Lab 8文件系统

实验目的

通过完成本次实验，希望能够达到以下目标

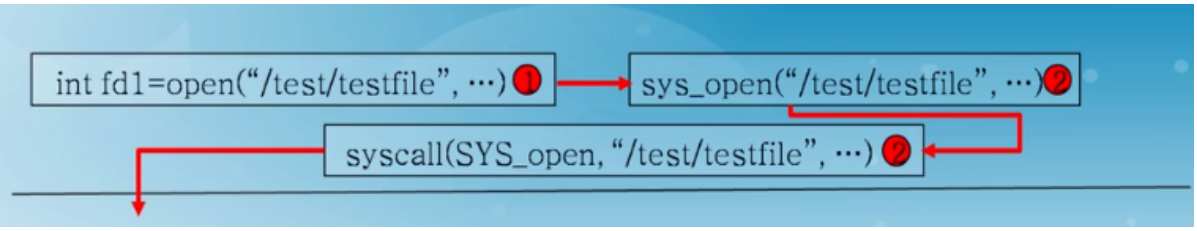
- 了解文件系统抽象层-VFS的设计与实现
- 了解基于索引节点组织方式的Simple FS文件系统与操作的设计与实现
- 了解“一切皆为文件”思想的设备文件设计
- 了解简单系统终端的实现

练习

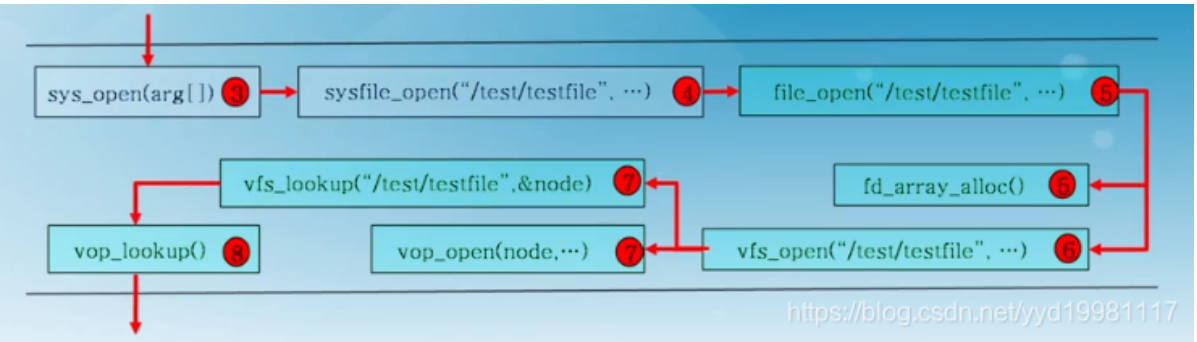
练习1: 完成读文件操作的实现（需要编码）

> 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，填写在kern/fs/sfs/sfs_inode.c中 的sfs_io_nolock()函数，实现读文件中数据的代码。

系统调用过程



当用户要打开文件时，会依次调用reopen() —>open()—>sys_open()—>syscall()，随后引发系统调用后进入内核态，然后会由sys_open内核函数处理系统调用，到了内核态后，通过中断处理例程，会调用到sys_open内核函数，并进一步调用sysfile_open内核函数。到了这里，需要把位于用户空间的字符串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。



文件系统抽象层（VFS）的处理流程：

1. 分配一个空闲的file数据结构变量file在文件系统抽象层的处理中，首先调用的是file_open函数，它要给这个即将打开的文件分配一个file数据结构的变量，这个变量其实是当前进程的打开文件数组current->fs_struct->filemap[]中的一个空闲元素（即还没用于一个打开的文件），而这个元素的索引值就是最终要返回到用户进程并赋值给变量fd1。到了这一步还仅仅是给当前用户进程分配了一个file数据结构的变量，还没有找到对应的文件索引节点。

   为此需要进一步调用vfs_open函数来找到path指出的文件所对应的基于inode数据结构的VFS索引节点node。vfs_open函数需要完成两件事情：通过vfs_lookup找到path对应文件的inode；调用vop_open函数打开文件。

2. 找到文件设备的根目录的索引节点需要注意，这里的vfs_lookup函数是一个针对目录的操作函数，它会调用vop_lookup函数来找到SFS文件系统中的/test目录下的testfile文件。为此，vfs_lookup函数首先调用get_device函数，并进一步调用vfs_get_bootfs函数（其实调用了）来找到根目录对应的inode。这个inode就是位于vfs.c中的inode变量bootfs_node。这个变量在init_main函数（位于kern/process/proc.c）执行时获得了赋值。

   找到根目录下的test子目录对应的索引节点，在找到根目录对应的inode后，通过调用vop_lookup函数来查找/和test这两层目录下的文件testfile所对应的索引节点，如果找到就返回此索引节点。

3. 把file和node建立联系。完成第3步后，将返回到file_open函数中，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/test/testfile文件的索引节点node。这时返回fd。经过重重回退，通过系统调用返回，用户态的syscall->sys_open->open->safe_open等用户函数的层层函数返回，最终把把fd赋值给fd1。自此完成了打开文件操作。

sfs调用

在sfs_inode.c中的sfs_node_dirops变量定义了".vop_lookup = sfs_lookup"，所以我们重点分析sfs_lookup的实现。

在sfs_lookup函数中以"/"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。

最后在读取文件或写文件时，通过sfs_write/sfs_read函数，调用sfs_io->sfs_io_nolock函数，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。

sfs_io_nolock函数主要用来将磁盘中的一段数据读入到内存中或者将内存中的一段数据写入磁盘。该函数会进行一系列的边缘检查，检查访问是否越界、是否合法。之后将具体的读/写操作使用函数指针统一起来，统一成针对整块的操作。然后完成不落在整块数据块上的读/写操作，以及落在整块数据块上的读写。

```
static int
sfs_io_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, void *buf, off_t offset,
size_t *alenp, bool write)
{   //创建一个磁盘索引节点指向要访问文件的内存索引节点
    struct sfs_disk_inode *din = sin->din;
    assert(din->type != SFS_TYPE_DIR);
    //确定读取的结束位置
    off_t endpos = offset + *alenp, blkoff;
    *alenp = 0;
```

```
    // 进行一系列的边缘，避免非法访问
    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);
    //确定是读操作还是写操作，并确定相应的系统函数
    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
// 判断被需要操作的区域的数据块中的第一块是否是完全被覆盖的，
// 如果不是，则需要调用非整块数据块进行读或写的函数来完成相应操作
      if ((blkoff = offset % SFS_BLKSIZE) != 0)  {
        // 第一块数据块中进行操作的偏移量
          blkoff = offset % SFS_BLKSIZE;
        // 第一块数据块中进行操作的数据长度
        size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
       //获取这些数据块对应到磁盘上的数据块的编号
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        //对数据块进行读或写操作
        if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0) {
            goto out;
        }
        //已经完成读写的数据长度
        alen += size;
        if (nblks == 0) {
            goto out;
        }
```

```
        buf += size, blkno++; nblks--;
    }

    读取中间部分的数据，将其分为大小为size的块，然后一块一块操作，直至完成
    size = SFS_BLKSIZE;
    while (nblks != 0)
    {
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        //对数据块进行读或写操作
        if ((ret = sfs_block_op(sfs, buf, ino, 1)) != 0) {
            goto out;
        }
        //更新相应的变量
        alen += size, buf += size, blkno++, nblks--;
    }

    // 最后一页，可能出现不对齐的现象:
    if ((size = endpos % SFS_BLKSIZE) != 0)
    {
         // 获取该数据块对应到磁盘上的数据块的编号
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        // 进行非整块的读或者写操作
        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;
}
```

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。执行：make qemu。如果能看看到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。

需要补充修改的函数有alloc_proc，load_icode和do_fork。

alloc_proc:

```
proc->filesp = NULL;      //初始化fs中的进程控制结构
```

load_icode:

原先lab5源码中，读取可执行文件是直接读取内存的，但在这里需要使用函数`load_icode_read`来从文件系统中读取`ELF header`以及各个段的数据。

原先Lab5的`load_icode`函数中并没有对`execve`所执行的程序传入参数，而我们需要在lab8中补充这个实现。

```
    //(3.1) 读取文件中的原始数据内容并解析elfhdr
    load_icode_read(fd, (void *)elf, sizeof(struct elfhdr), 0);
    //(3.2) 读取文件中的原始数据内容并根据elfhdr中的信息解析proghdr
    load_icode_read(fd, (void *)ph, sizeof(struct proghdr), elf->e_phoff);
```

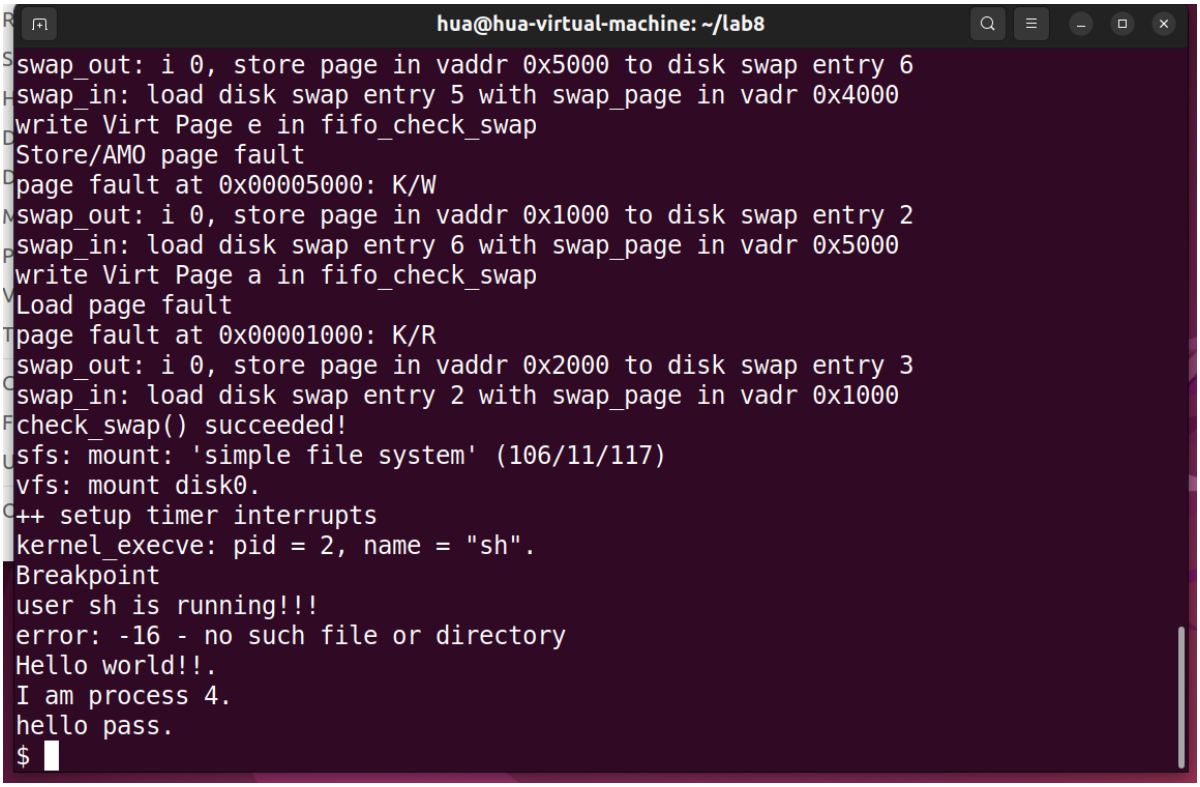
do_fork:

fork机制在原先lab5的基础上，多了file_struct结构的复制操作与执行失败时的重置操作。

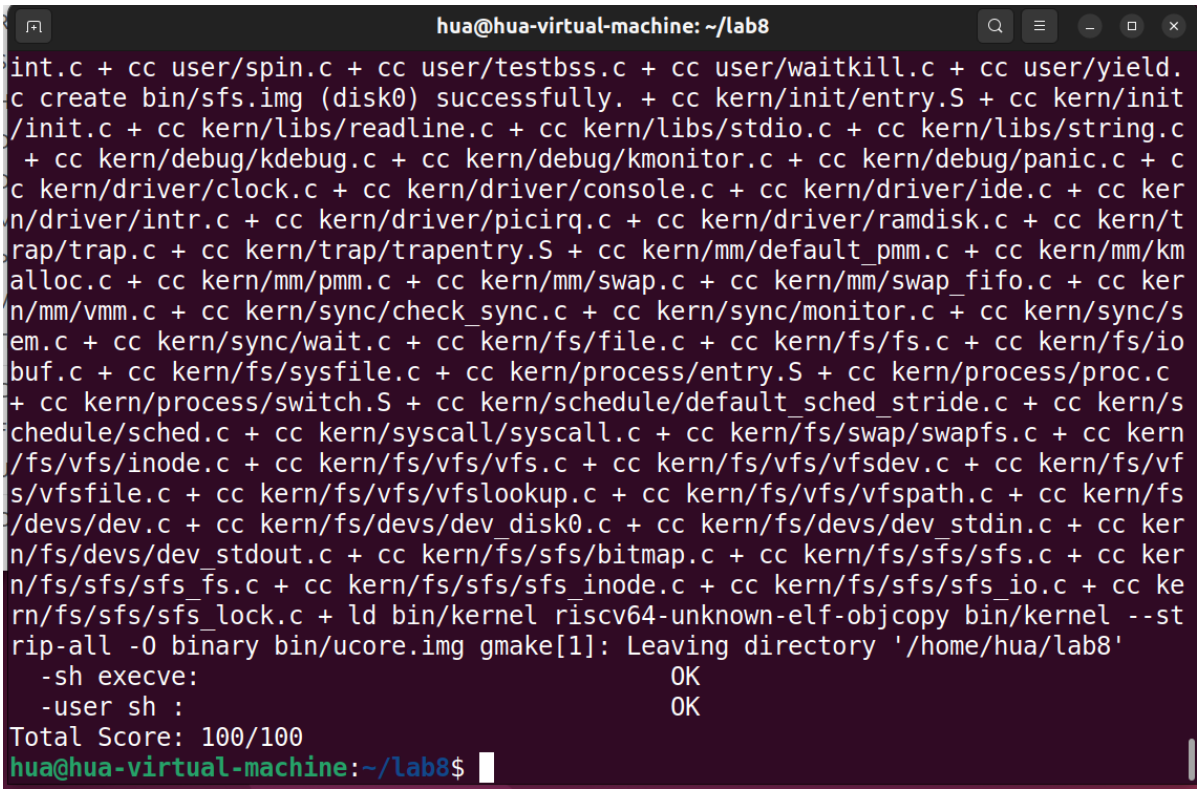
```
// Lab8:调用copy_files，将files_struct复制到proc
    if (copy_files(clone_flags, proc) != 0)
        goto bad_fork_cleanup_kstack;
```

实验结果

在make qemu中我们可以执行user文件夹下包含的一些简单指令：



make grade


hua@hua-virtual-machine: ~/lab8
int.c + cc user/spin.c + cc user/testbss.c + cc user/waitkill.c + cc user/yield.
c create bin/sfs.img (disk0) successfully. + cc kern/init/entry.S + cc kern/init
/init.c + cc kern/libs/readline.c + cc kern/libs/stdio.c + cc kern/libs/string.c
+ cc kern/debug/kdebug.c + cc kern/debug/kmonitor.c + cc kern/debug/panic.c + c
c kern/driver/clock.c + cc kern/driver/console.c + cc kern/driver/ide.c + cc ker
n/driver/intr.c + cc kern/driver/picirq.c + cc kern/driver/ramdisk.c + cc kern/t
rap/trap.c + cc kern/trap/trapentry.S + cc kern/mm/default_pmm.c + cc kern/mm/km
alloc.c + cc kern/mm/pmm.c + cc kern/mm/swap.c + cc kern/mm/swap_fifo.c + cc ker
n/mm/vmm.c + cc kern/sync/check_sync.c + cc kern/sync/monitor.c + cc kern/sync/s
em.c + cc kern/sync/wait.c + cc kern/fs/file.c + cc kern/fs/fs.c + cc kern/fs/io
buf.c + cc kern/fs/sysfile.c + cc kern/process/entry.S + cc kern/process/proc.c
+ cc kern/process/switch.S + cc kern/schedule/default_sched_stride.c + cc kern/s
chedule/sched.c + cc kern/syscall/syscall.c + cc kern/fs/swap/swapfs.c + cc kern
/fs/vfs/inode.c + cc kern/fs/vfs/vfs.c + cc kern/fs/vfs/vfsdev.c + cc kern/fs/vf
s/vfsfile.c + cc kern/fs/vfs/vfslookup.c + cc kern/fs/vfs/vfspath.c + cc kern/fs
/devs/dev.c + cc kern/fs/devs/dev_disk0.c + cc kern/fs/devs/dev_stdin.c + cc ker
n/fs/devs/dev_stdout.c + cc kern/fs/sfs/bitmap.c + cc kern/fs/sfs/sfs.c + cc ker
n/fs/sfs/sfs_fs.c + cc kern/fs/sfs/sfs_inode.c + cc kern/fs/sfs/sfs_io.c + cc ke
rn/fs/sfs/sfs_lock.c + ld bin/kernel riscv64-unknown-elf-objcopy bin/kernel --st
rip-all -O binary bin/ucore.img gmake[1]: Leaving directory '/home/hua/lab8'
-sh execve: OK
-user sh : OK
Total Score: 100/100
hua@hua-virtual-machine:~/lab8$