实验八：文件系统

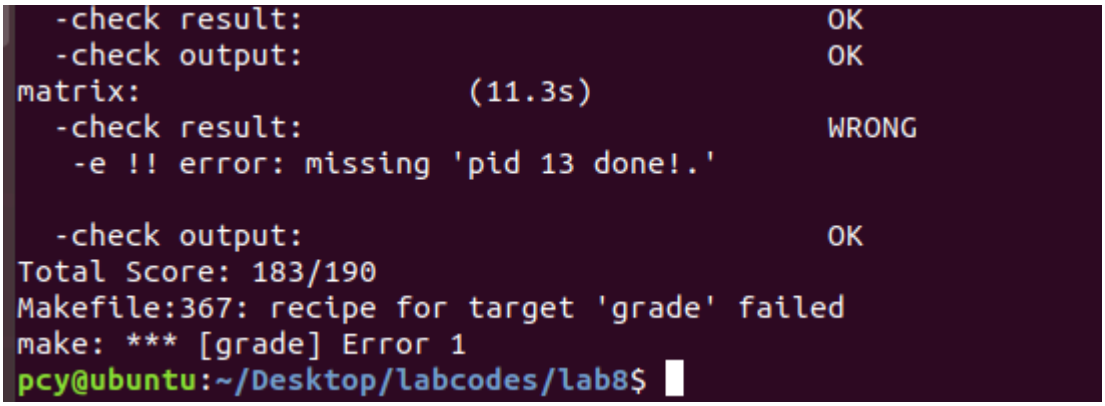
遇到的问题

和之前一样，在lab8_result中make grade之后发现得分是0，参考答案中在init_main()函数的`cprintf("init check memory pass.\n");`之前多了如下两行：

```
    assert(nr_free_pages_store == nr_free_pages());
    assert(kernel_allocated_store == kallocated());
```

经过验证，这两句会导致一些用户进程无法正常输出"init check memory pass."导致result错误。

并且在改了和之前同样的错误，按照步骤做完整个实验之后，仍然没有满分，可能是因为前几次实验的代码在“迁移”的过程中可能产生了错误或者没有完全“迁移”过来。



因为没有找到是哪里合并的有问题，所以重新从网上克隆了一份，把lab8新加的代码贴进去，执行make grade：

```
  -check result:                                OK
  -check output:                                OK
spin:                   (3.0s)
  -check result:                                OK
  -check output:                                OK
waitkill:               (3.4s)
  -check result:                                OK
  -check output:                                OK
forktest:               (2.8s)
  -check result:                                OK
  -check output:                                OK
forktree:               (5.3s)
  -check result:                                OK
  -check output:                                OK
priority:               (15.4s)
  -check result:                                OK
  -check output:                                OK
sleep:                  (11.5s)
  -check result:                                OK
  -check output:                                OK
sleepkill:              (2.8s)
  -check result:                                OK
  -check output:                                OK
matrix:                 (10.6s)
  -check result:                                OK
  -check output:                                OK
Total Score: 190/190
pcy@ubuntu:~/Desktop/labcodes/lab8$
```

实验目的

通过完成本次实验，希望能达到以下目标

- 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，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。在kern_init函数中，多了一个fs_init 函数的调用。fs_init 函数就是文件系统初始化的总控函数，它进一步调用了虚拟文件系统初始化函数 vfs_init，与文件相关的设备初始化函数 dev_init 和 Simple FS 文件系统的初始化函数 sfs_init。这三个初始化函数联合在一起，协同完成了整个虚拟文件系统、SFS文件系统和文件系统对应的设备(键盘、串口、磁盘)的初始化工作。

练习0：填写已有实验

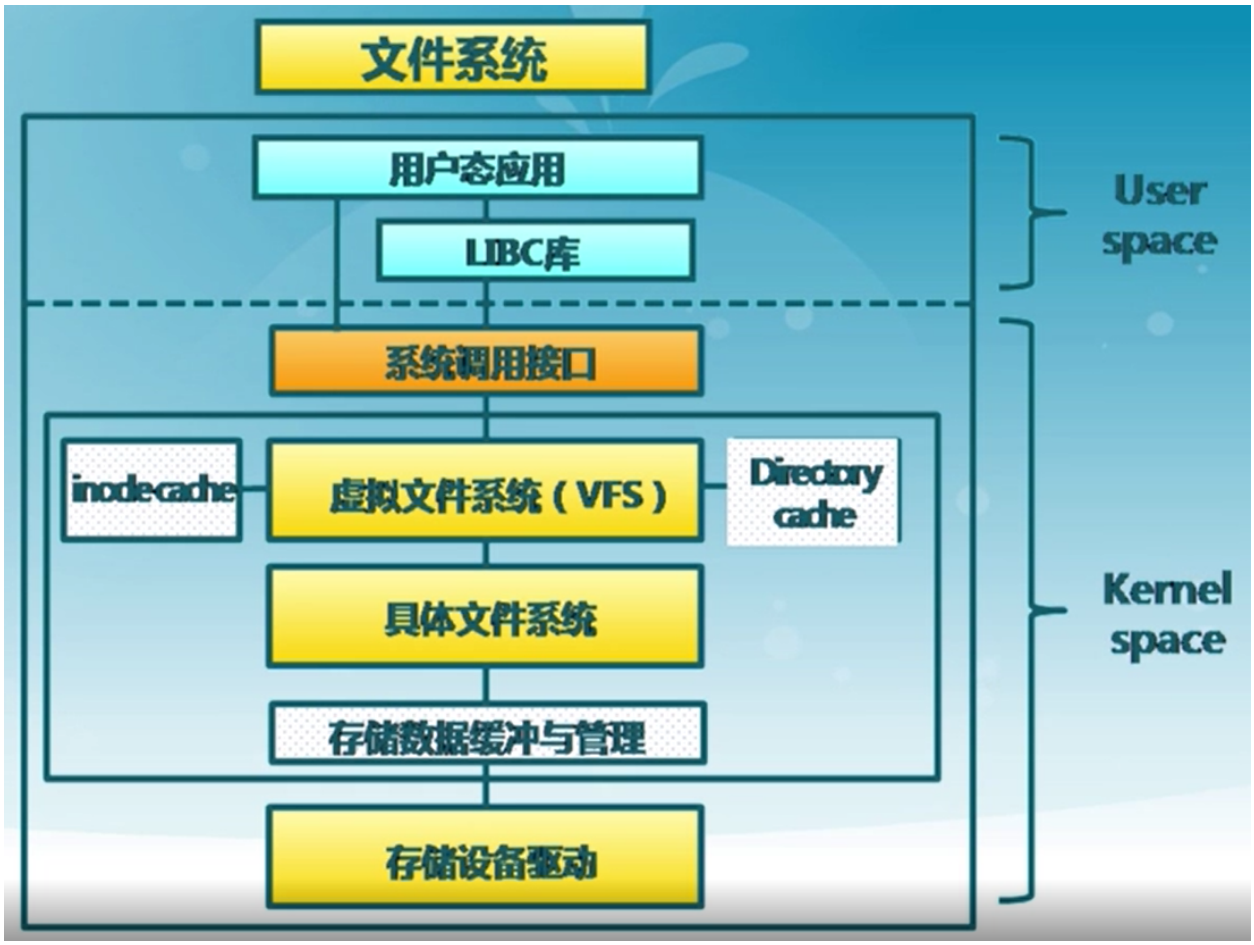
使用meld工具进行合并。需要修改的文件：proc.c、default_pmm.c、pmm.c、swap_fifo.c、vmm.c、trap.c、sche.c、monitor.c、check_sync.c。只需要依次将实验1-7的代码填入本实验中代码。

练习1: 完成读文件操作的实现（需要编码）

首先了解打开文件的处理流程，然后参考本实验后续的文件读写操作的过程分析，编写在sfs_inode.c中sfs_io_nolock读文件中数据的实现代码。

ucore的文件系统架构主要由四部分组成：

1. 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。
2. 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。向下提供一个抽象函数指针列表和数据结构来屏蔽不同文件系统的实现细节。
3. 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
4. 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。



打开一个文件的详细处理的流程： 例如某一个应用程序需要操作文件（增删读写等），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一具体文件系统(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。

一些重要的数据结构如下：

```
struct file {
    enum {
    FD_NONE, FD_INIT, FD_OPENED, FD_CLOSED,
    } status;        //访问文件的执行状态
    bool readable; //文件是否可读
    bool writable; //文件是否可写
    int fd;         //文件在filemap中的索引值
    off_t pos;     //访问文件的当前位置
    struct inode *node;//该文件对应的内存inode指针
    atomic_t open_count;//打开此文件的次数
};
```

`inode`数据结构是位于内存的索引节点，把不同文件系统的特定索引节点信息(甚至不能算是一个索引节点)统一封装起来，避免了进程直接访问具体文件系统.

```
struct inode {
    union { //包含不同文件系统特定inode信息的union域
    struct device __device_info;  //设备文件系统内存inode信息
    struct sfs_inode __sfs_inode_info; //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;  //此inode所属文件系统类型
    atomic_t ref_count;   //此inode的引用计数
    atomic_t open_count;  //打开此inode对应文件的个数
    struct fs *in_fs;     //抽象的文件系统,包含访问文件系统的函数指针
    const struct inode_ops *in_ops;   //抽象的inode操作,包含访问inode的函数指针
};
```

对应到我们的ucore上，具体的过程如下：

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，从而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/testfile"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，所以讲继续跟进看sfs_lookup函数，如下：

```
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 != '/');     //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。
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。

vop_ref_dec(node);
if (ret != 0) {
    return ret;
}
*node_store = subnode;  //当无法分解path后，就意味着找到了需要对应的inode节点，就可顺利返回了。
return 0;
}
```

看到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（例如“/test/file”），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。

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。 如下：

```
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;
}
```

完成 `sfs_io_nolock` 函数中读文件的过程(kern/fs/sfs/sfs_inode.c)

```
  //LAB8:EXERCISE1 这里分为三部分来读取文件，每次通过sfs_bmap_load_nolock函数获取文件索引编号，然
后调用sfs_buf_op完成实际的文件读写操作。
    /*
     * (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
    */
    if ((blkoff = offset % SFS_BLKSIZE) != 0) {//读取第一部分的数据
        //计算第一个数据块的大小
        size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
        //找到内存文件索引对应的block的编号ino
        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;
        }
        //完成实际的读写操作
        alen += size;
        if (nblks == 0) {
            goto out;
        }
        buf += size, blkno ++, nblks --;
    }
    //读取中间部分的数据，将其分为size大学的块，然后一次读一块直至读完
    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;
    }
```

给出设计实现”UNIX的PIPE机制“的概要设计方案

管道可以看作是由内核管理的一个缓冲区，一端连接进程A的输出，另一端连接进程B的输入。进程A会向管道中放入信息，而进程B会取出被放入管道的信息。当管道中没有信息，进程B会等待，直到进程A放入信息。当管道被放满信息的时候，进程A会等待，直到进程B取出信息。当两个进程都结束的时候，管道也自动消失。 管道基于fork机制建立，从而让两个进程可以连接到同一个PIPE上。 最开始的时候，管道两端都连接在同一个进程A上，当fork复制进程A得到进程B的时候，会将这两个连接也复制到新的进程B上。随后，每个进程关闭自己不需要的一个连接。例如进程A关闭向PIPE输入的链接，进程B关闭接收PIPE输出的连接。 从而二者即可通过该PIPE进行通信。 在UNIX中，管道的实现并没有使用专门的数据结构，而是借助了文件系统的file结构和VFS的索引节点inode。通过将两个 file 结构指向同一个临时的 VFS 索引节点，而这个 VFS 索引节点又指向一个物理页面而实现的。具体来说，

即有两个 file 数据结构，但它们定义文件操作例程地址是不同的，其中一个是向管道中写入数据的例程地址，而另一个是从管道中读出数据的例程地址。这样，用户程序的系统调用仍然是通常的文件操作，而内核却利用这种抽象机制实现了管道这一特殊操作。 当写进程向管道中写入时，它利用标准的库函数write()，系统根据库函数传递的文件描述符，可找到该文件的 file 结构。file 结构中指定了用来进行写操作的函数（即写入函数）地址，于是，内核调用该函数完成写操作。 写入函数在向内存中写入数据之前，必须首先检查 VFS 索引节点中的信息，同时满足如下条件时，才能进行实际的内存复制工作：

1. 内存中有足够的空间可容纳所有要写入的数据；
2. 内存没有被读程序锁定。

如果同时满足上述条件，写入函数首先锁定内存，然后从写进程的地址空间中复制数据到内存。否则，写入进程就休眠在 VFS 索引节点的等待队列中，接下来，内核将调用调度程序，而调度程序会选择其他进程运行。写入进程实际处于可中断的等待状态，当内存中有足够的空间可以容纳写入数据，或内存被解锁时，读取进程会唤醒写入进程，这时，写入进程将接收到信号。当数据写入内存之后，内存被解锁，而所有休眠在索引节点的读取进程会被唤醒。 管道的读取过程和写入过程类似。但是，进程可以在没有数据或内存被锁定时立即返回错误信息，而不是阻塞该进程，这依赖于文件或管道的打开模式。反之，进程可以休眠在索引节点的等待队列中等待写入进程写入数据。当所有的进程完成了管道操作之后，管道的索引节点被丢弃，而共享数据页也被释放。

练习2: 完成基于文件系统的执行程序机制的实现（需要编码）

改写proc.c中的load_icode函数和其他相关函数，实现基于文件系统的执行程序机制。

根据注释需要先初始化fs中的进程控制结构，即在`alloc_proc`函数的最后加上一句`proc->filesp = NULL;`完成初始化。

load_icode主要是将文件加载到内存中执行，根据注释的提示分为了七个步骤：

1. 建立内存管理器
2. 建立页目录
3. 将文件逐个段加载到内存中，这里要注意设置虚拟地址与物理地址之间的映射
4. 建立相应的虚拟内存映射表
5. 建立并初始化用户堆栈
6. 处理用户栈中传入的参数
7. 最后很关键的一步是设置用户进程的中断帧

此外，一旦发生错误还需要进行错误处理。

```
static int
load_icode(int fd, int argc, char **kargv) {
    assert(argc >= 0 && argc <= EXEC_MAX_ARG_NUM);
    //(1)建立内存管理器
    if (current->mm != NULL) {
        panic("load_icode: current->mm must be empty.\n");
    }
    //(2)建立页目录
    int ret = -E_NO_MEM;// E_NO_MEM代表因为存储设备产生的请求错误
    struct mm_struct *mm;// 建立内存管理器
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }
```

```
    struct Page *page;
    //(3)从文件加载程序到内存
    struct elfhdr __elf, *elf = &__elf;
    if ((ret = load_icode_read(fd, elf, sizeof(struct elfhdr), 0)) != 0) {
        goto bad_elf_cleanup_pgdir;
    }
    if (elf->e_magic != ELF_MAGIC) {
        ret = -E_INVAL_ELF;
        goto bad_elf_cleanup_pgdir;
    }
    struct proghdr __ph, *ph = &__ph;
    uint32_t vm_flags, perm, phnum;
    for (phnum = 0; phnum < elf->e_phnum; phnum ++) {
        off_t phoff = elf->e_phoff + sizeof(struct proghdr) * phnum;//循环读取程序的每个段
的头部
        if ((ret = load_icode_read(fd, ph, sizeof(struct proghdr), phoff)) != 0) {
            goto bad_cleanup_mmap;
        }
        if (ph->p_type != ELF_PT_LOAD) {
            continue ;
        }
        if (ph->p_filesz > ph->p_memsz) {
            ret = -E_INVAL_ELF;
            goto bad_cleanup_mmap;
        }
        if (ph->p_filesz == 0) {
            continue ;
        }
        vm_flags = 0, perm = PTE_U;
        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;
        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;
        //复制数据段和代码段
        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;
            }
            if ((ret = load_icode_read(fd, page2kva(page) + off, size, offset)) != 0) {
                goto bad_cleanup_mmap;
```

```
            }
            start += size, offset += size;
        }
        //建立BSS段
        end = ph->p_va + ph->p_memsz;
        if (start < la) {
            /* ph->p_memsz == ph->p_filesz */
            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));
        }
        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(page2kva(page) + off, 0, size);
            start += size;
        }
    }
    sysfile_close(fd);
    //(4)建立相应的虚拟内存映射表
    vm_flags = VM_READ | VM_WRITE | VM_STACK;//建立虚拟地址与物理地址之间的映射
    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);
    //(5)设置用户栈
    mm_count_inc(mm);
    current->mm = mm;
    current->cr3 = PADDR(mm->pgdir);
    lcr3(PADDR(mm->pgdir));
    //(6)处理用户栈中传入的参数，其中argc对应参数个数，uargv[]对应参数的具体内容的地址
    uint32_t argv_size=0, i;
    for (i = 0; i < argc; i ++) {
        argv_size += strnlen(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;
    for (i = 0; i < argc; i ++) {
        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);
    *(int *)stacktop = argc;
    //(7)设置进程的中断帧
    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)错误处理部分
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;
}
```

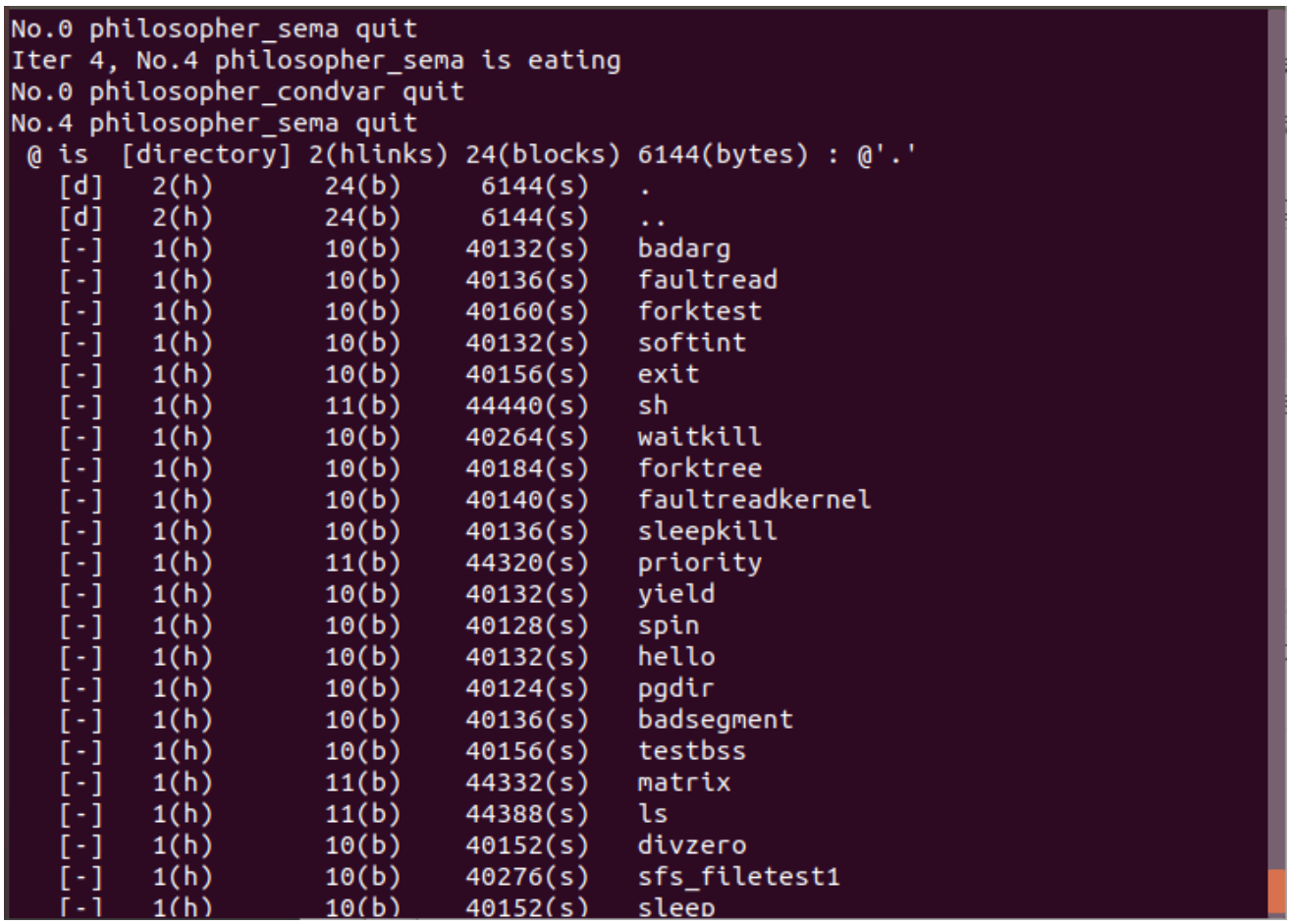
执行：make qemu。如果能看到sh用户程序的执行界面，则基本成功了。如果在sh用户界面上可以执行"ls","hello"等其他放置在sfs文件系统中的其他执行程序，则可以认为本实验基本成功。

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"的概要设方案，鼓励给出详细设计方案

硬链接机制的设计实现： vfs中预留了硬链接的实现接口 `int vfs_link(char *old_path, char *new_path);` 。在实现硬链接机制，创建硬链接link时，为new_path创建对应的file，并把其inode指向old_path所对应的inode，inode的引用计数加1。在unlink时将引用计数减去1即可。

软链接机制的设计实现： vfs中预留了软链接的实现接口 `int vfs_symlink(char *old_path, char *new_path);` 。在实现软链接机制，创建软连接link时，创建一个新的文件（inode不同），并把old_path的内容存放到文件的内容中去，给该文件保存在磁盘上时disk_inode类型为SFS_TYPE_LINK，再完善对于该类型inode的操作即可。unlink时类似于删除一个普通的文件。

实验结果：



参考:

https://github.com/chyyuu/ucore_os_lab/pull/58/commits/852d7074ff9109ffe0b3956cd5292d60fa139a7d

https://www.jianshu.com/p/0b15c4cdead9