lab8+lab6实验报告

操作系统实验报告：Lab6与Lab8

第一部分：Lab8 - 文件系统

实验目标

- 深入理解“一切皆为文件”的设计思想。
- 掌握基于索引节点（inode）的 Simple FS 文件系统的组织方式与操作实现。
- 理解文件系统抽象层 VFS (Virtual File System) 的作用与设计。

1.1 练习1: 完成读文件操作的实现

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。

首先阐述一下打开文件的操作。（本次实验感觉内容和前几次的比较割裂，并且新增加的部分学习起来也有点模糊，如果理解有误积极接收纠正）

首先从宏观上把握层次结构：

- **通用文件系统访问接口层**：该层提供了**一个从用户空间到文件系统的标准访问接口**。这一层访问接口让应用程序能够**通过一个简单的接口获得** `ucore` **内核的文件系统服务**。
- **文件系统抽象层**：向上提供一个一致的接口给内核其他部分（文件系统相关的系统调用实现模块和其他内核功能模块）访问，向下提供一个同样的抽象函数指针列表和数据结构屏蔽不同文件系统的实现细节。这样即使底层的具体文件系统不同，只需要单独实例化该具体的文件系统，而无需修改上层模块。
- **具体文件系统：** **此处以**`Simple FS`**文件系统层为例，**一个基于索引方式的简单文件系统实例。**向上通过各种具体函数实现以对应文件系统抽象层提出的抽象函数，向下访问外设接口。**
- **外设接口层**：**向上提供** `device` **访问接口屏蔽不同硬件细节。向下实现访问各种具体设备驱动的接口**，比如 `disk` 设备接口 / 串口设备接口 / 键盘设备接口等。

可以发现整个层次结构是一个对下层抽象的结构，上层不需要知道下层的具体的结构，下层对上层是抽象的，每个具体的下层会有单独实现的实例来适配上层的端口。

其中必须要了解具体的文件系统的结构硬盘文件系统SFS：

- **第 0 个块（4K）是超级块（** `superblock` **）**，它包含了关于文件系统的所有关键参数，当计算机被启动或文件系统被首次接触时，超级块的内容就会被装入内存。

- **第 1 个块放了一个 `root-dir` 的 `inode`**，用来记录根目录的相关信息。有关 `inode` 还将在后续部分介绍。**这里只要理解 `root-dir` 是 `SFS` 文件系统的根结点，通过这个 `root-dir` 的 `inode` 信息就可以定位并查找到根目录下的所有文件信息。**
- **从第 2 个块开始**，根据 `SFS` 中所有块的数量，用 1 个 `bit` 来表示一个块的占用和未被占用的情况。**freemap 是 SFS 文件系统用来记录“磁盘上每个块是否被占用”的位图（bitmap）。**

  它用 **1 个 bit** 来表示磁盘上的 **1 个块（block）**：

  - bit = 0 → 该块空闲
  - bit = 1 → 该块已被占用
- **最后在剩余的磁盘空间中，存放了所有其他目录和文件的 `inode` 信息和内容数据信息**

区分内存inode和磁盘inode也是对整个文件操作过程是很重要的：

磁盘 inode 是永久存储的“文件元数据记录”；内存 inode 是磁盘 inode 在运行时加载到内存后的对象，它不仅包含磁盘 inode 信息，还额外加入了便于内核管理的运行时字段（例如引用计数、是否修改、锁等）。

open的操作流程：

总体来说，**给该进程分配一个文件描述符 fd，根据路径找到对应文件的 inode，并打开它。**

根据系统层次我大概概括为以下的步骤：

1. 通过系统调用，执行用户空间的open操作，触发更底层的服务。（通用文件访问接口层）
2. 首先在进程的 fd_array 中分配一个 file 结构，随后通过 VFS 的 vfs_open/vfs_lookup 调用具体文件系统（SFS）的 lookup/create/open 操作。到了这一步还仅仅是给当前用户进程分配了一个 `file` 数据结构的变量，还没有找到对应的文件索引节点。（文件系统抽象层的收集分发工作）
3. SFS 通过目录项查找得到 inode 号，再从磁盘加载 inode 构造内存 inode。抽象文件系统会调用 vfs_open()，vfs_open() 会触发vfs_lookup()，`vfs_lookup` 函数是一个针对目录的操作函数，它会调用 `vop_lookup` 函数来找到SFS文件系统中的目录下的文件。`vop_lookup` 首先调用 `get_device` 函数找到根目录下的inode，这个就需要使用在硬盘文件系统SFS提到的布局，根目录在block1。通过调用 `vop_lookup` 函数来查找到根目录 `“/”` 下对应文件 `sfs_filetest1` 的索引节点。

   值得注意的是，寻找路径或者inode和open是解耦的。`vfs_lookup` 只负责根据路径找到对应的 inode（定位）；`vfs_open` 在 `lookup` 基础上还要“按 flags 打开文件（必要时创建/截断/增加 open_count），并返回 inode。
4. 最终将 inode 绑定到 file 结构并返回 fd。这里返回的inode是内存inode，是对磁盘inode的记录，比硬盘inode有更丰富的信息。（最终将内存 inode 指针保存到 file 结构（file->node）并返回 fd。该 inode 是内存中的 VFS inode 对象，它在加载时从磁盘 inode 读取并缓存元数据，同时还维护引用计数、锁、操作函数表等运行时信息，因此比磁盘 inode 更丰富。）

代码块

```
open()
  → syscall
  → sysfile_open()
  → file_open()                    // 分配 fd，调用 VFS
  → vfs_open()                     // lookup / create / vop_open
  → vfs_lookup()
  → get_device()                   // 选择起始目录 inode
  → vop_lookup()
  → sfs_lookup()
  → sfs_lookup_once()
  → sfs_dirent_search_nolock()     // name → ino
  → sfs_load_inode()               // 读取磁盘 inode，构造内存 inode

```

read的流程

根据对文件的读操作read的学习可以知道，读数据的时候有如下流程：

1. 用户调用 read(fd, buf, len) 后进入内核（通用文件访问接口层）
2. 内核通过 fd 在当前进程的 fd_array 中找到对应的 struct file，其中保存了当前文件偏移 pos 和对应的 inode 指针 node。随后 file_read 构造 iobuf 并调用 **vop_read**，VFS 根据 inode 的 in_ops 将读操作分发给具体文件系统（如 SFS）。
3. SFS 将 pos/offset 转换为磁盘块号映射关系，通过 `sfs_bmap_load_nolock` 查到数据块号（disk block number），然后调用 disk0 driver 去读 block。

   抽象文件系统层实际读文件是**循环读取文件**，每次读取 `buffer` 大小，那这里的具体文件系统也需要和抽象文件系统读取文件的方式适配。

   sysfiled_read循环调用file_read,file_read实际调用vop_read, `vop_read` 函数实际上是对 `sfs_read` 的包装, `sfs_read` 函数调用 `sfs_io` 函数。它有三个参数， `node` 是对应文件的 `inode` ， `iob` 是缓存， `write` 表示是读还是写的布尔值,sfs_io然后调用 `sfs_io_nolock` 函数进行读取文件操作。
4. 最后由 sysfile_read 将该缓冲区内容复制到用户空间 buffer。

file_read 把数据读到 kernel buffer（你贴的 `buffer = kmalloc(IOBUF_SIZE)` ），`copy_to_user(mm, base, buffer, alen)` 把内容复制到用户态 `data` 。

仔细分析文件读取过程，read-sysfile_read-file_read-vop_read。

1. 用户请求 len， `read(fd, data, len)` ，从文件当前位置开始，读 len 字节给我。
2. sysfile_read使用 `buffer = kmalloc(IOBUF_SIZE); // IOBUF_SIZE = 4096` ，如果我们申请10000B，内核不会直接一次性读 10000 字节到用户 buf，而是先读 4096 到 kernel buffer，再读 4096，再读剩下的 1808。所以这一层很重要的操作是**按 4096B 分块循环调用 file_read**

   原因可能如下：

固定大小 buffer 易管理

不用在内核里一次性分配巨大内存

便于处理异常/部分读取/阻塞等情况

3. `file_read` 的 offset可能是任何长度，不一定是从文件的起始位置开始，比如offset是1000，那就不是块对齐的，比如从blcoki的第1000B开始，然后读取4096B，那还面临跨块和尾部块不满的情况。

代码块

```
→ sys_read
→ sysfile_read
    → (循环) file_read
        → fd2file(fd) 得到 struct file
        → iobuf_init(buf, len, file->pos)
        → vop_read(file->node, iob)
            → sfs_read
                → sfs_io (加锁)
                    → sfs_io_nolock
                        → sfs_bmap_load_nolock (块映射)
                        → sfs_rbuf / sfs_rblock
                            → disk0_io
                                → ide_read_secs
        → file->pos += copied
    → copy_to_user(buf)

```

综上，`sfs_bmap_load_nolock` 需要完成偏移到数据块号的映射，处理块首不对齐块、块中完整块和块末不满块。

手册提示：每部分中都调用 `sfs_bmap_load_nolock` 函数得到 `blkno` 对应的 `inode` 编号，并调用 `sfs_rbuf` 或 `sfs_rblock` 函数读取数据（中间部分调用 `sfs_rblock`，起始和末尾部分调用 `sfs_rbuf`），调整相关变量。

代码块

```
    // (1) 处理首个不对齐块
    blkoff = offset % SFS_BLKSIZE;//SFS_BLKSIZE 4K
    // 不能触碰block0
    if (blkoff != 0) {
    // size: 本次要读/写的字节数
    // - 若后面还有完整块（nblks != 0），则把当前块剩余部分读/写完：SFS_BLKSIZE - blkoff
    // - 若没有完整块（nblks == 0），说明读/写区间完全落在同一个块中：endpos - offset，防止头不对齐，尾不满
        size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
```

```
        // 过块映射找到文件逻辑块 blkno 对应的磁盘数据块号 ino
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        // 对当前块进行部分读/写
        if ((ret = sfs_buf_op(sfs, buf, size, ino, blkoff)) != 0) {
            goto out;
        }
        // alen 累计实际完成的读/写长度
        alen += size;
        // 如果 nblks==0，说明请求区间只在这一块内，首块处理完就结束
        if (nblks == 0) {
            goto out;
        }
        // 更新 buf 指针、offset、blkno，准备进入下一块处理
        // buf 移动 size，offset 也移动 size
        buf += size;
        offset += size;
        blkno ++;
        nblks --;
    }

    // (2) 处理中间完整块
    while (nblks != 0) {
        // 找到当前逻辑块 blkno 对应的磁盘块号 ino
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
        }
        // 整块读/写：一次直接处理 1 个完整块（SFS_BLKSIZE 字节）
        // sfs_block_op 读取完整块
        if ((ret = sfs_block_op(sfs, buf, ino, 1)) != 0) {
            goto out;
        }
        alen += SFS_BLKSIZE;
        buf += SFS_BLKSIZE;
        offset += SFS_BLKSIZE;
        blkno ++;
        nblks --;
    }

    // (3) 处理末尾不对齐块
    // 找到末尾逻辑块 blkno 对应的磁盘块号 ino
    blkoff = endpos % SFS_BLKSIZE;
    if (blkoff != 0) {
        // 找到末尾逻辑块 blkno 对应的磁盘块号 ino
        if ((ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino)) != 0) {
            goto out;
```

```
56          }
57          // 对最后一块做部分读/写:
58          // - 从磁盘块 ino 的偏移 0 开始
59          // - 读/写 blkoff 个字节到 buf
60          if ((ret = sfs_buf_op(sfs, buf, blkoff, ino, 0)) != 0) {
61              goto out;
62          }
63          alen += blkoff;
64      }
```

设备文件系统stdin & stdout

补充从设备读取而不是普通文件系统，这在inode中有存储信息，如果是调用stdin或者stdout，在虚拟文件层调用get_device的时候，会发现使用多个是设备文件系统，这样在分发的时候会分发对应的函数。

代码块

```
1  //当fd = 0的时候，表示从stdin读取int read(int fd, void *base, size_t len) {
2      return sys_read(fd, base, len);
3  }
```

但是lab8将设备变成文件系统挂载，那使用stdin和stout的方式就有所改变，通过系统调用从统一文件层发起对对应设备的使用请求，**在真正运行用户程序 `main()` 之前，为它建立“标准输入 stdin”和“标准输出 stdout”的运行时环境**，让用户程序可以像在 Linux 下一样使用 `read` 和 `write`。

- **fd=0 固定绑定到 stdin 设备文件**
- **fd=1 固定绑定到 stdout 设备文件**

然后执行该应用程序，执行完后退出当前进程。

stdout的流程较为简单，这里阐述更为复杂的stdin的流程：对于stdin而言，由于qemu无法接收到键盘输入中断信号，将读取中断输入分为两个阶段。

1. 将数据从终端读入控制台缓冲区
2. 将数据从控制台读入用户区，通过 `cons_getc()` 从 cons.buf 取出字符，如果有取出的字符，则唤醒等待输入的进程，如果没有字符返回则让 进程等待。

代码块

```
1  硬件串口输入
2    ↓
3  sbi_console_getchar()（之间没有文件系统直接系统调用这个底层服务）
4    ↓
5  serial_proc_data()
```

```
  ↓
cons_intr() 把字符放到 cons.buf[512]
  ↓
dev_stdin_write(cons_getc() ) 从 cons.buf 取出一个字符，把它写入 stdin_buffer[4096]
  ↓
用户 read(0) 实际调用dev_stdin_read()（是 stdin 设备文件的底层读实现），从 stdin_buffer 读，如果没有字符，就让进程睡眠等待键盘输入唤醒。

```

1.2 练习2: 实现基于文件系统的执行程序机制

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 `exit`，`hello`（更多用户程序放在 `user` 目录下）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。

前期准备

1. 前期准备：do_execve 先从当前 mm 里用 copy_string/copy_kargv 把程序名、参数拷到内核缓冲，检查 argc 合法，然后调用 sysfile_open(argv[0], O_RDONLY) 打开目标文件。路径查找会经过 VFS，SFS 后端的确最终调用 sfs_lookup 找到对应的 inode，并返回一个 fd。
2. 替换地址空间：拿到 fd 后，如果当前进程已有 mm，就切换回内核页表，exit_mmap/put_pgdir/mm_destroy 清掉旧用户空间（不清理“内核空间 mm”，而是清理旧的用户地址空间）。
3. 加载新程序：调用 load_icode(fd, argc, kargv) 构建新的 mm、页表、装入 ELF、设置栈和 trapframe。
4. 设置进程名、关闭文件、返回用户态。

`load_icode` 函数的主要工作就是给用户进程建立一个能够让用户进程正常运行的用户环境。此函数完成了如下重要工作：

1. 调用 `mm_create` 函数来申请进程的内存管理数据结构 `mm` 所需内存空间，并对 `mm` 进行初始化；调用 `setup_pgdir` 来申请一个页目录表所需的一个页大小的内存空间，并把描述ucore内核虚空间映射的内核页表（`boot_pgdir` 所指）的内容拷贝到此新目录表中，最后让 `mm->pgdir` 指向此页目录表，这就是进程新的页目录表了，且能够正确映射内核虚空间；
2. 根据应用程序文件的文件描述符（`fd`）通过调用 `load_icode_read ()` 函数来加载和解析此ELF格式的执行程序，并调用 `mm_map` 函数根据ELF格式的执行程序说明的各个段（代码段、数据段、BSS段等）的起始位置和大小建立对应的 `vma` 结构，并把 `vma` 插入到 `mm` 结构中，从而表明了用户进程的合法用户态虚拟地址空间；

3. 调用根据执行程序各个段的大小分配物理内存空间，并根据执行程序各个段的起始位置确定虚拟地址，并在页表中建立好物理地址和虚拟地址的映射关系，然后把执行程序各个段的内容拷贝到相应的内核虚拟地址中，至此应用程序执行码和数据已经根据编译时设定地址放置到虚拟内存中了；
4. 需要给用户进程设置用户栈，为此调用 `mm_mmap` 函数建立用户栈的 `vma` 结构，明确用户栈的位置在用户虚空间的顶端，大小为 `256` 个页，即 `1MB` ，并分配一定数量的物理内存且建立好栈的虚地址<-->物理地址映射关系；
5. 至此,进程内的内存管理 `vma` 和 `mm` 数据结构已经建立完成，于是把 `mm->pgdir` 赋值到 `cr3` 寄存器中，即更新了用户进程的虚拟内存空间，设置 `uargc` 和 `uargv` 在用户栈中。此时的 `initproc` 已经被代码和数据覆盖，成为了第一个用户进程，但此时这个用户进程的执行现场还没建立好；
6. 先清空进程的中断帧，再重新设置进程的中断帧，使得在执行中断返回指令 `“iret”` 后，能够让CPU转到用户态特权级，并回到用户态内存空间，使用用户态的代码段、数据段和堆栈，且能够跳转到用户进程的第一条指令执行，并确保在用户态能够响应中断；

主要步骤

- **创建 mm / 页目录**：mm_create() 申请并初始化 struct mm；setup_pgdir(mm) 分配一页新的页目录，把内核高地址映射从 boot_pgdir 复制过去，mm->pgdir 指向它。
- **读取 ELF 头、程序头**：通过文件描述符 fd 调 load_icode_read（内部是 sysfile_seek/sysfile_read）先读 ELF 头校验魔数和程序头大小，再逐个程序头遍历。
- **建立 vma / 映射段 / 拷贝段内容**：
  - 过滤 p_type==ELF_PT_LOAD 的段，检查 p_filesz<=p_memsz，生成 vm_flags（R/W/X）和页表权限 perm。
  - mm_map(mm, ph.p_va, ph.p_memsz, vm_flags, NULL) 建立该段的 vma。
  - 循环为文件部分分配页 pgdir_alloc_page(mm->pgdir, la, perm)，调用 load_icode_read 把文件内容拷到新页对应的内核虚拟地址 page2kva(page)+off。
  - 对剩余的 BSS 部分分配页并 memset 0。
- **用户栈 vma 与预分配**：mm_map 在 [USTACKTOP-USTACKSIZE, USTACKTOP) 建立栈 vma，预先分配了顶端 4 页用户栈页。
- **把 argv/argc 写入用户栈**：从栈顶往下拷贝各个字符串，再拷贝 argv 指针数组，保持 16 字节对齐，记录最终 sp 和 uargv_ptr。
- **安装新 mm / 切换 satp**：mm_count_inc，current->mm=mm，current->pgdir=PADDR(mm->pgdir)，lsatp 切换页表。
- **设置 trapframe**：清零 tf，设置用户 sp、a0=argc、a1=uargv_ptr、epc=elf.e_entry，status 清掉 SPP、置 SPIE，确保从内核返回后跳到用户态第一条指令。

- **出错清理**：统一跳到 bad_* 标签，exit_mmap/put_pgdir/mm_destroy，最后都会 sysfile_close(fd)。

代码块

```
static int
load_icode(int fd, int argc, char **kargv)
{
    /*
     * 目标：把 fd 对应的用户程序（ELF 可执行文件）加载成“当前进程”的用户态地址空间，
     *       并在用户栈上构造 argc/argv，最后设置 trapframe，让 CPU 从 ELF 入口开始跑。
     *
     * 这一步本质上做了 execve 的核心：
     *   - 创建新的 mm（新的虚拟地址空间）
     *   - 建立新的页表（pgdir）
     *   - 解析 ELF -> 把 PT_LOAD 段映射进新地址空间并把文件内容拷进去
     *   - 建用户栈并把参数压入栈
     *   - 安装 mm 与页表（切 satp）
     *   - 设置 trapframe（sp/epc/a0/a1/status）
     *
     * 注意：这里是 load_icode（无锁/简化版），通常只允许 current->mm 为空，
     * 即“当前进程还没有用户地址空间”（比如刚 fork 出来的内核线程，准备 exec）。
     */

    // 防御性检查：如果当前进程已经有 mm，说明你试图在已有用户空间上再装一遍 ELF，
    // 这通常是实验框架不允许的（需要先释放旧 mm，再装新的）。
    if (current->mm != NULL)
    {
        panic("load_icode: current->mm must be empty.\n");
    }

    // 默认返回值：先假设内存不足，后面成功再置 0
    int ret = -E_NO_MEM;
    struct mm_struct *mm;

    /* (1) create mm */
    // mm_create：创建一个新的 mm_struct，内部一般会初始化 vma 链表、锁、计数等
    // 这代表“新的进程地址空间抽象”，但此时还没有页表（pgdir）
    if ((mm = mm_create()) == NULL)
    {
        goto out; // 失败就走统一退出路径
    }

    /* (2) setup pgdir */
    // setup_pgdir：为该 mm 建立新的页目录（pgdir），并复制/映射内核的那部分页表
    // 因为用户进程仍需要能进入内核（系统调用/中断陷入），所以内核高地址映射必须存在
```

```
42      // 成功后 mm->pgdir 是“内核虚拟地址”形式的页表根
43      if (setup_pgdir(mm) != 0)
44      {
45          goto bad_pgdir_cleanup_mm;
46      }
47
48      /* (3) load ELF header */
49      // 先读 ELF Header（固定大小），偏移是 0
50      // load_icode_read(fd, buf, len, offset)：从文件 fd 的 offset 处读 len 字节到
    buf
51      struct elfhdr elf;
52      if ((ret = load_icode_read(fd, &elf, sizeof(struct elfhdr), 0)) != 0)
53      {
54          goto bad_elf_cleanup_pgdir;
55      }
56
57      // 检查 ELF 魔数，防止把普通文件当可执行文件加载
58      if (elf.e_magic != ELF_MAGIC)
59      {
60          ret = -E_INVAL_ELF;
61          goto bad_elf_cleanup_pgdir;
62      }
63
64      // 检查 program header 的每项大小是否符合预期，防止结构不匹配导致错误解析
65      if (elf.e_phentsize != sizeof(struct proghdr))
66      {
67          ret = -E_INVAL_ELF;
68          goto bad_elf_cleanup_pgdir;
69      }
70
71      /* (3) load segments */
72      // vm_flags：VMA 的语义权限（VM_READ/VM_WRITE/VM_EXEC/VM_STACK）
73      // perm：页表项 PTE 的硬件权限位（U/R/W/X/V/A/D 等）
74      uint32_t vm_flags, perm;
75      struct Page *page = NULL;
76      int i;
77
78      // 遍历 ELF 的每一个 Program Header（ph）
79      // e_phoff：program header table 在文件中的偏移
80      // e_phnum：program header 的数量
81      for (i = 0; i < elf.e_phnum; i++)
82      {
83          struct proghdr ph;
84
85          // 读取第 i 个 ph
86          // 注意偏移计算：elf.e_phoff + i * sizeof(struct proghdr)
87          if ((ret = load_icode_read(fd, &ph, sizeof(struct proghdr),
```

```
                                    (off_t)(elf.e_phoff + (uint64_t)i *
sizeof(struct proghdr)))) != 0)
        {
            goto bad_cleanup_mmap;
        }

        // 只处理 PT_LOAD 段（需要装入内存的段：代码段/数据段/bss 所在段）
        if (ph.p_type != ELF_PT_LOAD)
        {
            continue;
        }

        // ELF 语义要求：文件里实际数据大小 filesz 不能超过内存应占大小 memsz
        // 否则说明 ELF 损坏或格式不合法
        if (ph.p_filesz > ph.p_memsz)
        {
            ret = -E_INVAL_ELF;
            goto bad_cleanup_mmap;
        }

        // 先从 ph.p_flags 转换出“VMA 权限”和“PTE 权限”
        vm_flags = 0;

        /*
         * perm 初始化：
         *   PTE_U：用户态可访问（否则用户访问会 fault）
         *   PTE_V：页表项有效
         *   PTE_A：Accessed 位。很多 RISC-V 实验内核为了避免第一次访问触发 fault，
         *          会在建映射时直接置 A（否则硬件可能因 A=0 抛异常）
         *   PTE_D：Dirty 位。通常对可写段置 D，避免第一次写因 D=0 fault
         */
        perm = PTE_U | PTE_V | PTE_A;
        if (ph.p_flags & ELF_PF_W) perm |= PTE_D;

        // ph.p_flags 的 R/W/X 映射到 vm_flags（供 mm_map 建立 VMA）
        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;

        // vm_flags 再映射到硬件 PTE 位：
        //   - 读权限 -> PTE_R
        //   - 写权限 -> PTE_W（通常也要 PTE_R，因为很多实现要求可写页也可读）
        //   - 执行权限 -> PTE_X
        if (vm_flags & VM_READ)
```

```
134             perm |= PTE_R;
135         if (vm_flags & VM_WRITE)
136             perm |= (PTE_W | PTE_R);
137         if (vm_flags & VM_EXEC)
138             perm |= PTE_X;
139 
140         /*
141          * mm_map: 在 mm 的 VMA 列表中建立一段 vma
142          * 参数:
143          *   start = ph.p_va
144          *   len   = ph.p_memsz（注意是 memsz，因为 bss 也要占地址空间）
145          *   flags = VM_*
146          *
147          * 这一步只是在“逻辑层”声明：这段虚拟地址属于进程、具有某些权限
148          * 还没真正分配物理页。
149          */
150         if ((ret = mm_map(mm, (uintptr_t)ph.p_va, (size_t)ph.p_memsz,
    vm_flags, NULL)) != 0)
151         {
152             goto bad_cleanup_mmap;
153         }
154 
155         /*
156          * 接下来真正把段装入内存:
157          * start: 当前要拷贝/清零的虚拟地址游标（从 p_va 开始）
158          * la: 按页对齐的页起始地址（RoundDown 到 PGSIZE）
159          * file_offset: 文件中该段数据的起始偏移（p_offset）
160          */
161         uintptr_t start = (uintptr_t)ph.p_va, end, la = ROUNDDOWN(start,
    PGSIZE);
162         off_t file_offset = (off_t)ph.p_offset;
163 
164         /* copy TEXT/DATA: 先拷贝 filesz 对应的“文件中存在的部分” */
165         end = (uintptr_t)(ph.p_va + ph.p_filesz);
166 
167         while (start < end)
168         {
169             // pgdir_alloc_page: 为 la 这一页分配一个物理页，并建立 la->page 的映射
170             // perm 是该页的 PTE 权限位（U/R/W/X...）
171             if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL)
172             {
173                 ret = -E_NO_MEM;
174                 goto bad_cleanup_mmap;
175             }
176 
177             // 计算在当前页里，从 start 到页末能写多少
178             size_t off = start - la;        // 当前页内偏移
```

```
179             size_t size = PGSIZE - off;     // 默认写到页末
180
181             la += PGSIZE;                   // 下一页起点（注意 la 已向前推进）
182             if (end < la)
183             {
184                 // 如果这次拷贝到不了整页（段尾在本页中间），缩短 size
185                 size -= la - end;
186             }
187
188             // 把文件里的 size 字节读到该页对应的内核虚拟地址处（page2kva）
189             // 注意：page2kva(page) 是这个物理页在内核中的映射，用于内核写入内容
190             if ((ret = load_icode_read(fd, page2kva(page) + off, size,
    file_offset)) != 0)
191             {
192                 goto bad_cleanup_mmap;
193             }
194
195             // 推进 start 和 file_offset
196             start += size;
197             file_offset += size;
198         }
199
200         /* build BSS: 把 memsz 中“文件没有的部分”清零（bss + 尾部 padding） */
201         end = (uintptr_t)(ph.p_va + ph.p_memsz);
202
203         // 情况 A：filesz 结束点和页边界之间还有一段空洞，需要在“最后一页”补零
204         // 此时 page 仍指向上一段循环中分配的最后一页
205         if (start < la)
206         {
207             if (start == end)
208             {
209                 // memsz == filesz：没有 bss，直接进入下一个段
210                 continue;
211             }
212
213             // off 的计算：start 落在上一页中间，la 已经是下一页的起点
214             // start + PGSIZE - la 就是 start 在上一页的页内偏移
215             size_t off = start + PGSIZE - la;
216             size_t size = PGSIZE - off;
217
218             // 如果 bss 结束也在这一页中间，缩短 size
219             if (end < la)
220             {
221                 size -= la - end;
222             }
223
224             // 把这一页的 bss 部分清零
```

```
225                 memset(page2kva(page) + off, 0, size);
226
227                 start += size;
228
229                 // 断言：清零后要么刚好到 end，要么刚好推进到 la（下一页边界）
230                 assert((end < la && start == end) || (end >= la && start == la));
231             }
232
233             // 情况 B：bss 需要跨越更多完整页，则分配新页并整页/部分页清零
234             while (start < end)
235             {
236                 if ((page = pgdir_alloc_page(mm->pgdir, la, perm)) == NULL)
237                 {
238                     ret = -E_NO_MEM;
239                     goto bad_cleanup_mmap;
240                 }
241
242                 size_t off = start - la;
243                 size_t size = PGSIZE - off;
244
245                 la += PGSIZE;
246                 if (end < la)
247                 {
248                     size -= la - end;
249                 }
250
251                 memset(page2kva(page) + off, 0, size);
252                 start += size;
253             }
254         }
255
256         /* (4) user stack */
257         /*
258          * 建立用户栈对应的 VMA：  [USTACKTOP-USTACKSIZE, USTACKTOP)
259          * vm_flags 加 VM_STACK 只是语义标记（便于调试/权限约束）
260          */
261         vm_flags = VM_READ | VM_WRITE | VM_STACK;
262         if ((ret = mm_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm_flags, NULL))
    != 0)
263         {
264             goto bad_cleanup_mmap;
265         }
266
267         /*
268          * 真实分配用户栈页：
269          * 这里预分配 4 页，确保 argv/环境构造时不会因缺页出错（简化实现）
270          * PTE_USER 一般是用户可读写的组合权限宏（含 U/V/R/W/A/D）
```

```
271         */
272        if (pgdir_alloc_page(mm->pgdir, USTACKTOP - PGSIZE, PTE_USER) == NULL ||
273            pgdir_alloc_page(mm->pgdir, USTACKTOP - 2 * PGSIZE, PTE_USER) == NULL
    ||
274            pgdir_alloc_page(mm->pgdir, USTACKTOP - 3 * PGSIZE, PTE_USER) == NULL
    ||
275            pgdir_alloc_page(mm->pgdir, USTACKTOP - 4 * PGSIZE, PTE_USER) == NULL)
276        {
277            ret = -E_NO_MEM;
278            goto bad_cleanup_mmap;
279        }
280
281        /* (6) setup argc/argv on user stack */
282        /*
283         * 栈从高地址向低地址增长:
284         *   - 先把每个参数字符串拷贝到栈上（从后往前压）
285         *   - 再在更低地址处放 argv 指针数组（uargv_ptr）
286         *   - 让 a0=argc, a1=uargv_ptr
287         */
288        uintptr_t stacktop = USTACKTOP;
289        uintptr_t uargv[EXEC_MAX_ARG_NUM];
290        uintptr_t sp = stacktop;
291
292        // 从最后一个参数开始压栈，保证 argv[0] 在较低地址的字符串区前面
293        for (i = argc - 1; i >= 0; i--)
294        {
295            // len 包含 '\0'，保证用户态当 C 字符串能正常读取
296            size_t len = strlen(kargv[i]) + 1;
297
298            // sp 下移 len 字节，为字符串留空间
299            sp -= len;
300
301            // 按机器字对齐（指针数组/ABI 更稳）
302            sp = ROUNDDOWN(sp, sizeof(uintptr_t));
303
304            // 栈溢出检查：不得越过栈底
305            if (sp < stacktop - USTACKSIZE)
306            {
307                ret = -E_TOO_BIG;
308                goto bad_cleanup_mmap;
309            }
310
311            /*
312             * 这里不用 copy_to_user 的原因:
313             * 你还没安装 satp 到新页表里（当前 CPU 还在旧地址空间），
314             * 所以直接用用户虚拟地址写会失败。
315             * 因此做法是：用 get_page 找到该用户虚拟地址对应的 Page，
```

```
316             * 然后通过 page2kva(p) 得到它在内核中的映射地址，把内容 memcpy 进去。
317             */
318            uintptr_t dst = sp;
319            const char *src = kargv[i];
320            size_t left = len;
321
322            while (left != 0)
323            {
324                // 找 dst 所在页的页起始 la
325                uintptr_t la = ROUNDDOWN(dst, PGSIZE);
326
327                // 查页表是否已有该页
328                struct Page *p = get_page(mm->pgdir, la, NULL);
329                if (p == NULL)
330                {
331                    // 没有则分配（理论上预分配了4页，但参数多/长时可能超出）
332                    if ((p = pgdir_alloc_page(mm->pgdir, la, PTE_USER)) == NULL)
333                    {
334                        ret = -E_NO_MEM;
335                        goto bad_cleanup_mmap;
336                    }
337                }
338
339                // 计算本页能拷多少
340                size_t off = dst - la;
341                size_t part = PGSIZE - off;
342                if (part > left)
343                {
344                    part = left;
345                }
346
347                // 拷贝到该页的内核映射地址
348                memcpy(page2kva(p) + off, src, part);
349
350                // 推进
351                dst += part;
352                src += part;
353                left -= part;
354            }
355
356            // 记录该参数字符串在用户栈上的地址（未来写 argv[i] 指针用）
357            uargv[i] = sp;
358        }
359
360        // ABI 对齐：很多平台要求栈 16 字节对齐（尤其对调用约定/性能）
361        sp = ROUNDDOWN(sp, 16);
362
```

```
363         // 为 argv 指针数组预留空间（argc 个指针 + 末尾 NULL）
364         size_t argv_size = (argc + 1) * sizeof(uintptr_t);
365         sp -= argv_size;
366         sp = ROUNDDOWN(sp, 16);
367 
368         if (sp < stacktop - USTACKSIZE)
369         {
370             ret = -E_TOO_BIG;
371             goto bad_cleanup_mmap;
372         }
373 
374         // uargv_ptr 是用户态 argv 数组的起始地址（将放到 a1）
375         uintptr_t uargv_ptr = sp;
376 
377         // 构造一个临时的 argv 指针数组（内核栈上），再整体拷贝到用户栈
378         uintptr_t argv_ptrs[EXEC_MAX_ARG_NUM + 1];
379         for (i = 0; i < argc; i++)
380         {
381             argv_ptrs[i] = uargv[i];
382         }
383         argv_ptrs[argc] = 0;
384 
385         /* copy argv_ptrs to user stack（同样用 page2kva 的方法跨页拷贝） */
386         {
387             uintptr_t dst = uargv_ptr;
388             const char *src = (const char *)argv_ptrs;
389             size_t left = argv_size;
390             while (left != 0)
391             {
392                 uintptr_t la = ROUNDDOWN(dst, PGSIZE);
393                 struct Page *p = get_page(mm->pgdir, la, NULL);
394                 if (p == NULL)
395                 {
396                     if ((p = pgdir_alloc_page(mm->pgdir, la, PTE_USER)) == NULL)
397                     {
398                         ret = -E_NO_MEM;
399                         goto bad_cleanup_mmap;
400                     }
401                 }
402                 size_t off = dst - la;
403                 size_t part = PGSIZE - off;
404                 if (part > left)
405                 {
406                     part = left;
407                 }
408                 memcpy(page2kva(p) + off, src, part);
409                 dst += part;
```

```
410             src += part;
411             left -= part;
412         }
413     }
414 
415     /* (5) install mm */
416     /*
417      * 到这里：新 mm、页表、段、栈、argv 都准备好了。
418      * 接下来把它正式装到 current 上，并切换硬件页表根寄存器 satp。
419      */
420     mm_count_inc(mm);                 // 引用计数+1（防止被提前释放）
421     current->mm = mm;                 // 当前进程关联这个 mm
422     current->pgdir = PADDR(mm->pgdir);// 保存页表根的物理地址（硬件用物理地址）
423     lsatp(PADDR(mm->pgdir));          // 切换到新页表（RISC-V 的 satp 类似 x86 的
    CR3）
424 
425     /* (7) setup trapframe */
426     /*
427      * trapframe 是“从内核回到用户态”时 CPU 寄存器的快照。
428      * 我们要设置：
429      *   - sp：用户栈顶（当前 sp）
430      *   - a0：argc
431      *   - a1：argv 指针数组地址（uargv_ptr）
432      *   - epc：ELF 入口地址（e_entry）
433      *   - status：清 SPP（返回到 U-mode），开 SPIE 让用户态能中断
434      */
435     struct trapframe *tf = current->tf;
436     uintptr_t sstatus = tf->status;           // 保留原 status 里的某些位（例如中
    断开关）
437     memset(tf, 0, sizeof(struct trapframe));  // 清空，避免脏寄存器带入用户态
438 
439     tf->gpr.sp = sp;          // 用户栈指针
440     tf->gpr.a0 = argc;        // argc 放 a0（约定）
441     tf->gpr.a1 = uargv_ptr;   // argv 指针数组地址放 a1（约定）
442     tf->epc = elf.e_entry;    // 从 ELF 入口开始执行
443 
444     // status 设置：
445     //   - 清掉 SSTATUS_SPP：确保 sret 返回到 U-mode
446     //   - 设置 SSTATUS_SPIE：用户态打开中断（或允许中断在返回后恢复）
447     tf->status = (sstatus & ~SSTATUS_SPP) | SSTATUS_SPIE;
448 
449     ret = 0;
450 
451 out:
452     // 关闭文件描述符：不管成功失败都要 close（避免 fd 泄漏）
453     sysfile_close(fd);
454     return ret;
```

```
455
456 bad_cleanup_mmap:
457     // 段/栈映射失败时，把 mm 里已经建立的 vma 和页映射清理掉
458     exit_mmap(mm);
459 bad_elf_cleanup_pgdir:
460     // 释放页表（pgdir）相关资源
461     put_pgdir(mm);
462 bad_pgdir_cleanup_mm:
463     // 释放 mm 本身
464     mm_destroy(mm);
465     goto out;
466 }
```

对比 lab5 版的差异

- **可执行来源**：实验8和实验5中 `load_icode（）` 函数代码最大不同的地方在于读取EFL文件的方式，实验5中是通过获取ELF在内存中的位置，直接从内核内嵌的用户程序镜像（数组或符号）拷贝；lab8 改为通过文件描述符 fd调用 `load_icode_read（）` 从 SFS 文件系统读取，完整走 sysfile_*。

代码块

```
1  static intload_icode_read(int fd, void *buf, size_t len, off_t offset) {
2      int ret;
3      if ((ret = sysfile_seek(fd, offset, LSEEK_SET)) != 0) {
4          return ret;
5      }
6      if ((ret = sysfile_read(fd, buf, len)) != len) {
7          return (ret < 0) ? ret : -1;
8      }
9      return 0;
10 }
```

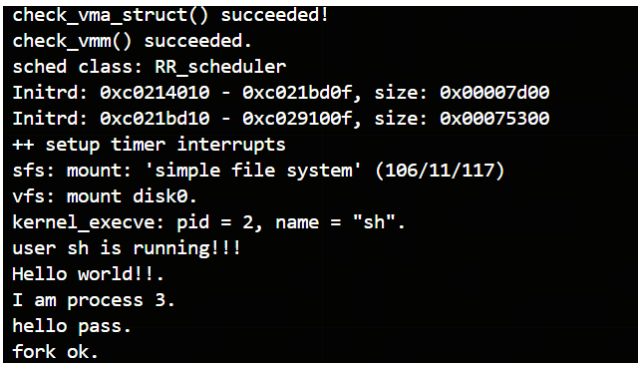
这个函数是用来从文件描述符 `fd` 指向的文件中读取数据到 `buf` 中的。函数的入参包括 `fd` （文件描述符）、 `buf` （指向读取数据的缓冲区）、 `len` （要读取的数据长度）和 `offset` （文件中的偏移量）。

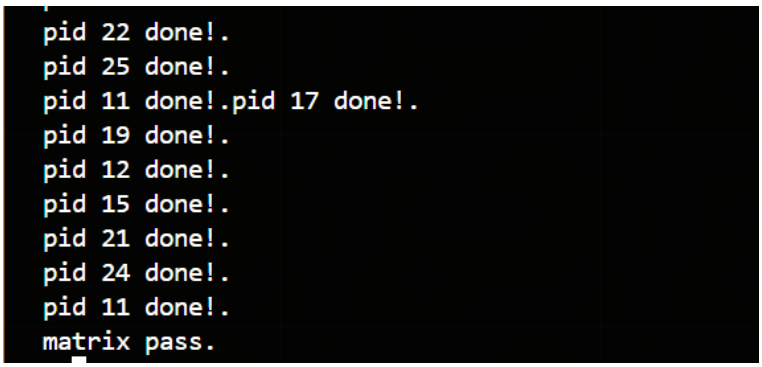
函数首先调用 `sysfile_seek` 函数将文件指针移动到指定的偏移量处。如果 `sysfile_seek` 返回错误码，则 `load_icode_read` 直接返回该错误码。

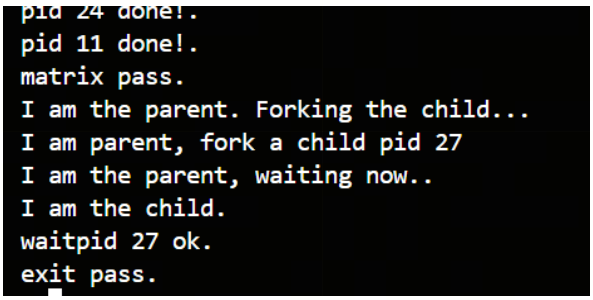
接着，函数调用 `sysfile_read` 函数从文件中读取指定长度的数据到 `buf` 中。如果读取的数据长度不等于要求的长度 `len` ，则 `load_icode_read` 返回错误码 `-1` ；如果读取过程中发生了错误，则 `load_icode_read` 返回 `sysfile_read` 函数的错误码。

最后，如果函数执行成功，则 `load_icode_read` 返回 `0` 。

- **读段/建映射：**
  - Lab5：直接从内存镜像 memcpy 到新页。
  - Lab8：逐个 load_icode_read ELF 头和程序头，用 mm_map 建 vma，pgdir_alloc_page 分配页后把文件内容读入 page2kva，BSS 零填。权限按 ELF R/W/X 组合出 PTE（U/V/A/D）。生成的 perm 传给 pgdir_alloc_page，建页表时就带上了对应段的硬件权限。
- **用户栈和 argv：**
  - Lab5：lab5 权限简单一些，栈顶 4 页用 assert 分配，且没有关闭文件的概念。
  - Lab8：lab8 PTE 权限包含 PTE_A/PTE_D，mm_map 建 1MB 栈，预分配顶端 4 页，错误路径还会 sysfile_close(fd)；从栈顶往下拷贝各字符串和 argv 指针，16B 对齐，设置 a0/a1。
- **函数签名与调用关系**
  - lab5 的 do_execve 直接接受 binary/size，不走文件系统。
  - lab8 的 do_execve 通过 sysfile_open 打开路径获取 fd，清理旧 mm 后调用 load_icode(fd, argc, kargv)。
- **其他细节**
  - 两者都创建新 mm/pgdir、mm_map 建 vma、分配页/拷贝 TEXT/DATA/清 BSS、设置 trapframe，最后切换 satp；只是 lab8 引入了文件系统读路径和参数栈布局。







1. kernel_execve 成功执行 sh：“从文件系统加载 ELF 并建立用户空间”成功

`kernel_execve: pid = 2, name = "sh".`
`user sh is running!!!`

- 说明 `kernel_execve("sh")` 找到了 `/sh`，并通过 `do_execve/load_icode` 走完整的 ELF 装载：解析 ELF header、映射段、拷贝段数据、建栈、设置 trapframe，切到用户态执行。

2. sh 内能继续运行 hello / 其它测试：不仅能 exec 一次，还能多次 exec + fork + 调度 + wait/exit

```
Hello world!!.
I am process 3.
hello pass.
...
matrix pass.
...
waitpid 27 ok.
exit pass.
```

它说明了几件事：

- **hello pass**：sh 能从文件系统里再 exec 出 `hello`，证明“sh -> exec -> load_icode -> 跑完返回 sh”这条路径通。
- 大量 `pid X is running ... done`：说明在反复 fork、创建多个进程、频繁切换的情况下，用户态程序能持续运行并最终正常退出。
- `waitpid 27 ok`：说明父进程能等待子进程回收，进程退出码/僵尸清理链路也通。
- `exit pass`：说明 exit 程序（或测试）能被执行并正确触发进程退出与回收。

当运行badsegment、divzero、spin等程序时报错，询问大模型后，得知是因为要通过这些测试，需要在 trap.c 里实现用户态异常处理：区分内核/用户陷入，用户态的非法指令/地址/页错误要么调用 pgfault_handler 分配/回收页，要么打印 trapframe 后 do_exit(-E_KILLED) 杀掉进程；只有内核态异常才 panic。这样异常进程会正确退出，父进程的 wait/kill 才能完成，以上测试自然能过。

接下来：

- 在 trap.c 中新增 do_pgfault 外部声明。
- 在 vmm.c 实现了基本的 do_pgfault：查找 vma、检查写权限、根据 vma 的 R/W/X 组合出 PTE 权限，按页对齐地址并用 pgdir_alloc_page 分配并映射，返回错误码。
- 在 vmm.h 对外声明 do_pgfault。
- 重写 exception_handler，区分内核态与用户态陷入：
  - 系统调用仍递增 epc 后调用 syscall。
  - 对页故障在用户态尝试调用 do_pgfault；失败则打印后杀掉进程。
  - 其他用户态异常打印后调用 do_exit(-E_KILLED)；内核态异常直接 panic。
  - 在页故障的写入场景传入 error_code=1，读取为0，使用 tf->tval 作为故障地址。

代码块

```
1  void exception_handler(struct trapframe *tf)
2  {
```

```
    bool in_kernel = trap_in_kernel(tf);
    int ret = 0;
    switch (tf->cause)
    {
    case CAUSE_MISALIGNED_FETCH:
        cprintf("Instruction address misaligned\n");
        ret = -E_INVAL;
        break;
    case CAUSE_FETCH_ACCESS:
        cprintf("Instruction access fault\n");
        ret = -E_INVAL;
        break;
    case CAUSE_ILLEGAL_INSTRUCTION:
        cprintf("Illegal instruction\n");
        ret = -E_INVAL;
        break;
    case CAUSE_BREAKPOINT:
        cprintf("Breakpoint\n");
        ret = -E_INVAL;
        break;
    case CAUSE_MISALIGNED_LOAD:
        cprintf("Load address misaligned\n");
        ret = -E_INVAL;
        break;
    case CAUSE_LOAD_ACCESS:
        cprintf("Load access fault\n");
        ret = -E_INVAL;
        break;
    case CAUSE_MISALIGNED_STORE:
        cprintf("AMO address misaligned\n");
        ret = -E_INVAL;
        break;
    case CAUSE_STORE_ACCESS:
        cprintf("Store/AMO access fault\n");
        ret = -E_INVAL;
        break;
    case CAUSE_USER_ECALL:
        tf->epc += 4;
        syscall();
        return;
    case CAUSE_SUPERVISOR_ECALL:
        cprintf("Environment call from S-mode\n");
        tf->epc += 4;
        syscall();
        return;
    case CAUSE_HYPERVISOR_ECALL:
        cprintf("Environment call from H-mode\n");
```

```
50          ret = -E_INVAL;
51          break;
52      case CAUSE_MACHINE_ECALL:
53          cprintf("Environment call from M-mode\n");
54          ret = -E_INVAL;
55          break;
56      case CAUSE_FETCH_PAGE_FAULT:
57      case CAUSE_LOAD_PAGE_FAULT:
58      case CAUSE_STORE_PAGE_FAULT:
59          if (!in_kernel && current != NULL && current->mm != NULL)
60          {
61              /* error_code: write bit for store faults */
62              uint32_t err = (tf->cause == CAUSE_STORE_PAGE_FAULT) ? 1 : 0;
63              if ((ret = do_pgfault(current->mm, err, tf->tval)) == 0)
64              {
65                  return;
66              }
67          }
68          cprintf("Page fault failed, ret %d\n", ret);
69          break;
70      default:
71          print_trapframe(tf);
72          ret = -E_INVAL;
73          break;
74      }
75
76      if (ret != 0)
77      {
78          if (in_kernel)
79          {
80              panic("unhandled trap in kernel.\n");
81          }
82          else
83          {
84              print_trapframe(tf);
85              do_exit(-E_KILLED);
86          }
87      }
88  }
```

这样用户态异常不会无限循环，内核态保持崩溃提示。

测试：

```
s/switch.S + cc kern/schedule/default_sched.c + cc k
schedule/sched.c + cc kern/syscall/syscall.c + cc ke
 cc kern/fs/vfs/vfs.c + cc kern/fs/vfs/vfsdev.c + cc
kup.c + cc kern/fs/vfs/vfspath.c + cc kern/fs/devs/d
s/devs/dev_stdin.c + cc kern/fs/devs/dev_stdout.c +
 + cc kern/fs/sfs/sfs_fs.c + cc kern/fs/sfs/sfs_inod
/sfs_lock.c + ld bin/kernel riscv64-unknown-elf-objc
.img gmake[1]: Leaving directory '/mnt/d/HuaweiMoveD
lab8'
  -sh execve:                                    OK
  -user sh :                                     OK
Total Score: 100/100
```

结论：**实验基本成功。**

1.3 扩展练习 Challenge 1: UNIX PIPE 机制设计方案

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 设计目标与语义约束

Pipe 是一种内核提供的单向字节流通道，`pipe()` 创建后返回两个文件描述符：读端 `fd[0]` 与写端 `fd[1]`。两端共享同一段内核缓冲区，写端写入的数据会进入缓冲，读端从缓冲取走数据。Pipe 的关键语义在于阻塞行为与端点消失时的返回值：当缓冲为空且仍存在写端时，读操作应阻塞等待；当缓冲为空且所有写端都关闭时，读操作返回 0 表示 EOF；当缓冲已满且仍存在读端时，写操作应阻塞等待空间；当所有读端都关闭时继续写入应失败返回 `-EPIPE`（若系统实现信号，可进一步触发 SIGPIPE；uCore 没信号时返回错误码即可）。此外，为了避免并发写者把内容“交错写进同一段小消息”，通常还需要满足“长度不超过 `PIPE_BUF` 的写入具有原子性”的约束：要么整段写进去，要么整体阻塞/失败，而不是被拆开穿插。

2 核心数据结构设计（最小可行）

实现上把 Pipe 视为一个可共享的内核对象 `pipe_t`，内部用环形缓冲保存字节流，并用一把锁保护“缓冲状态 + 端点计数 + 引用计数”。每个 fd（文件描述符）对应一个 `pipe_file_t` 端点视图，用于区分这是读端还是写端以及是否非阻塞。

pipe 共享对象本体

`pipe_t`：管道内核对象，包含环形缓冲区状态（buf/cap/head/tail/len）、端点计数（readers/writers）、同步原语（lock + 两个等待队列）、生命周期引用计数（refcnt）。

pipe 端点视图

`pipe_file_t`：每个 fd 对应一个端点对象，记录它指向哪个 pipe（p）、它是读端还是写端（end）、以及该 fd 的标志位（flags，如 O_NONBLOCK）。

同步相关

`spinlock_t`（或 mutex）：保护 pipe_t 内部共享状态的互斥锁。

`wait_queue_t`：阻塞读/写用的等待队列（rwait/wwait），读空就睡在 rwait，写满就睡在 wwait

代码块

```
#define PIPE_CAP_DEFAULT   4096
#define PIPE_BUF_ATOMIC    4096   // 可选：用于实现“<=PIPE_BUF 原子写”策略

typedef struct wait_queue wait_queue_t;
typedef struct spinlock   spinlock_t;

typedef struct pipe {
    spinlock_t  lock;

    char       *buf;        // 环形缓冲区
    size_t      cap;        // 总容量
    size_t      head;       // 读指针
    size_t      tail;       // 写指针
    size_t      len;        // 当前占用字节数

    int         readers;    // 读端引用数（dup/fork 后会增加）
    int         writers;    // 写端引用数

    wait_queue_t rwait;     // 缓冲为空，读者睡眠
    wait_queue_t wwait;     // 缓冲满，写者睡眠

    int         refcnt;     // 对 pipe 对象的总体引用（也可用 readers+writers 推导）
} pipe_t;

typedef enum { PIPE_END_READ = 0, PIPE_END_WRITE = 1 } pipe_end_t;

typedef struct pipe_file {
    pipe_t     *p;          // 指向共享 pipe
    pipe_end_t  end;        // 读端/写端
    int         flags;      // O_NONBLOCK 等
} pipe_file_t;
```

这套结构的好处是：读写都只围绕 `pipe_t` 的状态变化，端点语义（读/写、阻塞/非阻塞）则放在 `pipe_file_t` 中，和 uCore 的 `struct file->private_data` 组合起来非常自然。

3 必要接口与语义（不要求实现，只交代行为）

系统调用层至少需要 `sys_pipe`，有余力可再加 `sys_pipe2` 以支持创建时直接设置 `O_NONBLOCK/O_CLOEXEC`。

代码块

```
int sys_pipe(int pipefd[2]);                // pipefd[0]=read, pipefd[1]=write
int sys_pipe2(int pipefd[2], int flags);    // 支持 O_NONBLOCK 等
```

内核文件抽象层（file_operations）至少需要 `read/write/close` 三个操作；若你们实验框架有 `poll/select`，可以再加 `poll` 以支持事件通知。

代码块

```
ssize_t pipe_read(struct file *f, void *buf, size_t n);
ssize_t pipe_write(struct file *f, const void *buf, size_t n);
int     pipe_close(struct file *f);
int     pipe_poll(struct file *f, poll_table *pt);   // 可选
```

读操作语义这样描述：读端调用 `pipe_read` 时先加锁检查 `len`。若 `len>0`，从环形缓冲拷贝最多 `n` 字节到用户缓冲，更新 `head/len`，随后唤醒 `wwait`（因为写端可能在等空间），释放锁并返回实际读取字节数。若 `len==0` 且 `writers>0`，阻塞模式应睡眠在 `rwait`，被唤醒后重新检查条件再读；非阻塞模式则立即返回 `-EAGAIN`。若 `len==0` 且 `writers==0`，说明不会再有数据写入，应返回 0 表示 EOF。

写操作语义这样描述：写端调用 `pipe_write` 时先加锁检查 `readers`。若 `readers==0`，直接返回 `-EPIPE`。否则检查剩余空间 `cap-len`：若有空间则写入（可能分段写入直到用完空间或完成 n 字节），更新 `tail/len`，唤醒 `rwait`（因为读端可能在等数据），释放锁并返回写入字节数。若空间不足且为阻塞模式，则睡眠在 `wwait`，醒来后重试；若为非阻塞模式则返回 `-EAGAIN`。为了满足小写原子性，可以规定当 `n<=PIPE_BUF_ATOMIC` 时必须“要么一次性写完，要么整体阻塞/失败”，从而避免多个并发写者把这段小消息拆开交错。

关闭语义可这样描述：`pipe_close` 根据端点类型递减 `readers` 或 `writers`。若关闭写端导致 `writers` 变为 0，需要唤醒 `rwait`，让阻塞读者醒来并最终读到 EOF。若关闭读端导致 `readers` 变为 0，需要唤醒 `wwait`，让阻塞写者醒来并发现 `-EPIPE`。当 `readers==0 && writers==0` 且 `refcnt` 归零时释放 `pipe_t` 与缓冲。

4 同步互斥与竞态处理

Pipe 的并发风险集中在三类场景：多线程/多进程同时读写、读写与 close 并发、阻塞唤醒丢失。为此需要在 `pipe_t` 内使用同一把锁保护 `head/tail/len/readers/writers/refcnt`，保证状态更新的一致性。阻塞读写必须采用“循环检查条件”的标准模式：在持锁状态下判断 empty/full，不满足则把当前任务加入对应等待队列并原子地释放锁进入睡眠，被唤醒后重新加锁再判断，直到条件满足或端点消失产生 EOF/EPIPE。状态变化后必须唤醒对端等待队列（读到数据后唤醒写者，写入

数据后唤醒读者，close 也要唤醒对端），以避免对端永久睡眠。为了降低锁持有时间，数据拷贝可按块进行，但任何一次更新 `head/tail/len` 的操作必须在锁保护下完成。

1.4 扩展练习 Challenge 2: UNIX 软连接和硬连接机制设计方案

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。）

1 设计目标与语义约束

硬链接的本质是“给同一个 inode 再加一个名字”。目录项（name → inode）可以有多条指向同一个 inode，因此 inode 需要维护硬链接计数 `nlink`。`link(old, new)` 会在 `new` 的父目录中新增一个目录项指向 `old` 的 inode，并令 `nlink++`；`unlink(path)` 删除目录项并令 `nlink--`。当 `nlink==0` 时并不一定立即释放 inode，因为该文件可能仍被进程打开，因此还需要一个“打开引用计数/活动引用计数”，只有当 `nlink==0` 且“打开引用也为 0”时才真正回收数据块与 inode，这就实现了“文件被 unlink 后，已打开的 fd 仍可继续读写，直到最后关闭才真正消失”的 UNIX 行为。

软链接的本质是“一个特殊类型文件，其内容是目标路径字符串”。它并不直接增加目标 inode 的 `nlink`，也可以跨文件系统，且可以指向目录。`symlink(target, linkpath)` 创建一个类型为 symlink 的 inode，把 `target` 路径字符串保存进去，再在目录中创建名为 `linkpath` 的目录项指向该 symlink inode。路径解析（namei/lookup）默认会跟随 symlink，把其中保存的路径继续解析；而 `lstat/readlink` 这类接口需要提供“不跟随最后一跳 symlink”的能力。

2 需要扩展/新增的数据结构（最小可行）

你只需要在 inode 上补齐文件类型与链接计数，再保证目录项结构能完成“名字到 inode 的映射”。下面给一套抽象结构，你可以替换成 uCore/sfs 实际的 inode/dirent 命名。

代码块

```
typedef struct spinlock spinlock_t;
typedef unsigned int    ino_t;

typedef enum {
    IT_REG, IT_DIR, IT_SYMLINK
} inode_type_t;

typedef struct inode {
    spinlock_t    lock;
    ino_t         ino;
    inode_type_t  type;
```

```
12
13      unsigned int  nlink;      // 硬链接计数：link++ / unlink--
14      unsigned int  open_cnt;   // 活动打开引用：open++ / close--
15
16      size_t        size;
17
18      // symlink 内容：保存目标路径字符串（短的可内联，长的可落盘到数据块）
19      char         *symlink;    // 仅 type==IT_SYMLINK 使用
20
21      void         *fs_private; // 文件系统私有信息（数据块索引等）
22  } inode_t;
23
24  #define NAME_MAX 255
25  typedef struct dirent {
26      ino_t  ino;
27      char   name[NAME_MAX + 1];
28  } dirent_t;
```

如果你们 VFS 有 dentry 缓存，也可以再在内存里做 `dentry{name,inode,parent}` 的缓存对象，但对“最少设计”来说，dirent + inode 就足够解释 hardlink/symlink 的语义。

3 必要接口与语义（系统调用 + VFS/FS 操作）

系统调用层建议至少包含这五个：hardlink、unlink、symlink、readlink、lstat/stat（二选一也可以，但报告写全更像 UNIX）。

代码块

```
1  int     sys_link(const char *oldpath, const char *newpath);          // 硬链接
2  int     sys_unlink(const char *path);                                // 删除目录项
3  int     sys_symlink(const char *target, const char *linkpath);       // 软链接
4  ssize_t sys_readlink(const char *path, char *buf, size_t bufsz);     // 读出软链接内容
5  int     sys_lstat(const char *path, struct stat *st);                // 不跟随最后一跳 symlink
6  int     sys_stat(const char *path, struct stat *st);                 // 默认跟随 symlink
```

VFS/FS 内部语义可以在报告里这样描述：

`sys_link(old, new)` 的核心动作是两步合一的原子更新：先解析 `old` 得到目标 inode，再解析 `new` 得到其父目录 inode 与新名字，在父目录中新增 dirent 指向该 inode，同时把 inode 的 `nlink` 加一。实现时通常限制“不能对目录做硬链接”，并要求 old 与 new 位于同一文件系统（避免跨 fs 的 inode 引用）。

`sys_unlink(path)` 的核心动作是：在父目录中移除该名字对应的 dirent，并将其 inode 的 `nlink` 减一。如果 `nlink` 变为 0 但 `open_cnt>0`，则仅表示“名字已消失但对象仍存活”，真正释放要推迟到最后一次 close 把 `open_cnt` 归零时进行。若 `nlink==0 && open_cnt==0`，即可立即回收 inode 与其数据块。

`sys_symlink(target, linkpath)` 的核心动作是：创建一个新 inode，类型为 `IT_SYMLINK`，把 `target` 字符串保存为该 inode 的内容（短的可直接放内存，长的可写入数据块），并在 `linkpath` 的父目录创建一个 dirent 指向该 symlink inode，symlink inode 的 `nlink` 初始为 1（它自己作为一个目录项存在）。

`sys_readlink(path)` 的语义是：解析到 symlink inode 但不跟随它，直接把 symlink inode 内保存的路径字符串拷贝到用户缓冲并返回长度（通常不保证 NUL 结尾，调用者按返回长度处理）。

路径解析（namei/lookup）需要加入“跟随 symlink”的逻辑：默认情况下遇到 symlink 就读取其中的 target 路径并继续解析，但必须设置最大跳转次数（例如 8/16/40）来避免循环链接导致无限递归，超过阈值则返回 `ELOOP`。

4 同步互斥与竞态处理

link/unlink/symlink 都会同时修改“目录项集合”和“inode 元数据（nlink/type/size 等）”，必须保证两者一致，否则会出现目录里已经多了名字但 nlink 没加、或者 nlink 变了但目录项没落盘的错误状态。最直接的策略是对目录 inode 使用互斥锁保护 dirent 的增删改，同时对目标 inode 使用锁保护 `nlink/open_cnt/type` 等元数据更新，并规定统一的加锁顺序来避免死锁。推荐的锁顺序是先锁父目录 inode，再锁目标 inode；如果一次操作涉及两个目录（例如 rename 或跨目录 link），则按照 inode 号从小到大加锁，保证所有线程遵循同一顺序。open 与 unlink 的并发由 `open_cnt` 与 `nlink` 的分离来化解：unlink 只影响名字和 `nlink`，open/close 只影响 `open_cnt`，回收条件统一检查 `nlink==0 && open_cnt==0`，从而保证“删除名字不影响已打开文件继续使用”的语义正确。

第二部分：Lab6- 进程调度

实验目标

- 理解操作系统的调度管理机制，特别是调度器框架的设计思想。
- 熟悉 ucore 的系统调度器框架，并基于此框架实现缺省的Round-Robin (RR)调度算法。
- 学习并实现一个更高级的调度算法——Stride Scheduling，以替换缺省的RR算法。

2.1 练习1: 理解调度器框架的实现

请仔细阅读和分析调度器框架的相关代码，特别是以下两个关键部分的实现：

在完成练习0后，请仔细阅读并分析以下调度器框架的实现：

- 调度类结构体 sched_class 的分析：请详细解释 sched_class 结构体中每个函数指针的作用和调用时机，分析为什么需要将这些函数定义为函数指针，而不是直接实现函数。

sched_class 结构体中一共有5个函数指针：

1. `init(struct run_queue *rq)` 的作用是把该算法需要的运行队列结构准备好。
   a. 它只会在调度器初始化阶段被调用一次。
   b. RR 会在这里初始化就绪链表头、计数器；stride 会在这里初始化斜堆根指针、比较规则所需字段等。重点是：框架不该知道你内部用链表还是堆，所以把初始化交给算法。
2. `enqueue(struct run_queue *rq, struct proc_struct *proc)` 的作用是确定一个进程变成 RUNNABLE 之后，如何进入候选集合。
   a. 典型调用点在 `wakeup_proc()`、新建进程变为 runnable、以及某些被抢占后重新排队的场景。
   b. RR 的 enqueue 通常是把进程挂到链表尾；stride 的 enqueue 通常是把进程按 pass/stride 的顺序插入斜堆。框架只负责在状态转换正确之后调用它。
3. `dequeue(struct run_queue *rq, struct proc_struct *proc)` 的作用是一个进程不再 RUNNABLE 时，如何从候选集合移除。
   a. 典型调用点在进程要睡眠/阻塞、退出、或者被选中运行时（不同实现可能在“选中后”才删除，或“选中前”就删除）。
   b. RR 是链表删除；stride 是堆删除/弹出或标记式删除。框架同样不关心细节。
4. `pick_next(struct run_queue *rq)` 的作用是在一次调度点发生时，按该算法选出下一个要运行的进程。
   a. 典型调用点在 `schedule()` 中，框架在做完一堆通用检查（中断、锁、当前进程状态处理等）后，会调用 `pick_next` 取到 `next`，然后完成上下文切换。
   b. RR 通常取队首；stride 通常取 pass 最小者（即堆顶）。
5. `proc_tick(struct run_queue *rq, struct proc_struct *proc)` 的作用是每次时钟中断 tick 到来时，对当前运行进程的调度相关状态做更新，并在必要时请求重调度。
   a. 典型调用点在时钟中断处理逻辑里。
   b. RR 会减少 time_slice，用尽时设置 `proc->need_resched = 1`；stride 可能按 tick 更新 `proc->pass += proc->stride` 或者在时间片边界上更新并触发 `need_resched`。关键点在于：`proc_tick` 通常不直接做上下文切换，它更像记账 + 发出需要切换的信号，真正切换在 `schedule()`。

为什么要是函数指针，而不是直接写死函数？原因是为了把“机制”和“策略”硬隔离开,保证公共流程与不变量的一致性。调度框架要保证的东西是跨算法一致的：进程状态合法转换、并发保护、

上下文保存/恢复、当前进程指针更新、队列计数一致性等；而算法变化的只有五个动作：队列怎么初始化、怎么入队、怎么出队、怎么选下一个、每个 tick 怎么更新。

将这些差异点抽象为函数指针，相当于在初始化阶段把“插头”接到某一套策略实现（如 RR 或 stride）上，运行时框架始终通过 `sched_class->xxx()` 调用当前策略，无需在核心路径里写大量 if-else 判断“现在用哪种算法”。因此添加或切换调度算法时，只需要新增/替换一个 `sched_class` 的实现与必要的数据结构字段，而无需修改 `schedule()` 的主干逻辑，既降低了耦合与修改范围，也更容易保证框架代码的正确性与可维护性。

- 运行队列结构体 run_queue 的分析：比较lab5和lab6中 run_queue 结构体的差异，解释为什么lab6的 run_queue 需要支持两种数据结构（链表和斜堆）。

lab5 根本没把“可运行集合”单独维护起来，lab6 把“可运行集合”抽出来做成 run_queue，并且为了兼容不同算法在同一框架下切换，run_queue 同时准备了链表和斜堆两套容器。

  - lab5 的 `schedule()`，它不是从“运行队列”里选下一个，而是从全局 `proc_list`（系统里所有进程的链表）里开始，沿着链表一个个找 `PROC_RUNNABLE` 的进程，找到第一个就跑，找不到就跑 `idleproc`。这意味着 lab5 的“候选集合”其实是“所有进程”，调度时再去筛一遍。于是它根本不需要一个复杂的 run_queue，顶多只要全局进程链表就能跑起来；对应地也能看到 lab5 的 `wakeup_proc()` 只做了一件事：把进程状态改成 `PROC_RUNNABLE`，但并没有把它放进任何就绪队列，因为 lab5 本来就没有维护单独的就绪队列，反正 `schedule()` 每次都会去 `proc_list` 里扫一遍。
  - lab6 的 `run_queue`，它的核心变化是：把可运行进程集合从所有进程集合中剥离出来，专门用 `rq` 来维护，所以 `rq->proc_num` 统计的是就绪队列里 RUNNABLE 的数量，而不是系统进程总数。这样做有两个直接收益：第一，调度选择只需要在 RUNNABLE 集合里做，不用每次遍历所有进程去过滤一遍；第二，框架可以在 `wakeup_proc()` 这种状态转换的地方，把进程加入候选集合（enqueue），把选谁跑的问题交给调度算法实现，从而实现解耦。

为什么 lab6 的 run_queue 需要同时支持链表和斜堆的答案就在算法需求不一样。RR 这种时间片轮转，队列语义就是先进先出：唤醒/新建进程放到队尾，调度时取队首，时间片用完再丢回队尾，这套操作用链表最自然，入队出队都是 O(1)，实现也简单，所以 `run_list`（链表）是为 RR 服务的。Stride 的核心语义却不是 FIFO，而是每次选 pass 最小的那个进程（每次运行对应进程，对应进程的pass就会增大），也就是一个典型的优先队列/最小堆问题；如果仍然用链表维护，那么 `pick_next` 就只能每次在链表里线性扫描找最小 pass，复杂度会退化成 O(n)，进程一多调度开销就很明显。斜堆（skew heap）是教学里常用的可并堆结构，插入、删除最小值等操作均摊复杂度较好，实现比红黑树、二叉堆的接口更指针友好，很适合在内核里做按 key 取最小的就绪队列，所以 `lab6_run_pool` 是为 stride 服务的。

- 调度器框架函数分析：分析 sched_init()、wakeup_proc() 和 schedule() 函数在lab6中的实现变化，理解这些函数如何与具体的调度算法解耦。

- `sched_init()` 在 lab6 的变化点是把调度算法的选择和运行队列的初始化抽象出来。lab5 往往没有独立的 run_queue 概念（或即使有也只是链表），选择进程靠遍历 `proc_list` 过滤 RUNNABLE；而 lab6 明确做了两件事：先把全局的 `rq` 指向一个统一的运行队列对象 `__rq` 并设置通用参数 `max_time_slice`，再把 `sched_class` 指向 `default_sched_class` 并调用 `sched_class->init(rq)` 让具体算法在这块 `rq` 上建好它自己的数据结构（RR 用 `run_list`，stride 用 `lab6_run_pool`）。这一改动的含义是：框架只负责提供容器和统一入口，算法通过函数指针获得“在 rq 上如何组织候选集合”的控制权，从初始化阶段就完成了策略绑定，因此以后切换算法只需要换 `sched_class` 指向哪个实现，`sched_init()` 的骨架不变。
- `wakeup_proc()` 在 lab6 的变化点是唤醒不再只是改状态，而是把进入可运行集合交给调度类。你这段代码先做机制层必须做的事：关中断保护、把进程从非 RUNNABLE 改成 RUNNABLE、清掉 `wait_state`。真正体现解耦的是这一句：`if (proc != current) { sched_class_enqueue(proc); }`。这意味着 lab6 不再像 lab5 那样依赖 `schedule()` 以后去全局列表里扫一遍找 runnable，而是谁变 runnable，谁立刻入队。同时入队方式完全不在 `wakeup_proc()` 里写死：RR 的 enqueue 是链表尾插，stride 的 enqueue 是按 pass 插入斜堆；框架完全不关心。总的来说唤醒路径从“状态转换”扩展为“状态转换 + 候选集合维护”，而候选集合维护通过 `sched_class` 接口完成，从而把策略差异隔离在具体算法实现内部。
- `chedule()` 的变化是 lab6 解耦最核心的证据：lab5 的 `schedule()` 自己在 `proc_list` 上循环寻找下一个 RUNNABLE，而 lab6 的 `schedule()` 完全不再触碰“遍历所有进程”的逻辑，转而只做三件与算法无关的事。第一件是消耗触发信号：`current->need_resched = 0`，表明这次调度请求已经处理。第二件是维护当前进程的生命周期位置：如果 `current` 仍然 RUNNABLE，就把它放回候选集合 `sched_class_enqueue(current)`，这一步相当于把刚跑完时间片的人重新加入可选集合，但加入的位置由算法决定（RR 放队尾，stride 插堆后等待按 pass 再被选）。第三件是选择并且切走：通过 `next = sched_class_pick_next()` 让算法挑下一个候选，如果挑到了就 `sched_class_dequeue(next)` 把它从候选集合移除（保持 rq 的一致性），否则就跑 `idleproc`，最后统一用 `proc_run(next)` 做上下文切换。这里会发现这里框架只做什么时候入队/出队/切换的时机控制，而pick_next 究竟按 FIFO 还是按最小 pass”完全由调度类决定，所以 `schedule()` 的主干对 RR 和 stride 都成立，实现了解耦。

对于调度器框架的使用流程，请在实验报告中完成以下分析：

- 调度类的初始化流程：描述从内核启动到调度器初始化完成的完整流程，分析 default_sched_class 如何与调度器框架关联。

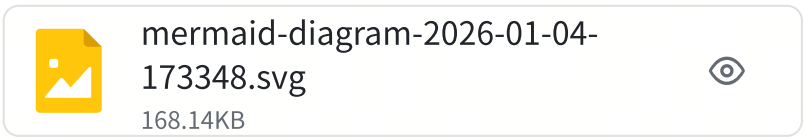
内核完成最基础的硬件与中断子系统初始化后，会建立进程管理的基本结构并创建 idleproc/current 等关键进程对象，同时完成时钟/定时器初始化以产生周期性的时钟中断；随后进入调度器初始化 `sched_init()`，首先初始化用于延时唤醒的 `timer_list`，再把全局调度类指针 `sched_class` 指向 `default_sched_class`，并把全局运行队列指针 `rq` 指向静态队列对象 `__rq`，设置通用调度参数如 `rq->max_time_slice`，最后调用 `sched_class-`

`>init(rq)` 让“默认调度算法”在这块 `rq` 上完成自身所需数据结构的初始化（RR 会初始化 `run_list`、stride 会初始化 `lab6_run_pool` 等）。从这一刻开始，调度框架不再直接操作某种具体队列，而是通过 `sched_class->enqueue/dequeue/pick_next/proc_tick` 来驱动调度过程；

因此 `default_sched_class` 与框架的关联点就是全局指针 `sched_class`，它在初始化时被挂载到框架上，后续框架通过它间接调用默认算法的实现。

- 进程调度流程：绘制一个完整的进程调度流程图，包括：时钟中断触发、proc_tick 被调用、schedule() 函数执行、调度类各个函数的调用顺序。并解释 need_resched 标志位在调度过程中的作用

流程图太大，这里提供svg形式。



need_resched 标志位的作用是把“产生调度需求”和“真正执行调度”解耦。时钟中断里通常不直接做完整调度（因为在中断上下文里、可能还持有锁、也不希望在非常深的内核路径里强行切走），而是由 `proc_tick` 这种轻量逻辑把 `need_resched` 置 1，相当于贴一个“我该换人了”的便签；等中断/系统调用/异常处理即将返回到安全点（常见就是 trap 返回路径）时，再检查这个标志并调用 `schedule()`，这样既避免了在不安全上下文里切换造成的竞态和死锁风险，也能保证响应足够及时。与此同时，`yield()` 这类“主动让出 CPU”的操作也通常就是把 `need_resched` 置位，让流程统一走同一条调度通道。

- 调度算法的切换机制：分析如果要添加一个新的调度算法（如stride），需要修改哪些代码？并解释为什么当前的设计使得切换调度算法变得容易。

首先新增一个 stride 调度类实现文件/模块，提供 `stride_init/stride_enqueue/stride_dequeue/stride_pick_next/stride_proc_tick` 这些函数，并定义一个 `struct sched_class stride_sched_class`，把函数指针字段填为上述实现；其次为 stride 所需的状态补充数据字段，常见包括在 `proc_struct` 中增加 `pass/stride/priority(或 tickets)` 等记账字段，以及在 `run_queue` 中准备优先队列容器字段（你们这里就是 `skew_heap_entry_t *lab6_run_pool`）；最后在 `sched_init()` 中把 `sched_class = &default_sched_class` 改为 `sched_class = &stride_sched_class`（或用宏/编译选项在两者间切换）。之所以当前设计让切换变得容易，是因为调度框架（`schedule/wakeup_proc` 等）只依赖 `sched_class` 这套稳定接口：框架永远只在固定时机调用 `enqueue/dequeue/pick_next/proc_tick`，并不直接操作链表或斜堆，也不写死“选队首/选最小 pass”的规则；因此新增或更换算法只是在同一接口下替换一套实现，修改范围被限制在调度类实现与少量数据承载字段上，而不需要改动框架主干逻辑，这正是“机制固定、策略可替换”的解耦收益。

2.2 练习2: 实现 Round Robin 调度算法

完成练习0后，建议大家比较一下（可用kdiff3等文件比较软件）个人完成的lab5和练习0完成后的刚修改的lab6之间的区别，分析了解lab6采用RR调度算法后的执行过程。理解调度器框架的工作原理后，请在此框架下实现时间片轮转（Round Robin）调度算法。

注意有“LAB6”的注释，你需要完成 kern/schedule/default_sched.c 文件中的 RR_init、RR_enqueue、RR_dequeue、RR_pick_next 和 RR_proc_tick 函数的实现，使系统能够正确地进行进程调度。代码中所有需要完成的地方都有“LAB6”和“YOUR CODE”的注释，请在提交时特别注意保持注释，将“YOUR CODE”替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。

提示，请在实现时注意以下细节：

- 链表操作：list_add_before、list_add_after等。
- 宏的使用：le2proc(le, member) 宏等。
- 边界条件处理：空队列的处理、进程时间片耗尽后的处理、空闲进程的处理等。

请在实验报告中完成：

- 比较一个在lab5和lab6都有, 但是实现不同的函数, 说说为什么要做这个改动, 不做这个改动会出什么问题
  - 提示: 如`kern/schedule/sched.c`里的函数。你也可以找个其他地方做了改动的函数。
- 描述你实现每个函数的具体思路和方法，解释为什么选择特定的链表操作方法。对每个实现函数的关键代码进行解释说明，并解释如何处理**边界情况**。
- 展示 make grade 的**输出结果**，并描述在 QEMU 中观察到的调度现象。
- 分析 Round Robin 调度算法的优缺点，讨论如何调整时间片大小来优化系统性能，并解释为什么需要在 RR_proc_tick 中设置 need_resched 标志。
- **拓展思考**：如果要实现优先级 RR 调度，你的代码需要如何修改？当前的实现是否支持多核调度？如果不支持，需要如何改进？

在ucore中，进程有如下几个状态：

- `PROC_UNINIT`：未初始化状态，需要进一步的初始化才能进入`PROC_RUNNABLE`的状态。
- `PROC_SLEEPING`：等待锁释放/主动交出CPU资源。
- `PROC_RUNNABLE`：准备好执行。
- `PROC_ZOMBIE`：进程退出。

一个进程的生命周期一般由如下过程组成：

`PROC_UNINIT->PROC_RUNNABLE->调度器选中执行->running`

`->wait` 等系统调用->阻塞-> `PROC_SLEEPING` ，等待相应的资源或者信号。

->被外部中断打断-> `PROC_RUNNABLE` 状态，等待下次被调用->等待的事件发生-> `PROC_RUNNABLE`

重复3~6，直到进程执行完毕，通过 `exit` 进入 `PROC_ZOMBIE` 状态，由父进程对他的资源进行回收，释放进程控制块。至此，这个进程的生命周期彻底结束

进程切换可以分为两类情形——**主动调度**与**被动调度**。前者确保内核线程在不可抢占的环境下仍然保持系统活性，后者确保用户进程能够被及时打断，避免独占 CPU。

RR调度算法中实现每个函数的具体思路和方法

当 CPU 正在运行某个进程 A，每次时钟中断都会进入 RR_proc_tick(rq, A)，A 的 time_slice 逐步减少；如果 A 在一个时间片内就自己发起 I/O（比如读盘）而进入阻塞，那它会在系统调用或异常路径上被设置为非 runnable 并 RR_dequeue，CPU 立刻调用调度器去 RR_pick_next 选另一个 runnable 的进程继续跑；如果 A 没阻塞也没结束，而是把 time_slice 正好跑到 0，那么最后一次 tick 只会把 need_resched 置 1，等从中断返回的那条路径进入 schedule()，调度框架会看到 current->need_resched，于是把 A 从 RUNNING 改回 RUNNABLE，并把 A 重新 RR_enqueue 到队尾（这一步就会因为 A 的 time_slice==0 而补满到 q），然后再 RR_pick_next 从队头挑一个 B 运行，完成一次轮转。

1.初始化，RR_init 的思路是把 run_queue 置为一个空的就绪队列。RR 依赖 rq->run_list 作为循环双向链表的哨兵头结点，所以必须用 list_init(&(rq->run_list)) 把头结点的 prev/next 都指回自身，表示队列为空且结构合法；再把 rq->proc_num = 0，让调度器知道当前没有可运行进程。这里不设置 max_time_slice 是因为它由外部（调度框架）统一决定和传入，避免在初始化阶段覆盖配置。边界上，空队列的正确表示完全依赖 list_init，因此这一步是后续所有 list_add/list_del 不出错的基础。

代码块

```
static void
RR_init(struct run_queue *rq)
{
    // LAB6: YOUR CODE
    list_init(&(rq->run_list));
    rq->proc_num = 0;
}
```

2.RR_enqueue 的思路：所有 runnable 进程入队都排到队尾，并确保它携带正确的时间片额度。

首先用 assert(list_empty(&(proc->run_link))) 防止同一个进程节点被重复挂进链表：双向链表如果重复插入会导致环断裂或形成多重链接，属于致命结构损坏；这个断言相当于强制要求入队前必须处于未入队状态。

插入操作选择 list_add_before(&(rq->run_list), &(proc->run_link)) 是因为 rq->run_list 是哨兵头结点，在哨兵之前插入等价于插到队尾（也就是 head->prev 的位置），这比手动找尾结点更安全也更快，并且不会因为队列为空/非空而分支处理，空队列时哨兵前就是哨兵本身，仍然成立。

时间片处理上，if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) proc->time_slice = rq->max_time_slice; 的含义是，如果该进程时间片用完（常见于刚被抢占下 CPU 后重新入队）就补满一轮；如果时间片被错误写成超过上限，也强制纠正为上限，避免某个进程占用 CPU 过久破坏公平。而对“还剩一点时间片”的情况（比如 I/O 阻塞后提前让出 CPU、唤醒回来）则保留剩余值，这能让交互型/IO-bound 任务更快完成短计算段，响应更好。最后 proc->rq = rq 记录所属队列，方便 dequeue 时做一致性校验；rq->proc_num++ 维护元数据，用于统计和调试。边界情况主要是：队列为空时插入仍然正确；重复入队用 assert 直接阻止；time_slice 异常值通过补满/纠正保证不会出现负值或无限大导致的调度异常。

代码块

```
static void
RR_enqueue(struct run_queue *rq, struct proc_struct *proc)
{
    // LAB6: YOUR CODE
    assert(list_empty(&(proc->run_link)));
    list_add_before(&(rq->run_list), &(proc->run_link));
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
        proc->time_slice = rq->max_time_slice;
    }
    proc->rq = rq;
    rq->proc_num ++;
}
```

3.RR_dequeue 的思路是把一个进程从就绪队列中移除（它可能要去阻塞、退出，或者被调度器选中即将运行并从队列摘下）。关键是保证“删的就是本队列里的那个节点”。

因此先用 assert(!list_empty(&(proc->run_link)) && proc->rq == rq) 做一致性检查：run_link 不能为空意味着它确实处在某个链表中，而 proc->rq == rq 则防止跨队列误删（比如误把别的 run_queue 的进程拿来 dequeue），这类 bug 会导致两个队列结构都损坏。

删除操作选 list_del_init(&(proc->run_link)) 而不是仅 list_del，理由是 del_init 会在把节点摘掉之后把该节点重新初始化为“未链接状态”（prev/next 指向自己），这样下一次 enqueue 时 list_empty(&proc->run_link) 才会成立，能够与 enqueue 的断言配合形成闭环，避免“删完了但节点指针还残留旧链接”的隐蔽错误。

最后 rq->proc_num-- 维护数量。边界上：如果队列里只有一个元素，list_del_init 也能正确把哨兵恢复成空链表；如果误删非本队列元素，会被断言提前截断。

代码块

```
static void
```

```
RR_dequeue(struct run_queue *rq, struct proc_struct *proc)
{
    // LAB6: YOUR CODE
    assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
    list_del_init(&(proc->run_link));
    rq->proc_num --;
}
```

4.RR_pick_next 的思路是从就绪队列中选择下一位运行进程，RR 选择规则是队头优先（最早进入队列的先运行）。实现时先用 list_next(&(rq->run_list)) 取哨兵的 next，也就是队头节点；

如果队列为空，哨兵的 next 还是哨兵本身，所以用 if (le != &(rq->run_list)) 判断空队列，空则返回 NULL（通常表示只能运行 idle 或无可运行进程）。不需要额外遍历，因为 RR 不做优先级比较，O(1) 取队头即可。得到链表节点后，用 le2proc(le, run_link) 从 run_link 指针反推 proc_struct*，这是典型的 container_of 模式：链表节点只是嵌在 PCB 里的一个字段，需要转回进程控制块。边界情况是队列为空直接返回 NULL，避免非法解引用；队列非空时总能取到一个合法进程指针。

代码块

```
static struct proc_struct *
RR_pick_next(struct run_queue *rq)
{
    // LAB6: YOUR CODE
    list_entry_t *le = list_next(&(rq->run_list));
    if (le != &(rq->run_list)) {
        return le2proc(le, run_link);
    }
    return NULL;
}
```

5.RR_proc_tick 的思路是把“时间片轮转”的抢占触发逻辑放在每次时钟 tick 中断里完成：

每来一次 tick，就消耗当前运行进程的一个时间片单位；当时间片耗尽，就设置“需要重新调度”的标志。

关键代码 if (proc->time_slice > 0) proc->time_slice--; 先保证不会把 time_slice 减成负数（这类负数会让后续逻辑混乱，甚至导致永远不触发 resched 或反复触发）；

紧接着 if (proc->time_slice == 0) proc->need_resched = 1; 表示时间片用尽，要求调度框架在中断返回后进入 schedule() 进行上下文切换。这里不直接在 tick 中断里做 enqueue/dequeue 或状态切换，是因为中断上下文做完整调度更复杂，通常框架采用“中断里记账 + 打标志，安全点再调度”的设计。

边界情况方面：如果 time_slice 原本就为 0（例如某些异常情况），不会继续减负，但会直接触发 need_resched，确保不会出现“时间片已耗尽却继续跑”的不公平；若进程在 q 之前主动阻塞/退

出，那它会在别的路径被 dequeue，tick 只负责当前正在运行者的时间片消耗。

代码块

```
static void
RR_proc_tick(struct run_queue *rq, struct proc_struct *proc)
{
    // LAB6: YOUR CODE
    if (proc->time_slice > 0) {
        proc->time_slice --;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}

```

RR 用 run_list 作为哨兵循环双向链表，list_add_before(head, node) 能在 O(1) 时间把进程插到队尾且兼容空队列；list_next(head) 能 O(1) 取队头；list_del_init(node) 删除后把节点恢复为未链接状态，能与 enqueue 的 list_empty 断言闭环配合，最大化避免链表结构被重复插入/误删导致的不可恢复错误。

比较kern/schedule/sched.c中的函数

lab5 achedule

代码块

```
void schedule(void)
{
    bool intr_flag;
    list_entry_t *le, *last;
    struct proc_struct *next = NULL;
    local_intr_save(intr_flag);
    {
        current->need_resched = 0;
        last = (current == idleproc) ? &proc_list : &(current->list_link);
        le = last;
        do
        {
            if ((le = list_next(le)) != &proc_list)
            {
                next = le2proc(le, list_link);
                if (next->state == PROC_RUNNABLE)
                {
                    break;
```

```
                }
            }
        } while (le != last);
        if (next == NULL || next->state != PROC_RUNNABLE)
        {
            next = idleproc;
        }
        next->runs++;
        if (next != current)
        {
            proc_run(next);
        }
    }
    local_intr_restore(intr_flag);
}
```

lab6 schecule

代码块

```
static inline void
sched_class_enqueue(struct proc_struct *proc)
{
    if (proc != idleproc)
    {
        sched_class->enqueue(rq, proc);
    }
}

static inline void
sched_class_dequeue(struct proc_struct *proc)
{
    sched_class->dequeue(rq, proc);
}

static inline struct proc_struct *
sched_class_pick_next(void)
{
    return sched_class->pick_next(rq);
}

void sched_class_proc_tick(struct proc_struct *proc)
{
    if (proc != idleproc)
    {
        sched_class->proc_tick(rq, proc);
```

```
    }
    else
    {
        proc->need_resched = 1;
    }
}

static struct run_queue __rq;

void sched_init(void)
{
    list_init(&timer_list);

    //sched_class = &default_sched_class;
    // stride_sched_class
    sched_class = &stride_sched_class;
    rq = &__rq;
    rq->max_time_slice = MAX_TIME_SLICE;
    sched_class->init(rq);

    cprintf("sched class: %s\n", sched_class->name);
}
void schedule(void)
{
    bool intr_flag;
    struct proc_struct *next;
    local_intr_save(intr_flag);
    {
        current->need_resched = 0;
        if (current->state == PROC_RUNNABLE)
        {
            sched_class_enqueue(current);
        }
        if ((next = sched_class_pick_next()) != NULL)
        {
            sched_class_dequeue(next);
        }
        if (next == NULL)
        {
            next = idleproc;
        }
        next->runs++;
        if (next != current)
        {
            proc_run(next);
        }
    }
```

```
74          local_intr_restore(intr_flag);
75      }
```

主要差异

- Lab5 在 sched.c 里直接遍历全局 proc_list 找 PROC_RUNNABLE，调度策略写死在 schedule()。
- Lab6 在 sched.c 引入 sched_class + run_queue：schedule() 只负责把可运行进程入队/出队并调用 pick_next，具体策略由 RR/stride 模块实现；时钟中断每 tick 调 sched_class_proc_tick() 驱动时间片。

为什么要改

- Lab6 要支持可替换的调度策略（RR、stride/优先级），stride 还需要按最小 stride（通常用堆/优先队列）高效选下一个进程；因此必须把选谁跑的策略从 schedule() 抽象成 sched_class，并维护就绪队列而不是每次扫全体进程。

make grade 的输出结果，并描述在 QEMU 中观察到的调度现象：

```
(base) root@wenxin:/mnt/d/HuaweiMoveData/Users/20105/Documents/xinan/dasanshang/OS/lab6# make grade
priority:                 (4.1s)
  -check result:                             OK
  -check output:                             OK
Total Score: 50/50
```

```
sched class: RR_scheduler
++ setup timer interrupts
kernel_execve: pid = 2, name = "priority".
set priority to 6
main: fork ok,now need to wait pids.
set priority to 1
set priority to 2
set priority to 3
set priority to 4
set priority to 5
child pid 3, acc 388000, time 2010
child pid 4, acc 388000, time 2010
child pid 5, acc 380000, time 2010
child pid 6, acc 396000, time 2010
child pid 7, acc 384000, time 2020
main: pid 0, acc 388000, time 2020
main: pid 4, acc 388000, time 2020
main: pid 5, acc 380000, time 2020
main: pid 6, acc 396000, time 2020
main: pid 7, acc 384000, time 2020
main: wait pids over
sched result: 1 1 1 1 1
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:577:
    initproc exit.
```

可以看到 make grade 的结果显示，程序获得满分。

在 make qemu 中，我们明显看到sched class: RR_scheduler，表明当前内核实际启用的调度类就是我们实现的 RR，并开启了定时器中断（sched class: RR_scheduler、++ setup timer interrupts），随后 fork 出多个子进程后，各个子进程在几乎相同的时间窗口内获得了接近的 CPU 运行份额——表现为每个 child 打印的 acc（累加量）数值彼此非常接近（大致都在 38～39 万附近），time 也基本一致（2010/2020 附近），说明进程之间被轮流分配 CPU，没有出现某一个进程长期独占或明显饥饿。

同时，最终统计结果 sched result: 1 1 1 1 1 是全部通过的均衡结果。

最后，所有用户态进程正常结束（all user-mode processes have quit），内核完成收尾检查（init check memory pass），随后因 initproc 退出触发 panic 停机（initproc exit）。

分析 Round Robin 调度算法的优缺点，讨论如何调整时间片大小来优化系统性能，并解释为什么需要在 RR_proc_tick 中设置 need_resched 标志

RR 的优点是公平 + 响应：它用 FIFO 就绪队列配合固定时间片，让每个 runnable 进程都能周期性拿到 CPU，不容易出现某个进程长期霸占导致其他进程饥饿，尤其适合交互型/多用户场景；另外它的实现简单、开销可控，队头取人、队尾入队都是 O(1)。

RR 的缺点主要来自时间片和中断的代价：时间片太小会把连续计算任务硬切碎，带来频繁上下文切换（保存/恢复寄存器、TLB/缓存扰动等），吞吐下降，很多时候只是让别的任务“看起来被照顾”但系统总体更慢；而当时间片/周期 tick 缩到微秒级，如果仍保持固定周期时钟中断，就会出现系统被“无意义唤醒/检查”淹没的极端情况——即使没竞争也不断打断，体感与能耗都变差，所以现代内核会走向 tickless/no-HZ 的按需定时：能预判进程会因 I/O 自然让出就不设周期 tick，只在可能长时间纯计算时用一次性定时器兜底公平与响应。

时间片大小的调整本质是在响应性 vs 吞吐/开销之间取平衡：时间片大，切换少、吞吐好，但交互响应变慢、短任务等待变长；时间片小，响应更快、更公平，但切换开销和缓存扰动急剧上升。经验做法是让时间片落在典型 CPU burst 的量级（覆盖大多数短 burst），并在高负载/多任务时更强调响应，在低负载或单任务时减少无谓抢占（这也是 tickless/no-HZ 的动机之一）。

在 `RR_proc_tick` 里设置 `need_resched` 的原因是职责分离与安全性：时钟中断里只做记账+触发，即把当前进程 `time_slice--`，一旦耗尽就打上 `need_resched=1`；真正的队列操作（enqueue/dequeue）、状态切换和上下文切换通常在中断返回后的调度点统一执行，这样避免在中断上下文里做复杂切换导致的重入、锁与栈/上下文一致性问题，同时也让调度框架能在一个统一位置处理“为何调度”（时间片耗尽、显式让出、唤醒抢占等）并复用同一套切换路径。

如果要实现优先级 RR 调度，你的代码需要如何修改？当前的实现是否支持多核调度？如果不支持，需要如何改进？

如果要实现优先级 RR（Priority Round Robin），代码需要从单一队列升级到分级队列/按优先级选择，有两种改法：第一种是多级就绪队列（MLFQ 风格的简化版），在 run_queue 里维护 run_list[NR_PRIO]（每个优先级一个 RR 队列）和对应的 proc_num[NR_PRIO]；enqueue 根据 proc->priority 把进程插到对应优先级队尾，并可让不同优先级有不同 max_time_slice（高优先级可更短时间片以保响应，或更长以给吞吐）；pick_next 从最高优先级非空队列取队头；dequeue 从其所在优先级队列删掉；proc_tick 仍只负责耗尽并置 need_resched，但还可以加入“时间片耗尽是否

降级/是否保持优先级”的策略。第二种是仍用一个队列但在 pick_next 里线性扫描挑最高优先级，这实现简单但每次选人 O(n)，在进程多时不划算。

至于多核：当前我们实现是单 run_queue + 单就绪队列的单核调度模型，不天然支持多核并发调度。多核下至少要解决两件事：一是并发安全（多个 CPU 同时 enqueue/dequeue/pick_next 会发生链表竞争，需要锁或关中断保护），二是负载均衡（不能所有核都从同一个队列取或只靠一个核调度）。改进方向通常是 per-CPU run_queue：每个 CPU 有自己的 run_queue 和就绪链表，enqueue 默认放到“本核队列”或按亲和度/负载选择目标核，pick_next 只从本核队列取，减少锁竞争；再加一个 periodic 或事件触发的 load balance（如工作窃取 work stealing：当某核空闲时从别的核队尾偷一些 runnable 过来），同时结合亲和度避免过度迁移导致缓存变冷。

2.3 扩展练习 Challenge 1: 实现 Stride Scheduling 调度算法

首先需要换掉RR调度器的实现，在sched_init中切换调度方法。然后根据此文件和后续文档对Stride度器的相关描述，完成Stride调度算法的实现。 注意有“LAB6”的注释，主要是修改default_sched_stride_c中的内容。代码中所有需要完成的地方都有“LAB6”和“YOUR CODE”的注释，请在提交时特别注意保持注释，将“YOUR CODE”替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。

请在实验报告中完成：

- 简要说明如何设计实现”多级反馈队列调度算法“，给出概要设计，鼓励给出详细设计

多级反馈队列要解决两个方面的问题，首先，它要优化周转时间，这通过先执行短工作来实现，然而，操作系统通常不知道工作要运行多久，而这又是SJF(或者STC F)等算法所必须的，其次，MLFQ希望给交互用户（例如用户坐在幕前，等着进程响应)很好的交互体验，因此需要降低响应时间，然而，像轮转这样的算法虽然降低了响应时间，周转时间却很差。

什么是周转时间？**周转时间 = 任务从提交到完成的总时间**。在SJF或者STCF中，有短任务尽快完成，不要让短任务被长任务堵住，但是面临的问题是不知道每个人物的运行时间，还有如果短任务持续进入，那长任务可能会被饿死。

什么是响应时间？**响应时间 = 用户发出请求到系统第一次响应的时间**。在交互式任务中，比如敲键盘后的响应，如果响应时间太长，那用户体验就会很不好。

多级反馈队列是一种结合“优先级 + 时间片轮转 + 动态反馈调整”的调度算法，虽然没有先验知识，但是可以在运行时通过运行时状态数据进行动态调整。

设计以下规则：

1.基本规则

1.如果 A 的优先级大于 B 的优先级，运行 A，不运行 B

2..如果 A 的优先级等于 B 的优先级，论转运行 A 和 B .

2.优先级改变规则，动态调整

1.工作进入系统时，放在最高优先级，也就是最上层队列.

2.工作用完整个时间片后，降低其优先级（移入下一级队列)

3.如果工作在其时间片以内主动释放CPU说明是交互型应用,则优先级不变，减少响应时间

3.为了防止饿死的操作：如果应用程序恶意冒充交互型应用的行为，抢占CPU资源，那该应用就会一直占据最高优先级，此时如果没有强制的休眠等干预进程操作，那CPU密集型长任务就会被饿死，通过以下两种手段防止这种现象。

1.经过一段时间，就将系统中所有工作重新加入最高优先级队列.

2.一旦工作用完了其在某一层中的时间配额，无论中间主动放弃了多少次CPU，就降低其优先级移入低一级队列.（而不是在每次进程调度时就刷新时间片剩余，这样I/O密集型任务反复让出CPU，其时间片剩余时间就一直刷新，很难降级）

- 简要证明/说明（不必特别严谨，但应当能够”说服你自己“），为什么Stride算法中，经过足够多的时间片之后，每个进程分配到的时间片数目和优先级成正比。

在stride算法中，**stride=pass+max_stride/priority**

不同进程设置的优先级不同，那每调用一次后前进的步伐即stride的单次累加值不同，很直观的感受是，如果优先级越大那单次stride前进步长就越小，而在stride算法中pick_next会通过堆选择stride最小的进程。那总体来看，优先级越大，stride变化越缓慢，更容易被选择，获得的时间片数目越多。

尝试从理论上进行证明，在stride算法中，每次选择stride小的那一个，目的是让stride区域平衡，不会出现一个进行被一直选择，因为该进程如果被选择stride就会增大，就会出现比该进程的更小的stride。

所以在足够长时间后，会出现一个“平衡”状态：

stride_i ≈ stride_j

又因为进程 i 每运行一次时间片，其 stride 增加：

pass_i = BIG_STRIDE / p_i

若进程 i 总共运行了 N_i 个时间片，则：

stride_i ≈ N_i * (BIG_STRIDE / p_i)

同理：

stride_j ≈ N_j * (BIG_STRIDE / p_j)

由 stride_i ≈ stride_j 得：

N_i * (BIG_STRIDE / p_i) ≈ N_j * (BIG_STRIDE / p_j)

综上：

N_i / N_j ≈ p_i / p_j

因此时间片分配与 priority 成正比

请在实验报告中简要说明你的设计实现过程。

根据lab6的框架，首先有如下总体的发现：

- 新增加的调度算法主要实现：
  - 根据自己算法的规则对任务队列进行维护，包括将runable进程加入队列，将被调度的线程移出队列；
  - 根据算法特性，在需要进行进程切换时选择对应的进程进行执行；
  - 处理时钟中断时的对应操作。
- 新增加的schedule文件中封装了更复杂的调度逻辑，首先是schedule函数，该函数会使用不同进程调度算法的pick_next算法，再将选择的进程从队列中移除，然后进行进程切换；
- wakeup_proc也进行了修改，通过该函数将sleeping或者阻塞的进程加入任务队列，等待下一次被调度。这个地方我记得在lab5答辩的时候，助教问过这个问题：在do_wait的时候如果当前进程有子进程，则将当前进程休眠，并且设置为等待子进程状态改变事件发生，那子进程如果变成了ZOMBIE，那怎么切换到父进程？
  - 父进程会被唤醒wake_up，然后加入到任务队列，等待被调度。

代码块

```
if (haskid)
    {
        current->state = PROC_SLEEPING;
        current->wait_state = WT_CHILD;
        schedule();
        if (current->flags & PF_EXITING)
        {
            do_exit(-E_KILLED);
        }
        goto repeat;
    }
```

这里简单讲解一下stride算法的实现方式：

- 采用优先队列存储进程信息。在堆的大小比较时，为了防止溢出回环，采用**用差值的符号来判断大小的方法。**

  在 n 位无符号整数中：在无符号计算里，需要对结果取模，因为无符号只能表示一定范围，为了能够处理回环的问题，我们采用差值溢出法，差值溢出就考虑了回环的情况，利用无符号计算方法得到的结果用有符号表示。

代码块

```
static int
proc_stride_comp_f(void *a, void *b)
```

```
{
     struct proc_struct *p = le2proc(a, lab6_run_pool);
     struct proc_struct *q = le2proc(b, lab6_run_pool);
     int32_t c = p->lab6_stride - q->lab6_stride;
     if (c > 0)
          return 1;
     else if (c == 0)
          return 0;
     else
          return -1;
}
```

- stride算法的初始化：初始化堆的根节点是null，可用线程是0，创造一个双向循环链表的表头。（这里run_quene支持用双向链表和堆来实现stride，我们选择用堆，提高排序的效率，用双向链表需要在插入和删除时从头遍历链表）

代码块

```
stride_init(struct run_queue *rq)
{
     list_init(&(rq->run_list));
     rq->lab6_run_pool = NULL;
     rq->proc_num = 0;
}
```

- stride算法的堆的维护

  采用我们自定义的排序算法将指定进程插入队列

  设置插入进程的时间片为最大时间片（这在一个进程时间片耗尽后但仍然是runable状态时非常重要，此时需要恢复时间片加入队列等待下次调度）。

  设置当前进程的任务队列为这个全局的任务队列以及维护队列中进程的数目。

  需要说明的是：优先队列中每一个节点就是一个 `skew_heap_entry_t` ，它只包含 3 个指针：parent / left / right。每个进程都有一个 `lab6_run_pool` 字段，这个字段就是该进程作为堆节点时的堆链接信息。

- stride算法的选择进程：选择任务堆的头节点，头节点是stride最小的。
- stride的时间中断处理：
  - 1.时钟中断触发时，如果当前进程的时间片有剩余就时间片减1；
  - 2.时钟中断触发时，如果当前进程的时间片耗尽，则将可调度设置为1，trap会执行调度算法选择stride最小的进程切换。

在 ucore 中，目前Stride是采用无符号的32位整数表示。则BigStride应该取多少，才能保证比较的正确性？

STRIDE_max－STRIDE_min≤PASS_max

队列中最大 stride 和最小 stride 之间的差值，永远不会超过最大步进值。

而最大PASS_max小于等于big_stride/2，所以有

PASS_max≤big_stride/2

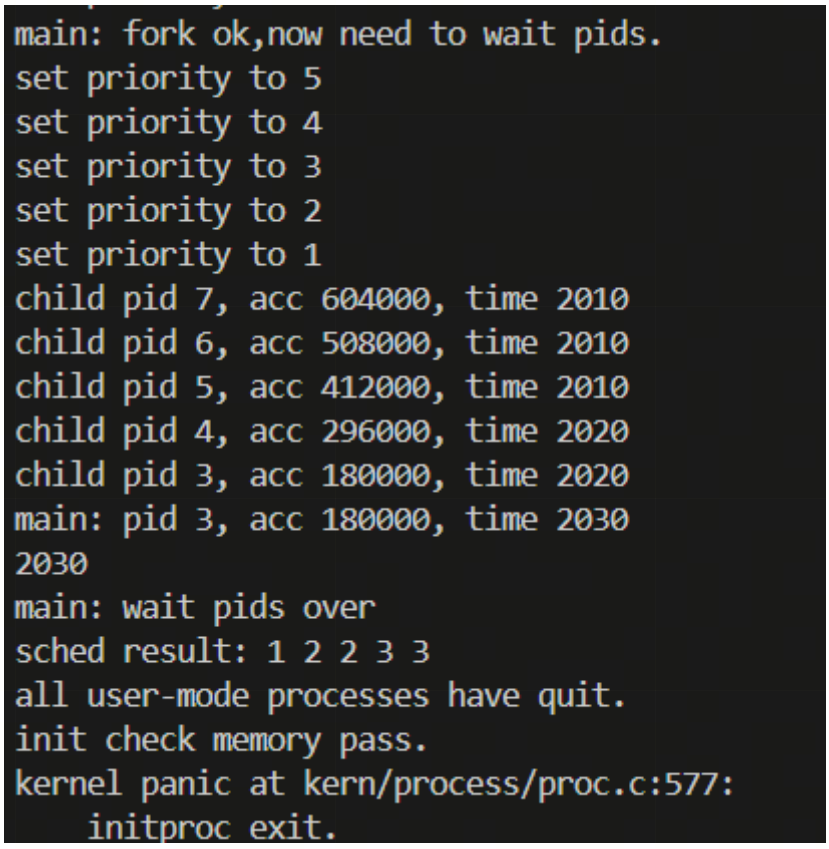
STRIDE_max－STRIDE_min≤big_stride/2

而STRIDE_max－STRIDE_min在32位无符号中最大值为2的31次方（比较大小时，会将结果转变为有符号数）

所以有big_stride/2＜2的31次方，即big_stride/2＜2的32次方，即小于等于2的31次方。

但是实验确定的是 `#define BIG_STRIDE (1 << 30)` ，猜测是因为不能把1左移到32位即符号位上，且此时2的30次方仍在合理范围内。

实验结果



2.4 扩展练习 Challenge 2：多种调度算法

在ucore上实现尽可能多的各种基本调度算法(FIFO, SJF,...)，并设计各种测试用例，能够定量地分析出各种调度算法在各种指标上的差异，说明调度算法的适用范围。

在现有框架基础上，实现了FIFO\PRIO\SJF调度算法

FIFO

FIFO 调度的基本思想是：**进程按进入就绪队列的先后顺序排队，调度时总是选择队首进程运行**，不会因为运行时间长短或优先级改变顺序。

`fifo_enqueue` 把进程插入到 `run_list` 的末端（尾插），`fifo_pick_next` 每次从队列头部取出第一个进程作为下一个执行者，因此保证了严格的先进先出顺序。

FIFO的优势是实现简单保证公平。

存在的问题是如果队首是一个运行时间极长的进程，后续短任务都会被拖延。同时如果交互式进程频繁让出CPU，每次都需要重新排队，那交互式进程可能长时间得不到执行。没有优先级，无法及时让出 CPU 给更紧急任务。（喜恶同因，根据不同场景而选择）

适用场景：适用于进程的处理时间相对均匀、资源争用较小的场景。

PRIO（静态优先级）

优先级调度的思想是：**在就绪队列中选择优先级最高的进程执行，优先级是静态的**。优先级来源于 `proc->lab6_priority`，并通过 `prio_value()` 把 0 修正为 1，避免出现“无效优先级”。在 `prio_pick_next` 中会遍历整个 `run_list`，寻找 `prio_value` 最大的进程作为 `best`，属于典型的**静态优先级调度**。采用**时间片递减，**如果当前进程的时间片降为 0，则设置 `need_resched = 1`。采取**抢占检测，**每个 tick 检查队列中是否存在优先级更高的进程，如果存在则立即触发 `need_resched = 1`。

优势是高优先级任务可更快获得 CPU，一旦出现更高优先级任务，当前任务会被迫让出 CPU（由 proc_tick 检测实现）。

缺点也显而易见，低优先级任务长期得不到执行被饿死，公平性差，同时调度计算pick_next的代价高，没有之前那些策略这么简单的选择，每次 `pick_next` 都遍历整个队列，复杂度 O(n)。（当然这里是可以改进的，比如学习stride算法，算用堆来维护任务队列）

适用场景：适用于需要根据进程的重要性进行调度的场景，如实时操作系统。

SJF（短作业优先 / Shortest Job First）

SJF 的核心思想是：**优先调度预计运行时间（CPU burst）最短的进程**，以降低平均等待时间。使用 `proc->lab6_burst` 表示 burst 估计值，如果为 0 则使用默认值 `SCHED_DEFAULT_BURST_TICKS`。在 `sjf_pick_next` 中遍历整个就绪队列，选择 burst 最小的进程运行。

这里实现的是非抢占SJF，即使在当前进程中有更小的调度时间的进程也不会抢占CPU,只能等运行线程让出CPU。

优点是SJF 在已知 burst 的情况下可最小化平均等待时间。

但是实系统无法准确知道进程未来执行时间，本实验中使用 `lab6_burst` 作为人工设定值。长任务可能饥饿，如果不断有短任务到来，长任务可能一直被推迟。目前没有抢占机制，若被选中的任务很长，仍可能阻塞后续任务。

适用场景：适用于处理时间可以预测且短作业较为常见的场景。

实验设计了两个测试文件，`sched_bench.c` 和 `sched_mix.c`，通过 `fork()` 出一组“行为可控”的子进程，让这些子进程以不同的方式占用 CPU / 让出 CPU，然后由父进程统计每个子进程的完成时间等数据，从而间接评估不同调度算法（FIFO / PRIO / SJF）的效果。

测试程序	构造 workload	触发调度方式	主要指标	关注点
sched_bench.c	5 个纯 CPU 任务，burst 不同	时间片/竞争调度	avg_turn, avg_wait, throughput	**总体性能**，特别是等待时间
sched_mix.c	CPU-bound + 频繁 yield 的交互任务混合	yield + 时间片 + 竞争	cpu/io 分组 avg_turn/ avg_wait + throughput	**交互性能、公平性、饥饿问题**

- 平均周转时间 avg_turn
- 平均等待时间 avg_wait
- 吞吐量 throughput，单位时间内完成的任务数量

第三部分

实验中涉及的重要知识

1.经过这几次实验，我发现一个很明显的现象就是处理回环。

- 在线程的id时要处理回环，通过设置的线程数目最大值是可用线程数目的两倍，和通过算法处理当回环从1开始分配后的重复问题。
- 在分配页的时候，内存空间是有限的，当分配页超过内存上限后会回到低地址，需要处理这种异常情况。
- 在本实验中处理stride累加溢出回环的问题。

2.本实验将read和write操作从上层到下层进行了拆分，用户可以读取任意长度数据，在syscall中将读取按照IOBUF_SIZE划分，提交到底层的抽象文件层和具体文件系统。在处理单个请求的时候，有需要划分成块首不对齐、完整块、块尾不满块。

3.设备（Everything is a file（万物皆文件））

目前实现了 stdin设备文件文件、stdout 设备文件、disk0设备。stdin设备就是键盘，stdout设备就是控制台终端的文本显示，而 disk0设备是承载 SFS 文件系统的磁盘设备。

操作系统是怎么在 open(path) 的时候，把 `"stdin:" / "stdout:" / "disk0:"` 识别成不同设备，并拿到对应的设备 inode 的？**注册 → 解析路径 → 查表返回 inode**。

1. 第一步：系统启动时把设备“注册”进 VFS 的设备表（vdev_list）

```
代码块
```

```
typedef struct {
    const char *devname;      // 设备名：stdin/stdout/disk0
    struct inode *devnode;    // 设备对应 inode（设备文件 inode）
    struct fs *fs;
    bool mountable;
    list_entry_t vdev_link;
} vfs_dev_t;

```

2. open(path) 时解析路径前缀，判断它是不是设备路径“

   根据之前的分析，我们直到在打开文件open的时候会调用vfs-open,vfs_open会调用vfs_lookup,在这里面首先会调用get_device,获取设备获取根目录，然后再调用vop_lookup调用具有文件系统或者设备的查找函数查找路径。

3. 第三步：在 vdev_list 查找 devname，找到对应 inode 并返回

   为什么拿到 inode 就够了？因为 inode->in_info 指向 device，这里我们再看看inode的结构体是怎么定义的。设备当作一个文件系统和直接的文件系统在这里做了区分，并且两者都有自己的 **inode_ops**(具体操作分发)。

代码块

```
struct inode {
    union {
        struct device __device_info;
        struct sfs_inode __sfs_inode_info;
    } in_info;
......
};
```

代码块

```
open("stdin:", O_RDONLY)
       |
       v
get_device() 解析出 devname="stdin"
       |
       v
vfs_get_root("stdin") 在 vdev_list 中查到 devnode(inode)
       |
       v
file->node = devnode
       |
       v
read(fd)
```

```
14    -> vop_read(file->node)
15         -> dev_read
16              -> dev = inode->in_info
17              -> dop_io(dev, iob, false)
18                   -> stdin_io(...)
19
```

VFS 层并不是每次都识别设备名，而是在 open 时通过路径决定 inode 类型；之后 read/write 等操作根据 inode 的 inode_ops 将请求分发到具体实现（设备驱动或具体文件系统）。

4.在真实系统中，文件的 inode 分为两类概念：

- **磁盘 inode（disk inode）**：
  存储在磁盘的固定位置（如 ext4 的 inode table），是持久化结构，记录文件的元数据（类型、权限、大小、时间戳、数据块指针等）。
  它通常不直接暴露给 VFS 层，而是由具体文件系统读取并解析。
- **内存 inode（in-memory inode，Linux 中是 `struct inode`）**：
  当文件被访问时，内核把磁盘 inode 的内容读入内存，并创建一个内存 inode 对象缓存这些信息；
  内存 inode 同时包含额外的运行时信息（引用计数、锁、操作函数表、缓存状态、dirty 标记等），因此比磁盘 inode 更丰富。

所以真实系统中是典型的关系：

> **磁盘 inode（持久化结构）** ⇄ 读取 / 缓存 ⇄ **内存 inode（运行时对象）**

5.Open流程

1.首先打开文件的系统调用 `open()`

2.进内核态系统调用sysfile_open()

3.把用户空间路径拷贝到内核空间。

4，首先调用的是file_open函数，它要给这个即将打开的文件分配一个file数据结构的变量，这个变量其实是当前进程的打开文件数组current->fs_struct->filemap[]中的一个空闲元素（即还没用于一个打开的文件），而这个元素的索引值就是最终要返回到用户进程并赋值给变量fd。

5.其中调用vfs_open，抽象层，来找到path对应的fd。

6.vfs_open通过vfs_lookup找到path对应文件的inode；调用vop_open函数打开文件。

7.vfs_lookup函数首先调用get_device函数，并进一步调用vfs_get_bootfs函数来找到根目录“/”对应的inode。这个inode就是位于vfs.c中的inode变量bootfs_node。这个变量在init_main函数（位于kern/process/proc.c）执行时获得了赋值。通过调用vop_lookup函数来查找到根目录“/”下对应文件sfs_filetest1的索引节点，如果找到就返回此索引节点。

8.在sfs_inode.c中的sfs_node_dirops变量定义了“.vop_lookup = sfs_lookup”8.sfs_lookup有三个参数：node，path，node_store。其中node是根目录“/”所对应的inode节点；path是文件sfs_filetest1的绝对路径/sfs_filetest1而node_store是经过查找获得的sfs_filetest1所对应的inode节点。

sfs_lookup函数以“/”为分割符，从左至右逐一分解path获得各个子目录和最终文件对应的inode节点。在本例中是调用sfs_lookup_once查找以根目录下的文件sfs_filetest1所对应的inode节点。当无法分解path后，就意味着找到了sfs_filetest1对应的inode节点，就可顺利返回了。

实验中未涉及但重要的知识

1.大文件的间接索引块、多级目录映射：

大文件涉及间接索引块，但该实验中直接从根目录下就可以找到对应的文件，没有涉及复杂的目录文件。也没有涉及**间接索引块。**

单独说一下简介索引块，ucore 里 SFS_NDIRECT 是 12，即直接索引的数据页大小为 12 * 4k = 48k，当使用一级间接数据块索引时，ucore 支持最大的文件大小为 12 * 4k + 1024 * 4k = 48k + 4m。

为什么是这样：因为如果启用了一级间接索引，一个块是4KB,每个指针是4byte，那这个间接索引可以指向4*1024/4个指针，就是1024个数据块，而这个文件就可以由直接数据块和间接数据块一起存储，就是4*4KB+1024*4KB

2.硬链接和软连接（实验中涉及硬链接但没有涉及软连接）

硬链接：给同一个inode增加一个新的目录项名字，即不同的路径指向同一个磁盘上的数据（inode），引用书会增加。

软连接：创建一个新的文件（有自己的 inode），里面存的是“目标路径字符串，访问它时系统会“跳转”到目标文件。软链接本身是一个独立文件，它有自己的 inode，它的数据内容是字符串 `"a.txt"` 或 `"/path/to/a.txt"`。访问软链接时，内核会读取软链接内容，得到路径，再对那个路径重新走一次 lookup。

一些疑惑和解答

inode 的设计目标是“一个文件（或目录）对应一个 inode，实验中涉及虚拟文件层inode、内存inode和磁盘inode

> **磁盘 inode**：在磁盘上，记录文件“元数据”和“数据块指针”，持久化存在。
> **内存 inode**：访问文件时加载到内核内存中的 inode 对象，包含磁盘 inode 信息 + 运行时状态。
> **VFS inode**：内存 inode 的“统一抽象接口”，用来屏蔽具体文件系统差异，内部通常挂着具体 FS 的私有 inode（例如 SFS 的 sfs_inode）。

所以：

✅ 磁盘 inode 是数据（on-disk record）
✅ 内存 inode 是对象（runtime object）
✅ VFS inode 是接口/抽象（interface & wrapper）

内存中的索引节点

代码块

```
/* inode for sfs */
struct sfs_inode {
    struct sfs_disk_inode *din;                     /* 磁盘节点 */
    uint32_t ino;                                   /* 节点索引，在该实验中一个节点占据一个block，但是真实的操作系统中会用唯一的变化而不是block编号 */
    bool dirty;                                     /* true if inode modified */
    int reclaim_count;                              /* 计数 */
    semaphore_t sem;                                /* 原子操作 */
    list_entry_t inode_link;                        /* entry for linked-list in sfs_fs */
    list_entry_t hash_link;                         /* entry for hash linked-list in sfs_fs */
};
```

磁盘中的索引节点

磁盘中的索引节点的信息被包含在了内存inode中，但是内存索引inode会包含更多的运行时信息，比如脏位

代码块

```
/* inode (on disk) */
struct sfs_disk_inode {
    uint32_t size;                                  /* size of the file (in bytes) */
    uint16_t type;                                  /* one of SYS_TYPE_* above */
    uint16_t nlinks;                                /* 硬连接数目 */
    uint32_t blocks;                                /* 块数 */
    uint32_t direct[SFS_NDIRECT];                   /* 直接相连的块数 */
    uint32_t indirect;                              /* 间接相连块，该块4096，可以存储4096/4个指针，就是1024个块来存放数据 */
//    uint32_t db_indirect;                           /* double indirect blocks */
//   unused
};
```

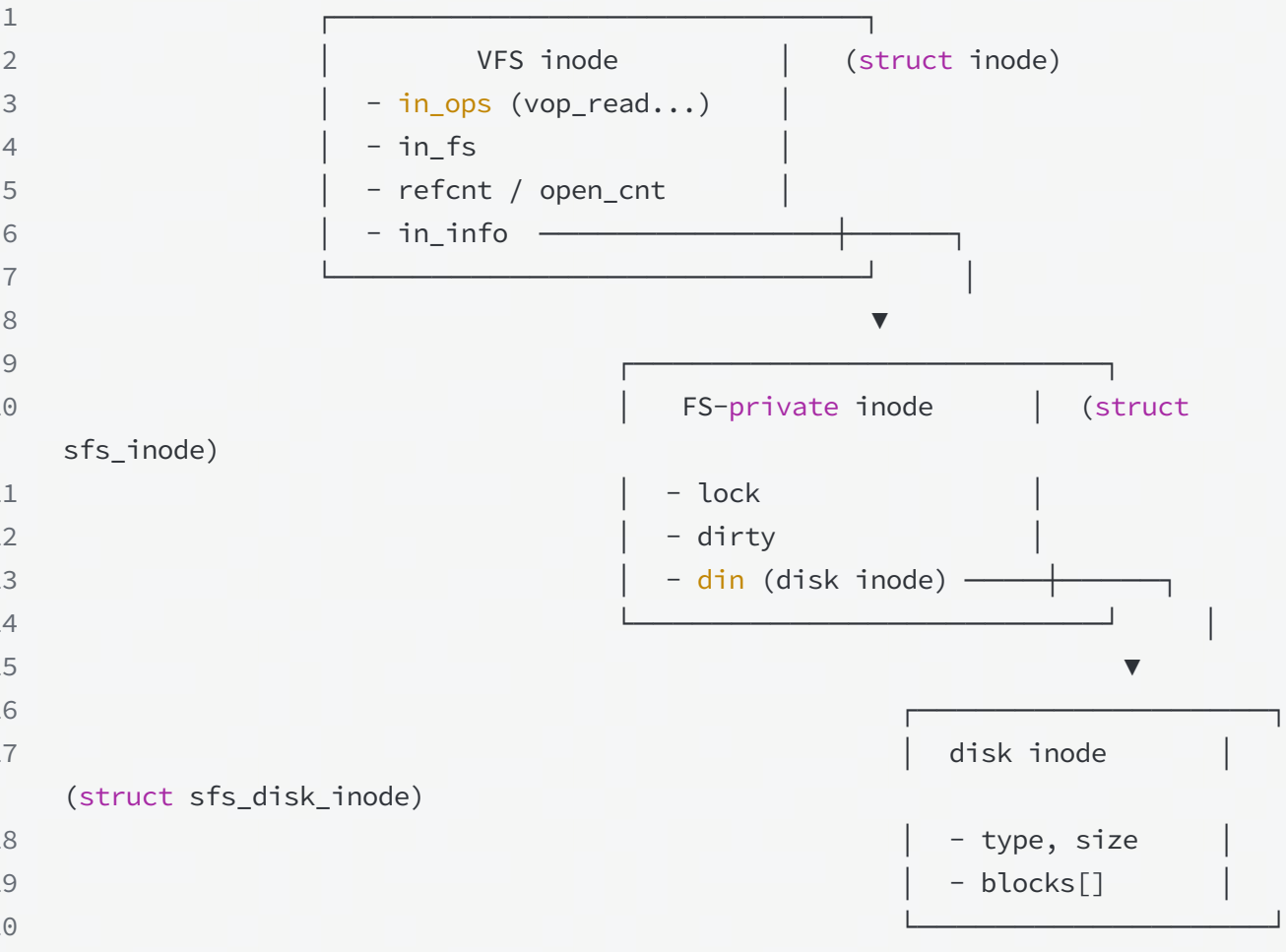
虚拟文件系统中的inode

该inode不仅封装了内存inode `sfs_inode` 或者磁盘inode `device` ，还加入了更丰富的程序运行时信息，比如引用计数器、打开计数器等。

代码块

```
// inode 抽象：设备与具体文件系统 inode 的统一封装
struct inode {
    union {
        struct device __device_info;
        struct sfs_inode __sfs_inode_info;
    } in_info;
    enum {
        inode_type_device_info = 0x1234,
        inode_type_sfs_inode_info,
    } in_type;
    int ref_count;
    int open_count;
    struct fs *in_fs;
    const struct inode_ops *in_ops;
};
```

代码块



```
21
```