操作系统 Lab 8——文件管理系统

2313226 肖俊涛 2312282 张津硕 2311983 余辰民

密码与网络空间安全学院

实验目的

通过完成本次实验，希望能够达到以下目标

- 了解文件系统抽象层-VFS的设计与实现
- 了解基于索引节点组织方式的Simple FS文件系统与操作的设计与实现
- 了解“一切皆为文件”思想的设备文件设计
- 了解简单系统终端的实现

前情回顾

Lab6 在前两章中，我们已经分别实现了内核进程和用户进程，并且让他们正确运行了起来。同时我们也实现了一个简单的调度算法，FIFO调度算法，来对我们的进程进行调度,可通过阅读实验五下的kern/schedule/sched.c 的 schedule 函数的实现来了解其FIFO调度策略。但是，单单如此就够了吗？显然，我们可以让ucore支持更加丰富的调度算法，从而满足各方面的调度需求。与实验五相比，实验六专门需要针对处理器调度框架和各种算法进行设计与实现，为此对ucore的调度部分进行了适当的修改，使得kern/schedule/sched.c 只实现调度器框架，而不再涉及具体的调度算法实现。而调度算法在单独的文件（`default_sched.[ch]`）中实现。在本次实验中，我们在`init/init.c`中加入了对`sched_init`函数的调用。这个函数主要完成调度器和特定调度算法的绑定。初始化后，我们在调度函数中就可以使用相应的接口，切换你实现的不同的调度算法了。这也是在C语言环境下对于面向对象编程模式的一种模仿。这样之后，我们只需要关注于实现调度类的接口即可，操作系统也同样不关心调度类具体的实现，方便了新调度算法的开发。本次实验，主要是熟悉ucore的系统调度器框架，以及基于此框架实现Round-Robin（RR） 调度算法。然后进一步完成Stride调度算法。

Lab7 在本章中我们来实现ucore中的同步互斥机制。在之前的几章中我们已经实现了进程以及调度算法，可以让多个进程并发的执行。在现实的系统当中，有许多多线程的系统都需要协同的完成某一项任务。但是在协同的过程中，存在许多资源共享的问题，比如对一个文件读写的并发访问等等。这些问题需要我们提供一些同步互斥的机制来让程序可以有序的、无冲突的完成他们的工作，这也是这一章内我们要解决的问题。我们最终实现的目标是解决“哲学家就餐问题”。“哲学家就餐问题”是一个非常有名的同步互斥问题：有五个哲学家围成一圈吃饭，每两个哲学家中间有一根筷子。每个需要就餐的哲学家需要两根筷子才可以就餐。哲学家处于两种状态之间：思考和饥饿。当哲学家处于思考的状态时，哲学家便无欲无求；而当哲学家处于饥饿状态时，他必须通过就餐来解决饥饿，重新回到思考的状态。如何让这5个哲学家可以不发生死锁的把这一顿饭吃完就是我们要解决的目标。

Lab8 实验七完成了在内核中的同步互斥实验。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(即实现sfs_io_nolock()函数)，重新实现基于文件系统的执行程序机制（即实现load_icode()函数），从而实现执行存储在磁盘上的文件以及文件读写等功能。与实验七相比，实验八增加了文件系统，并因此实现了通过文件系统来加载可执行文件到内存中运行的功能，导致对进程管理相关的实现比较大的调整。

Lab8 文件系统（Simple FS）项目组成

一级目录	二级目录 / 文件	核心作用	意义
kern	fs	内核态文件系统核心实现	Lab8 的主体，实现 VFS、SimpleFS、设备访问与系统调用
kern	process	进程管理模块	扩展进程以支持文件系统访问与基于文件的程序执行
kern	syscall	系统调用模块	提供用户态访问文件系统的内核接口
kern	init	内核初始化模块	增加文件系统初始化入口，完成 FS 子系统启动
user	*	用户态测试程序	验证文件系统功能正确性
user	libs	用户态文件系统接口库	为测试程序提供文件/目录操作 API
tools	*	辅助工具	构建 SimpleFS 磁盘镜像，支持文件系统实验
libs	*	通用库	提供文件系统相关的基础数据结构与定义

kern/fs —— 文件系统核心模块

1. 通用文件系统接口

目录 / 文件	核心作用	意义说明
kern/fs/fs.[ch]	文件系统初始化与总体管理	作为文件系统子系统入口，负责初始化 VFS、设备与具体文件系统
kern/fs/file.[ch]	内核态文件抽象与操作	提供内核态对文件对象的统一访问接口

目录 / 文件	核心作用	意义说明
kern/fs/sysfile.[ch]	文件系统相关系统调用实现	实现 open/read/write/close 等系统调用的内核逻辑
kern/fs/iobuf.[ch]	I/O 缓冲区抽象	在文件系统与设备之间传递数据，统一读写接口

2. 文件系统抽象层（VFS）

目录 / 文件	核心作用	意义说明
kern/fs/vfs	虚拟文件系统层	屏蔽具体文件系统差异，向上提供统一接口
kern/fs/vfs/vfs.[ch]	VFS 核心逻辑	管理挂载点、文件系统类型及全局操作
kern/fs/vfs/inode.[ch]	inode 抽象	统一不同文件系统中的文件节点表示
kern/fs/vfs/vfsfile.c	文件级操作接口	将文件操作映射到具体文件系统实现
kern/fs/vfs/vfsdev.c	设备节点支持	将设备以文件形式纳入 VFS 管理
kern/fs/vfs/vfslookup.c	路径解析	实现路径到 inode 的解析逻辑
kern/fs/vfs/vfspath.c	路径管理辅助	支持路径拼接与遍历

3. Simple FS（SFS）具体实现

目录 / 文件	核心作用	意义说明
kern/fs/sfs	SimpleFS 实现	Lab8 实验重点，实现一个可工作的磁盘文件系统
kern/fs/sfs/sfs.c	SFS 核心逻辑	实现 SFS 文件系统的整体操作框架
kern/fs/sfs/sfs_fs.c	文件系统级操作	支持挂载、卸载及超级块管理
kern/fs/sfs/sfs_inode.c	inode 管理	实现 SFS 中文件元数据管理
kern/fs/sfs/sfs_io.c	文件读写实现	实现文件数据块的读写逻辑
kern/fs/sfs/sfs_lock.c	并发控制	保证文件系统操作的同步互斥
kern/fs/sfs/bitmap.[ch]	空闲空间管理	管理磁盘块与 inode 的分配与回收

4. 设备层（文件系统 I/O 支撑）

目录 / 文件	核心作用	意义说明
kern/fs/devs	设备文件支持	为文件系统提供统一的设备访问接口
kern/fs/devs/dev.[ch]	设备抽象接口	定义设备读写与注册规范
kern/fs/devs/dev_disk0.c	磁盘设备实现	提供 SimpleFS 所需的底层磁盘 I/O
kern/fs/devs/dev_stdin.c	标准输入设备	将输入设备抽象为文件

目录 / 文件	核心作用	意义说明
kern/fs/devs/dev_stdout.c	标准输出设备	将输出设备抽象为文件

kern/process

文件	核心作用	意义说明
kern/process/proc.[ch]	进程结构与管理	增加 fs_struct 成员，支持进程级文件表
kern/process/entry.S	进程入口	支持用户程序从文件系统加载执行
kern/process/switch.S	上下文切换	保证文件系统相关状态在切换中正确保存

kern/syscall

文件	核心作用	意义说明
kern/syscall/syscall.[ch]	系统调用分发	将用户态文件操作转发至内核文件系统实现

user 目录

目录 / 文件	核心作用	意义说明
user/*.c	文件系统测试程序	验证文件创建、读写、执行等功能
user/sh.c	Shell 程序	通过命令交互测试文件系统完整性
user/libs	用户态文件库	封装文件与目录操作系统调用
user/libs/file.[ch]	文件操作接口	提供 fopen/read/write 等函数
user/libs/dir.[ch]	目录操作接口	支持目录遍历与管理

tools

文件	核心作用	意义说明
tools/mksfs.c	SFS 镜像生成工具	创建 SimpleFS 格式的磁盘镜像
tools/kernel.ld / user.ld	链接脚本	支持内核与用户程序布局
tools/boot.ld	启动链接脚本	支持内核启动

libs

目录	核心作用	意义说明
libs	公共库	提供 inode、stat、dirent 等文件系统基础定义

> 本实验通过引入文件系统子系统，在 uCore 中构建了从磁盘到用户程序执行的完整路径。通过 VFS 抽象层屏蔽具体文件系统实现差异，并以 SimpleFS 作为具体实现示例，实现了文件读写、目录管理以及基于文件系统的程序加载执行。实验同时扩展了进程管理与系统调用模块，使进程能够在文件系统支持下正常运行，标志着 uCore 操作系统原型功能的完整闭环。

练习1: 完成读文件操作的实现（需要编码）

> 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，填写在 kern/fs/sfs/sfs_inode.c中 的sfs_io_nolock()函数，实现读文件中数据的代码。

1.1 理论基础与处理流程

根据指导书，ucore 的文件系统架构采用了 VFS（虚拟文件系统）+ SFS（简单文件系统）的分层设计。

当用户进程调用 read 时，流程如下：

read (用户态) -> sys_read (内核态) -> sysfile_read -> file_read -> vop_read。

这里的 vop_read 是一个函数指针，在 SFS 中它指向 sfs_read，最终调用 sfs_io -> sfs_io_nolock。

SFS 的磁盘布局是我们实现读写的依据：

- Block 大小为 4KB (4096B)。
- 文件索引采用 Direct（直接索引）和 Indirect（间接索引）。
- 我们的任务是在 `sfs_io_nolock` 中，将磁盘上的 Block 数据搬运到内存 buffer 中。

1.2 `sfs_io_nolock` 实现分析

这是文件读写最底层的核心函数。我在实现时，严格按照指导书的提示，将读写过程拆解为三个部分，以处理“非对齐”的情况。

实现思路：

1. **处理起始非对齐部分（Head）**：如果 `offset` 不是 4K 对齐的，我们需要读/写当前 Block 的剩余部分。
2. **处理中间对齐部分（Body）**：中间部分是完整的 Block，可以直接按 Block 读写，效率最高。
3. **处理末尾非对齐部分（Tail）**：处理剩余不足一个 Block 的数据。

关键函数调用：

- `sfs_bmap_load_nolock`：这是核心辅助函数，传入逻辑块号（`blkno`），它会帮我们查 inode 的索引表，找到磁盘上对应的物理块号（`ino`）。
- `sfs_buf_op` / `sfs_block_op`：根据是读还是写，分别指向 `sfs_r/wbuf` 或 `sfs_r/wblock`。

1.3 代码实现

在 `kern/fs/sfs/sfs_inode.c` 中，我填充了 `sfs_io_nolock` 的核心逻辑：

```
static int
sfs_io_nolock(struct sfs_fs *sfs, struct sfs_inode *sin, void *buf, off_t
offset, size_t *alenp, bool write) {
    // ... (前置参数检查与函数指针初始化，略) ...

    int ret = 0;
    size_t size, alen = 0;
    uint32_t ino;
```

```
    uint32_t blkno = offset / SFS_BLKSIZE;          // Rd/Wr 起始块号
    uint32_t nblks = endpos / SFS_BLKSIZE - blkno;  // 需要完整读写的块数

    // LAB8:EXERCISE1 2311983

    // (1) 第一步：处理起始处未对齐的情况
    // 计算 offset 在当前块内的偏移量
    off_t blkoff = offset % SFS_BLKSIZE;

    if (blkoff != 0) {
        // 计算第一块需要操作的数据量：
        // 如果跨块了，就操作到块尾 (SFS_BLKSIZE - blkoff)
        // 如果没跨块，就操作到 endpos - offset
        size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);

        // 1. 通过 inode 找到对应的磁盘物理块号 (ino)
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        // 2. 读/写 这部分非对齐的数据
        if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0) {
            goto out;
        }

        // 更新累加值
        alen += size;
        if (nblks == 0) {
            goto out; // 如果都在这一块内，直接结束
        }
        buf += size;
        blkno++;
        nblks--;
    }

    // (2) 第二步：处理中间完全对齐的数据块
    // 这一步是效率最高的，按整块操作
    while (nblks > 0) {
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        // 调用块操作函数，直接读写一个 Block
        if ((ret = sfs_block_op(sfs, buf, ino, 1)) != 0) {
            goto out;
        }

        alen += SFS_BLKSIZE;
        buf += SFS_BLKSIZE;
        blkno++;
        nblks--;
    }

    // (3) 第三步：处理末尾剩余的非对齐数据
    if ((size = endpos % SFS_BLKSIZE) != 0) {
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
```

```
        }
        // 读写剩余部分，注意这里的 offset (块内偏移) 为 0
        if ((ret = sfs_buf_op(sfs, buf, size, ino, 0)) != 0) {
            goto out;
        }
        alen += size;
    }

out:
    *alenp = alen; // 返回实际操作的字节数
    // 如果是写操作且文件变大了，需要更新 inode 的 size 并标记为 dirty
    if (write && offset + alen > sin->din->size) {
        sin->din->size = offset + alen;
        sin->dirty = 1;
    }
    return ret;
}
```

练习2: 完成基于文件系统的执行程序机制的实现（需要编码）

> 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。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。

2.1 进程创建与初始化的修改 (`alloc_proc`)

在 Lab8 中，`proc_struct` 增加了一个成员 `struct files_struct *filesp`，用于管理进程打开的文件描述符表。在创建新进程时，必须将其初始化为空，否则可能导致野指针访问。

修改代码 (`kern/process/proc.c`):

```
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        // ... (Lab4-Lab6 的初始化代码) ...

        // LAB8 YOUR CODE : (update LAB6 steps)
        // 初始化文件描述符表指针为 NULL
        proc->filesp = NULL;
    }
    return proc;
}
```

2.2 进程复制的修改 (`do_fork`)

当父进程 fork 子进程时，除了内存空间（mm），还需要处理打开的文件。根据 `clone_flags`，子进程可能复制父进程的文件表，也可能共享。我在 `do_fork` 中添加了对 `copy_files` 的调用。

修改代码 (`kern/process/proc.c`):

```
int do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
```

```
    // ... (分配 proc, 检查) ...

    // LAB8:EXERCISE2 2311983
    // 1. 分配 proc
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }
    proc->parent = current;
    assert(current->wait_state == 0);

    // 2. 分配内核栈
    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }

    // 3. 【新增】复制或共享文件系统信息 (fs)
    // 如果 copy_files 失败，需要回滚之前的操作
    if (copy_files(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }

    // 4. 复制内存空间 (mm)
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_fs; // 注意这里要跳到 cleanup_fs
    }

    // ... (后续 copy_thread, hash_proc, wakeup_proc 等逻辑不变) ...

    // 错误处理部分需要增加 cleanup_fs
bad_fork_cleanup_fs:
    put_files(proc); // 释放文件引用
bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
}
```

2.3 进程切换的补充 (`proc_run`)

在 Lab8 中，指导书提示我们在 `switch_to` 之前可能需要刷新 TLB。虽然 ucore 的设计在 `lsatp` 时通常会自动处理，但为了保险起见，或者遵循指导书的特定要求，我在 `proc_run` 中显式加入了 TLB 刷新。

修改代码 (`kern/process/proc.c`):

```
void proc_run(struct proc_struct *proc) {
    if (proc != current) {
        // ... (local_intr_save 等) ...

        current = proc;
        lsatp(proc->pgdir);

        // LAB8 YOUR CODE
        // 切换页表后，应当刷新 TLB 以确保地址翻译正确
```

```
        flush_tlb();

        switch_to(&(prev->context), &(proc->context));
        // ...
    }
}
```

2.4 核心加载机制的重构 (`load_icode`)

在 Lab5 中，我们是直接解析内存中的 ELF 镜像（`binary` 指针）。而在 Lab8 中，`execve` 传入的是一个**文件描述符 (fd)**。

主要区别与实现逻辑：

1. **不再使用 memcpy**：所有的数据读取都必须通过 `load_icode_read(fd, ...)` 函数，它底层调用 `sysfile_seek` 和 `sysfile_read`。
2. **分段读取**：
   - 先读取 `ELF Header`，校验 Magic Number。
   - 根据 Header 中的信息，遍历读取 `Program Header`。
3. **内存映射与内容加载**：
   - 对每个 `PT_LOAD` 类型的段，调用 `mm_map` 建立 VMA。
   - 分配物理页 (`pgdir_alloc_page`)。
   - **关键**：直接调用 `load_icode_read` 将文件内容写入到 `page2kva(page)` (物理页对应的内核虚地址)。这比 Lab5 复杂，因为要处理文件偏移量 `offset`。
4. **参数处理**：处理 `argc` 和 `argv`，将它们拷贝到用户栈的顶部。

完整修改代码 (`kern/process/proc.c`)：

```
// load_icode - called by sys_exec-->do_execve
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) 建立内存管理器
    // 为当前进程创建一个新的 mm 结构
    if ((mm = mm_create()) == NULL) {
        goto bad_mm;
    }

    // (2) 建立页目录
    // 分配并初始化页目录表，mm->pgdir 指向内核虚拟地址
    if (setup_pgdir(mm) != 0) {
        goto bad_pgdir_cleanup_mm;
    }

    // (3) 从文件加载 ELF 格式的二进制代码
    struct elfhdr __elf, *elf = &__elf;
```

```
    // 使用 load_icode_read 从 fd 读取 ELF Header
    if ((ret = load_icode_read(fd, elf, sizeof(struct elfhdr), 0)) != 0) {
        goto bad_elf_cleanup_pgdir;
    }
    // 检查 Magic Number
    if (elf->e_magic != ELF_MAGIC) {
        ret = -E_INVAL_ELF;
        goto bad_elf_cleanup_pgdir;
    }

    struct proghdr __ph, *ph = &__ph;
    uint32_t vm_flags, perm;
    int i;
    // 遍历所有 Program Header
    for (i = 0; i < elf->e_phnum; i ++) {
        off_t phoff = elf->e_phoff + sizeof(struct proghdr) * i;
        // 读取 Program Header
        if ((ret = load_icode_read(fd, ph, sizeof(struct proghdr), phoff)) != 0)
{
            goto bad_cleanup_mmap;
        }
        // 只处理 LOAD 类型的段
        if (ph->p_type != ELF_PT_LOAD) {
            continue ;
        }
        if (ph->p_filesz > ph->p_memsz) {
            ret = -E_INVAL_ELF;
            goto bad_cleanup_mmap;
        }
        if (ph->p_memsz == 0) {
            continue ;
        }

        // 设置虚拟内存权限标志
        vm_flags = 0, 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;
        // RISC-V 特有的权限位设置
        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;

        // (3.3) 建立 VMA (Virtual Memory Area)
        if ((ret = mm_map(mm, ph->p_va, ph->p_memsz, vm_flags, NULL)) != 0) {
            goto bad_cleanup_mmap;
        }

        // (3.4) 分配物理内存并读取文件内容
        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;
```

```
        end = ph->p_va + ph->p_filesz;
        struct Page *page = NULL;

        // 循环处理每一页
        while (start < end) {
            // 分配物理页
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
                goto bad_cleanup_mmap;
            }
            off = start - la, size = PGSIZE - off, la += PGSIZE;
            if (end < la) {
                size -= la - end;
            }
            // 关键：调用 load_icode_read 将文件数据读入刚才分配的内存页中
            // page2kva(page) + off 是内核虚拟地址
            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（将剩余部分清零）
        end = ph->p_va + ph->p_memsz;
        if (start < la) {
            if (start < end) {
                off = start + PGSIZE - la, size = PGSIZE - off;
                if (end < la) {
                    size -= la - end;
                }
                memset(page2kva(page) + off, 0, size);
                start += size;
            }
        }
        while (start < end) {
            if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL) {
                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;
        }
    }

    // (4) 建立用户栈
    // 映射用户栈空间 (USTACK)
    vm_flags = VM_READ | VM_WRITE | VM_STACK;
    if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL))
!= 0) {
        goto bad_cleanup_mmap;
    }
    // 为用户栈分配物理页 (4页)
```

```
    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) 设置当前进程的 mm 和页目录基址 (CR3/satp)
    mm_count_inc(mm);
    current->mm = mm;
    current->pgdir = PADDR(mm->pgdir);
    lsatp(PADDR(mm->pgdir)); // 切换页表
    flush_tlb();             // 刷新 TLB

    // (6) 处理用户参数 (argc, argv) 并压入用户栈
    uintptr_t stacktop = USTACKTOP;
    uintptr_t uargv_ptrs[EXEC_MAX_ARG_NUM + 1];

    // 将具体参数字符串拷贝到栈中
    for (i = 0; i < argc; i ++) {
        size_t len = strlen(kargv[i]) + 1;
        stacktop -= len;
        uargv_ptrs[i] = stacktop;
        if (!copy_to_user(mm, (void *)stacktop, kargv[i], len)) {
            ret = -E_INVAL;
            goto bad_cleanup_mmap;
        }
    }
    uargv_ptrs[argc] = 0;

    // 对齐栈顶
    stacktop -= (stacktop % sizeof(uintptr_t));
    // 将参数指针数组拷贝到栈中
    stacktop -= (argc + 1) * sizeof(uintptr_t);
    if (!copy_to_user(mm, (void *)stacktop, uargv_ptrs, (argc + 1) *
sizeof(uintptr_t))) {
        ret = -E_INVAL;
        goto bad_cleanup_mmap;
    }

    // (7) 设置 Trapframe
    struct trapframe *tf = current->tf;
    uintptr_t sstatus = tf->status;
    memset(tf, 0, sizeof(struct trapframe));

    tf->gpr.sp = stacktop;     // 设置栈指针
    tf->epc = elf->e_entry;    // 设置入口地址 (ELF Entry)
    // 设置状态寄存器：清除 SPP (Supervisor Previous Privilege) 以便 iret 后回到用户态
    // 设置 SPIE (Supervisor Previous Interrupt Enable) 以便开启中断
    tf->status = (sstatus | SSTATUS_SPIE) & ~SSTATUS_SPP;
    tf->gpr.a0 = argc;         // 传递参数 argc
    tf->gpr.a1 = stacktop;     // 传递参数 argv

    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;
}
```

如图所示，最终在 `make qemu` 中，Shell 成功启动并能执行 `hello` 等命令，验证了我的代码逻辑是正确的。

```
root@Edison:/home/oslab/labcode/lab8# su - oslab
oslab@Edison:~$ cd labcode/lab8
oslab@Edison:~/labcode/lab8$ make qemu
+ cc kern/process/proc.c
+ cc kern/fs/sfs/sfs_inode.c
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img

OpenSBI v0.4 (Jul  2 2019 11:53:53)
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xc020004a (virtual)
  etext  0xc020b504 (virtual)
  edata  0xc0291060 (virtual)
  end    0xc0296910 (virtual)
Kernel executable memory footprint: 603KB
DTB Init
HartID: 0
DTB Address: 0x82200000
Physical Memory from DTB:
  Base: 0x0000000080000000
  Size: 0x0000000008000000 (128 MB)
  End:  0x0000000087ffffff
DTB init completed
memory management: default_pmm_manager
physcial memory map:
  memory: 0x08000000, [0x80000000, 0x87ffffff].
vapaofset is 18446744070488326144
check_alloc_page() succeeded!
Page table directory switch succeeded!
Kernel stack guardians set succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
use SLOB allocator
kmalloc_init() succeeded!
check_vma_struct() succeeded!
check_vmm() succeeded.
sched class: RR_scheduler
Initrd: 0xc0214010 - 0xc021bd0f, size: 0x00007d00
Initrd: 0xc021bd10 - 0xc029100f, size: 0x00075300
sfs: mount: 'simple file system' (111/6/117)
```

```
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.
fork ok.
badarg pass.
```

补充说明：关于 `badsegment` 和 `divzero` 测试用例在 RISC-V 架构下的行为分析

在完成练习二并执行 `make qemu` 进行用户程序测试时，我们观察到只有 `badsegment` 和 `divzero` 两个测试点输出了 "FAIL: T.T" 并触发了 panic，其他均正确通过。这并非内核实现的逻辑错误，而是由于我们使用的 ucore 是基于 **RISC-V 架构**，其硬件特性与原本针对 x86 架构编写的测试用例存在差异。具体分析如下：

1. `badsegment` 测试失败原因

```c
#include <stdio.h>
#include <ulib.h>

/* try to load the kernel's TSS selector into the DS register */

int
main(void) {
    // asm volatile("movw $0x28,%ax; movw %ax,%ds");
    panic("FAIL: T.T\n");
}
```

- **现象**：程序顺利执行到了 `panic("FAIL: T.T")`，意味着并没有发生预期的“非法访问”导致的进程终止。
- **原因**：原有的 `badsegment.c` 测试代码通常包含 x86 汇编指令（如 `movw %ax, %ds`），试图通过修改段寄存器来触发保护异常。然而，**RISC-V 架构中不存在段寄存器（Segment Registers）**，它是纯粹基于页表（Paging）的内存保护机制。原本的测试代码在 RISC-V 下要么无法编译（指令集不兼容），要么被注释掉，导致程序没有触发任何异常，从而继续执行到了 panic 语句。
- **修正逻辑**：在 RISC-V 下，要模拟“坏段”或非法内存访问，应当尝试让用户进程访问其权限之外的地址（例如内核空间地址）。
    - **修改方案**：将测试逻辑修改为向内核地址（如 `0xC0000000`）写入数据。这将触发 **Store Page Fault**，导致内核杀死该进程，从而通过测试（即不再输出 FAIL）。

2. `divzero` 测试失败原因

```
#include <stdio.h>
#include <ulib.h>

int zero;

int
main(void) {
    cprintf("value is %d.\n", 1 / zero);
    panic("FAIL: T.T\n");
}
```

- **现象**：控制台输出 `value is -1`，随即输出 `FAIL: T.T`。
- **原因**：这是 RISC-V 指令集架构（ISA）的硬件特性决定的。与 x86 架构在除以零时会触发硬件异常（Trap）不同，**RISC-V 规范规定整数除以零不会产生异常。**
  - 根据 RISC-V 标准，除以零的结果被定义为除数的所有位全为 1（即在补码表示下返回 `-1`）。
  - 因此，CPU 不会陷入异常，内核也就无法捕获并杀死进程。程序只是计算出了 `-1`，打印输出，然后按顺序执行下一行代码，即 `panic("FAIL: T.T")`。
- **结论**：此处的 FAIL 实际上证明了我们的系统符合 RISC-V 的硬件规范。若需在 RISC-V 上强行测试“异常终止”功能，需在代码中手动检查除数是否为零并触发非法指令或访问空指针。

综上所述，这两个测试点的报错是由于测试用例代码未针对 RISC-V 硬件特性进行适配所致，并非 Lab8 文件系统或进程加载机制的代码错误。

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

> 如果要在ucore里加入UNIX的管道（Pipe）机制，至少需要定义哪些数据结构和接口？（接口给出语义即可，不必具体实现。数据结构的设计应当给出一个（或多个）具体的C语言struct定义。在网络上查找相关的Linux资料和实现，请在实验报告中给出设计实现”UNIX的PIPE机制“的概要设方案，你的设计应当体现出对可能出现的同步互斥问题的处理。）

1.1 概述

管道（Pipe）是UNIX系统中一种重要的进程间通信（IPC）机制，它允许一个进程把数据写进去，另一个进程从中读出来。管道是一个单向的、先进先出（FIFO）的字节流，具有固定大小的缓冲区。

1.2 数据结构设计

在ucore中，管道作为一个特殊的文件类型来实现。需要定义以下数据结构：

```
/* 管道缓冲区大小，为4KB */
#define PIPE_BUF_SIZE 4096

/* 管道结构体 */
struct pipe {
    char buffer[PIPE_BUF_SIZE];      /* 管道数据缓冲区 */
    uint32_t read_pos;               /* 读位置指针 */
    uint32_t write_pos;              /* 写位置指针 */
    uint32_t count;                  /* 当前缓冲区中的数据量 */
    bool read_open;                  /* 读端是否打开 */
```

```
    bool write_open;                    /* 写端是否打开 */
    semaphore_t mutex;                  /* 互斥锁，保护管道结构 */
    wait_queue_t read_wait_queue;       /* 读等待队列 */
    wait_queue_t write_wait_queue;      /* 写等待队列 */
    struct inode *inode;                /* 关联的inode */
    int ref_count;                      /* 引用计数 */
};

/* 扩展inode结构 */
struct pipe_inode_info {
    struct pipe *pipe;                  /* 指向管道结构 */
};
```

设计说明：

- `buffer`：固定大小的环形缓冲区，用于存储管道数据
- `read_pos` / `write_pos`：使用环形缓冲区，通过模运算实现循环
- `count`：记录当前缓冲区中的数据量，用于判断空/满状态
- `read_open` / `write_open`：标识读写端状态，当写端关闭且缓冲区为空时，读端返回EOF
- `mutex`：保护管道结构的互斥访问
- `read_wait_queue` / `write_wait_queue`：当管道为空/满时，阻塞等待的进程队列

1.3 接口设计

1.3.1 系统调用接口

```
/*
 * 创建管道
 * 参数：pipefd[2] - 返回两个文件描述符，pipefd[0]为读端，pipefd[1]为写端
 * 返回：成功返回0，失败返回错误码
 */
int sys_pipe(int pipefd[2]);


/*
 * 从管道读端读取数据
 * 参数：fd - 文件描述符（必须是管道的读端）
 *       buf - 用户缓冲区
 *       count - 要读取的字节数
 * 返回：成功返回实际读取的字节数，失败返回错误码
 * 语义：
 *   - 如果管道为空且写端打开，阻塞直到有数据可读
 *   - 如果管道为空且写端关闭，返回0（EOF）
 *   - 如果管道非空，读取min(count, 可用数据)字节
 */
int sys_read(int fd, void *buf, size_t count);



/*
 * 向管道写端写入数据
 * 参数：fd - 文件描述符（必须是管道的写端）
 *       buf - 用户缓冲区
 *       count - 要写入的字节数
```

```
 * 返回：成功返回实际写入的字节数，失败返回错误码
 * 语义：
 *   - 如果管道满且读端打开，阻塞直到有空间可写
 *   - 如果读端关闭，写入会触发SIGPIPE信号（或返回EPIPE错误）
 *   - 如果管道有空间，写入min(count, 可用空间)字节
 */
int sys_write(int fd, const void *buf, size_t count);


/*
 * 关闭管道的一端
 * 参数：fd - 文件描述符
 * 返回：成功返回0，失败返回错误码
 * 语义：
 *   - 关闭读端：减少引用计数，如果引用计数为0则释放管道资源
 *   - 关闭写端：设置write_open=false，唤醒所有等待的读进程
 */
int sys_close(int fd);
```

1.3.2 内核内部接口

```
/* 创建管道对象 */
struct pipe *pipe_create(void);

/* 释放管道对象 */
void pipe_destroy(struct pipe *pipe);

/* 从管道读取数据（内核实现） */
int pipe_read(struct pipe *pipe, char *buf, size_t count);

/* 向管道写入数据（内核实现） */
int pipe_write(struct pipe *pipe, const char *buf, size_t count);

/* 增加管道引用计数 */
void pipe_ref_inc(struct pipe *pipe);

/* 减少管道引用计数 */
void pipe_ref_dec(struct pipe *pipe);
```

1.4 同步互斥问题处理

1.4.1 互斥访问

问题： 多个进程可能同时访问同一个管道，需要保证对管道结构的访问是原子的。

解决方案：

- 使用信号量`mutex`保护整个管道结构
- 所有对`read_pos`、`write_pos`、`count`等共享变量的访问都需要先获取`mutex`
- 使用现有的`semaphore_t`机制，通过`down()`和`up()`实现互斥

实现示例：

```
int pipe_read(struct pipe *pipe, char *buf, size_t count) {
```

```
    down(&pipe->mutex);  // 获取互斥锁

    // 检查管道状态
    while (pipe->count == 0 && pipe->write_open) {
        // 管道为空且写端打开，进入等待队列
        wait_current_set(&pipe->read_wait_queue, &wait, WT_PIPE);
        up(&pipe->mutex);
        schedule();
        down(&pipe->mutex);
    }

    // 读取数据...
    // 更新read_pos, count等

    up(&pipe->mutex);  // 释放互斥锁
    return n;
}
```

1.4.2 同步机制

问题1： 当管道为空时，读进程应该阻塞等待；当管道满时，写进程应该阻塞等待。

解决方案：

- 使用等待队列（`wait_queue_t`）实现阻塞/唤醒机制
- 当管道为空且写端打开时，读进程进入`read_wait_queue`并调用`schedule()`
- 当管道满且读端打开时，写进程进入`write_wait_queue`并调用`schedule()`
- 当有数据写入时，唤醒`read_wait_queue`中的进程
- 当有数据读出时，唤醒`write_wait_queue`中的进程

问题2： 写端关闭后，读进程应该能够读取剩余数据，然后返回EOF。

解决方案：

- 检查`write_open`标志
- 如果`write_open == false`且`count == 0`，返回0（EOF）
- 如果`write_open == false`但`count > 0`，继续读取剩余数据

问题3： 读端关闭后，写进程应该收到错误信号。

解决方案：

- 检查`read_open`标志
- 如果`read_open == false`，返回`-EPIPE`错误

1.4.3 原子性保证

问题： 需要保证读写操作的原子性，避免部分读写导致数据不一致。

解决方案：

- 在`mutex`保护下，一次性完成整个读写操作
- 对于大于`PIPE_BUF_SIZE`的写入，需要分多次进行，每次都在锁保护下完成

1.5 实现要点

1. **环形缓冲区实现：**

```
// 写入
uint32_t space = PIPE_BUF_SIZE - pipe->count;
uint32_t to_write = min(count, space);
uint32_t first_part = min(to_write, PIPE_BUF_SIZE - pipe->write_pos);
memcpy(pipe->buffer + pipe->write_pos, buf, first_part);
if (to_write > first_part) {
    memcpy(pipe->buffer, buf + first_part, to_write - first_part);
}
pipe->write_pos = (pipe->write_pos + to_write) % PIPE_BUF_SIZE;
pipe->count += to_write;
```

2. **与VFS集成：**
   - 在`inode`结构中添加`pipe_inode_info`类型
   - 实现管道的`vop_read`、`vop_write`、`vop_close`操作
   - 在`file_pipe()`函数中创建管道并分配文件描述符
3. **进程间共享：**
   - 通过`fork()`时复制文件描述符表，父子进程共享同一个管道
   - 使用引用计数管理管道的生命周期

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

> 如果要在ucore里加入UNIX的软连接和硬连接机制，至少需要定义哪些数据结构和接口？（接口给出语义即可，不必具体实现。数据结构的设计应当给出一个（或多个）具体的C语言struct定义。在网络上查找相关的Linux资料和实现，请在实验报告中给出设计实现”UNIX的软连接和硬连接机制“的概要设方案，你的设计应当体现出对可能出现的同步互斥问题的处理。）

2.1 概述

链接机制允许一个文件有多个名称，提供了文件系统的灵活性和便利性。UNIX系统支持两种链接：

- **硬链接（Hard Link）**：多个目录项指向同一个inode，共享相同的文件数据
- **软链接（Symbolic Link/Symlink）**：一个特殊的文件，其内容是指向目标文件的路径字符串

2.2 数据结构设计

2.2.1 硬链接

硬链接的实现相对简单，只需要在现有的inode结构中维护链接计数。在SFS文件系统中，`sfs_disk_inode`已经包含了`nlinks`字段：

```
/* SFS磁盘inode（已存在，见sfs.h） */
struct sfs_disk_inode {
    uint32_t size;                  /* 文件大小 */
    uint16_t type;                  /* 文件类型 */
    uint16_t nlinks;                /* 硬链接计数 */
    uint32_t blocks;                /* 块数 */
    uint32_t direct[SFS_NDIRECT];   /* 直接块 */
    uint32_t indirect;              /* 间接块 */
};
```

设计说明：

- `nlinks`字段记录指向该inode的目录项数量
- 创建硬链接时，`nlinks++`
- 删除硬链接时，`nlinks--`
- 当`nlinks == 0`时，可以删除inode和文件数据

2.2.2 软链接

软链接需要扩展inode结构，添加符号链接类型和存储目标路径：

```
/* 软链接inode信息 */
struct symlink_inode_info {
    char *target_path;              /* 目标路径字符串 */
    size_t target_len;              /* 目标路径长度 */
    semaphore_t mutex;              /* 保护target_path的互斥锁 */
};

/* 扩展inode类型枚举（在inode.h中） */
enum {
    inode_type_device_info = 0x1234,
    inode_type_sfs_inode_info,
    inode_type_symlink_info,        /* 新增：符号链接类型 */
};

/* 扩展inode union（在inode.h中） */
struct inode {
    union {
        struct device __device_info;
        struct sfs_inode __sfs_inode_info;
        struct symlink_inode_info __symlink_info;  /* 新增 */
    } in_info;
    enum {
        inode_type_device_info = 0x1234,
        inode_type_sfs_inode_info,
        inode_type_symlink_info,    /* 新增 */
    } in_type;
    int ref_count;
    int open_count;
    struct fs *in_fs;
    const struct inode_ops *in_ops;
};
```

设计说明：

- `target_path`：存储目标文件的路径字符串（可以是绝对路径或相对路径）
- `target_len`：路径长度，用于边界检查
- `mutex`：保护`target_path`的并发访问
- 软链接本身是一个文件，但类型为`SFS_TYPE_LINK`（在SFS中已定义）

2.2.3 目录项结构

目录项（dentry）在SFS中已经定义：

```
/* SFS磁盘目录项（已存在，见sfs.h） */
struct sfs_disk_entry {
    uint32_t ino;                    /* inode号 */
    char name[SFS_MAX_FNAME_LEN + 1]; /* 文件名 */
};
```

设计说明：

- 硬链接：多个`dentry`可以指向同一个`ino`
- 软链接：`dentry`指向一个类型为`SFS_TYPE_LINK`的inode

2.3 接口设计

2.3.1 系统调用接口

```
/*
 * 创建硬链接
 * 参数: oldpath - 已存在文件的路径
 *       newpath - 新链接的路径
 * 返回: 成功返回0，失败返回错误码
 * 语义:
 *   - 在newpath的父目录中创建新的目录项
 *   - 新目录项的ino指向oldpath的inode
 *   - 增加目标inode的nlinks计数
 *   - 硬链接不能跨文件系统
 *   - 不能对目录创建硬链接（避免循环）
 */
int sys_link(const char *oldpath, const char *newpath);


/*
 * 创建软链接
 * 参数: target - 目标文件路径（可以是相对或绝对路径）
 *       linkpath - 符号链接的路径
 * 返回: 成功返回0，失败返回错误码
 * 语义:
 *   - 创建新文件linkpath，类型为符号链接
 *   - 将target路径字符串存储到linkpath的inode中
 *   - 软链接可以跨文件系统
 *   - 可以对目录创建软链接
 */
int sys_symlink(const char *target, const char *linkpath);


/*
```

```
 * 读取软链接的目标路径
 * 参数: pathname - 符号链接的路径
 *       buf - 存储目标路径的缓冲区
 *       bufsiz - 缓冲区大小
 * 返回: 成功返回实际读取的字节数，失败返回错误码
 * 语义:
 *   - 如果pathname不是符号链接，返回EINVAL
 *   - 读取符号链接的target_path到buf中
 *   - 不跟随符号链接（与read()不同）
 */
int sys_readlink(const char *pathname, char *buf, size_t bufsiz);


/*
 * 删除链接（硬链接或软链接）
 * 参数: pathname - 要删除的链接路径
 * 返回: 成功返回0，失败返回错误码
 * 语义:
 *   - 如果是硬链接：删除目录项，减少inode的nlinks计数
 *   - 如果是软链接：直接删除符号链接文件
 *   - 如果nlinks减为0且没有进程打开该文件，删除inode和数据
 *   - 不能删除非空目录
 */
int sys_unlink(const char *pathname);
```

2.3.2 VFS层接口

```
/* VFS层已定义的接口（见vfs.h） */
int vfs_link(char *old_path, char *new_path);
int vfs_symlink(char *old_path, char *new_path);
int vfs_readlink(char *path, struct iobuf *iob);
int vfs_unlink(char *path);
```

2.3.3 文件系统层接口

```
/* SFS文件系统层接口 */
int sfs_link(struct inode *dir, const char *name, struct inode *target);
int sfs_symlink(struct inode *dir, const char *name, const char *target_path);
int sfs_readlink(struct inode *node, struct iobuf *iob);
int sfs_unlink(struct inode *dir, const char *name);
```

2.3.4 inode操作扩展

需要在`inode_ops`中添加符号链接相关的操作：

```
struct inode_ops {
    // ... 现有操作 ...

    /* 读取符号链接目标路径 */
    int (*vop_readlink)(struct inode *node, struct iobuf *iob);

    /* 创建硬链接 */
    int (*vop_link)(struct inode *dir, const char *name, struct inode *target);

    /* 创建软链接 */
    int (*vop_symlink)(struct inode *dir, const char *name, const char *target);
};
```

2.4 同步互斥问题处理

2.4.1 硬链接的同步互斥

问题1： 多个进程同时创建/删除硬链接，可能导致`nlinks`计数错误。

解决方案：

- 在SFS中已经定义了`mutex_sem`（见`sfs_fs`结构），用于保护`link/unlink`操作
- 所有修改`nlinks`的操作都需要先获取`mutex_sem`
- 使用原子操作或信号量确保`nlinks++`和`nlinks--`的原子性

实现示例：

```
int sfs_link(struct inode *dir, const char *name, struct inode *target) {
    struct sfs_fs *sfs = vop_fs(dir);
    lock_sfs_mutex(sfs);  // 获取互斥锁

    // 创建目录项
    // ...

    // 原子性更新nlinks
    target_din->nlinks++;
    sfs_sync_inode(target);  // 同步到磁盘

    unlock_sfs_mutex(sfs);  // 释放互斥锁
    return 0;
}
```

问题2： 删除文件时，需要检查`nlinks`，但可能有并发删除操作。

解决方案：

- 在`unlink`操作中，先获取锁，检查`nlinks`
- 如果`nlinks > 1`，只删除目录项并减少计数
- 如果`nlinks == 1`，需要检查是否有进程打开该文件（通过`open_count`）
- 只有当`nlinks == 0`且`open_count == 0`时，才真正删除inode和数据块

问题3： 目录操作的原子性。

解决方案：

- 目录的创建和删除操作需要在锁保护下进行

- 使用事务性操作，确保目录项的添加/删除是原子的
- 如果操作失败，需要回滚已做的修改

2.4.2 软链接的同步互斥

问题1： 多个进程同时读取/修改软链接的`target_path`。

解决方案：

- 在`symlink_inode_info`中使用`mutex`保护`target_path`
- 读取`target_path`时也需要获取锁（虽然读操作通常不需要互斥，但为了一致性）

实现示例：

```
int sfs_readlink(struct inode *node, struct iobuf *iob) {
    struct symlink_inode_info *sinfo = vop_info(node, symlink);

    down(&sinfo->mutex);  // 获取锁
    // 读取target_path到iob
    size_t to_copy = min(sinfo->target_len, iob->io_resid);
    memcpy(iob->io_base, sinfo->target_path, to_copy);
    iob->io_resid -= to_copy;
    up(&sinfo->mutex);  // 释放锁

    return to_copy;
}
```

问题2： 软链接解析时的循环检测。

解决方案：

- 在路径解析时，维护一个已访问的inode集合
- 如果遇到已访问的符号链接，说明存在循环，返回`ELOOP`错误
- 限制符号链接解析的最大深度（如32层）

实现示例：

```
#define MAX_SYMLINK_DEPTH 32

int resolve_symlink(struct inode *node, char *path, int depth) {
    if (depth > MAX_SYMLINK_DEPTH) {
        return -ELOOP;  // 循环链接
    }

    // 读取符号链接目标
    // 递归解析目标路径
    // ...
}
```

问题3： 软链接创建时的路径验证。

解决方案：

- 验证`target_path`的长度不超过`PATH_MAX`
- 验证路径字符串的有效性
- 在创建时不需要验证目标是否存在（软链接可以指向不存在的文件）

2.4.3 文件系统操作的原子性

问题： 创建/删除链接时，需要同时修改目录和inode，需要保证原子性。

解决方案：

- 使用两阶段提交：
    1. 第一阶段：在内存中完成所有修改
    2. 第二阶段：同步到磁盘
- 如果同步失败，需要能够回滚
- 使用日志（journal）机制记录操作，支持恢复

2.5 实现要点

1. **硬链接实现流程：**
    1. 解析oldpath，获取目标inode
    2. 检查目标是否为目录（不能对目录创建硬链接）
    3. 检查是否跨文件系统（硬链接不能跨文件系统）
    4. 获取锁（mutex_sem）
    5. 在newpath的父目录中创建新目录项，指向目标inode
    6. 增加目标inode的nlinks计数
    7. 同步目录和inode到磁盘
    8. 释放锁
2. **软链接实现流程：**
    1. 解析linkpath的父目录
    2. 创建新inode，类型为SFS_TYPE_LINK
    3. 分配symlink_inode_info结构
    4. 将target路径字符串存储到symlink_inode_info中
    5. 在父目录中创建目录项，指向新inode
    6. 同步到磁盘
3. **路径解析中的符号链接处理：**
    - 在 `vfs_lookup` 或 `sfs_lookup` 中，如果遇到符号链接类型的inode
    - 读取 `target_path`
    - 递归调用路径解析函数
    - 检测循环并限制深度
4. **与现有VFS的集成：**
    - 扩展 `inode_ops`，添加 `vop_readlink`、`vop_link`、`vop_symlink`
    - 在SFS中实现这些操作
    - 在VFS层调用文件系统特定的实现

本学期uCore总结

Lab1： 最小可执行内核与启动流程

本实验完成了基于 RISC-V 架构的最小可执行内核构建，并成功在 QEMU 模拟环境下通过 OpenSBI 固件启动内核。实验过程中，通过编写启动汇编代码，实现了内核执行前的基础环境初始化，包括栈空间设置和控制权从固件向内核的转移。同时，结合链接脚本对内核的代码段、数据段及未初始化数据段进行了合理布局，确保内核在内存中的正确装载与运行。

通过本实验，深入理解了 RISC-V 系统的启动流程及各阶段的职责划分，掌握了内核启动过程中汇编与 C 语言代码协同工作的方式，为后续内核功能的逐步扩展奠定了基础。

Lab2： 物理内存管理与页表机制

在实验一的基础上，本实验实现了操作系统对物理内存的初步管理，并建立了基本的虚拟内存映射机制。实验首先完成了对系统物理内存的探测与初始化，在此基础上实现了连续物理内存页分配器，使内核能够对物理内存进行统一分配与回收管理。随后，基于 RISC-V Sv39 规范构建了三级页表结构，完成了内核虚拟地址空间到物理地址空间的映射。

通过本实验，加深了对物理内存管理策略和分页机制的理解，掌握了虚拟内存与物理内存之间映射关系的建立过程，为进程地址空间管理和用户态程序运行提供了必要的内存基础。

Lab3： 中断与异常处理机制

本实验围绕 RISC-V 架构下的中断与异常处理机制展开，构建了完整的陷入（Trap）处理流程。通过设计 TrapFrame 结构体，实现了在中断或异常发生时对处理器上下文的保存与恢复；通过编写汇编级陷入入口代码，完成了从硬件自动跳转到内核异常处理函数的衔接。实验中重点实现了断点异常和时钟中断的处理逻辑，并验证了中断返回后程序能够正确继续执行。

本实验使系统具备了响应外部事件和异常的能力，为后续实现抢占式调度、多进程并发执行等机制提供了必要的底层支持，同时也加深了对体系结构与操作系统协作关系的理解。

Lab4： 进程与内核线程管理

在前述内存管理和中断机制的基础上，本实验实现了内核级线程管理和调度机制，使系统具备了多执行流并发运行的能力。实验中引入了线程控制块，用于描述线程的运行状态、上下文信息及资源占用情况；通过实现上下文切换机制，使处理器能够在不同线程之间切换执行；同时设计并实现了基本的调度器框架，实现了多线程的轮流运行。

此外，本实验进一步完善了虚拟内存管理结构，为每个执行实体提供独立的逻辑地址空间布局。通过本实验，系统从单一执行流模式演进为支持并发执行的多任务系统，初步体现了操作系统对处理器资源的统一调度与管理能力。

Lab5： 用户程序与系统调用机制

本实验在内核线程机制的基础上，引入了用户态进程，完成了从仅支持内核态执行到支持用户态程序运行的转变。实验中构造了第一个用户进程，并通过实现系统调用机制，使用户进程能够在需要内核服务时安全地陷入内核态执行。具体实现了 fork、exec、exit 和 wait 等系统调用，完成了用户进程创建、程序加载、退出及父子进程同步等基本功能。

通过本实验，深入理解了用户态与内核态之间的隔离机制以及特权级切换流程，掌握了用户进程生命周期的基本管理方法，为后续调度优化和资源管理提供了完整的进程模型。

Lab6： 调度算法设计与实现

本实验对调度模块进行了重构，将调度器框架与具体调度算法解耦，提高了系统调度机制的可扩展性。通过引入调度类接口，操作系统内核仅依赖统一的调度框架，而具体的调度策略在独立模块中实现。实验中基于该框架分别实现了 Round-Robin 调度算法和 Stride 调度算法，并通过运行结果验证了不同算法在公平性和调度策略上的差异。

本实验加深了对操作系统调度机制设计思想的理解，使系统具备了支持多种调度策略的能力，同时也体现了模块化设计和面向接口编程在操作系统中的应用价值。

Lab7： 同步与互斥机制

在系统已支持多进程并发执行的基础上，本实验实现了同步与互斥机制，以保证并发环境下共享资源访问的正确性。实验中通过实现锁和信号量等同步原语，解决了多个执行流同时访问临界区所可能引发的数据不一致问题。在此基础上，完成了经典的哲学家就餐问题，验证了所实现同步机制在避免死锁和保证系统正确性方面的有效性。

通过本实验，深入理解了并发执行中临界区、互斥访问及死锁产生条件等概念，提高了对并发程序设计与验证的能力。

Lab8： 文件系统与基于文件的程序执行

本实验在同步互斥机制的基础上，引入了文件系统支持，进一步完善了操作系统的功能结构。通过分析并实现简单文件系统（SFS）的读写接口，实现了对磁盘上文件的访问能力；同时重新实现了基于文件系统的程序加载机制，使用户程序能够从磁盘加载到内存中执行。由于程序加载方式的变化，实验中对进程管理相关模块进行了相应调整，以适配基于文件的执行模型。

通过本实验，系统从仅能运行内存中固定程序，发展为支持文件存储与动态加载程序的完整操作系统原型，进一步加深了对文件系统与进程管理之间关系的理解。

编号	实验名称	实验目标	核心机制	实验重点
Lab1	最小可执行内核与启动流程	构建可在 QEMU + OpenSBI 环境下启动的最小 RISC-V 内核，完成从上电到 C 语言内核入口的控制权转移	实现启动汇编代码，完成栈初始化与特权级切换；通过链接脚本完成内核内存布局；理解 QEMU、OpenSBI 与内核的启动协作关系	掌握 RISC-V 启动流程、内核加载机制与链接脚本原理，为后续内核开发奠定基础
Lab2	物理内存管理与页表机制	实现对物理内存的统一管理，并建立基本的虚拟内存映射机制	探测并初始化物理内存；实现连续物理页分配器；构建 RISC-V Sv39 三级页表；完成内核虚拟地址到物理地址的映射	理解物理内存管理策略与分页机制，掌握虚拟内存到物理内存的映射原理

编号	实验名称	实验目标	核心机制	实验重点
Lab3	中断与异常处理	建立完整的中断与异常处理机制，实现上下文保存与恢复	设计 TrapFrame 结构保存寄存器现场；实现 trap 入口汇编代码；处理中断、异常及时钟中断；正确返回原执行流	掌握 RISC-V 中断异常机制，理解上下文切换的底层实现，为抢占式调度提供基础
Lab4	进程（内核线程）管理	在单核系统上实现多执行流并发运行能力	引入线程控制块；实现上下文切换；设计调度器框架；建立线程虚拟地址空间结构	理解进程/线程抽象与调度机制，实现从单一执行流到多任务系统的转变
Lab5	用户程序与系统调用	支持用户态进程运行，实现用户态与内核态的安全切换	构造第一个用户进程；实现系统调用机制（fork / exec / exit / wait）；完成用户地址空间加载	掌握特权级隔离与系统调用流程，理解用户程序执行生命周期
Lab6	调度算法扩展	设计通用调度器框架并支持多种调度算法	将调度框架与具体算法解耦；实现 Round-Robin 调度算法；实现 Stride 调度算法	理解操作系统调度策略设计思想，提高调度公平性与可扩展性
Lab7	同步与互斥机制	提供并发执行环境下的同步与互斥支持	实现锁、信号量等同步原语；解决临界区访问问题；完成哲学家就餐问题	掌握并发控制与死锁避免方法，提升并发程序正确性
Lab8	文件系统	实现基于磁盘的文件系统与程序加载机制	实现文件读写接口；完善 SFS 文件系统；通过文件系统加载可执行程序；调整进程管理以适配文件执行	理解文件系统架构及其与进程管理的关系，构建完整操作系统运行环境

uCore（RISC-V）操作系统实验总体流程图（八个实验串联）

本流程图以“从可启动内核到具备进程、内存与文件系统的完整系统原型”为主线，将八个实验的关键目标与产出进行阶段化呈现。 各实验在功能与抽象层次上逐步递进，最终形成可运行用户程序、支持调度与同步并具备文件读写能力的 uCore 生态。

Lab1 · 启动与最小内核

最小可执行内核与启动流程

完成 QEMU + OpenSBI 环境下的内核引导，建立启动代码、栈与链接布局，确保控制权进入内核入口并输出基础运行信息。

QEMU / OpenSBI　启动汇编与链接脚本　内核入口与内存布局

阶段产出

可启动的内核骨架

形成可复用的启动路径，为后续内存、中断与进程机制提供执行载体。

Lab2 · 物理内存与页表

物理内存管理与基本映射

探测物理内存并建立连续页管理；实现 RISC-V Sv39 三级页表，完成内核虚拟地址到物理地址的初始映射。

物理页分配/回收　Sv39 页表　基础映射建立

阶段产出

可管理的内存资源

提供页分配与映射能力，为地址空间、进程与文件缓存等模块奠定基础。

Lab3 · 中断与异常

陷入处理与上下文保存恢复

构建 Trap 处理入口与 TrapFrame 结构，实现断点与时钟中断处理，验证陷入返回路径与上下文恢复正确性。

TrapFrame　stvec / sepc / scause　时钟中断

阶段产出

可抢占的执行基础

为调度触发、系统调用与用户态切换提供统一的陷入与返回机制。

Lab4 · 进程/线程管理

内核线程与基本调度框架

引入线程控制块与上下文切换机制；完善虚拟内存描述结构，建立线程级地址空间组织；实现多执行流轮转运行。

TCB / proc 结构　switch_to　地址空间描述

阶段产出

多执行流并发能力

使系统从单一控制流演进为可调度的并发内核执行模型。


Lab5 · 用户程序
用户态进程与系统调用
构造首个用户进程，实现 fork/exec/exit/wait 等系统调用与用户态运行路径，完成用户态与内核态的受控切换。
用户态/内核态隔离
系统调用路径
进程生命周期
阶段产出
可运行的用户程序模型
系统具备执行用户代码的能力，并通过系统调用提供受控服务接口。
Lab6 · 调度算法
调度框架与算法可扩展性
将调度框架与具体策略解耦，完成 RR 与 Stride 调度算法实现与切换，提升公平性与可配置性。
sched 框架化
RR
Stride
阶段产出
策略可替换的调度子系统
支持多策略调度实验与性能比较，为同步互斥与资源管理提供稳定运行环境。
Lab7 · 同步互斥
并发控制与死锁规避
实现锁与信号量等同步原语，解决并发执行下共享资源访问一致性问题，并以哲学家就餐问题验证正确性。
锁 / 信号量
临界区保护
死锁规避
阶段产出
可验证的并发正确性机制
为文件系统并发访问与进程间协作提供必要的同步基础。
Lab8 · 文件系统
SFS 文件读写与基于文件的程序装载
完善 SFS 读写路径（如 sfs_io_nolock），并重构基于文件系统的装载机制（load_icode），实现磁盘程序加载与文件访问。
inode / block 抽象
文件读写接口
从磁盘加载可执行文件
最终产出
小型完整 OS 原型
形成具备启动、内存、陷入、进程、调度、同步与文件系统的完整生态，可运行磁盘上的用户程序。
说明： 上述流程强调"能力递进关系"。Lab1–Lab3 建立可执行与可中断的内核运行基础；Lab4–Lab6 完成并发执行模型与调度策略扩展； Lab7 保证并发正确性；Lab8 引入持久化存储与基于文件的程序执行路径，使系统原型达到可用的最小闭环。

感谢李浩然老师和助教！！！