实验8报告文档

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。

解答：

(1) ucore的文件系统架构主要由四部分组成：

① 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。

② 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。

③ 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

④ 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) 打开文件的处理流程

① 首先是应用程序发出请求，请求硬盘中写数据或读数据，应用程序通过FS syscall接口执行系统调用，获得ucore操作系统关于文件的一些服务；

② 之后，一旦操作系统内系统调用得到了请求，就会到达VFS层面（虚拟文件系统），包含很多部分比如文件接口、目录接口等，是一个抽象层面，它屏蔽底层具体的文件系统；

③ VFS如果得到了处理，那么VFS会将这个iNode传递给SimpleFS，注意，此时，VFS中的iNode还是一个抽象的结构，在SimpleFS中会转化为一个具体的iNode；

④ 通过该iNode经过IO接口对于磁盘进行读写。

(3) 填写sfs_io_nolock()函数，实现读文件中数据的代码

当进行文件读取/写入操作时，最终uCore都会执行到sfs_io_nolock函数。在该函数中，我们要完成对设备上基础块数据的读取与写入。在进行读取/写入前，我们需要先将数据与基础块对齐，以便于使用sfs_block_op函数来操作基础块，提高读取/写入效率，这一部分不需要补充。

但一旦将数据对齐后会存在一个首尾读取的问题：待操作数据的前一小部分有可能在最前的一个基础块的末尾位置，且待操作数据的后一小部分有可能在最后的一个基础块的起始位置，我们需要分别对这第一和最后这两个位置的基础块进行读写/写入，因为这两个位置的基础块所涉及到的数据都是部分的。而中间的数据由于已经对齐好基础块了，所以可以每次通过sfs_bmap_load_nolock函数获取文件索引编号，然后直接调用sfs_block_op来读取/写入数据。以下是相关操作在函数中的补充实现如下：

```
// 对齐偏移。如果偏移没有对齐第一个基础块，则多读取/写入第一个基础块的末尾数据
    if ((blkoff = offset % SFS_BLKSIZE) != 0) {
        size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
        // 获取第一个基础块所对应的block的编号ino
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0)
            goto out;
        // 通过上一步取出的ino，读取/写入一部分第一个基础块的末尾数据
        if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0)
            goto out;
        alen += size;
        if (nblks == 0)
            goto out;
        buf += size, blkno ++, nblks --;
    }
    // 循环读取/写入对齐好的数据
    size = SFS_BLKSIZE;
    while (nblks != 0) {
        // 获取inode对应的基础块编号
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0)
            goto out;
        // 单次读取/写入一基础块的数据
        if ((ret = sfs_block_op(sfs, buf, ino, 1)) != 0)
            goto out;
        alen += size, buf += size, blkno ++, nblks --;
    }
    // 如果末尾位置没有与最后一个基础块对齐，则多读取/写入一点末尾基础块的数据
    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;
    }
```

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文件系统中的其他执行程序，则可以认为本实验基本成功。

解答：

在proc.c中，根据注释我们需要先在alloc_proc函数中初始化fs中的进程控制结构，即通过加上proc->filesp = NULL;完成初始化。

```

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_po
        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;
}

```

接下来我们修改load_icode函数，将将文件加载到内存中执行，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.设置用户进程的中断帧

下面是完整代码和详细解释。

```
1
2  static int
3  load_icode(int fd, int argc, char **kargv) {
4      assert(argc >= 0 && argc <= EXEC_MAX_ARG_NUM);
5           //(1)建立内存管理器
6      if (current->mm != NULL) {
7          panic("load_icode: current->mm must be empty.\n");
8      }
9
10     int ret = -E_NO_MEM;
11     // 创建proc的内存管理结构
12     struct mm_struct *mm;
13     if ((mm = mm_create()) == NULL) {
14         goto bad_mm;
15     }
16     if (setup_pgdir(mm) != 0) {
17         goto bad_pgdir_cleanup_mm;
18     }
19          //(2)建立页目录
20     struct Page *page;
21     struct elfhdr elf_content;
22     struct elfhdr *elf = &elf_content;
23     if ((ret = load_icode_read(fd, elf, sizeof(struct elfhdr), 0)) != 0) {
24         goto bad_elf_cleanup_pgdir;
25     }
26     // 判断读取入的elf header是否正确
27     if (elf->e_magic != ELF_MAGIC) {
28         ret = -E_INVAL_ELF;
29         goto bad_elf_cleanup_pgdir;
30     }
31     //(3)从文件加载程序到内存
32     // 根据每一段的大小和基地址来分配不同的内存空间
33     struct proghdr __ph, *ph = &__ph;
34     uint32_t vm_flags, perm, phnum;
35     for (phnum = 0; phnum < elf->e_phnum; phnum ++) {
```

```
36      //############ LAB8 从文件特定偏移处读取每个段的详细信息（包括大小、基地址等等）####
37          off_t phoff = elf->e_phoff + sizeof(struct proghdr) * phnum;
38          if ((ret = load_icode_read(fd, ph, sizeof(struct proghdr), phoff)) != 0)
39              goto bad_cleanup_mmap;
40          }
41          if (ph->p_type != ELF_PT_LOAD) {
42              continue ;
43          }
44          if (ph->p_filesz > ph->p_memsz) {
45              ret = -E_INVAL_ELF;
46              goto bad_cleanup_mmap;
47          }
48          if (ph->p_filesz == 0) {
49              // continue ;
50          }
51          vm_flags = 0, perm = PTE_U | PTE_V;
52          if (ph->p_flags & ELF_PF_X) vm_flags |= VM_EXEC;
53          if (ph->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;
54          if (ph->p_flags & ELF_PF_R) vm_flags |= VM_READ;
55
56          if (vm_flags & VM_READ) perm |= PTE_R;
57          if (vm_flags & VM_WRITE) perm |= (PTE_W | PTE_R);
58          if (vm_flags & VM_EXEC) perm |= PTE_X;
59
60          // 为当前段分配内存空间
61          if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0) {
62              goto bad_cleanup_mmap;
63          }
64          off_t offset = ph->p_offset;
65          size_t off, size;
66          uintptr_t start = ph->p_va, end, la = ROUNDDOWN(start, PGSIZE);
67
68          ret = -E_NO_MEM;
69
70          end = ph->p_va + ph->p_filesz;
71          while (start < end) {
72              // 设置该内存所对应的页表项
73              if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
74                  ret = -E_NO_MEM;
75                  goto bad_cleanup_mmap;
76              }
77              off = start - la, size = PGSIZE - off, la += PGSIZE;
78              if (end < la) {
79                  size -= la - end;
80              }
81  //################### LAB8 读取elf对应段内的数据并写入至该内存中###################
82              if ((ret = load_icode_read(fd, page2kva(page) + off, size, offset))
```

```
83                  goto bad_cleanup_mmap;
84              }
85              start += size, offset += size;
86          }
87          end = ph->p_va + ph->p_memsz;
88          // 对于段中当前页中剩余的空间（复制elf数据后剩下的空间），将其置为0
89          if (start < la) {
90              /* ph->p_memsz == ph->p_filesz */
91              if (start == end) {
92                  continue ;
93              }
94              off = start + PGSIZE - la, size = PGSIZE - off;
95              if (end < la) {
96                  size -= la - end;
97              }
98              memset(page2kva(page) + off, 0, size);
99              start += size;
100             assert((end < la && start == end) || (end >= la && start == la));
101         }
102         // 对于段中剩余页中的空间（复制elf数据后的多余页面），将其置为0
103         while (start < end) {
104             if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
105                 ret = -E_NO_MEM;
106                 goto bad_cleanup_mmap;
107             }
108             off = start - la, size = PGSIZE - off, la += PGSIZE;
109             if (end < la) {
110                 size -= la - end;
111             }
112             memset(page2kva(page) + off, 0, size);
113             start += size;
114         }
115     }
116     // 关闭读取的ELF
117     sysfile_close(fd);
118 //(4)建立相应的虚拟内存映射表
119     // 设置栈内存
120     vm_flags = VM_READ | VM_WRITE | VM_STACK;
121     if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL)) !
122         goto bad_cleanup_mmap;
123     }
124     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-PGSIZE , PTE_USER) != NULL);
125     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-2*PGSIZE , PTE_USER) != NULL);
126     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-3*PGSIZE , PTE_USER) != NULL);
127     assert(pgdir_alloc_page(mm->pgdir, USTACKTOP-4*PGSIZE , PTE_USER) != NULL);
128 //(5)设置用户栈
129     mm_count_inc(mm);
```

```
130     // 设置CR3页表相关寄存器
131     current->mm = mm;
132     current->cr3 = PADDR(mm->pgdir);
133     lcr3(PADDR(mm->pgdir));
134     //(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
135     // ######################LAB8 设置execve所启动的程序参数#####################
136     uint32_t argv_size = 0;
137     int i;
138     for (i = 0; i < argc; i ++) {
139         argv_size += strnlen(kargv[i],EXEC_MAX_ARG_LEN + 1)+1;
140     }
141 
142     uintptr_t stacktop = USTACKTOP - (argv_size/sizeof(long)+1)*sizeof(long);
143     // 直接将传入的参数压入至新栈的底部
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     //(7)设置进程的中断帧
156     struct trapframe *tf = current->tf;
157 
158     uintptr_t sstatus = tf->status;
159     memset(tf, 0, sizeof(struct trapframe));
160     /* LAB5:EXERCISE1 2112189
161      * should set tf->gpr.sp, tf->epc, tf->status
162      * NOTICE: If we set trapframe correctly, then the user level process can re
163      *          tf->gpr.sp should be user stack top (the value of sp)
164      *          tf->epc should be entry point of user program (the value of sepc
165      *          tf->status should be appropriate for user program (the value of
166      *          hint: check meaning of SPP, SPIE in SSTATUS, use them by SSTATUS
167      */
168     tf->gpr.sp = USTACKTOP;
169     tf->epc = elf->e_entry;
170     tf->status = (sstatus & ~SSTATUS_SPP) | SSTATUS_SPIE;
171     ret = 0;
172      //(8)错误处理部分
173 out:
174     return ret;
175 bad_cleanup_mmap:
176     exit_mmap(mm);
```

```
177 bad_elf_cleanup_pgdir:
178     put_pgdir(mm);
179 bad_pgdir_cleanup_mm:
180     mm_destroy(mm);
181 bad_mm:
182     goto out;
183 }
```

1. **内存管理数据结构的初始化**：通过调用 `mm_create` 函数申请并初始化进程的内存管理数据结构 `mm`。接着， `setup_pgdir` 被调用以分配一个页大小的内存空间，用作页目录表。此新分配的页目录表复制了描述 ucore 内核虚拟空间映射的内核页表（由 `boot_pgdir` 指向）的内容。最终， `mm->pgdir` 指向这个新的页目录表，成为进程的新页目录表，同时确保内核虚拟空间被正确映射。

2. **应用程序加载与虚拟地址空间建立**：根据应用程序文件的文件描述符（ `fd` ），通过调用 `load_icode_read()` 函数加载并解析该 ELF 格式的执行程序。接着，调用 `mm_map` 函数根据 ELF 执行程序的段（如代码段、数据段、BSS 段）的起始位置和大小来建立相应的虚拟内存区域（VMA）结构，并将这些 VMA 插入到 `mm` 结构中。这表明了用户进程的合法用户态虚拟地址空间。

3. **执行程序内存分配与映射**：根据执行程序的各个段的大小分配物理内存空间，并根据这些段的起始位置确定虚拟地址。随后，在页表中建立物理地址和虚拟地址之间的映射关系。然后，将执行程序的各个段内容拷贝到相应的内核虚拟地址中，使得应用程序的执行代码和数据按照编译时设定的地址放置于虚拟内存中。

4. **用户栈的设置**：通过调用 `mm_mmap` 函数来建立用户栈的虚拟内存区域（VMA）结构。用户栈被设定在用户虚拟空间的顶部，大小为 256 页，即 1MB。此外，分配相应的物理内存，并建立栈的虚拟地址与物理地址之间的映射关系。

5. **内存管理结构的完成与用户进程准备**：完成进程内的内存管理 VMA 和 `mm` 数据结构的建立后，将 `mm->pgdir` 的值赋给 CR3 寄存器，更新用户进程的虚拟内存空间。同时，在用户栈中设置 `uargc` 和 `uargv`。此时， `initproc` 已被代码和数据覆盖，转变为第一个用户进程，尽管此时用户进程的执行现场尚未完全建立。

6. **中断帧的设置**：首先清空进程的中断帧，然后重新设置进程的中断帧。这样，在执行中断返回指令 `iret` 后，CPU 能够切换到用户态特权级，返回到用户态内存空间。这允许使用用户态的代码段、数据段和堆栈，并能跳转至用户进程的第一条指令执行。同时，确保在用户态能够正确响应中断。

扩展练习 Challenge1：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。）

解答：

PIPE机制的设计方案：

管道是由内核管理的一个缓冲区,可用于具有亲缘关系进程间的通信。管道的一端连接一个进程的输出,另一端连接另一个进程的输入。

设立管道的目的是Linux下的进程的用户态地址空间都是相互独立的，因此两个进程在用户态是没法直接通信的，因为找不到彼此的存在；而内核是进程间共享的，因此用户进程间想通信只能通过内核作为中间人来传达信息。接下来讲述PIPE机制的具体工作流程。

为便于解释，现在我们假设一个用户进程A向用户进程B传输信息的情景。那么管道作为由内核管理的一个缓冲区，一端连接用户进程A的输出，另一端连接用户进程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。

基于以上内容，也可以解释为什么管道是单通道的：因为只有一个缓存，如果是双通道，那么两个进程同时向这块缓存写数据时，这样会导致数据覆盖，即一个进程的数据被另一个进程的数据覆盖．基于以上原因。我们只允许同一时间内只有一个进程对PIPE进行写操作。同样，由于PIPE读操作是把缓冲区数据取走，我们也只允许同一时间内只有一个进程对PIPE进行读操作。

PIPE机制的数据结构和接口：

首先定义管道的缓冲区结构体pipe_buffer，其C语言代码定义如下：

```
struct pipe_buffer {
    struct page *page;      // 缓冲区所在的页
    unsigned int offset;    // 缓冲区的偏移量
    unsigned int len;       // 缓冲区的长度
    unsigned int flags;     // 缓冲区的标志位
};
```

然后基于以上结构体，定义管道结构体pipe_inode_info，其C语言代码定义如下：

```
struct pipe_inode_info {
    wait_queue_t wait;                // 等待队列
    struct pipe_buffer *bufs;         // 缓冲区数组
    unsigned int nrbufs;              // 缓冲区数量
    unsigned int curbuf;              // 当前缓冲区
    unsigned int readers;             // 读者数量
    unsigned int writers;             // 写者数量
    unsigned int waiting_writers;     // 等待写者数量
    struct inode *inode_pipe;         // 管道对应的 inode
    list_entry_t pipe_inode_link;     // 管道链表
};
```

接下来定义管道的相关接口。首先是创建管道的函数do_pipe，其C语言代码如下：

```
int do_pipe(unsigned int pipefd[2]);
```

do_pipe接收一个大小为2的无符号整数数组pipefd，do_pipe函数最后会把分别对应管道读端和写端的两个文件描述符放到这个数组里面。如果创建管道成功，则do_pipe函数的返回值就是0，否则为其它的错误信息值。do_pipe函数具体工作流程如下：

1. 内核实例化两个空file结构体；
2. 创建带有pipe属性的inode结构；
3. 在当前进程文件描述符表中找出两个未使用的文件描述符;
4. 为这个inode关联file;
5. 针对可读和可写file结构，分别设置相应属性，主要是操作函数集合属性；
6. 关联文件描述符和file结构
7. 将两个文件描述符返回给用户；

然后就是管道读操作接口pipe_read,其C语言代码如下：

```
int pipe_read(int fd, void *buf, unsigned int count);
```

pipe_read函数负责从用户程序的接收数据缓冲区读取数据。首先，它将用户态缓冲区和大小(对应传入的参数buf和count)转换为iovec结构，并在一个循环中逐步将管道缓冲区中的数据复制到用户态缓冲区。这过程会持续直到用户缓冲区达到容量上限或者管道缓冲区中的所有数据均已读取。如果管道

缓冲区为空，pipe_read函数将导致当前读管道的进程阻塞，以便切换到其他进程执行。最终，一旦用户态的read函数返回，即可在用户缓冲区中获取到来自管道的数据。

最后是然后就是管道写操作接口pipe_write,其C语言代码如下：

```
1 int pipe_write(int fd, void *buf, unsigned int count);
```

实际上，PIPE的写过程其实可以认为就是读过程的逆操作。pipe_write函数负责向用户程序的发送数据缓冲区写入数据，它在一个循环中逐步将用户态缓冲区中的数据复制到管道缓冲区，和pipe_read复制数据的方向相反。而且pipe_write会检测如果管道缓冲区已满，将导致当前写入管道的进程阻塞。

扩展练习 Challenge2：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。）

解答：

软连接和硬连接机制的设计方案：

硬链接实际上是文件有着相同 inode 号、data block等信息，区别仅仅的文件名不同。也就是说硬链接是给文件一个副本，同时建立两者之间的连接关系。因此硬链接存在以下几点特性：

- 文件有相同的索引节点 inode 及数据块 data block；
- 不允许为目录创建硬链接；
- 不能跨越文件系统进行硬链接的创建；
- 修改任何一个硬链接都会影响其他硬链接，因为它们指向相同的数据块。
- 删除一个硬链接文件并不影响其他有相同 inode 号的文件，只有当所有硬链接都被删除时，文件的数据块才会被释放。

软链接(又叫符号链接)实际上保存了其代表的文件的绝对路径，是另外一种文件，在硬盘上有独立的区块，访问时替换自身路径。软链接类似于给文件一个快捷方式。因此软链接存在以下几点特性：

- 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；
- 可以链接到文件、目录或者其他软链接；
- 软链接可以跨越文件系统；
- 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）。

总的来说，硬链接通过共享相同的 inode 实现，而软链接则是创建一个新的 inode，并在其内容中保存原路径的信息。硬链接的删除需要注意被链接文件的引用计数，而软链接的删除则相对简单。无论是硬链接还是软链接，它们提供了一种有效的方式来在文件系统中建立链接和引用关系。此外，访问硬链接的方式与访问软链接是一致的。

软连接和硬连接机制的数据结构与接口：

在设计硬链接和软链接机制时，需要对 inode 进行扩展，扩展后的C语言代码如下：

```
struct inode {
    // 其他字段
    int ref_count;         // 引用计数
    // ...
};
```

同时还要引入新的数据结构表示链接，C语言代码如下：

```
struct link {
    struct inode *inode;  // 指向inode的指针
    int link_count;       // 链接计数
    bool is_symlink;      // 是否是软链接
};
```

相关接口设计如下：

创建硬链接：

创建硬链接时，系统为新路径创建一个文件，并将其 inode 指向原路径所对应的 inode。同时，原路径所对应 inode 的引用计数增加，表示有一个额外的硬链接指向它。该接口函数设计代码如下：

```
int create_hlink(const char *source_path, const char *target_path);
```

创建软链接：

创建软链接时，系统创建一个新的文件（新的 inode），并将其内容设置为原路径的内容。在磁盘上保存该文件时，该文件的 inode 类型被标记为 SFS_TYPE_LINK，同时需要对这种类型的 inode 进行相应的操作。该接口函数设计代码如下：

```
1 int create_slink(const char *source_path, const char *target_path);
```

删除硬链接：

删除硬链接时，除了需要删除硬链接的 inode，还需要将该硬链接所指向的文件的被链接计数减1。如果减到了0，表示没有其他硬链接指向该文件，此时需要将该文件对应的 inode 从磁盘上删除。该接口函数设计代码如下：

```
1 int remove_hlink(const char *link_path);
```

删除软链接：

删除软链接时则不像硬链接那样复杂，只需将软链接对应的 inode 从磁盘上删除即可。该接口函数设计代码如下：

```
1 int remove_slink(const char *link_path);
```

以上所有的函数，如果创建/删除链接成功，则返回值为0，否则为其它的错误信息值。