lab8 文件系统

- 课程网址：https://chyyuu.gitbooks.io/ucore_os_docs/content/lab8.html

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，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。

对实验报告的要求：

- 基于markdown格式来完成，以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 完成实验后，请分析ucore_lab中提供的参考答案，并请在实验报告中说明你的实现与参考答案的区别
- 列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）
- 列出你认为OS原理中很重要，但在实验中没有对应上的知识点

练习0

填写已有实验

根据实验要求，我们需要对部分代码进行改进，进一步比对发现，无需改进代码实现，直接使用即可。

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。
>
> 请在实验报告中给出设计实现”UNIX的PIPE机制“的概要设方案，鼓励给出详细设计方案

实现函数sfs_io_nolock

- 实现思路：

```
*（1）如果偏移量与第一个块不对齐，则从偏移量到第一个块的末尾对某些内容进行Rd/Wr
*注意：有用的函数：sfs\u bmap\u load\u nolock，sfs\u buf\u op
*Rd/Wr尺寸=（nblks！= 0）？（SFS\u BLKSIZE-blkoff）：（endpos-偏移）
*（2）Rd/Wr对齐块
*注意：有用的函数：sfs\u bmap\u load\u nolock，sfs\u block\u op
*（3）如果结束位置与最后一个块不对齐，则从开始到最后一个块的（结束位置%SFS\u BLKSIZE）Rd/Wr一些内容
*注意：有用的函数：sfs\u bmap\u load\u nolock，sfs\u buf\u op
```

- 代码实现：

```
/*
 * sfs_io_nolock - Rd/Wr a file contentfrom offset position to offset+ length
disk blocks<-->buffer (in memroy)
 * @sfs:      sfs file system
 * @sin:      sfs inode in memory
 * @buf:      the buffer Rd/Wr
 * @offset:   the offset of file
 * @alenp:    the length need to read (is a pointer). and will RETURN the really
Rd/Wr lenght
 * @write:    BOOL, 0 read, 1 write
 */
static int
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;
    assert(din->type != SFS_TYPE_DIR);
    off_t endpos = offset + *alenp, blkoff;
    *alenp = 0;
    // calculate the Rd/Wr end position
    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);
```

```
    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

  //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
    */
    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;
        }
        if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0) {
            goto out;
        }
        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;
    }
out:
    *alenp = alen;
    if (offset + alen > sin->din->size) {
        sin->din->size = offset + alen;
        sin->dirty = 1;
    }
    return ret;
}
```

(2) 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。

```
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
```

blkno 就是文件开始块的位置，nblks 是文件的大小

回答问题

> 请在实验报告中给出设计实现”UNIX的PIPE机制“的概要设方案，鼓励给出详细设计方案。

为了实现 UNIX 的 PIPE 机制，可以考虑在磁盘上保留一部分空间或者是一个特定的文件来作为 pipe 机制的缓冲区，接下来将说明如何完成对 pipe 机制的支持：

- 当某两个进程之间要求建立管道，假定将进程 A 的标准输出作为进程B的标准输入，那么可以在这两个进程的进程控制块上新增变量来记录进程的这种属性；并且同时生成一个临时的文件，并将其在进程A, B中打开;
- 当进程 A 使用标准输出进行 write 系统调用的时候，通过PCB中的变量可以知道，需要将这些标准输出的数据输出到先前提高的临时文件中去；
- 当进程 B 使用标准输入的时候进行 read 系统调用的时候，根据其PCB中的信息可以知道，需要从上述的临时文件中读取数据；
- 至此完成了对 pipe 机制的设计；

事实上，由于在真实的文件系统和用户之间还由一层虚拟文件系统，因此我们也可以不把数据缓冲在磁盘上，而是直接保存在内存中，然后完成一个根据虚拟文件系统的规范完成一个虚拟的 pipe 文件，然后进行输入输出的时候只要对这个文件进行操作即可；

练习2

> 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。
>
> 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

可以在 Lab 7 的基础上进行修改，读 elf 文件变成从磁盘上读，而不是直接在内存中读。

1.实现思路

load_icode 主要是将文件加载到内存中执行，从上面的注释可知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
- 8、发生错误还需要进行错误处理。

2.代码实现

```
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
    // 判断当前进程的 mm 是否已经被释放掉了
    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) {// 为进程创建一个新的 mm
        goto bad_mm;
    }

    //(2)建立页目录
    if (setup_pgdir(mm) != 0) {// 进行页表项的设置
        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;
    // 从磁盘上读取出 ELF 可执行文件的 elf-header
    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) {// 判断该 ELF 文件是否合法
        ret = -E_INVAL_ELF;
        goto bad_elf_cleanup_pgdir;
    }

    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
    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) {// 读取program header
            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;// 根据 ELF 文件中的信息，对各个段的权限进行设置
        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) {// 将这些段的虚拟内存地址设置为合法的
            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) {// 为 TEXT/DATA 段逐页分配物理内存空间
                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，将磁盘上的 TEXT/DATA 段读入到分配好的内存空间中去
                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) {// 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，因此进行清零初始化
            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) {// 如果 BSS 段还需要更多的内存空间的话，进一步进行分配
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {// 为
BSS 段分配新的物理内存页
                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);// 将分配到的空间清零初始化
            start += size;
        }
    }
    // 关闭传入的文件，因为在之后的操作中已经不需要读文件了
    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;// 设置用户栈的权限
    if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL)) !=
0) {// 将用户栈所在的虚拟内存区域设置为合法的
        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);// 切换到用户的内存空间，这样的话后文中在栈上设置参数部分的操作将大大简
化，因为具体因为空间不足而导致的分配物理页的操作已经交由page fault处理了，是完全透明的
    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 ++) {// 先算出所有参数加起来的长度
        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;// 设置中断帧
    memset(tf, 0, sizeof(struct trapframe));//初始化 tf，设置中断帧
    tf->tf_cs = USER_CS;// 需要返回到用户态，因此使用用户态的数据段和代码段的选择子
    tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
    tf->tf_esp = stacktop;// 栈顶位置为先前计算过的栈顶位置，注意在C语言的函数调用规范中，栈顶指针指向的位置应该是返回地址而不是第一个参数，这里让栈顶指针指向了第一个参数的原因在于，在中断返回之后，会跳转到ELF可执行程序的入口处，在该入口处会进一步使用call命令调用主函数，这时候也就完成了将 Return address 入栈的功能，因此这里无需画蛇添足压入返回地址
    tf->tf_eip = elf->e_entry;// 将返回地址设置为用户程序的入口
    tf->tf_eflags = FL_IF;// 允许中断，根据 IA32 的规范，eflags 的第 1 位需要恒为 1
    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;
}
```

3.回答问题

> 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"的概要设方案，鼓励给出详细设计方案;

观察到保存在磁盘上的 inode 信息均存在一个 nlinks 变量用于表示当前文件的被链接的计数，因而支持实现硬链接和软链接机制；

- 如果在磁盘上创建一个文件 A 的软链接 B，那么将 B 当成正常的文件创建 inode，然后将 TYPE 域设置为链接，然后使用剩余的域中的一个，指向 A 的 inode 位置，然后再额外使用一个位来标记当前的链接是软链接还是硬链接；
- 当访问到文件 B（read，write 等系统调用），判断如果 B 是一个链接，则实际是将对B指向的文件A（已经知道了 A 的 inode 位置）进行操作；
- 当删除一个软链接 B 的时候，直接将其在磁盘上的 inode 删掉即可；
- 如果在磁盘上的文件 A 创建一个硬链接 B，那么在按照软链接的方法创建完 B 之后，还需要将 A 中的被链接的计数加 1；
- 访问硬链接的方式与访问软链接是一致的；
- 当删除一个硬链接B的时候，除了需要删除掉 B 的 inode 之外，还需要将 B 指向的文件 A 的被链接计数减 1，如果减到了 0，则需要将 A 删除掉；

实验结果测试

- `make grade`最终的实验结果如下

```
-> % make grade
badsegment:              (s)
  -check result:                         OK
  -check output:                         OK
divzero:                 (s)
  -check result:                         OK
  -check output:                         OK
softint:                 (s)
  -check result:                         OK
  -check output:                         OK
faultread:               (s)
  -check result:                         OK
  -check output:                         OK
faultreadkernel:         (s)
  -check result:                         OK
  -check output:                         OK
hello:                   (s)
  -check result:                         OK
  -check output:                         OK
testbss:                 (s)
  -check result:                         OK
  -check output:                         OK
pgdir:                   (s)
  -check result:                         OK
  -check output:                         OK
yield:                   (s)
  -check result:                         OK
  -check output:                         OK
badarg:                  (s)
  -check result:                         OK
  -check output:                         OK
exit:                    (s)
  -check result:                         OK
  -check output:                         OK
spin:                    (s)
  -check result:                         OK
  -check output:                         OK
waitkill:                (s)
  -check result:                         OK
  -check output:                         OK
forktest:                (s)
  -check result:                         OK
  -check output:                         OK
forktree:                (s)
  -check result:                         OK
  -check output:                         OK
priority:                (s)
  -check result:                         OK
  -check output:                         OK
sleep:                   (s)
  -check result:                         OK
  -check output:                         OK
sleepkill:               (s)
```

```
  -check result:                          OK
  -check output:                          OK
matrix:                 (s)
  -check result:                          OK
  -check output:                          OK
Total Score: 190/190
```