实验八：文件系统

计31 张正 2013011418

一、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；
- 了解文件系统抽象层-VFS的设计与实现；

二、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。

练习0：填写已有实验

本实验依赖实验1/2/3/4/5/6/7。请把你做的实验1/2/3/4/5/6/7的代码填入本实验中代码中有“LAB1”/“LAB2”/“LAB3”/“LAB4”/“LAB5”/“LAB6” /“LAB7”的注释相应部分。并确保编译通过。注意：为了能够正确执行lab8的测试应用程序，可能需对已完成的实验1/2/3/4/5/6/7的代码进行进一步改进。

将第七次代码拷贝到lab8

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。

```
if ((blkoff = offset % SFS_BLKSIZE) != 0){
        size = (nblks != 0) ? (SFS_BLKSIZE - blkoff): (endpos - offset);

        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno , &ino)) != 0)
            goto out;
        //sfs_bmap_load_nolock(struct sfs_fs *sfs, struct sfs_inode *sin,
uint32_t index, uint32_t *ino_store) {
        if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff))  != 0)
            goto out;
        //    int (*sfs_buf_op)(struct sfs_fs *sfs, void *buf, size_t len, uint32_t
blkno, off_t offset);
        alen += size;
        if (nblks == 0) goto out;
        buf += size;
        blkno ++;
```

```
        nblks --;
    }

    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--;
    }

    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;
    }
```

先处理起始的没有对齐到块的部分，再以块为单位循环处理中间的部分，最后处理末尾剩余的部分。每部分中都调用sfs_bmap_load_nolock函数得到blkno对应的inode编号，并调用sfs_rbuf或sfs_rblock函数读取数据（中间部分调用sfs_rblock，起始和末尾部分调用sfs_rbuf），调整相关变量。

- 给出设计实现”UNIX的PIPE机制“的概要设方案，鼓励给出详细设计方案

设计一个pipe用的pipefs，在系统调用时创建两个file，一个只读，一个只写，并使这两个file连接到同一个inode上。这样便简单实现了pipe机制，使用一个临时保存输出的文件，当程序1输出时，将内存保存到这个文件；当程序2读入时，读取这个文件的内容即可。

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。

1、实验流程：

①建立内存管理器
②建立页目录表
③从硬盘上读取程序内容到内存
④建立相应的虚拟内存映射表
⑤设置好用户栈
⑥设置进程的中断帧
修改初始化fs中的进程控制结构

```
        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;
```

load_icode实现

```
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");
    }
```

```
    int ret = -E_NO_MEM;
    struct mm_struct *mm;
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }
//(2)建立页目录表
    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;
  //(3.1)读取elf文件头
    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;
   //(3.2)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 ;
        }
    //(3.3)设置好虚拟地址与物理地址之间的映射
        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;
    //(3.4)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;
        }
    //(3.5)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;
        }
    }
    //关闭文件，加载程序结束
    sysfile_close(fd);
  //(4)建立相应的虚拟内存映射表
    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);
    //(5)设置好用户栈
        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
        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
    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;
}
```

- 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

创建硬链接时，需要操作系统产生一个新的inode，其内容和之前的inode相同并增加引用计数。创建软连接时，相当于一个快捷方式，只需要将它设置为一个指针，指向原来的文件即可。