透彻的理解。本次实验接触到了相较于之前实验比较新的内容，而且代码量很大，自己查阅很多课后的资料，参考着答案进行一步一步地完成实验，认识到自己在这方面仍然有很大的提升空间，通过这学期的四次实验，我也对操作系统这门课有了新的认识，锻炼了我读写代码的能力，对操作系统背后的各种算法与实现有了深刻地了解。这四次实验带给我了很多，除了应该掌握的知识以外，还有对Linux系统的熟悉等等，最后也感谢这门课让我的课余生活十分充实.....

相关知识点

本实验中相关的知识点

- 文件和文件系统
- 文件管理与访问

本实验未涉及

- IO的管理与磁盘调度

参考资料

1. https://www.iteye.com/topic/816268
2. https://blog.csdn.net/kension/article/details/3796603
3. https://www.cnblogs.com/alantu2018/p/8461749.html