



哈爾濱工業大學(深圳)

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

MoOS 操作系统内核 设计文档

参赛队名: MoOS

队伍成员: 卢郡然、杨杰睿、唐楠青

指导老师: 夏文、仇洁婷

全国大学生计算机系统能力大赛

操作系统赛

内核实现赛道

2023 年 05 月

摘要

MoOS 是一个使用 Rust 编写，运行在 RISC-V 处理器架构上的内核。我们团队通过实现 MoOS，深入了解了操作系统架构并积累了 Rust 编程经验，现已成功通过初赛系统调用的所有测试用例，如图1所示，正在对于决赛阶段的重构进行详细规划和设计。

#	用户名	队伍	最后提交时间(ASC)	提交次数(ASC)	rank
1	202310464101015	你说对不队/ 河南科技大学	2023-04-18 14:30:26	22	102.0000
2	202318123101314	Titanix/ 哈尔滨工业大学（深圳）	2023-05-04 21:50:35	17	102.0000
3	202310698101003	PLNTRY/ 西安交通大学	2023-05-12 01:43:31	9	102.0000
4	202314430101195	编写吧INutOS/ 中国科学院大学	2023-05-19 15:39:44	17	102.0000
5	202318123101332	MoOS/ 哈尔滨工业大学（深圳）	2023-05-19 17:09:27	28	102.0000

图 1 初赛提交排行

在起初参考 rCore 实现 MoOS 的过程中，我们体会到了 rCore 架构的模块化所带来的便利和优点，因此决定更进一步继续参考另一基于 Rust 的微内核操作系统 Redox 的实现，以在当前 naive-os 分支的开发基础上构建出更加模块化、轻量级、安全性高的操作系统内核。以下部分介绍了我们当前在初赛阶段 naive-os 分支完成的初赛内核开发工作，以及当前调研所得后期望进行的优化方向：

表 1 MoOS 初赛阶段各组件的设计内容

组件	实现内容	优化方向
进程调度	基本的调度管理	减少不必要的上下文切换 使用 Redox 中 Scheme 的设计
内存管理	基本的内存管理	实现细粒度的内存分配 支持 Copy-on-write
文件系统	基本的虚拟文件系统和 fat32 支持	实现页缓存以优化文件读写 实现延迟写以减少磁盘 I/O
设备驱动	支持 qemu	更好地支持异步 I/O 支持比赛所用的开发板

目录

一、概述	1
1.1 前言	1
1.2 系统设计	3
二、模块设计	5
2.1 进程管理	5
2.1.1 概述	5
2.1.2 进程调度	6
2.1.3 进程控制块	6
2.1.4 陷阱与中断	7
2.2 内存管理	8
2.2.1 概述	8
2.2.2 地址空间	8
2.2.3 内存空间的管理	9
2.2.4 页表项与多级页表管理	10
2.3 文件系统	12
2.3.1 概述	12
2.3.2 虚拟文件系统	13
2.3.3 磁盘文件系统	17
2.3.4 磁盘驱动	17
2.3.5 文件系统相关系统调用	18
三、测试与交互	19
四、计划和展望	22
4.1 决赛阶段计划	22
4.1.1 系统架构调整	22
4.1.2 系统调用实现	22
4.1.3 预期特色功能	22
4.2 未来展望	23

一、概述

1.1 前言

MoOS 是一个在 RISC-V 处理器架构上运行的简单内核，它主要是用 Rust 语言实现的，也有部分使用 C++ 完成的用户态程序（如 shell 等）。初赛的主要工作是实现了一个简单的内核，通过了初赛评测所要求的所有测试，并且为我们小组成员熟悉操作系统的架构并学习 Rust 语言提供了一个很好的基础。前期工作主要参考 rCore 系列仓库的内容，我们的开发历史和代码在 GitLab 仓库的 naive-os 默认分支可见，后期工作中我们尚在对另一基于 Rust 的操作系统 Redox 进行调研，以期望参考其微内核实现的相关优化重构我们的 MoOS 内核。

我们团队没有使用无栈协程架构，尽管这是今年多支队伍应用的重点，并且无栈协程从设计之初确实虽然面向轻量级上下文切换，但在操作系统内核的开发上，暂时还没有看到足够广泛的在实际操作系统内核中应用的场景，无栈协程仍然属于相对新颖的尝试。我们队伍则更加青睐于基于 Rust 的微内核操作系统 Redox 的设计理念，通过 Rust 的所有权和生命周期机制减少系统中内存相关漏洞发生的可能性，并且通过微内核设计以足够的模块化以提高系统的安全性，再借助 Scheme 机制以减少不必要的上下文切换。事实上，去年的决赛一等奖队伍中 FTL OS 试图通过使用无栈协程来提高上下文切换的效率，在今年比赛中进行分享的 PLNTRY 队伍也使用无栈协程进行上下文切换的处理，但我们认为，更主要的提高内核执行效率的可能是 FTL OS 中提出的快速处理路径，这一点和 Redox 中提出的 Scheme 的思想类似。也因此我们决定保留有栈上下文切换的同时使用这些被实际应用的切换技巧来优化上下文切换的效率问题。

首先，我们需要理解上下文切换的开销是客观存在的，主要来源于保存和恢复 CPU 寄存器状态，以及可能的缓存无效化和内存映射更改，这是任何操作系统设计都很难避免的问题。通过无栈异步的上下文切换，虽说从理论上避免了内核栈的分配和管理开销，但从 FTL OS 的实践中我们可以看到，因为本质上保存的寄存器值没有减少（寄存器值就是程序当前状态，我们必须恢复执行的状态以继续执行切换前的进程），这样导致了同 FTL OS 文档中所述的无栈上下文切换中用户态-内核态的切换会比有栈更慢。FTL OS 采用快速处理路径来解决这个问题，避免了一些系统调用进行时对于内核态的切换，不过这种快速处理路径相当于跳过了上下文切换的过程，进程始终处于用户态运行，其实和无栈协程带来的优点（即上下文切换中不再使用内核栈，避免了内存分配和管理的开销）并没有结合，对编程者的架构控制能力反而也有更高的要求，提高了设计的复杂性。

因此我们还是保留了有栈上下文切换的设计，决定参考 Redox 中 Scheme 的实现方式。正如我们前文所述，这一设计和 FTL OS 中提到的快速处理路径有相似之处，都是为了避免了不必要的上下文切换以减少开销，尽管 Redox 原型实现中的 Scheme 没有专门针对系统调用的测试进行相关优化，但也说明实际优化进行的时候这种通用方案是得到了社区认可的。Scheme 虽然在设计上只用于处理各种类型的资源，包括设备、文件系统、网络接口等，在处理这些非核心的系统调用时可以避免进入内核态的切换开销，但这种相似的思想可以让我们基于此对 MoOS 的实现进行进一步优化。

不过受限于团队成员初次从头开始编写操作系统内核对细节的掌握情况，我们在初赛阶段 naive-os 分支的开发中没有实际应用到 Scheme 的设计，但我们有信心在接下来的两个月中完成我们的重构任务和开发，以构建出更加模块化、轻量级和安全的操作系统内核。

当前的 MoOS 操作系统内核由一系列模块化组件构成，这些组件在一起为上层应用程序提供了一个稳定可靠的运行环境。我们和通用的操作系统的主要功能一致，包括处理硬件中断，管理内存和任务调度，以及加载和运行用户空间的程序。我们的实现过程主要参考了 rCore（内存部分）以及 rcore-fs 衍生仓库（虚拟文件系统部分）。

目前支持的系统调用如下，也即所有初赛要求的系统调用，如下所示：

```
// file: kernel/src/syscall/mod.rs

const SYSCALL_GETCWD: usize = 17;
const SYSCALL_DUP: usize = 23;
const SYSCALL_DUP3: usize = 24;
const SYSCALL_MKDIRAT: usize = 34;
const SYSCALL_UNLINKAT: usize = 35;
const SYSCALL_UMOUNT: usize = 39;
const SYSCALL_MOUNT: usize = 40;
const SYSCALL_CHDIR: usize = 49;
const SYSCALL_OPENAT: usize = 56;
const SYSCALL_CLOSE: usize = 57;
const SYSCALL_PIPE2: usize = 59;
const SYSCALL_GETDENTS64: usize = 61;
const SYSCALL_READ: usize = 63;
const SYSCALL_WRITE: usize = 64;
const SYSCALL_FSTAT: usize = 80;
const SYSCALL_EXIT: usize = 93;
const SYSCALL_NANOSLEEP: usize = 101;
const SYSCALL_SCHED_YIELD: usize = 124;
const SYSCALL_TIMES: usize = 153;
```

```

const SYSCALL_UNAME: usize = 160;
const SYSCALL_GETTIMEOFDAY: usize = 169;
const SYSCALL_GETPID: usize = 172;
const SYSCALL_GETPPID: usize = 173;
const SYSCALL_BRK: usize = 214;
const SYSCALL_MUNMAP: usize = 215;
const SYSCALL_CLONE: usize = 220;
const SYSCALL_EXECVE: usize = 221;
const SYSCALL_MMAP: usize = 222;
const SYSCALL_WAITPID: usize = 260;

```

1.2 系统设计

MoOS 正常的使用通用操作系统的核心态和用户态的划分。核心态是运行在 Supervisor 模式下的内核，它负责操作系统的核心功能，如进程管理、内存管理和文件系统。用户态是运行在 User 模式下的用户程序，它们通过系统调用与内核交互，实现各种应用功能。项目中提供了一个基本的用户态程序 shell。它可以接收用户的命令，并执行相应的操作。项目还提供了 C 语言标准库，使得用户可以使用 C 语言开发自己的用户态程序。

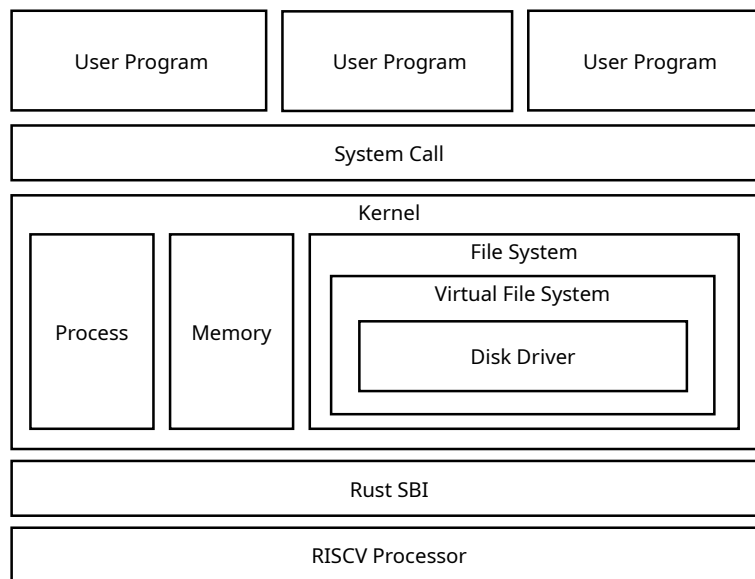


图 2 系统架构图

MoOS 的核心态内核与硬件之间通过 Rust SBI 进行通信。Rust SBI 是一个基于 Rust 语言开发的软件二进制接口（SBI），它实现了 RISC-V SBI 规范，并提供了一些扩展功能。Rust SBI 可以屏蔽硬件的差异，为上层的操作系统提供统一和安全的接口。MoOS

的核心态内核与用户态程序之间通过系统调用进行通信。MoOS 的系统调用遵循 Unix 规范，提供了一系列标准的接口，如文件操作、进程控制、信号处理等。

MoOS 的核心态内核主要由三个模块组成：进程管理、内存管理和文件系统。进程管理模块负责创建、销毁、调度、中断、唤醒等进程相关的功能；内存管理模块负责虚拟内存和物理内存的管理；文件系统包含一个虚拟文件系统，能够通过各级抽象统一调用接口，自管理缓存实现内存与外设的高速准确交互。具体而言，我们的操作系统内核的重要模块如下：

- `main.rs`: 这是操作系统的入口点。它负责初始化操作系统中的各个部分，然后开始用户程序的执行。
- `config.rs`: 这个模块定义了一些全局常量，如用户栈大小、内核栈大小、内核堆大小和内存末尾地址等，为内核的运行提供配置。
- `sbi.rs`: 这个模块是 Rust SBI 的封装，通过它，MoOS 的内核能与底层硬件进行交互。Rust SBI 实现了 RISC-V SBI 规范，为操作系统与硬件之间提供了统一和安全的接口。
- `timer.rs`: 这个模块提供了定时器相关的功能，如设置下一个定时器触发时间。
- `sync.rs`: 这个目录包含了同步原语，如原子操作和互斥体。
- `trap.rs`: 这个目录负责处理硬件陷阱，如时钟中断和系统调用。
- `entry.asm`: 这是系统的汇编语言入口点，它设置了初始的栈指针，然后跳转到 `rust_main` 函数。
- `lang_items.rs`: 这个模块提供了 Rust 的一些语言项，例如 `panic` 处理函数。
- `task/`: 这是进程管理模块，它包含了进程控制块 (PCB)，任务列表和调度算法，负责创建、销毁、调度、中断和唤醒等进程相关的功能。
- `mm/`: 这是内存管理模块，负责虚拟内存和物理内存的管理，包括内存分配、页表管理以及地址转换。
- `fs/`: 这是文件系统模块，它实现了一个虚拟文件系统，能够通过各级抽象统一调用接口，自管理缓存实现内存与外设的高速准确交互。

在用户态层次，我们提供了基本的用户态程序，即 `shell`。它可以接收用户的命令，并通过系统调用与内核交互，执行相应的操作。

- `syscall.rs`: 这个模块实现了 MoOS 的系统调用，遵循 Unix 规范，提供了一系列标准的接口，如文件操作、进程控制、信号处理等，它是内核与用户态程序之间的桥梁。

此外，为了实现这些功能，MoOS 还包含了一些其他的模块：

- `console.rs`: 这个模块负责控制台输入/输出，它提供了 `print` 和 `println` 等函数，可以在控制台上打印字符串。
- `boards/qemu.rs`: 这个目录包含了针对 QEMU 虚拟机的硬件配置和初始化代码。

以上为对我们的操作系统的基本介绍，接下来我们将更加详细的对 MoOS 的各个模块的设计进行解析，各部分的介绍内容将包括我们当前已有的 `naive-os` 实现、我们正在进行的改进以及在决赛阶段我们预计实现的内容。

二、 模块设计

2.1 进程管理

2.1.1 概述

进程管理是操作系统的一个重要功能，它负责对系统中的所有进程进行有效的管理和控制。为了实现这一功能，MoOS 将进程管理分为三个层次，分别是处理器管理、进程管理和进程控制。每个层次都对应了不同的抽象对象和功能模块。

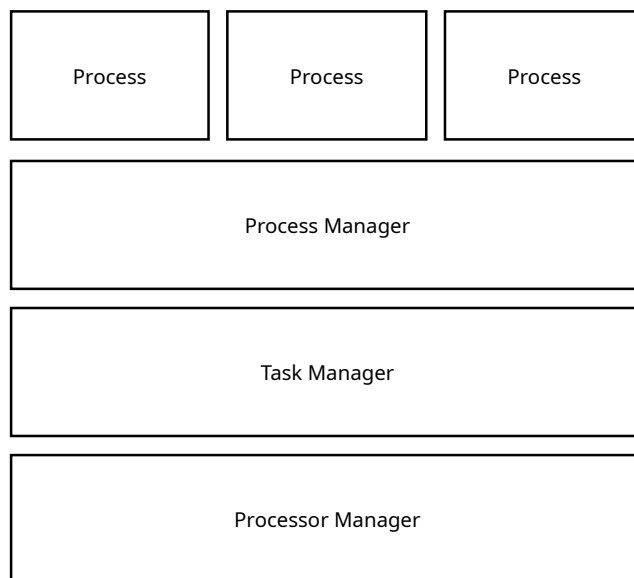


图 3 进程管理分层模块图

处理器管理是最高层次的进程管理，它主要负责处理机或 CPU 的分配和调度。在这一层次中，操作系统把处理机抽象为一个或多个逻辑处理器，每个逻辑处理器可以运行一个进程。操作系统维护了一个就绪队列，用来存放等待运行的进程。同时，操作系统还提供了一个调度器，用来根据一定的算法选择合适的进程送入处理机运行。

进程管理是中间层次的进程管理，它主要负责对系统中存在的所有进程进行统一的管理和维护。在这一层次中，操作系统把每个进程抽象为一个 PCB（进程控制块），PCB 是用来描述和记录进程各种状态信息的数据结构。操作系统维护了一个 PCB 集合或表格，用来存放所有进程的 PCB。在这一层次并未集成大量功能，其职责是保证多核心下对于 PCB 集合或表格的互斥访问。为了实现这一点，操作系统使用了锁或信号量等同步机制。

进程控制是最低层次的进程管理，它主要负责对单个进程进行具体的控制和操作。在这一层次中，操作系统把每个 PCB 作为一个控制器，用来控制该 PCB 所代表的进程。每个控制器里集成了该 PCB 所需的全部资源 and 功能模块，如内存分配、文件打开、信号处理等。由于这些资源和功能模块与内存系统、文件系统等其他子系统强耦合，因此这一层次也是操作系统与其他子系统交互最频繁的地方。

2.1.2 进程调度

目前，操作系统采用的轮转调度器是一种按照进程标识符的顺序就近原则进行调度的调度算法。被选中的进程将被加载并运行。这种调度方式简单有效，在有限的环境下能够发挥最大的作用。由于轮转调度器按照进程的 PID 顺序进行调度，每个进程都有公平执行的机会。

为了确保调度过程的正确性和一致性，操作系统在进行调度时采用了全局锁的机制，即只允许一个核心占有调度器。这样的设计保证了在多核系统中不会发生冲突或竞争条件。当一个核心正在执行调度操作时，其他核心必须等待，直到锁被释放。这种串行化的调度方式保证了调度过程的可靠性，防止了数据不一致或竞态条件的发生。针对多核系统为了提高并发性，后续我们还会对持有锁的区域进行细分。目前的锁机制使用的是 rust 核心库中提供的自旋锁，为了提高效率我们之后可能会需要实现睡眠锁机制提高 CPU 的利用效率

2.1.3 进程控制块

进程控制块用于管理和跟踪进程状态。其中的进程号用于唯一标识每个进程；运行状态字段跟踪进程的当前状态，如运行、就绪、阻塞等。这些信息使操作系统能够识别和管理不同的进程；进程上下文包含了进程的执行环境，包括程序计数器、寄存器状态和栈指针等。当进程被中断或切换时，操作系统可以使用进程控制块中的上下文信息保存和恢复进程的执行状态；陷阱帧的物理页号和页表字段存储了进程的内存映射信息。操作系统可以通过这些字段来管理进程的虚拟内存和物理内存之间的映射关系，以确保

进程能够正确访问其所需的内存空间；堆指针字段指示了进程的堆空间位置，使操作系统能够有效管理和分配进程的动态内存需求。父进程字段记录了进程的父子关系，方便进程间的通信和资源共享；进程控制块中的退出码字段记录了进程的终止状态，使父进程能够获取子进程的退出信息。此外，文件描述符列表存储了进程打开的文件和 I/O 设备的相关信息，用于进程间的文件共享和通信；运行时间字段记录了进程的累计运行时间，帮助操作系统进行调度决策和资源分配。工作目录字段存储了进程当前的工作目录路径，确保进程在文件系统中正确定位文件。

进程控制块提供了一个全面的视图，用于管理和控制进程的各个方面。它使我们的操作系统能够跟踪进程状态、分配和管理资源、进行进程间通信和同步，并支持调度和性能统计等。通过进程控制块，操作系统能够有效地管理多个进程的执行，确保系统的正常运行和资源利用。

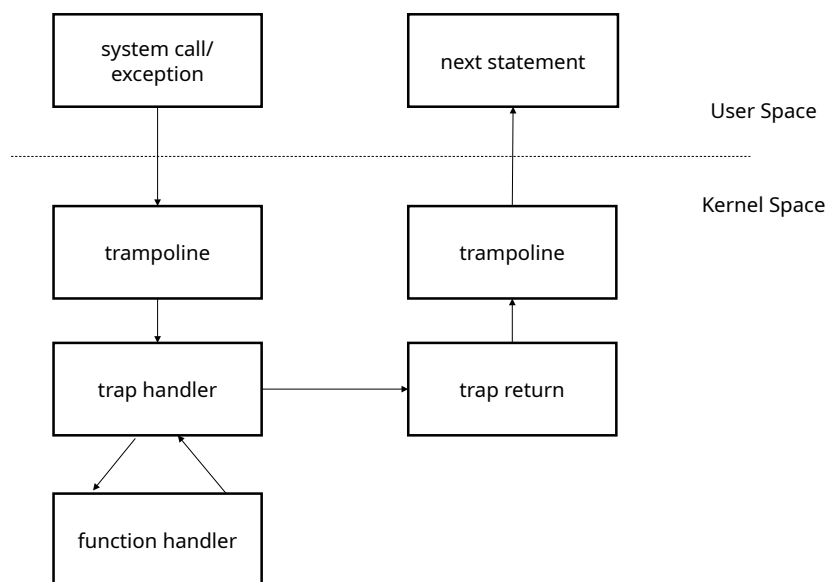


图 4 陷阱和中断

2.1.4 陷阱与中断

系统调用或者一些特性 (如懒分配) 都会触发中断，进入陷阱处理程序。在中断陷入时由跳板程序完成对上下文、地址空间的切换，跳板程序在内核与用户地址空间被映射到同一位置以保证在切换地址空间的时候运行流程不被打断，在陷阱处理程序中完成对系统调用和异常的识别和处理，以及完成一些功能 (mmap、COW)。在处理中断时会关中断防止二次中断。

2.2 内存管理

2.2.1 概述

物理内存是操作系统需要管理的一个重要资源, 而内存管理的目标就是让多个进程可动态地申请和释放内存, 同时要保证数据访问的安全性, 即访问同一块物理内存上的数据时不会发生冲突. 为了提高系统物理内存的动态使用效率, 同时保证应用间的安全性, 通过多级页表的管理物理内存, 把“有限”物理内存变成“无限”虚拟内存, 即虚拟地址空间. 以这种方式管理内存, 在每个进程的视角下它们都拥有所有的物理空间, 可随心所欲的使用. 操作系统实现了对物理内存的抽象, 使得应用设计者不必纠结于数据在内存中的位置. 一切都由操作系统通过页表管理将虚拟地址映射为物理地址.

2.2.2 地址空间

内存布局采用了分离的内核态和用户态页表, 允许内核和用户程序拥有各自独立的地址空间, 从而实现了内核和用户态的隔离. 内核态页表用于映射内核代码和数据, 而用户态页表则用于映射用户程序的代码和数据.

内核栈被放置在内核空间中, 定义为一个全局变量 `KERNEL_SPACE`, 该栈位置固定; 而每个进程都在内核空间中也有一个固定的位置, 该位置由进程的 `PID` 决定. 这种方式可以确保每个进程都有自己独立的内核栈, 使得进程在内核态执行时能够正确保存和恢复其上下文.

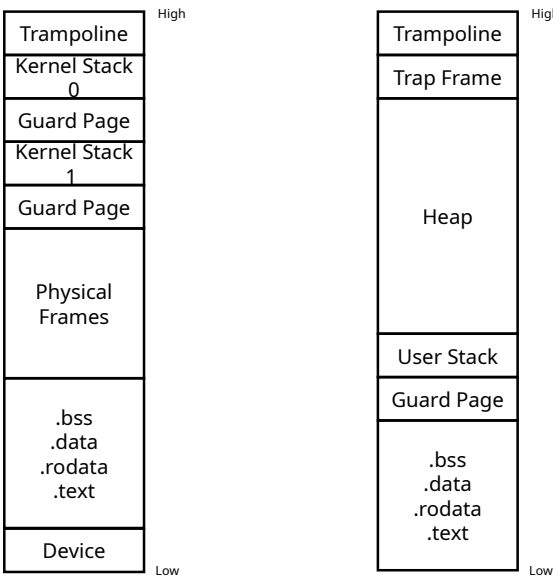


图 5 内核态与用户态地址空间

由于内核空间与用户空间分离, 为了实现内核态与用户态的数据交换, 采用了复制移动的方式。当数据需要从内核态传递到用户态或从用户态传递到内核态时, 数据会被复制到目标地址空间中, 确保了数据的隔离和安全性。这种方式可以有效地防止数据泄露或被未经授权的程序访问。

为了保证 Trap 的连贯性和处理异常的正确性, 在用户空间顶部映射了 Trampoline 和 Trap Frame。Trampoline 是一个小段代码, 用于实现从用户态到内核态的转换, 并将控制权传递给内核的异常处理程序。Trap Frame 则是保存了进入异常处理程序时的 CPU 上下文的数据结构, 以便异常处理程序可以正确地恢复执行状态。在内核以及用户空间中 Trampoline 被映射到同一块区域以使得切换页表时不打断执行流程。

```
// file: kernel/src/trap/context.rs

pub struct TrapFrame {
    /// general regs[0..31]
    pub x: [usize; 32],
    /// CSR sstatus
    pub sstatus: Sstatus,
    /// CSR sepc
    pub sepc: usize,
    /// Addr of Page Table
    pub kernel_satp: usize,
    /// kernel stack
    pub kernel_sp: usize,
    /// Addr of trap_handler function
    pub trap_handler: usize,
}
```

用户堆和页表项与内核堆一起由一个分配栈进行统一分配。这样可以有效地管理内存资源, 并提供对用户堆和页表项的动态分配和释放。这种设计简化了内存管理的复杂性, 提高了内存的利用效率。

2.2.3 内存空间的管理

内存空间的核心数据结构是 MemorySet, 其是页表, 内存空间和内存信息的同意抽象, 每个进程都含有一个该结构已表示该进程的内存信息。其含有两个成员: pagetable, 表示物理页表的位置; areas, 是一个数组, 其中保存了逻辑段的起始位置。MemorySet 还定义了一系列的成员函数用来对内存进行处理, 包括页表的映射和取消映射, 内存区域的创建和删除, 用指定的页表对虚拟地址进行地址翻译等。

```
// file: kernel/src/mm/memory_set.rs
pub struct MemorySet {
    pub page_table: PageTable,
    pub areas: Vec<MapArea>,
}
```

MemorySet 的设计利用了 C++ 中的 RAII 思想, 即”资源获取即初始化”, 将单个页帧这个物理资源与类绑定, 特别的是重定义了该类的析构函数的执行功能. 每当页帧分配器分配内存时, 即分配一个包含该页帧信息的类. 这样当 MemorySet 析构时, 其拥有的内存会自动返还给页帧分配器, 同时取消映射. 这样的实现不仅避免了手动释放同时有利于管理.

在物理页帧管理方面, MoOS 使用了一个页帧分配器 FrameAllocator, 以此来提供内核动态申请和释放内存的能力. 其有两个成员函数 alloc 和 delloc 分别对应页帧的分配和释放. 在分配时, 同时还要进行映射, 即将该页帧映射到当前进程的页表中. 在释放内存时不仅要将该页帧归还给页帧分配器, 同时还要取消该页帧在当前进程页表中的映射.

```
// file: kernel/src/mm/frame_allocator.rs
trait FrameAllocator {
    fn new() -> Self;
    fn alloc(&mut self) -> Option<PhysPageNum>;
    fn dealloc(&mut self, ppn: PhysPageNum);
}
```

而对进程而言, 我们对”页帧”这一概念进行了包装, 即 MemorySet 的 areas 中包含的不是页帧, 而是”逻辑段”. 每个逻辑段包含物理页帧的起始位置和结束位置. 当进程分配内存时, 其实际上调用 MemorySet 的 alloc 函数, 该函数根据给定的虚拟地址计算出需要的内存, 然后拓展相应的逻辑段的范围, 然后才调用 FrameAllocator 的 alloc 函数分配页帧.

2.2.4 页表项与多级页表管理

页表采用了常规的三级页表设计, 这意味着虚拟地址被分成三个层次的索引: 页目录、页表和页. 每个级别的索引都有自己的数据结构, 用于存储与之对应的物理页框的信息. 其中我们抽象出 PhysAddr, VirtAddr, PhysPageNum, VirtPageNum 四种类型用来表示物理地址, 虚拟地址, 物理页号和虚拟页号而不是直接使用 64 位硬件对应的数据类型. 这是因为这方便了我们定义各种成员函数来支持四种类型间的相互转换

```
// file: kernel/src/mm/address.rs

/// physical address
pub struct PhysAddr(pub usize);

/// virtual address
pub struct VirtAddr(pub usize);

/// physical page number
pub struct PhysPageNum(pub usize);

/// virtual page number
pub struct VirtPageNum(pub usize);
```

所有的页表项都采用相同的页大小即 4KB。这种设计可以简化页表的管理和处理逻辑,使得地址转换更加高效和一致。固定分页大小也有助于提供更好的内存管理和保护功能,以及更好地支持虚拟内存的功能,如页面置换和页面回写等。

多级页表实现的数据结构位类型 `PageTable`, 其包含两个成员: `root_ppn`, 表示页表根节点的物理页号; `frames`, 一个保存了页表所有节点所在的物理页帧。这里同样运用了 RAII 思想,将物理页帧与 `PageTable` 绑定。当 `PageTable` 声明周期结束时,这些物理页帧被自动释放。

```
// file: kernel/src/mm/page_table.rs

pub struct PageTable {
    root_ppn: PhysPageNum,
    frames: Vec<FrameTracker>,
}
```

`PageTable` 同时还实现了一系列成员函数以支持对页表的管理。`map` 和 `umap` 用来对给定的虚拟页号进行映射操作和解除映射。`translate` 用于“翻译”虚拟页号,即在多级页表上遍历以找到叶子节点。以上的这些操作方便了操作系统对内存进行更灵活的操作和管理。

```
// file: kernel/src/mm/page_table.rs

pub fn map(&mut self, vpn: VirtPageNum, ppn: PhysPageNum, flags: PTEFlags) {
    let pte = self.find_pte_create(vpn).unwrap();
    assert!(!pte.is_valid(), "vpn {:?} is mapped before mapping", vpn);
    *pte = PageTableEntry::new(ppn, flags | PTEFlags::V);
}
```

```
#[allow(unused)]
pub fn unmap(&mut self, vpn: VirtPageNum) {
    let pte = self.find_pte(vpn).unwrap();
    assert!(pte.is_valid(), "vpn {:?} is invalid before unmapping", vpn);
    *pte = PageTableEntry::empty();
}

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

通过页表，操作系统实现了对虚拟内存的精细控制和管理。它能够有效地将虚拟地址转换为物理地址，并提供了内存保护、页面置换和动态内存分配等功能。这些设计原则和数据结构的结合使得操作系统能够提供稳定、可靠且高效的内存管理功能。

2.3 文件系统

2.3.1 概述

文件系统是一个与硬件强耦合且非常复杂的部分。它主要负责两方面的工作：一是维护操作系统目录文件系统的结构和内容，二是实现操作系统与外设之间的数据交互。由于内存与外设之间存在着巨大的速度差异，为了保证高效地读写数据，这一部分需要设计大量分层缓存机制。同时，由于外设设备种类繁多且不同类型之间有很大差异，为了简化内核对外设访问的逻辑和代码量，这一部分还需要向内核提供一个统一且抽象化的接口。因此，我们将文件系统分为三个层次：接口层、文件节点层和数据缓存层。

接口层包含向上与内核通信的接口层，与负责与硬件和外设进行直接通信的硬件驱动层。它包括物理设备驱动和设备翻译两个子层。物理设备驱动是针对不同类型的物理设备（如 SD 卡、串口等）编写的程序，它可以直接控制物理设备进行读写操作。设备翻译是针对不同类型的磁盘布局（如 FAT32、EXT4 等）编写的程序，它可以将虚拟文件节点中表示数据位置和大小的信息转换为物理设备上具体扇区和字节位置。这两个子层共同构成了驱动层，它们可以屏蔽不同硬件和外设之间的差异，向上层提供一个统一且简单化的读写接口。

文件节点层，它主要负责管理文件节点。它将所有可能被内核访问到的字节流（如普通文件、管道、mmap 节点等）统一抽象为虚拟文件节点，并为每个虚拟文件节点提供了 `open_at`、`read_at`、`write_at`、`close` 等基本操作接口。同时，它还维护了一个完整的文件目录树结构，并为其提供了路径查找、创建、删除等操作接口。这些接口都是通过调用下层驱动层来实现具体功能的。虚拟文件系统层可以向内核提供一个统一且抽象

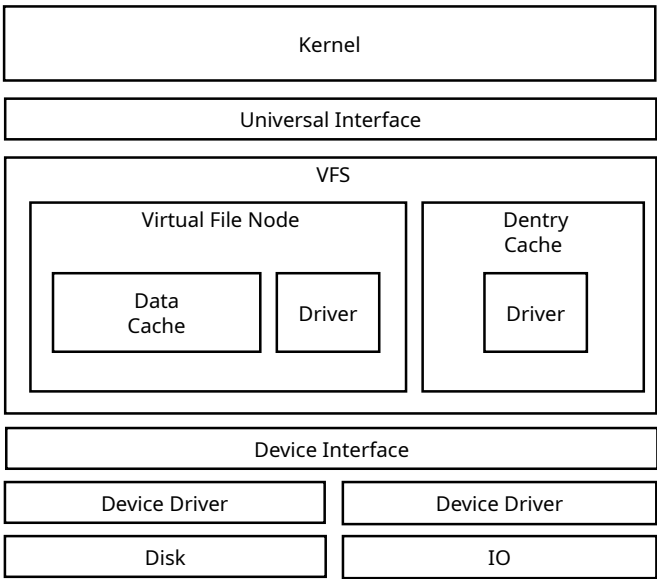


图 6 文件系统架构

化的访问接口，使得内核无需关心具体物理设备和磁盘布局。

每个虚拟文件节点都设置了一个独立的数据缓存区域，并根据不同类型节点采用不同策略来管理缓存内容。例如，在普通文件节点中，在内存有限的情况下采用一些置换算法来淘汰最近最少使用过的缓存块；在管道节点中，采用 FIFO 算法来按顺序传输缓存块；在 mmap 节点中，缓存写回时结合 mmap 标志位来阻止非法的写入等。数据缓存层可以利用大容量内存和局部性原理来提高数据读写速度，并减少对下层驱动层的调用次数。

在本次比赛中，由于我们有足够大的内存空间可供使用，并且所有读写需求都可以在程序开始和结束时集中处理（即挂载和卸载时），所以我们没有实现缓存换出机制，并且在挂载时就将所有需要访问到的数据全部加载进了缓存区域。这样做可以极大地提高程序运行效率，并简化缓存管理逻辑。这不完全符合实际操作系统的工作方式，但为了简化管理并且优化比赛中程序运行效率，我们暂时保留了这样的设计。

2.3.2 虚拟文件系统

虚拟文件系统是一个内核软件层，用来处理 Linux 标准文件系统相关的所有系统调用，这样一层在实际文件系统上的抽象层能为各种实际文件系统提供一个通用的接口。虚拟文件系统也是用户应用和文件系统之间操作的直接通信层，用于将用户期望进行文件系统的系统调用转换为对实际文件系统中的文件操作。

虚拟文件系统中实现了一些文件系统的通用接口，以使用通用的操作对不同的文件

类型进行操作，甚至于对于一些非文件类型的内核对象（如管道等）也能以一般文件的类型来调用对应的 `read_at` 和 `write_at` 方法进行读写。在虚拟文件系统中，我们还实现了对磁盘文件系统的缓存机制，包括维护的内存文件系统结构的 `dcache` 表（将在 `mount` 的时候完成初始化）。在这一部分中，我们引用了来自 `rcore-fs` 仓库中提供的虚拟文件系统层的设计，并根据我们的实际需要进行了修改，以适配我们系统调用编写过程中的需要。

具体而言，我们构造了多个结构，实现了 `Inode` 相关的 `trait`，以进行通用的 `Inode` 相关的操作（如文件读写、大小获取、类型判断等）。我们实现了如 `RegFileInode`, `TerminalInode`, `PipeInode` 等相关结构以复用通用的 `Inode` 相关方法以提高开发效率。我们还实现了 `Stat` 结构以获取文件元数据，该结构主要用于实现 `getdents` 系统调用。

另外，在文件系统中还维护了如下四个全局对象：

```
// file: kernel/src/task/mod.rs
lazy_static! {
    pub static ref global_dentry_cache: GlobalDentryCache = Default::default();
    pub static ref global_inode_table: GlobalInodeTable = Default::default();
    pub static ref global_open_file_table: GlobalOpenFileTable = Default::default();
    pub static ref global_buffer_list: GlobalBufferList = Default::default();
}
```

1. 全局目录项缓存（GlobalDentryCache）

```
pub struct GlobalDentryCache {
    pub table: Arc<Mutex<HashMap<String, Arc<Mutex<dyn Inode>>>>>>>,
}
```

全局目录项缓存用于在内存中维护目录项结构和 `inode` 的对应，目录项缓存目前使用的是互斥锁哈希表维护的结构，还是存在效率问题，将会使我们在后期优化的一个重点，因为将物理文件系统中必要的目录项缓存到内存中本身是为了性能提升，而加上互斥锁对频繁访问的目录项缓存会造成相对较大的性能影响，因此需要在后续进行更多的优化

2. 全局 Inode 表（GlobalInodeTable）

```
pub struct GlobalInodeTable {
    pub table: Arc<Mutex<Vec<Arc<Mutex<dyn Inode>>>>>>>,
}
```

全局 inode 表用于维护内存中的 Inode 结构，因为 FAT32 实际没有 Inode 结构，需要我们在读取之后构造正确的数据结构并保存在全局 Inode 表中。不过目前，该结构并未被很好的利用，虽然使用了 Inode 节点来管理文件元数据，但使用智能指针直接对单个节点进行了组织，此时组织起来的列表反而没有实际作用。但我们认为，以性能为目标的我们需要考虑在决赛阶段摆脱对智能指针的依赖，因为 inode 节点的维护完全可以作为一种非内存动态管理的方式进行，而是在文件系统挂载时确定可用的内存区域作为 inode 内容的存放位置，不需要对单一的节点进行单独的智能指针管理，这样使用更大粒度的锁（整个 inode 表的大锁以减少频繁获取锁造成的性能损失。这样也是考虑到了对于文件系统的读写操作更多集中在少量的节点进行，避免对过多的 inode 互斥锁的管理造成性能障碍。

3. 全局打开文件表（GlobalOpenFileTable）

```
pub struct GlobalOpenFileTable {
    table: Arc<Mutex<Vec<OpenFile>>>,
}
```

全局打开文件表用于在操作系统中记录由用户或者内核打开了的文件，并记录文件的实例指针指向实际文件的 inode。这里在目前而言是一个冗余设计，因我们在进程管理的部分实现的单个进程文件描述符表中维护的数据结构持有了一个指向文件 inode 的智能指针，所以不需要一直查到全局打开文件表才能够获取到具体 inode 的锁，但为了与通常的设计结构吻合，并且对于不恰当设计的修改留有余地，我们暂时保留了这个指针的使用，并且在未来对于全局 inode 表的修改了之后也能很方便的将指针替换为具体的 inode 标号以适配修改前后的代码。

4. 全局临时缓冲区列表（GlobalBufferList）

```
pub struct GlobalBufferList {
    list: Arc<Mutex<Vec<Arc<Mutex<Vec<u8>>>>>>>,
}
```

该部分用于维护内核中需要使用的临时内核对象，因为截止目前使用到的临时内核对象只有管道的所需要使用的缓冲区，所以该部分暂时使用 GlobalBufferList 命名。而在我们的设计当中，将来的使用的其他抽象的文件（如 socket 对象）也将由同一个管理全局内核对象的结构来统一管理。

虚拟文件系统中还实现了不同类的文件 INode 类型，用于对文件 INode trait 的实例化，在我们的实现中包括如下部分：RegFileINode, TerminalINode, PiPeINode 等。我们

目前没有另外对于目录类型另外实现一个 `INode`，而是在结构的内部进行区分，因为对于一般文件和目录文件的结构没有明显的区别，不同于另外的 `INode` 类型需要拥有完全不同的结构。

- `RegFileINode`

该结构是定义在内存中的一般文件 `INode`，用于记录一般文件的源信息和数据内容。这里因为是在内存中，我们为了简便编码，就直接将文件数据和文件 `INode` 绑定在一起，免去了查表的繁琐过程。一般文件的 `INode` 在物理文件系统加载完成之后就更新到内存中，免去了对更加底层细节的需要。

```
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct RegFileINode {
    pub readable: bool,
    pub writable: bool,
    pub dir: String,
    pub name: String,
    // Time related
    pub atime: Timespec,
    pub mtime: Timespec,
    pub ctime: Timespec,
    // Open mode
    pub flags: OpenFlags,
    // File data
    pub file: Vec<u8>,
}
```

- `TerminalINode`

该结构用于预先定义我们对于终端读写需要的 `INode` 类型，包括对于 `stdin`, `stdout`, `stderr` 三种终端文件类型读写时的读写对象。但实际读写时主要是把该结构类型作为占位和数据转移的中转，以便于上层调用的通用性。

```
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct TerminalINode {
    pub readable: bool,
    pub writable: bool,
    pub file: Vec<u8>,
}
```

- `PipeINode`

该结构用于临时对管道类型数据的转移，因为实际管道所在的内核对象由全局缓冲区列表管理，该部分仅为临时存放数据所用。

```
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct PipeINode {
    pub st: usize,
    pub buf: Vec<u8>,
}
```

对于不同类型的 INode 我们实现了通用的 `read_at`、`write_at` 方法，这样让我们在实现不同对于文件读写需要时候的通用性，简洁了代码的编写，提高了可维护性。

2.3.3 磁盘文件系统

在虚拟文件系统的设备翻译层负责处理不同的磁盘文件系统布局与虚拟节点之间的数据交互，处理、翻译来自虚拟文件层节点的读取、写入或者查找请求，将其翻译成具体磁盘文件系统的处理动作，交由下层驱动层驱动磁盘读写。

该层也具有独立的缓存以特定数据结构组织以加速磁盘增删改查的性能。如对于要求实现的 FAT32 磁盘文件系统为链式组成的对磁盘块进行缓存的结构，则该层维护了使用的磁盘块列表以用于写入时加速分配，避免读写实际磁盘的时候 IO 导致的效率下降

由于我们比赛阶段很大程度上并不关心磁盘的内容，该层还可以实现对磁盘的 raw 格式读写，用于内存不足时的换入换出，数据以原本形式连续存放于磁盘上，达到类似交换分区的效果，但该部分没有得到实际使用的机会，因为比赛中提供的内存足够的大，以至于我们没有使用交换分区的机会。

2.3.4 磁盘驱动

磁盘驱动的实现主要包括对底层磁盘操作的封装，例如读写磁盘块等基本操作，和对这些操作的调度和管理。在读写操作的实现中，我们引用了 `qemu` 提供的接口，使得我们能够通过简单的调用即可进行磁盘块的读写。这一部分我们直接参考了 `rCore-Tutorial` 的相关实现为了进一步优化效率，我们还采用了缓存机制，将磁磁盘块暂存在内存中，在上层的实际文件系统中形成块缓存，以加快后续的读写速度。

磁盘驱动的调度和管理则主要包括请求队列的维护，请求的调度策略，以及故障的处理等。请求队列的维护主要是将上层的读写请求加入到队列中，以便统一处理；请求的调度策略则是如何从队列中选择下一个需要处理的请求，这里我们采用了最先进入队

列的请求优先的策略；故障的处理则是当读写操作失败时如何处理，例如重新尝试读写或者报错返回。

尽管当前我们的磁盘驱动只支持 `qemu`，但我们在设计时尽可能考虑了通用性和扩展性，所以在之后需要支持决赛的开发板的时候，我们理论上只需实现新平台的读写接口，而无需对磁盘驱动的整体架构进行大的改动。这也是我们设计磁盘驱动时的一个重要原则，即尽可能的降低硬件平台的影响，使得我们的代码能够在不同的平台上重用。当然，这是一个理想的情况，具体的驱动编写中可能会遇到我们目前无法预料到的问题，也有待之后进一步解决。非常感谢您的反馈，我再次修改了以下的文本，尽可能地减少了重复的词汇和句式，提高了用词的多样性和自然度：

2.3.5 文件系统相关系统调用

初赛阶段，我们通过了所有的测试点，而文件系统所要求的系统调用占据其中一半以上。接下来，让我们深入探讨其中几个典型的系统调用的实现过程，如 `open`、`close`、`read`、`write` 等。在 MoOS 中，文件系统相关的系统调用均在 `kernel/src/syscall/fs.rs` 中得以实现。

- `open`: 该系统调用在 Unix 和 Linux 环境中负责打开文件，生成供后续操作的文件描述符。在 MoOS 系统中，它的实现过程如下：
 1. 解析文件路径，相对路径需要与当前工作目录相结合确定文件位置，绝对路径则直接定位。
 2. 利用文件路径寻找文件。这个过程从文件系统根目录出发，按照路径逐级查找。如遇文件不存在，可能会根据 `open` 系统调用的参数新建文件。
 3. 建立一个新的文件描述符并与找到的文件关联。在我们的设计中，文件描述符是一个数字，由操作系统管理，代表一个特定的打开文件。
 4. 将新建的文件描述符返回给调用者。
- `close`: 关闭一个打开的文件描述符，释放操作系统为该文件描述符分配的所有资源。其实现步骤如下：
 1. 检查文件描述符的有效性。如发现无效（例如，对应的文件未打开或者文件描述符超出了范围），则返回一个错误。
 2. 关闭文件描述符并释放其对应的资源。
- `read/write`: 用于从文件读取数据和向文件写入数据。这两个系统调用在实现上有很多共通之处。主要实现步骤如下：

1. 验证文件描述符的有效性，需要检查文件描述符是否有效。
2. 对于 read 系统调用，从文件的当前偏移处开始读取数据，并将数据拷贝到用户空间的缓冲区。对于 write 系统调用，将用户空间的缓冲区中的数据写入到文件的当前偏移处。
3. 更新文件的当前偏移。每次读或写操作之后，都会根据实际读取或写入的字节数更新文件的当前偏移。
4. 返回实际读取或写入的字节数给调用者。

以 `sys_write` 为示例，如下所示：

```
// file: kernel/src/syscall/fs.rs

/// write `buf` of length `len` to a file with `fd`
pub fn sys_write(fd: usize, buf: *const u8, len: usize) -> isize {
    let task = myproc();
    let fd_manager = &mut task.fd_manager;
    let fde = &fd_manager.fd_array[fd];
    if !fde.writable {
        return -1;
    }
    let buffers = translated_byte_buffer(myproc().memory_set.token(), buf, len);
    let mut sum = 0;
    for buffer in buffers {
        let mut open_file = fde.open_file.lock();
        let write_in = open_file
            .inode
            .lock()
            .write_at(open_file.offset, buffer)
            .unwrap();
        open_file.offset += write_in;
        sum += write_in;
    }
    return sum as isize;
}
```

三、测试与交互

我们在用户态实现了一个 shell 程序，可以用户简单的人机交互、调用用户态程序；一些简单的压力测试程序，以及测试用例自动化运行脚本程序，用于测例的自动运行。用户

可以使用 C 语言编写用户态程序参与测试。具体代码可以参考 `user_c/src/shell.cc`。

程序启动时，shell 会向用户打印一个欢迎消息，表明程序已经启动。

```
// file: user_c/src/shell.cc - main

// ...
char buf[256];

signed main() {
    printf("-----shell started!-----\n");
    int cc = 0;
    while(1) {
        // ...
    }
    return 0;
}
```

当用户开始与 shell 进行交互，程序负责捕捉和处理用户的每一个输入。这包括读取字符，处理退格和删除等特殊字符，直到接收到回车符：

```
// ...
while(1) {
    printf("shell %d,line %d>", getpid() ,++cc);
    char c = getchar();
    char *t = buf;
    while(c != 13){
        if(c == 0x08 || c == 0x7f) {
            if(t != buf) {
                printf("\x08 \x08");
                *t-- = 0;
            }
        } else {
            putchar(c);
            *t++ = c;
        }
        c = getchar();
    }
}
// ...
```

用户输入结束后，shell 需要对命令进行解析，理解用户的意图。例如，它能识

别”sexit”和”syield”这两个特殊命令，并执行特殊的判断：

```
// ...
while(1) {
    // ...
    while(c != 13) {
        // ...
    }
    *t = 0;
    if(strlen(buf) == 0) continue;
    if(!strcmp(buf, "sexit")) break;
    if(!strcmp(buf, "syield")) {
        sched_yield();
        continue;
    }
    // ...
}
```

当 shell 识别到其他系统命令时，它会创建新的子进程来执行这些命令，同时保持父进程的运行，等待子进程结束后打印出子进程的退出状态：

```
signed main() {
    // ...
    while(1) {
        // ...
        int pid = fork();
        if(pid == 0) {
            execve(buf, 0, 0);
        } else {
            int status;
            wait(&status);
            printf("[shell] exec exited with %d.\n", status >> 8);
        }
    }
    return 0;
}
```

在 shell 模式启动时输入 runtest 命令，或者在自动模式下直接运行会挂载磁盘并依次运行初赛测例文件，该部分是在 `user_c/src/runtests.cc` 中实现的。

四、计划和展望

4.1 决赛阶段计划

我们团队的决赛阶段计划主要分为三个部分：系统架构调整，系统调用实现，以及预期的特色功能。

4.1.1 系统架构调整

基于我们对 Redox 设计理念的理解以及在初赛阶段累积的经验，我们计划在决赛阶段进行系统架构的调整。虽然我们原始代码基于 rCore，而 rCore 已经采用了微内核的设计，但我们在开始编程之初为了编写代码的方便将很多部分糅合到了内核中，之后将采用微内核的设计理念，将操作系统内核中的必要组件精简至最小，例如内存管理和任务调度等，而其他非必要的组件，如文件系统、设备驱动等，将设计为用户空间的服务，并通过微内核进行管理。这样的设计能够为系统提供更强的隔离性和安全性，同时也使系统更加模块化和可维护。

4.1.2 系统调用实现

在系统调用的实现方面，我们将借鉴 Redox 的 Scheme 设计，尽量避免不必要的上下文切换，从而减少开销。我们将尝试将 Scheme 设计应用到系统调用的实现中，比如采用 FTL OS 中提出的快速处理路径处理简单的系统调用，复杂的系统调用则交给微内核处理。我们认为，这种设计既可以减少系统调用的开销，也可以提高系统的可扩展性。

4.1.3 预期特色功能

我们预期的特色功能包括以下几点：

- **网络协议栈**：我们预计将实现一个网络协议栈，在充分利用已有 crate 的情况下，构建一个能正常使用网络的操作系。当然，以微内核设计的 MoOS 的网络设备应当被移交给 Scheme 层来处理，而不是糅合在整个内核中。
- **全面的安全机制**：借助 Rust 的内存安全特性，我们计划在操作系统级别设计和实现一套全面的安全机制，包括但不限于访问控制、内存保护以及应用程序沙箱等，旨在从根本上提高系统的安全性。这一点还属于设想，具体的实现有待进一步考察。

4.2 未来展望

在本次比赛后,我们将继续研究和开发 MoOS。我们期望 MoOS 能够成为一个高效、安全、易用的操作系统内核,并能在更多的硬件平台上运行。同时,我们也希望 MoOS 能够对 Rust 在操作系统开发中的应用提供更多的经验和实践。我们也同样愿意在帮助对推动 Rust 在操作系统社区的使用,以及为促进 Rust 和 RISC-V 架构在教育和研究中的应用做出更多贡献。