Lab8 实验报告

练习0：填写/整合已有实验代码

本实验依赖 Lab2/3/4/5/6/7，需要把前序实验中标注为 `LAB2` / `LAB3` / `LAB4` / `LAB5` / `LAB6` / `LAB7` 的代码补齐并保证能够编译通过。

本报告中不再逐段展开粘贴前序实验的全部实现（避免与 Lab2~Lab7 报告重复），仅说明为了让 Lab8 的用户程序与测试能够正确运行，本人对"整合代码"做了如下与 Lab8 强相关的检查/补全：

- **进程切换与页表切换正确性**：在 `proc_run()` 中切换 `satp` 后刷新 TLB：

  （函数：`proc_run()`）

  ```
  lsatp(proc->pgdir);
  flush_tlb();  // LAB8: flush TLB after changing page table
  ```

- **文件表结构接入（filesp）**：在 `alloc_proc()` 中初始化 `filesp` 字段：

  （函数：`alloc_proc()`）

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

以上内容属于"练习0"的整合性工作：保证前序实验代码在 Lab8 环境下可用，并满足测试程序对进程/文件系统行为的依赖。

练习1：完成读文件操作的实现

调用链分析：从 read 到 sfs_io_nolock

文件读操作的完整调用链如下：

1. **用户态**：
   - `read(fd, data, len)` 用户进程发起读请求。
   - 实际调用 `sys_read(fd, data, len)` 进入内核。
2. **系统调用/文件系统抽象层**：
   - `sys_read` 解析参数，调用 `sysfile_read(fd, base, len)`。
   - `sysfile_read` 检查参数、分配内核 buffer，循环调用 `file_read` 读取数据。
   - `file_read` 通过 `fd2file` 获取文件结构体，初始化 iobuf，调用 `vop_read(file->node, iob)`
3. **VFS 层**：
   - `vop_read` 是一个宏，实际会调用具体文件系统的 `inode_ops->vop_read`，对于 SFS 文件系统就是 `sfs_read`。
4. **SFS 文件系统层**：
   - `sfs_read` 调用 `sfs_io(node, iob, 0)`，加锁后调用 `sfs_io_nolock` 完成实际读操作。
   - `sfs_io_nolock` 负责分块处理数据，最终通过 `sfs_bmap_load_nolock`、`sfs_rbuf`、`sfs_rblock` 等函数实现磁盘到内存的数据传输。

简要流程图：

```
read
  ↓
sys_read
  ↓
sysfile_read
  ↓
file_read
  ↓
vop_read (→ sfs_read)
  ↓
sfs_io
  ↓
sfs_io_nolock
```

这样，用户的 read 请求最终会通过多层抽象，落到 SFS 文件系统的 sfs_io_nolock 函数，完成实际的文件数据读取。

原理分析

在基于文件系统的操作系统中，文件读操作是一个核心功能。`sfs_io_nolock()` 函数实现了对Simple File System (SFS)中文件内容的读写操作。该函数的核心作用是将磁盘块中的数据与内存中的缓冲区进行交互。

关键概念：

- **块对齐问题**：文件系统以固定大小的块（SFS_BLKSIZE）为单位存储数据，但用户的读写请求可能不与块边界对齐
- **三阶段处理**：为了处理非对齐读写，需要分三个阶段进行：
    1. 首块不对齐处理：如果偏移量不是块大小的倍数，先处理首块中的部分数据
    2. 中间完整块处理：处理完全对齐的中间块，可以批量操作提高性能
    3. 末块不对齐处理：处理最后一块中的部分数据

实现代码

在 `kern/fs/sfs/sfs_inode.c` 的 `sfs_io_nolock()` 函数中，实现了完整的三阶段读写逻辑：

```
// (1) 处理首块不对齐的情况
if (blkoff != 0) {
    size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (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;
    buf += size;
    if (nblks == 0) {
        goto out;
    }
    blkno++;
    nblks--;
}

// (2) 处理中间完整块
```

```
if (nblks > 0) {
    if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
        goto out;
    }
    if ((ret = sfs_block_op(sfs, buf, ino, nblks)) != 0) {
        goto out;
    }
    alen += nblks * SFS_BLKSIZE;
    buf += nblks * SFS_BLKSIZE;
    blkno += nblks;
    nblks = 0;
}

// (3) 处理末块不对齐的情况
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;
}
```

关键函数说明：

- `sfs_bmap_load_nolock()`：根据逻辑块号获取实际物理块号
- `sfs_buf_op()`：处理部分块的读写（指向 `sfs_rbuf` 或 `sfs_wbuf`）
- `sfs_block_op()`：批量读写完整块（指向 `sfs_rblock` 或 `sfs_wblock`）

其实就是分三块主要逻辑：对首块不对齐的特殊处理、正常处理后面的nblks的读写、对末块不对齐的处理。其中，对末块不对齐的处理只需要更改alen的增加粒度为endpos%blocksize即可。

练习2：完成基于文件系统的执行程序机制的实现

原理分析

在Lab5中实现的 `load_icode()` 函数是从内存中加载ELF格式的二进制程序。而Lab8要求将程序存储在文件系统中，通过文件描述符（fd）来加载程序。这需要修改 `load_icode()` 函数的参数和内部实现，使其能够：

1. **动态读取ELF头**：从文件中读取ELF头，而不是从内存buffer中
2. **动态读取程序头表**：根据ELF头信息读取程序头表
3. **逐段加载程序**：将TEXT/DATA/BSS段从文件读入内存

实现代码

1. 文件读取辅助函数

在 `kern/process/proc.c` 中添加了 `load_icode_read()` 函数：

```
static int
load_icode_read(int fd, void *buf, size_t len, off_t offset)
{
    int ret;
```

```
    // 使用 sysfile_seek 定位到文件偏移量
    if ((ret = sysfile_seek(fd, offset, LSEEK_SET)) != 0)
    {
        return ret;
    }
    // 使用 sysfile_read 从文件读取数据
    if ((ret = sysfile_read(fd, buf, len)) != len)
    {
        return (ret < 0) ? ret : -1;
    }
    return 0;
}
```

该函数封装了从文件特定偏移量读取数据的操作，使用 `sysfile_seek()` 和 `sysfile_read()` 系统调用。

2. 主要加载函数

修改后的 `load_icode()` 函数采用与Lab5相同的八步骤框架，但关键改变是：

- 参数从 `(unsigned char *binary, size_t size)` 改为 `(int fd, int argc, char **kargv)`
- 使用 `load_icode_read()` 动态读取ELF信息，而非直接访问内存

(1) 创建新的内存管理结构

```
struct mm_struct *mm;
if ((mm = mm_create()) == NULL) {
    goto bad_mm;
}
```

(2) 设置页目录

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

(3.1) 从文件读取并解析ELF头

```
struct elfhdr elf_buf;
struct elfhdr *elf = &elf_buf;
if ((ret = load_icode_read(fd, elf, sizeof(struct elfhdr), 0)) != 0) {
    goto bad_pgdir_cleanup_mm;
}
```

(3.2) 读取程序头表

```
struct proghdr *ph_orig = (struct proghdr *)kmalloc(sizeof(struct proghdr) *
elf->e_phnum);
if (ph_orig == NULL) {
    goto bad_pgdir_cleanup_mm;
}
if ((ret = load_icode_read(fd, ph_orig, sizeof(struct proghdr) * elf->e_phnum,
elf->e_phoff)) != 0) {
    goto bad_ph_cleanup_pgdir;
}
```

((3.3)-(3.5) 加载各个程序段

遍历每个程序头，对于 `ELF_PT_LOAD` 类型的段：

- 使用 `mm_map()` 为TEXT/DATA/BSS段建立虚拟地址映射
- 使用 `load_icode_read()` 从文件读取段数据到临时缓冲区
- 使用 `pgdir_alloc_page()` 为每个页面分配物理页
- 使用 `memcpy()` 将文件数据复制到物理页
- 使用 `memset()` 将BSS段清零

```
unsigned char *from = (unsigned char *)kmalloc(ph->p_filesz);
if (from == NULL) {
    goto bad_cleanup_mmap;
}
if ((ret = load_icode_read(fd, from, ph->p_filesz, ph->p_offset)) != 0) {
    kfree(from);
    goto bad_cleanup_mmap;
}
```

(4) 建立用户栈 VMA 并预分配栈页

完成各个程序段映射后，需要为用户态栈建立虚拟内存区域（VMA）。本实现先用 `mm_map()` 在 `[USTACKTOP-USTACKSIZE, USTACKTOP)` 建立栈区映射，再额外在栈顶“预分配”4页物理内存，避免用户程序刚启动时因缺页而触发异常（也便于后续往栈上写入参数）。

```
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);
```

(6) 在用户栈上构造 argc/argv 与参数字符串

`argv` 的本质是“指针数组”，每个 `argv[i]` 都指向一个以 `\0` 结尾的参数字符串。实现时需要把“参数字符串内容”与“argv指针数组本身”都写入用户栈。

1）先统计所有参数字符串总长度（每个字符串都要包含末尾 `\0`）：

```
uint32_t argv_size = 0, i;
for (i = 0; i < argc; i++) {
  argv_size += strnlen(kargv[i], EXEC_MAX_ARG_LEN + 1) + 1;
}
```

2）计算在用户栈上的放置位置。

- `stacktop`：参数字符串区起始地址（靠近 `USTACKTOP` 的高地址处）
- `uargv`：argv 指针数组起始地址（位于参数字符串区"下方"的更低地址处）

```
uintptr_t stacktop = USTACKTOP - (argv_size / sizeof(long) + 1) * sizeof(long);
char **uargv = (char **)(stacktop - argc * sizeof(char *));
```

这里用 `sizeof(long)` 做了一个简单的对齐（避免栈上数据出现"奇怪的未对齐"）。

3）把每个参数字符串拷贝到用户栈，并在 `uargv[i]` 中写入该字符串的用户态虚拟地址。

因为内核态不能直接用用户态虚拟地址当作普通指针去写，所以代码通过 `get_pte()` 找到对应页表项，再把"用户虚拟地址"转换成"内核可访问的 kva"，最后 `strcpy/写指针`。

```
argv_size = 0;
for (i = 0; i < argc; i++) {
  uintptr_t str_addr = stacktop + argv_size;
  pte_t *pte = get_pte(mm->pgdir, str_addr, 0);
  void *kva_str = page2kva(pte2page(*pte)) + (str_addr & (PGSIZE - 1));
  strcpy((char *)kva_str, kargv[i]);

  uintptr_t argv_addr = (uintptr_t)&uargv[i];
  pte = get_pte(mm->pgdir, argv_addr, 0);
  void *kva_argv = page2kva(pte2page(*pte)) + (argv_addr & (PGSIZE - 1));
  *(char **)kva_argv = (char *)str_addr;

  argv_size += strnlen(kargv[i], EXEC_MAX_ARG_LEN + 1) + 1;
}
```

4）最后把 `argc` 写到 `uargv` 下方，并让最终用户态栈指针 `sp` 指向 `argc` 的位置：

```
stacktop = (uintptr_t)uargv - sizeof(int);
pte_t *pte = get_pte(mm->pgdir, stacktop, 0);
void *kva_argc = page2kva(pte2page(*pte)) + (stacktop & (PGSIZE - 1));
*(int *)kva_argc = argc;
```

因此，本实现中用户栈从高地址到低地址的布局可概括为：

```
高地址  USTACKTOP
  |  参数字符串区（连续的"xxx\0yyy\0..."）
  |  argv 指针数组（char* argv[argc]，每项指向上面的字符串）
  |  argc（int）  <- sp 最终指向这里
低地址
```

(5) 安装新的 mm/pgdir 并切换到该进程页表

完成映射与用户栈内容写入后，把新建的 `mm` 挂到 `current` 上，并把硬件页表寄存器切换到该进程的页目录：

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

这里 `lsatp(...)` 的效果是让 CPU 后续的地址翻译使用新页表，从而真正"进入"新程序的地址空间。

(7) 设置 trapframe：sp/epc/status

`trapframe` 是内核在"返回用户态"时用来恢复寄存器现场的数据结构。`exec` 的语义是"用新程序替换当前进程"，因此需要重置 trapframe，让用户态从新入口开始执行。

```
struct trapframe *tf = current->tf;
uintptr_t sstatus = tf->status;
memset(tf, 0, sizeof(struct trapframe));
tf->gpr.sp = stacktop;       // 用户栈指针，指向 argc
tf->epc = elf->e_entry;      // 用户程序入口（ELF e_entry）
tf->status = sstatus & ~(SSTATUS_SPP | SSTATUS_SPIE);
```

其中 `tf->epc = elf->e_entry` 会在"从内核返回用户态"的那一刻生效：trap 返回路径会用 trapframe 恢复寄存器，并把 `epc` 装载为下一条要执行的用户态 PC，从而跳到用户程序入口。

调用链补充：epc 如何真正让用户程序跑起来

在本实验中，`load_icode()` 设置 `tf->epc` 后，后续关键链路可概括为：

```
用户态 execve
  -> sys_exec
  -> do_execve
  -> load_icode（设置 pgdir + 用户栈 + tf->sp/tf->epc）
  -> 系统调用返回路径（trap return）
  -> 根据 trapframe 恢复寄存器并跳转到 tf->epc
  -> 从 elf->e_entry 开始执行新用户程序
```

调用链

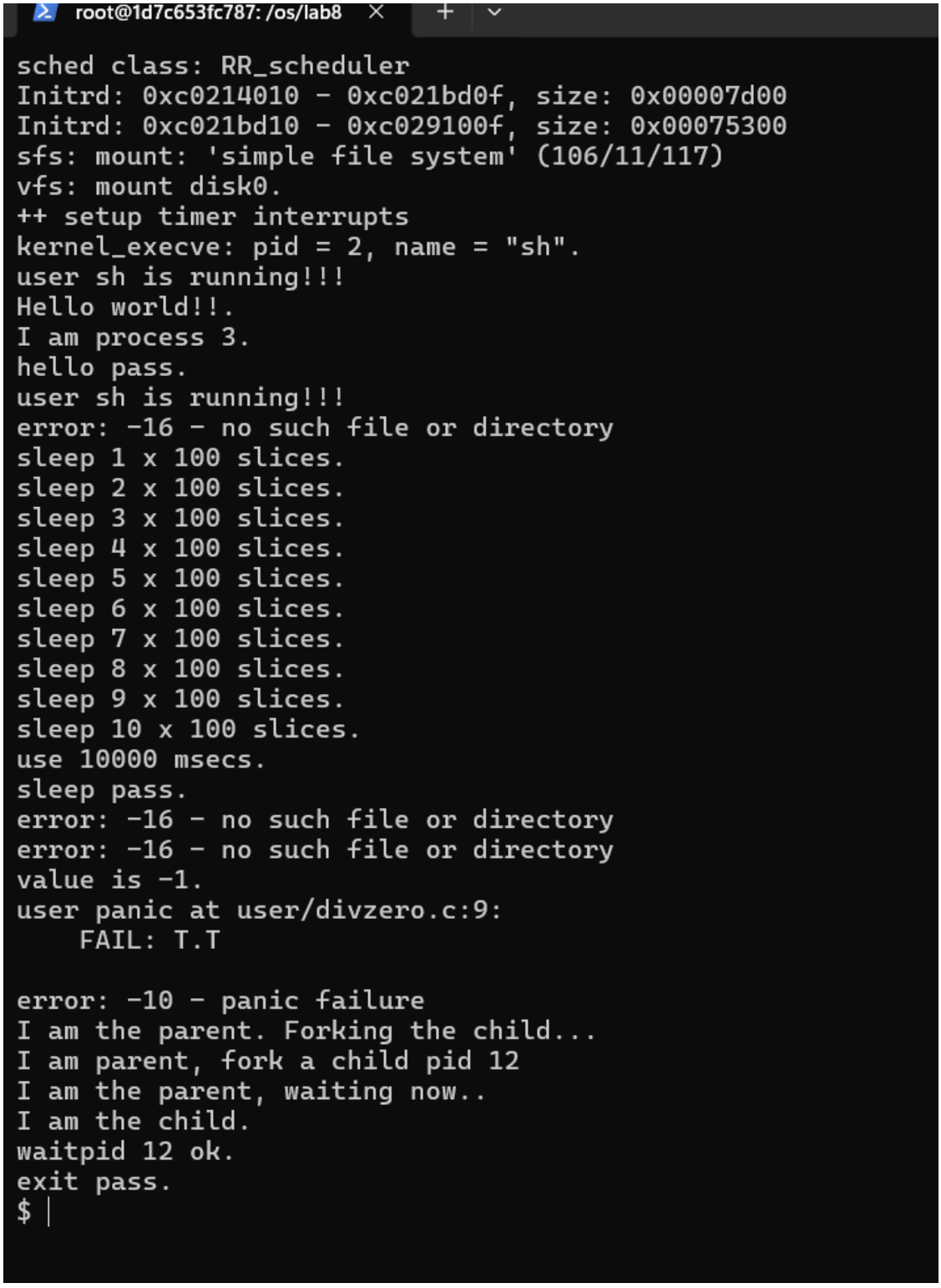
- `do_execve()` → `load_icode(fd, argc, kargv)` → `load_icode_read()` → `sysfile_seek()` / `sysfile_read()`
- 最终通过VFS接口调用到 `sfs_io_nolock()` 进行实际的磁盘I/O操作

补充说明：除 `execve` 的装载链路外，进程在运行过程中发生切换时（`proc_run()` 切换页表）需要 `flush_tlb()` 配合，确保切换到新的地址空间后 TLB 不残留旧映射。

验证成功

程序成功运行的标志：

1. 能够看到sh用户shell的执行界面
2. 能够在sh中执行exit、hello等用户程序
3. 这些程序都存储在SFS文件系统中的 `disk0/` 目录下
   简单测试了 `hello`、`sh`、`sleep`、`divzero`、`exit` 效果如下：

4. 

ps：`error: -16 - no such file or directory`是因为`divzero`我打成了`divezero`，所以没找到对应的程序（英语忘干净了）。

扩展练习 Challenge1：完成基于“UNIX 的 PIPE 机制”的设计方案

目标与语义

Pipe（管道）是一种典型的 UNIX 进程间通信（IPC）机制，用于在两个进程（常见场景是父子进程）之间建立“单向字节流”通道。`pipe()` 创建后返回两个文件描述符：读端 `rfd` 与写端 `wfd`。

- **读语义**：

    - 管道缓冲区为空且仍存在写端：`read()` 阻塞等待数据（或非阻塞模式返回 `-E_AGAIN`）。
    - 管道缓冲区为空且所有写端关闭：`read()` 返回 0（EOF）。
- **写语义**：
    - 管道缓冲区满且仍存在读端：`write()` 阻塞等待空间（或非阻塞模式返回 `-E_AGAIN`）。
    - 所有读端关闭：`write()` 失败（UNIX 上通常是 SIGPIPE；在 ucore 里可以简化为返回 `-E_PIPE`）。
- **继承/共享语义**：`fork()` 后子进程继承 fd，读端/写端引用计数应正确维护；`close()` 需递减引用并在最后一个引用关闭时释放对象。

至少需要的数据结构与接口设计

1）关键数据结构

管道的核心是一个受保护的共享缓冲区（常见实现是环形队列），以及用于实现阻塞读写的等待队列/条件变量。

```
#define PIPE_BUF_SIZE 4096

struct pipe_ring {
        char buf[PIPE_BUF_SIZE];
        size_t rpos;
        size_t wpos;
        size_t used;       // 当前已用字节数
};

typedef struct wait_queue wait_queue_t;

struct pipe_info {
        struct pipe_ring ring;

        // 引用计数：用于实现 EOF / EPIPE
        int readers;
        int writers;

        // 互斥保护：保护 ring/readers/writers 等共享状态
        volatile int lock;

        // 阻塞队列：无数据可读 / 无空间可写
        wait_queue_t rwait;
        wait_queue_t wwait;
};
```

设计说明：

- 读写操作都必须在 `lock` 保护下更新 `rpos/wpos/used`，避免并发破坏环形缓冲区状态。
- 写入数据后唤醒 `rwait`；读走数据后唤醒 `wwait`。
- `writers==0` 作为读端 EOF 判定；`readers==0` 作为写端 EPIPE 判定。

2）接口

- `int sys_pipe(int pipefd[2]);`
    - 创建一个 `pipe_info` 并分配两个 fd：`pipefd[0]` 为读端、`pipefd[1]` 为写端。
    - 初始化 `readers=1, writers=1`（只有当前进程在用）
- `ssize_t sys_read(int fd, void *buf, size_t len);`

    - 若 `fd` 为管道读端：从 `pipe_info` 读取最多 `len` 字节。
    - **管道有数据**：直接读，唤醒可能在 `wwait` 等待的写者
    - **管道空 + 还有写端开着**（`writers>0`）：加入 `rwait` 睡眠，等写者写数据
    - **管道空 + 写端全关了**（**writers==0**）：立即返回 0（EOF，文件结束）
- `ssize_t sys_write(int fd, const void *buf, size_t len);`
    - 向 `pipe_info` 写入最多 `len` 字节。
    - **管道有空间**：直接写，唤醒可能在 `rwait` 等待的读者
    - **管道满 + 还有读端开着**（`readers>0`）：加入 `wwait` 睡眠，等读者读走数据
    - **读端全关了**（`readers==0`）：返回 `-E_PIPE`（管道破裂错误）
- `int sys_close(int fd);`
    - 关闭对应端并更新 `readers/writers`；
    - 如果 `writers==0` 了，唤醒 `rwait`（让读者知道EOF了）
    - 如果 `readers==0` 了，唤醒 `wwait`（让写者知道管道破裂了）
    - 均为0时，释放 `pipe_info` 结构体

同步互斥问题与处理

问题1：读写并发导致缓冲区状态错乱

场景：进程A执行 `read()`，进程B执行 `write()`，两者同时修改 `rpos`、`wpos`、`used`。

错误做法（无锁保护）：

```
// Thread A (读)
size_t rpos = ring->rpos;
memcpy(buf, ring->buf + rpos, size);
ring->rpos = (rpos + size) % PIPE_BUF_SIZE;  // ← 可能被中断
ring->used -= size;                          // ← 读到这里时，B已修改了 used

// Thread B (写)  同时执行
memcpy(ring->buf + ring->wpos, data, size);
ring->wpos = (ring->wpos + size) % PIPE_BUF_SIZE;
ring->used += size;  // ← 与 A 的减法并发，导致 used 数据错乱
```

结果：`ring->used` 的最终值可能既不等于预期的写入量，也不等于预期的读取后剩余量。

正确做法（加锁保护）：

```
// sys_read()
acquire_lock(&pipe->lock);
{
    while (ring->used == 0) {
        if (pipe->writers == 0) {  // EOF 判定
            release_lock(&pipe->lock);
            return 0;
        }
        sleep(&pipe->rwait, &pipe->lock);  // 释放锁后睡眠，被唤醒时重新获锁
        // 从睡眠返回后重新获锁，继续在临界区内执行
    }
    // 此时确保 ring->used > 0
    size_t rpos = ring->rpos;
    memcpy(buf, ring->buf + rpos, size);
    ring->rpos = (rpos + size) % PIPE_BUF_SIZE;
    ring->used -= size;
```

```
    wake_up(&pipe->wwait);  // 唤醒可能在等待空间的写者
}
release_lock(&pipe->lock);
```

所有对 `ring` 和 `readers/writers` 的访问都在 `lock` 保护下进行，保证一致性。

问题2：虚假唤醒（Spurious Wakeup）导致错误返回

场景：多个读者等待同一个管道。写者写入1个字节，唤醒 `rwait` 队列，但两个读者都被唤醒了。第一个读者读走这1字节，第二个读者被唤醒时缓冲区又空了。

错误做法（用 `if` 检查）：

```
// 写者唤醒时
if (ring->used == 0) {
    return;  // 缓冲区为空，不唤醒
}
wake_up(&pipe->rwait);  // 唤醒所有等待读者

// 读者端（错误写法）
if (ring->used == 0) {
    // ← 只检查一次！
    sleep(&pipe->rwait);
}
// 从睡眠返回后，直接执行读操作
size_t size = ring->used;  // 如果另一个读者已读走数据，这里是 0
memcpy(buf, ring->buf + ring->rpos, size);  // ← 读 0 字节！
```

结果：第二个读者虽然被唤醒了，但读不到任何数据，不知道是EOF还是应该继续等待。

正确做法（用 `while` 重新检查）：

```
acquire_lock(&pipe->lock);
{
    while (ring->used == 0) {  // ← while，被唤醒后重新检查！
        if (pipe->writers == 0) {
            release_lock(&pipe->lock);
            return 0;  // EOF
        }
        // 睡眠（在 while 保护内）
        sleep(&pipe->rwait, &pipe->lock);
        // 被唤醒后自动重新获锁，继续 while 的条件判定
    }
    // 此时确保 ring->used > 0
    size_t rpos = ring->rpos;
    memcpy(buf, ring->buf + rpos, size);
    ring->rpos = (rpos + size) % PIPE_BUF_SIZE;
    ring->used -= size;
    wake_up(&pipe->wwait);
}
release_lock(&pipe->lock);
```

关键：`while (cond) sleep()` 会在每次被唤醒后，在 `lock` 保护下重新评估条件，避免虚假唤醒导致的错误行为。

问题3：EOF 与 EPIPE 的错误判定

场景1：将"暂时无数据"误判为 EOF

错误做法：

```
// 读端
if (ring->used == 0) {
    return 0;  // 错误！应该只在 writers==0 时才返回 0
}
```

问题：一时没有数据，但写端仍然开着，后续还会有数据进来。应该**阻塞等待**，而不是返回0。

场景2：未关闭读端时错误返回 EPIPE

错误做法：

```
// 写端
while (ring->used == PIPE_BUF_SIZE) {
    return -E_PIPE;  // 错误！应该只在 readers==0 时才返回 EPIPE
}
```

问题：缓冲区满，说明有读者在等待，应该**阻塞等待读者消费**，而不是报错。

正确做法：

```
// 读端（只有在 writers==0 时才返回 EOF）
acquire_lock(&pipe->lock);
{
    while (ring->used == 0) {
        if (pipe->writers == 0) {
            release_lock(&pipe->lock);
            return 0;  // ← 确认无写端后才返回 EOF
        }
        sleep(&pipe->rwait, &pipe->lock);
    }
    // ... 读操作 ...
}
release_lock(&pipe->lock);

// 写端（只有在 readers==0 时才返回 EPIPE）
acquire_lock(&pipe->lock);
{
    while (ring->used == PIPE_BUF_SIZE) {
        if (pipe->readers == 0) {
            release_lock(&pipe->lock);
            return -E_PIPE;  // ← 确认无读端后才返回 EPIPE
        }
        sleep(&pipe->wwait, &pipe->lock);
    }
    // ... 写操作 ...
}
release_lock(&pipe->lock);
```

问题4：fork() 后引用计数不一致

场景：父进程 `pipe(pipefd)` 后 `fork()`，子进程继承文件描述符。

错误做法（不更新引用计数）：

```
// 父进程
sys_pipe(pipefd);  // pipe_info: readers=1, writers=1

// fork 后
pid_t child = fork();
// 子进程现在也持有 pipefd[0] 和 pipefd[1]
// 但 pipe_info 的 readers/writers 还是 1! ← 错误

// 子进程关闭读端
close(pipefd[0]);  // readers-- 变成 0
// 错误地认为读端全关了，父进程的读端被"坑死"了
```

结果：父进程虽然还持有读端，但 `readers==0` 了，写者会返回 EPIPE，导致通信失败。

正确做法（fork 时同步更新计数）：

```
// 父进程
sys_pipe(pipefd);  // pipe_info: readers=1, writers=1

// fork
pid_t child = fork();
if (child == 0) {
    // 子进程：复制父进程的 fd_table 后，需要增加 pipe_info 的计数
    // (由内核的 do_fork -> copy_files 自动完成)
    // 现在 pipe_info: readers=2, writers=2

    close(pipefd[0]);  // readers-- 变成 1（父进程读端仍有效）
} else {
    // 父进程
    close(pipefd[0]);  // readers-- 变成 1（子进程读端仍有效）
}
```

关键：**每当文件被复制到新进程时，对应 `pipe_info` 的计数必须增加；每当 fd 被关闭时，计数必须减少。**只有当计数真的变为 0 时，才能安全地回收资源。

问题5：并发 close 与唤醒的顺序问题

错误做法（唤醒前没有更新计数）：

```
// sys_close()
if (is_read_end) {
    wake_up(&pipe->wwait);  // ← 唤醒写者
    pipe->readers--;         // ← 后更新计数!
}
```

问题：写者被唤醒时，`readers` 还没变为0，写者继续等待；等到 `readers--` 执行完，写者早已超时或被其他信号打断。

正确做法（先更新计数，后唤醒）：

```
// sys_close()
```

```
acquire_lock(&pipe->lock);
{
    if (is_read_end) {
        pipe->readers--;
        if (pipe->readers == 0) {  // ← 只有当引用真的归零时才唤醒
            wake_up(&pipe->wwait);  // 通知写者：读端全关了，返回 EPIPE
        }
    }
    if (is_write_end) {
        pipe->writers--;
        if (pipe->writers == 0) {
            wake_up(&pipe->rwait);  // 通知读者：写端全关了，返回 EOF
        }
    }
    // 如果两端都关闭了，释放 pipe_info
    if (pipe->readers == 0 && pipe->writers == 0) {
        kfree(pipe);
    }
}
release_lock(&pipe->lock);
```

同步互斥设计总结

问题	原因	解决方案
读写竞争	两个进程同时改 rpos/wpos/used	所有缓冲区操作用 `lock` 保护
虚假唤醒	多个等待者被唤醒，但数据不足	用 `while (cond) sleep()` 重新检查
EOF/EPIPE 误判	未检查 readers/writers	同时检查 `used` 和 `readers/writers`
引用计数不一致	fork/close 时不同步计数	fork 时 `readers/writers++`，close 时递减
唤醒顺序错误	唤醒后计数才更新，等待者看不到	先更新计数再唤醒，在 lock 保护内原子执行

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

目标与语义

UNIX 的链接机制建立在“目录项（name）→ inode（对象）”的映射之上。

- **硬链接（hard link）**：多个目录项指向同一个 inode（同一份数据）。每个目录项都是等价的。删除一个目录项只是从目录中移除该记录，inode 的链接计数 `nlinks` 减 1，只有当 `nlinks==0` 且没有进程打开该文件时，数据才会被真正回收。
- **软链接（symbolic link）**：创建一个特殊的新 inode，其内容保存一个目标路径字符串。当路径解析遇到软链接 inode 时，需要继续解析其中保存的目标路径，并限制最大解析深度避免循环链接导致无限递归。

数据结构设计

1）硬链接：inode 增加链接计数（nlinks）

在文件系统（SFS）的磁盘 inode 中增加 `nlinks` 字段并持久化，记录有多少目录项指向该 inode。

```
struct sfs_disk_inode {
        uint16_t type;
        uint16_t nlinks;          // 硬链接计数：有多少目录项指向这个 inode
        uint32_t size;
        uint32_t extents;         // 数据块数
        // ... direct/indirect block pointers ...
};
```

2）软链接：新增 inode 类型 + 目标路径存储

软链接是一种特殊的 inode 类型，其内容直接存储目标路径字符串（短路径可直接在 inode 内，长路径使用数据块）。

```
#define SFS_TYPE_SYMLINK 3      // 定义新的 inode 类型

#define SYMLINK_MAX 256

struct sfs_symlink_info {
        uint16_t type;          // SFS_TYPE_SYMLINK
        uint16_t nlinks;        // symlink 自身也可以被硬链接
        uint16_t target_len;    // 目标路径长度
        char target[SYMLINK_MAX];  // 目标路径字符串内容
};
```

接口语义说明

硬链接接口

`sys_link(oldpath, newpath)` - 创建硬链接

- 在 `newpath` 所在的目录中创建一个新的目录项，让它指向 `oldpath` 所指向的同一个 inode并把inode上的nlinks++
- 两个文件名现在共享同一份数据，修改其中一个另一个也会变

`sys_unlink(path)` - 删除文件或硬链接

- 从目录中删除一个文件名（目录项），同时将对应 inode 的引用计数 `nlinks--`
- 如果还有其他文件名指向这个 inode（nlinks > 0），数据保留，如果 nlinks 变成 0 且没有进程打开，才真正释放数据

软链接接口

`sys_symlink(target, linkpath)` - 创建软链接

- 创建软链接”。它的作用是：在 linkpath 这个位置新建一个软链接文件，软链接文件的内容就是 target 这个路径字符串。

> 以后如果访问 linkpath，操作系统会自动把 linkpath 解析成 target 指向的路径，然后继续查找和访问

`sys_readlink(linkpath, buf, size)` - 读取软链接目标

- 直接读取 linkpath 这个软链接文件中保存的 target 字符串（即当初 sys_symlink 时写进去的内容），而不会去解析或访问 target 指向的实际文件内容。

关键区别

硬链接	软链接
同一份数据，多个文件名	特殊文件，保存目标路径字符串
删除一个硬链接，数据不消失	删除软链接，目标文件不受影响
不能跨文件系统，不能链接目录	可以跨文件系统，可以链接任何东西
修改哪个名字数据都变	修改目标文件内容，通过链接访问也会变

同步互斥问题与处理

问题1：目录项与 nlinks 的并发更新不一致

场景：两个进程同时执行 `link()`，都要增加同一个 inode 的 `nlinks`。

错误做法（无保护）：

```
// Process A
old_inode->nlinks++;      // ← 读-改-写中断
sync_inode_to_disk(old_inode);

// Process B 同时执行
old_inode->nlinks++;      // ← 读的值可能是还没被 A 同步的旧值
sync_inode_to_disk(old_inode);

// 结果：nlinks 应该 +2，但可能只 +1
```

正确做法：

```
lock_inode(old_inode);
{
    old_inode->nlinks++;
    sync_inode_to_disk(old_inode);
}
unlock_inode(old_inode);
```

同时修改目录项和修改 inode 的操作必须在**目录级锁**保护下原子执行，避免目录项创建成功但 `nlinks++` 失败，或反之。

问题2：unlink 时回收条件的正确性

错误做法：

```
if (inode->nlinks == 0) {
    free_inode(inode);  // ← 错误！还有进程打开着这个文件
}
```

问题：某个进程可能已经 `open()` 了这个文件，即使目录项全被删除（nlinks==0），该进程仍在使用文件数据，不能立即释放。

正确做法：

```
if (inode->nlinks == 0 && inode->open_count == 0) {
    free_inode_blocks(inode);  // 只有两个条件都满足才释放
    free_inode(inode);
}
```

需要维护每个 inode 的 `open_count`（有多少个进程打开了这个文件），在 `open()` 时增加，在 `close()` 时减少。

问题3：路径解析时的 inode 引用计数

> 在多线程或并发环境下，路径查找（如 vfs_lookup）返回 inode 指针时，必须用引用计数（ref_count）保护这个 inode 的生命周期。否则，别的线程可能在该线程使用 inode 期间把它删掉（比如 unlink），导致该进程手里的 inode 指针变成“悬挂指针”，继续访问会出错甚至崩溃。因此，在路径解析到这个inode的时候，必须把这个inode的引用计数++，在释放的时候再--。

错误做法：

```
struct inode *vfs_lookup(const char *path) {
    // ...
    return inode;  // ← 返回了 inode 指针，但没有增加引用计数
}

// 调用者
struct inode *inode = vfs_lookup(path);
// 此时另一个线程可能删除了这个 inode
unlink(path);
// inode 已被释放，inode 成为悬挂指针！
inode->size;  // ← 段错误或数据错乱
```

正确做法：

```
struct inode *vfs_lookup(const char *path) {
    // ...
    inode_ref_inc(inode);  // ← 增加引用计数
    return inode;
}

// 调用者
struct inode *inode = vfs_lookup(path);
// 使用 inode...
inode_ref_dec(inode);  // ← 使用完后递减引用计数
```

只有当 `nlinks==0` 且 `ref_count==0` 时，才能真正释放 inode。

问题4：symlink 循环链接检测

场景：创建循环软链接，导致无限递归。

```
$ symlink("link_b", "/tmp/link_a")
$ symlink("link_a", "/tmp/link_b")
# 现在 link_a → link_b → link_a → link_b → ...（无限循环）
```

解决：在 VFS 路径解析时设置最大跟随深度（如 8 或 16），超过该深度返回错误。

问题5：symlink 跟随时的目录权限检查

场景：symlink 目标可能在没有读权限的目录中，或目标本身不存在。

错误做法：无条件跟随 symlink，导致权限绕过。

正确做法：跟随 symlink 时，需要继续检查解析路径中每一段的目录权限，确保用户有访问权。

同步互斥问题总结

问题	原因	解决方案
目录项与 nlinks 不一致	并发修改 inode 元数据	用 inode 锁保护 `nlinks++/--` 操作
unlink 后过早释放	没考虑 open_count	需同时满足 `nlinks==0 && open_count==0`
悬挂指针	路径解析后 inode 被删除	使用引用计数，确保使用期间 inode 不会释放
循环链接无限递归	symlink 循环指向	设置最大跟随深度限制（8 或 16）
权限绕过	跟随 symlink 不检查权限	跟随时继续检查路径中每段的权限

总结

本次 Lab8 涉及文件系统的多个核心机制，具体总结如下：

练习1：文件读写实现
实现了多层抽象下的文件读操作，解决了块对齐、分段读写等底层细节，保证了文件系统的数据访问效率和正确性。

练习2：基于文件系统的程序加载
实现了从文件系统中动态加载和执行用户程序，完善了 exec 机制，使系统能够直接从磁盘文件启动用户进程。

Challenge1：UNIX 管道机制设计
设计并分析了管道的核心数据结构、接口语义和并发同步问题，深入理解了进程间通信的实现原理和常见陷阱。

Challenge2：软/硬链接机制设计
梳理了硬链接和软链接的本质区别，明确了相关接口的实现思路，并详细分析了并发下的引用计数、回收条件和路径解析等关键问题。

通过本实验，进一步加深了对类 Unix 操作系统中文件系统、进程管理和 IPC 机制的理解。