

操作系统实验报告

实验名称：文件系统 实验地点：图书馆323

组号：56 小组成员：周钰宸 王志远 齐明杰

一、实验目的

- 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

二、实验过程

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。

答：

打开文件的处理流程

首先知道ucore文件管理分为四个层：

- **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。向下提供一个同样的抽象函数指针列表和数据结构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。

用户进程需要打开的文件已经存在在硬盘上。从开始到打开文件的总体调用链：

```
1 | main->runcmd->reopen|testfile->open->sys_open->sysfile_open->file_open->vfs_open->vop_open
```

(1) 通用文件访问接口层的处理流程

首先用户会在进程中调用 `open()` 函数，然后依次调用如下函数：`open->sys_open->sysfile_open`，从而引起系统调用进入到内核态。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 `"/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) 文件系统抽象层的处理流程

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（当前进程的打开文件数组 `current->fs_struct->filemap[]` 中的一个空闲元素），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。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。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) SFS文件系统层的处理流程

在第二步中，`vop_lookup` 函数调用了 `sfs_lookup` 函数。

下面分析sfs_lookup函数：

```
1 static int sfs_lookup(struct inode *node, char *path, struct inode
   **node_store) {
2     struct sfs_fs *sfs = fsop_info(vop_fs(node), sfs);
3     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。
4     vop_ref_inc(node);
5     struct sfs_inode *sin = vop_info(node, sfs_inode);
6     if (sin->din->type != SFS_TYPE_DIR) {
7         vop_ref_dec(node);
8         return -E_NOTDIR;
9     }
10    struct inode *subnode;
11    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。
12
13    vop_ref_dec(node);
14    if (ret != 0) {
15        return ret;
16    }
17    *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。
18    return 0;
19 }
```

`sfs_lookup` 函数先以“/”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。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。最后当无法分解path后，就意味着找到了需要对应的 `inode` 节点，就可顺利返回了。

完成sfs_io_nolock函数

`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。下面是该函数的代码：

```
1 /*
2  * sfs_io_nolock - Rd/Wr a file content from offset position to offset+
   length disk blocks<-->buffer (in memroy)
3  * @sfs:      sfs file system
4  * @sin:      sfs inode in memory
5  * @buf:      the buffer Rd/Wr
```

```

6  * @offset:   the offset of file
7  * @alenp:   the length need to read (is a pointer). and will RETURN the
really Rd/wr lenght
8  * @write:   BOOL, 0 read, 1 write
9  */
10 static int
11 sfs_io_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, void *buf, off_t
offset, size_t *alenp, bool write) {
12     struct sfs_disk_inode *din = sin->din;
13     assert(din->type != SFS_TYPE_DIR);
14     off_t endpos = offset + *alenp, blkoff;
15     *alenp = 0;
16     // calculate the Rd/wr end position
17     if (offset < 0 || offset >= SFS_MAX_FILE_SIZE || offset > endpos) {
18         return -E_INVALID;
19     }
20     if (offset == endpos) {
21         return 0;
22     }
23     if (endpos > SFS_MAX_FILE_SIZE) {
24         endpos = SFS_MAX_FILE_SIZE;
25     }
26     if (!write) {
27         if (offset >= din->size) {
28             return 0;
29         }
30         if (endpos > din->size) {
31             endpos = din->size;
32         }
33     }
34
35     int (*sfs_buf_op)(struct sfs_fs *sfs, void *buf, size_t len, uint32_t
blkno, off_t offset);
36     int (*sfs_block_op)(struct sfs_fs *sfs, void *buf, uint32_t blkno,
uint32_t nblks);
37     if (write) {
38         sfs_buf_op = sfs_wbuf, sfs_block_op = sfs_wblock;
39     }
40     else {
41         sfs_buf_op = sfs_rbuf, sfs_block_op = sfs_rblock;
42     }
43
44     int ret = 0;
45     size_t size, alen = 0;
46     uint32_t ino;
47     uint32_t blkno = offset / SFS_BLKSIZE;          // The NO. of Rd/wr
begin block
48     uint32_t nblks = endpos / SFS_BLKSIZE - blkno;  // The size of Rd/wr
blocks
49
50     //LAB8:EXERCISE1 YOUR CODE HINT: call sfs_bmap_load_nolock, sfs_rbuf,
sfs_rblock,etc. read different kind of blocks in file
51     /*
52      * (1) If offset isn't aligned with the first block, Rd/wr some content
from offset to the end of the first block
53      *      NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
54      *      Rd/wr size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) :
(endpos - offset)

```

```

55     * (2) Rd/wr aligned blocks
56     *      NOTICE: useful function: sfs_bmap_load_nolock, sfs_block_op
57     * (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
58     *      NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
59     如果偏移量(offset)没有与第一个块对齐, 那么从偏移量到第一个块的末尾读/写一些内容。
60     注意: 有用的函数包括 sfs_bmap_load_nolock 和 sfs_buf_op。
61     读/写的大小等于 (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset)。
62
63     读/写对齐的块。
64     注意: 有用的函数包括 sfs_bmap_load_nolock 和 sfs_block_op。
65
66     如果结束位置(end position)没有与最后一个块对齐, 那么从开始到最后一个块的 (endpos %
SFS_BLKSIZE) 读/写一些内容。
67     注意: 有用的函数包括 sfs_bmap_load_nolock 和 sfs_buf_op。
68     */
69     if ((blkoff = offset % SFS_BLKSIZE) != 0) {
70         size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
71         if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
72             goto out;
73         }
74         if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0) {
75             goto out;
76         }
77
78         alen += size;
79         buf += size;
80
81         if (nblks == 0) {
82             goto out;
83         }
84
85         blkno++;
86         nblks--;
87     }
88
89     if (nblks > 0) {
90         if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
91             goto out;
92         }
93         if ((ret = sfs_block_op(sfs, buf, ino, nblks)) != 0) {
94             goto out;
95         }
96
97         alen += nblks * SFS_BLKSIZE;
98         buf += nblks * SFS_BLKSIZE;
99         blkno += nblks;
100        nblks -= nblks;
101    }
102
103    if ((size = endpos % SFS_BLKSIZE) != 0) {
104        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
105            goto out;
106        }
107        if ((ret = sfs_buf_op(sfs, buf, size, ino, 0)) != 0) {
108            goto out;
109        }
110        alen += size;

```

```

111     }
112
113
114
115 out:
116     *alenp = alen;
117     if (offset + alen > sin->din->size) {
118         sin->din->size = offset + alen;
119         sin->dirty = 1;
120     }
121     return ret;
122 }

```

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 文件系统下的其他执行程序，则可以认为本实验基本成功。

答：

alloc_proc

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。

```

1 // alloc_proc - alloc a proc_struct and init all fields of proc_struct
2 static struct proc_struct *
3 alloc_proc(void) {
4     struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
5     if (proc != NULL) {
6         //LAB4:EXERCISE1 YOUR CODE
7         /*
8          * below fields in proc_struct need to be initialized
9          *      enum proc_state state;           // Process state
10         *      int pid;                         // Process ID
11         *      int runs;                       // the running
12         times of Proces
13         *      uintptr_t kstack;                // Process kernel
14         stack
15         *      volatile bool need_resched;      // bool value: need
16         to be rescheduled to release CPU?
17         *      struct proc_struct *parent;      // the parent
18         process
19         *      struct mm_struct *mm;            // Process's memory
20         management field
21         *      struct context context;          // Switch here to
22         run process
23         *      struct trapframe *tf;           // Trap frame for
24         current interrupt
25         *      uintptr_t cr3;                  // CR3 register:
26         the base addr of Page Directroy Table(PDT)
27         *      uint32_t flags;                  // Process flag
28         *      char name[PROC_NAME_LEN + 1];   // Process name
29         */
30     }
31 }

```

```

22 //LAB5 YOUR CODE : (update LAB4 steps)
23 /*
24  * below fields(add in LAB5) in proc_struct need to be initialized
25  *      uint32_t wait_state;           // waiting state
26  *      struct proc_struct *cptr, *yptr, *optr; // relations
between processes
27  */
28 //LAB6 YOUR CODE : (update LAB5 steps)
29 /*
30  * below fields(add in LAB6) in proc_struct need to be initialized
31  *      struct run_queue *rq;           // running queue
contains Process
32  *      list_entry_t run_link;           // the entry linked
in run queue
33  *      int time_slice;                 // time slice for
occupying the CPU
34  *      skew_heap_entry_t lab6_run_pool; // FOR LAB6 ONLY: the
entry in the run pool
35  *      uint32_t lab6_stride;           // FOR LAB6 ONLY: the
current stride of the process
36  *      uint32_t lab6_priority;         // FOR LAB6 ONLY: the
priority of process, set by lab6_set_priority(uint32_t)
37  */
38
39 //LAB8 YOUR CODE : (update LAB6 steps)
40 /*
41  * below fields(add in LAB6) in proc_struct need to be initialized
42  *      struct files_struct * filesp;   // file struct point
43  */
44     proc->state = PROC_UNINIT;
45     proc->pid = -1;
46     proc->runs = 0;
47     proc->kstack = NULL;
48     proc->need_resched = 0;
49     proc->parent = NULL;
50     proc->mm = NULL;
51     memset(&(proc->context), 0, sizeof(struct context));
52     proc->tf = NULL;
53     proc->cr3 = boot_cr3;
54     proc->flags = 0;
55     memset(proc->name, 0, PROC_NAME_LEN);
56     proc->wait_state = 0; //PCB新增的条目, 初始化进程等待状态
57     proc->cptr = proc->optr = proc->yptr = NULL; //设置指针
58     proc->filesp = NULL; //初始化fs中的进程控制结构
59 }
60
61 return proc;
62 }

```

load_icode

`load_icode` 函数的主要工作就是给用户进程建立一个能够让用户进程正常运行的用户环境。基本流程:

- 调用 `mm_create` 函数来申请进程的内存管理数据结构 `mm` 所需内存空间, 并对 `mm` 进行初始化;

- 调用setup_pgdir来申请一个页目录表所需的一个页大小的内存空间，并把描述ucore内核虚空间映射的内核页表（boot_pgdir所指）的内容拷贝到此新目录表中，最后让mm->pgdir指向此页目录表，这就是进程新的页目录表了，且能够正确映射内核虚空间；
- 将磁盘中的文件加载到内存中，并根据应用程序执行码的起始位置来解析此ELF格式的执行程序，并根据ELF格式的执行程序说明的各个段（代码段、数据段、BSS段等）的起始位置和大小建立对应的vma结构，并把vma插入到mm结构中，从而表明了用户进程的合法用户态虚拟地址空间；
- 调用根据执行程序各个段的大小分配物理内存空间，并根据执行程序各个段的起始位置确定虚拟地址，并在页表中建立好物理地址和虚拟地址的映射关系，然后把执行程序各个段的内容拷贝到相应的内核虚拟地址中
- 需要给用户进程设置用户栈，并处理用户栈中传入的参数
- 先清空进程的中断帧，再重新设置进程的中断帧，使得在执行中断返回指令“iret”后，能够让CPU转到用户态特权级，并回到用户态内存空间，使用用户态的代码段、数据段和堆栈，且能够跳转到用户进程的第一条指令执行，并确保在用户态能够响应中断；

大致可以简单分为以下部分：

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

完整代码如下：

```

1 // load_icode - 由 sys_exec-->do_execve 调用
2 // 加载新进程的代码和数据，建立其地址空间等
3 static int
4 load_icode(int fd, int argc, char **kargv) {
5     /* LAB8:EXERCISE2 YOUR CODE
6      * 提示：如何将带有处理器fd的文件加载到进程的内存中？如何设置argc/argv?
7      * 宏或函数：
8      * mm_create      - 创建内存管理结构
9      * setup_pgdir    - 在 mm 中设置分页目录
10     * load_icode_read - 读取程序文件的原始数据内容
11     * mm_map         - 建立新的虚拟内存区域
12     * pgdir_alloc_page - 为 TEXT/DATA/BSS/栈部分分配新内存
13     * lcr3           - 更新页目录地址寄存器 -- CR3
14     */
15     /* (1) 为当前进程创建一个新的mm
16      * (2) 创建一个新的PDT，并设置mm->pgdir = PDT的内核虚拟地址
17      * (3) 将二进制中的TEXT/DATA/BSS部分复制到进程的内存空间
18      * (3.1) 读取文件中的原始数据并解析elf头
19      * (3.2) 根据elf头信息读取文件中的原始数据并解析程序头
20      * (3.3) 调用mm_map来建立与TEXT/DATA相关的vma
21      * (3.4) 调用pgdir_alloc_page为TEXT/DATA分配页，读取文件内容并复制到新分配
    的页中
22      * (3.5) 调用pgdir_alloc_page为BSS分配页，这些页中的内容设置为零
23      * (4) 调用mm_map设置用户栈，并将参数放入用户栈中
24      * (5) 设置当前进程的mm, cr3, 重置pgdir（使用lcr3宏）
25      * (6) 在用户栈中设置uargc和uargv
26      * (7) 为用户环境设置陷阱帧
27      * (8) 如果上述步骤失败，应该清理环境

```

```

28     */
29     assert(argc >= 0 && argc <= EXEC_MAX_ARG_NUM);
30     //(1)建立内存管理器
31     if (current->mm != NULL) {
32         panic("load_icode: current->mm must be empty.\n");
33     }
34
35     int ret = -E_NO_MEM;
36     struct mm_struct *mm;
37     if ((mm = mm_create()) == NULL) {
38         goto bad_mm;
39     }
40     //(2)建立页目录
41     if (setup_pgdir(mm) != 0) {
42         goto bad_pgdir_cleanup_mm;
43     }
44
45     struct Page *page;
46
47     //(3)从文件加载程序到内存
48     struct elfhdr __elf, *elf = &__elf;
49     if ((ret = load_icode_read(fd, elf, sizeof(struct elfhdr), 0)) != 0) {
50         goto bad_elf_cleanup_pgdir;
51     }
52
53     if (elf->e_magic != ELF_MAGIC) {
54         ret = -E_INVALID_ELF;
55         goto bad_elf_cleanup_pgdir;
56     }
57
58     struct proghdr __ph, *ph = &__ph;
59     uint32_t vm_flags, perm, phnum;
60     for (phnum = 0; phnum < elf->e_phnum; phnum++) {
61         off_t phoff = elf->e_phoff + sizeof(struct proghdr) * phnum;
62         if ((ret = load_icode_read(fd, ph, sizeof(struct proghdr), phoff))
        != 0) {
63             goto bad_cleanup_mmap;
64         }
65         if (ph->p_type != ELF_PT_LOAD) {
66             continue;
67         }
68         if (ph->p_filesz > ph->p_memsz) {
69             ret = -E_INVALID_ELF;
70             goto bad_cleanup_mmap;
71         }
72         if (ph->p_filesz == 0) {
73             // continue;
74             // do nothing here since static variables may not occupy any
75             space
76         }
77         vm_flags = 0, perm = PTE_U | PTE_V; //建立虚拟地址与物理地址之间的映射
78         if (ph->p_flags & ELF_PF_X) vm_flags |= VM_EXEC;
79         if (ph->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;
80         if (ph->p_flags & ELF_PF_R) vm_flags |= VM_READ;
81         // modify the perm bits here for RISC-V
82         if (vm_flags & VM_READ) perm |= PTE_R;
83         if (vm_flags & VM_WRITE) perm |= (PTE_W | PTE_R);
84         if (vm_flags & VM_EXEC) perm |= PTE_X;

```



```

84         if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0)
85         {
86             goto bad_cleanup_mmap;
87         }
88         off_t offset = ph->p_offset;
89         size_t off, size;
90         uintptr_t start = ph->p_va, end, la = ROUNDDOWN(start, PGSIZE);
91
92         ret = -E_NO_MEM;
93
94         //复制数据段和代码段
95         end = ph->p_va + ph->p_filesz;
96         while (start < end) {
97             if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
98                 ret = -E_NO_MEM;
99                 goto bad_cleanup_mmap;
100             }
101             off = start - la, size = PGSIZE - off, la += PGSIZE;
102             if (end < la) {
103                 size -= la - end;
104             }
105             if ((ret = load_icode_read(fd, page2kva(page) + off, size,
offset)) != 0) {
106                 goto bad_cleanup_mmap;
107             }
108             start += size, offset += size;
109         }
110         end = ph->p_va + ph->p_memsz;
111
112         if (start < la) {
113             /* ph->p_memsz == ph->p_filesz */
114             if (start == end) {
115                 continue ;
116             }
117             off = start + PGSIZE - la, size = PGSIZE - off;
118             if (end < la) {
119                 size -= la - end;
120             }
121             memset(page2kva(page) + off, 0, size);
122             start += size;
123             assert((end < la && start == end) || (end >= la && start ==
1a));
124         }
125         while (start < end) {
126             if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
127                 ret = -E_NO_MEM;
128                 goto bad_cleanup_mmap;
129             }
130             off = start - la, size = PGSIZE - off, la += PGSIZE;
131             if (end < la) {
132                 size -= la - end;
133             }
134             memset(page2kva(page) + off, 0, size);
135             start += size;
136         }
137         sysfile_close(fd); //关闭文件，加载程序结束
138

```

```

139 // (4) 建立相应的虚拟内存映射表
140 vm_flags = VM_READ | VM_WRITE | VM_STACK;
141 if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags,
142 NULL)) != 0) {
143     goto bad_cleanup_mmap;
144 }
145 assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-PGSIZE, PTE_USER) !=
146 NULL);
147 assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-2*PGSIZE, PTE_USER) !=
148 NULL);
149 assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-3*PGSIZE, PTE_USER) !=
150 NULL);
151 assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-4*PGSIZE, PTE_USER) !=
152 NULL);
153
154 // (5) 设置用户栈
155 mm_count_inc(mm);
156 current->mm = mm;
157 current->cr3 = PADDR(mm->pgdir);
158 lcr3(PADDR(mm->pgdir));
159
160 // setup argc, argv
161 // (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
162 uint32_t argv_size=0, i;
163 for (i = 0; i < argc; i++) {
164     argv_size += strlen(kargv[i], EXEC_MAX_ARG_LEN + 1) + 1;
165 }
166
167 uintptr_t stacktop = USTACKTOP -
168 (argv_size/sizeof(long)+1)*sizeof(long);
169 char** uargv=(char**)(stacktop - argc * sizeof(char *));
170
171 argv_size = 0;
172 for (i = 0; i < argc; i++) {
173     uargv[i] = strcpy((char*)(stacktop + argv_size), kargv[i]);
174     argv_size += strlen(kargv[i], EXEC_MAX_ARG_LEN + 1) + 1;
175 }
176
177 stacktop = (uintptr_t)uargv - sizeof(int);
178 *(int*)stacktop = argc;
179
180 // (7) 设置进程的中断帧
181 struct trapframe *tf = current->tf;
182 // Keep sstatus
183 uintptr_t sstatus = tf->status;
184 memset(tf, 0, sizeof(struct trapframe));
185 tf->gpr.sp = stacktop;
186 tf->epc = elf->e_entry;
187 tf->status = sstatus & ~(SSTATUS_SPP | SSTATUS_SPIE);
188 ret = 0;
189 out:
190 return ret;
191 bad_cleanup_mmap:
192     exit_mmap(mm);
193 bad_elf_cleanup_pgdir:
194     put_pgdir(mm);
195 bad_pgdir_cleanup_mm:
196     mm_destroy(mm);

```

```
191 bad_mm:
192     goto out;
193
194 }
```

扩展练习 Challenge 1 : 完成基于UNIX的 PIPE 机制”的设计方案

如果要在 ucore 里加入 UNIX 的管道 (Pipe) 机制, 至少需要定义哪些数据结构和接口? (接口给出语义即

可, 不必具体实现。数据结构的设计应当给出一个(或多个) 具体的 C 语言 struct 定义。在网络上查找相关

的 Linux 资料和实现, 请在实验报告中给出设计实现“UNIX 的 PIPE 机制”的概要设方案, 你的设计应当体

现出对可能出现的同步互斥问题的处理。)

答:

管道可用于具有亲缘关系进程间的通信, 管道是由内核管理的一个缓冲区, 相当于我们放入内存中的一个纸条。管道的一端连接一个进程的输出。这个进程会向管道中放入信息。管道的另一端连接一个进程的输入, 这个进程取出被放入管道的信息。一个缓冲区不需要很大, 它被设计成为环形的数据结构, 以便管道可以被循环利用。当管道中没有信息的话, 从管道中读取的进程会等待, 直到另一端的进程放入信息。当管道被放满信息的时候, 尝试放入信息的进程会等待, 直到另一端的进程取出信息。当两个进程都终结的时候, 管道也自动消失。

在 Linux 中, 管道的实现并没有使用专门的数据结构, 而是借助了文件系统的file结构和VFS的索引节点inode。通过将两个 file 结构指向同一个临时的 VFS 索引节点, 而这个 VFS 索引节点又指向一个物理页面而实现的。

管道可以看作是由内核管理的一个**缓冲区**, 一端连接**进程A的输出**, 另一端连接**进程B的输入**。进程A会向管道中放入信息, 而进程B会取出被放入管道的信息。当管道中没有信息, 进程B会**等待**, 直到进程A放入信息。当管道被放满信息的时候, 进程A会等待, 直到进程B取出信息。当两个进程都结束的时候, 管道也自动消失。管道基于**fork**机制建立, 从而让两个进程可以连接到同一个PIPE上。

基于此, 我们可以模仿UNIX,设计一个**PIPE**机制。

1. 首先我们需要在磁盘上保留一定的区域用来作为PIPE机制的**缓冲区**, 或者创建一个文件为PIPE机制服务
2. 对系统文件初始化时将PIPE也**初始化**并创建相应的**inode**
3. 在内存中为PIPE**留一块区域**, 以便高效完成缓存
4. 当两个进程要建立管道时, 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
5. 当其中一个进程要对数据进行**写**操作时, 通过进程控制块的信息, 可以将其先**对临时文件PIPE进行修改**
6. 当一个进行需要对数据进行**读**操作时, 可以通过进程控制块的信息完成**对临时文件PIPE的读取**
7. 增添一些相关的系统调用支持上述操作

至此, **PIPE**的大致框架已经完成。

扩展练习 Challenge 2 : 完成基于UNIX的软连接和硬连接机制”的设计方案

如果要在 ucore 里加入 UNIX 的软连接和硬连接机制, 至少需要定义哪些数据结构和接口? (接口给出语义即

可，不必具体实现。数据结构的设计应当给出一个(或多个)具体的 C 语言 struct 定义。在网络上查找相关

的 Linux 资料和实现，请在实验报告中给出设计实现“UNIX 的软连接和硬连接机制”的概要设方案，你的设计应当体现出对可能出现的同步互斥问题的处理。)

答：

硬链接：是给文件一个副本，同时建立两者之间的连接关系

软链接：符号连接

硬链接和软链接的主要特征：

由于硬链接是有着相同 inode 号仅文件名不同的文件，因此硬链接存在以下几点特性：

- 文件有相同的 inode 及 data block;
- 只能对已存在的文件进行创建;
- 不能交叉文件系统进行硬链接的创建;
- 不能对目录进行创建，只可对文件创建;
- 删除一个硬链接文件并不影响其他有相同 inode 号的文件

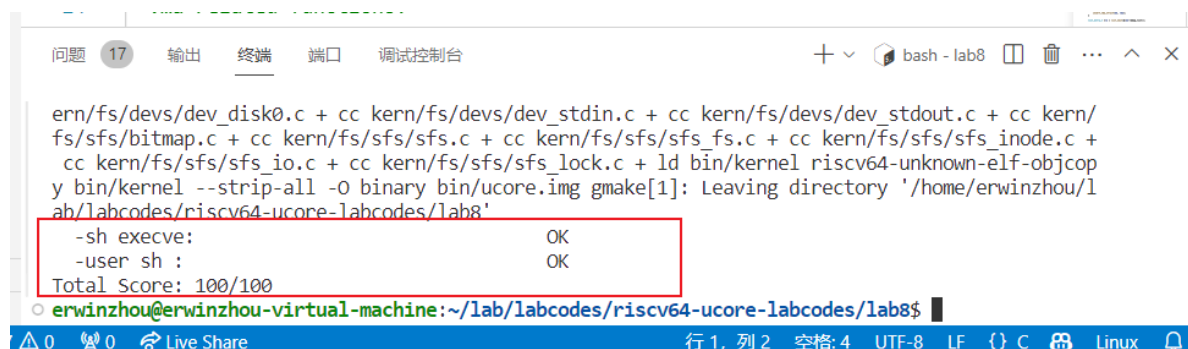
软链接的创建与使用没有类似硬链接的诸多限制：

- 软链接有自己的文件属性及权限等;
- 可对不存在的文件或目录创建软链接;
- 软链接可交叉文件系统;
- 软链接可对文件或目录创建;
- 创建软链接时，链接计数 i_nlink 不会增加;
- 删除软链接并不影响被指向的文件，但若被指向的原文件被删除，则相关软连接被称为死链接（即 dangling link，若被指向路径文件被重新创建，死链接可恢复为正常的软链接）。

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；

- 创建硬链接link时，为 new_path 创建对应的file，并把其inode指向old_path所对应的 inode，inode 的引用计数加1。
- 创建软连接link时，创建一个新的文件（inode不同），并把 old_path 的内容存放到文件的内容中去，给该文件保存在磁盘上时 disk_inode 类型为 SFS_TYPE_LINK，再完善对于该类型 inode 的操作即可。
- 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；但删除一个硬链接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；
- 硬链接的方式与访问软链接是一致的；

三、与参考答案的对比



```
ern/fs/devs/dev_disk0.c + cc kern/fs/devs/dev_stdin.c + cc kern/fs/devs/dev_stdout.c + cc kern/
fs/sfs/bitmap.c + cc kern/fs/sfs/sfs.c + cc kern/fs/sfs/sfs_fs.c + cc kern/fs/sfs/sfs_inode.c +
cc kern/fs/sfs/sfs_io.c + cc kern/fs/sfs/sfs_lock.c + ld bin/kernel riscv64-unknown-elf-objcop
y bin/kernel --strip-all -O binary bin/ucore.img gmake[1]: Leaving directory '/home/erwinzhou/l
ab/labcodes/riscv64-ucore-labcodes/lab8'
-sh execve: OK
-user sh : OK
Total Score: 100/100
erwinzhou@erwinzhou-virtual-machine:~/lab/labcodes/riscv64-ucore-labcodes/lab8$
```

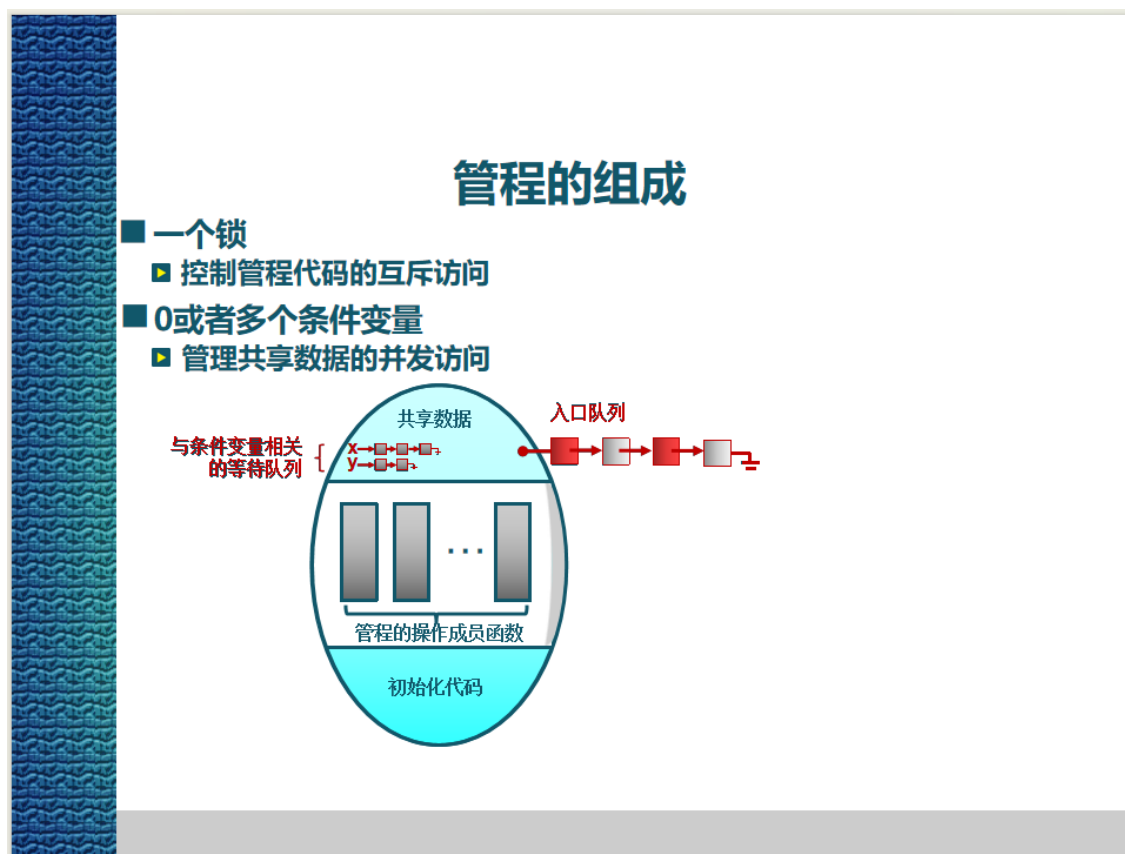
由于本实验的代码逻辑较为固定，因此我们小组所完成的代码与参考答案差别不大。

四、实验中的知识点

1. ucore模仿了UNIX的文件系统设计，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。向下提供一个同样的抽象函数指针列表和数据结构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。

2. 管程：使用信号变量，来解决生产者与消费者问题。



条件变量 (Condition Variable)

■ 条件变量是管程内的等待机制

- 进入管程的线程因资源被占用而进入等待状态
- 每个条件变量表示一种等待原因，对应一个等待队列

■ Wait()操作

- 将自己阻塞在等待队列中
- 唤醒一个等待者或释放管程的互斥访问

■ Signal()操作

- 将等待队列中的一个线程唤醒
- 如果等待队列为空，则等同空操作

管程的设计思路是面向对象的编程方法与操作系统的权限管理的结合
操作系统负责管理线程的状态
用一个变量与线程紧密绑定，并且不许其他线程访问和操作

条件变量实现

```
Class Condition {  
    int numWaiting = 0;  
    WaitQueue q;  
}
```

```
Condition::Wait(lock) {  
    numWaiting++;  
    Add this thread t to q;  
    release(lock);  
    schedule(); //need mutex  
    require(lock);  
}
```

```
Condition::Signal() {  
    if (numWaiting > 0) {  
        Remove a thread t from q;  
        wakeup(t); //need mutex  
        numWaiting--;  
    }  
}
```

用管程解决生产者-消费者问题

```
class BoundedBuffer {  
    ...  
    Lock lock;  
    int count = 0;  
    Condition notFull, notEmpty;  
}
```

```
BoundedBuffer::Deposit(c) {  
    lock->Acquire();  
    while (count == n)  
        notFull.Wait(&lock);  
    Add c to the buffer;  
    count++;  
    notEmpty.Signal();  
    lock->Release();  
}
```

```
BoundedBuffer::Remove(c) {  
    lock->Acquire();  
    while (count == 0)  
        notEmpty.Wait(&lock);  
    Remove c from buffer;  
    count--;  
    notFull.Signal();  
    lock->Release();  
}
```