操作系统：Lab8

> 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

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。

```
// SFS 文件系统的非锁定输入输出函数
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;

    // 确保文件类型不是目录
    assert(din->type != SFS_TYPE_DIR);

    off_t endpos = offset + *alenp, blkoff;
    *alenp = 0;

    // 计算读写结束位置
    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);
```

```
    // 根据读写标志选择相应的缓冲区和块操作函数
    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;          // 开始进行读写的块号
    uint32_t nblks = endpos / SFS_BLKSIZE - blkno;  // 读写的块数

    // code
    // (1) 如果偏移量与第一个块不对齐，则读写一些内容从偏移量到第一个块的末尾
    //     提示：使用 sfs_bmap_load_nolock, sfs_buf_op
    //           读写大小 = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos -
offset)
    if ((blkoff = offset % SFS_BLKSIZE) != 0)  {
        blkoff = offset % SFS_BLKSIZE;
        size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        // 将文件系统中块号 blkno 对应的逻辑块映射为物理磁盘上的块号 ino
        if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0) {
            goto out;
        }
        // 执行实际的数据读取或写入操作，根据 write 标志选择了读取还是写入操作
        alen += size;
        if (nblks == 0) {
            goto out;
        }
        buf += size, blkno++; nblks--;
    }

    // (2) 读写对齐的块
    //     提示：使用 sfs_bmap_load_nolock, sfs_block_op
    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--;
    }

    // (3) 如果结束位置与最后一个块不对齐，则读写一些内容从开头到最后一个块的
(endpos % SFS_BLKSIZE) 处
    //     提示：使用 sfs_bmap_load_nolock, sfs_buf_op
    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;
    }
    //code end
out:
    *alenp = alen;

    // 如果读写的数据范围超过了文件当前的大小，则更新文件的大小
    if (offset + alen > sin->din->size) {
        sin->din->size = offset + alen;
        sin->dirty = 1; // 将 dirty 标志设为 1，表示该节点需要写回磁盘
    }

    return ret;
}
```

- `size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);`
  - 如果 nblks 不等于 0，表示当前块之后仍有其他块需要读取或写入，则当前块的可用空间为 SFS_BLKSIZE - blkoff，其中 SFS_BLKSIZE 是块的大小，blkoff 是当前偏移量在块中的偏移量。
  - 如果 nblks 等于 0，表示当前块是最后一个块，即将读取或写入的数据在当前块内就可以完成。这时将计算剩余的要读取或写入的数据量，即 endpos - offset，endpos 是要读写的结束位置，offset 是当前偏移量。
- `buf += size`：将指针 buf 向后移动，跳过已经处理过的数据块大小。这是为了在下一次循环中指向下一个数据块的起始位置。
- `blkno++`：递增 blkno，将其更新为下一个块的块号。这是为了在下一次循环中指向下一个数据块。
- `nblks--`：递减 nblks，表示还需要处理的块数减少了一个。这是为了控制循环的次数，确保在所有需要处理的块都被处理完毕后退出循环

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。

修改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。

```
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        // Lab7内容
        // ...

        //LAB8:EXERCISE2 YOUR CODE HINT:need add some code to init fs in
proc_struct, ...
        // LAB8 添加一个filesp指针的初始化
```

```
        proc->filesp = NULL;
    }
    return proc;
}
```

实现 load_icode 函数

- 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；
- 简单来说，就是：
    1. 建立内存管理器
    2. 建立页目录
    3. 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

```
static int
load_icode(int fd, int argc, char **kargv) {
    assert(argc >= 0 && argc <= EXEC_MAX_ARG_NUM);
    if (current->mm != NULL) {
        panic("load_icode: current->mm must be empty.\n");
    }

    int ret = -E_NO_MEM;
    // (1) 为当前进程创建一个新的内存管理结构（mm）
    struct mm_struct *mm;

    // (2) 创建一个新的页目录表（PDT），并将 mm->pgdir 设置为页目录表的内核虚拟地址
    if ((mm = mm_create()) == NULL) {
```

```
        goto bad_mm;
    }
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }

    // (3) 复制 TEXT/DATA 段，将二进制文件中的 BSS 部分加载到进程的内存空间中
    struct elfhdr __elf, *elf = &__elf;
    struct proghdr __ph, *ph = &__ph;
    uint32_t vm_flags, perm, phnum;
    for (phnum = 0; phnum < elf->e_phnum; phnum++) {
        // (3.1) 获取二进制程序的文件头（ELF 格式）
        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;
        }
        // (3.3) 验证程序是否有效
        if (ph->p_type != ELF_PT_LOAD) {
            continue;
        }
        if (ph->p_filesz > ph->p_memsz) {
            ret = -E_INVAL_ELF;
            goto bad_cleanup_mmap;
        }
        // 如果程序段在文件中的大小超过了在内存中的预期大小，就认为这个 ELF 格式的程
序段是无效的
        if (ph->p_filesz == 0) {
            // do nothing here since static variables may not occupy any space
            continue;
        }

        // (3.5) 使用 mm_map 函数设置新的 VMA（ph->p_va, ph->p_memsz）
        vm_flags = 0, perm = PTE_U | PTE_V;
        // perm初始化为用户态和有效位
        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;
        // 修改 RISC-V 中的权限位
        if (vm_flags & VM_READ) perm |= PTE_R;
        if (vm_flags & VM_WRITE) perm |= (PTE_W | PTE_R);
        // 组合：如果一个页面被标记为可写，通常也意味着它是可读的。
        if (vm_flags & VM_EXEC) perm |= PTE_X;
        if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0) {
            goto bad_cleanup_mmap;
        }
        // 建立或修改虚拟地址空间中的映射关系
        // ph->p_va 是程序段在虚拟地址空间中的起始地址
        // ph->p_memsz 是程序段在内存中的大小。
        // vm_flags 是描述程序段权限和特性的标志位。

        off_t offset = ph->p_offset;
        size_t off, size;
        uintptr_t start = ph->p_va, end, la = ROUNDDOWN(start, PGSIZE);
        // 将 start 这个地址向下舍入到最接近的较小的 PGSIZE 的整数倍
        // 比如说就是 start是4095 则 la为0
```

```
        ret = -E_NO_MEM;

        // (3.6) 分配内存，并将每个程序段的内容从二进制文件的内存（from, from+end）
复制到进程的内存（la, la+end）
        end = ph->p_va + ph->p_filesz;
        // (3.6.1) 复制二进制程序的 TEXT/DATA 段
        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;
            // off 是当前所处位置和加载地址的偏移量
            // 大小就是这一页要读取的大小
            // la代表当前页的结尾处
            if (end < la) {
                size -= la - end;
            }
            // 如果在当前页就结束了，则读取的大小有所变化
            if ((ret = load_icode_read(fd, page2kva(page) + off, size,
offset)) != 0) {
                goto bad_cleanup_mmap;
            }
            // 读取到page2kva(page) + off处
            start += size, offset += size;
            // 读取的起始位置变化
        }
        end = ph->p_va + ph->p_memsz;

        // (3.6.2) 构建二进制程序的 BSS 段
        //（用于存放未初始化的全局变量和静态变量）
        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;
            }
            memset(page2kva(page) + off, 0, size);
            start += size;
        }
    }
```

```
    // (4) 构建用户栈内存
    sysfile_close(fd);

    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);
    // 分配了4K

    // (5) 设置当前进程的 mm、sr3，将 CR3 寄存器设置为页目录的物理地址
    mm_count_inc(mm);
    // 跟踪一个内存管理结构被多少个实体引用或持有
    current->mm = mm;
    // 将当前进程的内存管理结构设置为 mm
    current->cr3 = PADDR(mm->pgdir);
    // 将当前进程的 cr3 成员设置为 mm->pgdir 的物理地址
    lcr3(PADDR(mm->pgdir));
    // 切换不同进程的页表

    // 设置 argc、argv
    uint32_t argv_size = 0, i;
    for (i = 0; i < argc; i++) {
        argv_size += strnlen(kargv[i], EXEC_MAX_ARG_LEN + 1) + 1;
    }
    // 每次迭代中，它计算了参数字符串的长度（限制在 EXEC_MAX_ARG_LEN + 1 个字符以
内），并将其长度加到 argv_size 中。每个参数后面再加 1 是为了存储字符串结束符（通常是
null 终止符）
    uintptr_t stacktop = USTACKTOP - (argv_size / sizeof(long) + 1) *
sizeof(long);
    // USTACKTOP - 计算得到的参数部分占用的总字节数：这一步是为了计算出参数部分在用
户栈中的起始位置。
    char **uargv = (char **)(stacktop - argc * sizeof(char *));
    // 这个指针数组会存放每个参数的地址
    argv_size = 0;
    for (i = 0; i < argc; i++) {
        uargv[i] = strcpy((char *)(stacktop + argv_size), kargv[i]);
        argv_size += strnlen(kargv[i], EXEC_MAX_ARG_LEN + 1) + 1;
    }
    // 为什么有两次循环，第一次先算大小，第二次赋值
    stacktop = (uintptr_t)uargv - sizeof(int);
    *(int *)stacktop = argc;
    // 放个数

    // (6) 为用户环境设置陷阱帧（trapframe）
    struct trapframe *tf = current->tf;
    // 保持 sstatus
    uintptr_t sstatus = tf->status;
    memset(tf, 0, sizeof(struct trapframe));
    tf->gpr.sp = stacktop;
```

```
    tf->epc = elf->e_entry;
    tf->status = sstatus & ~(SSTATUS_SPP | SSTATUS_SPIE);
    // 中断使能状态 特权级别状态
    ret = 0;

    out:
    return ret;

    bad_cleanup_mmap:
    exit_mmap(mm);
    bad_elf_cleanup_pgdir:
    put_pgdir(mm);
    bad_pgdir_cleanup_mm:
    mm_destroy(mm);
    bad_mm:
    goto out;
}
```

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。）

UNIX的PIPE机制

- 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。
- 在Linux中，管道是一种使用非常频繁的通信机制。从本质上说，管道也是一种文件，但它又和一般的文件有所不同，管道可以克服使用文件进行通信的两个问题，具体表现为：
    - 限制管道的大小。实际上，管道是一个固定大小的缓冲区。在Linux中，该缓冲区的大小为1页，即 4K字节，使得它的大小不像文件那样不加检验地增长。使用单个固定缓冲区也会带来问题，比如在写管道时可能变满，当这种情况发生时，随后对管道的write()调用将默认地被阻塞，等待某些数据被读取，以便腾出足够的空间供write()调用写。
    - 读取进程也可能工作得比写进程快。当所有当前进程数据已被读取时，管道变空。当这种情况发生时，一个随后的read()调用将默认地被阻塞，等待某些数据被写入，这解决了read()调用返回文件结束的问题。

实现

- 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

1. **数据结构定义**：

    a. **pipe_t 结构体**：表示管道的基本信息，包括读写指针、缓冲区等。

```
struct pipe_t {
    char buffer[PIPE_SIZE]; // 管道缓冲区
    int read_pos;           // 读指针
    int write_pos;          // 写指针
    semaphore_t sem_read;   // 读信号量
    semaphore_t sem_write;  // 写信号量
    mutex_t mutex;          // 互斥锁
};
```

    b. **file_t 结构体的修改**：用于在文件描述符表中表示管道。

```
struct file_t {
    int type;       // 文件类型，例如普通文件、管道等
    union {
        // 其他文件类型的定义
        struct pipe_t* pipe; // 管道文件
    };
};
```

2. **接口定义**：

    a. **pipe_create()**：创建管道并返回相关的文件描述符。

```
int pipe_create(struct file_t** read_end, struct file_t** write_end);
```

    b. **pipe_read()**：从管道中读取数据。

```
int pipe_read(struct file_t* file, void* buffer, size_t size);`
```

c. **pipe_write()**：向管道中写入数据。

```
int pipe_write(struct file_t* file, const void* buffer, size_t size);`
```

d. **pipe_close()**：关闭管道。

```
int pipe_close(struct file_t* file);
```

3. **同步互斥问题处理**：

a. **信号量和互斥锁**：使用信号量和互斥锁确保在多线程或多进程环境下对管道的安全访问。

b. **管道缓冲区管理**：通过读写指针的合理管理，避免读写指针冲突和数据溢出。

c. **原子操作**：在关键代码段使用原子操作，确保对共享数据的原子性访问。

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。）

1. 软链接和硬链接

- 链接简单说实际上是一种文件共享的方式，是POSIX中的概念，主流文件系统都支持链接文件。
- 链接可以简单地理解为 Windows 中常见的快捷方式（或是 OS X 中的替身），Linux 中常用它来解决一 些库版本的问题，通常也会将一些目录层次较深的文件链接到一个更易访问的目录中。在这些用途上， 我们通常会使用到软链接（也称符号链接）
- 硬链接：与普通文件没什么不同，inode 都指向同一个文件在硬盘中的区块
    - 硬链接，以文件副本的形式存在。但不占用实际空间。
    - 不允许给目录创建硬链接。
    - 硬链接只有在同一个文件系统中才能创建。
    - 删除其中一个硬链接文件并不影响其他有相同 inode 号的文件。
- 软链接：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。
    - 软链接是存放另一个文件的路径的形式存在。
    - 软链接可以跨文件系统 ，硬链接不可以。
    - 软链接可以对一个不存在的文件名进行链接，硬链接必须要有源文件。
    - 软链接可以对目录进行链接。

2. 实现思路

在设计支持UNIX的软连接和硬连接机制的系统中，需要定义以下数据结构和接口。

数据结构：

1. Inode 结构体：

```
struct inode {
    // 其他 inode 相关信息
    // ...

    // 硬链接计数器
    int link_count;

    // 软链接信息
    struct softlink {
        char target_path[MAX_PATH_LEN];  // 软链接目标路径
    } softlink;

    // 添加其他可能需要的字段
    // ...
};
```

2. File 结构体：

```
struct file {
    // 文件相关信息
    // ...

    // 文件描述符类型（普通文件、目录、软链接等）
    int file_type;

    // 若是软链接，存储软链接信息
    struct softlink {
        char target_path[MAX_PATH_LEN];  // 软链接目标路径
    } softlink;

    // 添加其他可能需要的字段
    // ...
};
```

接口

1. 创建硬链接的接口：

```
int hardlink(const char *old_path, const char *new_path);
```

语义： 创建一个新的硬链接，将文件系统中的 old_path 对应的 Inode 添加到 new_path 的目录项中，并且增加相应的硬链接计数。

2. 创建软链接的接口：

```
int symlink(const char *target_path, const char *link_path);
```

语义： 创建一个新的软链接，将软链接目标路径 target_path 存储在 link_path 的 Inode 中，并将 link_path 添加到相应的目录项。

3. 读取软链接目标的接口：

```
size_t readlink(const char *path, char *buf, size_t bufsize);
```

语义： 读取软链接文件 path 中存储的目标路径，将其复制到 buf 中，最多复制 bufsize 个字节。

4. 删除硬链接的接口：

```
int unreadlink(const char *path);
```

语义： 删除文件系统中 path 对应的硬链接，减少相应 Inode 的硬链接计数。当计数为零时，释放相应的 Inode 和数据块。

5. 删除软链接的接口：

```
int unsymlink(const char *path);
```

语义： 删除文件系统中 path 对应的软链接。

同步互斥问题处理：

- 硬链接计数的原子性操作： 对于硬链接计数的修改需要进行原子性操作，避免多个进程同时增加或减少计数导致不一致的情况。
- 目录项的同步：在进行链接或删除操作时，需要对目录项的修改进行同步，以防止多个进程同时对同一目录进行操作。
- 软链接目标路径的同步： 在读取软链接目标路径或修改软链接目标路径时，需要对软链接的 Inode 进行同步，避免竞争条件。