Lab8实验报告

练习1：文件读写中的块对齐处理

非对齐起始块的处理

当文件读写操作的起始偏移量（offset）未与块边界对齐时，首先通过 sfs_bmap_load_nolock 函数将当前的逻辑块号映射为对应的磁盘块号，随后调用 sfs_buf_op从该块内部的偏移位置开始进行数据的读或写操作。实际操作的数据长度由当前块内剩余空间与总读写需求共同决定，确保每次读写最多处理到当前块的末尾。

实现代码：

代码块

```
if (blkoff != 0) {
    if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
        goto out;
    }
    size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
    if ((ret = sfs_buf_op(sfs, buffer, size, ino, blkoff)) != 0) {
        goto out;
    }
    alen += size;
    if (nblks == 0) {
        goto out;
    }
    buffer += size;
    blkno++, nblks--;
}
```

对齐整块的批量处理

对于完全对齐的块，系统以循环方式调用 sfs_bmap_load_nolock 获取每个逻辑块对应的磁盘块号，并利用sfs_block_op 以整块为单位进行读写操作，从而提升I/O处理的效率。

实现代码：

代码块

```
while (nblks != 0) {
    if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
        goto out;
```

```
    }
    if ((ret = sfs_block_op(sfs, buffer, ino, 1)) != 0) {
        goto out;
    }
    alen += SFS_BLKSIZE;
    buffer += SFS_BLKSIZE;
    blkno++, nblks--;
}
```

非对齐尾块的处理

若读写操作的结束位置（endpos）未对齐到块边界，则对最后一个数据块从其起始位置开始，使用sfs_buf_op读写剩余的部分数据。

实现代码示例：

代码块

```
blkoff = endpos % SFS_BLKSIZE;
if (blkoff != 0) {
    if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
        goto out;
    }
    if ((ret = sfs_buf_op(sfs, buffer, blkoff, ino, 0)) != 0) {
        goto out;
    }
    alen += blkoff;
}
```

练习2：基于文件系统的程序加载机制

核心改动概述

在Lab8中，我们对程序执行机制进行了重要扩展，使其能够基于文件系统加载和执行用户程序。主要改进包括将程序装载的来源从内存镜像改为文件系统，并通过扩展 load_icode 及相关调用路径实现了完整的文件系统支持。

程序装载入口的转变

Lab8的核心改进在于程序不再以内核中预置的内存镜像形式提供，而是以普通文件的形式存放在文件系统中。为实现这一点：

- load_icode 函数的输入参数由内存地址改为文件描述符（fd）。
- 在函数内部，所有ELF相关数据（如ELF头、程序头、段内容）均通过文件系统接口读取。
- 程序段内容不再从内存复制，而是通过指定文件偏移的方式直接从文件中加载到用户地址空间。

这使得内核能够从SFS文件系统中加载任意可执行文件，不再依赖于编译时链接的程序镜像。

执行流程与文件系统的集成

为配合新的加载机制，执行流程进行了相应调整：

- do_execve不再直接接收程序镜像，而是接收用户提供的程序路径。
- 内核通过文件系统接口打开该路径对应的文件，并获得文件描述符。
- 该文件描述符作为程序内容的来源传递给 load_icode。

这一改动使得 exec 调用能够真正基于文件系统完成程序加载，实现了从“内存程序切换”到“文件程序加载”的转变。

新地址空间中的程序装载

在 load_icode 函数中，我们保留了Lab6中已有的地址空间构建逻辑，并增加了文件系统相关的步骤：

- 复用原有逻辑：创建新的内存管理结构（mm_struct）、建立页表、映射ELF程序段、构建用户栈。
- 新增步骤：程序段内容改为从文件系统读取；在装载完成后显式关闭文件描述符。

这种设计保证了程序执行与文件系统在内存布局上的解耦，符合“程序从文件加载，加载后关闭文件”的标准执行模型。

完整用户执行环境的建立

在完成文件装载后，load_icode 直接在新地址空间中设置用户态的运行时环境：

- 设置用户程序的入口地址。
- 设置用户栈指针。
- 切换页表并返回用户态执行。

这使得用户程序（如shell）能够以完整的执行环境启动，并进一步通过文件系统执行其他用户程序。

实现效果

通过上述关键修改：

- 内核能够从SFS文件系统中加载并执行用户程序。
- Shell可以作为普通用户程序启动。
- 在Shell中可以继续执行存放在文件系统中的其他可执行程序（如hello、exit）。

这表明基于文件系统的执行程序机制已在Lab8中成功实现。

测试结果

```
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
swap_in: load disk swap entry 6 with swap_page in vadr 0x5000
write Virt Page a in fifo_check_swap
Load page fault
page fault at 0x00001000: K/R
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000
check_swap() succeeded!
sfs: mount: 'simple file system' (106/11/117)
vfs: mount disk0.
++ setup timer interrupts
kernel_execve: pid = 2, name = "sh".
Breakpoint
user sh is running!!!
Hello world!!.
I am process 3.
hello pass.
I am the parent. Forking the child...
I am parent, fork a child pid 5
I am the parent, waiting now..
I am the child.
waitpid 5 ok.
exit pass.
```

Challenge1：UNIX管道机制设计方案

管道机制简介

管道是UNIX系统中最经典的进程间通信（IPC）机制之一，它在逻辑上表现为一个先进先出（FIFO）的队列。管道本质上是由内核管理的一个固定大小的缓冲区，通常占用一页或多页内存，而不对应磁盘上的任何数据块。

主要特性包括：

- 通信模型：通常为半双工通信。pipe() 系统调用创建两个文件描述符，分别用于读（read end）和写（write end）。
- 文件抽象：在VFS层，管道被抽象为一种特殊的inode，使得进程可以使用标准的 read() 和 write() 系统调用来操作管道，体现了“一切皆文件”的设计哲学。
- 同步特性：管道具有内置的同步机制。当缓冲区为空时，读进程阻塞；当缓冲区已满时，写进程阻塞，从而隐式地协调了读写速度。

数据结构设计

管道核心控制结构（pipe_info）

该结构体维护管道的环形缓冲区状态、读写指针及同步所需的等待队列，它将嵌入到VFS的inode中。

定义：

代码块

```
#define PIPE_SIZE 4096  // 缓冲区大小，通常为一页内存

struct pipe_info {
    /* 同步互斥原语 */
    struct semaphore mutex;         // 互斥信号量，保护缓冲区及指针的原子操作
    struct wait_queue wait_reader; // 读者等待队列，缓冲区空时读者在此等待
    struct wait_queue wait_writer; // 写者等待队列，缓冲区满时写者在此等待

    /* 环形缓冲区管理 */
    off_t head;  // 写指针，指示下一个写入位置（取模运算得到实际索引）
    off_t tail;  // 读指针，指示下一个读取位置（head==tail表示缓冲区空）

    /* 状态与引用计数 */
    bool is_closed;     // 管道关闭标记
    int reader_count;  // 读端引用计数
    int writer_count;  // 写端引用计数（两者归零时释放内存）

    /* 数据存储 */
    char *buffer;       // 内核缓冲区指针，通过kmalloc分配
};
```

VFS层inode结构扩展

为了支持管道，需要在inode结构中增加管道专用信息的联合体成员。

扩展：

代码块

```
struct inode {
    union {
        struct device __device_info;
        struct sfs_inode __sfs_inode_info;
        struct pipe_info __pipe_info;  // 新增：管道专用信息
    } in_info;
    enum {
        inode_type_device_info = 0x1234,
        inode_type_sfs_inode_info,
```

```
        inode_type_pipe_info,          // 新增：标识该inode为管道
    } in_type;
    int ref_count;
    int open_count;
    const struct inode_ops *in_ops;    // 管道将实现独立的读写操作
};
```

接口设计

管道操作接口（inode_ops）

管道需要实现专门的读写和关闭操作，以处理内存缓冲区的特殊性。

函数定义示例：

代码块

```
/**
 * 管道读取操作
 * @param node 管道inode
 * @param iob I/O缓冲区描述符
 * @return 成功返回0，失败返回错误码
 * 同步语义：
 * 1. 获取互斥锁。
 * 2. 若缓冲区为空且写端未关闭 -> 释放锁，在wait_reader中睡眠，被唤醒后重试。
 * 3. 若缓冲区为空且写端已关闭 -> 返回0（EOF）。
 * 4. 若有数据 -> 读取数据，更新tail，唤醒wait_writer，释放锁。
 */
int pipe_read(struct inode *node, struct iobuf *iob);

/**
 * 管道写入操作
 * @param node 管道inode
 * @param iob I/O缓冲区描述符
 * @return 成功返回0，失败返回错误码
 * 同步语义：
 * 1. 获取互斥锁。
 * 2. 若读端已关闭 -> 返回-EPIPE（Broken Pipe）。
 * 3. 若缓冲区已满 -> 释放锁，在wait_writer中睡眠，被唤醒后重试。
 * 4. 若有空间 -> 写入数据，更新head，唤醒wait_reader，释放锁。
 */
int pipe_write(struct inode *node, struct iobuf *iob);

/**
 * 管道关闭操作
 * @param node 管道inode
 * @return 成功返回0
```

```
 * 同步语义：
 * 1. 递减reader_count或writer_count。
 * 2. 唤醒所有在等待队列中的进程（通知状态变化，如EOF）。
 * 3. 若所有引用计数归零，释放buffer内存。
 */
int pipe_close(struct inode *node);
```

系统调用接口

用户态通过系统调用创建管道。

定义：

代码块

```
/**
 * 创建管道系统调用
 * @param fd_store 用于返回两个文件描述符的数组指针
 * @return 成功返回0，失败返回错误码
 * 实现逻辑：
 * 1. 分配一个新的管道inode。
 * 2. 分配内核缓冲区。
 * 3. 在当前进程的文件描述符表中分配两个空闲项。
 * 4. fd[0]绑定为只读，fd[1]绑定为只写，均指向同一个inode。
 */
int sys_pipe(int *fd_store);
```

同步互斥设计

锁使用协议

管道的同步机制遵循“获取锁->检查状态->（可能睡眠）->执行操作->唤醒对端->释放锁”的基本模式。

读取操作的同步逻辑伪代码：

代码块

```
void synchronized_pipe_read() {
    lock(pipe->mutex);  // 进入临界区

    while (is_empty(pipe)) {
        if (no_writers_left(pipe)) {
            unlock(pipe->mutex);
            return EOF;  // 写端关闭，返回文件结束
        }
        // 缓冲区空，等待写者
```

```
10            // wait操作原子地释放mutex并进入睡眠
11            wait(pipe->wait_reader, pipe->mutex);
12        }
13
14        read_data_from_buffer();
15        // 读取数据后腾出空间，唤醒写者
16        wakeup(pipe->wait_writer);
17        unlock(pipe->mutex);  // 退出临界区
18    }
```

关键并发场景处理

- 原子性保护：通过信号量（mutex）确保对head和tail指针的更新是原子的，防止多个写操作导致数据覆盖。
- 虚假唤醒处理：进程被唤醒后必须使用while循环重新检查缓冲区状态，不能直接进行读写，避免多读者同时被唤醒或信号中断导致的错误。
- 读写端关闭的同步：
  - 读端关闭：写进程尝试写入时将收到错误。
  - 写端关闭：读进程读完所有数据后将收到EOF（返回0），而非继续阻塞。

具体实施方案

1. 内核数据结构扩展：修改 kern/fs/vfs/inode.h，添加 struct pipe_info 定义并扩展 struct inode。
2. 管道核心逻辑实现：新建 kern/fs/pipe/pipe.c 和pipe.h，实现 pipe_open、pipe_read、pipe_write、pipe_close，正确使用同步原语。
3. 文件层与系统调用对接：修改 kern/fs/file.c 实现 file_pipe，连接VFS和进程管理层；修改 kern/fs/sysfile.c 实现 sysfile_pipe，处理用户空间数据拷贝。
4. 测试验证：
   - 基础测试：父子进程通信，验证数据完整性。
   - 阻塞测试：验证缓冲区满/空时的正确阻塞行为。
   - 并发测试：多个进程同时读写，验证数据不丢失。
   - 关闭测试：验证读写端关闭时的正确错误处理。

Challenge2：UNIX软链接与硬链接机制设计方案

链接机制简介

软链接（Symbolic Link）

软链接是一个特殊的文件，其内容存储着目标文件的路径字符串。创建软链接时，系统会分配新的inode和数据块，但数据块中存储的是目标路径文本。访问软链接时，系统会读取路径并重新进行路径解析。

特点：可跨文件系统、可指向目录、可指向不存在的目标（悬空链接）。删除软链接仅删除链接文件本身，不影响目标文件。

硬链接（Hard Link）

硬链接本质上是为同一个inode创建多个目录项引用。创建硬链接时，系统在目标目录中创建新的目录项，指向源文件的inode编号，同时递增inode的链接计数。删除文件时仅减少链接计数，计数归零时才真正释放存储空间。

限制：不能跨文件系统、不能为目录创建（避免循环）、所有硬链接地位平等。

数据结构设计

inode结构扩展

inode是文件的核心，存储元数据和数据位置，不关心文件名但包含硬链接计数和符号链接目标路径。

扩展：

代码块

```
struct ufs_inode {
    /* 文件类型和权限 */
    uint32_t i_mode;           // 文件模式，包含类型和权限位（新增S_IFLNK表示符号链接）

    /* 链接管理 */
    uint32_t i_nlink;          // 硬链接计数，记录指向此inode的目录项数量（需原子访问）

    /* 文件基本信息 */
    uint64_t i_size;           // 文件大小（字节）
    uint32_t i_blocks;         // 占用的磁盘块数

    /* 时间戳 */
    uint32_t i_atime;          // 最后访问时间
    uint32_t i_mtime;          // 最后修改时间（内容）
    uint32_t i_ctime;          // 最后状态改变时间（元数据）

    /* 数据块索引 */
    uint32_t i_data[15];       // 数据块指针数组（直接、间接指针）

    /* 符号链接专用字段 */
    uint32_t i_symlink_target; // 目标inode号（可选优化）
    char i_symlink_path[256];  // 目标路径字符串（最大255字符+终止符）
```

```
23
24      /* 同步原语 */
25      struct spinlock lock;       // 保护i_nlink等关键字段
26      struct semaphore sem;       // 控制对inode内容的并发访问
27  };
```

dentry结构扩展

dentry建立文件名到inode的映射，是硬链接的载体。一个inode可对应多个dentry。

扩展：

代码块

```
1   struct dentry {
2       /* 文件名信息 */
3       char name[NAME_MAX];        // 文件名（最大长度通常255）
4       uint32_t ino;               // 对应的inode编号
5
6       /* 文件类型缓存 */
7       uint8_t type;               // 文件类型（新增DT_LNK表示符号链接）
8
9       /* 目录树结构 */
10      struct dentry *parent;      // 父目录项指针
11      struct list_head child;     // 子目录项链表头
12
13      /* 引用计数和同步 */
14      atomic_t ref_count;         // 原子引用计数
15      struct spinlock lock;       // 保护dentry结构的并发修改
16  };
```

超级块（super_block）扩展

超级块管理整个文件系统的全局资源，包括inode和磁盘块的总量与剩余数量。

扩展示例：

代码块

```
1   struct super_block {
2       /* 文件系统标识 */
3       uint32_t s_magic;           // 魔数，标识文件系统类型
4
5       /* 资源总量统计 */
6       uint32_t s_inodes_count;    // inode总数
7       uint32_t s_blocks_count;    // 磁盘块总数
8
```

```
    /* 可用资源统计 */
    uint32_t s_free_inodes_count;  // 空闲inode数
    uint32_t s_free_blocks_count;  // 空闲磁盘块数

    /* 链接全局管理 */
    atomic_t s_active_links;    // 整个文件系统中硬链接的总数（原子操作）

    /* 同步原语 */
    struct rwlock s_lock;       // 读写锁，保护超级块整体结构
    struct semaphore s_sem;     // 信号量，用于互斥修改关键全局状态
};
```

接口设计

硬链接接口

创建硬链接：

代码块

```
/**
 * 创建硬链接
 * @param oldpath 源文件路径
 * @param newpath 链接文件路径
 * @return 成功返回0，失败返回错误码
 * 同步要求：需要原子递增inode的i_nlink，防止竞争条件
 */
int vfs_link(const char *oldpath, const char *newpath);
```

删除链接（unlink）：

代码块

```
/**
 * 删除硬链接（unlink通用接口）
 * @param pathname 要删除的链接路径
 * @return 成功返回0，失败返回错误码
 * 同步要求：递减i_nlink时需加锁，计数为0时触发文件删除
 */
int vfs_unlink(const char *pathname);
```

获取链接状态：

代码块

```c
/**
 * 获取文件链接状态
 * @param inode 目标inode
 * @return 硬链接计数
 * 同步要求：需要原子读取i_nlink
 */
uint32_t vfs_get_nlink(struct inode *inode);
```

软链接接口

创建符号链接：

代码块

```c
/**
 * 创建符号链接
 * @param target 目标路径
 * @param linkpath 链接文件路径
 * @return 成功返回0，失败返回错误码
 * 同步要求：创建新inode需获取父目录锁，防止并发创建同名链接
 */
int vfs_symlink(const char *target, const char *linkpath);
```

读取符号链接内容：

代码块

```c
/**
 * 读取符号链接内容
 * @param path 符号链接路径
 * @param buf 缓冲区
 * @param bufsiz 缓冲区大小
 * @return 成功返回读取字节数，失败返回错误码
 * 同步要求：读取i_symlink_path需要inode读锁
 */
ssize_t vfs_readlink(const char *path, char *buf, size_t bufsiz);
```

解析符号链接（跟随链接）：

代码块

```c
/**
 * 解析符号链接（跟随链接）
 * @param path 符号链接路径
 * @param resolved_path 解析后的路径缓冲区
```

```
 * @return 成功返回0，失败返回错误码
 * 同步要求：需要递归获取各级inode锁，防止解析过程中的竞争条件
 */
int vfs_realpath(const char *path, char *resolved_path);
```

通用文件系统接口扩展

查找inode（支持符号链接解析）：

代码块

```
/**
 * 查找inode（支持符号链接解析）
 * @param dir 起始目录inode
 * @param name 文件名
 * @param flags 查找标志（支持NOFOLLOW等）
 * @return 找到的inode指针
 * 同步要求：需要持有父目录锁进行查找
 */
struct inode *vfs_lookup(struct inode *dir, const char *name, int flags);
```

创建文件/目录/链接：

代码块

```
/**
 * 创建文件/目录/链接
 * @param dir 父目录inode
 * @param name 名称
 * @param mode 创建模式（包含文件类型）
 * @return 成功返回inode指针，失败返回错误
 * 同步要求：需获取目录inode的互斥锁，防止并发创建冲突
 */
struct inode *vfs_create(struct inode *dir, const char *name, uint32_t mode);
```

同步互斥设计

锁层次结构

硬链接操作需遵循固定的锁获取顺序以防止死锁：

1. 超级块锁（当需要修改全局结构时）。
2. 目录inode锁（父目录）。
3. 目标inode锁（已存在的文件）。

4. dentry锁（目录项缓存）。

硬链接操作锁协议：

代码块

```
void link_operation_protocol() {
    // 查找源文件
    lock_parent_directory();
    lock_source_inode();

    // 创建链接
    lock_target_parent_directory();

    // 原子操作：增加i_nlink
    atomic_inc(&inode->i_nlink);

    // 创建目录项
    create_dentry_under_lock();

    unlock_all();
}
```

原子操作保护

硬链接计数需使用CPU原子指令确保操作的不可分割性。

原子操作定义：

代码块

```
typedef struct {
    volatile uint32_t counter;
} atomic_t;

#define atomic_inc(v) (__sync_add_and_fetch(&(v)->counter, 1))
#define atomic_dec(v) (__sync_sub_and_fetch(&(v)->counter, 1))
#define atomic_read(v) ((v)->counter)

/* inode链接计数操作 */
static int inode_inc_nlink(struct inode *inode) {
    uint32_t old = atomic_inc(&inode->i_nlink);
    if (old == UINT32_MAX) {
        atomic_dec(&inode->i_nlink);
        return -EMLINK;  // 链接数超限
    }
    return 0;
}
```

软链接循环检测

符号链接解析时需要防止无限递归，需实现循环检测和深度限制机制。

循环检测实现：

代码块

```
#define MAX_SYMLINK_DEPTH 40  // 最大解析深度

struct symlink_resolve_ctx {
    struct inode *visited[MAX_SYMLINK_DEPTH];
    int depth;
    spinlock_t lock;  // 上下文锁
};

int check_symlink_cycle(struct symlink_resolve_ctx *ctx, struct inode *inode) {
    spin_lock(&ctx->lock);

    for (int i = 0; i < ctx->depth; i++) {
        if (ctx->visited[i] == inode) {
            spin_unlock(&ctx->lock);
            return -ELOOP;  // 检测到循环
        }
    }

    if (ctx->depth >= MAX_SYMLINK_DEPTH) {
        spin_unlock(&ctx->lock);
        return -ELOOP;
    }

    ctx->visited[ctx->depth++] = inode;
    spin_unlock(&ctx->lock);
    return 0;
}
```

具体实施方案

1. VFS层扩展：
    - 扩展 struct inode，增加链接计数和符号链接字段。
    - 在VFS接口层添加软硬链接操作函数原型。
    - 实现原子操作和基本锁机制。
2. 文件系统实现：

- 在具体文件系统（如sfs）中实现链接相关操作。
- 设计磁盘数据结构：扩展inode磁盘格式，增加 i_nlink 字段；设计符号链接数据存储方案。
- 实现事务性操作保证一致性。

3. 路径解析增强：
   - 修改路径查找逻辑，支持符号链接解析。
   - 实现循环检测和深度限制机制。
   - 添加 O_NOFOLLOW 等标志位支持。
4. 系统调用层：
   - 添加系统调用：sys_link、sys_unlink、sys_symlink、sys_readlink。
   - 修改现有的 open、stat 等系统调用以支持链接。
5. 测试验证：
   - 单元测试：链接创建、删除、解析功能。
   - 并发测试：多线程下的竞争条件处理。
   - 恢复测试：系统崩溃后的文件系统一致性验证。