《操作系统》实验报告

实验题目	文件系统设计与实现

一、实验目的

1. 了解基本的文件系统系统调用的实现方法；

2. 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；

3. 了解文件系统抽象层-VFS 的设计与实现；

二、实验项目内容

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，重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。

1. 完成读文件操作的实现。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。

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 -j 16。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。

三、实验过程或算法（源程序）

1. 前期准备：连接到 WSL，进入 Ubuntu，启动 Docker 并进入容器，从仓库中克隆不含答案的项目；

2. 修改 Makefile：

```
LAB1     := -DLAB1_EX2 -DLAB1_EX3 #-D_SHOW_100_TICKS -D_SHOW_SERIAL_INPUT
LAB2     := -DLAB2_EX1 -DLAB2_EX2 -DLAB2_EX3
LAB3     := -DLAB3_EX1 -DLAB3_EX2
LAB4     := -DLAB4_EX1 -DLAB4_EX2
```

需要将 LAB1~LAB4 全部放开，才能够成功编译；

3. 实现读文件操作：

```c
#ifdef LAB4_EX1
//LAB4: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) {
        if(nblks){
        size = SFS_BLKSIZE - blkoff;
        }else{
        size  = 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;
    }
#endif
```

在 kern/fs/sfs/sfs_inode.c 文件下，使用 sfs_io_nolock()函数进行读取文件操作。过程如下：先计算一些辅助变量，并处理一些特殊情况，然后赋值 sfs_buf_op 为 sfs_rbuf，sfs_block_op 为 sfs_rblock，设置读取的函数操作。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。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。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。设计思路详见练习 1。

4. 基于文件系统的执行程序机制：

```c
#ifdef LAB4_EX2
/* LAB4:EXERCISE2 YOUR CODE  HINT:how to load the file with handler fd  in to process's memory? how to setup argc/argv?
 * MACROs or Functions:
 *  mm_create        - create a mm
 *  setup_pgdir      - setup pgdir in mm
 *  load_icode_read  - read raw data content of program file
 *  mm_map           - build new vma
 *  pgdir_alloc_page - allocate new memory for  TEXT/DATA/BSS/stack parts
 *  lcr3             - update Page Directory Addr Register -- CR3
 */
/* (1) create a new mm for current process
 * (2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT
 * (3) copy TEXT/DATA/BSS parts in binary to memory space of process
 *    (3.1) read raw data content in file and resolve elfhdr
 *    (3.2) read raw data content in file and resolve proghdr based on info in elfhdr
 *    (3.3) call mm_map to build vma related to TEXT/DATA
 *    (3.4) callpgdir_alloc_page to allocate page for TEXT/DATA, read contents in file
 *          and copy them into the new allocated pages
 *    (3.5) callpgdir_alloc_page to allocate pages for BSS, memset zero in these pages
 * (4) call mm_map to setup user stack, and put parameters into user stack
 * (5) setup current process's mm, cr3, reset pgidr (using lcr3 MARCO)
 * (6) setup trapframe for user environment (You have done in LAB3)
 * (7) store argc and kargv to a0 and a1 register in trapframe
 * (8) if up steps failed, you should cleanup the env.
 */
    if (current->mm != NULL) {
        panic("load_icode: current->mm must be empty.\n");
    }
```

```c
    int ret = -E_NO_MEM;
    struct mm_struct *mm;
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }
    struct __elfhdr ___elfhdr__;
    struct elfhdr32 __elf, *elf = &__elf;
    if ((ret = load_icode_read(fd, &___elfhdr__, sizeof(struct __elfhdr), 0)) != 0) {
        goto bad_elf_cleanup_pgdir;
    }
    _load_elfhdr((unsigned char*)&___elfhdr__, &__elf);

    if (elf->e_magic != ELF_MAGIC) {
        ret = -E_INVAL_ELF;
        goto bad_elf_cleanup_pgdir;
    }

    struct proghdr _ph, *ph = &_ph;
    uint32_t vm_flags, phnum;
    uint32_t perm = 0;
    struct Page *page;
    for (phnum = 0; phnum < elf->e_phnum; phnum ++) {
    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 ;
    }
```

```c
    if (ph->p_filesz > ph->p_memsz) {
        ret = -E_INVAL_ELF;
        goto bad_cleanup_mmap;
    }
    vm_flags = 0;
    //ptep_set_u_read(&perm);
    perm |= PTE_U;
    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;
    if (vm_flags & VM_WRITE) perm |= PTE_W;

    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_2N(start, PGSHIFT);

    end = ph->p_va + ph->p_filesz;
    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;
        }
        if ((ret = load_icode_read(fd, page2kva(page) + off, size, offset)) != 0) {
            goto bad_cleanup_mmap;
        }
        fence_i(page2kva(page)+off, size);
        start += size, offset += size;
    }
```

```c
    end = ph->p_va + ph->p_memsz;

    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);
        fence_i(page2kva(page) + off, 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);
        fence_i(page2kva(page) + off, size);
        start += size;
    }
}
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;
}
```

```c
mm_count_inc(mm);
current->mm = mm;
current->cr3 = PADDR(mm->pgdir);
lcr3(PADDR(mm->pgdir));
uintptr_t stacktop = USTACKTOP - argc * PGSIZE;
char **uargv = (char **)(stacktop - argc * sizeof(char *));
int i;
for (i = 0; i < argc; i ++) {
    uargv[i] = strcpy((char *)(stacktop + i * PGSIZE), kargv[i]);
}
struct trapframe *tf = current->tf;
memset(tf, 0, sizeof(struct trapframe));
tf->tf_era = elf->e_entry;
tf->tf_regs.reg_r[LOONGARCH_REG_SP] = USTACKTOP;
uint32_t status = 0;
status |= PLV_USER; // set plv=3(User Mode)
status |= CSR_CRMD_IE;
tf->tf_prmd = status;
tf->tf_regs.reg_r[LOONGARCH_REG_A0] = argc;
tf->tf_regs.reg_r[LOONGARCH_REG_A1] = (uint32_t)uargv;
ret = 0;
out:
    return ret;
bad_cleanup_mmap:
    panic("bad_cleanup_mmap");
    exit_mmap(mm);
bad_elf_cleanup_pgdir:
    panic("bad_elf_cleanup_pgdir");
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    panic("bad_pgdir_cleanup_mm");
    mm_destroy(mm);
bad_mm:
    panic("bad_mm");
    goto out;
#endif
```

在 kern/process/proc.c 文件下，通过 load_icode()函数，实现基于文件系统的执行程序机制。设计思路见练习 2。

5. 完成以上代码编写后，编译运行该项目，运行结果如下：

```
root@DESKTOP-ELVOHAA:/mnt/c/Users/Jackson1125/lab4# cd ucore-loongarch32
root@DESKTOP-ELVOHAA:/mnt/c/Users/Jackson1125/lab4/ucore-loongarch32# make
  DEP kern/fs/devs/dev_stdout.c
  DEP kern/fs/devs/dev_stdin.c
  DEP kern/fs/devs/dev_disk0.c
  DEP kern/fs/devs/dev.c
  DEP kern/fs/sfs/sfs_lock.c
```

```
root@DESKTOP-ELVOHAA:/mnt/c/Users/Jackson1125/lab4/ucore-loongarch32# make qemu -j 164000+0 records in
4000+0 records out
2048000 bytes (2.0 MB, 2.0 MiB) copied, 0.825767 s, 2.5 MB/s
create obj/initrd.img (obj/rootfs) successfully.
loongson32_init: num_nodes 1
loongson32_init: node 0 mem 0x2000000
++setup timer interrupts
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xA0000120 (phys)
  etext 0xA0022000 (phys)
  edata 0xA025CC20 (phys)
  end   0xA025FF00 (phys)
Kernel executable memory footprint: 2296KB
memory management: default_pmm_manager
memory map:
    [A0000000, A2000000]

freemem start at: A02A0000
free pages: 00001D60
## 00000020
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
check_slab() succeeded!
kmalloc_init() succeeded!
check_vma_struct() succeeded!
check_pgfault() succeeded!
check_vmm() succeeded.
sched class: stride_scheduler
proc_init succeeded
Initrd: 0xa005ff50 - 0xa0253f4f, size: 0x001f4000, magic: 0x2f8dbe2a
ramdisk_init(): initrd found, magic: 0x2f8dbe2a, 0x00000fa0 secs
sfs: mount: 'simple file system' (352/148/500)
vfs: mount disk0.
kernel_execve: pid = 2, name = "sh".
user sh is running!!!
```

实验算法函数源代码：

练习 1. 完成读文件操作的实现

```
#ifdef LAB4_EX1
   //LAB4: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) {
            if(nblks){
            size = SFS_BLKSIZE - blkoff;
```

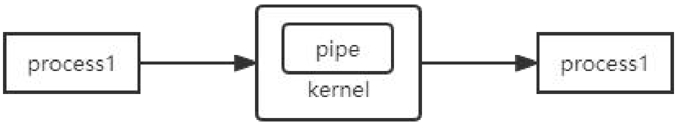
```c
            }else{
            size  = endpos - offset;
            }
            if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
                goto out;
            }
            if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0) {
                goto out;
            }
            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;
        }
    #endif
```

“UNIX 的 PIPE 机制”概要设计方案：

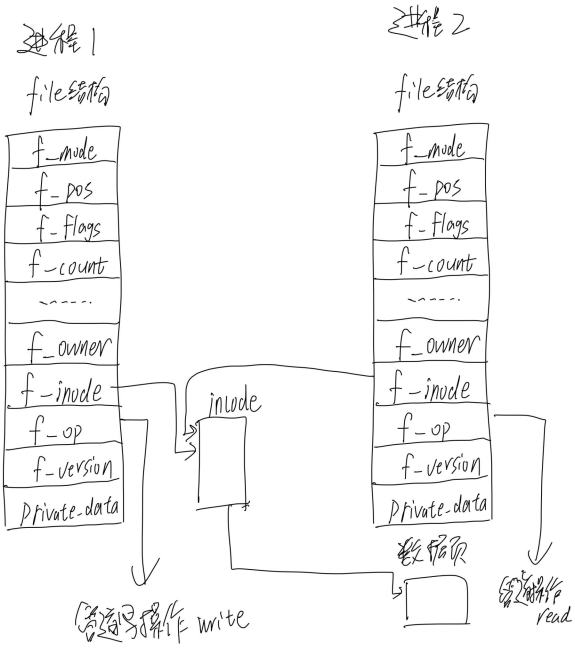
主要通过管道 PIPE 实现，管道可用于具有亲缘关系进程间的通信，有名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。管道是由内核管理的一个缓冲区，相当于我们放入内存中的一个纸条。管道的一端连接一个进程的输出。这个进程会向管道中放入信息。管道的另一端连接一个进程的输入，这个进程取出被放入管道的信息。一个缓冲区不需要很大，它被设计成为环形的数据结构，以便管道可以被循环利用。当管道中没有信息的话，从管道中读取的进程会等待，直到另一端的进程放入信息。当管道被放满信息的时候，尝试

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。



UNIX 的 PIPE 机制的主要函数是管道读函数 pipe_read() 和管道写函数 pipe_wrtie()：

管道写函数通过将字节复制到 VFS 索引节点指向的物理内存而写入数据，而管道读函数则通过复制物理内存中的字节而读出数据。当然，内核必须利用一定的机制同步对管道的访问，为此，内核需要使用锁、等待队列和信号。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，如下图：



当写进程向管道中写入时，它利用标准的库函数 write()，系统根据库函数传递的文件描述符，可找到该文件的 file 结构。file 结构中指定了用来进行写操作的函数（即写入函数）地址，于是，内核调用该函数完成写操作。写入函数在向内存中写入数据之前，必须首先检查 VFS 索引节点中的信息，进行实际的内存复制工作；

写入函数首先锁定内存，然后从写进程的地址空间中复制数据到内存。否则，写入进程就休眠在 VFS 索引节点的等待队列中，接下来，内核将调用调度程序，而调度程序会选择其他进程运行。写入进程实际处于可中断的等待状态，当内存中有足够的空间可以容纳写入数据，或内存被解锁时，读取进程会唤醒写入进程，这时，写入进程将接收到信号。当数据写入内存之后，内存被解锁，而所有休眠在索引节点的读取进程会被唤醒。

管道的读取过程和写入过程类似。

练习 2. 完成基于文件系统的执行程序机制的实现

```
#ifdef LAB4_EX2
```

```
    /* LAB4:EXERCISE2 YOUR CODE  HINT:how to load the file with handler fd  in
to process's memory? how to setup argc/argv?
     * MACROs or Functions:
     *  mm_create        - create a mm
     *  setup_pgdir      - setup pgdir in mm
     *  load_icode_read  - read raw data content of program file
     *  mm_map           - build new vma
     *  pgdir_alloc_page - allocate new memory for  TEXT/DATA/BSS/stack parts
     *  lcr3             - update Page Directory Addr Register -- CR3
     */
    /* (1) create a new mm for current process
     * (2) create a new PDT, and mm->pgdir= kernel virtual addr of PDT
     * (3) copy TEXT/DATA/BSS parts in binary to memory space of process
     *    (3.1) read raw data content in file and resolve elfhdr
     *    (3.2) read raw data content in file and resolve proghdr based on info
in elfhdr
     *    (3.3) call mm_map to build vma related to TEXT/DATA
     *    (3.4) callpgdir_alloc_page to allocate page for TEXT/DATA, read contents
in file
     *          and copy them into the new allocated pages
     *    (3.5) callpgdir_alloc_page to allocate pages for BSS, memset zero in
these pages
     * (4) call mm_map to setup user stack, and put parameters into user stack
     * (5) setup current process's mm, cr3, reset pgidr (using lcr3 MARCO)
     * (6) setup trapframe for user environment (You have done in LAB3)
     * (7) store argc and kargv to a0 and a1 register in trapframe
     * (8) if up steps failed, you should cleanup the env.
     */
        if (current->mm != NULL) {
            panic("load_icode: current->mm must be empty.\n");
        }

        int ret = -E_NO_MEM;
        struct mm_struct *mm;
        if ((mm = mm_create()) == NULL) {
            goto bad_mm;
        }
        if (setup_pgdir(mm) != 0) {
            goto bad_pgdir_cleanup_mm;
        }
        struct __elfhdr ___elfhdr__;
        struct elfhdr32 __elf, *elf = &__elf;
        if ((ret = load_icode_read(fd, &___elfhdr__, sizeof(struct __elfhdr),
0)) != 0) {
```

```
            goto bad_elf_cleanup_pgdir;
        }
        _load_elfhdr((unsigned char*)&___elfhdr__, &__elf);

        if (elf->e_magic != ELF_MAGIC) {
            ret = -E_INVAL_ELF;
            goto bad_elf_cleanup_pgdir;
        }

        struct proghdr _ph, *ph = &_ph;
        uint32_t vm_flags, phnum;
        uint32_t perm = 0;
        struct Page *page;
        for (phnum = 0; phnum < elf->e_phnum; phnum ++) {
        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;
        }
        vm_flags = 0;
        //ptep_set_u_read(&perm);
        perm |= PTE_U;
        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;
        if (vm_flags & VM_WRITE) perm |= PTE_W;

        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_2N(start, PGSHIFT);

        end = ph->p_va + ph->p_filesz;
        while (start < end) {
```

```c
            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;
            }
            if ((ret = load_icode_read(fd, page2kva(page) + off, size, offset)) !=
0) {
                goto bad_cleanup_mmap;
            }
            fence_i(page2kva(page)+off, size);
            start += size, offset += size;
        }

        end = ph->p_va + ph->p_memsz;

        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);
            fence_i(page2kva(page) + off, 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);
            fence_i(page2kva(page) + off, size);
            start += size;
```

```c
        }
    }
    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;
    }

    mm_count_inc(mm);
    current->mm = mm;
    current->cr3 = PADDR(mm->pgdir);
    lcr3(PADDR(mm->pgdir));
    uintptr_t stacktop = USTACKTOP - argc * PGSIZE;
    char **uargv = (char **)(stacktop - argc * sizeof(char *));
    int i;
    for (i = 0; i < argc; i ++) {
        uargv[i] = strcpy((char *)(stacktop + i * PGSIZE), kargv[i]);
    }
    struct trapframe *tf = current->tf;
    memset(tf, 0, sizeof(struct trapframe));
    tf->tf_era = elf->e_entry;
    tf->tf_regs.reg_r[LOONGARCH_REG_SP] = USTACKTOP;
    uint32_t status = 0;
    status |= PLV_USER; // set plv=3(User Mode)
    status |= CSR_CRMD_IE;
    tf->tf_prmd = status;
    tf->tf_regs.reg_r[LOONGARCH_REG_A0] = argc;
    tf->tf_regs.reg_r[LOONGARCH_REG_A1] = (uint32_t)uargv;
    ret = 0;
    out:
        return ret;
    bad_cleanup_mmap:
        panic("bad_cleanup_mmap");
        exit_mmap(mm);
    bad_elf_cleanup_pgdir:
        panic("bad_elf_cleanup_pgdir");
        put_pgdir(mm);
    bad_pgdir_cleanup_mm:
        panic("bad_pgdir_cleanup_mm");
        mm_destroy(mm);
    bad_mm:
        panic("bad_mm");
        goto out;
```

```
#endif
```

“UNIX 的硬链接和软链接机制”如下：

观察到保存在磁盘上的 inode 信息均存在一个 nlinks 变量用千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。

四、实验结果及分析

实验结果截图：

```
root@DESKTOP-ELVOHAA:/mnt/c/Users/Jackson1125/lab4/ucore-loongarch32# make qemu -j 164000+0 records in
4000+0 records out
2048000 bytes (2.0 MB, 2.0 MiB) copied, 0.825767 s, 2.5 MB/s
create obj/initrd.img (obj/rootfs) successfully.
loongson32_init: num_nodes 1
loongson32_init: node 0 mem 0x2000000
++setup timer interrupts
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xA0000120 (phys)
  etext 0xA0022000 (phys)
  edata 0xA025CC20 (phys)
  end   0xA025FF00 (phys)
Kernel executable memory footprint: 2296KB
memory management: default_pmm_manager
memory map:
    [A0000000, A2000000]

freemem start at: A02A0000
free pages: 00001D60
## 00000020
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
check_slab() succeeded!
kmalloc_init() succeeded!
check_vma_struct() succeeded!
check_pgfault() succeeded!
check_vmm() succeeded.
sched class: stride_scheduler
proc_init succeeded
Initrd: 0xa005ff50 - 0xa0253f4f, size: 0x001f4000, magic: 0x2f8dbe2a
ramdisk_init(): initrd found, magic: 0x2f8dbe2a, 0x00000fa0 secs
sfs: mount: 'simple file system' (352/148/500)
vfs: mount disk0.
kernel_execve: pid = 2, name = "sh".
user sh is running!!!
```