lab8：文件系统

薄照轩 石爽 马思远

练习0：填写已有实验

已将实验2/3/4/5/6/7中完成的代码填入对应注释位置，主要包括：

1. **proc.c中的alloc_proc函数**：完成了进程控制块的初始化，包括状态、PID、栈、上下文等基本信息，以及实验8新增的文件结构指针初始化

```
// lab5 add:
proc->wait_state = 0;
proc->cptr = proc->optr = proc->yptr = NULL;
// lab6 add:
proc->rq = NULL;
list_init(&(proc->run_link));
proc->time_slice = 0;
skew_heap_init(&(proc->lab6_run_pool));
proc->lab6_stride = 0;
proc->lab6_priority = 1;
// lab8 add:
proc->filesp = NULL;
```

2. **pmm.c中的copy_range函数**：fork时复制父进程的页面到子进程

```
// (1) 获取父进程页面的内核虚拟地址
void *src_kvaddr = page2kva(page);

// (2) 获取子进程新分配页面的内核虚拟地址
void *dst_kvaddr = page2kva(npage);

// (3) 复制整个页面的内容 (4KB)
memcpy(dst_kvaddr, src_kvaddr, PGSIZE);

// (4) 在子进程的页表中建立映射
ret = page_insert(to, npage, start, perm);
```

3. 其他实验中实现的相关函数，确保代码能够正确编译通过

练习1：完成读文件操作的实现

在`sfs_io_nolock()`函数中实现了文件读取功能，主要思路是分三种情况处理文件读写：

1. **处理起始位置未对齐到块边界的情况**
    - 计算块内偏移量和需要读写的大小

- 通过`sfs_bmap_load_nolock`获取物理块号
- 使用`sfs_buf_op`进行部分块的读写操作

2. **处理中间对齐的完整块**

- 使用循环处理所有完整块
- 每次处理一个完整块，使用`sfs_block_op`进行整块读写

3. **处理末尾未对齐的部分**

- 计算末尾需要读写的大小
- 获取最后一个块的物理块号并进行部分读写

实现代码如下：

```
// (1) 处理起始位置未对齐到块边界的情况
blkoff = offset % SFS_BLKSIZE;
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) 处理中间对齐的完整块
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 += SFS_BLKSIZE;
    buf += SFS_BLKSIZE;
    blkno++;
    nblks--;
}
```

```
    // (3) 处理末尾未对齐的部分
    size = endpos % SFS_BLKSIZE;

    if (size != 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;
    }
```

练习2：完成基于文件系统的执行程序机制的实现

改写了`load_icode`函数和相关函数，实现从文件系统加载并执行程序的功能：

1. 通过文件系统打开并读取可执行文件
2. 解析ELF格式的可执行文件
3. 为新进程创建内存空间并加载程序段
4. 设置进程的初始上下文和栈信息
5. 完成进程的创建和启动

核心函数

`load_icode`函数需要从文件系统加载ELF可执行文件。与lab5相比：

- lab5：程序在内存中，直接memcpy
- lab8：程序在文件系统中，需要通过文件接口读取

`load_icode`的整体流程：

1. 当前进程应该没有内存空间(mm==NULL)，因为do_execve已经释放了
2. 创建新的mm和页目录表
3. 从ELF文件加载代码段、数据段
4. 设置用户栈，把argc/argv放进去
5. 设置trapframe，让进程返回用户态时跳转到程序入口

```
    static int
    load_icode(int fd, int argc, char **kargv)
    {

        if (current->mm != NULL) {
            panic("load_icode: current->mm must be empty.\n");
        }

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

```
// (1) 创建新的mm结构体
// mm_struct管理进程的虚拟内存空间，包含vma链表和页目录
if ((mm = mm_create()) == NULL) {
    goto bad_mm;
}

// (2) 创建新的页目录表
// 复制内核页表，用户程序也能访问内核代码(通过syscall)
if (setup_pgdir(mm) != 0) {
    goto bad_pgdir_cleanup_mm;
}

// (3) 解析ELF文件，加载程序段到内存
struct Page *page;
struct elfhdr __elf, *elf = &__elf;

// (3.1) 从文件读取ELF头(前52字节)
// ELF头包含：魔数、入口地址、程序头表偏移等信息
if ((ret = load_icode_read(fd, elf, sizeof(struct elfhdr), 0)) != 0) {
    goto bad_elf_cleanup_pgdir;
}

// 检查ELF魔数 0x7f454c46 (0x7f E L F)
if (elf->e_magic != ELF_MAGIC) {
    ret = -E_INVAL_ELF;
    goto bad_elf_cleanup_pgdir;
}

// (3.2) 遍历程序头表，加载每个可加载段
// 程序头表描述了ELF文件中的各个段(segment)
struct proghdr __ph, *ph = &__ph;
uint32_t vm_flags, perm, phnum;
for (phnum = 0; phnum < elf->e_phnum; phnum++) {
    // 计算第phnum个程序头在文件中的偏移
    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;
    }

    // 只处理PT_LOAD类型的段（代码段、数据段等）
    // 其他类型如PT_NOTE、PT_DYNAMIC等跳过
    if (ph->p_type != ELF_PT_LOAD) {
        continue;
    }
    // 文件大小不能超过内存大小（BSS段的memsz > filesz）
    if (ph->p_filesz > ph->p_memsz) {
        ret = -E_INVAL_ELF;
        goto bad_cleanup_mmap;
    }
    if (ph->p_filesz == 0) {
        // continue;  // 纯BSS段，没有文件内容
    }
```

```
        // (3.3) 设置虚拟内存区域(VMA)的标志和页表权限
        // vm_flags用于VMA, perm用于页表项
        vm_flags = 0;
        perm = PTE_U | PTE_V;   // 用户态可访问 + 有效位
        if (ph->p_flags & ELF_PF_X) vm_flags |= VM_EXEC;   // 可执行
        if (ph->p_flags & ELF_PF_W) vm_flags |= VM_WRITE;  // 可写
        if (ph->p_flags & ELF_PF_R) vm_flags |= VM_READ;   // 可读

        // 为RISC-V设置页表权限位
        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;
        }

        // (3.4) 分配页面并从文件读取TEXT/DATA内容
        off_t offset = ph->p_offset;
        size_t off, size;
        uintptr_t start = ph->p_va, end, la = ROUNDDOWN(start, PGSIZE);

        ret = -E_NO_MEM;

        // 复制TEXT/DATA段
        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;
            }
            // 从文件读取内容到页面
            if ((ret = load_icode_read(fd, page2kva(page) + off, size, offset)) !=
0) {
                goto bad_cleanup_mmap;
            }
            start += size;
            offset += size;
        }

        // (3.5) 处理BSS段（零填充）
        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);
            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;
        }
    }

    // 文件读完了，关闭它
    sysfile_close(fd);

    // (4) 设置用户栈
    // 栈从USTACKTOP向下增长，大小为USTACKSIZE
    vm_flags = VM_READ | VM_WRITE | VM_STACK;
    if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL)) !=
0) {
        goto bad_cleanup_mmap;
    }
    // 预先分配4个物理页面作为初始栈空间
    // 后续如果栈不够用会触发page fault，再分配
    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);

    // (5) 激活新的内存空间
    // 增加mm的引用计数，设置进程的mm和页目录
    mm_count_inc(mm);
    current->mm = mm;
    current->pgdir = PADDR(mm->pgdir);
    lsatp(PADDR(mm->pgdir));   // 切换到新页表

    // (6) 在用户栈中设置argc和argv
    // 计算所有参数字符串的总长度
    uint32_t argv_size = 0;
    int i;
    for (i = 0; i < argc; i++) {
        argv_size += strnlen(kargv[i], EXEC_MAX_ARG_LEN + 1) + 1;  // +1是'\0'
    }
```

```
    uintptr_t stacktop = USTACKTOP;

    // 在栈顶下面分配参数字符串的空间，8字节对齐
    stacktop -= (argv_size + sizeof(long) - 1) & ~(sizeof(long) - 1);
    char *arg_str = (char *)stacktop;

    // 再往下分配argv指针数组的空间
    stacktop -= argc * sizeof(char *);
    char **uargv = (char **)stacktop;

    // 把参数字符串复制到用户栈，同时设置argv指针
    char *p = arg_str;
    for (i = 0; i < argc; i++) {
        uargv[i] = strcpy(p, kargv[i]);  // strcpy返回目标地址
        p += strnlen(kargv[i], EXEC_MAX_ARG_LEN + 1) + 1;
    }

    // RISC-V ABI要求栈指针16字节对齐
    stacktop = stacktop & ~(uintptr_t)15;

    // (7) 设置trapframe，这决定了sret后CPU的状态
    struct trapframe *tf = current->tf;
    memset(tf, 0, sizeof(struct trapframe));

    tf->gpr.sp = stacktop;       // 用户栈指针
    tf->epc = elf->e_entry;      // 程序入口地址（main之前的_start）

    // status: 清除SPP位(返回用户态)，设置SPIE位(返回后开中断)
    tf->status = (read_csr(sstatus) & ~SSTATUS_SPP) | SSTATUS_SPIE;

    // RISC-V调用约定：a0=第一个参数(argc)，a1=第二个参数(argv)
    tf->gpr.a0 = argc;
    tf->gpr.a1 = (uintptr_t)uargv;

    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;
}
```

实验结果

```
ubuntu@bozhaoxuan:/opt/riscv/lab8$ make grade
er/spin.c + cc user/testbss.c + cc user/waitkill.c + cc user/yield.c create bin/sfs.img (disk0) successfully.
 + cc kern/init/entry.S + cc kern/init/init.c + cc kern/libs/readline.c + cc kern/libs/stdio.c + cc kern/libs
/string.c + cc kern/debug/kdebug.c + cc kern/debug/kmonitor.c + cc kern/debug/panic.c + cc kern/driver/clock.
c + cc kern/driver/console.c + cc kern/driver/dtb.c + cc kern/driver/ide.c + cc kern/driver/intr.c + cc kern/
driver/picirq.c + cc kern/driver/ramdisk.c + cc kern/trap/trap.c + cc kern/trap/trapentry.S + cc kern/mm/defa
ult_pmm.c + cc kern/mm/kmalloc.c + cc kern/mm/pmm.c + cc kern/mm/vmm.c + cc kern/sync/check_sync.c + cc kern/
sync/monitor.c + cc kern/sync/sem.c + cc kern/sync/wait.c + cc kern/fs/file.c + cc kern/fs/fs.c + cc kern/fs/
iobuf.c + cc kern/fs/sysfile.c + cc kern/process/entry.S + cc kern/process/proc.c + cc kern/process/switch.S
+ cc kern/schedule/default_sched.c + cc kern/schedule/default_sched_stride.c + cc kern/schedule/sched.c + cc
kern/syscall/syscall.c + cc kern/fs/swap/swapfs.c + cc kern/fs/vfs/inode.c + cc kern/fs/vfs/vfs.c + cc kern/f
s/vfs/vfsdev.c + cc kern/fs/vfs/vfsfile.c + cc kern/fs/vfs/vfslookup.c + cc kern/fs/vfs/vfspath.c + cc kern/f
s/devs/dev.c + cc kern/fs/devs/dev_disk0.c + cc kern/fs/devs/dev_stdin.c + cc kern/fs/devs/dev_stdout.c + cc
kern/fs/sfs/bitmap.c + cc kern/fs/sfs/sfs.c + cc kern/fs/sfs/sfs_fs.c + cc kern/fs/sfs/sfs_inode.c + cc kern/
fs/sfs/sfs_io.c + cc kern/fs/sfs/sfs_lock.c + ld bin/kernel riscv64-unknown-elf-objcopy bin/kernel --strip-al
l -O binary bin/ucore.img gmake[1]: Leaving directory '/opt/riscv/lab8'
  -sh execve:                                OK
  -user sh :                                 OK
Total Score: 100/100
```

```
ubuntu@bozhaoxuan:/opt/riscv/lab8$ make qemu
check_vmm() succeeded.
sched class: RR_scheduler
Initrd: 0xc0214010 - 0xc021bd0f, size: 0x00007d00
Initrd: 0xc021bd10 - 0xc029100f, size: 0x00075300
sfs: mount: 'simple file system' (106/11/117)
vfs: mount disk0.
++ setup timer interrupts
kernel_execve: pid = 2, name = "sh".
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的PIPE机制设计方案

1. 数据结构设计

```
// 管道缓冲区结构
#define PIPE_BUF_SIZE 4096  // 管道缓冲区大小
struct pipe_buffer {
    char buf[PIPE_BUF_SIZE];  // 实际存储数据的缓冲区
```

```
    int head;                   // 缓冲区头部索引(读指针)
    int tail;                   // 缓冲区尾部索引(写指针)
    int count;                  // 当前数据字节数
};

// 管道控制结构
struct pipe_inode {
    struct inode vnode;         // 继承VFS的inode结构
    struct pipe_buffer buffer;// 管道缓冲区
    semaphore_t sem_read;       // 读同步信号量
    semaphore_t sem_write;      // 写同步信号量
    lock_t pipe_lock;           // 管道操作互斥锁
    int readers;                // 读端数量
    int writers;                // 写端数量
    bool is_closed;             // 管道是否已关闭
};

// 管道文件操作接口集
struct pipe_file_operations {
    int (*read)(struct file *file, struct iobuf *iob);
    int (*write)(struct file *file, struct iobuf *iob);
    int (*close)(struct file *file);
    // 其他必要的文件操作...
};
```

2. 主要接口设计

（1）管道创建接口

```
/**
 * pipe_create - 创建一个新管道
 * @read_fd: 输出参数，管道读端文件描述符
 * @write_fd: 输出参数，管道写端文件描述符
 *
 * 功能：创建一个匿名管道，返回两个文件描述符，
 * 分别用于读和写操作。管道采用FIFO方式工作。
 * 返回0表示成功，负数表示错误码。
 */
int pipe_create(int *read_fd, int *write_fd);
```

（2）管道读写接口

```
/**
 * pipe_read - 从管道读取数据
 * @file: 管道文件结构体
 * @iob: 数据缓冲区
 *
 * 功能：从管道中读取数据到缓冲区。若管道为空且有写者，
```

```
 * 则阻塞等待；若管道为空且无写者，返回0表示EOF。
 * 返回实际读取的字节数，负数表示错误。
 */
int pipe_read(struct file *file, struct iobuf *iob);

/**
 * pipe_write - 向管道写入数据
 * @file: 管道文件结构体
 * @iob: 待写入的数据缓冲区
 *
 * 功能：将数据写入管道。若管道满且有读者，
 * 则阻塞等待；若管道无读者，返回EPIPE错误。
 * 返回实际写入的字节数，负数表示错误。
 */
int pipe_write(struct file *file, struct iobuf *iob);
```

（3）管道关闭接口

```
/**
 * pipe_close - 关闭管道的读端或写端
 * @file: 管道文件结构体
 *
 * 功能：关闭管道的读端或写端，根据文件描述符类型判断。
 * 当读写端都关闭时，释放管道资源。
 * 返回0表示成功，负数表示错误。
 */
int pipe_close(struct file *file);
```

3. 同步互斥处理

1. **互斥机制**：通过`pipe_lock`保证对管道缓冲区的操作是原子的，防止多个进程同时读写导致的数据不一致。
2. **同步机制**：
   - 读操作：当缓冲区为空时，读进程通过`sema_down(&sem_read)`阻塞等待，直到有数据写入
   - 写操作：当缓冲区满时，写进程通过`sema_down(&sem_write)`阻塞等待，直到有数据被读取
   - 数据写入后唤醒读信号量，数据读取后唤醒写信号量
3. **特殊情况处理**：
   - 当所有写端关闭，读进程读取完剩余数据后返回0（EOF）
   - 当所有读端关闭，写进程写入时会收到SIGPIPE信号
   - 采用部分写入机制，当缓冲区空间不足时写入部分数据并返回实际写入字节数

Challenge2：UNIX的软连接和硬连接机制设计方案

1. 数据结构设计

```
// 硬连接相关扩展（inode结构扩展）
struct inode {
    // 原有字段...
    int link_count;           // 硬连接计数，为0时可删除
    // 其他原有字段...
};

// 软连接inode结构
#define SYMLINK_MAX_LEN 1024  // 软连接路径最大长度
struct symlink_inode {
    struct inode vnode;       // 继承VFS的inode结构
    char target_path[SYMLINK_MAX_LEN];  // 目标文件路径
    lock_t symlink_lock;      // 软连接操作锁
};

// 目录项结构扩展（用于支持连接）
struct sfs_disk_entry {
    char name[SFS_MAX_FNAME_LEN];  // 文件名
    uint32_t ino;                  // 对应的inode编号（硬连接共享）
    enum {
        SFS_FILE,
        SFS_DIR,
        SFS_HARDLINK,
        SFS_SYMLINK
    } type;                        // 文件类型，区分连接类型
};
```

2. 主要接口设计

（1）硬连接接口

```
/**
 * link - 创建硬连接
 * @oldpath: 源文件路径
 * @newpath: 硬连接路径
 *
 * 功能：为源文件创建一个新的硬连接，增加inode的link_count。
 * 硬连接与源文件共享同一个inode，删除时仅减少link_count。
 * 返回0表示成功，负数表示错误码。
 */
int link(const char *oldpath, const char *newpath);

/**
 * unlink - 删除硬连接
 * @path: 要删除的硬连接路径
 *
 * 功能：删除指定的硬连接，减少inode的link_count。
 * 当link_count减为0时，真正删除文件数据。
 * 返回0表示成功，负数表示错误码。
```

```
 */
int unlink(const char *path);
```

（2）软连接接口

```
/**
 * symlink - 创建软连接
 * @target: 目标文件路径
 * @linkpath: 软连接路径
 *
 * 功能：创建一个指向目标文件的软连接，自身是一个特殊文件，
 * 存储目标文件的路径信息。
 * 返回0表示成功，负数表示错误码。
 */
int symlink(const char *target, const char *linkpath);

/**
 * readlink - 读取软连接目标路径
 * @path: 软连接路径
 * @buf: 存储目标路径的缓冲区
 * @bufsize: 缓冲区大小
 *
 * 功能：读取软连接指向的目标文件路径。
 * 返回实际读取的字节数，负数表示错误。
 */
ssize_t readlink(const char *path, char *buf, size_t bufsize);
```

（3）路径解析接口（扩展）

```
/**
 * vfs_resolve_path - 解析路径，处理软连接
 * @path: 输入路径
 * @resolved: 输出解析后的绝对路径
 * @max_len: 最大长度限制
 * @follow_links: 是否跟随软连接（1=跟随，0=不跟随）
 *
 * 功能：解析路径，遇到软连接时根据follow_links参数决定是否展开。
 * 处理软连接的循环引用问题。
 * 返回0表示成功，负数表示错误码。
 */
int vfs_resolve_path(const char *path, char *resolved, size_t max_len, int
follow_links);
```

3. 同步互斥处理

1. **硬连接同步**：

- 通过inode的引用计数（link_count）实现资源管理
- 修改link_count时需要加锁保护，防止并发修改导致计数错误
- 删除文件时需检查link_count，确保最后一个引用被删除时才释放资源

2. **软连接同步**：

- 通过`symlink_lock`保证对软连接inode的并发访问安全
- 路径解析时使用计数器限制软连接跟随深度（如8层），防止循环引用导致死循环
- 读取软连接内容时加锁，防止读取过程中被修改

3. **目录操作同步**：

- 对目录项的操作（创建/删除连接）需要加目录锁
- 确保硬连接不能跨越文件系统，软连接可以跨文件系统
- 不允许对目录创建硬连接（保持文件系统结构清晰）

4. **特殊情况处理**：

- 软连接指向的目标文件被删除后，软连接变为"悬空链接"
- 硬连接不能指向目录，避免形成文件系统环
- 跨文件系统操作时，硬连接不被允许，软连接可以创建