Lab 8

操作系统课程实验报告

实验名称：Lab8 文件系统

小组成员： 2113388 高涵 2112849唐静蕾 2113999 陈翊炀

一、实验目的

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(即实现sfs_io_nolock()函数)，从新实现基于文件系统的执行程序机制（即实现load_icode()函数），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、实验内容

练习1：加载应用程序并执行

> 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

首先分析下打开一个文件的详细处理的流程。

如果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。

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` 。

4. 看到 `sfs_lookup` 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。

5. 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` 。

整个 `sfs_io_nolock()` 函数实现的功能是：

1. 先计算一些辅助变量，并处理一些特殊情况（比如越界），然后有 `sfs_buf_op = sfs_rbuf,sfs_block_op = sfs_rblock` ，设置读取的函数操作。
2. 接着进行实际操作，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。
3. 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。
4. 完成后如果 `offset + alen > din->fileinfo.size` （写文件时会出现这种情况，读文件时不会出现这种情况， `alen` 为实际读写的长度），则调整文件大小为 `offset + alen` 并设置 `dirty` 变量。

需要完成的主要是第二部分，即：

```
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 大小的块，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;
}
```

该函数主要负责从文件的指定偏移位置（offset）开始读取指定长度（size）的数据。以下是对代码功能的简述：

1. 处理第一部分的数据（可能不足一个块大小）：
    - 如果偏移位置 `offset` 不是块大小的整数倍，计算出未满一个块的偏移量 `blkoff`。
    - 计算第一个数据块的大小 `size`，如果还有剩余块（`nblks` 不为零），则为一个块减去 `blkoff`；否则，为 `endpos - offset`。

- 调用 `sfs_bmap_load_nolock` 函数找到内存文件索引 `sin` 对应的块号 `blkno`，以及块对应的磁盘块号 `ino`。
- 调用 `sfs_buf_op` 函数，通过 `ino` 号和 `blkoff`，将数据从磁盘块读入缓冲区 `buf`。
- 更新变量 `alen`（已经读取的总字节数）。
- 如果没有剩余块（`nblks == 0`），则跳出循环，结束读取。

2. 处理中间部分的数据：
   - 进入循环，每次读取一个完整的块（块大小为 `SFS_BLKSIZE`）。
   - 调用 `sfs_bmap_load_nolock` 函数找到内存文件索引 `sin` 对应的块号 `blkno`，以及块对应的磁盘块号 `ino`。
   - 调用 `sfs_block_op` 函数，通过 `ino` 号，将整个块的数据从磁盘块读入缓冲区 `buf`。
   - 更新变量 `alen`、`buf` 和 `nblks`。
3. 处理最后部分的数据：
   - 如果 `endpos` 不是块大小的整数倍，计算出最后一个块的大小 `size`。
   - 调用 `sfs_bmap_load_nolock` 函数找到内存文件索引 `sin` 对应的块号 `blkno`，以及块对应的磁盘块号 `ino`。
   - 调用 `sfs_buf_op` 函数，通过 `ino` 号，将数据从磁盘块读入缓冲区 `buf`。
   - 更新变量 `alen`。

最终，代码通过以上步骤完成了对指定范围内文件数据的读取。在实现中，通过调用不同的文件系统接口函数（如 `sfs_bmap_load_nolock` 和 `sfs_buf_op`），实现了对不同情况的处理。

练习2: 完成基于文件系统的执行程序机制的实现

> 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  static int
2  load_icode(int fd, int argc, char **kargv) {
3  
4      assert(argc >= 0 && argc <= EXEC_MAX_ARG_NUM);
5      if (current->mm != NULL) {
6          panic("load_icode: current->mm must be empty.\n");
7      }
8  
9      int ret = -E_NO_MEM;
10     struct mm_struct *mm;
11     if ((mm = mm_create()) == NULL) {
12         goto bad_mm;
```

```
    }
    if (setup_pgdir(mm) != 0) {
    //建立虚拟地址和物理地址之间的映射关系
        goto bad_pgdir_cleanup_mm;
    }

    struct Page *page;
    //主要差别在第三步
//读取文件中的原始数据内容并解析elfhdr
    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 ++) {
    //遍历 ELF 文件中的程序头表，找到类型为ELF_PT_LOAD的节。这些节包含了需要加载到内存的程序段的信息。
        off_t phoff = elf->e_phoff + sizeof(struct proghdr) * phnum;
        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 ;
            // do nothing here since static variables may not occupy any space
        }
        vm_flags = 0, perm = PTE_U | PTE_V;
        //对于每个满足条件的程序段，代码根据段的属性（读、写、执行）设置虚拟内存的标志位。
        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;
        // modify the perm bits here for RISC-V
        if (vm_flags & VM_READ) perm |= PTE_R;
```

```
        if (vm_flags & VM_WRITE) perm |= (PTE_W | PTE_R);
        if (vm_flags & VM_EXEC) perm |= PTE_X;
        //代码在内存中分配空间，并从文件中读取程序段的内容，将其加载到内存中
        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;
     //将可执行文件中的程序段加载到内存中，并设置合适的虚拟内存映射。
        end = ph->p_va + ph->p_filesz;
        while (start < end) {//遍历可执行文件中的所有程序段
        //通过pgdir_alloc_page函数为当前程序段分配一个物理页，并将其映射到指定的虚拟地址la
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
                ret = -E_NO_MEM;
                goto bad_cleanup_mmap;
            }
//代码计算当前页中需要加载的内容的偏移量（off）和大小（size），并将指针la向前移动一页
            off = start - la, size = PGSIZE - off, la += PGSIZE;
            if (end < la) {//当前程序段已经加载完毕，不需要再继续加载,进入下一次循环。
                size -= la - end;
            }
            if ((ret = load_icode_read(fd, page2kva(page) + off, size, offset)) != 0) {
                goto bad_cleanup_mmap;
            }
            start += size, offset += size;
        }
 //完成了加载程序段的循环后，代码更新end的值为当前程序段的虚拟地址加上程序段的内存大小
        end = ph->p_va + ph->p_memsz;


//检查是否还有未加载的部分。如果当前页的开始地址start小于la，说明存在未加载的部分。此时，代码会将未加载的部分填充为零，并更新start的值。
        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);
```

```
101             start += size;
102             assert((end < la && start == end) || (end >= la && start == la));
103         }
104 
105         //将剩余未加载的部分全部填充为零。这个循环的逻辑和之前的循环类似，只是不再读取文件中的数据，而是直接将内容设置为零
106         while (start < end) {
107             if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
108                 ret = -E_NO_MEM;
109                 goto bad_cleanup_mmap;
110             }
111             off = start - la, size = PGSIZE - off, la += PGSIZE;
112             if (end < la) {
113                 size -= la - end;
114             }
115             memset(page2kva(page) + off, 0, size);
116             start += size;
117         }
118     }
119     sysfile_close(fd);
120 
121     vm_flags = VM_READ | VM_WRITE | VM_STACK;//call mm_map to setup user stack, and put parameters into user stack
122      //设置用户栈的映射。用户栈是用来存放程序执行时的参数和局部变量等数据
123     if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL)) != 0) {
124         goto bad_cleanup_mmap;
125     }
126 
127     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-PGSIZE , PTE_USER) != NULL);
128     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-2*PGSIZE , PTE_USER) != NULL);
129     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-3*PGSIZE , PTE_USER) != NULL);
130     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-4*PGSIZE , PTE_USER) != NULL);
131 
132     mm_count_inc(mm);//setup current process mm, cr3, reset pgidr
133     current->mm = mm;
134     current->cr3 = PADDR(mm->pgdir);
135     lcr3(PADDR(mm->pgdir));
136 
137     //setup argc, argv
138     uint32_t argv_size=0, i;
139     for (i = 0; i < argc; i ++) {
140         argv_size += strnlen(kargv[i],EXEC_MAX_ARG_LEN + 1)+1;
141     }
142 
143     uintptr_t stacktop = USTACKTOP - (argv_size/sizeof(long)+1)*sizeof(long);
144     char** uargv=(char **)(stacktop  - argc * sizeof(char *));
```

```
145
146     argv_size = 0;
147     for (i = 0; i < argc; i ++) {
148         uargv[i] = strcpy((char *)(stacktop + argv_size ), kargv[i]);
149         argv_size +=  strnlen(kargv[i],EXEC_MAX_ARG_LEN + 1)+1;
150     }
151
152     stacktop = (uintptr_t)uargv - sizeof(int);
153     *(int *)stacktop = argc;
154
155     struct trapframe *tf = current->tf;//setup trapframe for user environment
156     // Keep sstatus
157     uintptr_t sstatus = tf->status;
158     memset(tf, 0, sizeof(struct trapframe));
159     tf->gpr.sp = stacktop;
160     tf->epc = elf->e_entry;
161     tf->status = sstatus & ~(SSTATUS_SPP | SSTATUS_SPIE);
162     ret = 0;
163 out:
164     return ret;
165 bad_cleanup_mmap:
166     exit_mmap(mm);
167 bad_elf_cleanup_pgdir:
168     put_pgdir(mm);
169 bad_pgdir_cleanup_mm:
170     mm_destroy(mm);
171 bad_mm:
172     goto out;
173 }
```

总体实现逻辑：

1. 首先，函数会检查参数的有效性，并进行一些初始化工作，比如创建内存管理结构（mm_struct）和建立虚拟地址和物理地址之间的映射关系。
2. 然后，函数会读取可执行文件的 ELF 头部信息，并对其进行验证，确保文件格式正确。
3. 接下来，函数会遍历可执行文件中的程序头表，找到类型为 ELF_PT_LOAD 的节，这些节包含了需要加载到内存的程序段的信息。
4. 对于每个需要加载的程序段，函数会根据段的属性（读、写、执行）设置虚拟内存的标志位，并在内存中分配空间，然后从文件中读取程序段的内容，将其加载到内存中。
5. 加载程序段时，如果某个程序段的大小超过一页的大小，函数会进行分页处理，确保所有内容都能正确加载到内存中。
6. 加载完所有程序段后，函数会设置用户栈的映射，用来存放程序执行时的参数和局部变量等数据。

7. 最后，函数会设置好用户环境的 trapframe 结构，准备开始执行用户程序。同时，如果在加载过程中发生错误，函数会进行相应的清理工作并返回相应的错误码

完成 `load_icode` 函数后 `make qemu` ，sh用户程序的执行界面成功运行起来，如下图所示：

```
Load page fault
page fault at 0x00001000: K/R
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
check_swap() succeeded!
sfs: mount: 'simple file system' (106/11/117)
vfs: mount disk0.
++ setup timer interrupts
kernel_execve: pid = 2, name = "sh".
Breakpoint
user sh is running!!!
```

在用户程序界面输入hello，程序成功执行放至在user文件下的hello.c文件：

```
user sh is running!!!
Hello world!!.
I am process 3.
hello pass.
$
```

按照题目所说的，可以认为本次实验基本成功了。

Make grade成功：

```
okup.c + cc kern/fs/vfs/vfsfile.c + cc kern/fs/vfs/inode.c + cc kern/fs/vfs/vfs.
c + cc kern/fs/devs/dev_stdin.c + cc kern/fs/devs/dev_disk0.c + cc kern/fs/devs/
dev_stdout.c + cc kern/fs/devs/dev.c + cc kern/fs/sfs/sfs.c + cc kern/fs/sfs/sfs
_io.c + cc kern/fs/sfs/sfs_lock.c + cc kern/fs/sfs/sfs_fs.c + cc kern/fs/sfs/sfs
_inode.c + cc kern/fs/sfs/bitmap.c + ld bin/kernel riscv64-unknown-elf-objcopy b
in/kernel --strip-all -O binary bin/ucore.img make[1]: 离开目录“/home/gaohan/ris
cv64-ucore-labcodes/lab8”
  -sh execve:                                    OK
  -user sh :                                     OK
Total Score: 100/100
gaohan@gaohan-virtual-machine:~/riscv64-ucore-labcodes/lab8$
```