

Lab8—文件系统

一、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；

二、实验内容：

练习0：填写已有实验

利用 meld 将 lab1-7 的代码导入即可。

练习1：完成读文件操作的实现

ucore文件系统的总体结构如下图所示：



就像计算机网络一样，采用分层结构，下层为上层提供服务（接口），使得上层不用操心底层的具体实现，而上层通过调用接口使用下层实现的功能。

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 (file)，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。它的作用范围是某一具体进程。

其中，索引节点 (inode) 是一个比较重要的结构，内核会给每一个新建文件分配一个索引节点 inode，它包含了一个文件的长度、创建及修改时间、权限、所属关系、磁盘中的位置等信息。一个文件系统维护了一个索引节点的数组，每个文件或目录都与索引节点数组中的唯一一个元素对应。系统给每个索引节点分配了一个号码，也就是该节点在数组中的索引号，称为索引节点号。

内容:

按照要求，需要完善 sfs_inode.c 中的 sfs_io_nolock 函数。

首先看一下已经有了的部分代码：

```
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; // 建立了内存中的inode和disk上的inode的对应关系
    assert(din->type != SFS_TYPE_DIR);
    off_t endpos = offset + *alenp, blkoff; // 计算以下endposition
    *alenp = 0;
    // calculate the Rd/wr end position 根据endpos的计算结果做一些处理
    if (offset < 0 || offset >= SFS_MAX_FILE_SIZE || offset > endpos) {
        return -E_INVALID;
    }
    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;
        }
    }

    // 定义了两个函数指针，分别针对以字节和以块（页面大小）为单位进行读/写
    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) { // 如果是写
        sfs_buf_op = sfs_wbuf, sfs_block_op = sfs_wblock;
    }
    else { // 如果是读
        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 读/写起始块的编号
uint32_t nblks = endpos / SFS_BLKSIZE - blkno;  // The size of Rd/Wr blocks
需要读/写的块数量

out:
*alenp = alen;
if (offset + alen > sin->din->size) {
    sin->din->size = offset + alen;
    sin->dirty = 1;
}
return ret;
}

```

其中前两个传入参数的相关定义如下：

1. sfs_fs结构体定义：

```

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 */
};

```

这是对磁盘上SFS文件系统的布局也就是对下面这个结构的定义：



2. sfs_inode结构体定义：

```

/* inode for sfs */
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 */
};

```

这是内存中的索引节点定义，其中包含了磁盘上的 inode 信息，正好对应 `sfs_io_nolock()` 中的首行代码。

根据注释

```

//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
 */

```

大概可以明白需要注意的一些点（以读为例）：offset 指示了读文件的起始位置，但是这个位置可能并没有和要读取的第一块的起始地址对齐，因此对于第一块，我们采用 `sfs_rbuf` 的形式以字节为单位从 offset 读到第一块的末尾；同样，对于计算得到的 end position 来说，其指示了读文件的结束位置，但该位置可能并没有和需要读取的最后一块的结束地址对齐，因此也是采用 `sfs_rbuf` 的形式从最后一块的起始地址读到 endpos 的位置；那么对于中间的来讲，由于都是完整的块，则可以采用 `sfs_rblock` 以块为单位来进行读操作。

补充代码如下：

```

static int
sfs_io_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, void *buf, off_t
offset, size_t *alenp, bool write) {
    .....
    //按照前面讲过的，读取文件分为三部分，每一部分大致都是按照先用
sfs_bmap_load_nolock 来定位磁盘上实际的文件，然后使用 sfs_buf_op / sfs_block_op 以
字节或以块为单位进行读操作。

    // 第一部分，读取开头部分的数据
    if ((blkoff = offset % SFS_BLKSIZE) != 0) {

```

```

        size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset); //
计算第一个数据块的大小
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            /* sfs_bmap_load_nolock - according to the DIR's inode and the
logical index of block in inode, find the NO. of disk block. 也就是根据逻辑索引
找到磁盘上对应 block 的编号 ino */
            goto out;
        }

        // 进行读操作，sfs_buf_op 是前面定义的函数指针，根据参数write的值会被赋以相应
读/写函数。同时注意到第一块不采用以块为单位的读。
        if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0) {
            goto out;
        }

        alen += size;
        if (nblks == 0) { // nblks == 0 代表已完成指定大小块的读操作。
            goto out;
        }
        buf += size, blkno ++, nblks --;
    }

    // 第二部分，读取中间部分的数据，一次读一块
    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 --;
    }

    // 第三部分，读取结束部分的数据， 同样是采用 sfs_buf_op
    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;
    }

out:
    *alenp = alen;
    if (offset + alen > sin->din->size) {
        sin->din->size = offset + alen;
        sin->dirty = 1;
    }
    return ret;
}

```

问题：

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，鼓励给出详细设计方案。

首先说明一下PIPE：

管道（PIPE）是一种进程间的通信机制。通常，管道由一个进程创建，用于父进程和子进程间的通讯，一个进程内的管道是没有意义的。管道是半双工的，数据只流向一个方向即从一个进程流向另一个进程（其中一个读管道，一个写管道）。

结合以上管道的特点，可以设想一个方案：

首先得有一个缓冲区作为管道，使得进程A可以往里写数据而进程B可以从中读数据，这个缓冲区作为两个进程共享的一块区域，如此就需要在 PCB 里增加一项用以记录该区域的地址，通过练习2知道 PCB 里已经新增了一个文件指针 filesp，那么该区域就可以看作是一个文件，利用 filesp 来指向它；同时，为了让每一个进程清楚自己是读进程还是写进程，就又需要在PCB里再增加一个成员变量 W/R 用来指示当前进程的功能是读还是写。所以

- 如果进程A和进程B之间要建立管道，假如是A写B读，则首先要设置PCB中变量W/R，A为写B为读，同时生成一个文件，并将其在进程A，B中打开，让二者的 filesp 指向该文件；
- 当进程 A write 的时候，通过 filesp 可以知道需要将这些数据写入到哪个文件；
- 当进程 B read 的时候，通过 filesp 可以知道从哪个文件读取数据；
- 完成读写后将创建的文件删除。

练习2：完成基于文件系统的执行程序机制的实现

内容：

按要求查看proc.c中的load_icode函数，首先注意到 HINT 中的以下语句

how to load the file with handler fd in to process's memory

这提示了我们需要把文件装载到进程的内存空间中，那么对一个进程而言，它就应该知道自己当前所打开的文件，这只能通过在 PCB 中增加表项实现，所以看到现在的 PCB 中新增了一项

```
struct files_struct *filesp;           // the file related info(pwd,
files_count, files_array, fs_semaphore) of process
```

用来描述跟当前进程有关的文件的相关信息，其定义如下：

```
/*
 * process's file related information
 */
struct files_struct {
    struct inode *pwd;           // inode of present working directory
    struct file *fd_array;       // opened files array
    int files_count;             // the number of opened files
    semaphore_t files_sem;       // lock protect sem
};
```

同时，需要在alloc_proc函数中对这一新的表项进行初始化：

```
proc->filesp = NULL;
```

然后对于 load_icode 函数，该函数需要完成的工作注释基本涵盖全了，于是就按顺序写下去。

这部分很大程度是参考了答案，靠自己完成较为困难... 大体上，注意使用注释给出的函数，同时在调用的时候采用条件判断语句，在接收到代表错误的返回值时进行相应的错误处理。

前两个比较容易：

(1) create a new mm for current process

这是要给进程创建一个新的内存管理单元，那么在创建之前，先得判断当前的 mm 是否已经释放掉了，如果没有，报错退出，反之则调用 mm_create() 来创建新的 mm，同时，如果创建失败，需要做相应的错误处理。

```
// 判断当前进程的 mm 是否已经被释放掉了
if (current->mm != NULL) {
    panic("load_icode: current->mm must be empty.\n");
}

int ret = -E_NO_MEM;    // Request failed due to memory shortage
struct mm_struct *mm; // 定义 mm
if ((mm = mm_create()) == NULL) { // 为进程创建一个新的 mm
    goto bad_mm; // 错误处理
}
```

(2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT

建立一个新的页目录表，通过调用 setup_pgdir 来实现。

```
if (setup_pgdir(mm) != 0) {
    goto bad_pgdir_cleanup_mm;
}

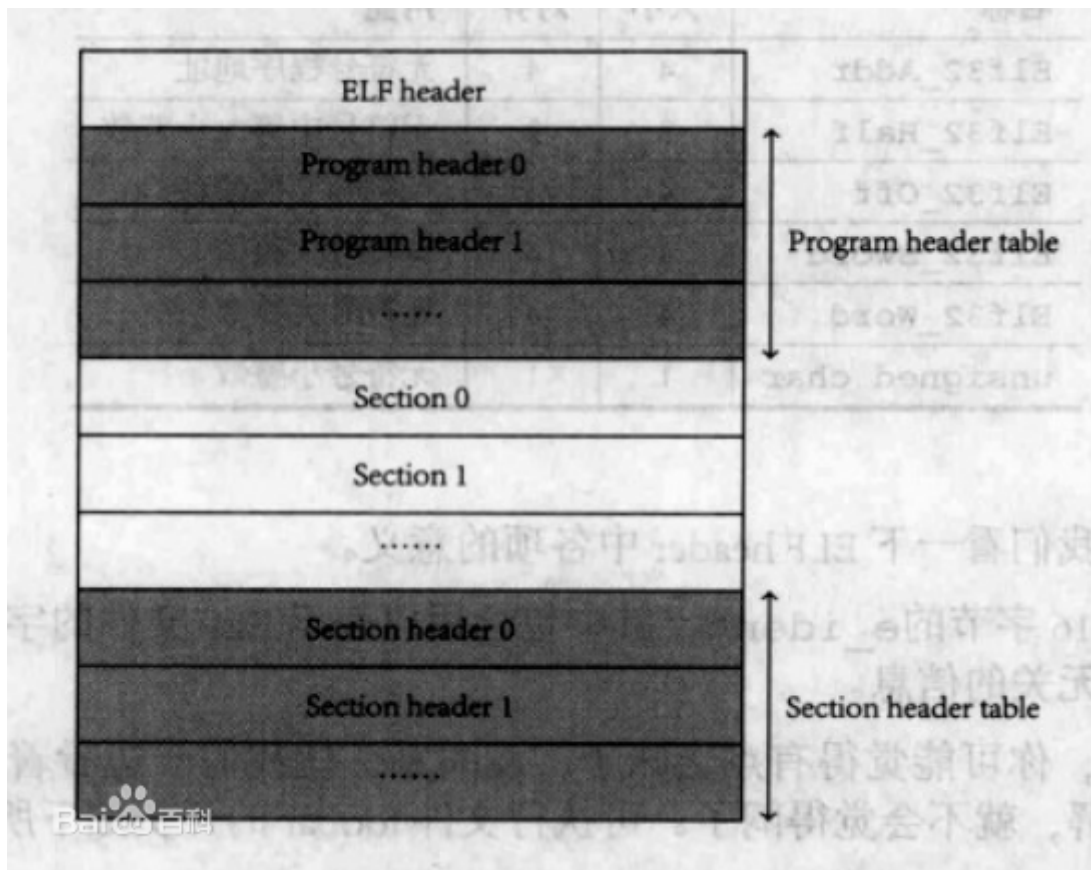
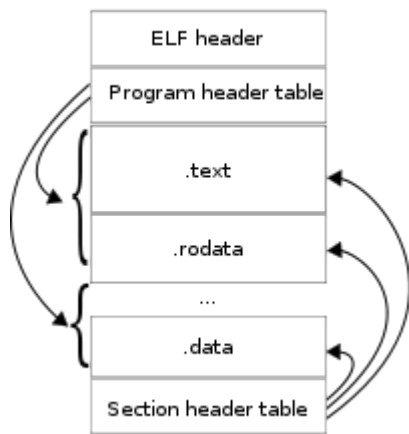
struct Page *page;
```

下面开始有点麻烦。

(3) copy TEXT/DATA/BSS parts in binary to memory space of process

需要把磁盘中文件的代码段/数据段/ BSS 段搬到内存中。

这里先看一下 elf 文件的格式：



其中 ELF header 的位置是固定的，其余部分是可变的，具体如何由 ELF header 来说明。所以首先应该读取 ELF header。

结合注释，同样也是要求先读取 elf 文件头，然后再利用 elf 头信息来获取程序头表。

(3.1) read raw data content in file and resolve elfhdr

```

struct elfhdr __elf, *elf = &__elf;
// 读取 elf 文件头
if ((ret = load_icode_read(fd, elf, sizeof(struct elfhdr), 0)) != 0) {
    goto bad_elf_cleanup_pgdir;
}

// 判断该 ELF 文件是否合法
if (elf->e_magic != ELF_MAGIC) {
    ret = -E_INVALID_ELF; // Invalid elf file
    goto bad_elf_cleanup_pgdir;
}

```

根据 elfhdr 的定义可以看出 e_magic 必须等于 ELF_MAGIC。


```

/* file header */
struct elfhdr {
    uint32_t e_magic;      // must equal ELF_MAGIC
    uint8_t e_elf[12];
    uint16_t e_type;       // 1=relocatable, 2=executable, 3=shared object,
4=core image
    uint16_t e_machine;    // 3=x86, 4=68K, etc.
    uint32_t e_version;    // file version, always 1
    uint32_t e_entry;      // entry point if executable
    uint32_t e_phoff;      // file position of program header or 0
    uint32_t e_shoff;      // file position of section header or 0
    uint32_t e_flags;      // architecture-specific flags, usually 0
    uint16_t e_ehsize;     // size of this elf header
    uint16_t e_phentsize;  // size of an entry in program header
    uint16_t e_phnum;      // number of entries in program header or 0
    uint16_t e_shentsize;  // size of an entry in section header
    uint16_t e_shnum;      // number of entries in section header or 0
    uint16_t e_shstrndx;   // section number that contains section name strings
};

```

```

// (3.2) read raw data content in file and resolve proghdr based on info in
elfhdr
struct proghdr __ph, *ph = &__ph;
uint32_t vm_flags, perm, phnum;
// 根据 elf-header 中的信息, 找到每一个 program header
// elf->e_phnum 代表了程序段的数目
for (phnum = 0; phnum < elf->e_phnum; phnum++) {
    off_t phoff = elf->e_phoff + sizeof(struct proghdr) * phnum; // 读取程序
的每个段的头部
    if ((ret = load_icode_read(fd, ph, sizeof(struct proghdr), phoff)) != 0)
{ // 读取program header
    goto bad_cleanup_mmap;
}
    if (ph->p_type != ELF_PT_LOAD) {
        continue;
    }
    if (ph->p_filesz > ph->p_memsz) {
        ret = -EINVAL;
        goto bad_cleanup_mmap;
    }
    if (ph->p_filesz == 0) {
        continue;
    }

    // (3.3) call mm_map to build vma related to TEXT/DATA
    vm_flags = 0, perm = PTE_U;
    // 根据刚刚读取到的 proghdr 中的信息, 对各个段的权限进行设置
    if (ph->p_flags & ELF_PF_X) vm_flags |= VM_EXEC;
    if (ph->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;
    if (ph->p_flags & ELF_PF_R) vm_flags |= VM_READ;
    if (vm_flags & VM_WRITE) perm |= PTE_W;
    // 建立虚拟地址与物理地址之间的映射
    if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0) {
        goto bad_cleanup_mmap;
    }
}

```

```

}
off_t offset = ph->p_offset;
size_t off, size;
uintptr_t start = ph->p_va, end, la = ROUNDDOWN(start, PGSIZE);

ret = -E_NO_MEM;

// (3.4) call pgdir_alloc_page to allocate page for TEXT/DATA, read
contents in file and copy them into the new allocated pages
// 为代码段/数据段分配页框, 将其读入内存
// 计算数据段和代码段终止地址
end = ph->p_va + ph->p_filesz;
while (start < end) {
    if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
        ret = -E_NO_MEM;
        goto bad_cleanup_mmap;
    }
    off = start - la, size = PGSIZE - off, la += PGSIZE;
    if (end < la) {
        size -= la - end;
    }
    //分配完成, 将代码段/数据段读入内存
    if ((ret = load_icode_read(fd, page2kva(page) + off, size, offset))
    != 0) {
        goto bad_cleanup_mmap;
    }
    start += size, offset += size;
}

// (3.5) call pgdir_alloc_page to allocate pages for BSS, memset zero in
these pages 同上方式来读取 BSS 段
end = ph->p_va + ph->p_memsz;
// 如果存在 BSS 段, 并且先前的 TEXT/DATA 段分配的最后一页没有被完全占用, 则剩余的部
分被BSS段占用
if (start < la) {
    if (start == end) {
        continue ;
    }
    off = start + PGSIZE - la, size = PGSIZE - off;
    if (end < la) {
        size -= la - end;
    }
    memset(page2kva(page) + off, 0, size);
    start += size;
    assert((end < la && start == end) || (end >= la && start == la));
}

// 如果 BSS 段还需要更多的内存空间的话, 进一步进行分配
while (start < end) {
    if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
        ret = -E_NO_MEM;
        goto bad_cleanup_mmap;
    }
    off = start - la, size = PGSIZE - off, la += PGSIZE;
    if (end < la) {
        size -= la - end;
    }
    // memset zero in these pages

```

```

        memset(page2kva(page) + off, 0, size);
        start += size;
    }
}
sysfile_close(fd); // 关闭文件，读取结束

```

(4) call mm_map to setup user stack, and put parameters into user stack

```

// 设置用户栈的权限
vm_flags = VM_READ | VM_WRITE | VM_STACK;
if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL))
!= 0) {
    goto bad_cleanup_mmap;
}
assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-PGSIZE, PTE_USER) != NULL);
assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-2*PGSIZE, PTE_USER) != NULL);
assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-3*PGSIZE, PTE_USER) != NULL);
assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-4*PGSIZE, PTE_USER) != NULL);
mm_count_inc(mm);

```

(5) setup current process's mm, cr3, reset pgidr (using lcr3 MARCO)

```

current->mm = mm;
current->cr3 = PADDR(mm->pgdir);
lcr3(PADDR(mm->pgdir));

```

(6) setup uargc and uargv in user stacks

设置用户栈中传入的参数，其中 uargc 为一个整数，用来统计运行程序时送给 main 函数的命令行参数的个数，uargv[] 为一个数组指针，用来存放指向输入的字符串参数的指针，每一个元素指向一个参数。

```

uint32_t argv_size=0, i;
// 所有参数的总长度
for (i = 0; i < argc; i++) {
    argv_size += strlen(kargv[i], EXEC_MAX_ARG_LEN + 1) + 1;
}

uintptr_t stacktop = USTACKTOP - (argv_size/sizeof(long)+1)*sizeof(long);
char** uargv=(char **)(stacktop - argc * sizeof(char *));

argv_size = 0;
//把 kargv 中的参数放置到栈空间同时使 uargv 指向这些参数
for (i = 0; i < argc; i++) {
    uargv[i] = strcpy((char *) (stacktop + argv_size), kargv[i]);
    argv_size += strlen(kargv[i], EXEC_MAX_ARG_LEN + 1) + 1;
}

stacktop = (uintptr_t)uargv - sizeof(int);
*(int *)stacktop = argc;

```

(7) setup trapframe for user environment

设置中断帧

```

struct trapframe *tf = current->tf;
memset(tf, 0, sizeof(struct trapframe));
tf->tf_cs = USER_CS;
tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
tf->tf_esp = stacktop;
tf->tf_eip = elf->e_entry;
tf->tf_eflags = FL_IF;
ret = 0;

```

(8) if up steps failed, you should cleanup the env.

错误处理

```

out:
    return ret;
bad_cleanup_mmap:
    exit_mmap(mm);
bad_elf_cleanup_pgdir:
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
bad_mm:
    goto out;

```

问题:

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，鼓励给出详细设计方案

首先说明硬链接和软链接的区别：

- 软链接就像是windows中的快捷方式，其指向源文件，但是大小和内容跟源文件不同，同时如果删除源文件，快捷方式也就没用了；
- 硬链接则可以认为是对源文件的一份copy，其大小，内容与源文件完全一致。修改源文件，硬链接文件会同步更新，同时删除源文件后，这份copy的文件当然仍然是存在的。本质上，硬链接文件和源文件是由同一个inode 映射过来的。

实现：

硬盘上索引节点的定义：

```

struct sfs_disk_inode {
    uint32_t size; // 如果inode表示常规文件，则size是文件大小
    uint16_t type; // inode的文件类型
    uint16_t nlinks; // 此inode的硬链接数
    uint32_t blocks; // 此inode的数据块数的个数
    uint32_t direct[SFS_NDIRECT]; // 此inode的直接数据块索引值（有SFS_NDIRECT个）
    uint32_t indirect; // 此inode的一级间接数据块索引值
};

```

假设要在磁盘上创建文件 A 的一个链接 B。

- 创建软链接：创建文件 B 的 inode，其中 type 设置为链接，indirect 指向文件 A 的 inode 的地址，如果 type 中没有明确指示是软链接还是硬链接，就还需要额外一位来进行标记。当访问文件

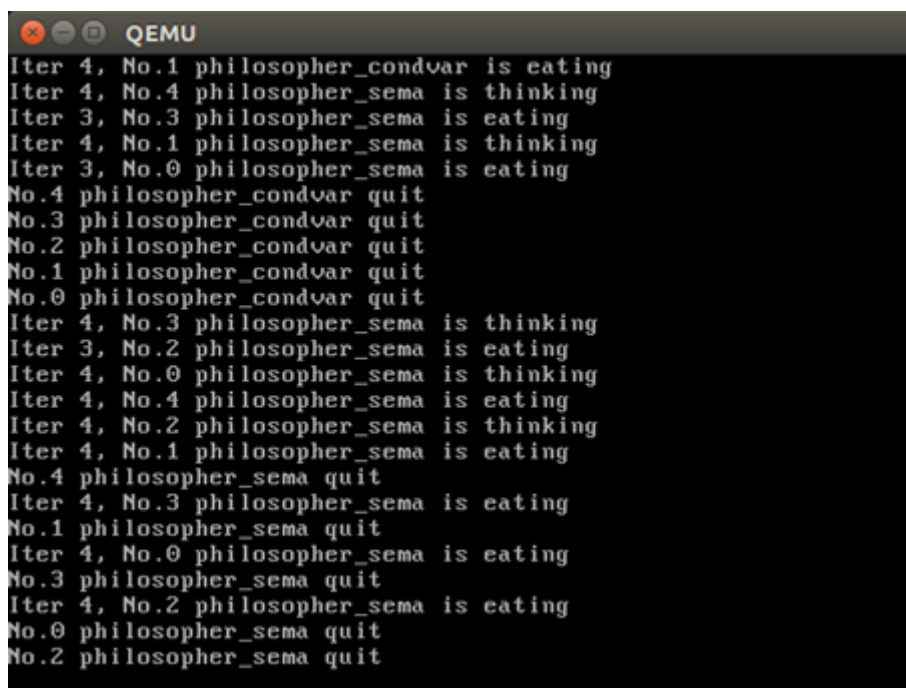
B 的时候，根据 type 为链接，知道实际应该访问的是 indirect 所指向的 inode 对应的文件，也就是文件 A。删除文件 B 时，直接删除其 inode 即可。若是修改文件 A，则不需要对文件 B 进行操作。

- 创建硬链接：文件 B 与文件 A 共享同一个 inode，同时需要将文件 A 的内容全部复制一份放到文件 B 中，且将该 inode 中的硬链接数 nlinks 加1。访问文件 B 就相当于访问 A，因为实际上是在访问同一个 inode。删除文件 B，实际就是删除了这个 inode 到 B 的映射关系，同时还要将 nlinks 减1。若是修改文件 A，则文件 B 也要同步更新，这也是因为操作系统是按照 inode 来进行操作的，所以两个文件都会被修改。

三、实验结果：

这次执行 make qemu 不太顺利，一开始失败后直接尝试执行 lab8_result，没想到答案也运行不起来，然后开始在 lab8 项目中找问题。发现虽然每次 make 都报错，但出错的点都不在本次实验上，后面就基本上是看见 kernel panic 在哪个地方就去把相应的 result 复制过来...一番折腾后，没想到最后一点错误是因为在 lab8 的 default_sched.c 中还是采用的 RR 调度策略，需要把之前在 lab6 中实现的 stride 调度复制过来，最后运行成功。

make qemu:



```
QEMU
Iter 4, No.1 philosopher_condvar is eating
Iter 4, No.4 philosopher_sema is thinking
Iter 3, No.3 philosopher_sema is eating
Iter 4, No.1 philosopher_sema is thinking
Iter 3, No.0 philosopher_sema is eating
No.4 philosopher_condvar quit
No.3 philosopher_condvar quit
No.2 philosopher_condvar quit
No.1 philosopher_condvar quit
No.0 philosopher_condvar quit
Iter 4, No.3 philosopher_sema is thinking
Iter 3, No.2 philosopher_sema is eating
Iter 4, No.0 philosopher_sema is thinking
Iter 4, No.4 philosopher_sema is eating
Iter 4, No.2 philosopher_sema is thinking
Iter 4, No.1 philosopher_sema is eating
No.4 philosopher_sema quit
Iter 4, No.3 philosopher_sema is eating
No.1 philosopher_sema quit
Iter 4, No.0 philosopher_sema is eating
No.3 philosopher_sema quit
Iter 4, No.2 philosopher_sema is eating
No.0 philosopher_sema quit
No.2 philosopher_sema quit
```

执行 ls:

```
QEMU - Press Ctrl-Alt to exit mouse grab
[d] 2(h) 23(b) 5888(s) .
[d] 2(h) 23(b) 5888(s) ..
[-] 1(h) 10(b) 40383(s) softint
[-] 1(h) 11(b) 44571(s) priority
[-] 1(h) 11(b) 44584(s) matrix
[-] 1(h) 10(b) 40391(s) faultreadkernel
[-] 1(h) 10(b) 40381(s) hello
[-] 1(h) 10(b) 40382(s) badarg
[-] 1(h) 10(b) 40404(s) sleep
[-] 1(h) 11(b) 44694(s) sh
[-] 1(h) 10(b) 40380(s) spin
[-] 1(h) 11(b) 44640(s) ls
[-] 1(h) 10(b) 40386(s) badsegment
[-] 1(h) 10(b) 40435(s) forktree
[-] 1(h) 10(b) 40410(s) forktest
[-] 1(h) 10(b) 40516(s) waitkill
[-] 1(h) 10(b) 40404(s) divzero
[-] 1(h) 10(b) 40381(s) pgdir
[-] 1(h) 10(b) 40385(s) sleepkill
[-] 1(h) 10(b) 40408(s) testbss
[-] 1(h) 10(b) 40381(s) yield
[-] 1(h) 10(b) 40406(s) exit
[-] 1(h) 10(b) 40385(s) faultread
lsdir: step 4
$
```

执行 hello :

```
$ hello
Hello world!?.
I am process 14.
hello pass.
$
```

四、实验总结:

这次实验挺困难的, 练习1中编写 sfs_io_nolock 里的代码相对是容易的, 明白了需要分三部分进行文件的读取, 同时需要调用不同的读函数后还是能很清楚地理解这一函数做了什么的, 不过对于练习2, load_icode 中需要补充的内容挺多, 其中不少操作到现在也不是太明白, 还需要进一步学习...

结束本次实验, 首先是认识了文件系统的分层结构, 一个打开文件的函数 open 会调用 syscall 进入到下一层, 然后一步步向下调用, 每一层都定义了相应的函数, 直到最后得到代表该文件的文件描述符 fd 后再一路返回。除此之外, 印象最深的也就是 inode 这个数据结构了吧, 大概知道了操作系统就是靠 inode 找到目标文件和目录的, 因此硬盘中的 inode 就代表了一个实际位于磁盘上的文件, 如果将该文件读入内存, 则需要在内存中为其创建对应的 inode。