uCore实验八报告

黄郭斌 计科1502班 201507010206

一. 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。

二. 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；

三. 实验分析及结果

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。可以看到，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

需要修改的文件罗列如下：

```
proc.c
default_pmm.c
pmm.c
swap_fifo.c
vmm.c
trap.c
sche.c
monitor.
check_sync.c123456789
```

同样只需要依次将实验1-7的代码填入本实验中代码即可，无需进行其他的修改。

练习1 完成读文件操作的实现

要求是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。 根据实验指导书，我们可以了解到，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。
- `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。

接下来分析下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。

在介绍ucore中打开文件的具体流程之前，先简单分析下一些重要的数据结构，如下： 首先是`file`数据结构：

```
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
};1234567891011
```

接下来`inode`数据结构，它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
};1234567891011121314
```

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;

  }12345678910111213141516171819
  ```

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;
}1234567891011121314
```

这样我们就大概了解一个文件操作的具体流程，接下来我们需要完成`sfs_io_nolock`函数中读文件的过程，代码如下，这里只将我们所需要填写的部分罗列出来了：

```
static int
sfs_io_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, void *buf, off_t offset, size_t *alenp,
bool write) {
......
......

if ((blkoff = offset % SFS_BLKSIZE) != 0) {                    //读取第一部分的数据
    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) {   //找到内存文件索引对应的
block的编号ino
        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;
}
......
......12345678910111213141516171819202122232425262728293031323334353637383940414243444 5
```

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。

练习2 完成基于文件系统的执行程序机制的实现

实验要求改写`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。 在`proc.c`中，根据注释我们需要先初始化fs中的进程控制结构，即在`alloc_proc`函数中我们需要做一下修改，加上一句`proc->filesp = NULL;`从而完成初始化。 修改之后`alloc_proc`函数如下：

```
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;     //初始化fs中的进程控制结构
    }
    return proc;
}  12345678910111213141516171819202122232425262728
```

然后就是要实现`load_icode`函数，具体的实现及注释如下所示：

```
static int
load_icode(int fd, int argc, char **kargv) {

    /* (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.
     */
    assert(argc >= 0 && argc <= EXEC_MAX_ARG_NUM);
    //(1)建立内存管理器
    if (current->mm != NULL) {    //要求当前内存管理器为空
        panic("load_icode: current->mm must be empty.\n");
    }

    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;
    }

    //(2)建立页目录
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }
    struct Page *page;//建立页表

    //(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) {//读取elf文件头
        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 ++) {  //e_phnum代表程序段入口地址数目，即多少各段
        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;//建立虚拟地址与物理地址之间的映射
        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;      //计算数据段和代码段终止地址
        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;
```

```
            }
            //每次读取size大小的块，直至全部读完
            if ((ret = load_icode_read(fd, page2kva(page) + off, size, offset)) != 0) {
//load_icode_read通过sysfile_read函数实现文件读取
                goto bad_cleanup_mmap;
            }
            start += size, offset += size;
        }
        //建立BSS段
        end = ph->p_va + ph->p_memsz;   //同样计算终止地址

        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));
        }

        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;
            }
            //每次操作size大小的块
            memset(page2kva(page) + off, 0, size);
            start += size;
        }
    }
    sysfile_close(fd);//关闭文件，加载程序结束

    //(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);
```

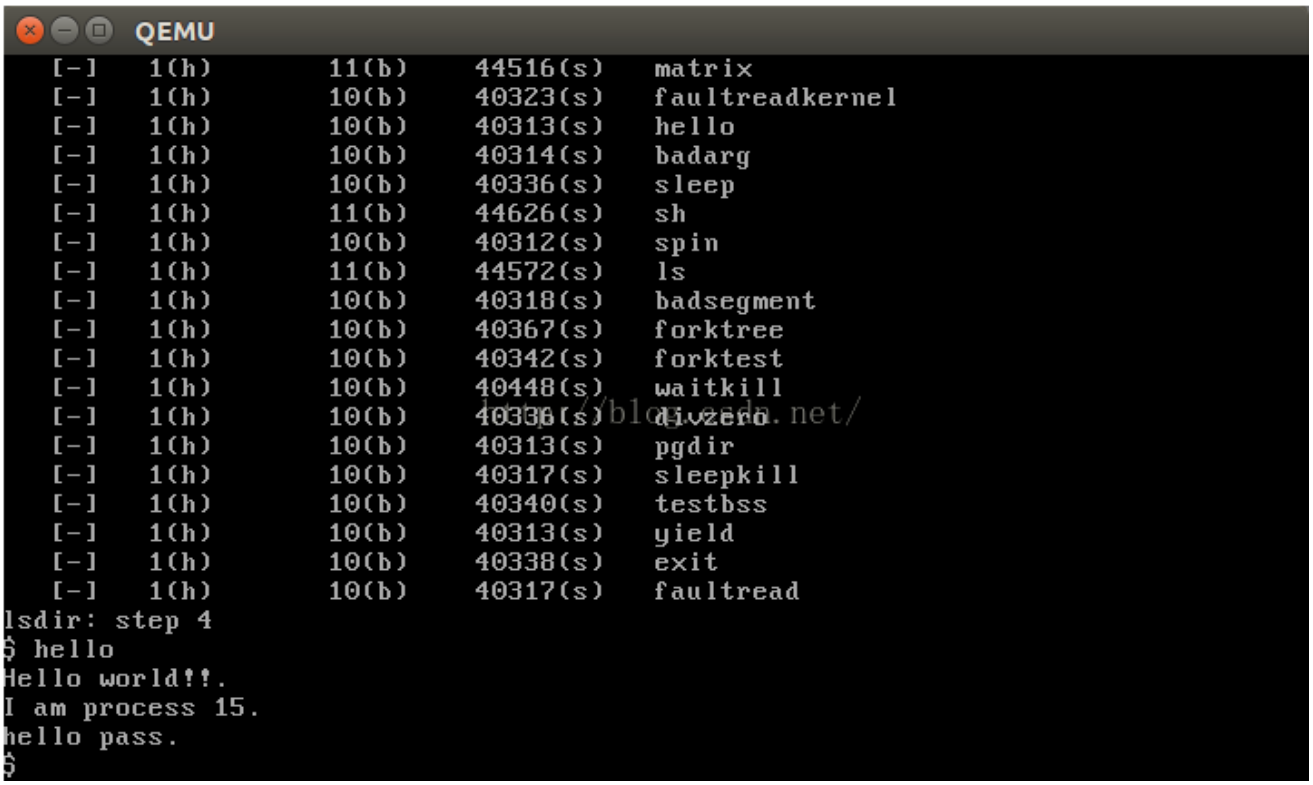
```
    lcr3(PADDR(mm->pgdir));

    //(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 ++) {         //将所有参数取出来放置uargv
        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);   //计算当前用户栈顶
    *(int *)stacktop = argc;
    //(7)设置进程的中断帧
    struct trapframe *tf = current->tf;
    memset(tf, 0, sizeof(struct trapframe));//初始化tf，设置中断帧
    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;           //返回
bad_cleanup_mmap:
    exit_mmap(mm);
bad_elf_cleanup_pgdir:
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
bad_mm:
    goto out;
}
```

详细的注释已经在代码中标注出来了。 `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一共七个步骤：

- 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

当然一旦发生错误还需要进行错误处理。

3. 运行结果



四. 实验体会

本次实验通过阅读从用户态追踪到内核态的关于打开一个文件的代码，从而让我了解了文件系统关于如何将物理层的硬盘资料映射到用户可见的文件系统接口，也就是目录和逻辑文件的视图。

第二部分通过改写之前写的程序，实现了将程序从硬盘中，通过IO系统和文件系统讲程序成功加载值内存中，并使用之前所学的构造进程的方法，实现了完整的程序的加载和运行。