



## ROS666 小型内核设计手册

所属高校	合肥工业大学
队伍名称	HFUT666
队伍编号	T202419359994630
比赛方向	OS 原理-方向 1（小型内核实现）
队伍成员	万立志、卢继鹏、高培骏
指导老师	田卫东、周红鹃



# 目录

- ROS666 概述
  - 项目概述
  - 本项目的意义与价值
  - 创新点
  - 整体架构
  - 小组在项目中的收获
- 操作系统启动
  - 1. BIOS 启动及操作系统加载
  - 2. 操作系统启动
  - 3. 操作系统初始化
    - 3.1 清空 BSS 段
    - 3.2 初始化陷入
    - 3.3 初始化内存管理
      - 3.3.1 初始化内核堆分配器
      - 3.3.2 初始化物理页帧分配器
      - 3.3.3 创建和激活内核地址空间
    - 3.4. 初始化文件系统
      - 3.4.1 初始化磁盘块设备
      - 3.4.2 打开文件系统
      - 3.4.3 获取文件系统根目录
    - 3.5. 初始化进程管理
    - 3.6. 建立时钟中断和定时器
    - 3.7. 运行用户进程
  - 4. 运行用户进程
- 系统陷入
  - 概念
  - 基本原理
    - 触发机制
    - 处理过程
    - 用途方面
  - 具体实现
    - 系统陷入触发
      - 系统调用相关函数调用
      - 系统调用指令底层转换
      - 与系统陷入处理程序的关联
      - 系统调用参数传递与陷入触发
    - 陷入处理程序执行
      - trap\_handler 函数入口及基本操作
      - 任务管理与陷入处理的关联

- 上下文保存与恢复
  - `__switch`函数
  - `schedule`函数
- 内核服务提供与返回结果
  - 系统调用处理与内核服务提供
  - 进程管理相关内核服务
- 存储管理模块
  - 存储管理模块概述
    - 模块简介
    - 模块的核心作用
    - 模块结构
    - 模块的设计目标
    - 模块在操作系统中的地位
  - 存储管理/地址
    - 虚拟地址 (VirtualAddress)
      - 核心方法
        - 与 `usize` 互相转换
        - 与 虚拟页号 `VirtualPageNumber` 互相转换
        - 计算相关地址
    - 物理地址 (PhysicalAddress)
      - 核心方法
        - 与 `usize` 互相转换
        - 与 物理页号 `PhysicalPageNumber` 互相转换
        - 计算相关地址
        - 内存操作
    - 虚拟页号 (VirtualPageNumber)
      - 物理页号 (PhysicalPageNumber)
    - 小结
  - 存储管理/动态内存分配 (Slab 堆分配器)
    - 简介
    - 动态内存分配的基础概念
      - 堆内存
      - 堆分配器
    - Slab 分配器
    - 堆分配器
      - 初始化
      - 分配
      - 释放
    - 互斥的堆内存分配器
    - 注册
    - 小结
  - 存储管理/物理页帧

- 简介
- 页帧的基本概念
  - 1. 什么是页帧?
  - 2. 页帧管理的核心目标
- 页帧管理的实现
  - 1. 栈帧分配器
  - 2. 初始化
    - 栈帧分配器的初始化
    - 帧分配和释放
  - FrameTracker
- 小结
- 存储管理/多级页表
  - 简介
  - 多级页表的基础概念
    - 1. 页表的作用
    - 2. SV39 分页机制
    - 3. 地址解析过程
    - 4. 页表项 (PTE)
  - 页表的核心数据结构
    - 页表项 (PTE)
      - 1. 创建新的页表项
      - 2. 提取物理页号 (PPN)
      - 3. 提取权限标志 (Flags)
    - 页表 (PageTable)
    - 页表的核心方法
      - 1. 创建新的页表
      - 2. 获取页表对应的 satp 寄存器值
      - 3. 映射虚拟地址到物理地址
      - 4. 取消映射
      - 5. 查找页表项
  - 内核态访问用户态地址
  - 小结
- 存储管理/地址空间
  - 简介
  - 基本概念
    - 结构
    - 组成部分
  - 核心数据结构
    - 逻辑段 (MapArea)
    - 地址空间 (MemorySet)
  - 核心功能
    - 1. 创建地址空间

- 初始化空地址空间
  - 创建内核地址空间
  - 创建用户地址空间
  - 映射跳板页
  - 2. 克隆地址空间
  - 3. 添加逻辑段
  - 4. 移除逻辑段
  - 5. 激活地址空间
  - 小结
- 存储管理模块总结
  - 模块的重要性
  - 核心设计特点
  - 结束语
- 进程管理模块
  - 进程管理模块概述
    - 进程管理模块的定义与作用
    - 主要功能组成部分
      - 进程创建与销毁
      - 进程调度
      - 进程状态转换
      - 进程间通信（IPC）
    - 与其他模块的交互关系
      - 与内存管理模块的交互：
      - 与文件系统模块的交互：
    - 进程管理流程
      - 初始化
      - 任务的创建：
      - 任务的执行
      - 任务的调度
  - 进程控制块
    - 进程控制块的原理
      - 定义与作用
      - 信息存储内容
    - 进程控制块的实现
      - 进程控制块数据结构ProcessControlBlock
      - 进程控制块方法实现
  - 进程状态切换
    - 简介
    - 进程状态数据结构
      - 创建状态
      - 就绪状态
      - 运行状态
      - 阻塞状态
      - 终止状态

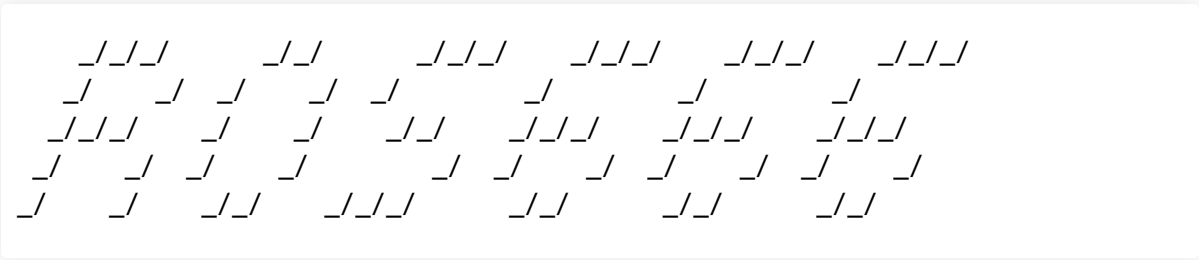
- 进程状态切换描述
- 进程状态切换的触发因素
  - 任务执行
  - 系统调用
  - I/O 操作
  - 中断
  - 进程调度策略
- 进程状态切换的实现机制
  - 通过操作系统内核代码实现
  - 涉及的队列操作
  - 上下文保存与恢复
- 进程调度
  - 进程调度的定义与目的
    - 定义
    - 目的
  - 进程调度的触发时机
    - 进程状态变化
    - 新进程创建
    - 中断处理完成
  - 进程调度的主要策略
    - 先来先服务 (FCFS)
    - 短作业优先 (SJF)
    - 时间片轮转 (RR)
    - 优先级调度
    - 先来先服务+时间片轮转
  - 进程调度的实现细节
    - 进程创建与就绪队列初始化
      - 进程创建流程
      - 就绪队列管理
    - 进程调度循环
      - 获取任务
      - 设置任务状态与上下文
      - 上下文切换与执行
    - 进程切换与状态更新 (时间片轮转机制)
      - 时间片耗尽检测
      - 进程状态更新与队列操作
    - 进程结束处理
      - 进程终止触发
      - 资源回收与状态变更
- 文件系统模块
  - 文件系统模块概述
    - 简介
    - 核心设计目标

- 模块结构
- 总体功能概述
- 总结
- 文件系统/块设备接口及缓冲层
  - 简介
  - 块设备接口
  - 块设备缓冲层
    - 块缓存管理器
    - 块缓存结构
    - 块缓存层接口
  - 总结
- 文件系统/磁盘布局与索引节点
  - 简介
  - 磁盘布局
  - 超级块（Super Block）
  - 位示图（Bitmap）
    - 位示图结构
    - 位示图操作
  - 磁盘索引节点（DiskInode）
    - 索引节点访问
    - 索引节点扩充和清空
      - 扩充
      - 清空
  - 总结
- 文件系统/文件系统抽象
  - 物理文件系统
  - 文件系统操作
    - 创建文件系统
    - 打开文件系统
    - 分配和释放 inode
    - 分配和释放数据块
    - 获取 inode 位置
    - 获取根目录 inode
  - 总结
- 文件系统/文件和目录管理
  - 内存 Inode
    - 内存 Inode 方法
      - 创建内存 Inode
      - 读写内存 Inode
      - 增加和清除文件大小
    - 查找文件或目录
    - 列出目录内容
    - 获取元数据



- 文件目录管理
    - OS Inode
    - 文件读写
    - 操作系统文件访问接口
  - 总结
- 系统调用
  - 系统调用的原理
    - 用户态与内核态的隔离
    - 系统调用与函数调用的区别
    - 系统调用的触发机制
    - 系统调用处理流程
    - 内核态到用户态的返回机制
    - 系统调用的安全性和可靠性保障
  - 系统调用的过程
  - 系统调用的实现
    - 系统调用号定义与分配
    - 系统调用接口函数实现
      - sys\_openat:
      - sys\_close
      - sys\_read
      - sys\_write
      - sys\_exit
      - sys\_yield
      - sys\_get\_time\_of\_day
      - sys\_shutdown
      - sys\_clone
      - sys\_execve
      - sys\_wait4
  - 系统调用的封装
- 功能测试
  - 1. 内核启动和关闭
  - 2. 用户态程序和命令行参数解析
  - 3. 进程调度
  - 4. 文件系统

# ROS666 概述



## 项目概述

本项目是合肥工业大学 HFUT666 团队为 [2024年全国大学生计算机系统能力大赛-操作系统设计赛\(华东区域赛\)-OS原理赛道](#) 比赛开发的项目。该项目从零开始，使用 Rust 编程语言，在 RISC-V 架构上实现了一个简单的类 Unix 操作系统内核。

项目团队成员包括：

- 万立志 @MagicalMagic | [GitHub](#) | [gitlab.eduxiji.net](#)
- 高培骏 @FengSheng0804 | [GitHub](#) | [gitlab.eduxiji.net](#)
- 卢继鹏 @Gerrnperl | [GitHub](#) | [gitlab.eduxiji.net](#)。

项目主要参考了 [rCore-Tutorial-Book-v3](#) 教程; 块设备驱动、部分测试代码以及部分内核实现参考了 [rCore-Tutorial-v3](#)、[slab\\_allocator - Slab allocator for no\\_std systems](#). 和 [virtio-drivers](#) 等开源项目的代码; 项目的实现也离不开 [Rust 语言](#) 及其生态、[rCore OS 社区](#) 生态的支持。在此，我们对这些开源项目 and 社区表示感谢。

OSKernel2024-ROS666项目实现了操作系统的核心功能，包括裸机内核启动、标准输入输出、陷入与陷入返回机制、动态内存分配（使用Slab Allocator）、分页机制、时钟中断、进程管理与调度、文件系统支持、用户态程序执行以及系统调用等。

在这些内核功能之上，项目完成了 stdio 和文件系统的读写、打开、关闭操作，以及进程的分叉（clone/fork）、执行（execve）、等待（wait）、退出（exit）、让权休眠（sched\_yield）等进程管理功能，并实现了获取时间（gettime）和系统关闭（shutdown）等系统调用。

在开发过程中，团队共进行了260余次提交，累计编写了约8500行代码。项目通过多个版本的迭代，逐步完成了从基本的裸机程序 Hello World 到完善的进程管理、文件系统支持以及用户程序执行等核心任务。

OSKernel2024-ROS666项目不仅是一次技术挑战和实践机会，也是团队成员在操作系统原理、Rust编程以及团队协作等方面的一次全面提升。通过参与这个项目，团队成员深入理解了操作系统的内部机制和工作原理，积累了宝贵的开发经验和知识。

我们深知项目仍有许多不足之处，离一个 *比较完整的操作系统内核* 还有很大的差距，但我们也为项目的成果感到自豪。在未来，我们将继续完善项目，提高代码质量，增加更多功能，以期更好地服务于教育和技术交流。

## 本项目的意义与价值

操作系统是计算机系统中最重要基础软件之一，是连接硬件与软件的桥梁。它不仅决定了计算机资源的管理效率，还直接影响系统的安全性、稳定性与用户体验。

在计算机教育中，操作系统课程具有举足轻重的地位，如何通过实践帮助学生理解复杂的内核机制、从理论到实现掌握系统设计的精髓，是教学中的一大挑战。

本项目以 Rust 编程语言为基础，采用模块化设计和精细化实现，开发了一个小型内核系统。

通过实现裸机启动、动态内存分配、分页机制、进程管理与调度、文件系统、用户态程序及系统调用等核心模块，我们为教学和研究提供了一个兼具现代性与安全性的操作系统内核项目。

我们的项目在操作系统教学中的所能起到的帮助：

### 1. 弥合理论与实践的鸿沟

操作系统课程中涉及的知识点繁多，包括内存管理、进程调度、系统调用等等，学生在课堂上学习这些理论时常感到抽象和难以理解。

而本项目通过从零开发一个小型内核，提供了完整的代码实现与模块化设计，让学生能够通过动手实践，直观感受到操作系统各模块如何协同工作，从而加深对理论的理解。

例如，学生可以在代码中观察分页机制如何实现虚拟内存到物理内存的映射，或者亲自调试进程调度算法，了解时间片轮转的实际效果。

这种从理论到代码再到实际运行的完整学习路径，极大地提升了教学的效果。

### 2. 覆盖核心模块，完整展现操作系统的设计思想

本项目已经实现了从裸机内核启动到系统调用的全链路功能，覆盖了操作系统中最核心的模块：

1. 裸机启动：帮助学生了解从硬件启动到操作系统加载的过程；
2. 动态内存分配：通过 Slab Allocator 实现高效内存管理，展示内存分配器的核心思想；
3. 分页机制：实现虚拟内存管理，帮助学生理解内存隔离与权限控制的重要性；
4. 进程管理与调度：通过时间片轮转与进程状态切换，完整展现任务调度的核心逻辑；
5. 文件系统与系统调用：为用户态程序提供标准输入输出、文件操作及进程控制等系统接口，展现操作系统对上层应用的支持。

这些功能的实现不仅涵盖了操作系统课程的核心内容，还为学生提供了一个可运行的内核系统，真正做到理论与实践的深度融合。

### 3. 提升编程能力与系统思维

在操作系统教学中，学生经常面临如何设计复杂系统的问题。通过本项目，学生能够学习到模块化设计的思想，每个模块（如内存管理、进程调度等）既独立又协作，便于理解和维护。此外，学生还可以学习到如何使用现代化工具链（如 Rust 编译器和裸机开发工具）进行系统级开发，这种实践能力对于未来从事嵌入式系统开发、操作系统研究或高性能计算开发具有重要意义。

为什么选用 Rust 语言？

在本项目中，我们选择 Rust 作为开发语言，主要基于其独特的优势，特别是面向系统安全性和开发效率的突出特性。

操作系统的核心模块（如动态内存分配、进程调度等）需要对内存进行频繁的管理与操作。传统的 C 语言虽然高效，但容易因为指针错误、空指针访问或数据竞争导致系统崩溃甚至安全漏洞。而 Rust 的所有权机制、借用检查和编译时内存安全性保障，可以从根本上杜绝这类问题。例如，在实现动态内存分配（Slab Allocator）时，Rust 的借用检查器确保了多线程场景下不会发生内存访问冲突，提高了内存分配器的安全性与鲁棒性。

此外 Rust 提供了与 C/C++ 相当的性能，同时通过零开销抽象，使得复杂数据结构的实现既高效又易于维护。在本项目中，我们在实现分页机制和进程调度时充分利用了 Rust 的高性能特性，例如通过枚举和模式匹配高效处理进程状态切换。

Rust 还有拥有成熟的裸机开发生态（如 core 库、alloc 库）和完善的工具链支持（如 cargo 构建系统），这使得开发裸机内核的过程更加高效。

此外 Rust 提供了明确的模块化体系结构，通过 crate 和模块的组合，可以轻松实现内核的模块化设计。我们在项目中将内核功能划分为多个模块（如内核、用户程序库、用户程序、文件系统、堆内存分配器等），提高了代码的可读性和可维护性，还方便了针对单一模块进行学习与调试。

值得一提的是，这种模块化设计也为后续扩展内核功能提供了良好的基础。

最后本项目通过基于 Rust 的小型内核开发，探索了如何以现代化、安全、高效的方式实现操作系统核心模块。这不仅为操作系统教学提供了新思路，也为未来其他同学们在此项目基础上进行创新研究提供了一个完善的平台。我们相信该项目经过后续我们自身以及其他同学们的不断完善和扩展，项目将在推动操作系统教学发展以及小型内核实现和创新的道路上继续发挥更大的作用。

## 创新点

- **基于 Rust 语言的现代化内核设计**

本项目完全使用 Rust 语言开发，充分利用 Rust 的内存安全性、高性能和模块化设计特性，实现了一个现代化的小型内核。Rust 的所有权机制、借用检查和编译时内存安全性保障，有效避免了内存错误和数据竞争问题，提高了内核的安全性和稳定性。通过 Rust 的模块化设计和 crate 生态，我们实现了内核的模块化设计，提高了代码的可读性和可维护性。

- **Slab Allocator 动态内存分配器**

本项目实现了一个高效的 Slab Allocator 动态内存分配器，用于管理内核的动态内存分配。Slab Allocator 通过预先分配一定数量的固定大小的内存块，然后根据需要 will 内存块分配给请求者，在减少内存内碎片的同时，提高了内存分配的效率。

通过 Slab Allocator，我们实现了内核的动态内存分配功能，为内核的进程管理和文件系统提供了基础支持。

- **用户程序与函数库和构建流分离**

本项目将用户程序与函数库和构建流分离，用户程序通过独立的函数库调用系统调用，实现了用户程序与内核的分离。函数库已实现的函数遵循 gcc 和 Linux 程序库的结构，系统调用也遵循 Linux 系统调用的规范，使得用户程序的编写更加方便。

用户程序构建依赖于在用户程序构建时，指定链接脚本、编译选项等。用户程序可以以此为构建依赖，在 build.rs 构建脚本中调用此库提供的函数，配置用户程序的构建。如此可以分离用户程序构建依赖，使得用户程序构建更加简洁，并提高用户程序的独立性。

- 系统调用接口模块

内核和用户程序间的系统调用使用枚举进行封装，并定义在独立的模块中，实现了系统调用的统一管理和调用，避免了系统调用重复定义和混乱的问题。

- 非阻塞 Rust SBI IO的阻塞式封装

Rust SBI IO 接口是一个非阻塞的 IO 接口，在 IO 缓冲无数据时会立即返回，这对于内核的 IO 操作是不方便的。

本项目实现了非阻塞 Rust SBI IO的阻塞式封装，通过封装 Rust SBI IO 接口，实现了对 Rust SBI IO 的阻塞式调用，使得内核的 IO 操作更加方便。对于用户程序，阻塞是让权的，在等待时，用户程序会让出 CPU，等待 IO 完成后再继续执行。

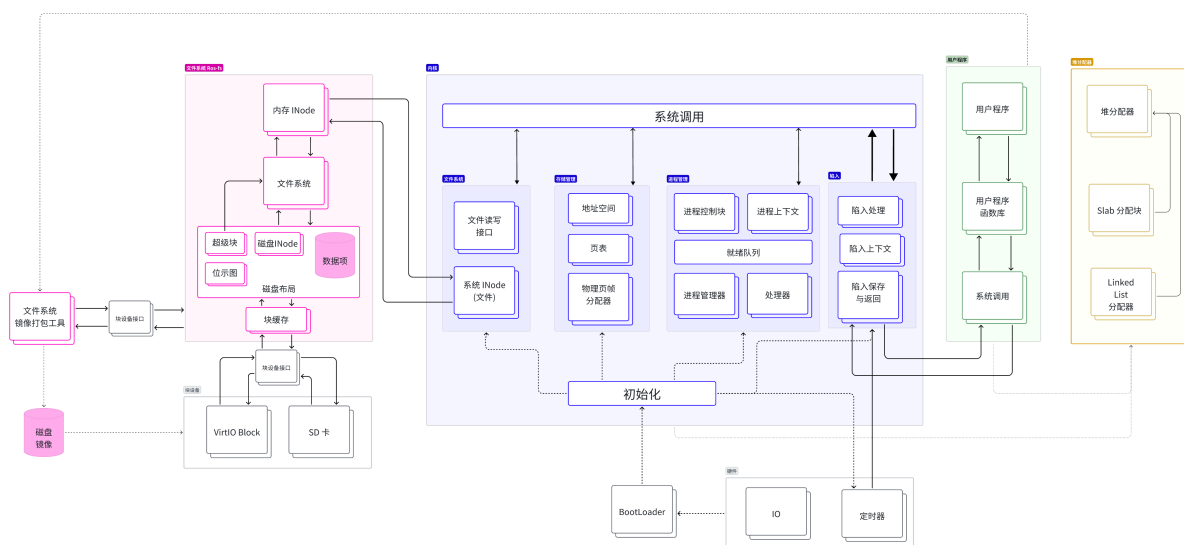
- 充分利用 RAII 机制实现资源管理

本项目充分利用 Rust 的 RAII 机制，通过实现 Drop trait，实现了对资源的自动释放和管理，避免了资源泄漏和内存泄漏的问题。

- 完善的构建与调试配置

项目提供了完善的构建与调试配置，包括 Makefile 构建脚本，并且充分结合 VSCode 提供的任务及调试配置，覆盖了从环境检查、编译、打包、运行到调试的全流程，使得开发调试更加方便。

## 整体架构



系统主要模块包括公共模块、文件系统模块、文件系统镜像打包工具、动态内存分配器、内核源码、用户程序、用户程序函数库和用户程序构建依赖。

```

.
├── .cargo                                # Cargo 配置
│   └── config.toml
├── .github                              # GitHub Actions 配置
│   └── workflows
│       └── cargo-doc.yml                # 项目 cargo doc 代码文档的持续集成配置
├── .vscode                             # VSCode 配置
│   ├── extensions.json                 # VSCode 集成调试所需的插件
│   ├── launch.json                    # 调试配置
│   ├── settings.json                  # VSCode 设置
│   └── tasks.json                     # 任务配置
├── bootloader                          # RustSBI 启动引导程序
│   └── rustsbi-qemu-release
├── build.rs                            # 内核构建脚本
├── common                              # 适用于内核和用户程序的公共模块，如系统调用接
└── □
    ├── Cargo.toml
    ├── README.md
    └── src
        ├── lib.rs
        └── syscall.rs
├── docs                                # 项目文档
│   └── ...
├── LICENSE                             # 开源许可证 (GPL-3.0)
├── Makefile                            # 项目构建 Makefile 脚本
├── README.md
├── ros-fs                              # 文件系统模块
│   ├── Cargo.toml
│   ├── README.md
│   └── src
│       ├── bitmap.rs                  # 位示图管理
│       ├── block_cache.rs             # 块缓存管理
│       ├── block_dev.rs               # 块设备接口
│       ├── fs.rs                      # 文件系统抽象层
│       ├── layout                     # 磁盘布局层
│       │   ├── dir_entry.rs           # 磁盘布局/目录项
│       │   ├── disk_inode.rs         # 磁盘布局/索引节点
│       │   ├── mod.rs                 # 磁盘布局模块入口
│       │   └── super_block.rs         # 磁盘布局/超级块
│       ├── lib.rs                     # 文件系统模块入口
│       └── virt_fs.rs                 # 文件和目录操作 (内存 Inode 部分)
├── ros-fs-fuse                          # 文件系统镜像打包工具
│   ├── Cargo.toml
│   ├── README.md
│   └── src
│       ├── block_file.rs
│       └── main.rs
├── scripts                              # 项目脚本
│   └── check-dev-requirements.sh       # 检查开发环境脚本
├── slab_allocator                       # Slab Allocator 动态内存分配器
│   ├── Cargo.toml
│   ├── README.md
│   └── src
│       ├── lib.rs                     # Slab Allocator 模块入口及堆内存分配器实现
│       └── slab.rs                    # Slab Allocator 实现

```

src	# 内核源码
drivers	# 设备驱动
block	# 块设备驱动
mod.rs	# 块设备驱动模块入口
sdcard.rs	# SD 卡驱动
virtio_block.rs	# VirtIO 块设备驱动
mod.rs	# 设备驱动模块入口
entry.asm	# 内核入口汇编代码
fs	# 文件系统（文件和目录操作，系统 Inode 部分）
inode.rs	# 系统 Inode，提供读写接口
mod.rs	# 文件系统模块入口
stdio.rs	# 基于文件描述符的标准输入输出
io	# 输入输出
log.rs	# 日志输出
mod.rs	# 输入输出模块入口
stdio.rs	# SBI 标准输入输出（Console 相关IO）
language_item.rs	# Rust 语言项
linker.ld	# 内核链接脚本
main.rs	# 内核入口及初始化
mm	# 内存管理
address.rs	# 地址抽象
frame_allocator.rs	# 物理页帧分配器
heap_allocator.rs	# 堆内存分配器（Slab Allocator 实例）
init.rs	# 内存管理初始化
memory_set.rs	# 逻辑段和地址空间
mod.rs	# 内存管理模块入口
page_table.rs	# 页表管理
sbi.rs	# RustSBI 接口封装
syscall.rs	# 系统调用接口实现
task	# 进程管理
context.rs	# 进程上下文
manager.rs	# 进程管理器
mod.rs	# 进程管理模块入口
pid.rs	# 进程 PID 分配回收管理
processor.rs	# 进程处理器
stack.rs	# 进程内核栈
switch.asm	# 进程切换汇编代码
switch.rs	# 进程切换实现
task.rs	# 进程控制块
timer.rs	# 时钟中断相关
trap	# 陷入与陷入返回
context.rs	# 陷入上下文
handler.rs	# 陷入处理
init.rs	# 陷入初始化
mod.rs	# 陷入模块入口
trap.asm	# 陷入与陷入返回汇编代码
utils	# 工具模块
macros.rs	# 宏定义，包括用于引用外部定义符号的宏
mod.rs	# 工具模块入口
safety.rs	# 安全性封装相关
user	# 用户程序
build.rs	# 用户程序构建脚本（导入 user-build 执行）
Cargo.toml	
README.md	
src	# 用户程序源码

```

├── bye/bin/main.rs      # bye, 测试退出、时间获取、时间控制
├── calc/bin/main.rs    # calc, 简单的四则运算计算器
├── cat/bin/main.rs     # cat, 读取文件
├── echo/bin/main.rs    # echo, 测试标准输入输出和命令行参数解析
├── fstest/bin/main.rs  # fstest, 测试文件系统读写
├── hello/bin/main.rs   # hello, 和 bye 一致
├── initproc/bin/main.rs # initproc, 系统第一个用户进程, 主要是打开 she
ll
├── ls/bin/main.rs      # ls, 文件系统目录读取
├── rrtest/bin/main.rs  # rrtest, 测试时间片轮转进程调度, 并发执行 hel
lo 和 bye
├── sh/bin/main.rs      # sh, 简单的 shell, 参数解析和程序执行
├── shutdown/bin/main.rs # shutdown, 关机
├── touchwith/bin/main.rs # touchwith, 创建文件, 并写入内容
├── user-build          # 用户程序构建流依赖
├── Cargo.toml
├── README.md
├── src
├── lib.rs              # 用户程序构建流依赖模块入口
├── linker.template.ld  # 用户程序链接脚本模板
├── user-lib            # 用户程序函数库
├── Cargo.toml
├── README.md
├── src                # 用户程序函数库源码
├── fcntl.rs           # 文件控制
├── heap_allocator.rs  # 用户程序堆内存分配器 (Slab Allocator 实例)
├── language_item.rs   # Rust 语言项
├── lib.rs              # 用户程序函数库模块入口
├── sched.rs           # 调度相关
├── stdio.rs           # 标准输入输出
├── sys                # 系统相关
├── mod.rs
├── time.rs            # 时间获取
├── wait.rs            # 进程等待
├── syscall.rs         # 系统调用, 访管指令封装
├── unistd.rs          # 定义在 POSIX 标准中并且实现了的系统调用封装

```

## 小组在项目中的收获

我们项目组的三个成员通过开发本项目，每个人收获到了很多，包括技术能力的提升、对操作系统的深入理解、成就感与职业自信心的提升等。

首先就是技术能力的全面提升。

通过本项目，我们从零开始构建了一个小型操作系统内核，深刻理解了操作系统的核心原理，并掌握了如何将复杂的理论知识转化为实际代码。

项目中全面使用Rust语言开发，这不仅让我们熟悉了Rust的语法和特性，也让我们对使用Rust语法开发其他项目有了更大的信心。

其次就是对操作系统的理解更加深入。

我们共同完成了从裸机启动到用户态程序运行的全链路开发，全面掌握了操作系统从硬件到软件的关键交互机制。



通过本项目，我们不仅理解了操作系统的基本原理，也深刻认识到其在计算机体系结构中的核心地位。

无论是Slab分配器的实现、分页机制的映射逻辑，还是进程调度的状态切换，我们都深入研究了实现细节并不断优化。因此动手实践搭配课堂的学习让我们对操作系统各个模块的认识不断加深。

最后是成就感与职业自信心的提升。

从最初的构想到完整内核系统的实现，我们经历了从零到一的创造过程。这种从无到有的开发经历让我们获得了极大的成就感，同时也增强了我们在系统开发领域的自信心。

在内核开发过程中经常会遇到各种问题，有一些问题甚至是几近于无法调试的，在实现进程切换时错误的上下文和切换返回地址；在实现分页机制时错误的页表映射和非法的内存访问；在实现文件系统时非法的 DMA 映射和错误的驱动程序实现等等。

每一个 Bug 似乎都是不可逾越的障碍，在 GDB 也无法有效调试的情况下，这些问题曾让我们感到无比的挫败。然而，在不断的尝试和调试中，我们最终找到了解决问题的方法，这种突破困难的经历让我们更加坚信自己的能力。

我们每个人相信，这次合作开发的经历将成为未来职业发展的重要基石，无论是在操作系统领域的进一步研究，还是在实际工程项目中，我们都将更加游刃有余。

未来，我们还将继续在此项目基础上不断优化已有的模块同时去实现其他还没有实现的功能。我们不会因为比赛的结束而结束我们项目的开发，我们的目标永远是实现一个更加完善、更加完美的小型内核。

# 操作系统启动

## 1. BIOS 启动及操作系统加载

BIOS 是计算机的基本输入输出系统（Basic Input/Output System），负责初始化硬件和加载操作系统内核。在 RISC-V 架构中，BIOS 位于机器模式（M-mode）下，是系统启动的第一部分。

系统启动主要分为三个步骤：

1. Stage 1, 由固化在硬件或者 QEMU 中的代码负责，主要是初始化 CPU 和内存。在此阶段，Stage 1 程序会被加载到内存的固定位置，程序计数器会被初始化为固定的地址，在 QEMU 中为 0x1000，然后跳转到这个地址执行。Stage 1 程序的主要工作是加载 Stage 2 程序，并跳转到 Stage 2 程序的入口地址。
2. stage 2, 由 Stage 1 加载并跳转到的程序，主要是加载操作系统内核。这一阶段由 Rust SBI 作为 Boot Loader 实现，在 Boot Loader 启动后，跳转到固定位置 0x80200000，至此，系统控制权将转交给操作系统内核。
3. 操作系统内核，由 Stage 2 加载并跳转到的程序，是操作系统的核心部分，负责管理硬件资源、进程调度、内存管理等。

在编译操作系统时，我们可以通过链接脚本指定操作系统内核的入口地址：

```
ENTRY(_start)
BASE_ADDRESS = 0x80200000;

SECTIONS
{
    . = BASE_ADDRESS;
    .text : {
        *(.text.entry)
        /* ... */
    }
    /* ... */
}
```

在这里，我们使用 `. = BASE_ADDRESS;` 将内核的入口地址设置为 `0x80200000`，并且将操作系统入口 `_start` 设置为入口地址。这样在加载内核时，可以直接跳转到这个地址执行。

## 2. 操作系统启动

在操作系统内核启动时，首先需要建立栈指针 `sp`，将其指向内核栈区域的底部，如此，内核可以使用栈来保存函数调用的上下文信息，实现函数调用返回。

```
.section .bss.stack
.align 12

.global bottom_stack_bottom
bottom_stack_bottom:
    .space 4096 * 16

.global bottom_stack_top
bottom_stack_top:
```

在这里，我们在 `.bss` 段中定义了一个内核栈区域，大小为 `4096 * 16` 字节，这个区域的边界使用 `bottom_stack_bottom` 和 `bottom_stack_top` 标记。

在 `_start` 函数中，使用 `la sp, bottom_stack_top` 将栈指针 `sp` 设置为栈区域的顶部，至此，内核栈的建立完成，可以进行函数调用和返回。

```
.section .text.entry
.global _start
_start:
    la sp, bottom_stack_top
    call _kernel_entry
```

在 `_start` 函数最后，调用 `_kernel_entry` 函数，进入操作系统内核的入口函数。

## 3. 操作系统初始化

### 3.1 清空 BSS 段

BSS 段是未初始化的全局变量和静态变量所在的内存区域。在内核启动时，需要将 BSS 段清零，以确保这些变量的初始值为 0。

- .text: 代码段，存放程序的代码。
- .data: 数据段，存放程序的全局变量和静态变量。
- .rodata: 只读数据段，存放只读数据。
- .bss: 未初始化数据段，存放未初始化的全局变量和静态变量。

在链接脚本中，我们可以在 .bss 段的定义中定义边界指针，如下：

```
__data_end = .;
__bss_start_stack = .;
.bss : {
    *(.bss.stack)
    __bss_start = .;
    *(.bss .bss.*)
    *(.sbss .sbss.*)
}

. = ALIGN(4K);
__bss_end = .;
```

在这里，我们定义了 \_\_bss\_start 和 \_\_bss\_end 两个标记，分别表示 BSS 段的起始和结束地址。

这样，我们即可通过 extern 引用这两个标记，然后在 \_kernel\_entry 函数中将 BSS 段清零：

```
fn clear_bss() {
    unsafe extern "C" {
        fn __bss_start();
        fn __bss_end();
    }

    for i in __bss_start as usize..__bss_end as usize {
        unsafe {
            core::ptr::write_volatile(i as *mut u8, 0);
        }
    }
}
```

core::ptr::write\_volatile 函数会在指定地址进行 volatile 写操作，确保编译器不会对这个操作进行优化，从而保证 BSS 段的清零操作不会被优化掉。

### 3.2 初始化陷入

陷入是由软件主动触发的异常，在 CPU 内部触发。当用户态程序执行访管指令或者指令执行出现异常时，会触发陷入。

陷入主要包括:

1. 指令未对齐
2. 指令异常
3. 非法指令
4. 断点
5. 加载未对齐
6. 加载异常
7. 存储未对齐
8. 存储异常
9. 用户态环境调用 (系统调用 `ecall`)
10. 管理员环境调用
11. 机器环境调用
12. 指令页错误
13. 加载页错误
14. 存储页错误

通过在 `stvec` 寄存器中设置陷入处理函数的地址, 可以在陷入发生时跳转到指定的处理函数。

```
fn set_kernel_trap_entry() {  
    unsafe {  
        stvec::write(trap_from_kernel as usize, TrapMode::Direct);  
    }  
}
```

在这里, 我们通过 `stvec::write` 函数将陷入处理函数的地址设置为 `trap_from_kernel` 函数的地址, 并且设置陷入模式为 `TrapMode::Direct`。由于内核级陷入没有恢复的必要, 因此我们直接在处理函数中进行 `panic` 处理。

## 3.3 初始化内存管理

内存管理是操作系统的核心功能之一, 负责管理系统的物理内存, 为用户程序提供内存分配和释放的功能。初始化内存管理时, 需要完成以下工作:

1. 初始化内核堆分配器
2. 物理页帧分配器
3. 激活内核地址空间

### 3.3.1 初始化内核堆分配器

内核堆分配器是用于管理内核堆区, 为内核提供动态内存分配和释放的功能。在启用堆分配器之后, 即可使用 Rust 的 `alloc` 库进行内存分配和释放。

首先, 定义一个全局的静态变量 `HEAP` 作为堆区:

```

/// 内核堆大小
pub const KERNEL_HEAP_SIZE: usize = 0x60_0000;
/// 内核堆区
static mut HEAP: [u8; KERNEL_HEAP_SIZE] = [0; KERNEL_HEAP_SIZE];

```

使用 `static mut` 可以将 HEAP 堆区放到 `.bss` 段中，这样在内核启动时，可以将 BSS 段清零，同时也会将 HEAP 堆区清零。

使用 `#[global_allocator]` 定义一个全局的堆分配器：

```

#[global_allocator]
static HEAP_ALLOCATOR: LockedHeap = LockedHeap::empty();

```

设置 Rust 语言项 `alloc_error_handler`，用于处理内存分配失败的情况：

```

#[alloc_error_handler]
fn alloc_error_handler(layout: core::alloc::Layout) -> ! {
    panic!("Heap allocation error: {:?}", layout)
}

```

在分配失败，例如内存不足 `Out of Memory` 时，会调用 `alloc_error_handler` 函数进行处理。

最后，初始化 `HEAP_ALLOCATOR` 堆分配器：

```

HEAP_ALLOCATOR
    .lock()
    .init(HEAP.as_ptr() as usize, KERNEL_HEAP_SIZE);

```

### 3.3.2 初始化物理页帧分配器

操作系统在建立页表之前，需要能够管理所有物理内存的分配和释放。

内核代码结束后（即 BSS 段结束后）到内存结束前的区域是可用的物理内存区域，我们需要初始化物理页帧分配器，用于管理这些物理内存区域。

```

pub fn init_frame_allocator() {
    let start = PhysicalAddress::from(extern_global!(__kernel_end) as usize)
        .ceil_page()
        .into();
    let end = PhysicalAddress::from(MEMORY_END).floor_page().into();

    FRAME_ALLOCATOR.ref_cell.borrow_mut().init(start..end);
}

```

start 和 end 分别表示内核结束地址和内存结束地址，这两个地址之间的区域是可用的物理内存区域。

FRAME\_ALLOCATOR 是一个全局的物理页帧分配器，通过 SyncRefCell 包装，可以保证多线程安全。在初始化时，简单的将其管理的未使用的物理页帧区域设置为 start..end。

```
pub fn init(&mut self, range: Range<PhysicalPageNumber>) {  
    self.unused = range;  
}
```

### 3.3.3 创建和激活内核地址空间

在内核启动时，需要创建内核地址空间，将内核的代码、数据和堆区映射到内核的虚拟地址空间中。

在创建内核地址空间时，需要完成以下工作：

1. 创建内核页表;
2. 映射跳板页 trampoline;
3. 映射内核代码段 .text;
4. 映射内核数据段 .data 和 .rodata;
5. 映射内核堆区 .bss;
6. 映射剩余的堆区;
7. 映射内存映射IO区域，实现 DMA。

内核地址空间的映射使用线性恒等映射，即将虚拟地址和物理地址一一对应，这样可以简化内核的地址转换过程。映射时，从相应的内存段创建逻辑段，设置相应的权限和属性，然后将逻辑段加入到地址空间中。

```
memory_set.push(  
    MapArea::new(  
        VPNRange::from_addr(  
            VirtualAddress::from(section_start),  
            VirtualAddress::from(section_end),  
        ),  
        MapType::Linear,  
        MapPermission::Read | MapPermission::Execute,  
    ),  
    None,  
);
```

在创建内核地址空间后，需要将内核地址空间激活，即将内核页表的地址设置到 satp 寄存器中，这样 CPU 在访问内存时，会使用内核页表进行地址转换。

```
pub fn activate(&self) {
    let satp = self.page_table.token();
    unsafe {
        sstatus::set_sum();
        satp::write(satp);
        asm!("sfence.vma"); // 刷新 TLB
    }
}
```

在这里，我们通过 `satp::write` 函数将内核页表的地址设置到 `satp` 寄存器中，然后通过 `sfence.vma` 指令刷新 TLB 缓存，确保 CPU 在访问内存时，使用最新的页表进行地址转换。

## 3.4. 初始化文件系统

文件系统是操作系统的重要组成部分，负责管理文件和目录，提供文件的读写和删除等功能。

在初始化文件系统时，我们首先需要初始化磁盘块设备，然后打开磁盘上的文件系统，获取文件系统的根目录。

### 3.4.1 初始化磁盘块设备

对于 QEMU 虚拟机，我们可以使用 `virtio` 驱动来模拟磁盘块设备，然后通过 `virtio` 驱动来访问磁盘块设备。

```
let vaddr = MMIO[0].0;
let header = NonNull::new(vaddr as *mut VirtIOHeader).unwrap();
let transport = unsafe { MmioTransport::new(header) }.unwrap();
let virtio_blk = VirtIOBlk::new(transport).expect("failed to create VirtIOBlock");
VirtIOBlock(Mutex::new(virtio_blk))
```

`virtio` 的设备驱动由 `rcore-os/virtio-drivers` 提供，我们可以通过 `VirtIOBlk::new` 函数来创建一个 `virtio` 块设备。

在创建 `virtio` 块设备后，我们可以通过 `read_block` 和 `write_block` 函数来读写磁盘块设备。

### 3.4.2 打开文件系统

文件系统镜像由文件系统镜像打包工具 `ros-fs-fuse` 创建，在内核启动时可以通过驱动程序读取文件系统镜像，在此基础上打开已有的文件系统。

在打开文件系统时，我们首先需要读取文件系统的超级块，然后根据超级块的信息初始化文件系统。

```
let cache = get_cache(0, dev.clone()).expect("get cache failed");
```

在这里，我们通过 `get_cache` 函数来获取文件系统第一个缓存块。读取该缓存块的内容，然后根据超级块的信息初始化文件系统。

```

assert!(super_block.check());

let inode_total_blocks = super_block.inode_bitmap_blocks + super_block.inode_area_blocks;
let inode_bitmap = Bitmap::new(1, super_block.inode_bitmap_blocks as usize);
let data_bitmap = Bitmap::new(
    (1 + inode_total_blocks) as usize,
    super_block.data_bitmap_blocks as usize,
);

let fs = FileSystem { /* ... */ };

```

在初始化文件系统时，我们首先检查超级块的有效性，然后根据超级块的信息创建位图，然后根据位图的信息初始化文件系统。

### 3.4.3 获取文件系统根目录

根目录 inode 存放在文件系统 inode 区的第一个 inode 中，我们可以通过 inode 区的起始地址和根目录 inode 的编号来获取根目录 inode。

```

pub fn get_disk_inode_pos(&self, inode_id: u32) -> (u32, usize) {
    let inode_size = core::mem::size_of::<DiskInode>();
    let inode_per_block = (BLOCK_SIZE / inode_size) as u32;
    let inode_bid = self.inode_area_start + inode_id / inode_per_block;
    let inode_offset = (inode_id % inode_per_block) as usize * inode_size;
    (inode_bid, inode_offset)
}

fn root_inode(&self) -> MemInode {
    let fs = self.lock();
    let (root_inode_bid, root_inode_offset) = fs.get_disk_inode_pos(0);
    MemInode::new(/* ... */)
}

```

get\_disk\_inode\_pos 函数用于计算 inode 的位置，然后通过 root\_inode 函数获取根目录 inode。

## 3.5. 初始化进程管理

进程管理是操作系统的核心功能之一，负责管理系统的进程和线程，提供进程的创建、调度和销毁等功能。

在初始化进程管理时，我们首先需要初始化进程管理器，然后就绪第一个用户进程 initproc。

进程管理器是一个全局的进程管理器，其简单地包含一个就绪队列。处理器将以时间片轮转的方式从就绪队列中选择进程运行。

第一个用户进程 initproc 是用户态程序的入口，其主要工作是启动sh。



```

pub static ref TASK_MANAGER: SyncRefCell<TaskManager> = {
    SyncRefCell {
        ref_cell: RefCell::new(TaskManager::new()),
    }
};

pub static ref INIT_PROC: Arc<SyncRefCell<ProcessControlBlock>> = {
    Arc::new(SyncRefCell::new({
        let inode = open_file("initproc", OpenFlags::READONLY).unwrap();
        let data = inode.read_all();
        ProcessControlBlock::new(data.as_slice())
    })))
};

pub fn init() {
    TaskManager::put_task(INIT_PROC.clone());
}

```

这里，我们通过 TASK\_MANAGER 全局变量来初始化进程管理器，然后通过 INIT\_PROC 全局变量来初始化第一个用户进程 initproc。

在初始化进程管理器后，我们将第一个用户进程 initproc 加入到就绪队列中。

### 3.6. 建立时钟中断和定时器

操作系统的时间管理和进程调度需要时钟中断和定时器的支持，时钟中断是由硬件定时器产生的中断，用于定时触发操作系统的调度和时间管理。

首先，我们需要启用时钟中断，设置定时器中断使能位即可。

```
riscv::register::sie::set_stimer();
```

调用 Rust SBI 提供的接口设置下一个时钟中断触发时间：

```

pub fn sbi_set_timer(stime_value: u64) {
    rustsbi::Forward {}.set_timer(stime_value);
}

pub fn set_next_timeout(timeout_us: usize) {
    sbi_set_timer((get_time() + timeout_us * (CLOCK_FREQ / USEC_PER_SEC)) as u64);
}

timer::set_next_timeout(trap::handler::TIMER_INTERVAL_USEC);

```

通过 timer::set\_next\_timeout 函数设置下一个时钟中断触发时间，在时钟中断触发时，继续调用 timer::set\_next\_timeout 函数设置下一个时钟中断触发时间，即可实现定时器的循环触发。

```
scause::Trap::Interrupt(const { Interrupt::SupervisorTimer as usize }) => {
    timer::set_next_timeout(TIMER_INTERVAL_USEC);
    // 切换到下一个任务
    TaskManager::cycle_to_next();
}
```

在时钟中断处理函数中，我们首先设置下一个时钟中断触发时间，然后调用进程管理器的 `cycle_to_next` 函数切换到下一个任务，即实现时间片轮转调度。

在用户态程序看来，其在执行过程中时间片到时会被时钟中断打断然后返回继续执行，但实际上是操作系统在时钟中断处理函数中返回到了下一个任务，直到该进程再次被调度时才会继续执行。

### 3.7. 运行用户进程

在操作系统中，用户进程的运行是通过一个无限循环来实现的。

在这个循环中，操作系统会不断地从任务管理器中获取下一个要运行的任务，并进行上下文切换。具体实现如下：

```
pub fn run_tasks() {
    loop {
        let mut this = PROCESSOR.inner_borrow_mut();
        if let Some(task) = TaskManager::get_task() {
            let idle_ctx = this.get_idle_ctx_ptr();
            let mut pcb = task.inner_borrow_mut();
            let next_ctx = &pcb.ctx as *const TaskCtx;
            pcb.status = ProcessStatus::Running;
            drop(pcb);
            this.current = Some(task);
            drop(this);
            unsafe {
                __switch(idle_ctx, next_ctx);
            }
        }
    }
}
```

## 4. 运行用户进程

在操作系统所有初始化工作完成后，即可通过 `run_tasks` 函数运行用户进程。

此时就绪队列中的第一个用户进程 `initproc` 将被调度执行。

```

pub fn run_tasks() {
    loop {
        let mut this = PROCESSOR.inner_borrow_mut();
        if let Some(task) = TaskManager::get_task() {
            let idle_ctx = this.get_idle_ctx_ptr();
            let mut pcb = task.inner_borrow_mut();
            let next_ctx = &pcb.ctx as *const TaskCtx;
            pcb.status = ProcessStatus::Running;
            drop(pcb);
            this.current = Some(task);
            drop(this);
            unsafe {
                __switch(idle_ctx, next_ctx);
            }
        }
    }
}

pub fn schedule(swapped_task_ctx_ptr: *mut TaskCtx) {
    let mut this = PROCESSOR.inner_borrow_mut();
    let idle_ctx = this.get_idle_ctx_ptr();
    drop(this);
    unsafe {
        __switch(swapped_task_ctx_ptr, idle_ctx);
    }
}

```

idle "任务" 并不是一个用户进程，其只在系统调度流中用作占位符。

在系统中，存在两个进程控制流：

一个是运行在内核系统栈上的系统调度流，在这个调度流中，就绪队列的第一个就绪进程会被选中，切换到处理器运行，在时间片结束后，会被切换出处理器，然后回到系统调度流中，继续调度下一个进程；

另一个是运行在进程内核栈上的由进程自己推动的调度流，例如在调用 `sys_shed_yield` 时，进程内核栈上的调度流会调用 `schedule` 函数，主动将自己切换出处理器。

在 `__switch` 中，会将当前控制流快照（通用寄存器、栈指针等）保存到当前进程的上下文中，然后将下一个进程的上下文加载到 CPU 中，在这一过程中，`__switch` 原有的返回链接寄存器 `ra` 的值会被保存替换，在 `__switch` 函数返回时，CPU 会跳转到下一个进程的上下文中继续执行。

```

.section .text
.global __switch
__switch: # a0: current_task_ctx_ptr, a1: next_task_ctx_ptr
    # 在当前任务上下文空间里保存 CPU 当前的寄存器快照
    sd ra, 0*8(a0) # 保存返回地址寄存器 ra
    sd sp, 1*8(a0) # 保存堆栈指针寄存器 sp
    .set rept_i, 0
    .rept 12 # s0~s11
        .set plus2, 2 + rept_i
        STORE_SR %rept_i, %plus2*8, a0 # 保存 s0~s11 寄存器
        .set rept_i, rept_i+1
    .endr

    # 恢复到下一个任务上下文空间里的寄存器快照
    ld ra, 0*8(a1) # 恢复返回地址寄存器 ra
    ld sp, 1*8(a1) # 恢复堆栈指针寄存器 sp
    .set rept_i, 0
    .rept 12 # s0~s11
        .set plus2, 2 + rept_i
        LOAD_SR %rept_i, %plus2*8, a1
        .set rept_i, rept_i+1
    .endr

    ret

```

在最开始，idle 上下文中 ra、sp 等寄存器的值是随意的（这里设置为 0），在第一次调度（调用 \_\_switch 函数）时，idle 上下文会被设为系统调度流，即 run\_tasks 函数中的 \_\_switch 函数所在的上下文。

在 schedule 函数中，会将当前进程的上下文保存，然后调用 \_\_switch 函数切换到 idle 上下文中，切换到系统调度流中，即回到 run\_tasks 函数中，继续调度下一个进程。

## 系统陷入

### 概念

在操作系统领域，系统陷入通常是指用户程序通过执行特定指令，主动请求操作系统内核服务的一种机制。这是一种受控的异常情况，使得处理器从用户态转换为内核态。常见的系统调用如进程管理、文件操作和内存管理都会通过系统陷入来请求内核提供服务。这些系统调用的执行方式体现了系统陷入作为用户程序与内核之间交互接口的定义。

### 基本原理

#### 触发机制

系统陷入是由软件指令主动触发的一种异常。在程序执行过程中，当遇到特定的陷入指令（如某些操作系统中的系统调用指令）时，处理器会暂停当前正在执行的用户程序的正常指令流。例

如，在许多操作系统中，当用户程序需要执行输入 / 输出操作（如读取文件）时，它会执行一个系统调用指令，这个指令就会触发系统陷入。

它与硬件中断有所不同。硬件中断是由外部硬件设备（如键盘按键、网络数据包到达网卡等）产生的异步信号来触发，而系统陷入是程序自身有计划地产生的。

## 处理过程

当系统陷入发生时，处理器会自动将当前程序的状态保存起来。这包括程序计数器（PC）的值，它指向当前正在执行的指令的下一条指令的地址，以及处理器的其他状态信息，如寄存器的值等。这样做是为了在系统陷入处理完成后能够恢复原来程序的执行。

然后，处理器会根据陷入的类型（不同的系统调用或其他陷入原因会有不同的类型编号），查找对应的陷入处理程序（Trap Handler）。这个处理程序通常是操作系统内核的一部分。例如，对于一个读取文件的系统调用陷入，会找到文件系统相关的陷入处理程序来处理这个请求。

陷入处理程序会执行相应的操作。以文件读取为例，它会检查文件权限、从存储设备读取数据到内存缓冲区等操作。在处理程序完成操作后，会将结果返回给触发陷入的用户程序。如果操作成功，可能会将读取的数据传递给用户程序；如果失败，会返回一个错误码。

最后，处理器会恢复之前保存的程序状态，包括恢复程序计数器和寄存器的值，使得用户程序能够从系统陷入指令之后的下一条指令继续执行。

## 用途方面

- **系统调用实现：**

系统陷入是实现系统调用的重要机制。系统调用是用户程序请求操作系统内核服务的接口。例如，用户程序想要创建一个新的进程、分配内存或者进行网络通信等操作，这些功能是由操作系统内核提供的。通过系统陷入，用户程序可以安全地从用户态进入内核态，让内核执行相应的操作，从而实现系统调用。

- **异常处理和调试支持：**

它也用于处理一些软件异常情况。比如，在一些编程语言中，如果程序出现了除零错误或者数组越界访问等情况，可能会通过系统陷入机制来通知操作系统，操作系统可以采取相应的措施，如终止违规的程序或者进行调试信息收集。同时，在调试过程中，调试工具可以利用系统陷入来设置断点，当程序执行到断点位置（通过特殊的陷入指令）时，就可以暂停程序执行，方便开发者查看程序状态。

## 具体实现

### 系统陷入触发

#### 系统调用相关函数调用

用户程序通过执行特定指令（如系统调用指令）主动触发系统陷入。在代码中，当用户程序需要操作系统内核提供服务时，如进行文件操作、进程管理等，会调用相应的系统调用函数，这些函数最终会转换为触发系统陷入的指令。

以ProcessControlBlock结构体的exec方法为例：

```
pub fn exec(self: &Arc<SyncRefCell<ProcessControlBlock>>, app_data: &[u8],
args: Vec<String>) {
    let (memory_set, mut user_sp, entry) = MemorySet::from_elf_app(app_data);
    let trap_ctx_ppn = PhysicalPageNumber::from(
        &memory_set
            .translate(VirtualPageNumber::from(VirtualAddress::from(TRAP_CONTEXT)))
            .unwrap(),
    );

    for i in (0..args.len()).rev() {
        user_sp -= 1;
        *get_translated_refmut(memory_set.token(), user_sp as *mut u8) = 0;
        for byte in args[i].bytes().rev() {
            user_sp -= 1;
            *get_translated_refmut(memory_set.token(), user_sp as *mut u8)
= byte;
        }
    }
    let argv_start = user_sp;
    let argc = args.len();

    user_sp -= user_sp % core::mem::size_of::<usize>();
    let mut pcb = self.inner_borrow_mut();
    pcb.memory_set = memory_set;
    pcb.trap_ctx_ppn = trap_ctx_ppn;
    let ctx = pcb.get_trap_ctx();
    *ctx = TrapCtx::init_app_context(
        entry,
        user_sp,
        KERNEL_SPACE.ref_cell.borrow_mut().token(),
        pcb.kernel_stack.top(),
        trap_handler as usize,
    );

    *ctx.a(1) = argc as usize;
    *ctx.a(2) = argv_start;
}
```

这里在设置新应用程序的陷阱上下文（TrapCtx）时，涉及到将trap\_handler函数的地址传递进去，这暗示了在执行这个应用程序过程中，当发生系统陷入时，会跳转到trap\_handler进行处理，体现了系统陷入的触发机制与系统调用相关操作的紧密联系。

## 系统调用指令底层转换

1. RISC-V 系统调用指令概述 在 RISC-V 架构中，系统调用指令为 ecall。当执行 ecall 指令时，处理器会进入异常处理模式，并根据预先设置的异常向量表跳转到相应的处理程序。系统调用号存放在 a7 寄存器中，而参数则按照约定存放在其他寄存器中。

```
asm!(
    "ecall",
    inlateout("a0") args[0] => ret,
    in("a1") args[1],
    in("a2") args[2],
    in("a7") callid,
);
```

2. 与陷入处理程序的关联 在代码中，`trap_handler` 函数是系统陷入的总处理程序入口。当执行 `ecall` 指令触发系统陷入后，处理器会跳转到这个处理程序。在 `ProcessControlBlock` 结构体的相关方法中，涉及到设置陷阱上下文并关联 `trap_handler` 函数地址的操作。这为 `ecall` 指令触发系统陷入后的处理做了准备。可以推测，当执行 `ecall` 指令时，处理器会根据当前程序的状态信息以及预先设置的与 `trap_handler` 相关的信息，找到并跳转到 `trap_handler` 函数进行处理。
3. 系统调用号与参数传递 为了实现系统调用处理程序根据系统调用号来决定执行哪个具体的系统调用处理逻辑，必须在执行 `ecall` 指令前，将系统调用号正确地存放在 `a7` 寄存器中。并将系统调用的参数按照约定设置到 `a0 - a6` 等寄存器中。这样，当系统陷入发生后，内核中的系统调用处理函数可以从这些寄存器中获取正确的参数值来执行相应的系统调用处理逻辑。
4. 与任务上下文和内存管理的关系 执行 `ecall` 指令触发系统陷入之前，当前任务的上下文需要被保存，以便在系统调用处理完成后能够恢复任务的执行。这一步操作与 `TaskCtx` 结构体以及 `__switch` 函数相关。在执行 `ecall` 指令时，会涉及到一些指令来触发上下文保存操作，或者与这些上下文保存相关的代码协同工作。在系统调用处理过程中，会涉及到内存管理操作。例如，在 `ProcessControlBlock` 结构体的 `exec` 方法中，执行新应用程序时需要加载程序代码和数据到内存，这可能涉及到 `ecall` 指令与内存管理模块之间的交互。在执行 `ecall` 指令时，需要传递一些与内存管理相关的信息，以便内核在处理系统调用时能够正确地进行内存操作，同时确保系统调用处理过程中的内存安全性和正确性。

## 与系统陷入处理程序的关联

`ProcessControlBlock` 结构体的 `new` 方法中，在初始化进程控制块时，设置了陷阱上下文相关操作，其中将 `trap_handler` 函数的地址作为参数传递，这为系统陷入触发后能够正确跳转到处理程序做了准备。

```

pub fn new(app_data: &[u8]) -> Self {
    let (memory_set, user_sp, entry) = MemorySet::from_elf_app(app_data);
    let trap_ctx_ppn = PhysicalPageNumber::from(
        &memory_set
            .translate(VirtualPageNumber::from(VirtualAddress::from(TRAP_CON
TEXT)))
            .unwrap(),
    );
    let pid = PidAllocator::alloc_pid();
    let kernel_stack = KernelStack::new(&pid);
    let kernel_stack_top = kernel_stack.top();
    let pcb = Self {
        pid,
        kernel_stack,
        status: ProcessStatus::Ready,
        ctx: TaskCtx::goto_trap_return(kernel_stack_top),
        memory_set,
        trap_ctx_ppn,
        base_size: user_sp,
        parent: None,
        children: Vec::new(),
        exit_code: 0,
        fd_table: vec![
            Some(Arc::new(Stdin)),
            Some(Arc::new(Stdout)),
            Some(Arc::new(Stdout)),
        ],
    };
    let ctx = pcb.get_trap_cx();
    *ctx = TrapCtx::init_app_context(
        entry,
        user_sp,
        KERNEL_SPACE.ref_cell.borrow_mut().token(),
        kernel_stack_top,
        trap_handler as usize,
    );
    pcb
}

```

## 系统调用参数传递与陷入触发

在exec方法中，除了上述提到的设置陷阱上下文等操作，还涉及到将应用程序的入口地址、堆栈指针等参数传递给陷阱上下文的初始化函数TrapCtx::init\_app\_context，为新应用程序的执行准备了环境，同时也与系统陷入触发后的处理流程相关。当系统陷入触发后，处理程序可以根据这些传递的参数正确地恢复程序执行的环境，如将程序计数器设置为应用程序的入口地址，从而使得新应用程序能够在系统陷入处理完成后从正确的位置开始执行。

## 陷入处理程序执行

当系统陷入发生后，处理器会根据陷入的类型查找对应的陷入处理程序（Trap Handler）。



- 以switch.txt中的\_\_switch函数为例，它在任务切换相关的系统陷入处理中起到关键作用。当进行任务切换时，\_\_switch函数会保存当前任务的上下文，并恢复下一个任务的上下文，从而实现任务的切换操作，这是系统陷入处理程序执行的一部分具体工作。

### trap\_handler 函数入口及基本操作

在处理系统陷入时，首先需要接收一个 TrapCtx 类型的可变引用参数，这个参数包含了系统陷入发生时的各种状态信息，例如程序计数器和寄存器值等。接下来，为了在陷入处理完成后能够恢复任务的执行状态，需要保存当前任务的上下文到指定的内存位置。随后，需要判断系统陷入的类型，如果类型为用户环境调用，则调用 syscall::syscall\_handler 函数来处理系统调用，这是陷入处理程序的核心部分，不同的系统调用会在这个函数中得到相应的处理。处理完毕后，需要更新陷阱上下文，确保传入的 cx 参数包含了最新的任务陷阱上下文信息。最后，为了使任务能够继续执行，需要将之前保存的任务上下文重新加载到相应的内存位置。这一系列步骤共同确保了系统陷入后能够正确处理并恢复任务的执行。

```

#[unsafe(no_mangle)]
pub fn trap_handler() -> ! {
    set_user_trap_entry();
    let scause = scause::read().cause();
    let stval = stval::read();
    match scause {
        // 用户态环境调用异常
        scause::Trap::Exception(const { Exception::UserEnvCall as usize })
=> {
            let ctx = Processor::current_trap_cx().unwrap();
            ctx.sepc += 4; // 跳过 ecall 指令
            let ret = syscall(Syscall::from(*ctx.a(7)), [*ctx.a(0), *ctx.a
(1), *ctx.a(2)]);
            // syscall 之后上下文可能会被修改, 所以需要重新获取
            let ctx = Processor::current_trap_cx().unwrap();
            *ctx.a(0) = ret as usize; // 将系统调用返回值存入 a0 寄存器
        }
        // 存储错误或存储页错误异常
        scause::Trap::Exception(const { Exception::StoreFault as usize })
| scause::Trap::Exception(const { Exception::StorePageFault as usiz
e }) => {
            todo!("终止进程");
        }
        // 非法指令异常
        scause::Trap::Exception(const { Exception::IllegalInstruction as us
ize }) => {
            todo!("终止进程");
        }
        // 监督者定时器中断
        scause::Trap::Interrupt(const { Interrupt::SupervisorTimer as usiz
e }) => {
            // todo!("Timer interrupt");
            // 设置下一个定时器中断
            set_next_timeout(TIMER_INTERVAL_USEC);
            // 切换到下一个任务
            TaskManager::cycle_to_next();
        }
        // 未处理的陷阱
        _ => {
            panic!("未处理的陷阱: {:?}, stval: {:#x}", scause, stval);
        }
    };
    // 返回到用户态
    trap_return();
}

```

## 任务管理与陷入处理的关联

在 `cycle_to_next` 和 `replace_to_next` 方法中, 在任务切换等操作中会与陷入处理程序相互协作。例如, `cycle_to_next` 方法将当前任务状态设置为就绪, 放入就绪队列, 然后调用 `Processor::schedule` 方法进行任务调度, 这涉及到陷入处理程序执行后的任务切换操作。

在 processor.rs 中的 Processor 结构体相关方法中，schedule 方法用于实际的任务调度，其调用 \_\_switch 函数来切换任务上下文，这个过程与陷入处理程序紧密相关。陷入处理程序在处理完系统陷入后，会通过这些任务管理和调度相关的函数来决定是否切换任务以及如何切换任务，从而保证系统的正常运行和任务的合理调度。

## 上下文保存与恢复

系统陷入时，处理器会自动保存当前程序的上下文，包括程序计数器（PC）、通用寄存器的值等信息。TaskCtx结构体用于保存任务的上下文信息，如ra、sp以及s数组。

### \_\_switch函数

在\_\_switch汇编函数里，详细实现了上下文的保存和恢复操作。通过将当前任务的寄存器值存储到TaskCtx结构体对应的内存位置，以及从下一个任务的TaskCtx结构体中加载寄存器值，确保任务切换前后程序状态的正确维护。

```
# 任务切换
.altmacro
.macro STORE_SR regi, offset, base
    sd s\regi, \offset(\base) # 将寄存器 s\regi 的值存储到基址 \base 的偏移
\offset 处
.endm
.macro LOAD_SR regi, offset, base
    ld s\regi, \offset(\base) # 从基址 \base 的偏移 \offset 处加载值到寄存器
s\regi
.endm

.section .text
.global __switch
__switch: # a0: current_task_ctx_ptr, a1: next_task_ctx_ptr
    # 在当前任务上下文空间里保存 CPU 当前的寄存器快照
    sd ra, 0*8(a0) # 保存返回地址寄存器 ra
    sd sp, 1*8(a0) # 保存堆栈指针寄存器 sp
    .set rept_i, 0
    .rept 12 # s0~s11
        .set plus2, 2 + rept_i
        STORE_SR %rept_i, %plus2*8, a0 # 保存 s0~s11 寄存器
        .set rept_i, rept_i+1
    .endr

    # 恢复下一个任务的寄存器
    ld ra, 0*8(a1) # 恢复返回地址寄存器 ra
    ld sp, 1*8(a1) # 恢复堆栈指针寄存器 sp
    .set rept_i, 0
    .rept 12 # s0~s11
        .set plus2, 2 + rept_i
        LOAD_SR %rept_i, %plus2*8, a1 # 恢复 s0~s11 寄存器
        .set rept_i, rept_i+1
    .endr

    ret # 返回
```

## schedule函数

这个函数在任务调度时，通过调用 `__switch` 函数，传入当前任务上下文指针和空闲任务上下文指针，实现了将当前任务上下文保存到 `switched_task_cx_ptr` 指向的位置，并恢复空闲任务上下文，从而完成任务切换过程中的上下文保存与恢复操作。

```
pub fn schedule(switched_task_cx_ptr: *mut TaskCtx) {
    let mut this = PROCESSOR.inner_borrow_mut();
    let idle_ctx = this.get_idle_ctx_ptr();
    drop(this);
    unsafe {
        __switch(switched_task_cx_ptr, idle_ctx);
    }
}
```

## 内核服务提供与返回结果

陷入处理程序会执行相应的内核服务操作，如文件系统操作、进程管理操作等，这些操作在代码中分散在各个相关模块中。内核服务完成后，会将结果返回给触发陷入的用户程序。如果操作成功，可能会将所需的数据传递给用户程序；如果失败，会返回一个错误码，用户程序根据返回结果继续执行后续操作。在代码中，通过各种数据结构和函数调用约定来实现结果的传递和程序的继续执行，如在系统调用返回时，会根据返回值在用户程序中进行相应的错误处理或数据使用逻辑。

### 系统调用处理与内核服务提供

对于不同的系统调用号来创建一个新的进程，这涉及到内核中进程管理相关的服务，包括复制进程控制块、分配新的进程 ID、设置进程状态等操作，从而为用户程序提供了创建新进程的内核服务。

### 进程管理相关内核服务

在 `ProcessControlBlock` 结构体的方法实现了许多与进程管理相关的内核服务操作，这些操作在系统调用处理过程中会被调用，从而为用户程序提供相应的服务。例如，`fork` 方法用于创建新的进程，其会进行一系列复杂的操作，包括复制父进程的内存空间、创建新的内核栈、分配新的进程 ID 以及设置新进程的各种属性，这些操作都是内核为用户程序提供的创建进程的服务。`exec` 方法用于在当前进程上执行新的应用程序，会加载新的应用程序代码和数据到内存，设置新的陷阱上下文，并更新进程的相关属性，如程序入口地址、堆栈指针等，这为用户程序提供了执行新应用程序的内核服务。

# 存储管理模块

## 存储管理模块概述

### 模块简介

存储管理模块是操作系统内核的重要组成部分，其主要职责是为操作系统提供高效、稳定的内存和存储管理能力。在基于 **RISC-V** 架构的系统中，存储管理模块实现了**虚拟内存地址空间的抽象、物理页帧的分配与管理、内核动态内存分配以及多级页表的维护与操作**。通过这些功能，存储管理模块为操作系统提供了可靠的存储支持，同时提升了内存管理的灵活性和效率。

本模块设计的核心思想是利用**分层结构**和**模块化**实现，将存储管理划分为多个子功能模块，每个模块分别负责特定的任务，并通过统一的接口协同工作。通过后续的介绍我们容易得知这种设计方式不仅仅提升了本模块的可维护性和扩展性，更重要的是**代码逻辑变得更加清晰、使得用户阅读起来更加容易**。

## 模块的核心作用

### 1. 虚拟地址空间管理

通过分页机制为每个进程提供独立的虚拟地址空间，隔离不同进程之间的内存访问，确保系统的安全性和稳定性。

### 2. 物理页帧管理

对物理内存页帧的分配和回收进行精确控制，为内核和用户进程提供可靠的内存分配接口。

### 3. 动态内存分配

实现内核级的动态内存分配器，用于满足内核在运行时的内存需求，支持高效的内存分配与回收。

### 4. 多级页表管理

结合硬件支持的多级页表机制，实现虚拟地址到物理地址的高效映射，提供虚实地址转换的基础支持。

## 模块结构

存储管理模块由以下几个核心部分组成，每个部分独立负责特定的功能，并通过接口进行交互：

### 1. 地址抽象与管理 (address.rs)

提供对虚拟地址和物理地址的抽象和操作方法，是模块中所有内存管理功能的基础。

### 2. 物理页帧分配器 (frame\_allocator.rs)

管理物理内存的使用情况，提供页帧的分配与回收功能，为内核和用户进程提供物理内存支持。

### 3. 动态内存分配器 (/slab\_allocator) 使用 Slab 和 LinkedList 分配器实现内核和用户程序的动态内存分配，支持内核和用户程序在运行时的内存需求。

### 4. 多级页表管理 (page\_table.rs)

实现基于 RISC-V SV39 模式的多级页表管理，支持虚拟地址和物理地址的映射与权限控制。

### 5. 地址空间管理 (memory\_set.rs)

定义地址空间的结构与管理逻辑，包括逻辑段 (segment) 的划分和多级页表的管理。

### 6. 模块初始化 (init.rs)

负责存储管理模块的初始化，包括页帧分配器、堆分配器和地址空间的初始化。

## 模块的设计目标

### 1. 安全性

通过独立的虚拟地址空间和权限控制机制，确保内核与用户进程、进程与进程之间的内存隔离。

### 2. 高效性

利用缓存和分页机制，减少内存访问的延迟，同时优化物理内存的利用率。

### 3. 灵活性

提供动态内存分配支持，允许内核在运行时根据需要分配或释放内存资源。

### 4. 可扩展性

通过模块化设计和统一接口，方便未来的功能扩展和代码维护。

## 模块在操作系统中的地位

存储管理模块位于操作系统内核的核心位置，作为桥梁连接内核的其他模块与硬件设备：

- **向上**：为任务调度模块、文件系统模块等提供内存分配和地址管理支持。
- **向下**：通过多级页表和物理页帧分配器与硬件的内存管理单元（MMU）进行交互，完成地址转换和内存分配。

通过这一模块，操作系统能够屏蔽底层物理内存的复杂性，为用户程序提供统一且高效的内存管理接口。

## 存储管理/地址

在启用分页系统的情况下，物理内存被划分为大小相等的页框，每个页框的大小为4KB。页框的编号从0开始，逐个递增。页框的编号称为物理地址。

而与之对应的逻辑地址空间被划分为大小相等的页，每个页的大小也为4KB。页的编号称为逻辑地址（虚拟地址）。

操作系统通过将物理地址与虚拟地址分离，实现了进程间的内存隔离和虚拟地址空间的灵活管理。在 RISC-V 架构下，虚拟地址和物理地址的设计基于 **SV39 分页机制**。

在本项目中，`src/mm/address.rs` 文件提供了对虚拟地址（`VirtAddr`）和物理地址（`PhysAddr`）的抽象。以下内容将详细阐述其设计和实现。

在 RISC-V 的 **SV39 分页机制**下，虚拟地址和物理地址的结构化设计如下：

### 虚拟地址（VirtualAddress）

虚拟地址结构如下：

```
| VPN[2] | VPN[1] | VPN[0] | Page Offset |
```

- **VPN（Virtual Page Number）**：虚拟页号，由 `VPN[2]`、`VPN[1]` 和 `VPN[0]` 三级组成，每级占 9 位，共 27 位。
- **Page Offset**：页内偏移量，占 12 位，标识页内的具体字节位置。

- **总长度**: 39 位，一个虚拟地址仍占用 64 位，高 25 位未使用。

在代码中，虚拟地址被表示为：

```
pub struct VirtualAddress(pub usize);
```

`VirtualAddress` 是对虚拟地址的抽象，底层以 `usize` 表示。该结构提供了多种方法用于解析虚拟地址的各部分。

## 核心方法

### 与 `usize` 互相转换

由于只有低 39 位有效，在转换时只保留低 39 位。

```
impl From<usize> for VirtualAddress {
    fn from(address: usize) -> Self {
        // 只保留低 39 位
        Self(address & ((1 << VIRTUAL_ADDRESS_WIDTH_SV39) - 1))
    }
}
impl From<VirtualAddress> for usize {
    fn from(address: VirtualAddress) -> Self {
        address.0
    }
}
```

### 与 虚拟页号 `VirtualPageNumber` 互相转换

只需要将虚拟页号左移 12 位，即可得到虚拟地址。

```
impl From<VirtualAddress> for VirtualPageNumber {
    fn from(address: VirtualAddress) -> Self {
        VirtualPageNumber(address.0 >> PAGE_OFFSET_WIDTH_SV39)
    }
}
impl From<VirtualPageNumber> for VirtualAddress {
    fn from(page_number: VirtualPageNumber) -> Self {
        VirtualAddress(page_number.0 << PAGE_OFFSET_WIDTH_SV39)
    }
}
```

## 计算相关地址

```
impl VirtualAddress {
    /// 计算页内偏移量
    pub fn page_offset(&self) -> usize {
        self.0 & (PAGE_SIZE_SV39 - 1)
    }

    /// 计算对齐到页边界的虚拟地址
    pub fn floor(&self) -> VirtualAddress {
        VirtualAddress(self.0 & !(PAGE_SIZE_SV39 - 1))
    }

    /// 计算向上对齐到页边界的虚拟地址
    pub fn ceil(&self) -> VirtualAddress {
        VirtualAddress((self.0 + PAGE_SIZE_SV39 - 1) & !(PAGE_SIZE_SV39 -
1))
    }

    /// 计算对齐到页边界的虚拟页号
    pub fn floor_page(&self) -> VirtualPageNumber {
        VirtualPageNumber::from(self.floor())
    }

    /// 计算向上对齐到页边界的虚拟页号
    pub fn ceil_page(&self) -> VirtualPageNumber {
        VirtualPageNumber::from(self.ceil())
    }
}
```

## 物理地址 (PhysicalAddress)

物理地址的结构较为简单，直接表示物理页号和页内偏移量。格式如下：

```
| Physical Page Number | Page Offset |
```

- **Physical Page Number (PPN)** : 物理页号，占据高 44 位。
- **Page Offset**: 页内偏移，占据低位 12 位。
- **总长度**: 56 位，一个物理地址仍占用 64 位，高 8 位未使用。

在代码中，物理地址被表示为：

```
pub struct PhysicalAddress(pub usize);
```

PhysAddr 同样以 `usize` 表示，并提供了类似的解析方法。

### 核心方法

与 `usize` 互相转换



```
impl From<usize> for PhysicalAddress {
    fn from(address: usize) -> Self {
        // 只保留低 56 位
        Self(address & ((1 << PHYSICAL_ADDRESS_WIDTH_SV39) - 1))
    }
}
impl From<PhysicalAddress> for usize {
    fn from(address: PhysicalAddress) -> Self {
        address.0
    }
}
```

与 物理页号 `PhysicalPageNumber` 互相转换

```
impl From<PhysicalAddress> for PhysicalPageNumber {
    fn from(address: PhysicalAddress) -> Self {
        PhysicalPageNumber(address.0 >> PAGE_OFFSET_WIDTH_SV39)
    }
}
impl From<PhysicalPageNumber> for PhysicalAddress {
    fn from(page_number: PhysicalPageNumber) -> Self {
        PhysicalAddress(page_number.0 << PAGE_OFFSET_WIDTH_SV39)
    }
}
```

计算相关地址

```
impl PhysicalAddress {
    /// 计算物理地址在其所在页内的偏移量
    pub fn page_offset(&self) -> usize {
        self.0 & (PAGE_SIZE_SV39 - 1)
    }

    /// 计算对齐到页边界的物理地址
    pub fn floor_page(&self) -> PhysicalAddress {
        PhysicalAddress(self.0 & !(PAGE_SIZE_SV39 - 1))
    }

    /// 计算向上对齐到页边界的物理地址
    pub fn ceil_page(&self) -> PhysicalAddress {
        PhysicalAddress((self.0 + PAGE_SIZE_SV39 - 1) & !(PAGE_SIZE_SV39 -
1))
    }
}
```

内存操作

需要获取物理地址指向数据的（可变）引用，以实现对物理内存的读写操作。这里提供了 `get_mut` 方法，用于获取指向物理地址的可变引用。

```
impl PhysicalAddress {
    pub fn get_mut<T>(&self) -> &'static mut T {
        unsafe { &mut *(self.0 as *mut T) }
    }
}
```

这里简单的将指针指向的数据视为泛型类型 `T`，并返回其可变引用。

## 虚拟页号 (VirtualPageNumber)

虚拟页号是虚拟地址的一部分，在查找页表时进行地址转换时总是以页为单位。因此，虚拟页号的抽象也是必要的。

在代码中，虚拟页号被表示为：

```
pub struct VirtualPageNumber(pub usize);
```

`VirtualPageNumber` 同样以 `usize` 表示，并提供了类似的解析方法。除了和 `usize` 以及 `VirtualAddress` 互相转换外，还提供了获取其三级分页页表索引的方法。

```
impl VirtualPageNumber {
    pub fn indexes(&self) -> [usize; 3] {
        let vpn = self.0;
        [vpn >> 18 & 0x1ff, vpn >> 9 & 0x1ff, vpn & 0x1ff]
    }
}
```

只需要将虚拟页号右移相应位数，并使用掩码提取出三级页表索引即可。

## 物理页号 (PhysicalPageNumber)

物理页号的抽象与虚拟页号类似，提供了与 `usize` 互相转换的方法，在实现中，物理页号表示为：

```
pub struct PhysicalPageNumber(pub usize);
```

`PhysicalPageNumber` 同样以 `usize` 表示，并提供了类似的解析方法。除了和 `usize` 以及 `PhysicalAddress` 互相转换外，还提供了获取其所在物理页数据的方法。

```

impl PhysicalPageNumber {
    /// 获取物理页号所在的物理地址页的可变引用
    pub fn get_mut<T>(&self) -> &'static mut T {
        let addr: PhysicalAddress = (*self).into();
        unsafe { &mut *(addr.0 as *mut T) }
    }

    /// 获取物理页号所在的物理地址页的引用
    pub fn get_ref<T>(&self) -> &'static T {
        let addr: PhysicalAddress = (*self).into();
        unsafe { &*(addr.0 as *const T) }
    }

    /// 获取物理页号所在的物理地址页的页表项数组的可变引用
    pub fn get_pte_array(&self) -> &'static mut [PageTableEntry] {
        let addr: PhysicalAddress = (*self).into();
        unsafe { core::slice::from_raw_parts_mut(addr.0 as *mut PageTableEntry, 512) }
    }

    /// 获取物理页号所在的物理地址页的字节数组的可变引用
    pub fn get_bytes_array(&self) -> &'static mut [u8] {
        let addr: PhysicalAddress = (*self).into();
        unsafe { core::slice::from_raw_parts_mut(addr.0 as *mut u8, 4096) }
    }
}

```

在开发过程中，我们经常需要获取物理页号所在的物理地址页的页表项数组或字节数组的引用，以实现页表项或者其他字节数据的读写操作。这里提供了 `get_pte_array` 和 `get_bytes_array` 方法，用于获取页表项数组和字节数组的可变引用。

同时，为了适应更多的场景，这里还提供了 `get_mut` 和 `get_ref` 方法，用于获取物理页号所在的物理地址页的可变引用和引用。

## 小结

`address.rs` 文件实现了对虚拟地址和物理地址的基本抽象，提供了地址解析的核心方法，为存储管理模块中的页表管理、地址空间分配等功能奠定了基础。

## 存储管理/动态内存分配（Slab 堆分配器）

### 简介

在没有动态内存分配的操作系统中，内核和用户程序只能使用静态内存分配，即在编译时确定内存大小。例如，为了存放一个数组，需要提前分配足够的内存空间。

```

let mut array = [0; 1024]; // 静态分配 1024 个元素的数组

```

但是，这个方法存在以下问题：

- 静态内存分配需要提前确定内存大小，不适用于动态数据结构。
- 静态内存分配可能导致内存浪费或溢出。例如，这个数组可能只在极少数情况下才被使用，或者只在极端情况下才需要 1024 个元素，但是却占用了 1024 个元素的内存空间。

为了解决这些问题，操作系统需要提供动态内存分配的支持。

动态内存分配允许程序在运行时根据需要分配和释放内存，提高内存利用率和灵活性。

在本项目中，动态内存分配器在 `slab_allocator` 包中实现，主要采用 Slab 分配器的设计。Slab 分配器是一种高效的内存分配器，通过预先分配一定数量的固定大小的内存块（Slab），并在需要时分配和回收这些内存块，以减少内存碎片和提高性能。

本节将详细分析堆分配器的设计与实现，并通过代码解析堆分配的核心功能。

## 动态内存分配的基础概念

### 堆内存

堆内存是在内核和用户态程序执行时动态分配的内存区域，其实际使用大小随程序运行时的需要而变化。

对于一个需要使用堆内存的变量，其在生命周期开始时，需要向堆分配器请求一块内存；在生命周期结束时，需要释放这块内存。

在此内核和用户程序库的实现中，堆内存是一块静态分配的内存区域的包装和抽象，在这块静态内存区域上实现堆分配器，即可将这块静态内存区域的子区域作为申请的内存块，实现内存的动态分配。

### 堆分配器

堆分配器是一个管理堆内存的模块，负责分配和释放内存。一个堆分配器在其初始化时即拥有了一块静态内存区域，可以在这块静态内存区域上进行内存的动态分配。

堆分配器的设计目标：

1. **高效性**：减少分配与释放的时间开销。
2. **低碎片率**：尽可能减少内存碎片。
3. **线程安全**：支持并发环境中的内存分配。

此内核和用户程序库的堆分配器采用 Slab 分配器的设计，通过预先分配一定数量的固定大小的内存块（Slab），并在需要时分配和回收这些内存块，以减少内存碎片和提高性能。

而对于大块的内存分配，堆分配器转用 LinkedList 分配器，通过链表的方式管理大块内存的分配与释放。

如此，便可实现了一个高效、低碎片率的堆分配器，对于小内存分配，其不会产生较大的内存碎片，对于大内存分配，其也能够高效地进行内存分配。

## Slab 分配器

Slab 分配器是一种简单的内存分配器，它将内存分配为固定大小的块，并在需要时分配这些块。

Slab 分配器的核心思想是将内存划分为多个固定大小的块，每个块称为一个 Slab。Slab 分配器维护一个空闲链表，用于存储空闲的 Slab。当需要分配内存时，Slab 分配器从空闲链表中取出一个 Slab，并将其分配给请求者。当释放内存时，Slab 分配器将 Slab 放回空闲链表。

Slab 分配器的优点是高效、低碎片率，适用于分配固定大小的内存块。

```
pub struct Slab {
    pub block_size: usize,
    pub block_num: usize,
    free_list: FreeList,
}

struct FreeList {
    len: usize,
    head: Option<&'static mut FreeBlock>,
}

struct FreeBlock {
    next: Option<&'static mut FreeBlock>,
}
```

每个空闲链表链表项 FreeBlock 结构体“占用”一个固定大小的内存块，并指向下一个空闲块。一个空闲链表即可将同种大小的空闲块串联起来，

在分配时，只需从链表中取出一个空闲块；在释放时，只需将空闲块插入链表头部即可。

```
pub fn push(&mut self, free_block: &'static mut FreeBlock) {
    free_block.next = self.head.take();
    self.head = Some(free_block);
    self.len += 1;
}

pub fn pop(&mut self) -> Option<&'static mut FreeBlock> {
    self.head.take().map(|free_block| {
        self.head = free_block.next.take();
        self.len -= 1;
        free_block
    })
}
```

在扩充 Slab 分配器时，只需要在新分配的内存区域上以 Slab 的大小建立一个新的空闲链表，并将其加入到 Slab 分配器的链表中即可。

```
pub fn grow(&mut self, start: Address, slab_size: usize) {
    let block_num = slab_size / self.block_size;
    self.block_num += block_num;
    // 添加到 self.free_list
    for i in 0..block_num {
        let block = (start + i * self.block_size) as *mut FreeBlock;
        let block = unsafe { &mut *block };
        self.free_list.push(block);
    }
}
```

## 堆分配器

```
pub struct Heap {
    /// Slab 分配器数组
    slabs: [slab::Slab; SLABS_NUM],
    /// Fallback Linked List Allocator
    fallback: linked_list_allocator::Heap,
    /// 用户请求的字节数
    user: usize,
    /// 实际分配的字节数
    allocated: usize,
    /// 堆中的总字节数
    total: usize,
}
```

此堆分配器包含了两种内存分配器：Slab 分配器和 LinkedList 分配器。

Slab 分配器用于分配小块内存，在这里，我们定义了 SLABS\_NUM (=7) 种不同大小的 Slab 分配器，分别用于分配 64、128、256、512、1024、2048 和 4096 字节大小的内存块。

LinkedList 分配器用于分配大块内存，当 Slab 分配器无法满足用户请求时，堆分配器会转用 LinkedList 分配器。

### 初始化

在堆分配器构建时，其各个子分配器都是空的，在初始化时，需要将一段连续的内存空间划分给各个子分配器。

```
pub unsafe fn init(&mut self, start: usize, size: usize) {
    let alloc_size = size / ALLOCATORS_NUM;
    let total_slab_size = alloc_size * SLABS_NUM;
    let fallback_size = alloc_size;
    unsafe {
        self.add_to_heap(start, start + total_slab_size);
        self.init_fallback(start + total_slab_size, fallback_size);
    }
}
```

在初始化时，堆内存分配器以相同的权重将初始内存分配给各个子分配器，即确保每个子分配器管理的内存尽可能相等，以满足不同大小内存块的分配需求。

```
unsafe fn init_fallback(&mut self, mut start: usize, size: usize) {
    start = (start + size_of::<usize>() - 1) & (!size_of::<usize>() + 1);
    unsafe {
        self.fallback.init(start as *mut u8, size);
    }
    self.total += size;
}

pub unsafe fn add_to_heap(&mut self, mut start: usize, mut end: usize) {
    start = (start + size_of::<usize>() - 1) & (!size_of::<usize>() + 1);
    end &= !size_of::<usize>() + 1;
    assert!(start <= end);
    let new_heap_size = end - start;
    self.total += new_heap_size;
    let slab_size = new_heap_size / SLABS_NUM;
    for slab_i in 0..SLABS_NUM {
        let slab_start = start + slab_i * slab_size;
        match self.get_inner(slab_i) {
            AllocType::Slab(i) => {
                self.slabs[i].grow(slab_start, slab_size);
            }
            AllocType::Fallback => {
                unreachable!("Fallback allocator should not be initialized
here");
            }
        }
    }
}
```

这里，add\_to\_heap 方法将一段连续的内存空间划分给各个 Slab 分配器，在分配时，将内存空间平分给各个 Slab 分配器，每个 Slab 分配器获得从总内存起始start + i \* slab\_size 到 start + (i+1) \* slab\_size 的内存空间。

由于 Linked List 分配器的内存分配方式不同，其需要连续的内存空间，因此在初始化时，需要将一段连续的内存空间划分给 Linked List 分配器。在这之后不随 Slab 分配器的内存分配而变化。

## 分配

堆分配器的分配方法是一个简单的分配器选择方法，它根据用户请求的内存大小选择合适的 Slab 分配器或 Linked List 分配器进行内存分配。

```
pub fn get_slab_index(mut size: usize) -> usize {
    if size <= MIN_ALLOC_SIZE {
        return 0;
    }
    if size > MAX_SLAB_SIZE {
        return ALLOCATORS_NUM - 1; // 回退到链表分配器
    }
    // 64 -> 0, 65-128 -> 1, 129-256 -> 2, ...
    size = size.next_power_of_two();
    size.trailing_zeros() as usize - 6
}
```

get\_slab\_index 方法根据用户请求的内存大小选择合适的 Slab 分配器，如果请求的内存大小小于等于最小内存块大小，则选择第一个 Slab 分配器；如果请求的内存大小大于最大 Slab 分配器的内存块大小，则选择 Linked List 分配器；否则，选择最接近且大于等于请求内存大小的 Slab 分配器。

```
pub fn alloc(&mut self, layout: Layout) -> Result<NonNull<u8>, AllocError>
{
    let size = layout.size();
    match self.select_allocator(&layout) {
        AllocType::Slab(i) => {
            let ret = self.slabs[i].alloc();
            if ret.is_ok() {
                self.user += size;
                self.allocated += self.slabs[i].block_size;
            }
            ret
        }
        AllocType::Fallback => {
            let ret = self.fallback.allocate_first_fit(layout);
            if ret.is_ok() {
                self.user += size;
                self.allocated += size;
                Ok(ret.unwrap())
            } else {
                Err(AllocError)
            }
        }
    }
}
```

alloc 方法根据用户请求的内存大小选择合适的分配器，并调用相应的分配方法进行内存分配。如果分配成功，则更新用户请求的字节数和实际分配的字节数。分配操作会返回一个非空的内存指针，或者返回分配错误。

## 释放

和分配类似，堆分配器的释放方法也是一个简单的释放器选择方法，它根据用户请求的内存大小选择合适的 Slab 分配器或 Linked List 分配器进行内存释放。



```
pub fn dealloc(&mut self, ptr: NonNull<u8>, layout: Layout) {
    let size = layout.size();
    match self.select_allocator(&layout) {
        AllocType::Slab(i) => {
            self.slabs[i].dealloc(ptr);
            self.user -= size;
            self.allocated -= self.slabs[i].block_size;
        }
        AllocType::Fallback => {
            unsafe { self.fallback.deallocate(ptr, layout) };
            self.user -= size;
            self.allocated -= size;
        }
    }
}
```

`dealloc` 方法根据用户请求的内存大小选择合适的分配器，并调用相应的释放方法进行内存释放。如果释放成功，则更新用户请求的字节数和实际分配的字节数。

## 互斥的堆内存分配器

堆内存分配器不只运行在内核态，也运行在用户态。在用户态，堆内存分配器需要支持并发环境，因此需要加锁以保证线程安全。

`LockedHeap` 是一个对 `Heap` 的封装，它在堆内存分配器的基础上增加了互斥锁，以保证堆内存分配器的线程安全性。

```
#[cfg(feature = "use_spin")]
pub struct LockedHeap(Mutex<Heap>);
```

在使用 `LockedHeap` 时，需要先使用 `.lock()` 方法获取互斥锁，然后再调用堆内存分配器的方法。

## 注册

堆内存分配器需要在内核启动时注册，以便内核和用户程序能够使用堆内存分配器。

注册堆内存分配器首先需要为其实现 `Rust alloc` 库的 `GlobalAlloc` trait，这样，`Rust` 的内存分配器就能够使用堆内存分配器进行内存分配，从而可以在内核和用户程序中使用堆内存分配器及构建在其之上的动态数据结构，如 `Vec`、`Box` 等。

```
#[cfg(feature = "use_spin")]
unsafe impl alloc::GlobalAlloc for LockedHeap {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        self.0.lock().alloc(layout).ok()
            .map_or(core::ptr::null_mut(), |allocation| allocation.as_ptr
            ())
    }
    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout) {
        self.0.lock()
            .dealloc(unsafe { NonNull::new_unchecked(ptr) }, layout)
    }
}
```

实现时，只需要简单地调用堆内存分配器的 `alloc` 和 `dealloc` 方法即可。

实现该 `trait` 后，即可向 Rust 的内存分配器注册堆内存分配器，使得 Rust 的内存分配器能够使用堆内存分配器进行内存分配。

我们需要定义一个全局的 `HEAP\_ALLOCATOR` 静态变量，通过 `#[global\_allocator]` 属性将其注册为 Rust 的全局内存分配器。

最后，将一块静态内存空间划分给堆内存分配器，并初始化堆内存分配器。

```
``rust
#[global_allocator]
static HEAP_ALLOCATOR: LockedHeap = LockedHeap::empty();
static mut HEAP: [u8; HEAP_SIZE] = [0; HEAP_SIZE];

#[allow(static_mut_refs)]
/// 初始化内核堆内存分配器
pub fn init_heap() {
    unsafe {
        HEAP_ALLOCATOR.lock()
            .init(HEAP.as_ptr() as usize, HEAP_SIZE);
    }
}
```

同时，我们也需要处理分配失败的情况，例如内存不足等。使用 `#[alloc_error_handler]` 属性，我们可以定义一个全局的内存分配错误处理函数，当内存分配失败时，Rust 的内存分配器会调用该函数。

```
#[alloc_error_handler]
fn alloc_error_handler(layout: core::alloc::Layout) -> ! {
    panic!("Heap allocation error: {:?}", layout)
}
```

## 小结

堆分配器是一个管理堆内存的模块，负责分配和释放内存。堆分配器通过 Slab 分配器和 Linked List 分配器实现内存的动态分配，提高内存利用率和灵活性。

在本节中，我们详细介绍了堆分配器的设计与实现，包括 Slab 分配器的设计思想、核心功能和代码实现。通过堆分配器，操作系统可以支持动态内存分配，为内核和用户程序提供强大的内存管理能力。

## 存储管理/物理页帧

### 简介

物理页帧管理是操作系统存储管理模块中的一个核心功能。它负责对物理内存进行精细化管理，通过页帧分配器为系统内核和用户程序提供物理内存支持。

页帧管理的任务主要包括：

- 物理内存的划分和标记。
- 页帧的高效分配与回收。
- 确保内存分配过程中的线程安全。

在本项目中，物理页帧管理由 `src/mm/frame_allocator.rs` 文件实现。主要使用栈式分配器（Stack Allocator）来管理物理页帧。以下内容将详细阐述其设计和实现。

### 页帧的基本概念

#### 1. 什么是页帧？

- 物理内存被划分为大小固定的块，这些块称为 **页帧（Page Frame）**。
- 页帧是内存管理的基本单位。在本系统中，页帧的大小通常为 4KB。

#### 2. 页帧管理的核心目标

- **高效性**：快速分配和回收页帧。
- **线程安全**：在多任务环境中，保证页帧分配操作的安全性。
- **低碎片率**：尽量减少内存碎片的产生。

### 页帧管理的实现

#### 1. 栈帧分配器

栈帧分配器是一种常见的页帧分配器，它通过栈的方式管理未使用的页帧。

在栈帧分配器中，未使用的物理页帧范围是一个左闭右开的区间，表示从未使用的物理页帧的起始页号到结束页号的范围。

释放帧栈维护了一个已释放的物理页帧列表，当需要分配物理页帧时，首先从已释放的物理页帧列表中弹出一个物理页帧，如果列表为空，则从未使用的物理页帧范围中分配一个物理页帧。

```
pub struct StackFrameAllocator {
    /// 未使用的物理页帧范围。
    unused: Range<PhysicalPageNumber>,
    /// 已释放的物理页帧列表。
    freed: Vec<PhysicalPageNumber>,
}
```

## 2. 初始化

在本节中，将详细介绍如何初始化栈帧分配器，并使其准备好进行页帧的分配与释放操作。

### 栈帧分配器的初始化

栈帧分配器的初始化过程主要包括两个步骤：

1. **设置未使用的物理页帧范围**：将未使用的物理页号范围设置为传入的范围。
2. **初始化全局栈帧分配器**：将未使用的物理页号范围设置为从内核结束地址到内存结束地址的范围。

以下是栈帧分配器初始化的代码实现：

```
impl StackFrameAllocator {
    /// 初始化栈帧分配器
    ///
    /// 将未使用的物理页号范围设置为传入的范围
    pub fn init(&mut self, range: Range<PhysicalPageNumber>) {
        self.unused = range;
    }

    /// 初始化全局栈帧分配器
    ///
    /// 将未使用的物理页号范围设置为从内核结束地址到内存结束地址的范围
    pub fn init_frame_allocator() {
        // 获取内核结束地址，并向上取整到页边界
        let start = PhysicalAddress::from(extern_global!(__kernel_end) as u
size)
            .ceil_page()
            .into();
        // 获取内存结束地址，并向下取整到页边界
        let end = PhysicalAddress::from(MEMORY_END).floor_page().into();
        FRAME_ALLOCATOR.ref_cell.borrow_mut().init(start..end);
    }
}
```

### 帧分配和释放

在栈帧分配器初始化之后，可以通过以下两个函数进行帧的分配和释放：

1. **alloc\_frame**: 从全局栈帧分配器分配一个物理页帧，并返回一个 FrameTracker 实例。
2. **dealloc\_frame**: 从全局栈帧分配器释放一个物理页帧。

以下是代码实现：

```
impl StackFrameAllocator {
    pub fn alloc_frame() -> Option<FrameTracker> {
        FRAME_ALLOCATOR
            .ref_cell
            .borrow_mut()
            .alloc()
            .map(|frame| FrameTracker::new(frame))
    }

    pub fn dealloc_frame(frame: PhysicalPageNumber) {
        FRAME_ALLOCATOR.ref_cell.borrow_mut().dealloc(frame);
    }
}
```

通过上述实现，栈帧分配器能够有效地管理物理页帧，为系统内核和用户程序提供可靠的物理内存支持。

好的，我们来详细看看 FrameTracker 的设计和实现。

## FrameTracker

FrameTracker 是一个用于追踪物理页帧分配和释放的结构体。每个 FrameTracker 实例都会追踪一个物理页帧，并在实例被销毁时自动释放该物理页帧。这种设计确保了内存资源的安全管理，防止内存泄漏。

FrameTracker 结构体的定义如下：

```
#[derive(Debug)]
pub struct FrameTracker {
    pub frame: PhysicalPageNumber,
}
```

这个结构体包含一个 PhysicalPageNumber 类型的字段 frame，用于存储被追踪的物理页帧编号。

FrameTracker 提供了一个 new 函数，用于创建新的实例：

```
impl FrameTracker {
    pub fn new(frame: PhysicalPageNumber) -> Self {
        // 清零帧中的所有字节
        frame.get_bytes_array().iter_mut().for_each(|x| *x = 0);
        Self { frame }
    }
}
```

在 new 函数中，我们首先将帧中的所有字节清零，然后返回一个包含该帧的新 FrameTracker 实例。

为了确保在 `FrameTracker` 实例被销毁时自动释放物理页帧，我们实现了 `Drop trait`：

```
impl Drop for FrameTracker {
    /// 当 FrameTracker 实例被销毁时，自动释放物理页帧
    fn drop(&mut self) {
        StackFrameAllocator::dealloc_frame(self.frame);
    }
}
```

通过实现 `Drop trait`，当 `FrameTracker` 实例离开其作用域时，`drop` 方法会被自动调用，从而释放被追踪的物理页帧。

## 小结

物理页帧管理器通过位图数据结构高效地管理物理内存，提供了简单且可靠的页帧分配与回收接口。它是存储管理模块的重要基础，为操作系统的内存管理提供了强有力的支持。

# 存储管理/多级页表

## 简介

多级页表管理是现代操作系统存储管理模块的重要组成部分，负责实现虚拟地址到物理地址的高效映射。在 RISC-V 架构下，SV39 分页机制采用三级页表结构，支持灵活的地址转换和权限控制。本项目中的多级页表管理由 `src/mm/page_table.rs` 文件实现。

本部分将通过理论介绍与代码解析相结合的方式，详细阐述多级页表的设计与实现，包括页表项的定义、页表的结构以及核心操作接口。

## 多级页表的基础概念

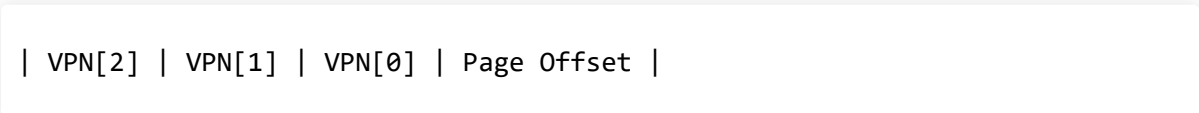
### 1. 页表的作用

页表是一种数据结构，用于存储虚拟地址到物理地址的映射关系。在多级页表中，虚拟地址解析通过逐级查表完成。页表不仅支持地址映射，还能通过权限位实现对地址访问的控制。

### 2. SV39 分页机制

SV39 分页机制是 RISC-V 架构下的一种页表设计，支持 39 位虚拟地址空间。它通过在 `satp` (Supervisor Address Translation and Protection, 监管者地址转换和保护寄存器) 寄存器中设置根页表的物理页号 (PPN)，并将模式位 (第 60-63 位) 设置为 8，启用页表。

RISC-V 的 SV39 模式采用三级页表，虚拟地址的结构如下：



- **VPN[2], VPN[1], VPN[0]**：三级虚拟页号，每级占 9 位，共 27 位。

- **Page Offset**: 页内偏移, 占 12 位。
- **虚拟地址长度**: 39 位, 一个虚拟地址占用 64 位, 高 25 位未使用。

### 3. 地址解析过程

地址解析的基本流程:

1. 从 satp 寄存器中获取根页表的物理页号 (PPN)。
2. 通过虚拟地址的 VPN 逐级查表, 获取页表项。
  1. 第一级页表: VPN[2] -> 二级页表。
  2. 第二级页表: VPN[1] -> 三级页表。
  3. 第三级页表: VPN[0] -> 物理页号 (PPN)。
4. 通过页内偏移量获取物理地址。
5. 检查权限位, 判断是否有权限访问。

这一过程在硬件中由 MMU (Memory Management Unit, 内存管理单元) 完成, 并且使用 TLB (Translation Lookaside Buffer, 快表) 来加速地址解析。

同时, 作为操作系统内核, 我们也需要实现同样的地址解析过程, 以便管理虚拟地址空间。

### 4. 页表项 (PTE)

页表项记录了一个虚拟页号对应的物理页帧信息, 包括:

- **物理页号 (PPN)**: 指向下一级页表或物理页帧。
- **权限位 (Flags)**: 读、写、执行权限等。
  - RSW: 保留位。
  - D: 脏位, 表示页面是否被修改。
  - A: 访问位, 表示页面是否被访问。
  - G: 全局位, 表示页面是否全局有效。
  - U: 用户位, 表示页面是否为用户态可访问。
  - X: 执行位, 表示页面是否可执行。
  - W: 写位, 表示页面是否可写。
  - R: 读位, 表示页面是否可读。
  - V: 有效位, 表示是否有效。

页表项的格式:

<b>63-54</b>	<b>53-10</b>	<b>9-8</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>
保留位	物理页号	RSW	D	A	G	U	X	W	R	V

## 页表的核心数据结构

### 页表项 (PTE)

在代码中, 页表项由 PageTableEntry 结构表示:

```
#[repr(C)]
pub struct PageTableEntry {
    pub bits: usize, // 页表项的位字段
}
```

### 1. 创建新的页表项

通过指定物理页号（PPN）和权限标志（Flags），创建一个新的页表项：

```
pub fn new(phy_page: PhysicalPageNumber, flags: PTEFlags) -> Self {
    Self {
        bits: (phy_page.0 << 10) | flags.bits() as usize,
    }
}
```

### 2. 提取物理页号（PPN）

根据页表项的位字段创建物理页号

```
impl From<&PageTableEntry> for PhysicalPageNumber {
    fn from(pte: &PageTableEntry) -> Self {
        PhysicalPageNumber::from(pte.bits >> 10 & ((1 << PHYSICAL_PAGE_NUMBER_WIDTH_SV39) - 1))
    }
}
```

### 3. 提取权限标志（Flags）

从页表项中提取权限标志：

```
impl From<&PageTableEntry> for PTEFlags {
    fn from(pte: &PageTableEntry) -> Self {
        Self::from_bits_truncate(pte.bits as u8)
    }
}
```

## 页表（PageTable）

PageTable 结构体和其实现提供了一个页表的抽象，用于管理虚拟内存到物理内存的映射。以下是其设计的详细说明：

```
pub struct PageTable {
    pub root: PhysicalPageNumber,
    pub frames: Vec<FrameTracker>,
}
```



- root: 根页表的物理页号。
- frames: 帧跟踪器的向量，用于跟踪分配的帧。

## 页表的核心方法

### 1. 创建新的页表

创建一个新的页表，将根页表的物理页号设置为 root，并初始化帧跟踪器。

```
pub fn new() -> Self {
    let root = StackFrameAllocator::alloc_frame().unwrap();
    Self {
        root: root.frame,
        frames: alloc::vec![root],
    }
}
```

### 2. 获取页表对应的 satp 寄存器值

获取页表对应的 satp 寄存器值，用于设置根页表。

在获取到根页表的物理页号后，还需要再加上模式位 ( $8 \ll 60$ ) 以启用分页机制。

```
pub fn token(&self) -> usize {
    self.root.0 | (8 << 60)
}
```

### 3. 映射虚拟地址到物理地址

根据虚拟页号查找页表项，并将其映射到给定的物理页号和标志位

```
pub fn map(&mut self, vpn: VirtualPageNumber, ppn: PhysicalPageNumber, flags: PTEFlags) {
    let entry = self.find_pte_mut(vpn).unwrap();
    if (*entry).valid() {
        panic!("map: {:?} has been mapped", vpn);
    }
    *entry = PageTableEntry::new(ppn, flags | PTEFlags::Valid);
}
```

### 4. 取消映射

取消虚拟地址到物理地址的映射

```
pub fn unmap(&mut self, vpn: VirtualPageNumber) {  
    let entry = self.find_pte(vpn).unwrap();  
    if !(*entry).valid() {  
        panic!("unmap: {:?} has not been mapped", vpn);  
    }  
    *entry = PageTableEntry::default();  
}
```

## 5. 查找页表项

在查找页表项时，需要逐级查表，获取对应的页表项。

同时，对于不同情况下的查表操作，我们需要进行不同的处理：

在映射时，如果页表项不存在，需要分配新的页表项；

在其他情况下，如果页表项不存在，应当返回 None。

由于前者涉及到对页表对象的修改，为了避免借用检查器的限制，我们将这两种情况分开实现。

```

pub fn find_pte_mut(&mut self, vpn: VirtualPageNumber) -> Option<&mut PageT
ableEntry> {
    let indexes = vpn.indexes();
    let mut ppn = self.root;
    for i in 0..3 {
        let entry = &mut ppn.get_pte_array()[indexes[i]];
        if i == 2 {
            return Some(entry);
        }
        if !entry.valid() {
            let frame = StackFrameAllocator::alloc_frame().unwrap();
            *entry = PageTableEntry::new(frame.frame, PTEFlags::Valid);
            self.frames.push(frame);
        }
        ppn = PhysicalPageNumber::from(entry);
    }
    None
}

```

这里，我们首先获取虚拟页号的三级索引，然后逐级查表，直到找到页表项或者遇到空项时分配新的页表项。

对于只读的查表操作，我们可以通过 `find\_pte` 方法实现，其与 `find\_pte\_mut` 类似，但在 `!entry.valid()` 时直接返回 `None`。

#### ##### 6. 翻译虚拟页号到页表项和物理地址

在查找页表项的基础上，我们可以实现虚拟页号到页表项和物理地址的翻译操作：

```

```rust
pub fn translate(&self, vpn: VirtualPageNumber) -> Option<PageTableEntry> {
    self.find_pte(vpn).map(|entry| *entry)
}

/// 根据虚拟地址查找页表项，并计算物理地址
pub fn translate_addr(&self, va: VirtualAddress) -> Option<PhysicalAddress>
> {
    self.find_pte(VirtualPageNumber::from(va)).map(|entry| {
        let floor = PhysicalAddress::from(PhysicalPageNumber::from(entry));
        let offset = va.page_offset();
        PhysicalAddress::from(usize::from(floor) + offset)
    })
}

```

## 内核态访问用户态地址

在启动分页机制后，内核态无法直接访问用户态的地址，因为用户态地址可能会被映射到不同的物理页帧中。

为了解决这个问题，我们需要在内核态访问用户态地址时，进行地址转换。

我们需要实现一个函数，用于将用户态地址翻译为内核态地址，以便内核可以访问用户态的数据。

```
fn get_translated_slices<T>(  
    token: usize,  
    ptr: *const u8,  
    len: usize,  
    get_slice: impl Fn(&PhysicalPageNumber, usize, usize) -> T,  
) -> Vec<T> {  
    let page_table = PageTable::from_satp(token);  
    let mut start = ptr as usize;  
    let end = start + len;  
    let mut slices = Vec::new();  
    while start < end {  
        let start_va = VirtualAddress::from(start);  
        let vpn = start_va.floor_page();  
        let ppn = PhysicalPageNumber::from(&page_table.translate(vpn).unwrapp());  
        let end_va = VirtualAddress::from(vpn + VirtualPageNumber(1));  
        let end_va = end_va.min(VirtualAddress::from(end));  
        let slice = get_slice(&ppn, start_va.page_offset(), end_va.page_offset());  
        slices.push(slice);  
        start = usize::from(end_va);  
    }  
    slices  
}
```

get\_translated\_slices 函数接受页表的 token、指针和长度，以及一个闭包函数 get\_slice，用于根据物理页号、起始偏移和结束偏移获取切片。

内核可以通过该函数将用户地址空间区域翻译为内核地址空间的切片，以便访问用户态的数据。

在函数内部，我们首先创建一个页表对象，然后根据指针和长度，将用户地址空间区域划分为多个页，逐页进行翻译。

对于每个页，我们获取其虚拟页号，然后通过页表查找获取物理页号，再根据起始偏移和结束偏移获取切片。

最后，我们将翻译后的切片收集到一个向量中并返回。

类似地，我们还可以翻译用户地址空间中的字符串：

```
pub fn get_translated_string(token: usize, ptr: *const u8) -> String {
    let page_table = PageTable::from_satp(token);
    let mut string = String::new();
    let mut start = ptr as usize;
    loop {
        let ch: u8 = *(page_table
            .translate_addr(VirtualAddress::from(start))
            .unwrap())
            .get_mut();
        if ch == 0 {
            return string;
        } else {
            string.push(ch as char);
            start += 1;
        }
    }
}
```

在这个函数中，我们首先创建一个页表对象，然后根据指针逐字节获取用户地址空间中的字符，直到遇到空字符（'\0'）为止。

最后，我们将获取到的字符拼接成字符串并返回。

## 小结

多级页表管理通过 `page_table.rs` 文件中的核心数据结构和方法，完整实现了虚拟地址到物理地址的映射机制。它是存储管理模块的基础，为虚拟内存管理提供了高效和灵活的支持。

在接下来的部分，我们将详细阐述 **地址空间管理** 的实现，探讨如何组织和管理逻辑段与虚拟地址空间。

## 存储管理/地址空间

### 简介

地址空间管理是操作系统内存管理中的核心功能，用于将虚拟地址映射到物理地址，并提供灵活的逻辑段管理。每个地址空间包含多个逻辑段（Segment），用于组织程序的代码、数据、堆、栈等不同用途的内存区域。

逻辑段和地址空间的设计分别对应于段表和页表的机制，通过多级页表实现虚拟地址到物理地址的映射，提供了内存隔离、内存保护和内存共享等功能，在页表的基础上，又增加了逻辑段的概念，使得内存管理更加灵活。二者的结合，构成了段页式内存管理的基本框架。

在本项目中，地址空间管理由 `src/mm/memory_set.rs` 文件实现，核心功能包括：

- 定义逻辑段。
- 管理虚拟地址到物理地址的映射。
- 支持动态调整地址空间的布局。

本节将通过理论介绍与代码解析相结合的方式，详细阐述地址空间的设计与实现。

## 基本概念

### 结构

地址空间由多个逻辑段组成，每个逻辑段包含以下关键信息：

- **起始地址与结束地址**：定义逻辑段在虚拟地址空间中的范围。
- **权限**：控制逻辑段的访问属性，如可读、可写、可执行。
- **映射类型**：定义逻辑段如何与物理内存关联（直接映射或分配页帧）。

地址空间结构在操作系统中的主要用途包括：

- 将用户程序的代码、数据、堆和栈等分配到独立的虚拟地址空间。
- 提供对内核和用户地址空间的隔离，确保系统安全性。

### 组成部分

1. **逻辑段（MapArea）** 逻辑段管理了一段虚拟内存，提供映射和访存权限控制
2. **地址空间（MemorySet）** 地址空间包含了一组虚拟内存，基于三级页表实现了虚拟内存到物理内存的映射

## 核心数据结构

### 逻辑段（MapArea）

逻辑段是段表机制在地址空间管理中的虚拟逻辑化的体现，用于管理一段虚拟内存空间。

段的划分由应用程序和内核的开发者根据需求进行，典型的段包括：

- 代码段（.text）：存放程序的指令代码。
- 数据段（.data）：存放程序的全局变量。
- 只读数据段（.rodata）：存放只读数据。
- 未初始化数据段（.bss）：存放未初始化的全局变量。
- 堆（heap）：动态分配的内存空间。
- ...

对于每个段，可以设置不同的访问权限，如可读、可写、可执行等。例如，代码段（.text）通常是只读、可执行的。

在未启用分页机制的情况下，逻辑段直接映射到物理内存，开发者可以直接访问物理地址。

启动分页机制后，段需要通过页表进行映射，开发者访问的是虚拟地址，由操作系统内核负责将虚拟地址映射到物理地址，实现内存隔离和保护。

逻辑段由 MapArea 结构体表示：

```
pub struct Segment {
    vpn_range: VPNRange,
    data: BTreeMap<VirtualPageNumber, FrameTracker>,
    map_type: MapType,
    permission: MapPermission,
}
```

- `vpn_range`: 虚拟页号范围，表示逻辑段的起始和结束页号。
- `data`: 虚拟页号到物理页号的映射表。
- `map_type`: 映射类型，用于区分直接映射和页帧映射。
- `permission`: 权限标志，控制逻辑段的访问权限。

映射类型（`MapType`）定义了逻辑段与物理内存的关联方式：

- `Linear`: 线性恒等映射，逻辑页号和物理页号呈线性一一对应。
- `Framed`: 分配页帧映射，逻辑页号映射到分配的页帧。

在此内核实现中，内核内存使用线性映射，用户内存使用分配页帧映射。

逻辑段中的权限标志可以使用 `bitflags` 宏定义权限标志：

```
bitflags! {
    pub struct MapPermission: u8 {
        const Read = 1 << 1;
        const Write = 1 << 2;
        const Execute = 1 << 3;
        const User = 1 << 4;
    }
}
```

逻辑段的权限包括可读、可写、可执行和用户态权限。通过对 `MapPermission` 进行组合，可以实现不同的权限控制。

## 地址空间（MemorySet）

地址空间是操作系统中用于管理虚拟地址空间的数据结构，包含了多个逻辑段和一个多级页表。

对于每个程序（包括用户态程序和内核），都有一个独立的地址空间，用于管理程序的代码、数据、堆和栈等内存区域。对于用户态程序来说，其他的地址空间对其是不可见的，可以视为其占有整个虚拟地址空间。

地址空间由 `MemorySet` 结构体表示：

```
pub struct MemorySet {
    pub page_table: PageTable,
    pub areas: Vec<MapArea>,
}
```

- `page_table`: 多级页表，用于管理虚拟地址到物理地址的映射。

- `areas`: 逻辑段列表, 存储当前地址空间的所有逻辑段。

## 核心功能

### 1. 创建地址空间

#### 初始化空地址空间

创建一个新的地址空间, 初始化页表和逻辑段列表:

```
pub fn empty() -> Self {  
    Self {  
        page_table: PageTable::new(),  
        areas: Vec::new(),  
    }  
}
```

#### 创建内核地址空间

在内核启动时, 需要创建内核地址空间, 将内核的代码、数据和堆区映射到内核的虚拟地址空间中。

在创建内核地址空间时, 需要完成以下工作:

1. 创建内核页表;
2. 映射跳板页 `trampoline`;
3. 映射内核代码段 `.text`;
4. 映射内核数据段 `.data` 和 `.rodata`;
5. 映射内核堆区 `.bss`;
6. 映射剩余的堆区;
7. 映射内存映射IO区域, 实现 DMA。

可以在编译链接时通过链接脚本暴露内核的代码段、数据段和堆区的起始地址和结束地址, 以便在内核启动时进行映射。

内核地址空间的映射使用线性恒等映射, 即将虚拟地址和物理地址一一对应, 这样可以简化内核的地址转换过程。映射时, 从相应的内存段创建逻辑段, 设置相应的权限和属性, 然后将逻辑段加入到地址空间中。



```
memory_set.push(
    MapArea::new(
        VPNRange::from_addr(
            VirtualAddress::from(section_start),
            VirtualAddress::from(section_end),
        ),
        MapType::Linear,
        MapPermission::Read | MapPermission::Execute,
    ),
    None,
);
```

### 创建用户地址空间

在执行用户程序前，需要为用户程序创建一个独立的地址空间。在创建用户地址空间时，需要完成以下工作：

1. 解析并校验 ELF 文件，获取用户程序的代码段、数据段和堆区的起始地址和结束地址。
2. 映射跳板页 trampoline。
3. 映射 ELF 文件中定义的用户程序的每一个段，包括代码段、数据段等。在映射时，通过 ELF 文件中为每个段设置的权限和属性，创建相应的逻辑段，然后将逻辑段加入到地址空间中。
4. 建立用户栈，映射用户栈区。

首先，我们需要使用 `xmas-elf` 库从自文件系统加载的 ELF 文件字节数据中解析出 ELF 文件头和程序头表，然后根据程序头表中的信息，创建用户地址空间。

```
let elf = ElfFile::new(elf).unwrap();
let elf_header = elf.header;
let elf_magic = elf_header.pt1.magic;
assert_eq!(elf_magic, [0x7f, 0x45, 0x4c, 0x46], "invalid elf file");
let ph_count = elf_header.pt2.ph_count();
```

`ph_count` 表示 ELF 文件中的程序头表的数量，我们需要遍历程序头表，获取每个程序头表的信息，然后根据程序头表的类型和大小，创建相应的逻辑段。

```
let ph = elf.program_header(i).unwrap();
if ph.get_type().unwrap() != xmas_elf::program::Type::Load {
    continue;
}
```

首先获取虚拟地址范围：

```
let vpn_start = VirtualAddress::from(ph.virtual_addr() as usize).floor_page();
let vpn_end = VirtualAddress::from(ph.virtual_addr() as usize + ph.mem_size() as usize)
    .ceil_page();
let vpn_range = VPNRange::new(vpn_start, vpn_end);
```

对于用户程序地址空间，我们使用分配页帧映射的方式，将用户程序的代码段、数据段和堆区映射到分配的页帧上。

通过 `ph.flags().is_read()`、`ph.flags().is_write()` 和 `ph.flags().is_execute()` 获取程序头表的权限信息，然后根据权限信息创建逻辑段。此外，用户态程序至少需要用户权限。

有了这些信息后，我们就可以创建逻辑段了：

```
let map_type = MapType::Framed;
let permission = {
    let mut flags = MapPermission::User;
    if ph.flags().is_read() {
        flags |= MapPermission::Read;
    }
    if ph.flags().is_write() {
        flags |= MapPermission::Write;
    }
    if ph.flags().is_execute() {
        flags |= MapPermission::Execute;
    }
    flags
};
let map_area = MapArea::new(vpn_range, map_type, permission);
```

最后，将逻辑段加入到地址空间中：

```
max_end = map_area.vpn_range.end();
let data = Some(&elf.input[ph.offset() as usize..][..ph.file_size() as usize]);
memory_set.push(map_area, data);
```

## 映射跳板页

跳板页是用户态和内核态之间的中转页，用于实现用户态到内核态的切换。用户态程序和操作系统内核都需要映射跳板页，以便在用户态和内核态之间进行切换。

其实际代码存放在内核代码段的 `trampoline` 节区中：

```
.text : {
    *(.text.entry)
    . = ALIGN(4K);
    __strampoline = .;
    *(.text.trampoline);
    . = ALIGN(4K);
    *(.text .text.*)
}
```

跳板代码即陷入保存和恢复的代码，用于实现用户态到内核态的切换。

在映射时，将跳板页映射到内核地址空间的虚拟地址 TRAMPOLINE 处。TRAMPOLINE 值设置为 `usize::MAX - PAGE_SIZE_SV39 + 1`，即指定跳板代码映射到地址空间的最高地址。

## 2. 克隆地址空间

用户程序在 `clone` (即 `fork`) 时，需要创建一个新的地址空间，在最开始，子进程的地址空间与父进程的地址空间相同，但是地址空间的内容是独立的。

在 Linux 中，父子进程的地址空间是通过 `copy-on-write` 机制实现的，即父子进程共享同一个物理页帧，只有在其中一个进程尝试写入时，才会复制物理页帧。

这里，为了简化实现，我们将创建一个新的地址空间，然后将遍历父进程的逻辑段，将父进程逻辑段对应的物理页帧复制到子进程逻辑段对应的物理页帧中。

```
pub fn from_existed_user(user_space: &MemorySet) -> Self {
    let mut memory_set = MemorySet::empty();
    memory_set.map_trampoline();
    for area in user_space.areas.iter() {
        let new_area = MapArea::from_another(area);
        memory_set.push(new_area, None);
        for vpn in area.vpn_range {
            let src_ppn = PhysicalPageNumber::from(&user_space.translate(vpn).unwrap());
            let dst_ppn = PhysicalPageNumber::from(&memory_set.translate(vpn).unwrap());
            dst_ppn
                .get_bytes_array()
                .copy_from_slice(src_ppn.get_bytes_array());
        }
    }
    memory_set
}
```

## 3. 添加逻辑段

通过 `push` 方法向地址空间中添加逻辑段，并建立虚拟地址到物理地址的映射：

```
pub fn push(&mut self, mut area: MapArea, data: Option<&[u8]>) {
    area.map(&mut self.page_table);
    if let Some(data) = data {
        area.copy_from(&mut self.page_table, data);
    }
    self.areas.push(area);
}
```

这里，`area.map` 方法将逻辑段映射到页表中，`area.copy_from` 方法将数据复制到物理页帧中。

最后，将逻辑段加入到地址空间的逻辑段列表中。

#### 4. 移除逻辑段

通过 `remove_area` 方法从地址空间中移除逻辑段，并解除虚拟地址到物理地址的映射：

```
pub fn remove_area(&mut self, start_vpn: VirtualPageNumber) {
    let mut index = None;
    for (i, area) in self.areas.iter().enumerate() {
        if area.vpn_range.start == start_vpn {
            index = Some(i);
            break;
        }
    }
    if let Some(index) = index {
        let mut area = self.areas.remove(index);
        area.unmap(&mut self.page_table);
    }
}
```

首先，遍历地址空间中的逻辑段列表，找到需要移除的逻辑段，然后调用 `area.unmap` 方法解除逻辑段的映射。

#### 5. 激活地址空间

在切换进程或线程时，需要激活新的地址空间，将新的地址空间的页表加载到 `satp` 寄存器中，实现地址空间的切换。

```
pub fn activate(&self) {
    let satp = self.page_table.token();
    unsafe {
        sstatus::set_sum();
        satp::write(satp);
        asm!("sfence.vma"); // 刷新 TLB
    }
}
```

在激活地址空间时，首先获取页表的 `satp` 寄存器值，然后将 `satp` 寄存器值写入 `satp` 寄存器，最后使用 `sfence.vma` 指令刷新 TLB 缓存。

在激活地址空间时，需要设置 SUM 位，以允许内核态访问用户态内存。

## 小结

地址空间管理通过逻辑段与多级页表的结合，实现了虚拟地址空间的灵活组织与高效管理。它为内核和用户进程提供了强大的内存管理支持，是操作系统存储管理模块的重要组成部分。

## 存储管理模块总结

### 模块的重要性

存储管理模块是操作系统内核的核心组成部分，负责管理物理内存、虚拟地址空间以及动态内存分配。它通过模块化和分层设计实现了复杂功能的清晰分工，为操作系统提供了高效、稳定和灵活的存储支持。

本项目的存储管理模块涵盖了以下关键部分：

- **地址抽象与管理**：提供虚拟地址和物理地址的抽象与操作方法，构建了整个内存管理的基础。
- **物理页帧管理**：使用位图数据结构高效跟踪和管理物理页帧，提供了可靠的分配和回收机制。
- **多级页表管理**：基于 RISC-V SV39 分页机制，支持虚拟地址到物理地址的高效映射和权限控制。
- **地址空间管理**：通过逻辑段和多级页表的结合，实现虚拟地址空间的灵活组织与管理。
- **堆分配器设计**：基于 Slab 和 LinkedList 分配器，实现了高效、低碎片率的动态内存分配。

### 核心设计特点

#### 1. 模块化设计

- 各子模块（如地址管理、页表管理、地址空间管理）独立实现，易于扩展和维护。
- 提供统一的接口，实现内存管理功能的松耦合。

#### 2. 高效性

- 使用位图和多级页表等高效的数据结构，优化了内存管理的性能。
- 动态内存分配的线性分配策略保证了分配的快速性。

#### 3. 安全性

- 地址空间的权限控制和进程间的内存隔离确保了系统的稳定性和安全性。

#### 4. 灵活性

- 地址空间支持动态调整，满足不同任务对内存的需求。
- 堆分配器为内核提供了按需分配的能力，适应运行时的多样化需求。

## 结束语

存储管理模块为操作系统的正常运行提供了强有力的支持，是操作系统开发的核心环节之一。本项目的实现涵盖了从地址抽象到动态分配的完整功能链，为进一步的学习和开发奠定了坚实的基础。

础。

通过模块化设计和代码实践，我们不仅能够更好地理解存储管理的内部机制，还可以为实际操作系统的开发积累宝贵经验因而后续我们还将继续在本基础上进行改善。

# 进程管理模块

## 进程管理模块概述

### 进程管理模块的定义与作用

进程管理模块是操作系统内核中至关重要的一个部分，它主要负责进程的创建、销毁、调度、状态转换以及进程间的通信等一系列操作。其核心目标是确保多个进程能够在计算机系统中高效、有序地运行，合理地共享系统资源。

### 主要功能组成部分

#### 进程创建与销毁

- 创建功能

1. 分配 PID

从全局的 PID 分配器获取一个尚未使用的 PID 分配给新进程，确保每个进程都有唯一的标识符。

2. 初始化内存空间

为进程分配内存，构建相应的内存映射关系，涉及从磁盘加载程序代码和数据到内存、设置堆、栈等内存区域的初始状态等操作。

3. 设置初始状态

将进程状态初始化为就绪（Ready）状态，表示其已准备好被调度执行；同时初始化程序计数器指向程序的入口地址，寄存器等执行相关信息设置为默认值（清空），并初始化栈空间、文件描述符表（继承父进程部分或者初始化为标准输入输出对应的描述符）。

4. 关联父子进程关系

如果通过父进程使用 `fork` 创建子进程，则需要在父进程和子进程的相关字段中记录彼此的关系，在子进程中记录父进程的引用，在父进程中将子进程添加到其子进程列表中。

- 销毁功能

1. 资源回收

释放进程所占用的各种资源，包括内存空间、关闭打开的文件，并将其占用的其他系统资源进行清理回收，避免资源泄漏。

2. 更新父子进程关系

通知父进程子进程已经终止，若父进程正在等待子进程结束，唤醒父进程进行相应的处理；同时将子进程从父进程的子进程列表中移除。

### 3. PCB 释放

最后将进程对应的 PCB 结构体所占用的内存空间释放掉，完成整个进程的终止流程，使其彻底从系统中移除。

## 进程调度

- **调度策略选择**

我们采用先来先服务+时间片轮转的思想，选择下一个执行进程。

- **就绪队列管理**

维护就绪队列，用于存放处于就绪状态的进程，当有进程进入就绪状态时将其添加到相应队列中，调度器从这些队列中按照调度策略选取进程运行。

- **进程切换操作**

当需要进行进程切换时（当前运行进程的时间片用完），需要保存当前正在运行进程的上下文信息到其对应的 PCB 中，然后从就绪队列中选取下一个要执行的进程，将该进程的上下文信息恢复到硬件寄存器等相关位置，使得下一个进程可以开始执行，这个切换过程我们借助底层的汇编代码实现，确保切换的正确性和高效性。

- **调度器实现**

调度器是进程调度功能的核心实现部分。根据先来先服务+时间片轮转的调度策略，从存放所有准备好运行的进程的就绪队列中选择合适的进程，将其状态转换为运行状态，并将 CPU 控制权交给这个进程。调度器还需要考虑时间片剩余情况，以优化系统的整体性能。

## 进程状态转换

- **状态定义**

进程通常有多种状态，如创建、就绪、运行、阻塞和终止等。进程管理模块负责跟踪和管理进程的状态变化。

- **转换机制**

模块通过一系列的事件和操作来触发进程状态的转换。这些事件包括系统调用、中断等。例如，当一个进程执行 `sched_yield` 系统调用时，进程管理模块会将该进程从运行状态转换为就绪状态，并触发调度器选择下一个进程运行。

## 进程间通信（IPC）

尚未实现

## 与其他模块的交互关系

### 与内存管理模块的交互：

进程管理模块在创建和销毁进程时，需要与内存管理模块紧密合作。在创建进程时，需要从内存管理模块获取足够的内存空间来存储进程的代码、数据和栈等信息；在销毁进程时，要将进程占

用的内存归还给内存管理模块。当一个进程动态分配内存时，进程管理模块和内存管理模块会协同工作，记录内存的分配情况，并在进程结束时正确地释放这些内存。

### 与文件系统模块的交互：

有些进程在运行过程中需要进行文件操作，如打开、读取、写入和关闭文件等。进程管理模块会与文件系统模块合作，为进程提供文件描述符等资源，以便进程能够方便地访问文件。当一个进程打开一个文件时，进程管理模块会协助文件系统模块为该进程分配一个文件描述符，并将其记录在进程的相关数据结构中，方便后续的文件操作。

## 进程管理流程

### 初始化

在内核入口函数中，执行`task::init()`函数，其作用是初始化任务管理器，并将系统的第一个进程：INIT\_PROC进程添加到任务管理器的就绪队列中，并为其创建一个进程控制块，进程状态从Create状态更改为Ready状态，其中INIT\_PROC的代码和数据来自`initproc`只读文件。由于最先加入到就绪队列的是INIT\_PROC任务，所以优先执行INIT\_PROC任务。

### 任务的创建：

除了INIT\_PROC以外的所有任务都是由`user/src`中的`sh`文件产生的，它的作用是使用`get_command`接收用户输入的命令，使用`fork`（返回值为0时表示子进程）创建对应命令的进程。

### 任务的执行

在用户程序中，我们已经通过`sh.rs`创建了任务，接着对`pid`进行判断，如果是子进程（`fork==0`），则执行`execve`函数产生名为`sys_execve`（用来执行相应的任务）的系统调用；如果是父进程（`fork!=0`），则等待子进程执行完毕，并显示出相关的信息。

### 任务的调度

1. 为了实现时间片轮转的调度算法，我们采用抢占式调度，通过定时器中断产生信号，因此，需要通过执行`enable_timer_interrupt`函数开启定时器中断，并执行`set_next_timeout(trap::handler::TIMER_INTERVAL_USEC)`设置下一次定时器中断的时间。
2. 接着，执行`run_tasks`函数（实现对当前的Processor对象的字段的更新），这个函数会不断循环，尝试从任务管理器中获取任务并运行。在执行任务之前，需要先获取空闲任务上下文指针，获取任务的PCB，从PCB中获取当前任务上下文的指针。然后将进程状态从Ready状态更改为Running状态表示进程正在执行，接着将Processor对象的`current`（当前正在运行的任务）设置为刚刚获取到的任务，并把Processor对象的`idle_ctx`设置为获取到的任务上下文，从而实现对任务的调度。

## 进程控制块

### 进程控制块的原理

#### 定义与作用



进程控制块（PCB）是操作系统用于记录和管理进程状态及相关信息的核心数据结构。它就像是进程在操作系统中的“身份证”，包含了进程运行过程中所需要的各种关键信息。从进程的创建到销毁，操作系统通过 PCB 来跟踪和控制进程的行为。

## 信息存储内容

### 1. 进程标识信息

包括进程标识符（PID），它是在系统范围内唯一标识一个进程的数字。PID 用于区分不同的进程，就像每个人的身份证号码一样，方便操作系统在众多进程中快速定位和管理特定的进程。此外，还包括父进程的 PID，用于构建进程之间的父子关系，这个关系对于进程层次结构的管理和资源继承等操作非常重要。

### 2. 处理器状态信息

当进程被暂停或切换时，需要保存当前的处理器状态，以便后续恢复执行。主要包括程序计数器（PC），它指向进程下一条要执行的指令地址；寄存器内容，如通用寄存器、栈指针寄存器。这些信息能够确保进程在被中断或暂停后，再次被调度时可以从原来的位置准确地继续执行。

3. **进程调度信息** 用于支持进程调度操作。由于我们采用的是先来先服务+时间片轮转法，所以最重要的信息是进程的时间片信息，此外，还包括进程的状态（如创建、就绪、运行、阻塞、终止），这些信息帮助调度器确定哪个进程应该在什么时候获得 CPU 执行权。

4. **进程的内存信息** 记录进程的内存布局和使用情况。包括进程的代码段、数据段和栈段的起始地址和大小等信息。通过这些信息，操作系统可以对进程的内存进行有效的管理，如在进程切换时正确地切换内存映射，或者在进程需要更多内存时进行内存分配操作。

5. **进程间通信信息（尚未实现）** 如果进程参与了进程间通信（IPC），PCB 中会包含相关的信息。例如，对于使用消息队列进行通信的进程，PCB 可能会记录消息队列的标识符以及该进程在消息队列中的相关状态（如是否有未读取的消息等）；对于共享内存通信方式，会记录共享内存区域的起始地址和大小等信息。

## 进程控制块的实现

### 进程控制块数据结构ProcessControlBlock

ProcessControlBlock结构体是整个进程管理模块的核心数据结构，它全面地记录了与一个进程相关的各种关键信息。

#### 字段描述：

- **pid**：通过PidHandler类型记录进程的唯一标识符，用于区分不同的进程，在进程调度、资源管理等多个环节都会依赖该 ID 进行相关操作。
- **kernel\_stack**：使用KernelStack结构体表示进程对应的内核栈，涉及内核栈的创建、使用以及销毁等生命周期管理，为进程在内核态的函数调用、局部变量存储等操作提供栈空间支持。
- **status**：通过ProcessStatus枚举类型来标识进程当前所处的状态，便于系统根据状态进行相应的调度和管理决策，例如将就绪状态的进程投入运行等操作。
- **ctx**：保存任务上下文信息的TaskCtx结构体，在任务切换等操作中起着关键作用，确保进程切换时能正确恢复和保存执行现场的寄存器等关键信息。

- `memory_set`: 代表进程的内存集合, 通过`MemorySet`类型管理进程所占用的内存空间, 包括内存的分配、映射以及地址转换等相关操作, 确保进程在合法的内存范围内进行数据读写等操作。
- `trap_ctx_ppn`: 记录陷阱上下文的物理页号, 用于在处理中断、异常等陷阱情况时准确地定位和操作相关的内存页面, 与底层的硬件异常处理机制紧密相关。
- `base_size`: 存储一个基础大小相关的`usize`值, 其具体含义可能与进程的内存布局、初始栈大小或者其他与内存使用相关的基准量有关, 具体取决于整个系统的设计逻辑。
- `parent`: 使用`Option<Weak<SyncRefCell<ProcessControlBlock>>>`来表示父进程, 通过弱引用 (`Weak`) 的方式避免循环引用, 同时可以关联到父进程的相关信息, 在诸如进程资源回收、继承关系处理等方面有重要作用。
- `children`: 是一个`Vec<Arc<SyncRefCell<ProcessControlBlock>>>`类型的向量, 用于存储该进程的所有子进程, 方便对父子进程关系进行管理, 例如在父进程结束时对子进程的处理等操作。
- `exit_code`: 记录进程结束时的退出码, 用于向父进程或者其他相关系统组件反馈进程执行的结果状态, 通常在进程正常或异常结束后会设置相应的退出码值。
- `fd_table`: 即前面定义的文件描述符表类型`FdTable`, 用于管理进程打开的文件相关资源, 通过文件描述符来索引对应的文件对象, 实现对文件的读写等操作。

## 进程控制块方法实现

- `get_trap_ctx`函数

该函数用于获取陷阱上下文的可变引用, 通过调用`self.trap_ctx_ppn` (陷阱上下文的物理页号相关操作) 的`get_mut`函数来实现, 返回的可变引用可以用于进一步修改陷阱上下文的相关内容, 例如在初始化或者处理陷阱情况时更新其中的寄存器值等信息。

- `get_user_token`函数

用于获取用户态的令牌, 其实现是通过调用进程的内存集合 (`memory_set`) 的`token`函数来获取相应的值, 具体这个令牌在整个系统中的作用可能与用户态权限管理、内存访问限制等方面相关, 根据系统设计而定。

- `get_status`函数

简单地返回进程当前的状态, 通过直接访问`self.status`字段, 外部代码可以方便地获取进程所处的状态信息, 以便进行相应的流程控制和管理操作。

- `new`函数

`new`函数用于创建一个新的进程控制块实例。

### 具体步骤:

1. 首先, 根据传入的应用程序数据 (`app_data`), 通过 `MemorySet::from_elf_app(app_data)` 函数获取内存集合、用户栈指针 (`user_sp`) 以及程序入口地址 (`entry`) 等信息, 用于初始化进程的内存相关属性。
2. 接着, 通过 `PidAllocator::alloc_pid()` 分配一个新的进程 ID, 并利用该 ID 创建对应的内核栈 (`KernelStack::new(&pid)`), 同时获取内核栈的栈顶地址 (`kernel_stack_top`)。
3. 然后, 构建一个初始状态的`ProcessControlBlock`实例, 设置其进程 ID、内核栈、初始状态为`Ready`、任务上下文 (通过

TaskCtx::goto\_trap\_return(kernel\_stack\_top)根据栈顶地址初始化任务上下文)、内存集合、陷阱上下文的物理页号、基础大小以及初始的空子进程列表、默认退出码为 0，并初始化文件描述符表包含标准输入、输出等基本文件对象。

4. 最后，获取新创建的进程控制块的陷阱上下文引用 (pcb.get\_trap\_cx())，并通过 TrapCtx::init\_app\_context函数基于前面获取的程序入口地址、用户栈指针、内核空间令牌以及栈顶地址和陷阱处理函数地址等信息来初始化陷阱上下文，最终返回完整初始化后的进程控制块实例。

源代码：

```
pub fn new(app_data: &[u8]) -> Self {
    let (mut memory_set, user_sp, entry) = MemorySet::from_elf_app(app_data);
    let trap_ctx_ppn = PhysicalPageNumber::from(
        &memory_set
            .translate(VirtualPageNumber::from(VirtualAddress::from(TRAP_CONTEXT)))
            .unwrap(),
    );
    let pid = PidAllocator::alloc_pid();
    let kernel_stack = KernelStack::new(&pid);
    let kernel_stack_top = kernel_stack.top();

    let pcb = Self {
        pid,
        kernel_stack,
        status: ProcessStatus::Ready,
        ctx: TaskCtx::goto_trap_return(kernel_stack_top),
        memory_set,
        trap_ctx_ppn,
        base_size: user_sp,
        parent: None,
        children: Vec::new(),
        exit_code: 0,
        fd_table: vec![
            Some(Arc::new(Stdin)),
            Some(Arc::new(Stdout)),
            Some(Arc::new(Stdout)),
        ],
    };
    let ctx = pcb.get_trap_cx();
    *ctx = TrapCtx::init_app_context(
        entry,
        user_sp,
        KERNEL_SPACE.ref_cell.borrow_mut().token(),
        kernel_stack_top,
        trap_handler as usize,
    );
    pcb
}
```

- alloc\_fd函数

该函数用于在进程的文件描述符表中分配一个新的文件描述符，它会遍历当前的文件描述符表，如果发现有空的文件描述符位置（即元素为None），则返回该位置对应的索引作为新的文件描述符；如果整个表都已满，则向表中添加一个空元素（None），并返回新添加元素的索引（即表的长度减 1）作为新分配的文件描述符。

**源代码：**

```
pub fn alloc_fd(&mut self) -> usize {
    for (fd, file) in self.fd_table.iter().enumerate() {
        if file.is_none() {
            return fd;
        }
    }
    self.fd_table.push(None);
    self.fd_table.len() - 1
}
```

- fork函数

fork函数用于创建一个新的子进程，这个子进程会在很大程度上复制父进程的相关状态和资源配置，包括内存布局、文件描述符表等，同时具有自己独立的进程标识符、内核栈等关键元素，创建完成后子进程处于就绪状态，等待被调度执行。

**源代码：**

```

pub fn fork(
    self: &Arc<SyncRefCell<ProcessControlBlock>>,
) -> Arc<SyncRefCell<ProcessControlBlock>> {
    let mut parent = self.inner_borrow_mut();
    let memory_set = MemorySet::from_existed_user(&parent.memory_set);

    let trap_ctx_ppn = PhysicalPageNumber::from(
        &memory_set
            .translate(VirtualPageNumber::from(VirtualAddress::from(
                TRAP_CONTEXT)))
            .unwrap(),
    );

    let pid = PidAllocator::alloc_pid();
    let kernel_stack = KernelStack::new(&pid);
    let kernel_stack_top = kernel_stack.top();
    let child = Arc::new(SyncRefCell::new(ProcessControlBlock {
        pid,
        kernel_stack,
        status: ProcessStatus::Ready,
        ctx: TaskCtx::goto_trap_return(kernel_stack_top),
        memory_set,
        trap_ctx_ppn,
        base_size: parent.base_size,
        parent: Some(Arc::downgrade(self)),
        children: Vec::new(),
        exit_code: 0,
        fd_table: parent.fd_table.fd_clone(),
    }));
    parent.children.push(child.clone());

    let ctx = child.inner_borrow().get_trap_cx();
    ctx.kernel_virt_sp = kernel_stack_top;

    child
}

```

- exec函数

exec函数的主要功能是让当前进程执行一个新的应用程序，它会替换当前进程的内存空间、程序入口等关键执行相关的内容，基于传入的新的应用程序数据重新初始化进程的内存集合、陷阱上下文等，使得进程能开始执行新的程序逻辑，而不是原来的程序内容，相当于在同一个进程的“壳”下切换执行不同的程序任务。

**源代码：**

```

pub fn exec(self: &Arc<SyncRefCell<ProcessControlBlock>>, app_data: &[u8]) {
    let (memory_set, user_sp, entry) = MemorySet::from_elf_app(app_data);
    let trap_ctx_ppn = PhysicalPageNumber::from(
        &memory_set
            .translate(VirtualPageNumber::from(VirtualAddress::from(
                TRAP_CONTEXT)))
            .unwrap(),
    );
    let mut pcb = self.inner_borrow_mut();
    pcb.memory_set = memory_set;
    pcb.trap_ctx_ppn = trap_ctx_ppn;
    let ctx = pcb.get_trap_ctx();
    *ctx = TrapCtx::init_app_context(
        entry,
        user_sp,
        KERNEL_SPACE.ref_cell.borrow_mut().token(),
        pcb.kernel_stack.top(),
        trap_handler as usize,
    );
}

```

## 进程状态切换

### 简介

进程状态切换是指在操作系统中，进程从一种运行状态转变为另一种运行状态的过程。进程在其生命周期内会处于不同的状态，如创建（Create）、就绪（Ready）、运行（Running）、阻塞（Wait）和终止（Stopped）状态，进程状态切换就是进程在这些状态之间的动态转换。这种转换是由操作系统内核根据进程自身的行为、外部事件的触发或者系统调度策略来控制和管理。

### 进程状态数据结构

进程在其生命周期内会处于不同的状态，主要包括创建（Create）、就绪（Ready）、运行（Running）、阻塞（Wait）和终止（Stopped）状态。

#### 创建状态

创建状态是进程生命周期的起始阶段。当一个新进程被创建时，它首先进入创建状态。这个阶段是操作系统为新进程分配必要资源并进行初始化的过程，是进程从无到有的关键阶段。

#### 就绪状态

进程已经准备好执行，只要获得 CPU 资源就可以立即运行。此时进程已经完成了初始化工作，等待被调度器选中。

#### 运行状态

进程正在占用 CPU 资源执行其指令。在我们实现的小型操作系统中，同一时刻只有一个进程处于运行状态。

## 阻塞状态

进程由于等待某些事件的发生而暂停执行。在阻塞状态下，进程不能占用 CPU 资源，即使 CPU 处于空闲状态也不会执行。

## 终止状态

进程已经完成任务或者由于错误等原因结束运行。此时，进程的资源会被操作系统回收，进程控制块也会被释放或者标记为无效。

ProcessStatus是一个枚举类型，它定义了进程的五种状态：创建状态、就绪状态、运行状态、阻塞状态和终止状态，这种方式使得代码在处理进程状态时更加清晰可读。

进程控制块是存储进程相关信息的关键数据结构，其中包含一个用于表示进程状态的字段status。

```
pub enum ProcessStatus {  
    Create,  
    Ready,  
    Running,  
    Wait,  
    Stopped,  
}
```

## 进程状态切换描述

- **创建到就绪**：进程创建态是进程初始诞生的阶段，在此期间操作系统为新进程分配必要资源，如内存空间（包括代码段、数据段、栈段等）、初始化进程控制块（PCB）等。当这些初始化操作完成后，进程就从创建态切换到就绪态，进入就绪队列等待被调度器选中。
- **就绪到运行**：当一个处于就绪状态的进程被操作系统的调度器选中，获得 CPU 资源开始执行时，就会从就绪状态切换到运行状态。这体现了系统对 CPU 资源的分配，使进程有机会在 CPU 上运行指令序列。
- **运行到就绪**：有多种情况致使进程从运行状态切换回就绪状态。常见的是时间片用完，在时间片轮转调度策略下，进程占用 CPU 达到规定时间片长度后，即便任务未完成，也会被强制暂停，释放 CPU 资源给其他就绪进程，自身切换回就绪态，重新进入就绪队列等待下次调度。
- **运行到阻塞**：当进程在运行中需等待特定事件发生才能继续执行时，会从运行状态切换到阻塞状态。如从磁盘读取文件或向网络发送数据，因 I/O 设备速度比 CPU 慢，进程不能一直占用 CPU 等待 I/O 完成，此时释放 CPU 资源，进入阻塞状态，等待 I/O 操作结束。此外，进程等待某些系统资源（如信号量、互斥锁等），若资源暂时不可用，也会进入阻塞状态。
- **阻塞到就绪**：当进程等待的事件完成后，会从阻塞状态切换回就绪状态。比如进程等待的 I/O 操作完成，I/O 设备向 CPU 发送中断信号，操作系统中断处理程序接收信号后，将等待该 I/O 操作的进程从阻塞态唤醒，使其回到就绪态，重新加入就绪队列，等待被调度器选中再次获得 CPU 资源继续执行任务。
- **运行到终止**：当进程完成任务或出现无法继续运行的错误时，会从运行状态切换到终止状态。例如，应用程序正常退出，其对应进程执行退出系统调用exit，操作系统回收该进程占

用的所有资源，包括内存、打开的文件等，然后将进程状态从运行态切换为终止态，结束其生命周期。若进程执行非法操作（导致系统错误，也会被强制终止，从运行态切换到终止态。

## 进程状态切换的触发因素

### 任务执行

在任务执行的过程中，会一直执行run\_tasks函数，其中有一个loop循环，会一直从就绪队列中拿出新的任务进行执行，此时会将进程状态从就绪状态切换到执行状态。

run\_tasks 函数的主要功能是实现一个循环来持续处理任务相关的操作。在这个函数内部，首先进入一个无限循环，意味着它会不断地重复执行后续的操作流程，以保证任务能够持续被调度和执行。

```
pub fn run_tasks() {
    loop {
        let mut this = PROCESSOR.inner_borrow_mut();
        if let Some(task) = TaskManager::get_task() {
            let idle_ctx = this.get_idle_ctx_ptr();
            let mut pcb = task.inner_borrow_mut();
            let next_ctx = &pcb.ctx as *const TaskCtx;
            pcb.status = ProcessStatus::Running;
            drop(pcb);
            this.current = Some(task);
            drop(this);
            unsafe {
                __switch(idle_ctx, next_ctx);
            }
        }
    }
}
```

### 系统调用

许多系统调用会导致进程状态的切换。如当进程执行wait函数时，父进程会等待子进程结束，在等待过程中，父进程可能会从运行状态切换到阻塞状态。在wait函数中会执行sys\_yield系统调用，其内部会执行replace\_to\_next，将当前的进程阻塞，切换到下一个进程执行。

replace\_to\_next 函数主要用于处理当前任务结束后的相关收尾及后续调度工作。



```
pub fn replace_to_next(exit_code: i32) {
    let task = Processor::take_current().unwrap();
    let mut pcb = task.inner_borrow_mut();
    pcb.exit_code = exit_code;
    pcb.status = ProcessStatus::Stopped;
    let mut initproc = INIT_PROC.inner_borrow_mut();
    for child in pcb.children.iter() {
        child.inner_borrow_mut().parent = Some(Arc::downgrade(&INIT_PROC));
        initproc.children.push(child.clone());
    }
    drop(initproc);
    pcb.children.clear();
    pcb.memory_set.recycle();
    drop(pcb);
    drop(task);
    let mut _unused = TaskCtx::default();
    Processor::schedule(&mut _unused as *mut _);
}
```

## I/O 操作

当进程发起 I/O 请求，如读取磁盘文件或者向网络发送数据时，由于 I/O 设备的速度通常比 CPU 慢很多，进程会进入阻塞状态等待 I/O 操作完成。

## 中断

由于我们采用的是先来先服务+时间片轮转法，其中需要使用到定时器中断，当定时器中断发生时，会执行trap\_handler函数，在其中会调用cycle\_to\_next()函数，正在运行的进程会被强制暂停，从运行状态切换到就绪状态，然后调度器会选择下一个就绪进程投入运行。

cycle\_to\_next 函数的核心作用在于完成当前任务的状态切换以及后续的任务调度相关操作，以保障系统中任务能够有序轮转执行。

```
pub fn cycle_to_next() {
    let task = Processor::take_current().unwrap();
    let mut pcb = task.inner_borrow_mut();
    let ctx = &mut pcb.ctx as *mut TaskCtx;
    pcb.status = ProcessStatus::Ready;
    drop(pcb);
    TaskManager::put_task(task);
    Processor::schedule(ctx);
}
```

## 进程调度策略

不同的调度策略会导致进程状态的切换。例如，优先级调度策略下，如果有一个更高优先级的进程进入就绪状态，当前正在运行的低优先级进程可能会被抢占，从运行状态切换到就绪状态，让高优先级进程获得 CPU 资源并运行。

但是，在我们实现的小型操作系统中不会产生因为进程调度策略而产生的进程状态。

# 进程状态切换的实现机制

## 通过操作系统内核代码实现

当触发状态切换的事件发生时，通过系统调用处理程序来实现状态切换。系统调用程序会修改进程控制块中的状态字段来记录进程的新状态。

## 涉及的队列操作

操作系统一般会维护多个进程队列，如就绪队列和阻塞队列。状态切换通常伴随着进程在不同队列之间的移动。当进程从运行状态切换到就绪状态时，它会被放回就绪队列的适当位置，等待下一次调度。如果是从就绪状态切换到运行状态，进程会从就绪队列中被取出，调度器将 CPU 资源分配给它。而从运行或就绪状态切换到阻塞状态时，进程会被移动到相应的阻塞队列，直到等待的事件完成，再从阻塞队列移回就绪队列。

```
pub struct TaskManager {  
    ready_queue: VecDeque<Arc<SyncRefCell<ProcessControlBlock>>>,  
}
```

## 上下文保存与恢复

当进程从运行状态切换出去时，无论是切换到就绪状态还是阻塞状态，都需要保存当前的上下文信息。这包括保存 CPU 寄存器的值、程序计数器（PC）等，这些信息保存在 PCB 中。当进程再次被调度运行时，会从 PCB 中恢复这些上下文信息，使得进程能够从上次暂停的地方继续执行，就好像没有被中断过一样。例如，在进程切换时，内核会使用特殊的指令来将当前运行进程的寄存器状态保存到其 PCB 中，然后将即将运行进程的寄存器状态从其 PCB 中恢复到 CPU 寄存器中，从而实现平滑的状态切换和执行流程的延续。

- 上下文数据结构

- **ra（返回地址寄存器）字段**

返回地址寄存器（ra）用于存储函数调用返回后的下一条指令的地址。当一个函数调用另一个函数时，调用函数的执行流程会跳转到被调用函数的入口地址，此时处理器会将调用函数中当前指令的下一条指令地址保存到ra寄存器中。在被调用函数执行完毕后，通过ra寄存器中的地址，程序可以准确地返回到调用函数中继续执行。

- **sp（栈指针寄存器）字段**

栈指针寄存器（sp）指向当前栈的栈顶位置。栈在程序运行过程中用于存储局部变量、函数参数和函数调用的返回信息等。在函数调用时，栈指针会根据需要动态地调整，为新的局部变量和调用信息分配空间。当函数返回时，栈指针会恢复到之前的位置，释放相应的栈空间。

- **s（保存寄存器数组，包含s0 - s11寄存器）字段**

s0 - s11这些寄存器通常是处理器中的通用寄存器，用于存储操作数、中间计算结果等各种数据。

```
pub struct TaskCtx {
    ra: usize,           // 返回地址寄存器
    sp: usize,           // 栈指针寄存器
    s: [usize; 12],      // 保存寄存器数组, 包含 s0-s11 寄存器
}
```

- 获取当前任务上下文

`current_trap_cx` 函数的主要功能是获取当前的陷阱上下文 (`TrapCtx`), 并以可变的静态引用 (`&'static mut TrapCtx`) 形式返回, 同时借助 `Option` 类型来妥善处理可能出现的获取失败情况。

```
pub fn current_trap_cx() -> Option<&'static mut TrapCtx> {
    let current = Self::get_current()?;
    let trap_cx = current.ref_cell.borrow_mut().get_trap_cx();
    Some(trap_cx)
}
```

- 恢复任务上下文

`goto_trap_return` 函数主要用于构建并返回一个特定类型 (从函数返回值类型 `Self` 推测其所属的自定义结构体类型) 的实例, 该实例的构建与陷阱返回相关的操作存在关联。函数接收一个参数 `kernel_stack_ptr`, 其类型为 `usize`, 这个参数大概率表示内核栈指针, 意味着它会参与到后续返回实例的构建中, 用于确定栈相关的状态信息。

```
pub fn goto_trap_return(kernel_stack_ptr: usize) -> Self {
    Self {
        ra: trap_return as usize,
        sp: kernel_stack_ptr,
        s: [0; 12],
    }
}
```

## 进程调度

### 进程调度的定义与目的

#### 定义

进程调度是操作系统内核中的一个关键功能, 它负责决定在某个时刻哪个进程可以占用 CPU 资源来执行。就像是交通警察指挥车辆 (进程) 通行, 合理安排它们对道路 (CPU) 的使用。

#### 目的

主要是为了提高 CPU 的利用率, 确保多个进程能够公平、高效地共享 CPU 资源。同时, 要满足不同进程的响应时间要求, 例如对于一些实时性要求高的进程 (如多媒体播放、工业控制等), 要及时分配 CPU 时间, 避免出现卡顿或者失控的情况。

# 进程调度的触发时机

## 进程状态变化

当一个进程从运行状态变为阻塞状态，或者时间片用完，进程被强制暂停，从运行变为就绪时，就会触发调度。此时，操作系统需要从就绪队列中选择一个新的进程来运行。

## 新进程创建

当新进程被创建并进入就绪状态后，也会触发调度。系统需要决定是立即让新进程运行，还是让它等待，这取决于调度策略和当前系统的负载情况。

## 中断处理完成

当时间片用完产生的时钟中断处理完成后，也需要进行调度。因为中断可能改变了进程的状态或者系统资源的可用性，所以要重新安排进程对 CPU 的占用。

# 进程调度的主要策略

在我们实现的小型操作系统中，主要实现的是基于先来先服务+时间片轮转法的进程调度策略。

## 先来先服务（FCFS）

- **原理：**按照进程到达就绪队列的先后顺序来分配 CPU 资源。先到达就绪队列的进程先被调度运行，就像排队买票一样，谁先来谁先买。
- **优点：**简单、公平，易于理解和实现。
- **缺点：**对长作业进程比较有利，可能导致短作业进程长时间等待。例如，一个大型数据处理进程先进入就绪队列，它可能会占用 CPU 很长时间，使得后面的小任务进程迟迟无法得到 CPU 资源。

## 短作业优先（SJF）

- **原理：**优先选择预计执行时间短的进程来运行。调度器需要对每个进程的执行时间有一个估计，然后选择最短的那个先执行。
- **优点：**可以有效地减少平均周转时间，对于短作业进程能够快速响应。
- **缺点：**长作业进程可能会因为不断有短作业进程插入而导致饥饿（。而且，准确估计每个进程的执行时间在实际应用中比较困难。

## 时间片轮转（RR）

- **原理：**将 CPU 时间划分成固定长度的时间片，每个进程轮流占用 CPU 一个时间片的时间。当一个进程的时间片用完后，无论其是否完成任务，都要暂停并将 CPU 资源让给下一个就绪进程。
- **优点：**能够保证每个进程都能在一定时间内得到 CPU 资源，适用于分时系统，能够提供较好的交互响应性能。
- **缺点：**频繁的上下文切换可能会导致系统开销增加，影响系统的整体性能。

## 优先级调度

- **原理：**为每个进程赋予一个优先级，优先级高的进程优先获得 CPU 资源。优先级可以由用户指定，也可以由系统根据进程的性质（）、资源需求等因素来确定。
- **优点：**可以根据进程的重要性和紧急程度来合理分配资源，对于一些关键任务（如系统维护进程、实时控制进程）能够及时响应。
- **缺点：**可能会导致低优先级进程饥饿，而且确定优先级的标准可能比较复杂，容易出现不公平的情况。

## 先来先服务+时间片轮转

### 1. 结合原则

这种调度思路是将先来先服务（FCFS）的公平性原则和时间片轮转（RR）的高效性原则相结合。FCFS 保证按照进程到达就绪队列的先后顺序进行调度，就像在银行排队办理业务一样，先到的先服务。而 RR 则是为了避免长作业进程长时间占用 CPU，导致其他进程长时间等待，通过给每个进程分配一个固定的时间片来限制单个进程对 CPU 的独占时间。

### 2. 就绪队列构建与初始调度顺序

#### ◦ 就绪队列初始化

系统中有一个就绪队列，用于存放所有已经准备好运行但还未获得 CPU 资源的进程。当新进程创建完成并进入就绪状态后，就会被添加到这个就绪队列的末尾。

```
pub struct TaskManager {  
    ready_queue: VecDeque<Arc<SyncRefCell<ProcessControlBlock>>>,  
}
```

#### ◦ 首次调度依据：

调度开始时，按照 FCFS 的方式，选择就绪队列头部的进程作为第一个被调度的对象。因为它是最早进入就绪队列的，所以先获得 CPU 资源，这体现了对先到达进程的公平对待。

### 3. 时间片轮转机制的融入

#### ◦ 时间片设定与分配：

为每个进程分配一个固定长度的时间片，这个时间片的长度是根据系统的性能和对响应时间的要求来确定的。在我们的小型操作系统内核中，设置的时间片的长度是10ms。

#### ◦ 进程执行与时间片耗尽处理：

在进程执行期间，系统会记录进程已经使用的时间。当进程使用的时间达到时间片长度时，如果进程还没有主动放弃 CPU，就会被强制暂停。这种强制暂停确保了每个进程不会过长时间地占用 CPU，使得其他进程也有机会获得 CPU 资源。

#### ◦ 进程状态转换与队列调整：

当一个进程的时间片耗尽后，它的状态从运行状态转换为就绪状态。然后，这个进程会被移动到就绪队列的末尾，等待下一次调度。这一步骤很关键，它保证了调度仍然遵循 FCFS 的顺序，同时又实现了时间片轮转。

### 4. 动态循环调度过程

整个调度过程是一个动态的循环。调度器不断地检查就绪队列，按照 FCFS 的顺序从头部选择进程，为其分配时间片并启动执行。在执行过程中，根据进程是主动放弃 CPU 还是时间片耗尽来进行不同的处理，包括状态转换和队列调整。这个循环会持续进行，只要系统中有就绪进程存在，就会一直按照这种方式进行调度，从而在保证公平性的基础上，有效地利用 CPU 资源，提高系统的整体性能和响应速度。

## 进程调度的实现细节

### 进程创建与就绪队列初始化

#### 进程创建流程

- 分配进程标识符 (PID)

当创建一个新进程时，首先通过 `PidAllocator::alloc_pid()` 函数从 `PID_ALLOCATOR` 中获取一个唯一的 PID。`PidAllocator` 维护了未使用的 PID 范围和回收的 PID 列表。如果回收列表中有可用的 PID，则优先使用回收的 PID；否则，从 `unused` 范围中分配一个新的 PID。分配成功后，将该 PID 封装在 `PidHandler` 结构体中返回，`PidHandler` 在被丢弃时会自动释放 PID。

- **PID分配器结构体**

```
pub struct PidAllocator {
    unused: Range<usize>,           // 未使用的PID范围
    recycled: Vec<usize>,           // 回收的PID列表
}
```

- **PidAllocator的方法实现**

- **new方法**

用于创建一个新的 `PidAllocator` 实例。

初始化 `unused` 字段为 `0..(usize::MAX - 256)`，表示从 0 开始的可用 PID 范围，同时创建一个空的 `recycled` 列表。

```
impl PidAllocator {
    /// 创建一个新的PID分配器
    fn new() -> Self {
        Self {
            unused: 0..(usize::MAX - 256), // 保留256个PID
            recycled: Vec::new(),
        }
    }
}
```

- **alloc方法**

用于分配一个新的 PID。

首先，检查 `recycled` 列表中是否有可用的 PID。如果有，从列表中弹出一个 PID（使用 `pop` 方法），并将其封装在 `PidHandler` 结构体中返回。

如果recycled列表为空，则检查unused范围是否还有可用的 PID。如果unused.start小于unused.end，表示还有未使用的 PID，将unused.start作为新分配的 PID 返回，并将unused.start加 1，指向下一个可用的 PID。

如果没有可用的 PID（recycled列表为空且unused范围已用尽），则返回None。

```
impl PidAllocator {
    /// 分配一个新的PID
    fn alloc(&mut self) -> Option<PidHandler> {
        // 优先使用回收的PID
        if let Some(pid) = self.recycled.pop() {
            return Some(PidHandler(pid));
        }
        // 如果没有回收的PID，则使用未使用的PID
        if self.unused.start < self.unused.end {
            let pid = self.unused.start;
            self.unused.start += 1;
            return Some(PidHandler(pid));
        }
        None
    }
}
```

#### ▪ dealloc方法

用于释放一个 PID，将其添加到回收列表中。

首先，检查要释放的 PID 是否在unused范围之后或者已经在recycled列表中。如果是，会触发panic，表示尝试释放无效的 PID。

然后，将 PID 添加到recycled列表中，以便后续重新分配。

```
impl PidAllocator {
    /// 释放一个PID
    fn dealloc(&mut self, pid: usize) {
        // 检查PID是否在未使用的范围内并且是否已经在回收列表中
        if pid >= self.unused.start {
            panic!("尝试释放未分配的PID: {:?}", pid);
        }
        if self.recycled.iter().any(|&f| f == pid) {
            panic!("尝试释放已释放的PID: {:?}", pid);
        }
        // 将PID添加到回收列表中
        self.recycled.push(pid);
    }
}
```

#### ▪ alloc\_pid方法

用于直接从PID\_ALLOCATOR中分配一个新的 PID。

通过PID\_ALLOCATOR.ref\_cell.borrow\_mut()获取PidAllocator的可变借用，然后调用alloc方法分配 PID。如果分配成功，返回PidHandler结构体封装的PID；如果分配失败（alloc返回None），unwrap会触发panic。

```
impl PidAllocator {
    /// 分配一个新的PID并返回处理器
    pub fn alloc_pid() -> PidHandler {
        PID_ALLOCATOR.ref_cell.borrow_mut().alloc().unwrap()
    }
}
```

#### ◦ PidHandler结构体与Drop实现

PidHandler结构体用于封装 PID，并在其被丢弃时自动释放 PID。

```
pub struct PidHandler(pub usize);

impl Drop for PidHandler {
    fn drop(&mut self) {
        PID_ALLOCATOR.ref_cell.borrow_mut().dealloc(self.0);
    }
}
```

- PidHandler结构体只包含一个usize类型的字段，用于存储 PID。
- 实现了Drop trait，当PidHandler实例被丢弃时，会自动调用drop方法。在drop方法中，通过PID\_ALLOCATOR.ref\_cell.borrow\_mut().dealloc(self.0)将封装的 PID 释放回PidAllocator的回收列表中。

#### • 构建进程控制块 (PCB)

使用 `ProcessControlBlock::new(app_data)` 来创建新进程的 PCB。首先，通过 `MemorySet::from_elf_app(app_data)` 构建进程的内存集合，并获取用户栈指针和程序入口地址。接着，根据内存集合计算陷阱上下文的物理页号。然后，创建与该进程对应的内核栈 `KernelStack::new(&pid)`，并获取内核栈顶地址。创建一个初始化为陷阱返回状态的任务上下文 `TaskCtx::goto_trap_return(kernel_stack_top)`，其中 `ra` 设置为 `trap_return` 函数的地址，`sp` 设置为内核栈顶地址，`s` 寄存器数组初始化为 0。最后，构建 `ProcessControlBlock` 结构体实例，包含了分配的 PID、内核栈、初始状态、任务上下文、内存集合、陷阱上下文物理页号、父进程信息、子进程列表、退出码以及文件描述符表。创建完成后，将新进程的 PCB 封装，以便进行安全的共享和可变借用。

#### ◦ ProcessControlBlock结构体定义

`ProcessControlBlock`结构体用于存储进程的各种信息，包括进程 ID、内核栈、状态、任务上下文、内存集合、陷阱上下文物理页号、父进程和子进程信息、退出码以及文件描述符表等。



```

pub struct ProcessControlBlock {
    pub pid: PidHandler,           // 进程ID
    pub kernel_stack: KernelStack, // 内核栈
    pub status: ProcessStatus,     // 进程状态
    pub ctx: TaskCtx,              // 任务上下文
    pub memory_set: MemorySet,     // 内存集合
    pub trap_ctx_ppn: PhysicalPageNumber, // 陷阱上下文的物理页号
    pub base_size: usize,          // 基础大小
    pub parent: Option<Weak<SyncRefCell<ProcessControlBlock>>>, // 父进程
    pub children: Vec<Arc<SyncRefCell<ProcessControlBlock>>>, // 子进程
    pub exit_code: i32,            // 退出码
    pub fd_table: FdTable,         // 文件描述符表
}

```

#### ◦ ProcessControlBlock方法实现

- new方法
  - 构建内存集合并获取相关信息
  - 分配 PID 并创建内核栈
  - 初始化任务上下文
  - 初始化陷阱上下文

```

impl ProcessControlBlock {
    pub fn new(app_data: &[u8]) -> Self {
        // 1. 构建内存集合并获取相关信息
        let (memory_set, user_sp, entry) = MemorySet::from_elf_app(app_data);
        let trap_ctx_ppn = PhysicalPageNumber::from(
            &memory_set
                .translate(VirtualPageNumber::from(VirtualAddresses::from(TRAP_CONTEXT)))
                .unwrap(),
        );
        // 2. 分配PID并创建内核栈
        let pid = PidAllocator::alloc_pid();
        let kernel_stack = KernelStack::new(&pid);
        let kernel_stack_top = kernel_stack.top();
        // 3. 初始化任务上下文
        let pcb = Self {
            pid,
            kernel_stack,
            status: ProcessStatus::Ready,
            ctx: TaskCtx::goto_trap_return(kernel_stack_top),
            memory_set,
            trap_ctx_ppn,
            base_size: user_sp,
            parent: None,
            children: Vec::new(),
            exit_code: 0,
            fd_table: vec![
                Some(Arc::new(Stdin)),
                Some(Arc::new(Stdout)),
                Some(Arc::new(Stdout)),
            ],
        };
        // 4. 初始化陷阱上下文
        let ctx = pcb.get_trap_ctx();
        *ctx = TrapCtx::init_app_context(
            entry,
            user_sp,
            KERNEL_SPACE.ref_cell.borrow_mut().token(),
            kernel_stack_top,
            trap_handler as usize,
        );

        pcb
    }
}

```

- 初始化陷阱上下文

在 `ProcessControlBlock::new` 函数中，还会进一步初始化陷阱上下文。通过 `TrapCtx::init_app_context` 函数，将程序入口地址、用户栈指针、内核空间的页表根节点地址、内核栈顶地址以及陷阱处理函数地址等信息填充到陷阱上下文中。

- `TrapCtx`结构体定义

```

pub struct TrapCtx {
    pub x: [usize; 32],           // 通用寄存器数组，包含 x0-x3
1 寄存器
    pub sstatus: usize,          // 保存特权级状态寄存器
    pub sepc: usize,             // 保存异常程序计数器寄存器
    pub kernel_satp: usize,      // 保存内核页表基地址寄存器
    pub kernel_sp: usize,        // 保存内核栈指针寄存器
    pub kernel_trap_handler: usize, // 保存内核陷阱处理函数地址
}

```

#### ◦ TrapCtx方法实现

##### ▪ init\_app\_context方法

- 初始化陷阱上下文时，先创建TrapCtx结构体实例ctx并设置sepc为应用程序入口地址、kernel\_satp为内核页表根节点地址、kernel\_sp为内核栈顶地址、kernel\_trap\_handler为陷阱处理函数地址。
- 随后将用户栈指针存入寄存器a2，以便在某些情况下传递给陷阱处理程序或后续执行使用。
- 接着针对 k210 平台，按要求将用户栈指针对齐到 8 字节，若原始指针未对齐则计算对齐后的指针，并把调整量存于寄存器a1，对齐后的指针存于a2，以满足内存访问对齐要求。
- 最后返回包含完整陷阱上下文信息的ctx实例，在进程创建时由ProcessControlBlock::new调用TrapCtx::init\_app\_context初始化新进程陷阱上下文，保障陷阱处理时进程状态的正确保存与恢复。

```

impl TrapCtx {
    pub fn init_app_context(
        entry: usize,
        user_sp: usize,
        kernel_token: usize,
        kernel_stack_top: usize,
        trap_handler: usize,
    ) -> Self {
        let mut ctx = Self {
            x: [0; 32],
            sstatus: 0,
            sepc: entry,
            kernel_satp: kernel_token,
            kernel_sp: kernel_stack_top,
            kernel_trap_handler: trap_handler,
        };

        ctx.x[12] = user_sp;

        let mut user_sp_aligned = user_sp &!(core::mem::size_of::() - 1);
        if user_sp != user_sp_aligned {
            user_sp_aligned -= core::mem::size_of::();
            *ctx.x(1) = user_sp - user_sp_aligned;
            *ctx.x(2) = user_sp_aligned;
        }
        ctx
    }
}

```

## 就绪队列管理

新创建的进程或通过 fork 创建的子进程，会通过 TaskManager::put\_task(task) 函数将其添加到 TaskManager 的就绪队列中。在 TaskManager 结构体中，就绪队列被定义一个双向队列，用于存放准备执行的任务。

- 添加到就绪队列

- TaskManager 结构体定义

TaskManager 结构体用于管理任务的调度和执行，其中包括一个就绪队列 (ready\_queue)，用于存放准备执行的任务。

```

pub struct TaskManager {
    ready_queue: VecDeque<Arc<SyncRefCell<ProcessControlBlock>>>,
    // 就绪队列
}

```

- TaskManager 的方法实现

- new 方法

用于创建一个新的TaskManager实例，初始化就绪队列。

```
impl TaskManager {  
    /// 创建一个新的任务管理器实例  
    pub fn new() -> Self {  
        TaskManager {  
            ready_queue: VecDeque::new(),  
        }  
    }  
}
```

- put方法

用于将任务添加到就绪队列。

```
impl TaskManager {  
    /// 将任务放入就绪队列  
    pub fn put(&mut self, task: Arc<SyncRefCell<ProcessControlBlock>>) {  
        self.ready_queue.push_back(task);  
    }  
}
```

- put\_task方法

用于将任务添加到全局任务管理器（TASK\_MANAGER）的就绪队列中。

```
lazy_static! {  
    /// 全局任务管理器  
    pub static ref TASK_MANAGER: SyncRefCell<TaskManager> = {  
        SyncRefCell {  
            ref_cell: RefCell::new(TaskManager::new()),  
        }  
    };  
}  
  
impl TaskManager {  
    /// 将任务放入全局任务管理器的就绪队列  
    pub fn put_task(task: Arc<SyncRefCell<ProcessControlBlock>>) {  
        let mut this = TASK_MANAGER.ref_cell.borrow_mut();  
        this.put(task);  
    }  
}
```

## 进程调度循环

### 获取任务

在 `Processor::run_tasks` 函数中，通过一个无限循环不断进行进程调度。在每次循环开始时，首先通过 `PROCESSOR.inner_borrow_mut()` 获取 `Processor` 的可变借用，然后调用

TaskManager::get\_task() 尝试从就绪队列中获取一个任务。如果就绪队列中有任务，get\_task 函数会从队列头部取出一个任务并返回；如果就绪队列为空，则返回 None。

- TaskManager的相关方法

- get\_task方法

用于从全局任务管理器的就绪队列中获取任务的方法。

```
lazy_static! {  
    /// 全局任务管理器  
    pub static ref TASK_MANAGER: SyncRefCell<TaskManager> = {  
        SyncRefCell {  
            ref_cell: RefCell::new(TaskManager::new()),  
        }  
    };  
}  
  
impl TaskManager {  
    /// 从全局任务管理器的就绪队列中取出任务  
    pub fn get_task() -> Option<Arc<SyncRefCell<ProcessControlBlock>>> {  
        >>> {  
            let mut this = TASK_MANAGER.ref_cell.borrow_mut();  
            this.get()  
        }  
    }  
}
```

### 设置任务状态与上下文

若成功获取到任务，首先，通过 this.get\_idle\_ctx\_ptr() 获取 Processor 的空闲任务上下文指针。然后，对获取到的任务调用 inner\_borrow\_mut 方法获取其可变借用，接着获取任务上下文指针，用于后续的上下文切换操作。之后，将任务的状态设置为运行，表示该任务即将在 CPU 上执行。最后，释放对 pcb 的可变借用，并将当前任务设置为获取到的任务，准备进行上下文切换。

- Processor结构体定义

Processor结构体用于负责任务的调度和切换，其中包含当前正在运行的任务（current）和空闲任务上下文（idle\_ctx）等字段。

```
pub struct Processor {  
    current: Option<Arc<SyncRefCell<ProcessControlBlock>>>,    // 当前正在运行的任务  
    idle_ctx: TaskCtx,   // 空闲任务上下文  
}
```

- run\_tasks函数定义

通过获取任务、设置任务状态为运行、获取和更新相关上下文指针，最后进行上下文切换，确保任务能够正确地在 CPU 上执行，并且在任务切换时能够保存和恢复任务的执行上下文，

保证系统的稳定运行。

```
pub fn run_tasks() {
    loop {
        let mut this = PROCESSOR.inner_borrow_mut();
        if let Some(task) = TaskManager::get_task() {
            let idle_ctx = this.get_idle_ctx_ptr();
            let mut pcb = task.inner_borrow_mut();
            let next_ctx = &pcb.ctx as *const TaskCtx;
            pcb.status = ProcessStatus::Running;
            drop(pcb);
            this.current = Some(task);
            drop(this);
            unsafe {
                __switch(idle_ctx, next_ctx);
            }
        }
    }
}
```

### 上下文切换与执行

调用 `__switch(idle_ctx, next_ctx)` 进行上下文切换。在切换过程中，会保存当前任务的上下文信息到当前任务上下文空间，然后从下一个任务的上下文空间恢复下一个任务的寄存器值，最后通过 `ret` 指令返回，使得下一个任务开始执行。当任务执行完一个时间片或者主动放弃 CPU 后，会再次回到 `Processor::run_tasks` 函数的循环开头，继续进行下一轮的任务调度。

### 进程切换与状态更新（时间片轮转机制）

#### 时间片耗尽检测

在 `Processor::run_tasks` 函数中，通过定时器来跟踪任务的执行时间。当任务执行时间达到设定的时间片长度时，就认为时间片耗尽。

#### 进程状态更新与队列操作

当时间片耗尽时，`Processor` 会调用 `TaskManager::cycle_to_next()` 函数。在 `cycle_to_next` 函数中，首先通过 `Processor::take_current().unwrap()` 获取当前正在执行的任务，并将其从 `Processor` 中移除。然后，对获取到的任务调用 `inner_borrow_mut` 方法获取可变借用，获取任务上下文指针，并将任务状态更新为就绪，表示该任务已经完成了本次时间片的执行，准备再次进入就绪队列等待下一次调度。接着，释放对 `pcb` 的可变借用，并通过 `TaskManager::put_task(task)` 将任务放回就绪队列末尾，实现时间片轮转的效果。最后，调用 `Processor::schedule(ctx)` 进行上下文切换，选择就绪队列中的下一个任务执行。

- **`cycle_to_next()`函数定义**

`cycle_to_next`函数主要实现时间片轮转调度，当进程时间片用完，它获取当前任务，将其状态更新为就绪并放回就绪队列末尾，随后进行上下文切换以执行下一个任务，确保各进程公平共享 CPU 资源。

```
pub fn cycle_to_next() {
    let task = Processor::take_current().unwrap();
    let mut pcb = task.inner_borrow_mut();
    let ctx = &mut pcb.ctx as *mut TaskCtx;
    pcb.status = ProcessStatus::Ready;
    drop(pcb);
    TaskManager::put_task(task);
    Processor::schedule(ctx);
}
```

- **replace\_to\_next函数定义**

replace\_to\_next函数在进程结束时发挥作用，获取当前任务后，设置退出码并将状态更新为停止，接着处理子进程相关信息，把它们关联到INIT\_PROC下，同时回收进程占用内存，最后创建默认上下文并调度新任务，保证系统在进程结束后能继续稳定运行，有效管理进程状态转换、资源回收与调度。

```
pub fn replace_to_next(exit_code: i32) {
    let task = Processor::take_current().unwrap();
    let mut pcb = task.inner_borrow_mut();
    pcb.exit_code = exit_code;
    pcb.status = ProcessStatus::Stopped;
    let mut initproc = INIT_PROC.inner_borrow_mut();
    for child in pcb.children.iter() {
        child.inner_borrow_mut().parent = Some(Arc::downgrade(&INIT_PROC));
        initproc.children.push(child.clone());
    }
    drop(initproc);
    pcb.children.clear();
    pcb.memory_set.recycle();
    drop(pcb);
    drop(task);
    let mut _unused = TaskCtx::default();
    Processor::schedule(&mut _unused as *mut _);
}
```

## 进程结束处理

### 进程终止触发

当进程执行 exit 系统调用或者遇到错误等情况导致进程无法继续执行时，会触发进程终止操作。此时，Processor 会调用 TaskManager::replace\_to\_next(exit\_code) 函数，exit\_code 用于传递进程结束的原因等信息。

### 资源回收与状态变更

在 replace\_to\_next 函数中，首先通过 Processor::take\_current().unwrap() 获取当前任务，并对其调用 inner\_borrow\_mut 方法获取可变借用。然后，将进程的退出码设置为传入的 exit\_code，并将进程状态更新为停止。接着，处理进程的子进程相关信息，将当前进程的所有子进程转移到 INIT\_PROC 下。之后，释放当前进程的子进程列表，回收进程占用的内存（。最



后，释放对 pcb 的可变借用，释放对当前任务的引用（。然后，创建一个默认的任务上下文，并将其可变指针作为参数传递给 `Processor::schedule(&mut _unused as *mut _)`，触发进程调度，选择下一个任务执行，继续系统的运行。

# 文件系统模块

## 文件系统模块概述

### 简介

**文件系统**是操作系统的重要组成部分，负责**管理磁盘上的文件和目录**，提供对持久化存储的访问。通过文件系统，用户和程序可以方便地创建、读取、写入和删除文件，而无需关心底层的存储细节。

在本项目结合实际需求实现了以下核心功能：

- **文件与目录的管理**：支持创建、读取、写入、删除文件，以及操作目录。
- **文件系统接口**：提供抽象的系统调用接口，屏蔽底层实现细节。
- **简易文件系统 (ros-fs)**：实现了一个高效且模块化的文件系统，包含块设备接口、块缓存管理以及磁盘布局。
- **内核中的文件系统支持**：通过集成 ros-fs，在内核中实现文件系统的初始化、文件描述符管理，以及文件读写功能。

本文件系统模块的目标是提供一个简单、高效、易于扩展的文件系统框架，同时支持在用户态和内核态的文件操作。

### 核心设计目标

#### 1. 模块化设计：

- 采用松耦合模块化设计，文件系统由多个层次构成，包括块设备接口层、块缓存层和磁盘布局层，各层次之间通过接口通信，便于扩展和维护。

#### 2. 高效性：

- 使用块缓存机制，减少磁盘 I/O，提高文件操作性能。

#### 3. 可靠性：

- 通过简单的磁盘布局设计和位图管理，保证文件系统的稳定性和数据一致性。

#### 4. 灵活性：

- 支持文件和目录操作，提供用户态和内核态的文件访问功能，满足不同场景的需求。

### 模块结构

文件系统模块的实现主要分为以下几个部分：

#### 1. 块设备接口层

- 提供统一的接口与底层存储设备交互，实现对磁盘块的读取与写入操作。

- 接口定义在 `ros-fs/src/block_dev.rs` 中，支持多种存储设备的适配。

## 2. 块缓存层

- 管理块的缓存，避免频繁的磁盘访问。通过 LRU 缓存策略，提升磁盘操作性能。
- 实现在 `ros-fs/src/block_cache.rs` 中。

## 3. 磁盘布局层

- 定义文件系统在磁盘上的布局，包括超级块、位图、索引节点和数据块等。
- 实现在 `ros-fs/src/layout/` 中，核心文件包括 `super_block.rs` 等。

## 4. 文件系统抽象层

- 在磁盘布局上建立文件系统抽象，提供文件和目录物理结构的管理接口。
- 实现在 `ros-fs/src/fs.rs` 中。

## 5. 文件和目录操作

- 提供文件和目录的创建、读取、写入、删除等操作。
- 实现在 `ros-fs/src/virt_fs.rs` 和 `src/fs.rs` 中。

## 6. 文件系统系统调用

- 提供用户态和内核态的文件系统接口，支持文件的打开、关闭、读写等操作。
- 实现在 `src/syscall.rs` 中。

## 7. 文件系统镜像打包工具

- 提供将文件系统镜像打包为二进制文件的工具，用于加载到内核中。
- 实现在 `ros-fs-fuse/` 中。

## 8. 块设备驱动

- 提供块设备的抽象接口，支持不同类型的块设备，如 VirtIO 块设备 和 SD 卡等。
- 实现在 `src/drivers/block/` 中。

# 总体功能概述

本文件系统模块实现了以下核心功能：

## 1. 文件操作

- 支持创建、读取、写入和删除文件。
- 提供顺序读写的接口，支持文件的随机访问。

## 2. 目录操作

- 支持创建、读取、写入和删除目录。
- 提供目录项的管理和查找功能。

## 3. 文件系统接口

- 提供抽象的文件系统接口，包括文件打开、关闭、读写和目录操作等系统调用。
- 支持用户态和内核态的文件操作。

## 4. 块设备驱动

- 提供块设备的抽象接口，支持不同类型的块设备，如 VirtIO 块设备 和 SD 卡等。
- 实现了读取和写入块数据的接口。

## 5. 块缓存

- 在内存中缓存磁盘块，减少磁盘访问，提高文件操作性能。
- 实现了块缓存管理器和块缓存替换策略。

## 6. 内核支持

- 集成 ros-fs 文件系统，支持内核态的文件操作。
- 提供文件描述符管理，实现系统调用接口。

## 7. 文件系统镜像打包工具

- 提供将文件系统镜像打包为二进制文件的工具，用于加载到内核中。
- 支持文件系统镜像的生成和加载，以及文件校验。

# 总结

本项目文件系统模块通过模块化设计实现了文件和目录的管理功能，并结合块缓存和简化的磁盘布局，提供了高效可靠的文件操作支持。通过与内核的深度集成，该文件系统模块不仅满足了内核态文件操作的需求，还为用户态应用提供了完整的文件系统接口。

在接下来的部分中，我们将详细介绍文件系统的各个子模块，包括块设备接口、块缓存层、磁盘布局以及内核中的文件系统集成，更加深入地来介绍此部分我们的工作。

# 文件系统/块设备接口及缓冲层

## 简介

## 块设备接口

文件系统模块的核心功能之一是与块设备交互，读取和写入磁盘上的块数据。

一个块设备应当具有以下特性：

- 读写操作：提供读取和写入块的操作。
- 随机访问：可以通过块号直接访问任意块。
- 固定块大小：块设备应当有固定的块大小，例如 512 字节。

而对于不同的硬件块设备，其底层实现和接口可能存在差异，例如，SD 卡、磁盘、VirtIO 块设备等。这些设备需要不同的驱动程序和接口来访问。

为了屏蔽底层硬件的差异，我们定义了一个块设备接口层，提供了统一的接口，用于读取和写入磁盘上的块数据。

一个块设备驱动程序只需要实现这个接口，就可以被文件系统模块使用。

```
pub trait BlockDevice: Send + Sync + Any {
    /// 读取一个块
    ///
    /// ## 参数
    /// - `block_id`: 块 ID
    /// - `buf`: 目标缓冲区
    fn read_block(&self, block_id: usize, buf: &mut [u8]);
    /// 写入一个块
    ///
    /// ## 参数
    /// - `block_id`: 块 ID
    /// - `buf`: 源缓冲区
    fn write_block(&self, block_id: usize, buf: &[u8]);
}
```

- **read\_block**: 读取指定块号的数据到缓冲区 buf。
- **write\_block**: 将缓冲区 buf 的数据写入到指定块号。

## 块设备缓冲层

块缓存层是文件系统模块的重要组成部分，用于缓解块设备读写的性能瓶颈。通过在内存中缓存磁盘块的数据，块缓存层能够：

- 缓存块数据：缓存磁盘块数据，减少对磁盘的访问。
- 块数据同步：保证缓存中的数据与磁盘数据一致。
- 块数据替换：当缓存满时，根据一定的策略替换缓存中的数据。
- 块数据读写：提供读写缓存中的块数据的接口。
- 块数据管理：管理缓存中的块数据，包括分配、释放等操作。

### 块缓存管理器

在块缓存层中，我们使用先进先出（FIFO）算法作为块替换策略。当缓存满时，将替换最早进入缓存的块。

块缓冲管理器的实现如下：

```
pub struct BlockCacheManager {
    queue: VecDeque<(usize, Arc<Mutex<BlockCache>>>>,
}
```

其中，queue 是一个双端队列，用于存储缓存块的数据。每个缓存块由块号和块数据组成。

块缓存管理器提供了对缓存块的获取接口：

```

pub fn get_cache(
    &mut self,
    id: usize,
    dev: Arc<dyn BlockDevice>,
) -> Option<Arc<Mutex<BlockCache>>> {
    for (cache_id, cache) in self.queue.iter() {
        if *cache_id == id {
            return Some(cache.clone());
        }
    }
    if self.queue.len() >= BLOCK_CACHE_SIZE {
        let found = self
            .queue
            .iter()
            .enumerate()
            .find(|(_, (_, cache))| Arc::strong_count(cache) == 1);
        if let Some((idx, _)) = found {
            self.queue.remove(idx);
        } else {
            return None;
        }
    }
    let cache = Arc::new(Mutex::new(BlockCache::new(id, dev.clone())));
    self.queue.push_back((id, cache.clone()));
    Some(cache)
}

```

get\_cache 方法用于获取指定块号的缓存块，如果缓存中不存在，则创建一个新的缓存块。当缓存满时，根据 FIFO 策略替换最早进入缓存，并且只替换引用计数为 1 的缓存块。

创建一个全局的块缓存管理器以供文件系统模块使用：

```

lazy_static! {
    pub static ref BLOCK_CACHE_MANAGER: Mutex<BlockCacheManager> =
        Mutex::new(BlockCacheManager::new());
}

```

块缓存管理器是一个全局的单例对象，用于管理所有的缓存块。

## 块缓存结构

一个缓存块的结构如下：

```

pub struct BlockCache {
    cache: [u8; BLOCK_SIZE],
    id: usize,
    dev: Arc<dyn BlockDevice>,
    modified: bool,
}

```

一个缓存块缓存了一个块的数据，包括块号、块数据、块设备、以及是否被修改的标志。

缓存块对应了一个磁盘块，其在创建时会从磁盘上读取数据。

我们只需要通过块设备接口将指定块号的数据加载到缓存中：

```
pub fn new(block_id: usize, device: Arc<dyn BlockDevice>) -> Self {
    let mut cache = [0u8; BLOCK_SIZE];
    device.read_block(block_id, &mut cache);
    Self {
        cache,
        block_id,
        is_dirty: false,
        device,
    }
}
```

为了使得当缓冲块被销毁时，能够将数据写回磁盘，我们为缓冲块实现了 Drop trait：

```
impl Drop for BlockCache {
    fn drop(&mut self) {
        self.sync();
    }
}
```

sync 方法检查缓存块是否被修改，如果被修改，则将数据写回磁盘。

```
pub fn sync(&mut self) {
    if self.modified {
        self.dev.write_block(self.id, &self.cache);
        self.modified = false;
    }
}
```

## 块缓存层接口

块缓冲层建立在块设备接口之上，提供了读写缓存块的接口。

```

pub fn read_at<T, R>(&self, offset: usize, adapter: impl FnOnce(&T) -> R) -
> Option<R>
where
    T: Sized,
{
    let t = self.get_ref_at::

```

`read_at` 方法用于读取缓存块中的数据，`modify_at` 方法用于修改缓存块中的数据。

这两个方法都接受一个偏移量 `offset`，用于指定数据在缓存块中的位置。

`adapter` 参数是一个闭包，用于对数据进行处理，在获取到块数据后，将数据传递给闭包进行处理，以提供更加灵活的接口。

其中，`get_ref_at` 和 `get_mut_at` 方法用于获取缓存块中的数据的引用。其只是对获取指定偏移处的数据方法 `get_at` 的封装。

```

fn get_addr_at(&self, offset: usize) -> usize {
    &self.cache[offset] as *const _ as usize
}

fn get_at<T>(&self, offset: usize) -> Option<usize>
where
    T: Sized,
{
    let t_size = core::mem::size_of::<T>();
    if offset + t_size > BLOCK_SIZE {
        return None;
    }
    Some(self.get_addr_at(offset))
}

pub fn get_ref_at<T>(&self, offset: usize) -> Option<&T>
where
    T: Sized,
{
    let addr = self.get_at::<T>(offset)?;
    Some(unsafe { &*(addr as *const T) })
}

pub fn get_mut_at<T>(&mut self, offset: usize) -> Option<&mut T>
where
    T: Sized,
{
    let addr = self.get_at::<T>(offset)?;
    self.modified = true;
    Some(unsafe { &mut *(addr as *mut T) })
}

```

get\_at 方法用于获取指定偏移处的数据的地址，其检查偏移量是否越界，并返回数据的地址。get\_ref\_at 和 get\_mut\_at 方法将数据地址转换为对应类型的不可变和可变引用。

## 总结

块设备接口层是文件系统模块的基础，负责提供底层块存储的抽象和接口实现。通过内存块设备的模拟，我们能够快速验证文件系统的功能，同时为接入实际硬件设备提供了统一的框架支持。

块缓存层通过缓存磁盘块的数据，显著优化了块设备的读写性能，并提供了线程安全的缓存管理接口。通过 LRU 缓存策略和延迟写回机制，块缓存层有效减少了磁盘 I/O 操作，为文件系统的高效运行提供了强大的支持。

在下面一个部分中，我们将基于项目详细阐述文件系统的磁盘布局设计，以及如何使用索引节点和数据块管理文件和目录。

## 文件系统/磁盘布局与索引节点

### 简介



磁盘布局与索引节点（Layout and DiskInode）是文件系统的核心组成部分，负责组织和管理文件及目录在磁盘上的存储位置。通过对磁盘空间的合理划分和索引节点的高效设计，文件系统能够实现文件的快速定位和操作。

在本项目中，磁盘布局的设计遵循简化和高效的原则，将磁盘划分为以下几个部分：

- **超级块（Super Block）**：存储文件系统的元信息。
- **索引节点位示图**：用于索引节点的分配和回收。
- **索引节点区**：存储文件和目录的元数据。
- **数据块位示图**：用于数据块的分配和回收。
- **数据块区**：存储文件和目录的实际数据内容。

## 磁盘布局

文件系统的磁盘布局是文件系统在磁盘上的组织结构，用于存储文件和目录的元数据和数据。在本项目中，文件系统的磁盘布局的划分如下：

### 1. 超级块：

- 存储校验魔数，用于检测文件系统格式是否正确。
- 存储文件系统的全局信息，如磁盘块数量、索引节点数量等。
- 用于初始化和管理工作系统。

### 2. 索引节点位示图：

- 用于分配和回收索引节点。
- 通过位示图记录每个索引节点的使用情况。

### 3. 索引节点区：

- 存储文件和目录的元数据。
- 每个索引节点包含文件的大小、数据块指针等信息。

### 4. 数据块位示图：

- 用于分配和回收数据块。
- 通过位示图记录每个数据块的使用情况。

### 5. 数据块区：

- 存储文件和目录的实际数据内容。
- 每个数据块存储固定大小的数据。

接下来我们将依次介绍超级块、位示图、索引节点的设计和实现。

## 超级块（Super Block）

超级块是文件系统的元信息，用于存储文件系统的全局信息，一个文件系统只有一个超级块。超级块的数据结构定义如下：

```
pub struct SuperBlock {
    pub magic: u32,           // 文件系统的魔数, 用于校验
    pub total_blocks: u32,    // 总磁盘块数
    pub inode_bitmap_blocks: u32, // 索引节点位图占用的块数
    pub data_bitmap_blocks: u32, // 数据块位图占用的块数
    pub inode_area_blocks: u32,  // 索引节点区占用的块数
    pub data_area_blocks: u32,   // 数据块区占用的块数
}
```

超级块指示了 inode 和 数据块 的位示图及其内容区域的大小。

- **magic**: 用于校验文件系统格式是否正确。
- **total\_blocks**: 磁盘块的总数。
- **inode\_bitmap\_blocks** 和 **data\_bitmap\_blocks**: 用于分配和回收索引节点和数据块的位图。
- **inode\_area\_blocks** 和 **data\_area\_blocks**: 分别用于存储索引节点和数据块的磁盘块数。

## 位示图 (Bitmap)

位示图是一种用于记录磁盘块或索引节点使用情况的数据结构，通过位示图可以高效地分配和回收磁盘块或索引节点。

### 位示图结构

一个位示图管理了块设备某一范围内的空闲块，通过位示图可以快速查找空闲块并进行分配。位示图的数据结构定义如下：

```
pub struct Bitmap {
    start_block: usize,
    blocks: usize,
}
```

- **start\_block**: 位示图的起始块号。
- **blocks**: 位示图占用的块数。

位示图由多个块组成，每个块包含 64 个 u64 类型的位，位示图的每一位对应一个物理磁盘块，0 表示空闲，1 表示已分配

一个位示图的结构如下：

u64[0]	u64[1]	u64[2]	u64[3]	
u64[4]	u64[5]	u64[6]	u64[7]	
...	...	...	...	
u64[60]	u64[61]	u64[62]	u64[63]	

## 位示图操作

位示图提供了以下操作：

- alloc：分配一个空闲块
- free：释放一个块
- maximum：获取位示图的最大块数

位示图的每个块使用一个块缓存来缓存，当需要修改位示图时，首先获取对应的块缓存

然后修改块缓存中的数据，最后将块缓存写回到块设备中。位示图的实现使用了 Arc 类型来引用块设备，以便在多个线程之间共享块设备

alloc 方法用于分配一个空闲块，它首先遍历位示图的每个块，查找第一个空闲块，然后将该块标记为已分配。

```
pub fn alloc(&self, dev: &Arc<dyn BlockDevice>) -> Option<usize> {
    for id in 0..self.blocks {
        let cache = get_cache((id + self.start_block) as usize, Arc::clone
(dev))
            .expect("cannot get cache");
        let pos = cache
            .lock()
            .modify_at(0, |bitmap_block: &mut BitmapBlock| {
                for (pos, bits64) in bitmap_block.iter_mut().enumerate() {
                    // 找到第一个有空闲的组的第一个为 0 (空闲) 的位
                    if *bits64 != u64::MAX {
                        let bits64 = bits64.trailing_ones() as usize;
                        bitmap_block[pos] |= 1 << bits64;
                        return Some(id * BLOCK_BITS + pos * 64 + bits64);
                    }
                }
                None
            })
            .expect("cannot modify cache");
        if pos.is_some() {
            return pos;
        }
    }
    None
}
```

在查找时，可以一次比较 64 位，如果 64 位都为 1，那么这一组的位示图数为 `u64::MAX`，表示这一组块都已经分配，可以直接跳过这一组。

`free` 方法用于释放一个块，它首先计算位示图号（即在位示图中的第几个 bit），然后将该 bit 标记为空闲。

```
pub fn free(&self, dev: &Arc<dyn BlockDevice>, bit: usize) {
    let (block, pos, bit) = Self::extract_bit_position(bit);
    let cache = get_cache((block + self.start_block) as usize, Arc::clone(dev))
        .expect("cannot get cache");
    cache
        .lock()
        .modify_at(0, |bitmap_block: &mut BitmapBlock| {
            assert!(bitmap_block[pos] & (1 << bit) != 0);
            bitmap_block[pos] &= !(1 << bit);
        })
        .expect("cannot modify cache");
}
```

其中，`extract_bit_position` 方法用于计算位示图号对应的块号、在位示图中的坐标：

```
fn extract_bit_position(mut bit: usize) -> (usize, usize, usize) {
    let block = bit / BLOCK_BITS;
    bit %= BLOCK_BITS;
    let pos = bit / 64;
    let bit = bit % 64;
    (block, pos, bit)
}
```

## 磁盘索引节点 (DiskInode)

在文件系统的不同层级都有相应的一个索引节点，用于存储在该层级的元数据信息。

从下往上分配为：

- **磁盘索引节点**：存储文件和目录的元数据信息。
- **内存索引节点**：用于内存中的索引节点缓存。
- **文件系统索引节点**：在操作系统中用于表示文件和目录的数据结构，文件的打开、读写等操作都是通过文件系统索引节点来完成。

磁盘索引节点是文件系统的核心数据结构，一个 Inode 包含了文件的元数据信息，包括文件类型、大小、数据块索引等。磁盘索引节点的设计直接影响了文件系统的性能和可靠性。

我们使用混合索引结构来支持大小文件的高效存取，通过直接、间接和双重间接指针来定位文件的数据块。这样，小文件可以通过直接指针快速定位到数据块，避免了额外的间接寻址开销；大文件则通过间接和双重间接指针来支持更大的文件大小，避免索引节点存储开销过大。

磁盘索引节点的数据结构定义如下：

```
pub struct Inode {
    pub size: u32, // 文件的大小
    pub direct_blocks: [u32; INODE_DIRECT_BLOCKS], // 直接指向数据块的指针
    pub indirect_block: u32, // 间接块指针
    pub double_indirect_block: u32, // 双重间接块指针
}
```

- **size**: 文件的大小，单位为字节。
- **direct\_blocks**: 最多 28 个直接指针，指向文件的数据块。
- **indirect\_block**: 单级间接指针，指向一个包含数据块地址的块，最多可指向 128 个数据块
- **double\_indirect\_block**: 双级间接指针，通过两级索引指向文件数据块，最多可指向 128 \* 128 个数据块。

## 索引节点访问

在初始化时，简单地将索引节点的数据初始化为 0，表示文件为空。

```
pub fn init(&mut self, r#type: InodeType) {
    self.size = 0;
    self.r#type = r#type;
    self.direct.iter_mut().for_each(|x| *x = 0);
    self.indirect = 0;
    self.double_indirect = 0;
}
```

其中，InodeType 用于表示索引节点的类型，包括文件和目录。

使用 `r#` 前缀来避免关键字冲突，如 `type` 是 Rust 的关键字，因此使用 `r#type` 来表示类型。

在访问索引节点时，我们需要根据偏移量在文件中的不同位置选择在直接指针、间接指针和双重间接指针中查找

对于直接指针，我们可以直接根据偏移量计算出数据块的索引，然后直接读取或写入数据。

```
fn get_direct_block(&self, offset: usize) -> u32 {
    self.direct[offset]
}
```

对于间接指针，我们需要先读取间接块，然后根据偏移量在间接块中查找数据块的索引，最后读取或写入数据。

```
fn get_indirect_block(&self, offset: usize, dev: &Arc<dyn BlockDevice>) ->
u32 {
    get_cache(self.indirect as usize, dev.clone())
        .expect("cannot get cache")
        .lock()
        .read_at(0, |indirect: &IndirectBlock| {
            indirect[offset as usize - INODE_DIRECT_BLOCKS]
        })
        .expect("cannot read cache")
}
```

这里，我们使用了 `get_cache` 方法来获取间接块的缓存，传入闭包来读取间接块的数据。

对于双重间接指针，我们需要先读取双重间接块，然后根据偏移量在双重间接块中查找间接块的索引，再根据偏移量在间接块中查找数据块的索引，最后读取或写入数据。

```
fn get_double_indirect_block(&self, offset: usize, dev: &Arc<dyn BlockDevice>) -> u32 {
    let (level1_offset, level2_offset) =
        Self::extract_double_indirect_block(offset).expect("invalid offset");
    let cache =
        get_cache(self.double_indirect as usize, dev.clone()).expect("cannot get cache");
    let indirect_block = cache
        .lock()
        .read_at(0, |indirect: &IndirectBlock| indirect[level1_offset])
        .expect("cannot read cache");
    get_cache(indirect_block as usize, dev.clone())
        .expect("cannot get cache")
        .lock()
        .read_at(0, |indirect: &IndirectBlock| indirect[level2_offset])
        .expect("cannot read cache")
}
```

其中，`extract_double_indirect_block` 方法用于计算双重间接指针的偏移量：

```
fn extract_double_indirect_block(offset: usize) -> Option<(usize, usize)> {
    let start = INODE_DOUBLE_INDIRECT_START;
    if offset < start {
        return None;
    }
    let offset = offset - start;
    let level1_offset = offset / INODE_INDIRECT_BLOCKS;
    let level2_offset = offset % INODE_INDIRECT_BLOCKS;
    Some((level1_offset, level2_offset))
}
```

它计算偏移在两级索引中的位置，一个位置二元组，分别偏移在一级索引表的偏移和二级索引表的偏移，

在这三种访问方式之上，我们包装一层 `translate` 方法，用于根据偏移量选择合适的访问方式：

```
```rust
pub fn translate(&self, offset: u32, dev: &Arc<dyn BlockDevice>) -> u32 {
    let offset = offset as usize;
    if offset < INODE_DIRECT_BLOCKS {
        self.get_direct_block(offset)
    } else if offset < INODE_DOUBLE_INDIRECT_START {
        self.get_indirect_block(offset, dev)
    } else {
        self.get_double_indirect_block(offset, dev)
    }
}
```

基于 translate 方法，我们可以实现读取和写入数据的操作：

```

pub fn read_at(&self, mut offset: usize, dev: &Arc<dyn BlockDevice>, buf: &mut [u8]) -> usize {
    let mut read = 0;
    let end = (offset + buf.len()).min(self.size as usize);
    if offset >= end {
        return 0;
    }
    let mut offset_block = offset / BLOCK_SIZE;
    loop {
        let current_block_end = ((offset / BLOCK_SIZE + 1) * BLOCK_SIZE).min(end);
        let size_to_read = current_block_end - offset;
        let dst = &mut buf[read..read + size_to_read];
        let cache = get_cache(
            self.translate(offset_block as u32, dev) as usize,
            dev.clone(),
        )
        .expect("cannot get cache");
        cache.lock().read_at(0, |block: &DataBlock| {
            let start = offset % BLOCK_SIZE;
            dst.copy_from_slice(&block[start..start + size_to_read]);
        });
        read += size_to_read;
        offset_block += 1;
        offset = current_block_end;
        if current_block_end == end {
            break;
        }
    }
    read
}

pub fn write_at(&mut self, mut offset: usize, dev: &Arc<dyn BlockDevice>, buf: &[u8]) -> usize {
    let mut written = 0;
    let end = (offset + buf.len()).min(self.size as usize);
    if offset >= end {
        return 0;
    }
    let mut offset_block = offset / BLOCK_SIZE;
    loop {
        let current_block_end = ((offset / BLOCK_SIZE + 1) * BLOCK_SIZE).min(end);
        let size_to_write = current_block_end - offset;
        let src = &buf[written..written + size_to_write];
        let cache = get_cache(
            self.translate(offset_block as u32, dev) as usize,
            dev.clone(),
        )
        .expect("cannot get cache");
        cache.lock().modify_at(0, |block: &mut DataBlock| {
            let start = offset % BLOCK_SIZE;
            block[start..start + size_to_write].copy_from_slice(src);
        });
        written += size_to_write;
        offset_block += 1;
        offset = current_block_end;
        if current_block_end == end {
            break;
        }
    }
    written
}

```



```

        written += size_to_write;
        offset_block += 1;
        offset = current_block_end;
        if current_block_end == end
        {
            break;
        }
    }
    self.size = self.size.max(end as u32);
    written
}

```

`read_at` 和 `write_at` 方法相当类似，都是根据偏移量和缓冲区大小，逐块读取或写入数据。

## 索引节点扩充和清空

### 扩充

在文件系统中，文件的大小可能会发生变化，因此我们需要支持索引节点的扩充和清空操作。

在扩充时，我们首先需要根据文件大小计算需要的数据块数量，然后根据当前数据块数量和需要的数据块数量的差值，分配或释放数据块。

计算所需的数据块数量比较简单，由于一个数据块只能存储一个文件块，因此文件大小除以数据块大小向上取整即可得到所需的数据块数。

```

fn calc_required_block_for(size: u32) -> u32 {
    size.div_ceil(BLOCK_SIZE as u32)
}

```

数据块以外，我们还需要考虑为这些数据块建立索引表所需要的索引块数量，索引块数量加上数据块数量即为文件所需的总块数。

```

pub fn required_blocks_for(&self, size: u32) -> u32 {
    let data_blocks = Self::calc_required_block_for(size) as usize;
    let mut index_blocks = 0;
    if data_blocks > INODE_DIRECT_BLOCKS {
        index_blocks += 1;
    }
    if data_blocks > INODE_DOUBLE_INDIRECT_START {
        index_blocks += 1;
        index_blocks +=
            (data_blocks - INODE_DOUBLE_INDIRECT_START).div_ceil(INODE_INDIRECT_BLOCKS);
    }
    data_blocks as u32 + index_blocks as u32
}

```

如果数据块数小于等于直接索引块数，不需要额外的索引块；

如果数据块数大于直接索引块数，需要一级索引块；

如果数据块数大于一级索引块数，需要二级索引块，二级索引块中的每个索引指向一个一级索引块，共需要  $(\text{数据块数} - \text{一级索引块数}) / \text{一级索引块数}$  个索引块。

计算出所需的总块数后，我们可以根据当前数据块数量和所需的数据块数量的差值，分配或释放数据块。

```
pub fn extend_size(&mut self, new_size: u32, new_blocks: Vec<u32>, dev: &Arc<dyn BlockDevice>) {
    let mut offset = self.required_blocks_for(self.size) as usize;
    self.size = new_size;
    let mut new_blocks = new_blocks.into_iter().peekable();
    // ...
}
```

这里，我们将新的数据块列表作为参数传入，然后转为迭代器，用于逐个分配新的数据块。调用其上的 `next` 方法，可以获取下一个数据块的索引。

我们依次从直接指针、间接指针和双重间接指针中分配或释放数据块，直到达到所需的数据块数量。

直接索引只需要将数据块位置写入直接指针即可；

```
```rust
while offset < INODE_DIRECT_BLOCKS {
    if let Some(block) = new_blocks.next() {
        self.direct[offset] = block;
        offset += 1;
    } else {
        return;
    }
}
```

在 `new_blocks` 迭代器空时，我们可以提前返回，表示数据块分配完毕。

对于间接索引，我们需要先分配间接块，然后将数据块位置写入间接块。

```

if offset == INODE_DIRECT_BLOCKS {
    // 分配一级索引表块
    self.indirect = new_blocks.next().unwrap();
}

let indirect_table_cache =
    get_cache(self.indirect as usize, dev.clone()).expect("cannot get cache");

while offset < INODE_DOUBLE_INDIRECT_START {
    if let Some(block) = new_blocks.next() {
        indirect_table_cache
            .lock()
            .modify_at(0, |indirect: &mut IndirectBlock| {
                indirect[offset - INODE_DIRECT_BLOCKS] = block;
            })
            .expect("cannot modify cache");
        offset += 1;
    } else {
        return;
    }
}

```

在这里，我们首先分配一级索引块，然后获取一级索引块的缓存，逐个写入数据块的位置。

对于双重间接索引，我们需要先分配双重间接块，然后分配一级索引块，再分配二级索引块，最后将数据块位置写入二级索引块。

```

let (mut level1_offset, mut level2_offset) =
    Self::extract_double_indirect_block(offset).unwrap_or((0, 0));

/** 分配直接和间接索引块 */

if offset == INODE_DOUBLE_INDIRECT_START {
    // 分配二级索引表的一级索引块
    self.double_indirect = new_blocks.next().unwrap();
}

let double_indirect_level1_cache =
    get_cache(self.double_indirect as usize, dev.clone()).expect("cannot get cache");

// 二级索引块缓存
let mut double_indirect_level2_cache = None;

while new_blocks.peek().is_some() {
    if level2_offset == 0 && double_indirect_level2_cache.is_none() {
        // 分配二级索引表中的二级索引块
        // 写入二级索引表的一级索引块
        let block = new_blocks.next().unwrap();
        double_indirect_level1_cache
            .lock()
            .modify_at(0, |indirect: &mut IndirectBlock| {
                indirect[level1_offset] = block;
            })
            .expect("cannot modify cache");
        double_indirect_level2_cache =
            Some(get_cache(block as usize, dev.clone()).expect("cannot get cache"));

        // 更新索引偏移
        level1_offset += 1;
    } else {
        // 分配二级索引表中的数据块
        // 写入二级索引表的二级索引块
        let block = new_blocks.next().unwrap();
        double_indirect_level2_cache
            .as_ref()
            .unwrap()
            .lock()
            .modify_at(0, |indirect: &mut IndirectBlock| {
                indirect[level2_offset] = block;
            })
            .expect("cannot modify cache");

        // 更新索引偏移
        if level2_offset == INODE_INDIRECT_BLOCKS - 1 {
            level2_offset = 0;
            double_indirect_level2_cache = None;
        } else {
            level2_offset += 1;
        }
    }
}

```

```
}
```

level1\_offset 和 level2\_offset 分别表示二级索引表中的一级索引块和二级索引块的偏移。其在分配前根据已有的文件大小进行定位。例如，在文件大小已经分配到了二级索引时，level1\_offset 和 level2\_offset 就指向目前分配到的二级索引块的位置，否则为 0。

在分配二级索引块时，level2\_offset 逐级递增，当达到一级索引块的最大容量时，level2\_offset 归零，level1\_offset 递增，表示分配下一个一级索引块。

在 level1\_offset 迭代时，二级索引块的缓存 double\_indirect\_level2\_cache 会被清空，表示需要分配二级索引块，分配后将其缓存保存在 double\_indirect\_level2\_cache 中。

### 清空

清空时，只需要逐级释放数据块即可。

```

pub fn clear_size(&mut self, dev: &Arc<dyn BlockDevice>) -> Vec<u32> {
    let mut recycled = Vec::new();
    self.size = 0;
    for i in 0..INODE_DIRECT_BLOCKS {
        if self.direct[i] != 0 {
            recycled.push(self.direct[i]);
            self.direct[i] = 0;
        }
    }
    if self.indirect != 0 {
        let cache = get_cache(self.indirect as usize, dev.clone()).expect(
("cannot get cache"));
        cache.lock()
            .read_at(0, |indirect: &IndirectBlock| {
                for &block in indirect.iter().filter(|&b| b != 0) {
                    recycled.push(block);
                }
            })
            .expect("cannot read cache");
        recycled.push(self.indirect);
        self.indirect = 0;
    }
    if self.double_indirect != 0 {
        let cache =
            get_cache(self.double_indirect as usize, dev.clone()).expect("c
annot get cache");
        cache.lock()
            .read_at(0, |indirect: &IndirectBlock| {
                for &block in indirect.iter().filter(|&b| b != 0) {
                    let cache =
                        get_cache(block as usize, dev.clone()).expect("cann
ot get cache");
                    cache.lock()
                        .read_at(0, |indirect: &IndirectBlock| {
                            for &block in indirect.iter().filter(|&b| b !
= 0) {
                                recycled.push(block);
                            }
                        })
                        .expect("cannot read cache");
                    recycled.push(block);
                }
            })
            .expect("cannot read cache");
        recycled.push(self.double_indirect);
        self.double_indirect = 0;
    }
    recycled
}

```

## 总结

磁盘布局与索引节点的设计是文件系统的核心，提供了文件和目录在磁盘上的组织和管理能力。通过超级块存储全局信息、位图管理磁盘分配、索引节点存储文件元数据以及数据块区存储实际内容，文件系统实现了高效可靠的文件操作。

在下面一个部分中，我们将基于项目详细阐述文件系统的目录结构和文件操作接口的实现。

## 文件系统/文件系统抽象

在有块设备接口和磁盘布局管理之后，我们可以在磁盘上建立磁盘 Inode 表示一个文件的物理结构。在此基础上，我们需要建立文件系统抽象，对磁盘上的 Inode 进行管理。

在对文件系统进行抽象时，我们需要引入两层文件系统抽象：

- 物理文件系统：负责管理磁盘上的 Inode 和数据块，提供对磁盘 Inode 的访问，在磁盘物理数据上建立文件系统抽象。
- 逻辑文件系统：负责管理文件和目录的元数据，提供对文件和目录的操作接口，对用户提​​供文件系统抽象。

在本节，我们主要介绍物理文件系统的设计和实现。

### 物理文件系统

物理文件系统是建立在物理磁盘块设备上的文件系统，提供了文件系统的底层基本操作，包括创建文件系统、打开文件系统、分配 inode、分配数据块等。

一个文件系统包含了一个块设备，一个 inode 位图，一个数据块位图，inode 区域的起始块号，数据区域的起始块号，结构如下：

```
pub struct FileSystem {
    pub dev: Arc<dyn BlockDevice>,
    pub inode_bitmap: Bitmap,
    pub data_bitmap: Bitmap,
    inode_area_start: u32,
    data_area_start: u32,
}
```

文件系统之下是我们文件系统磁盘数据，包括：

- 超级块 SuperBlock, 包含了文件系统的元信息
- inode 位示图 Bitmap, 用于标记 inode 块的使用情况
- inode 区域, 用于存储 inode 结构
- 数据块位示图 Bitmap, 用于标记数据块的使用情况
- 数据区域, 用于存储文件数据

有了文件系统，我们就可以将这些布局管理起来，实现对磁盘上的文件系统的管理。

### 文件系统操作

#### 创建文件系统

对于一个空的磁盘，或者要使用我们的文件系统的磁盘，我们需要先创建一个文件系统，即对磁盘进行格式化，建立文件系统的基本结构。

创建文件系统时，需要初始化超级块、inode 位图、数据块位图等信息。

我们首先得到了磁盘上所有的块数，我们要计算各个区域的起始块号和大小，然后初始化超级块、inode 位图、数据块位图等信息。

```
let inode_bitmap = Bitmap::new(1, inode_bitmap_blocks as usize);
let inode_count = inode_bitmap.maximum();
// 所有 inode 所需要的块数
let inode_area_blocks =
    (inode_count * core::mem::size_of::<DiskInode>()).div_ceil(BLOCK_SIZE)
as u32;
let inode_total_blocks = inode_bitmap_blocks + inode_area_blocks;
let remaining_blocks = total_blocks - inode_total_blocks - 1;
let data_bitmap_blocks = remaining_blocks.div_ceil(4097);
let data_area_blocks = remaining_blocks - data_bitmap_blocks;
let data_bitmap = Bitmap::new(
    (1 + inode_bitmap_blocks + inode_area_blocks) as usize,
    data_bitmap_blocks as usize,
);
```

需要计算以下信息：

- `inode_count`：inode 数量，我们会将磁盘上的一个区域全部用于 inode，因此 inode 数量等于 inode 位图的最大数量。
- `inode_area_blocks`：inode 区域所占块数，每个磁盘块可以存储多个 inode，因此 inode 区域的大小等于 inode 数量乘以 inode 结构体大小再除以块大小。
- `inode_total_blocks`：inode 区域总块数，inode 区域所占块数加上 inode 位图所占块数。
- `remaining_blocks`：剩余块数，磁盘总块数减去 inode 区域总块数和超级块所占块数。这些块将用于数据区域。
- `data_bitmap_blocks`：数据块位图所占块数，数据块位图的大小等于数据块数量除以块大小。
- `data_area_blocks`：数据区域所占块数，剩余块数减去数据块位图所占块数。

有了这些信息，我们就可以初始化 inode 位图和数据块位图，然后就可以创建文件系统了。

```
let fs = Self {
    dev: dev.clone(),
    inode_bitmap,
    data_bitmap,
    inode_area_start: 1 + inode_bitmap_blocks,
    data_area_start: 1 + inode_total_blocks + data_bitmap_blocks,
};
```

在这之后，我们需要清空磁盘，将超级块信息写入磁盘，创建根目录。

清空磁盘，所有块都置为 0：



```

for i in 0..total_blocks {
    let cache = get_cache(i as usize, dev.clone()).expect("get cache failed");
    cache.lock().modify_at(0, |data_block: &mut DataBlock| {
        data_block.iter_mut().for_each(|x| *x = 0);
    });
}

```

写入超级块信息：

```

let cache = get_cache(0, dev.clone()).expect("get cache failed");
cache.lock().modify_at(0, |super_block: &mut SuperBlock| {
    super_block.init_with(
        total_blocks,
        inode_bitmap_blocks,
        inode_area_blocks,
        data_bitmap_blocks,
        data_area_blocks,
    );
});

```

创建根目录：

```

let (root_inode_bid, root_inode_offset) = fs.get_disk_inode_pos(0);
let cache = get_cache(root_inode_bid as usize, dev.clone()).expect("get cache failed");
cache
    .lock()
    .modify_at(root_inode_offset as usize, |disk_inode: &mut DiskInode| {
        disk_inode.init(InodeType::Dir);
    });

```

## 打开文件系统

在系统启动时，我们需要打开文件系统，加载文件系统的元信息，以便后续对文件系统的操作。

打开文件系统时，我们需要读取超级块信息，初始化 inode 位图和数据块位图。

```

pub fn open(dev: Arc<dyn BlockDevice>) -> Arc<Mutex<Self>> {
    let cache = get_cache(0, dev.clone()).expect("get cache failed");
    let fs = cache
        .lock()
        .read_at(0, |super_block: &SuperBlock| {
            // 初始化文件系统
        })
        .expect("read super block failed");
    Arc::new(Mutex::new(fs))
}

```

在 `read_at` 方法中，我们已经读取了超级块信息，接下来我们需要根据超级块信息初始化 inode 位图和数据块位图。

```
assert!(super_block.check(), "super block magic number error");

let inode_total_blocks =
    super_block.inode_bitmap_blocks + super_block.inode_area_blocks;

let inode_bitmap = Bitmap::new(1, super_block.inode_bitmap_blocks as usize);
let data_bitmap = Bitmap::new(
    (1 + inode_total_blocks) as usize,
    super_block.data_bitmap_blocks as usize,
);

Self {
    dev,
    inode_bitmap,
    data_bitmap,
    inode_area_start: 1 + super_block.inode_bitmap_blocks,
    data_area_start: 1 + inode_total_blocks + super_block.data_bitmap_block
s,
}
```

首先，我们检查超级块的魔数是否正确，魔数是我们用来识别文件系统的标志。然后，我们根据超级块信息初始化 inode 位图和数据块位图。

### 分配和释放 inode

操作 inode 位示图，分配和释放 inode。

```
pub fn alloc_inode(&self) -> u32 {
    self.inode_bitmap
        .alloc(&self.dev)
        .expect("alloc inode failed") as u32
}

pub fn free_inode(&self, inode_id: u32) {
    self.inode_bitmap.free(&self.dev, inode_id as usize);
}
```

分配 inode 时，我们调用 inode 位图的 `alloc` 方法，该方法会返回一个空闲 inode 的编号。释放 inode 时，我们调用 inode 位图的 `free` 方法，将 inode 标记为空闲。

### 分配和释放数据块

操作数据块位示图，分配和释放数据块。

```

pub fn alloc_data_block(&self) -> u32 {
    self.data_bitmap
        .alloc(&self.dev)
        .expect("alloc data block failed") as u32
    + self.data_area_start
}

pub fn free_data_block(&self, data_block_id: u32) {
    let cache = get_cache(data_block_id as usize, self.dev.clone()).expect(
        "get cache failed");
    cache.lock().modify_at(0, |data_block: &mut DataBlock| {
        data_block.iter_mut().for_each(|x| *x = 0);
    });
    self.data_bitmap
        .free(&self.dev, (data_block_id - self.data_area_start) as usize);
}

```

分配数据块时，我们调用数据块位图的 alloc 方法，该方法会返回一个空闲数据块的编号。释放数据块时，我们调用数据块位图的 free 方法，将数据块标记为空闲。

## 获取 inode 位置

根据 inode 编号，获取 inode 在磁盘上的位置。

```

pub fn get_disk_inode_pos(&self, inode_id: u32) -> (u32, usize) {
    let inode_size = core::mem::size_of::<DiskInode>();
    let inode_per_block = (BLOCK_SIZE / inode_size) as u32;
    let inode_bid = self.inode_area_start + inode_id / inode_per_block;
    let inode_offset = (inode_id % inode_per_block) as usize * inode_size;
    (inode_bid, inode_offset)
}

```

我们首先计算 inode 结构体的大小，然后计算每个块可以存储的 inode 数量。根据 inode 编号，我们可以计算 inode 所在的块号和偏移量。

## 获取根目录 inode

根目录 inode 存放在磁盘的第一个 inode 中，我们可以通过 inode 编号 0 获取根目录 inode。

```

fn root_inode(&self) -> MemInode {
    let fs = self.lock();
    let (root_inode_bid, root_inode_offset) = fs.get_disk_inode_pos(0);
    MemInode::new(
        root_inode_bid as usize,
        root_inode_offset as usize,
        self.clone(),
        fs.dev.clone(),
    )
}

```

我们首先获取根目录 inode 在磁盘上的位置，然后创建一个内存 inode 对象。

## 总结

物理文件系统是文件系统的底层抽象，负责管理磁盘上的 inode 和数据块，提供对磁盘 inode 的访问。在本节，我们介绍了物理文件系统的设计和实现，包括创建文件系统、打开文件系统、分配 inode、分配数据块等操作。

在下一节，我们将介绍逻辑文件系统的设计和实现，包括文件和目录的管理接口。

## 文件系统/文件和目录管理

文件和目录管理是文件系统的重要组成部分，负责实现文件的创建、删除、读取、写入等操作，以及目录的创建、删除和内容列举等功能。在本项目中，文件和目录的管理基于以下核心设计：

- **文件与目录的统一抽象**：通过索引节点（MemInode）表示文件和目录的元数据，实现文件与目录管理的统一接口。
- **层次化目录结构**：支持目录树结构，允许文件和子目录嵌套。
- **简洁高效的接口**：提供易用的文件和目录操作接口，屏蔽底层实现细节。

本部分将详细介绍文件和目录管理的实现，包括数据结构设计、操作方法和使用示例。

### 内存 Inode

在物理文件系统之上，我们还需要提供一层虚拟文件系统抽象，用于管理文件和目录的元数据。我们引入内存 Inode（MemInode）结构，用于表示文件和目录的元数据。以下是内存 Inode 的定义：

```
pub struct MemInode {
    block_id: usize,
    block_offset: usize,
    fs: Arc<Mutex<FileSystem>>,
    dev: Arc<dyn BlockDevice>,
}
```

内存 Inode 直接对应物理文件系统中的索引节点（Inode），包含了文件或目录的元数据信息，例如文件大小、类型、数据块指针等。通过这层接口，我们可以屏蔽底层物理文件系统的细节，提供统一的文件和目录管理接口。

### 内存 Inode 方法

#### 创建内存 Inode

在创建时，我们首先需要将内存 Inode 与一个磁盘上未建立的磁盘 Inode 关联起来，并初始化磁盘 Inode 的元数据信息。然后，我们需要更新父目录的目录项，将新文件或目录的元数据信息写入到父目录中。

```

pub fn create(&self, name: &str, inode_type: InodeType) -> Arc<MemInode> {
    let mut fs = self.fs.lock();
    let inode_id = fs.alloc_inode();
    let (block_id, block_offset) = fs.get_disk_inode_pos(inode_id);
    let inode = MemInode::new(
        block_id as usize,
        block_offset,
        Arc::clone(&self.fs),
        Arc::clone(&self.dev),
    );
    let _ = inode.modify_disk_inode(|disk_inode| {
        disk_inode.init(inode_type);
    });
    let _ = self.modify_disk_inode(|disk_inode| {
        let sub_count = (disk_inode.size as usize) / DIR_ENTRY_SIZE;
        let new_size = disk_inode.size + DIR_ENTRY_SIZE as u32;
        self.increase_size(new_size, disk_inode, &mut fs);
        disk_inode.write_at(
            sub_count * DIR_ENTRY_SIZE,
            &self.dev,
            DirEntry::new(inode_id, name).as_bytes(),
        );
    });
    crate::block_cache::block_cache_sync_all();
    Arc::new(inode)
}

```

## 读写内存 Inode

对内存 Inode 的访问都会转化为对底层物理文件系统的磁盘 Inode 的访问，从而实现文件和目录的管理。

这里提供转发读写的方法：

```

fn read_disk_inode<R>(&self, adapter: impl FnOnce(&DiskInode) -> R) -> Option<R> {
    let cache = get_cache(self.block_id, Arc::clone(&self.dev)).expect("get cache failed");
    cache.lock().read_at(self.block_offset, adapter)
}

fn modify_disk_inode<R>(&self, adapter: impl FnOnce(&mut DiskInode) -> R) -> Option<R> {
    let cache = get_cache(self.block_id, Arc::clone(&self.dev)).expect("get cache failed");
    cache.lock().modify_at(self.block_offset, adapter)
}

```

读写操作及其适配器会被直接转发到底层的磁盘 Inode 上。

```

pub fn read_at(&self, offset: usize, buf: &mut [u8]) -> usize {
    let _fs = self.fs.lock();
    let read_size =
        self.read_disk_inode(|disk_inode| disk_inode.read_at(offset, &self.
dev, buf));
    read_size.unwrap()
}

pub fn write_at(&self, offset: usize, buf: &[u8]) -> usize {
    let mut fs = self.fs.lock();
    let write_size = self.modify_disk_inode(|disk_inode| {
        let new_size = offset as u32 + buf.len() as u32;
        self.increase_size(new_size, disk_inode, &mut fs);
        disk_inode.write_at(offset, &self.dev, buf)
    });
    crate::block_cache::block_cache_sync_all();
    write_size.unwrap()
}

```

读写时，我们会先获取文件系统的锁，然后调用 `read_disk_inode` 和 `modify_disk_inode` 方法，将读写操作转发到底层的磁盘 Inode 上。读写适配器指定了将文件或缓冲区的数据写入缓冲区或文件。

### 增加和清除文件大小

磁盘 Inode 提供了增加文件大小的方法，我们只需要将需要的磁盘块分配给磁盘 Inode 增加文件大小即可。

```

pub fn increase_size(
    &self,
    new_size: u32,
    disk_inode: &mut DiskInode,
    fs: &mut MutexGuard<FileSystem>,
) {
    if new_size <= disk_inode.size {
        return;
    }
    let required = disk_inode.required_delta_blocks_for(new_size);
    let new_blocks = (0..required)
        .map(|_| fs.alloc_data_block())
        .collect::<Vec<_>>();
    disk_inode.extend_size(new_size, new_blocks, &self.dev);
}

```

这里，我们首先检查新的文件大小是否大于当前文件大小，然后计算需要增加的数据块数量，分配数据块并更新文件大小。

同样，我们还提供了清除文件大小的方法，用于释放数据块。

```
pub fn clear(&self) {
    let fs = self.fs.lock();
    self.modify_disk_inode(|disk_inode| {
        let _size = disk_inode.size;
        let deallocated = disk_inode.clear_size(&self.dev);
        deallocated.into_iter().for_each(|block_id| {
            fs.free_data_block(block_id);
        });
    })
    .expect("modify disk inode failed");
    crate::block_cache::block_cache_sync_all();
}
```

释放的数据块会被传入到 `free_data_block` 方法中，告知文件系统释放这些数据块。

## 查找文件或目录

在一些场景下，我们需要查找文件或目录，以通过文件名查找文件或目录的内存 Inode。

```

pub fn find(&self, name: &str) -> Option<Arc<MemInode>> {
    let fs = self.fs.lock();
    let inode = self
        .read_disk_inode(|disk_inode| {
            self.find_inode_id(name, disk_inode).map(|inode_id| {
                let (block_id, block_offset) = fs.get_disk_inode_pos(inode_
id);

                let inode = MemInode::new(
                    block_id as usize,
                    block_offset,
                    Arc::clone(&self.fs),
                    Arc::clone(&self.dev),
                );
                inode
            })
        })
        .expect("read disk inode failed");
    inode.map(|inode| Arc::new(inode))
}

pub fn find_inode_id(&self, name: &str, disk_inode: &DiskInode) -> Option<u
32> {
    assert!(disk_inode.get_type() == InodeType::Dir);
    let sub_count = (disk_inode.size as usize) / DIR_ENTRY_SIZE;
    for sub_id in 0..sub_count {
        let sub_offset = sub_id * DIR_ENTRY_SIZE;
        let mut buf = DirEntry::default();
        disk_inode.read_at(sub_offset, &self.dev, buf.as_mut_bytes());
        let buf_name = buf.name();
        if buf_name == name || &buf_name[..buf_name.len() - 1] == name {
            return Some(buf.inode());
        }
    }
    None
}

```

查找主要由 `find_inode_id` 方法实现，该方法会遍历目录项，查找指定名称的目录项，并返回对应的 inode 编号。

`find` 方法在使用 `find_inode_id` 方法查找到 inode 编号后，创建内存 Inode 并返回。

目前我们只支持单级目录。

## 列出目录内容



```

pub fn ls(&self) -> Vec<String> {
    let mut res = Vec::new();
    let _fs = self.fs.lock();
    self.read_disk_inode(|disk_inode| {
        assert!(disk_inode.get_type() == InodeType::Dir);
        let sub_count = (disk_inode.size as usize) / DIR_ENTRY_SIZE;
        for sub_id in 0..sub_count {
            let sub_offset = sub_id * DIR_ENTRY_SIZE;
            let mut buf = DirEntry::default();
            disk_inode.read_at(sub_offset, &self.dev, buf.as_mut_bytes());
            res.push(buf.name().to_string());
        }
    })
    .expect("read disk inode failed");
    res
}

```

在 `ls` 方法中，我们首先获取文件系统的锁，然后读取目录项，将目录项的名称添加到列表中。

## 获取元数据

元数据存储在磁盘 Inode 中，这里提供转发获取元数据的方法：

```

pub fn get_size(&self) -> u32 {
    let size = self
        .read_disk_inode(|disk_inode| disk_inode.size)
        .expect("read disk inode failed");
    size
}

pub fn get_type(&self) -> InodeType {
    let inode_type = self
        .read_disk_inode(|disk_inode| disk_inode.get_type())
        .expect("read disk inode failed");
    inode_type
}

```

读取元数据时，只需简单地读取磁盘 Inode 中的元数据即可。

## 文件目录管理

文件目录管理是文件系统的核心功能之一，有了内存 Inode，我们就可以实现文件和目录的管理。

文件目录管理在操作系统内核中实现。

### OS Inode

Windows/Linux 操作系统提供的逻辑文件都是 *顺序文件 + 暴露读写指针* 的形式。在顺序文件中，文件的读写指针是文件的一部分，文件的读写指针指向文件中的某个位置，文件的读写操作

都是基于读写指针进行的。

在内存 Inode 之上，我们还需要封装一层 OS Inode，用于管理文件的读写指针、文件状态等信息。

```
pub struct OSInode {
    readable: bool,
    writable: bool,
    inode: Mutex<InodeData>,
}

pub struct InodeData {
    pub inode: Arc<MemInode>,
    pub offset: usize,
}
```

OS Inode 包含了文件的读写权限、内存 Inode 以及读写指针等信息。通过 OS Inode，我们可以实现文件的读写操作。

## 文件读写

这里的文件读写是面向文件系统使用者的，我们提供了简单的读写接口，用于读写文件。

读写时，我们首先需要获取文件系统的锁，会遍历用户缓冲区，然后调用内存 Inode 的读写方法，将读写操作转发到内存 Inode 上，逐个读写缓冲区的数据。

同时，我们还需要更新文件的读写指针，以便下次读写操作。

```

fn read(&self, mut buf: UserBuffer) -> usize {
    let mut inode = self.inode.lock();
    let inode = &mut *inode;
    let mut read_size = 0;
    for i in 0..buf.buffer.len() {
        let newly_read = inode.inode.read_at(inode.offset, buf.buffer[i]);
        if newly_read == 0 {
            break;
        }
        inode.offset += newly_read;
        read_size += newly_read;
    }
    read_size
}

fn write(&self, buf: UserBuffer) -> usize {
    let mut inode = self.inode.lock();
    let inode = &mut *inode;
    let mut write_size = 0;
    for i in 0..buf.buffer.len() {
        let newly_written = inode.inode.write_at(inode.offset, buf.buffer
[i]);
        if newly_written == 0 {
            break;
        }
        inode.offset += newly_written;
        write_size += newly_written;
    }
    write_size
}

```

## 操作系统文件访问接口

到目前为止，我们已经实现了内存 Inode 和 OS Inode，以及文件的读写操作。这意味着我们可以分层、精确地管理文件。

在操作系统中，我们还需要提供文件访问接口，用于打开、关闭、读写文件。

打开文件时，我们需要根据文件名和打开标志查找文件，然后创建 OS Inode，用于管理文件的读写指针。

```
pub fn open_file(name: &str, flags: OpenFlags) -> Option<Arc<OSInode>> {
    let readable = flags.readable();
    let writable = flags.writable();
    if flags.contains(OpenFlags::CREATE) {
        return create_file(name, flags);
    }
    let inode = ROOT_INODE.find(name)?;
    if flags.contains(OpenFlags::TRUNCATE) {
        inode.clear();
    }
    Some(Arc::new(OSInode::new(readable, writable, inode)))
}
```

对于带有 `CREATE` 标志的打开，我们会调用 `create\_file` 方法创建文件。

```
```rust
fn create_file(name: &str, flags: OpenFlags) -> Option<Arc<OSInode>> {
    let readable = flags.readable();
    let writable = flags.writable();
    if let Some(inode) = ROOT_INODE.find(name) {
        inode.clear();
        return Some(Arc::new(OSInode::new(readable, writable, inode)));
    }
    let inode = ROOT_INODE.create(name, InodeType::File);
    Some(Arc::new(OSInode::new(readable, writable, inode)))
}
```

创建文件时，我们首先查找文件，如果文件已经存在，则清除文件内容；否则，创建文件。

有了访问接口，我们就可以提供相应的系统调用了，包括：

- openat：打开文件
- close：关闭文件
- read：读取文件
- write：写入文件 具体的系统调用实现见 [系统调用](#) 等。

## 总结

文件系统是操作系统的重要组成部分，负责管理文件和目录，提供文件的读写、创建、删除等功能。在本项目中，我们实现了一个简单的文件系统，包括物理文件系统、内存 Inode、OS Inode 等模块，提供了文件和目录的管理接口。

# 系统调用

## 系统调用的原理

用户在程序中调用操作系统所提供的一些子功能。通常也把被调用的操作系统功能，称为系统调用。

## 用户态与内核态的隔离

操作系统为了保证系统的安全性和稳定性，将处理器的执行模式分为用户态和内核态。用户程序在用户态下运行，受到诸多限制，例如不能直接访问硬件设备、不能随意修改系统关键数据结构等。而内核态则拥有最高权限，可以执行诸如设备驱动、内存管理、进程调度等关键操作。

这种隔离机制是基于处理器的硬件支持实现的。在 CPU 的特权级机制下，不同的指令在不同的特权级下有不同的执行权限。某些特权指令只能在内核态下执行。

## 系统调用与函数调用的区别

### 1. 进入和退出方式不同

**系统调用：** INT/IRET

**函数调用：** CALL/RET

### 2. CPU状态变化不同

**系统调用：** 用户态 → 系统态 → 用户态

**函数调用：** 无CPU状态变化

## 系统调用的触发机制

- **指令陷入：**

当用户程序需要执行内核提供的服务时，通过执行一条特殊的陷入指令来发起系统调用。这条指令会导致处理器从用户态切换到内核态，并将执行流程转移到预先定义好的内核入口点。

- **参数传递：**

在触发系统调用之前，用户程序需要将系统调用号和相关参数传递给内核。在RISC - V架构中，系统调用号存放在寄存器a7，而系统调用的参数则存放在寄存器a0 - a6中。

## 系统调用处理流程

- **中断向量表与入口点：**

当陷入指令执行后，处理器会根据中断向量表来确定系统调用的内核入口点。这个入口点通常是一个通用的系统调用处理程序或分发器。

- **系统调用分发器：**

内核中的系统调用分发器会首先获取系统调用号，然后根据这个号码来调用相应的具体系统调用处理函数。这个分发器就像是一个调度中心，它根据用户程序的请求将执行流程引导到正确的内核服务模块。

- **具体系统调用处理函数：**

每个具体的系统调用处理函数实现了特定的内核服务。以文件读写系统调用为例，文件读写处理函数会与文件系统模块交互，执行诸如查找文件、验证权限、读写磁盘数据等操作。这些处理函数可以访问内核的数据结构和硬件资源，因为它们是在内核态下执行的。

## 内核态到用户态的返回机制

在系统调用处理函数完成服务后，它需要将执行流程从内核态切换回用户态，以使用户程序能够继续执行。这个返回过程通常涉及恢复用户程序在触发系统调用时保存的上下文信息，包括寄存器状态、程序计数器等。

内核会将系统调用的返回值存放在寄存器a0中，然后通过执行sret来切换回用户态。用户程序在返回后可以从这个寄存器中获取系统调用的返回值，并根据返回值继续执行后续操作。

## 系统调用的安全性和可靠性保障

- **参数验证：**

内核在处理系统调用时，会对用户程序传递的参数进行验证。例如，对于文件操作的系统调用，内核会检查文件路径是否合法、读写权限是否符合要求等。这样可以防止用户程序通过恶意参数来破坏系统的稳定性或访问未经授权的资源。

- **上下文保存与恢复：**

系统调用过程中准确地保存和恢复用户程序的上下文是保证系统调用可靠性的关键。如果上下文恢复出现错误，可能会导致用户程序执行异常或崩溃。因此，操作系统会使用专门的数据结构来安全地保存和恢复这些信息。

## 系统调用的过程

### 1. 用户程序发起调用

- **参数准备：**当用户程序需要使用操作系统提供的某种服务时，如文件读写、进程创建等，首先会在用户空间准备好系统调用所需的参数。
- **系统调用号指定：**每个系统调用都有一个对应的系统调用号。用户程序需要以某种方式指定要执行的系统调用号，这个号码用于在内核中识别具体的系统调用。
- **触发调用指令执行：**在参数准备好和系统调用号确定后，用户程序通过执行一条特殊的指令来触发系统调用。在我们的操作系统中，使用的是 RISC - V 架构中的“ecall”指令，这条指令会导致处理器从用户态切换到内核态，开启系统调用的处理流程。

### 2. 陷入内核态及参数传递

- **陷入机制：**当触发系统调用的指令执行后，处理器会根据硬件的中断机制进入内核态。这个过程类似于硬件中断，但系统调用是一种软件触发的、有目的的陷入。在进入内核态时，处理器会保存当前用户程序的一些关键上下文信息，如程序计数器（PC）、寄存器状态等，以便在系统调用返回后能够恢复用户程序的执行。
- **参数传递方式：**我们通过将系统调用号存放在指定的寄存器中来实现对系统调用的参数传递。其中a0-a2 存放参数，a7存放系统调用号，a0存放返回值。这样，内核可以从这些寄存器中获取系统调用号和参数信息。

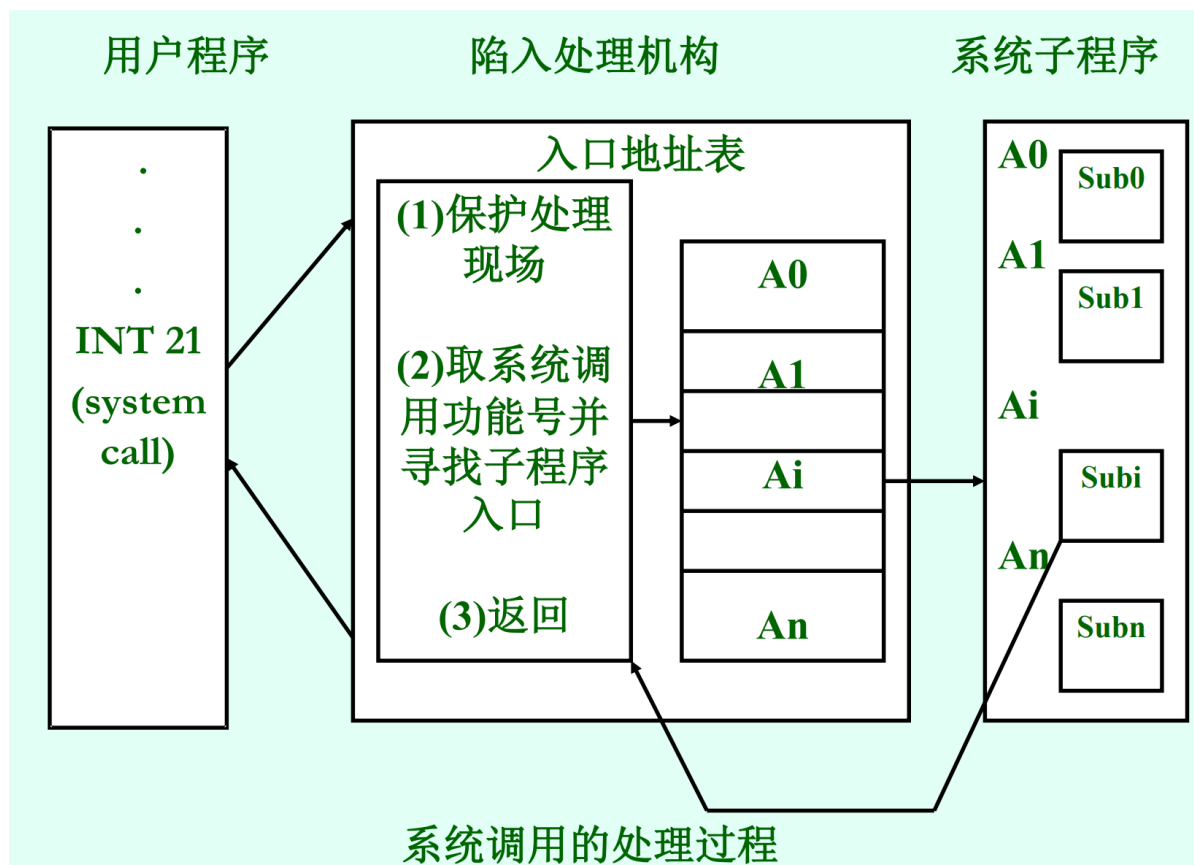
### 3. 内核中的系统调用分发与处理

- **系统调用分发器：**内核中有一个系统调用分发器，它是系统调用处理的入口点。分发器首先会从相应的寄存器中获取系统调用号。然后，根据这个系统调用号，通过一个类似于“match”的机制，将执行流程引导到具体的系统调用处理函数。
- **具体系统调用处理函数：**每个系统调用号对应一个具体的处理函数，这些函数实现了特定的内核服务。我们实现的系统调用函数包括：sys\_openat、sys\_close、sys\_read、sys\_write、sys\_exit、sys\_yield、sys\_get\_time\_of\_day、sys\_shutdown、sys\_clone、

sys\_execve、sys\_wait4。这些处理函数可以访问和操作内核的数据结构和硬件资源，因为它们是在内核态下执行的。

#### 4. 内核态到用户态的返回

- **返回值设置**：在系统调用处理函数完成其任务后，会将返回值放置在寄存器a0中。这个返回值可以是系统调用的执行结果，如文件打开系统调用返回的文件描述符，或者是错误码，表示系统调用执行过程中出现的问题。
- **恢复用户程序上下文**：内核会从之前保存的用户程序上下文信息中恢复寄存器状态、程序计数器等内容，然后通过执行“sret”指令将处理器的执行模式从内核态切换回用户态。这样，用户程序就可以从系统调用指令的下一条指令开始继续执行，并从寄存器a0中获取系统调用的返回值。



## 系统调用的实现

### 系统调用号定义与分配

我们使用枚举类型来定义为每个系统调用分配一个唯一的编号。

```
#[derive(Debug, Clone, Copy)]
pub enum Syscall {
    OpenAt = 56,           // 打开文件
    Close = 57,            // 关闭文件
    Read = 63,             // 从文件描述符读取
    Write = 64,            // 写入文件描述符
    Exit = 93,             // 终止当前进程
    SchedYield = 124,      // 让出处理器
    GetTimeOfDay = 169,    // 读取当前时间
    Shutdown = 210,        // 关机 (shutdown)
    Clone = 220,           // 创建子进程 (fork)
    Execve = 221,          // 执行程序 (exec)
    Wait4 = 260,           // 等待子进程 (waitpid)
}
```

这样在用户程序发起系统调用时，通过传递这个编号，内核就能确定具体要执行的系统调用服务。

## 系统调用接口函数实现

在用户空间提供一组与系统调用对应的函数声明，这些函数会进行一些参数检查和预处理工作，然后通过RISC-V 架构中的`ecall`指令触发系统调用陷入内核。

系统调用函数名	参数说明	参数类型
<code>sys_close</code>	<code>fd</code> : 文件描述符	<code>fd</code> : <code>usize</code> 类型
<code>sys_openat</code>	<code>path</code> : 文件路径 <code>flags</code> : 打开文件相关标志	<code>path</code> : <code>&amp;str</code> 类型 <code>flags</code> : <code>usize</code> 类型
<code>sys_read</code>	<code>fd</code> : 文件描述符 <code>buffer</code> : 用于存储读取数据的缓冲区	<code>fd</code> : <code>usize</code> 类型 <code>buffer</code> : <code>&amp;mut [u8]</code> 类型
<code>sys_write</code>	<code>fd</code> : 文件描述符 <code>buffer</code> : 要写入的数据	<code>fd</code> : <code>usize</code> 类型 <code>buffer</code> : <code>&amp;mut [u8]</code> 类型
<code>sys_exit</code>	<code>code</code> : 退出码	<code>code</code> : <code>usize</code> 类型
<code>sys_sched_yield</code>	无	无
<code>sys_get_time_of_day</code>	<code>ts</code> : 指向 <code>TimeVal</code> 结构体的指针，用于存储当前时间 <code>tz</code> : 指向 <code>TimeZone</code> 结构体的指针，用于存储时区信息	<code>ts</code> : <code>*mut TimeVal</code> 类型 <code>tz</code> : <code>*mut TimeZone</code> 类型
<code>sys_shutdown</code>	无	无
<code>sys_clone</code>	无	无
<code>sys_execve</code>	<code>path</code> : 程序路径	<code>path</code> : <code>&amp;str</code> 类型



系统调用函数名	参数说明	参数类型
sys_wait4	pid: 子进程的进程 ID exit_code: 指向i32类型的指针, 用于存储子进程的退出码	pid: isize类型 exit_code: *mut i32类型

## sys\_openat:

sys\_openat函数主要用于实现打开文件相关的系统调用功能。

首先，为了后续能基于当前处理器所处的进程环境来进行相应操作，我们使用Processor::get\_current().unwrap()获取当前的处理器相关信息。接着，调用get\_translated\_string函数对传入的文件路径进行处理，在此过程中，利用Processor::current\_user\_token().unwrap()获取当前用户的相关令牌，并将原始的文件路径指针path传入，得到可识别的字符串形式的路径，方便后续操作。然后，调用open\_file函数并传入转换后的文件路径字符串以及由OpenFlags::from\_bits(flags).unwrap()解析出的打开文件标志，尝试打开文件，若成功打开则会得到对应的文件inode信息。如果成功获取到了inode，那么就通过当前处理器对应的可修改状态来分配一个文件描述符，即调用alloc\_fd方法获取可用的文件描述符编号fd，随后将获取到的inode存放到当前处理器对应进程的文件描述符表中，最后将该文件描述符编号fd转换为SyscallRet类型作为函数的返回值，表示文件成功打开并返回对应的文件描述符；而如果open\_file函数未能成功打开文件，也就是没有获取到相应的inode，此时函数则直接返回-1，以此来告知调用者文件打开操作失败。

```
pub fn sys_openat(path: *const u8, flags: usize) -> SyscallRet {
    let current = Processor::get_current().unwrap();
    let path = get_translated_string(Processor::current_user_token().unwrap(), path as *const u8);
    let inode = open_file(path.as_str(), OpenFlags::from_bits(flags).unwrap());
    if let Some(inode) = inode {
        let fd = current.inner_borrow_mut().alloc_fd();
        current.inner_borrow_mut().fd_table[fd] = Some(inode);
        fd as SyscallRet
    } else {
        -1
    }
}
```

## sys\_close

sys\_close函数的功能主要是实现关闭文件描述符对应的文件这一系统调用操作。

首先，函数通过Processor::get\_current().unwrap()获取当前处理器相关的信息，然后，再利用current.inner\_borrow\_mut()以可变借用的方式得到对应的进程控制块（pcb），这是因为后续要对进程控制块里的文件描述符表进行操作，所以需要可变访问权限。接着，会进行两项检查，一是判断传入的文件描述符fd是否超出了当前进程控制块中文件描述符表的长度范围，如果fd大于等于pcb.fd\_table.len()，意味着传入的文件描述符是不合法的，此时函数直接返回-1，表示关闭文件操作失败；二是检查文件描述符表中对应位置的元素是否为None，即

`pcb.fd_table[fd].is_none()`，若此条件成立，说明该文件描述符并没有关联有效的文件资源，同样属于不合法的情况，也会返回-1来表示操作失败。只有当上述两项检查都通过时，才会执行关闭操作，也就是将文件描述符表中对应位置的元素设置为None，即`pcb.fd_table[fd]=None`，以此释放该文件描述符所关联的文件资源，完成关闭文件的操作，最后函数返回0，代表文件关闭成功。

```
pub fn sys_close(fd: usize) -> SyscallRet {
    let current = Processor::get_current().unwrap();
    let mut pcb = current.inner_borrow_mut();
    if fd >= pcb.fd_table.len() {
        return -1;
    }
    if pcb.fd_table[fd].is_none() {
        return -1;
    }
    pcb.fd_table[fd] = None;
    0
}
```

## sys\_read

`sys_read`函数旨在实现从指定文件描述符对应的文件中读取数据这一系统调用功能。

首先，通过`Processor::current_user_token().unwrap()`获取当前用户的相关令牌。接着，利用`Processor::get_current().unwrap()`获取当前处理器相关信息，因为后续需要依据进程控制块里文件描述符表的情况来判断操作的合法性，所以我们使用`task.inner_borrow()`以不可变借用的方式获取对应的进程控制块（pcb）。

随后，会进行两项重要的合法性检查。第一项检查和上面的关闭文件的检查相同，主要检查文件描述符是否超出当前进程控制块中文件描述符表的长度范围；第二项检查是在文件描述符合法的前提下，通过`if let Some(file)= &pcb.fd_table[fd]`语句判断文件描述符表中对应位置的元素是否存在有效的文件对象，若存在，则克隆该文件对象，并进一步检查该文件是否可读，若!`(file.readable())`，也就是文件不具备可读属性，同样返回-1来表示读取操作失败。

当上述合法性检查全部通过后，才会执行读取操作。先是通过`drop(pcb)`语句释放之前获取的进程控制块的不可变借用，然后调用文件对象的`read`方法，传入根据用户令牌、传入的缓冲区指针`buffer`以及期望读取的长度`len`所构建的`UserBuffer`，尝试从文件中读取相应长度的数据，最后将读取操作返回的字节数转换为`isize`类型作为函数的返回值，若成功读取到数据则返回对应字节数，若读取失败则返回-1，以此作为此次系统调用`sys_read`的执行结果。

```

pub fn sys_read(fd: usize, buffer: *mut u8, len: usize) -> SyscallRet {
    let token = Processor::current_user_token().unwrap();
    let task = Processor::get_current().unwrap();
    let pcb = task.inner_borrow();
    if fd >= pcb.fd_table.len() {
        return -1;
    }
    if let Some(file) = &pcb.fd_table[fd] {
        let file = file.clone();
        if !(file.readable()) {
            return -1;
        }
        drop(pcb);
        file.read(UserBuffer::new(get_mut_translated_byte_slices(
            token, buffer, len,
        ))) as isize
    } else {
        -1
    }
}

```

## sys\_write

sys\_write函数主要用于实现向指定文件描述符对应的文件写入数据的系统调用功能。

前面的检查和sys\_read类似，只有在最后检查的时候，检查的是文件对象的可写属性而不是可读属性，具体的实现方法是：通过if let Some(file)= &pcb.fd\_table[fd]语句查看该文件描述符对应位置是否存在有效的文件对象，若存在则克隆这个文件对象，之后进一步检查该文件是否具备可写属性，若!(file.writable())，也就是文件不可写，同样返回-1来表示此次写入操作失败。

当上述所有合法性检查都顺利通过后，才会执行写入操作。先是通过drop(pcb)释放之前获取的进程控制块的不可变借用，然后调用克隆后的文件对象的write方法，传入依据用户令牌、传入的缓冲区指针buffer以及期望写入的长度len所构建的UserBuffer，尝试将相应数据写入文件中，最后把写入操作实际写入的字节数转换为isize类型作为函数的返回值，若成功写入了数据则返回对应字节数，若写入失败则返回-1，以此来体现sys\_write这个系统调用的执行结果。

```
pub fn sys_write(fd: usize, buffer: *const u8, len: usize) -> SyscallRet {
    let token = Processor::current_user_token().unwrap();
    let task = Processor::get_current().unwrap();
    let pcb = task.inner_borrow();
    if fd >= pcb.fd_table.len() {
        return -1;
    }
    if let Some(file) = &pcb.fd_table[fd] {
        let file = file.clone();
        if !(file.writable()) {
            return -1;
        }
        drop(pcb);
        file.write(UserBuffer::new(get_mut_translated_byte_slices(
            token, buffer, len,
        ))) as isize
    } else {
        -1
    }
}
```

## sys\_exit

sys\_exit函数的主要功能是处理进程的退出操作。

当该函数被调用时，会接收一个表示退出码的i32类型参数code，这个退出码用于向父进程或者系统反馈当前进程结束时的状态信息。接着，在函数内部调用task::manager::TaskManager::replace\_to\_next(code)，执行后续的进程退出及切换相关操作。

```
pub fn sys_exit(code: i32) {
    task::manager::TaskManager::replace_to_next(code);
}
```

## sys\_yield

sys\_yield函数主要用于实现进程主动让出处理器资源的功能，也就是通常所说的“让步”操作。

当该函数被调用时，它会首先调用task::manager::TaskManager::cycle\_to\_next()这一操作。在这一过程中，任务管理器（TaskManager）会依据系统预先设定好的任务调度策略（先来先服务+时间片轮转）来执行相应的进程切换逻辑。其会先暂停当前正在运行的进程，将其相关的执行上下文（寄存器状态、程序计数器等关键信息）进行保存，以便后续该进程再次被调度执行时可以准确地恢复现场。然后，从就绪队列或者其他符合调度条件的进程集合中挑选出下一个合适的进程，将其对应的执行上下文信息恢复到处理器的相关寄存器等硬件资源上，使得下一个进程能够开始执行，从而实现进程之间的切换。在完成上述进程切换操作后，函数直接返回0，以此来表示sys\_yield这个系统调用执行成功，即当前进程已经成功让出处理器资源，让系统可以调度其他进程继续运行。

```
pub fn sys_yield() -> SyscallRet {
    task::manager::TaskManager::cycle_to_next();
    0
}
```

## sys\_get\_time\_of\_day

sys\_get\_time\_of\_day函数主要用于获取当前的时间信息并填充到相应的结构体中，以此来响应对应的系统调用需求。

首先，针对传入的用于存储时间值的指针ts，调用get\_translated\_byte\_slices函数来获取经过转换后的字节切片信息。为了确保获取到符合当前用户权限且正确长度的字节切片，在调用这个函数时，传入当前用户的相关令牌、将ts指针转换为\*const u8类型以及TimeVal结构体的大小。接着，通过unsafe块，利用获取到的字节切片信息，将其首字节的指针转换为\*mut TimeVal类型，并解引用得到可变引用ts，这样就能对其内部成员进行赋值操作了。

同样地，对于传入的用于存储时区信息的指针tz（类型为\*mut TimeZone），也执行类似的操作。

之后，调用get\_time\_us函数获取当前时间，其返回值以微秒为单位。然后，依据获取到的时间信息进行赋值操作，将总时间值除以1000000得到的商赋值给ts.tv\_sec，代表秒数；将总时间值除以1000000的余数赋值给ts.tv\_usec，代表微秒数，传给TimeVal结构体中的时间部分。然后，将tz.tz\_minuteswest赋值为0，表示时区的分钟偏移量为0；将tz.tz\_dsttime赋值为0，表示夏令时相关的设置为0。若执行完成，则返回0。

```
pub fn sys_get_time_of_day(ts: *mut TimeVal, tz: *mut TimeZone) -> SyscallRet {
    let ts_buffer = get_translated_byte_slices(
        Processor::current_user_token().unwrap(),
        ts as *const u8,
        core::mem::size_of::<TimeVal>(),
    );
    let ts = unsafe { &mut *(ts_buffer[0].as_ptr() as *mut TimeVal) };
    let tz_buffer = get_translated_byte_slices(
        Processor::current_user_token().unwrap(),
        tz as *const u8,
        core::mem::size_of::<TimeZone>(),
    );
    let tz = unsafe { &mut *(tz_buffer[0].as_ptr() as *mut TimeZone) };
    let time = get_time_us();
    ts.tv_sec = time / 1_000_000;
    ts.tv_usec = time % 1_000_000;
    tz.tz_minuteswest = 0;
    tz.tz_dsttime = 0;
    0
}
```

## sys\_shutdown

sys\_shutdown函数的核心作用是执行系统关闭相关的操作。

函数调用了`crate::sbi::sbi_shutdown(false)`；通过调用`sbi_shutdown`函数来触发实际的系统关闭动作。

对于`sbi_shutdown`函数，当传入的`failure`参数为`true`时，意味着系统是因为出现故障而要执行关机操作；而当`failure`参数为`false`时，表明是正常的关机情况，并非由于系统故障导致。由于我们传入的参数是`false`，说明属于是正常关机。

```
pub fn sys_shutdown() -> ! {
    warn!("System shutdown");
    crate::sbi::sbi_shutdown(false);
}

pub fn sbi_shutdown(failure: bool) -> ! {
    if failure {
        rustsbi::Forward {}.system_reset(
            ResetType::Shutdown.into(),
            ResetReason::SystemFailure.into(),
        );
    } else {
        rustsbi::Forward {}.system_reset(ResetType::Shutdown.into(), ResetReason::NoReason.into());
    }
    unreachable!("sbi_shutdown");
}
```

## sys\_clone

`sys_clone`函数主要用于实现创建新进程的系统调用功能，实现进程的复制与新进程的相关设置及管理。

首先，通过`Processor::get_current().unwrap()`获取当前处理器信息。接着，调用`current.fork()`方法创建一个新进程，这里的`fork`操作会基于当前进程的状态进行复制，同时为新进程分配独立的内核栈、进程标识符等必要元素，使得新进程与当前进程有相似的初始状态，不过二者后续会独立运行。

随后，通过`new_task.inner_borrow().pid.0`获取新进程的进程标识符（`pid`）。再获取新进程的陷阱上下文，并将其参数寄存器`a(0)`的值设置为`0`，这样做的目的是规定在新进程开始执行时，其从系统调用返回的值为`0`，符合通常情况下子进程创建后`clone`系统调用对于子进程返回值的设定逻辑。

完成上述新进程相关的初始设置后，将新进程添加到任务管理器中。把新进程添加进去后，按照先来先服务+实现片轮转的调度策略参与到后续的运行流程中。

最后，将新进程的进程标识符（`pid`）转换为`SyscallRet`类型并返回，这个返回值对于父进程来说，用于后续对新创建子进程的相关操作或状态查询。

```
pub fn sys_clone() -> SyscallRet {
    let current = Processor::get_current().unwrap();
    let new_task = current.fork();
    let new_task_pid = new_task.inner_borrow().pid.0;
    let ctx = new_task.inner_borrow().get_trap_cx();
    *ctx.a(0) = 0; // return 0 for the child process
    TaskManager::put_task(new_task);
    new_task_pid as SyscallRet
}
```

## sys\_execve

sys\_execve函数主要用于实现执行新程序的系统调用功能，它会替换当前进程的执行内容，使其开始运行指定的新程序。

首先，函数通过调用get\_translated\_string函数对传入的文件路径指针path进行处理，获取符合当前用户权限且经过转换后的、可用的文件路径字符串。

接着，使用open\_file函数以只读方式打开对应路径的文件，如果能够成功打开，就会获取到对应的文件inode信息。随后，调用inode的read\_all方法读取文件中的所有数据，并通过as\_slice方法将读取到的数据转换为切片形式存储在data变量中，这个data切片便包含了要执行的新程序的相关数据内容，比如可执行文件的二进制数据等。

之后，调用Processor::get\_current().unwrap()获取当前处理器信息，进而调用当前进程的exec方法，并将前面获取到的包含新程序数据的data切片传入。在这些操作都顺利完成后，函数返回0，表示新程序成功加载并开始执行。否则函数返回-1，表示这个系统调用执行失败，没能成功执行新的程序。

```
pub fn sys_execve(path: *const u8) -> SyscallRet {
    let path = get_translated_string(Processor::current_user_token().unwrap(
    ), path as *const u8);
    if let Some(app_inode) = open_file(path.as_str(), OpenFlags::READONLY)
    {
        let data = app_inode.read_all();
        let data = data.as_slice();
        let current = Processor::get_current().unwrap();
        current.exec(data);
        0
    } else {
        println!("Failed to load app data: {}\n", path);
        -1
    }
}
```

## sys\_wait4

sys\_wait4函数主要用于实现父进程等待子进程结束并获取子进程退出码的系统调用功能。

首先，获取当前处理器信息，获取对应的进程控制块。接着，初始化`is_child_stopped`和`child_index`分别初始化为`false`和`None`。然后，通过一个循环遍历`pcb.children`列表中的所有子进程，在每次循环中，先获取子进程的进程控制块（`child_pcb`），接着判断子进程的进程标识符（`child_pcb.pid.0`）与传入的`pid`是否相等或者传入的`pid`是否为-1，如果条件为真，就将当前子进程在列表中的索引位置记录到`child_index`变量中，并且进一步判断该子进程的状态，如果其状态为`ProcessStatus::Stopped`，则将`is_child_stopped`变量设置为`true`。完成遍历判断后，如果`child_index`仍然为`None`，说明没有找到符合条件的子进程，此时函数直接返回-1，表示等待子进程操作失败。

而当`is_child_stopped`为`true`时，也就是找到了处于停止状态的符合条件的子进程，会先从`pcb.children`列表中移除该子进程，然后使用断言确保此时该子进程的强引用计数为1，即只有当前父进程在引用它，避免出现意外的资源管理问题。接着，再次获取已移除子进程的进程控制块，从中获取子进程的退出码（`child_pcb.exit_code`）以及进程标识符（`child_pcb.pid.0`），再通过`get_translated_refmut`函数将子进程的退出码赋值到对应的内存位置，最后将子进程的进程标识符返回，代表成功等到了指定子进程结束，并返回了子进程的相关信息。

但若`is_child_stopped`为`false`，也就是找到了符合条件的子进程但它还未处于停止状态，此时函数返回-2，表示子进程尚未停止，还需继续等待。



```

pub fn sys_wait4(pid: isize, exit_code_ptr: *mut i32) -> SyscallRet {
    let task = Processor::get_current().unwrap();
    let mut pcb = task.inner_borrow_mut();

    let mut is_child_stopped = false;
    let mut child_index = None;

    for (index, child) in pcb.children.iter().enumerate() {
        let child_pcb = child.inner_borrow();
        if child_pcb.pid.0 == pid as usize || pid == -1 {
            child_index = Some(index);
            if child_pcb.status == ProcessStatus::Stopped {
                is_child_stopped = true;
            }
        }
    }

    if child_index.is_none() {
        return -1;
    }

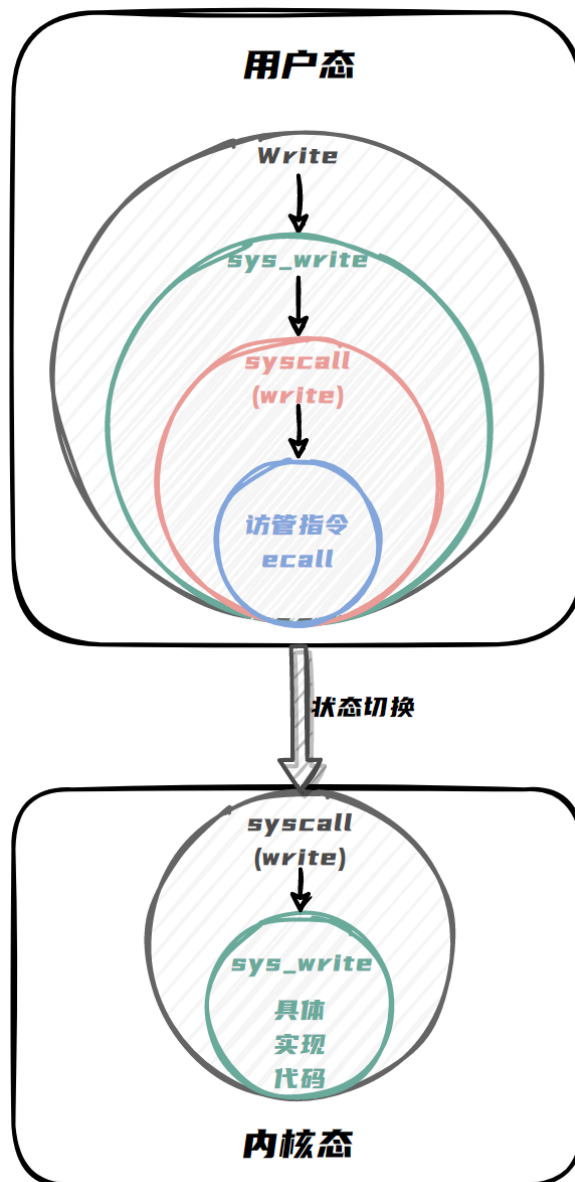
    if is_child_stopped {
        let child = pcb.children.remove(child_index.unwrap());
        assert_eq!(Arc::strong_count(&child), 1);
        let child_pcb = child.inner_borrow();
        let child_exit_code = child_pcb.exit_code;
        let child_pid = child_pcb.pid.0;
        *get_translated_refmut(pcb.memory_set.token(), exit_code_ptr) = child_exit_code;

        child_pid as SyscallRet
    } else {
        -2
    }
}

```

## 系统调用的封装

操作系统为了保证系统的安全性和稳定性，将处理器的执行模式分为用户态和内核态。而且操作系统的底层系统调用通常与硬件和内核的细节紧密相关，基于安全性原则，用户态不应该能直接使用系统调用，也就是说我们不能直接暴露系统调用接口，因此需要对系统调用的封装。系统调用的封装结构如下图所示：



用户态函数名	内核态函数名
close	sys_close
openat	sys_openat
read	sys_read
write	sys_write
exit	sys_exit
sched_yield	sys_sched_yield
get_time_of_day	sys_get_time_of_day
shutdown	sys_shutdown
fork	sys_clone
execve	sys_execve
wait	sys_wait4

# 功能测试

本文档将在用户程序层面综合测试内核的功能。

主要从内核启动和关闭、用户态程序和命令行参数解析、进程调度、文件系统等方面进行测试，展示内核的功能。

## 1. 内核启动和关闭

使用 `make launch-qemu-system.release` 启动内核。

首先编译用户程序和内核，并打包成镜像文件。

```
zmake x startup-test:make
> make launch-qemu-system.release
cargo build -p user --release
Compiling user v0.1.0 (/home/gerrnperl/dev/startup-test/user)
Finished 'release' profile [optimized] target(s) in 0.57s
cargo run -p ros-fs-fuse -- --app initproc sh hello bye fstest rrttest ls cat touchwith shutdown echo calc --target target/riscv64gc-unknown-none-elf/release --output target/riscv64gc-unknown-none-elf/release/fs.img
Running 'dev' profile [unoptimized + debuginfo] target(s) in 0.04s
Running 'target/riscv64gc-unknown-linux-gnu/debug/ros-fs-fuse --app initproc sh hello bye fstest rrttest ls cat touchwith shutdown echo calc --target target/riscv64gc-unknown-none-elf/release --output target/riscv64gc-unknown-none-elf/release/fs.img'
[INFO] 正在打包应用程序 target/riscv64gc-unknown-none-elf/release/ -> target/riscv64gc-unknown-none-elf/release/fs.img
[INFO] 装载应用程序: target/riscv64gc-unknown-none-elf/release/initproc
[INFO] 装载应用程序: target/riscv64gc-unknown-none-elf/release/sh
[INFO] 装载应用程序: target/riscv64gc-unknown-none-elf/release/hello
[INFO] 装载应用程序: target/riscv64gc-unknown-none-elf/release/bye
[INFO] 装载应用程序: target/riscv64gc-unknown-none-elf/release/fstest
[INFO] 装载应用程序: target/riscv64gc-unknown-none-elf/release/rrttest
[INFO] 装载应用程序: target/riscv64gc-unknown-none-elf/release/ls
[INFO] 装载应用程序: target/riscv64gc-unknown-none-elf/release/cat
[INFO] 装载应用程序: target/riscv64gc-unknown-none-elf/release/touchwith
[INFO] 装载应用程序: target/riscv64gc-unknown-none-elf/release/shutdown
[INFO] 装载应用程序: target/riscv64gc-unknown-none-elf/release/echo
[INFO] 装载应用程序: target/riscv64gc-unknown-none-elf/release/calc
[INFO] 正在校验应用程序
[INFO] 校验应用程序: target/riscv64gc-unknown-none-elf/release/initproc 通过
[INFO] 校验应用程序: target/riscv64gc-unknown-none-elf/release/sh 通过
[INFO] 校验应用程序: target/riscv64gc-unknown-none-elf/release/hello 通过
[INFO] 校验应用程序: target/riscv64gc-unknown-none-elf/release/bye 通过
[INFO] 校验应用程序: target/riscv64gc-unknown-none-elf/release/fstest 通过
[INFO] 校验应用程序: target/riscv64gc-unknown-none-elf/release/rrttest 通过
[INFO] 校验应用程序: target/riscv64gc-unknown-none-elf/release/ls 通过
[INFO] 校验应用程序: target/riscv64gc-unknown-none-elf/release/cat 通过
[INFO] 校验应用程序: target/riscv64gc-unknown-none-elf/release/touchwith 通过
[INFO] 校验应用程序: target/riscv64gc-unknown-none-elf/release/shutdown 通过
[INFO] 校验应用程序: target/riscv64gc-unknown-none-elf/release/echo 通过
[INFO] 校验应用程序: target/riscv64gc-unknown-none-elf/release/calc 通过
cargo build --release
Finished 'release' profile [optimized] target(s) in 0.06s
rust-objcopy --strip-all target/riscv64gc-unknown-none-elf/release/ros666 -O binary target/riscv64gc-unknown-none-elf/release/ros666.bin
```

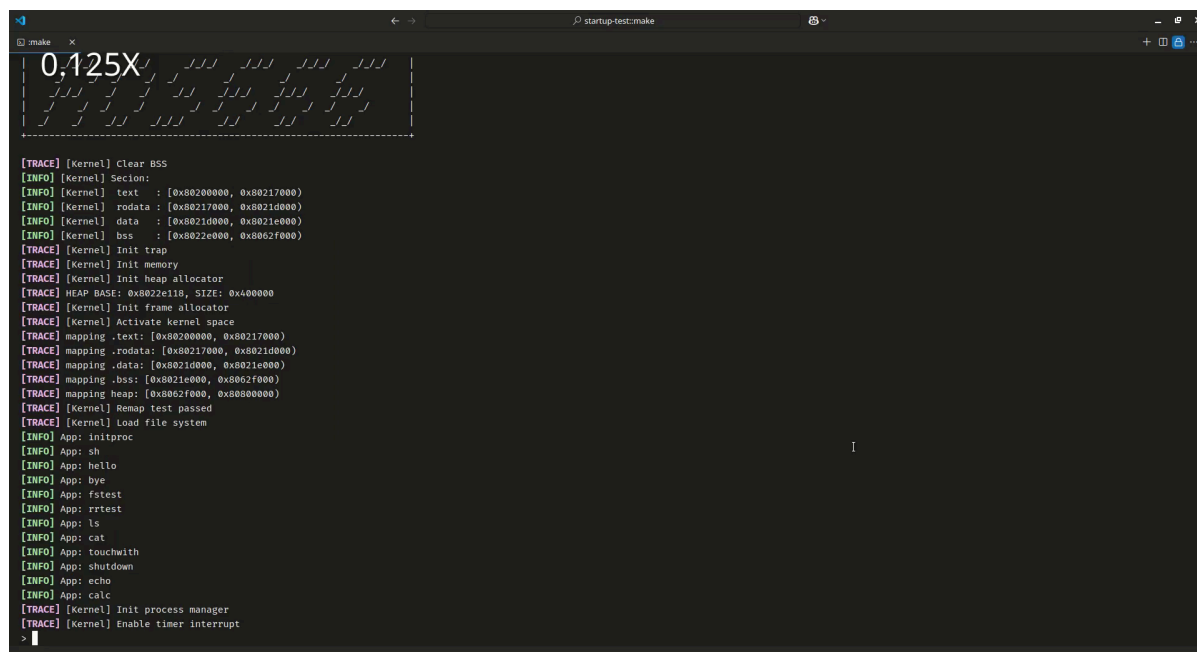
启动 QEMU, 加载 Rust SBI Boot Loader 和内核。

```
zmake x startup-test:make
[rustsbi] Building version 0.4.0-alpha.1, adapting to RISC-V SBI v2.0.0
0:125X
[rustsbi] Implementation : RustSBI-QEMU Version 0.2.0-alpha.3
[rustsbi] Platform Name : riscv-virtio,qemu
[rustsbi] Platform SMP : 1
[rustsbi] Platform Memory : 0x80000000..0x88000000
[rustsbi] Boot HART : 0
[rustsbi] Device Tree Region : 0x87e00000..0x87ef0000
[rustsbi] Firmware Address : 0x80000000
[rustsbi] Supervisor Address : 0x80200000
[rustsbi] pmp0: 0x80000000..0x80000000 (-wr)
[rustsbi] pmp2: 0x80000000..0x80200000 (-r-)
[rustsbi] pmp3: 0x80200000..0x88000000 (xwr)
[rustsbi] pmp4: 0x88000000..0x00000000 (-wr)

[TRACE] [Kernel] Clear BSS
[INFO] [Kernel] Section:
[INFO] [Kernel] text : [0x80200000, 0x80217000)
[INFO] [Kernel] rodata : [0x80217000, 0x8021d000)
[INFO] [Kernel] data : [0x8021d000, 0x8021e000)
[INFO] [Kernel] bss : [0x8022e000, 0x8022f000)
[TRACE] [Kernel] Init trap
[TRACE] [Kernel] Init memory
[TRACE] [Kernel] Init heap allocator
[TRACE] [Kernel] HEAP BASE: 0x8022e118, SIZE: 0x400000
[TRACE] [Kernel] Init frame allocator
[TRACE] [Kernel] Activate kernel space
```

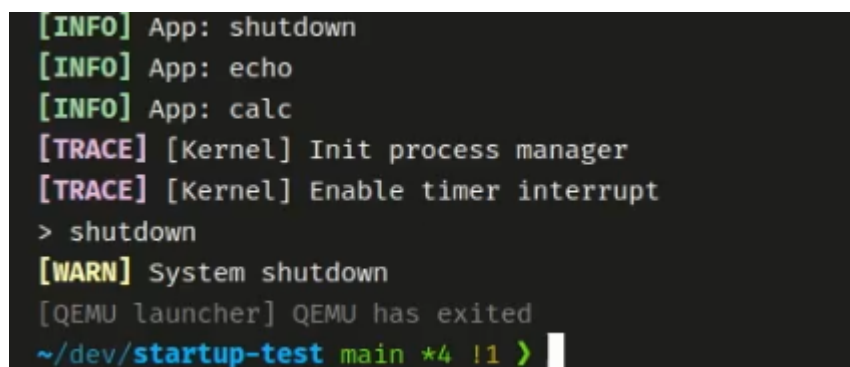
内核启动成功，显示内核信息。

在内核启动时，会初始化内核存储管理、陷入处理、时钟、进程管理等，从内核启动信息可以看到内核初始化的过程。



```
[TRACE] [Kernel] Clear BSS
[INFO] [Kernel] Section:
[INFO] [Kernel] text : [0x80200000, 0x80217000)
[INFO] [Kernel] rodata : [0x80217000, 0x8021d000)
[INFO] [Kernel] data : [0x8021d000, 0x8021e000)
[INFO] [Kernel] bss : [0x8022e000, 0x8062f000)
[TRACE] [Kernel] Init trap
[TRACE] [Kernel] Init memory
[TRACE] [Kernel] Init heap allocator
[TRACE] HEAP BASE: 0x8022e110, SIZE: 0x400000
[TRACE] [Kernel] Init frame allocator
[TRACE] [Kernel] Activate kernel space
[TRACE] mapping .text: [0x80200000, 0x80217000)
[TRACE] mapping .rodata: [0x80217000, 0x8021d000)
[TRACE] mapping .data: [0x8021d000, 0x8021e000)
[TRACE] mapping .bss: [0x8021e000, 0x8062f000)
[TRACE] mapping heap: [0x8062f000, 0x80800000)
[TRACE] [Kernel] Remap test passed
[TRACE] [Kernel] Load file system
[INFO] App: initproc
[INFO] App: sh
[INFO] App: hello
[INFO] App: bye
[INFO] App: fstest
[INFO] App: rrttest
[INFO] App: ls
[INFO] App: cat
[INFO] App: touchwith
[INFO] App: shutdown
[INFO] App: echo
[INFO] App: calc
[TRACE] [Kernel] Init process manager
[TRACE] [Kernel] Enable timer interrupt
> |
```

使用 shutdown 命令关闭内核。或者使用 Ctrl+A X 退出 QEMU。



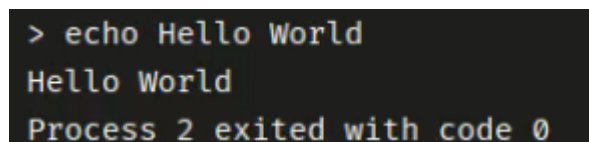
```
[INFO] App: shutdown
[INFO] App: echo
[INFO] App: calc
[TRACE] [Kernel] Init process manager
[TRACE] [Kernel] Enable timer interrupt
> shutdown
[WARN] System shutdown
[QEMU launcher] QEMU has exited
~/dev/startup-test main *4 !1 > |
```

## 2. 用户态程序和命令行参数解析

系统启动之后，会自动运行 initproc 和 sh 程序。

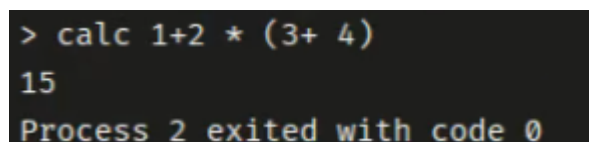
在 sh 程序中，可以执行用户程序，并传递参数。

运行 echo Hello World 命令。



```
> echo Hello World
Hello World
Process 2 exited with code 0
```

运行 calc 程序，计算  $1 + 2 * (3 + 4)$ 。



```
> calc 1+2 * (3+ 4)
15
Process 2 exited with code 0
```

### 3. 进程调度

内核支持基于时间片轮转的进程调度。

运行 `rrtest` 程序，测试进程调度。

其会并发执行 `hello` 和 `bye` 程序。

```
[Bye] Tick 0
[Hello] Tick 1
[Bye] Tick 1
[Bye] Tick 2
[Hello] Tick 2
[Bye] Tick 3
[Hello] Tick 3
[Hello] Tick 4
[Bye] Tick 4
[Bye] Tick 5
[Hello] Tick 5
```

```
[Bye] Tick 6
[Bye] Tick 7
[Hello] Tick 7
[Bye] Tick 8
[Hello] Tick 8
[Hello] Tick 9
[Hello] Time: 62.000255
[Bye] Tick 9
[Bye] Time: 62.001074
Process 2 exited with code 0
> Process 3 exited with code 0
```

可以看到，`hello` 和 `bye` 程序可以并发执行，并且存在不可预测的交替执行情况。

### 4. 文件系统

可以使用 `ls` 命令查看文件系统中的文件。

使用 `touchwith` 创建文件，并向文件中写入内容。

使用 `cat` 查看文件内容。

这里将 `Hello World!!!` 写入 `hello.txt` 文件。

然后使用 `cat` 查看文件内容。

```
[TRACE] [Kernel] Init process manager
[TRACE] [Kernel] Enable timer interrupt
> ls
initproc      sh      hello  bye      fstest  rrtest  ls      cat      touchwith s
hutdown echo    calc
Process 2 exited with code 0
> touchwith hello.txt Hello World!!!
Process 2 exited with code 0
> ls
initproc      sh      hello  bye      fstest  rrtest  ls      cat      touchwith s
hutdown echo    calc    hello.txt
Process 2 exited with code 0
> cat hello.txt
Hello World!!!
Process 2 exited with code 0
> █
```

可以看到，文件被成功创建，并且内容被成功写入。并且可以通过 cat 命令查看文件内容。