Lab8 实验报告

第一部分：实验目的

本次 Lab8 的核心目标是把系统“跑用户程序 + 文件读写”这两条关键链路补齐到可工作的状态：

Exercise 1（文件系统/SFS）：实现 sfs_io_nolock() 的读写路径，支持任意 offset/len 的文件读写，正确处理“首块非对齐 / 中间整块 / 尾块非对齐”的组合情况。

Exercise 2（进程/exec）：实现 load_icode(fd, argc, kargv)，完成 ELF 可执行文件装载、用户栈参数构造（argc/argv）、以及返回用户态前的 trapframe 初始化。

第二部分：代码实现

A. Exercise 1：SFS 文件读写 sfs_io_nolock() 设计与实现

A1. 设计思路

SFS 以块（SFS_BLKSIZE）为单位组织数据。一次读写请求给出 (offset, len)，可能跨越多个块，也可能落在块的中间。因此实现上拆成三段最稳妥：

首块非对齐：若 offset % BLKSIZE != 0，先把当前块剩余部分读/写完，使位置推进到块边界。

中间整块：对齐后用块读写接口一次处理整块（循环处理多个块）。

尾块非对齐：如果最后还剩不足一块的数据，用“缓冲区读写接口”处理部分块。

其中：

sfs_bmap_load_nolock(sfs, sin, file_blk_index, &ino) 负责把“文件内第几个数据块”映射到“磁盘上的块号 ino”（必要时分配）。

读写接口二选一：

部分块：sfs_rbuf/sfs_wbuf（本实现通过函数指针 sfs_buf_op 统一）

整块：sfs_rblock/sfs_wblock（通过 sfs_block_op 统一）

A2. 代码实现

(1) 计算起始块号、跨越块数、块内偏移

```
uint32_t blkno = offset / SFS_BLKSIZE;                  // The NO. of Rd/Wr begin block
uint32_t nblks = endpos / SFS_BLKSIZE - blkno;    // The size of Rd/Wr blocks

char *bufp = (char *)buf;
off_t pos = offset;

blkoff = pos % SFS_BLKSIZE;
```

blkno：当前偏移落在哪个文件数据块内（文件块索引）。

nblks：从起始块到 endpos 之间有多少“完整跨越的块”（注意这里后续会配合首块非对齐逻辑递减）。

blkoff：块内偏移，决定是否需要先处理“首块非对齐”。

(2) 首块非对齐：只读/写当前块剩余部分

```
if (blkoff != 0) {
    size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - pos);
    if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
        goto out;
```

```
        }
        if ((ret = sfs_buf_op(sfs, bufp, size, ino, blkoff)) != 0) {
                goto out;
        }
        alen += size;
        bufp += size;
        pos += size;
        blkno++;
        if (nblks != 0) {
                nblks--;
        }
        blkoff = 0;
}
```

size = ...：本次在“首块”最多读/写到块末尾；如果总请求只落在同一个块内，则用 endpos - pos 作为 size。
sfs_bmap_load_nolock(...)：把文件的第 blkno 块映射到磁盘块号 ino（写时可能触发分配）。
sfs_buf_op(..., blkoff)：从块内偏移 blkoff 开始做部分块读/写。
更新推进量：alen/bufp/pos 同步推进；blkno++ 进入下一块；若原本跨越多个块则 nblks--。
blkoff = 0：后续进入对齐状态。

(3) 中间整块：循环按块读/写

```
while (nblks != 0) {
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
                goto out;
        }
        if ((ret = sfs_block_op(sfs, bufp, ino, 1)) != 0) {
                goto out;
        }
        alen += SFS_BLKSIZE;
        bufp += SFS_BLKSIZE;
        pos += SFS_BLKSIZE;
        blkno++;
        nblks--;
}
```

每次循环处理 1 个整块：先映射得到磁盘块号 ino，再用 sfs_block_op(..., 1) 读/写一个块。
每次推进固定 SFS_BLKSIZE 字节，同时推进 blkno 和递减 nblks，直到整块段完成。

(4) 尾块非对齐：处理最后不足一块的剩余字节

```
if (pos < endpos) {
        size = endpos - pos;
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
                goto out;
```

```
        }
        if ((ret = sfs_buf_op(sfs, bufp, size, ino, 0)) != 0) {
                goto out;
        }
        alen += size;
}
```

若还有剩余（pos < endpos），说明需要处理尾块部分数据。
这时从该块起始位置偏移 0 开始读/写 size=endpos-pos 字节即可（因为中间整块已经对齐推进过了）。
同样通过 bmap 找块号、通过 buf_op 做部分块操作。

(5) 统一出口：返回实际读写长度，并维护文件大小/dirty

```
out:
        *alenp = alen;
        if (offset + alen > sin->din->size) {
                sin->din->size = offset + alen;
                sin->dirty = 1;
        }
        return ret;
```

*alenp = alen：向上层返回实际完成的读写字节数。
写入可能导致文件增长：若 offset+alen 超过原 din->size，更新文件大小并标记 inode 脏（需要后续落盘）。

B. Exercise 2：ELF 装载与用户栈构造 load_icode() 设计与实现

B1. 设计思路

exec 的本质是“丢弃旧用户地址空间，建立一个全新的用户地址空间并跳到新程序入口执行”。实现上分为 7 个阶段：

前置条件：current->mm 必须为空（上层 do_execve 已释放旧 mm）。
创建 mm 并建立页目录（复制内核映射）。
读取 ELF Header，校验魔数。
遍历 Program Header：对 PT_LOAD 段建立 VMA、分配页、把文件内容拷入内存，并对 BSS 清零。
建立用户栈 VMA，并预分配若干栈页。
切换到新页表后，将 kargv 拷贝到用户栈，构造用户态 argv[] 指针数组。
初始化 trapframe（sp/a0/a1/epc/status），确保 sret 返回到用户态执行。
同时必须保证失败路径能正确释放资源（exit_mmap/put_pgdir/mm_destroy）并恢复到内核页表。

B2. 代码分段与解释

(1) 前置检查 + 创建 mm + 建立 pgdir

```
if (current->mm != NULL)
```

```
{
    panic("load_icode: current->mm must be empty.\n");
}

int ret = -E_NO_MEM;
struct mm_struct *mm = NULL;

// (1) create a new mm
if ((mm = mm_create()) == NULL)
{
    goto out;
}
// (2) setup new pgdir
if ((ret = setup_pgdir(mm)) != 0)
{
    goto bad_pgdir_cleanup_mm;
}
```

current->mm 非空说明 exec 流程不符合预期（容易造成地址空间泄漏/混乱），直接 panic。
mm_create() 创建新的内存管理结构；setup_pgdir(mm) 分配新的页目录并拷贝内核部分映射，为后续映射用户段做准备。
使用 goto 组织统一错误清理路径，避免遗漏释放。

(2) 读取并校验 ELF Header

```
struct elfhdr elf;
if ((ret = load_icode_read(fd, &elf, sizeof(elf), 0)) != 0)
{
    goto bad_elf_cleanup_pgdir;
}
if (elf.e_magic != ELF_MAGIC)
{
    ret = -E_INVAL_ELF;
    goto bad_elf_cleanup_pgdir;
}
```

load_icode_read 封装了 seek + read，保证从指定偏移读取指定长度。
校验 ELF_MAGIC，不合法则返回 -E_INVAL_ELF，避免把非 ELF 文件当成可执行程序映射进内存。

(3) 遍历 Program Header：筛选 PT_LOAD 并计算 VMA/PTE 权限

```
struct Page *page = NULL;
for (uint32_t ph_idx = 0; ph_idx < elf.e_phnum; ph_idx++)
{
    struct proghdr ph;
```

```
    off_t phoff = elf.e_phoff + ph_idx * sizeof(struct proghdr);
    if ((ret = load_icode_read(fd, &ph, sizeof(ph), 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;
    }

    uint32_t vm_flags = 0;
    uint32_t perm = PTE_U | PTE_V;
    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_READ)   { perm |= PTE_R; }
    if (vm_flags & VM_WRITE) { perm |= (PTE_W | PTE_R); }
    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;
    }
    ...
}
```

ph.p_type != ELF_PT_LOAD：只装载可映射段，其余段跳过。
filesz <= memsz：ELF 基本合法性检查，防止越界。
将 ELF 段权限映射到：
VMA 级别：VM_READ/VM_WRITE/VM_EXEC
PTE 级别：PTE_R/PTE_W/PTE_X，并始终带上 PTE_U|PTE_V 使其成为有效的用户页映射。
mm_map(mm, va, memsz, flags, NULL)：为该段建立 VMA（注意用 memsz 覆盖 BSS）。

(4) 装载文件内容：按页分配并把文件拷入页内（处理页内偏移）

```
uintptr_t start = ph.p_va;
uintptr_t end_file = ph.p_va + ph.p_filesz;
uintptr_t la = ROUNDDOWN(start, PGSIZE);
off_t file_pos = ph.p_offset;
```

```
// (3.4) load file-backed part
while (start < end_file)
{
    if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL)
    {
        ret = -E_NO_MEM;
        goto bad_cleanup_mmap;
    }
    size_t page_off = start - la;
    size_t copy_sz = PGSIZE - page_off;
    uintptr_t la_next = la + PGSIZE;
    if (end_file < la_next)
    {
        copy_sz -= la_next - end_file;
    }
    if ((ret = load_icode_read(fd, page2kva(page) + page_off, copy_sz, file_pos)) != 0)
    {
        goto bad_cleanup_mmap;
    }
    start += copy_sz;
    file_pos += copy_sz;
    la = la_next;
}
```

la = ROUNDDOWN(start, PGSIZE)：段起始地址可能不是页对齐的，所以需要从所在页的页首开始分配。
每次循环分配一页 pgdir_alloc_page(mm->pgdir, la, perm)，确保对应虚拟页存在映射。
page_off：本次写入在页内的偏移。
copy_sz：本页能容纳的剩余字节数；若最后一页不足整页，则裁剪到 end_file。
load_icode_read(..., page2kva(page) + page_off, copy_sz, file_pos)：把文件内容直接读到该页的内核映射地址中，实现“文件 -> 内存”。

(5) BSS 清零：处理“部分页剩余清零”+“后续整页清零”

```
uintptr_t end_mem = ph.p_va + ph.p_memsz;
if (start < la)
{
    if (start == end_mem)
    {
        continue;
    }
    size_t off = start + PGSIZE - la;
    size_t zero_sz = PGSIZE - off;
    if (end_mem < la)
```

```
        {
            zero_sz -= la - end_mem;
        }
        memset(page2kva(page) + off, 0, zero_sz);
        start += zero_sz;
    }
    while (start < end_mem)
    {
        if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL)
        {
            ret = -E_NO_MEM;
            goto bad_cleanup_mmap;
        }
        size_t off = start - la;
        size_t zero_sz = PGSIZE - off;
        uintptr_t la_next = la + PGSIZE;
        if (end_mem < la_next)
        {
            zero_sz -= la_next - end_mem;
        }
        memset(page2kva(page) + off, 0, zero_sz);
        start += zero_sz;
        la = la_next;
    }
```

end_mem 是段在内存中的结束位置（包含 BSS）。
第一段 if (start < la) 处理“最后一次文件装载后，当前页剩余空间需要清零”的情况，保证文件尾到页尾这一段变为 0。
第二段 while (start < end_mem) 为纯 BSS 部分分配新页并清零：
分配页保证映射存在；
memset 把该页中属于 BSS 的范围置 0（最后一页同样按 end_mem 裁剪）。

(6) 建立用户栈 VMA，并预分配若干栈页

```
uint32_t vm_flags = VM_READ | VM_WRITE | VM_STACK;
if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL)) != 0)
{
    goto bad_cleanup_mmap;
}
// allocate a few stack pages near the top
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);
```

mm_map 为用户栈区间建立 VMA（包含读写与栈属性）。
预分配多页的目的：避免程序一进入用户态就因为访问栈而触发缺页（某些实验环境未必实现完整按需分配/缺页处理），让 exec 更稳定。

(7) 将新 mm 安装到 current，并切换页表（satp）

```
mm_count_inc(mm);
current->mm = mm;
current->pgdir = PADDR(mm->pgdir);
lsatp(PADDR(mm->pgdir));
```

mm_count_inc(mm)：引用计数管理，防止被过早释放。
current->mm/current->pgdir 更新后，lsatp() 切换到新页表；此后对“用户栈地址”的写入才能落到正确的新地址空间。

(8) 构造用户栈参数：复制字符串 + 写入 argv 指针数组

```
uintptr_t stacktop = USTACKTOP;
uintptr_t uargv[EXEC_MAX_ARG_NUM + 1];
for (int i = argc - 1; i >= 0; i--)
{
    size_t len = strnlen(kargv[i], EXEC_MAX_ARG_LEN) + 1;
    stacktop -= len;
    stacktop = ROUNDDOWN(stacktop, sizeof(uintptr_t));
    if (!copy_to_user(mm, (void *)stacktop, kargv[i], len))
    {
        ret = -E_INVAL;
        goto bad_cleanup_mmap_restore;
    }
    uargv[i] = stacktop;
}
uargv[argc] = 0;
stacktop -= (argc + 1) * sizeof(uintptr_t);
stacktop = ROUNDDOWN(stacktop, sizeof(uintptr_t));
uintptr_t uargv_addr = stacktop;
if (!copy_to_user(mm, (void *)uargv_addr, uargv, (argc + 1) * sizeof(uintptr_t)))
{
    ret = -E_INVAL;
    goto bad_cleanup_mmap_restore;
}
```

采用“从栈顶向下生长”的方式放置参数字符串，符合栈增长方向。
strnlen(...)+1：包含 \0，保证用户态看到的是标准 C 字符串。
ROUNDDOWN(stacktop, sizeof(uintptr_t))：做指针宽度对齐，避免用户态按指针读取时出现未对齐问题。
copy_to_user(mm, user_dst, kernel_src, len)：安全地把内核里的 kargv[i] 拷贝到用户地址

stacktop。
uargv[i] = stacktop：记录每个字符串在用户栈中的地址。
最后把 uargv[] 数组本身也写到用户栈里，并得到 uargv_addr，用于放入 a1。
(9) 初始化 trapframe：设置 sp/a0/a1/epc/status（确保返回用户态）

```
struct trapframe *tf = current->tf;
uintptr_t sstatus = read_csr(sstatus);
memset(tf, 0, sizeof(struct trapframe));
tf->gpr.sp = stacktop;
tf->gpr.a0 = (uintptr_t)argc;
tf->gpr.a1 = (uintptr_t)uargv_addr;
tf->epc = elf.e_entry;
tf->status = sstatus & ~SSTATUS_SPP;
tf->status |= SSTATUS_SPIE;
```

sp = stacktop：用户态栈顶指针。
a0/a1：按调用约定传参（argc、argv）。
epc = elf.e_entry：设置用户程序入口地址。
status：
~SSTATUS_SPP：清除 SPP，表示 sret 返回到 U-mode。
SSTATUS_SPIE：让返回用户态后中断使能状态符合预期。
(10) 统一成功/失败收尾：关闭 fd，失败时恢复到内核页表并释放 mm

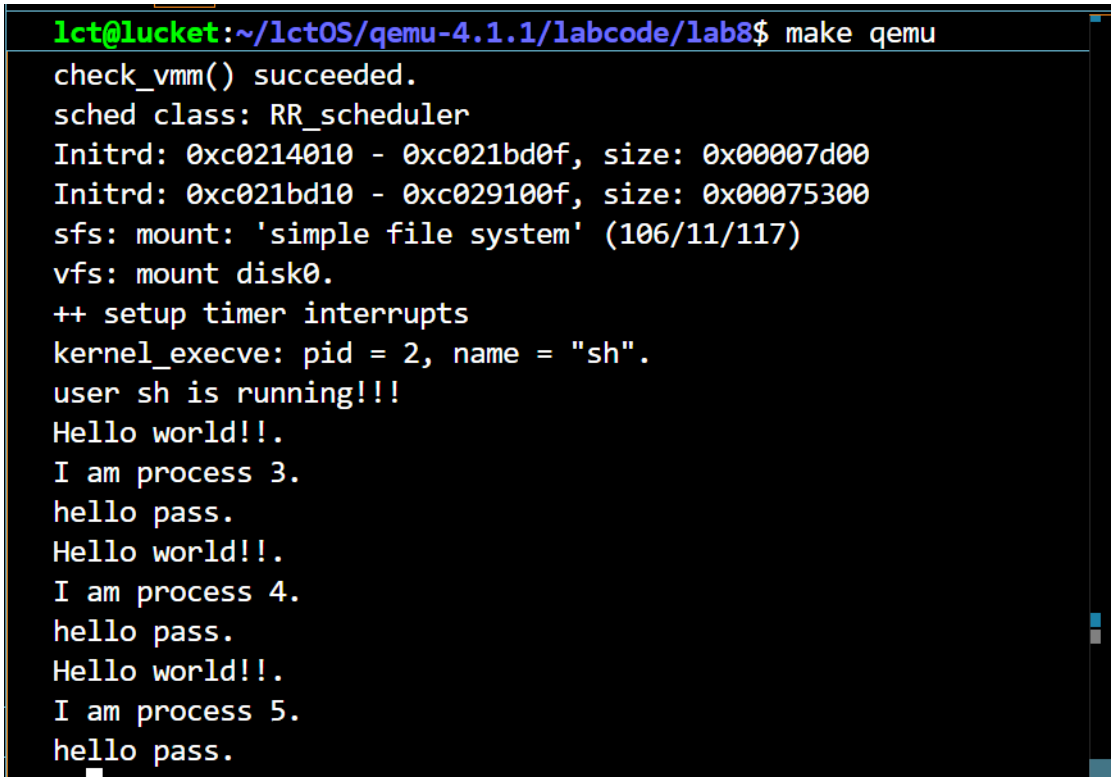
```
out:
    sysfile_close(fd);
    return ret;

bad_cleanup_mmap_restore:
    lsatp(boot_pgdir_pa);
    current->mm = NULL;
    current->pgdir = boot_pgdir_pa;
bad_cleanup_mmap:
    exit_mmap(mm);
bad_elf_cleanup_pgdir:
    put_pgdir(mm);
bad_pgdir_cleanup_mm:
    mm_destroy(mm);
    goto out;
```

sysfile_close(fd) 放在统一出口，保证无论成功失败都能关闭可执行文件 fd。
失败恢复路径中先切回 boot_pgdir_pa，避免继续使用半初始化的用户页表。
释放顺序体现依赖关系：
exit_mmap(mm) 清理 VMA 映射与相关页；
put_pgdir(mm) 释放页目录页；

mm_destroy(mm) 销毁 mm 结构体本身。

第三部分：实验结果和结论

1.终端输出



启动与初始化阶段输出显示：

OpenSBI 启动成功并识别 QEMU Virt 平台，进入内核加载：(THU.CST) os is loading ...

物理内存从 DTB 正确解析（128MB），页表与内存管理自检通过：

check_alloc_page() succeeded!

Page table directory switch succeeded!

check_pgdir() succeeded! / check_boot_pgdir() succeeded!

check_vmm() succeeded.

SFS 文件系统与 VFS 挂载成功：

sfs: mount: 'simple file system' (106/11/117)

vfs: mount disk0.

定时器中断初始化成功：++ setup timer interrupts

用户态执行阶段输出显示：

内核成功执行 sh：

kernel_execve: pid = 2, name = "sh".

user sh is running!!!

用户程序多次执行成功，并能创建新进程：

Hello world!!.

I am process 3. / I am process 4. / I am process 5.

hello pass.

2. 结果分析

Exercise1（sfs_io_nolock）验证点
现象：SFS 能成功挂载，且 sh 以及后续 hello 能从磁盘读出并执行，且 hello pass 连续多次出现。
分析：用户程序的装载依赖从 SFS 中读取可执行文件内容；多次执行与多进程输出说明 inode 的分块读写逻辑能够稳定支持：

非块对齐的读取（例如 ELF header/段内容不一定块对齐）；
跨多块的连续读取；
尾部不足一块的读取。
因此 sfs_io_nolock 的三段式处理（首块非对齐/整块/尾块）在运行时被实际覆盖并表现正确。
Exercise2（load_icode）验证点
现象：kernel_execve: pid = 2, name = "sh". 且随后出现 user sh is running!!!。
分析：这表明 do_execve -> load_icode 成功完成了：

从文件系统读取 ELF；
建立新的用户地址空间（VMA + 页表映射）；
设置用户栈与 argc/argv；
正确初始化 trapframe（epc/sp/a0/a1/status），最终从内核返回用户态并运行入口函数。

3.结论
系统能够完成从内核启动到文件系统挂载的完整流程。
exec 链路可用：sh 能成功加载并进入用户态运行。
基于 SFS 的用户程序读取与执行可重复进行，hello pass 多次通过，说明本次 Lab8 的两个 Exercise 实现达到预期功能目标。

基于 ucore 的 UNIX Pipe 机制设计方案

1 概要设计方案

在 ucore 中实现管道 (Pipe) 机制，核心思想是将管道视为一种特殊的文件 (Everything is a file)。管道本质上是内核管理的一段内存缓冲区，通过虚拟文件系统 (VFS) 接口暴露给用户进程。

- **文件系统层**：管道不对应磁盘上的物理数据块，而是对应内存中的一段环形缓冲区。需要定义一种新的 inode 类型（如 `pipe_inode_info`），并为其实现对应的 `inode_ops`（文件操作函数表）。
- **资源管理**：当通过系统调用创建管道时，内核分配一个 inode 和相应的内存缓冲区。
- **生命周期**：管道拥有读端和写端。调用 `pipe()` 返回两个文件描述符，一个用于读，一个用于写。当所有读端和写端都关闭时，释放缓冲区和 inode 资源。

2 数据结构设计

核心数据结构是 `pipe_inode_info`，它承载了管道的状态、缓冲区指针以及用于同步的信号量和等待队列。该结构体通常作为 inode 的私有数据存在。

Listing 1: pipe_inode_info 结构体定义

```c
#include <sem.h>
#include <wait.h>

#define PIPE_SIZE 4096  // 管道缓冲区大小，通常为一页

struct pipe_inode_info {
    // -- 同步互斥保护 --
    semaphore_t mutex;          // 互斥信号量，保护对缓冲区和索引的原子访问

    // -- 条件变量/等待队列 --
    wait_queue_t reader_queue;  // 读等待队列：当缓冲区为空时，读者在此等待
    wait_queue_t writer_queue;  // 写等待队列：当缓冲区为满时，写者在此等待

    // -- 数据缓冲区 --
    char *buffer;               // 内核缓冲区指针（由 kmalloc 分配）

    // -- 环形缓冲区状态 --
    size_t head;                // 写指针索引（写入位置，单调递增）
    size_t tail;                // 读指针索引（读取位置，单调递增）
```

```
    // 注意:
    // 有效数据量 = head - tail;
    // 实际缓冲区读写位置 = index % PIPE_SIZE

    // -- 引用计数与状态 --
    int nreaders;               // 打开此管道的读端数量
    int nwriters;               // 打开此管道的写端数量
    bool is_closed;             // 标记管道是否已完全关闭
};
```

3 接口语义

需要实现以下核心接口，对应 VFS 的 inode_ops 操作：

3.1 pipe(int fd[])

语义：创建一个新的管道。

- 分配一个新的 inode 和 pipe_inode_info 结构。
- 分配内核缓冲区 (buffer)。
- 初始化互斥信号量 (mutex) 和等待队列 (reader_queue, writer_queue)。
- 在当前进程的文件描述符表中分配两个空闲项。
- fd[0] 关联管道的读端 (Read End)，fd[1] 关联管道的写端 (Write End)。
- 初始化引用计数：nreaders = 1, nwriters = 1。

3.2 pipe_read(struct inode *node, struct iobuf *iob)

语义：从管道读取数据。

1. 获取互斥锁 down(&mutex)。
2. **循环检查**：若缓冲区为空 (head == tail)：
   - 若 nwriters == 0 (写端全关闭)，则返回 0 (EOF)。
   - 否则，当前进程状态设为 PROC_SLEEPING，加入 reader_queue，释放锁并调度 schedule()。唤醒后重新获取锁。
3. 从缓冲区复制数据到用户空间 (iob)，更新 tail 指针。
4. 唤醒 writer_queue 中的写者 (因为腾出了空间)。
5. 释放互斥锁 up(&mutex)。

3.3 pipe_write(struct inode *node, struct iobuf *iob)

语义：向管道写入数据。

1. 获取互斥锁 `down(&mutex)`。
2. **检查**：若 `nreaders == 0` (读端全关闭)，则返回 `-E_PIPE` 错误 (或发送 SIGPIPE 信号)。
3. **循环检查**：若缓冲区已满 (`head - tail >= PIPE_SIZE`)：
   - 当前进程加入 `writer_queue` 并睡眠，释放锁并调度。唤醒后重新获取锁。
4. 从用户空间复制数据到缓冲区，更新 `head` 指针。
5. 唤醒 `reader_queue` 中的读者 (因为有了新数据)。
6. 释放互斥锁 `up(&mutex)`。

3.4 pipe_close(struct inode *node)

语义：关闭管道的一端。

1. 获取互斥锁。
2. 根据关闭的是读端还是写端，递减 `nreaders` 或 `nwriters`。
3. 若 `nreaders` 降为 0，唤醒所有等待的写者 (通知 Broken Pipe)。
4. 若 `nwriters` 降为 0，唤醒所有等待的读者 (通知 EOF)。
5. 若 `nreaders == 0` 且 `nwriters == 0`，释放缓冲区内存和 inode 资源。
6. 释放互斥锁。

4 同步与互斥问题的处理

管道实现了一个典型的 **生产者-消费者 (Producer-Consumer)** 模型。

4.1 互斥访问 (Mutual Exclusion)

问题：多个进程可能同时持有管道的读端或写端 (如父子进程)。并发修改 `head`/`tail` 指针会导致状态不一致。

处理：使用 `semaphore_t mutex`。

- 在进入 `read`/`write`/`close` 关键区前执行 `down(&mutex)`。
- 退出关键区时执行 `up(&mutex)`。
- 保证同一时刻只有一个进程能操作管道内部状态。

4.2 同步等待 (Synchronization)

读空 (Read Empty)：

- 当 `head == tail` 时，读者无法继续。
- **动作**：读者睡眠在 `reader_queue`。
- **唤醒**：写者执行 `write` 后，调用 `wakeup_queue(&reader_queue)`。

写满 (Write Full)：

- 当 `head - tail >= PIPE_SIZE` 时，缓冲区无空间。
- **动作**：写者睡眠在 `writer_queue`。
- **唤醒**：读者执行 `read` 腾出空间后，调用 `wakeup_queue(&writer_queue)`。

4.3 边界情况 (Edge Cases)

- **EOF**：读者读取时，发现缓冲区为空且 `nwriters == 0`，直接返回 0。
- **Broken Pipe**：写者写入时，发现 `nreaders == 0`，报错返回。

UNIX 软链接和硬链接机制设计实现方案

1 摘要

本文档详细描述了在 ucore 操作系统中实现 UNIX 软链接和硬链接机制的设计方案。该方案通过扩展现有文件系统结构，添加必要的数据结构和接口，实现标准的 Unix 链接操作，并考虑了同步互斥问题。

2 设计概述

2.1 硬链接（Hard Link）

- 多个文件名指向同一个 inode
- 共享相同的文件数据块
- 增加 inode 的链接计数（nlinks）
- 不允许对目录创建硬链接

2.2 软链接（Soft Link）

- 创建包含目标路径的特殊文件
- 有自己的 inode，不同于目标文件
- 可以跨文件系统
- 可以指向不存在的文件

3 数据结构设计

3.1 扩展 inode 结构

```c
// 在kern/fs/sfs/sfs.h中添加文件类型定义
#define SFS_TYPE_LINK  0x0400  // 软链接文件类型
#define SFS_MAX_FPATH_LEN 256   // 最大软链接路径长度

// 扩展sfs_disk_inode结构体，支持链接类型
struct sfs_disk_inode {
    uint32_t size;                                  /* 文件大小（字节）
        ↪ */
    uint16_t type;                                  /* 文件类型，
        ↪ SFS_TYPE_* */
    uint16_t nlinks;                                /* 硬链接数量 */
    uint32_t blocks;                                /* 数据块数量 */
    uint32_t direct[SFS_NDIRECT];                   /* 直接数据块 */
    uint32_t indirect;                              /* 间接数据块 */
    char symlink_path[SFS_MAX_FPATH_LEN];           /* 软链接的目标路径
        ↪ */
};
```

3.2 扩展 inode 操作接口

```c
// 在inode_ops结构中添加链接相关操作
struct inode_ops {
    // ... 原有的操作函数 ...
    int (*vop_link) (struct inode *node, const char *new_path);
    int (*vop_symlink) (const char *target_path, const char *link_path);
    int (*vop_readlink) (struct inode *node, struct iobuf *iob);
    int (*vop_followlink) (struct inode *node, struct inode **
        ↪ target_node);
};
```

4 接口设计（语义描述）

4.1 硬链接接口语义

```c
/*
 * vfs_link - 创建硬链接
```

```
 * 语义：在new_path位置创建指向old_path文件的硬链接，增加文件的链接计数
 */
int vfs_link(char *old_path, char *new_path);

/*
 * vop_link - 在文件系统层实现硬链接
 * 语义：在文件系统层面增加指定inode的链接计数，并在新路径创建目录项
 */
int vop_link(struct inode *node, const char *new_path);

/*
 * sfs_link - SFS文件系统中的硬链接实现
 * 语义：SFS文件系统具体的硬链接实现
 */
static int sfs_link(struct inode *node, const char *new_path);
```

4.2 软链接接口语义

```
/*
 * vfs_symlink - 创建软链接
 * 语义：创建一个特殊文件，内容为target_path，路径为link_path
 */
int vfs_symlink(char *target_path, char *link_path);

/*
 * vfs_readlink - 读取软链接内容
 * 语义：读取软链接文件的内容，即其指向的目标路径
 */
int vfs_readlink(char *path, struct iobuf *iob);

/*
 * vop_symlink - 在文件系统层创建软链接
 * 语义：文件系统层面创建包含目标路径的特殊文件
 */
int vop_symlink(const char *target_path, const char *link_path);

/*
 * vop_readlink - 读取软链接内容
```

```
 * 语义：从软链接的inode中读取目标路径
 */
int vop_readlink(struct inode *node, struct iobuf *iob);

/*
 * vop_followlink - 跟随软链接，获取目标文件
 * 语义：解析软链接，返回目标文件的inode
 */
int vop_followlink(struct inode *node, struct inode **target_node);

/*
 * sfs_symlink - SFS文件系统中的软链接实现
 * 语义：SFS文件系统具体的软链接实现
 */
static int sfs_symlink(const char *target_path, const char *link_path);

/*
 * sfs_readlink - 读取软链接内容
 * 语义：SFS文件系统具体的读取软链接内容实现
 */
static int sfs_readlink(struct inode *node, struct iobuf *iob);
```

5 同步互斥处理

在实现链接机制时，需要特别注意同步互斥问题：

```
// 在sfs.h中增加互斥锁定义
struct sfs_fs {
    // ... 原有定义 ...
    semaphore_t mutex_sem;                    /* 用于链接/取消链接和重命名的信号量 */
};
```

解决思路和方法：

1. 在修改 inode 的链接计数（nlinks）时，需要对 inode 加锁（lock_sin），防止并发修改
2. 在进行目录操作（创建或删除目录项）时，需要对文件系统加锁（lock_sfs_fs），确保目录一致性

3. 在创建或删除链接时，同时获取 inode 锁和文件系统锁，避免死锁

4. 实现循环引用检测机制，防止软链接或硬链接形成循环

6 循环检测

为了避免软链接和硬链接形成循环，需要实现路径循环检测：

```
// 在路径解析时检测循环
static int
check_symlink_loop(struct inode *start_node, char *path) {
    // 实现循环检测逻辑
    // 遍历路径，跟踪已访问的inode，防止循环
    return 0; // 无循环返回0
}
```

7 注意事项

1. 硬链接限制：不允许对目录创建硬链接，以避免破坏文件系统层次结构

2. 跨文件系统：软链接可以跨文件系统，硬链接不能跨文件系统

3. 删除处理：删除原文件时，硬链接仍可访问文件内容，软链接变悬空

4. 权限检查：创建链接时需要检查相应权限

5. 文件系统一致性：确保在系统崩溃时文件系统的一致性

8 总结

该设计通过扩展现有的 VFS 和 SFS 结构，实现了软链接和硬链接机制。主要特点包括：

- 保持了与现有文件系统架构的兼容性
- 正确处理了同步互斥问题
- 提供了完整的接口定义
- 考虑了安全性问题（如循环检测）