哈爾濱工業大學(深圳)

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

Nighthawk

设计文档

参赛队名 Nighthawk

队伍成员 关雄正、王峻阳、冼志炜

指导老师 夏文、仇洁婷

2025 年 6 月 30 日

摘 要

Nighthawk OS 是使用 Rust 编写，支持 RISC-V 和 LoongArch 指令集架构，采用异步无栈协程架构的操作系统。

目前初赛阶段得分情况如下：

（还没上榜）

Nighthawk 各个模块完成情况如下表：

表 0-1: 模块完成情况

模块	完成情况
无栈协程	基于 rust 的 future 机制实现的无栈进程切换 ,能够在进程之间快速地切换调度，多核间以 M:N 调度算法对进程调度，充分发挥多核优势
进程管理	统一的进程线程抽象，方便内核管理的同时增强进程与线程对于 POSIX 的兼容性
内存管理	实现基本的内存管理功能。使用懒分配和 Copy-on-Write 优化策略。支持共享内存区域映射，便于高效的进程间通信和资源共享。
文件系统	基于 Linux 设计的虚拟文件系统。实现页缓存加速文件读写，实现 Dentry 缓存加速路径查找。支持 FAT32（基于 rust-fatfs）和 Ext4（基于 lwext4-rust）等主流文件系统。
进程通信	实现了符合 POSIX 标准的信号系统 ,支持用户自定义信号处理例程 ;实现了共享内存通信，适配内核其他异步功能
设备驱动	支持设备树解析，自动化设备发现与配置。实现 PLIC 支持，异步处理中断事件，提升外设响应速度
网络模块	模块化设计，支持灵活扩展 Udp，Tcp 等多种网络协议。异步事件处理框架与多核调度协同工作，确保网络通信在复杂应用场景下高效可靠

目 录

摘要 I

1 概述 1

1.1 宏内核设计 1

2 多架构设计 3

2.1 RISC-V 架构 3

2.2 LoongArch 架构 3

3 系统调用 4

3.1 系统调用接口 4

3.2 异步协程调度与任务执行 4

3.3 用户态与内核态切换 5

3.4 异步系统调用处理 5

3.5 系统调用分发与错误处理 6

3.6 系统调用列表 6

4 进程设计与管理 8

4.1 多核无栈协程设计 8

4.2 进程控制块 14

5 设备驱动 18

5.1 设备树解析 18

5.2 块设备驱动 19

5.3 字符设备驱动 20

5.4 网络设备驱动 20

5.5 设备管理架构 21

6 文件系统设计 22

6.1 虚拟文件系统 (VFS) 22

6.2 文件描述符与文件表 29

6.3 磁盘文件系统实现 29

6.4 特殊文件系统 30

6.5 页缓存与块缓存 31

7 进程间通信 34

7.1 信号机制 34

7.2 管道机制 37

7.3 共享内存机制 38

8 内存管理 38

8.1 物理内存管理 38

8.2 内核动态内存分配 39

8.3 虚拟地址空间 39

8.4 按需分页与缺页异常 40

8.5 写时复制 (Copy-on-Write) 40

9 网络模块 41

9.1 数据链路层设备 42

9.2 网络层 IP 协议 42

9.3 传输层 UDP 与 TCP 43

9.4 套接字 API 45

9.5 网络数据包处理 45

10 附录 47

11 总结与展望 47

11.1 主要工作成果 47

11.2 技术创新与特色 47

11.3 开发经验总结 48

11.4 未来发展方向 48

11.5 项目意义与影响 48

1 概述

现代操作系统内核的设计与实现，是计算机科学领域中极具挑战性的工作之一。Nighthawk 操作系统，作为一个基于 RISC-V 与 Loongarch 双架构的教学性内核，其目标不仅是探索 Rust 语言在操作系统开发中的应用潜力 ,更是为了深入实践现代操作系统设计的核心思想。

本文首先阐述了 Nighthawk 的整体架构设计 ,重点介绍了我们在进程与线程模型、内存管理、文件系统、进程间通信(IPC)以及系统调用等方面所做的设计决策和创新实践。Nighthawk 采用宏内核(Monolithic Kernel)架构 ,将所有核心系统服务都运行在内核态 ,以此来获得更高的执行效率。我们充分利用 Rust 语言的特性 ,如所有权(Ownership)、生命周期(Lifetime)和借用检查 (Borrow Checker) , 在没有垃圾回收器 (Garbage Collector) 的情况下，保证了内核的内存安全和并发安全。

在进程管理方面，我们实现了基于优先级抢占的调度算法，并设计了高效的上下文切换机制。内存管理模块则采用了分页机制，实现了虚拟内存、按需分页和写时复制 (Copy-on-Write) 等功能。文件系统方面，我们构建了一个可扩展的虚拟文件系统 (VFS) 层，并在此基础上实现了 ext4 与 FAT32 文件系统。

通过 Nighthawk 的开发 ,我们不仅加深了对操作系统原理的理解 ,也积累了利用 Rust 进行底层系统开发的宝贵经验。我们相信 ,Nighthawk 的设计与实现 ,可以为后续的操作系统研究和教学提供一个有价值的参考。

1.1 宏内核设计

操作系统内核架构主要分为宏内核 (Monolithic Kernel) 和微内核 (Microkernel) 两大类。

1. 宏内核将操作系统的核心功能，如进程管理、内存管理、文件系统和设备驱动等，全部实现在一个单一的、巨大的内核程序中。所有这些服务都运行在内核态，它们之间可以直接调用函数，通信效率高。Linux、UNIX 和 Windows 都是典型的宏内核设计。
    1. 优点：性能高，因为服务间通信是简单的函数调用，开销小。
    2. 缺点：内核代码庞大、复杂，一个模块的 bug 可能会导致整个系统崩溃。开发和维护难度大。
2. 微内核只在内核中保留最基本的功能，如 IPC (进程间通信)、基本的调度和内存管理。其他服务如图形服务器、文件系统、设备驱动等都作为独立的用户态进程运行。

1. 优点：模块化好，稳定性和安全性高。一个服务崩溃不会影响整个系统。可以独立地更新或替换服务模块。
2. 缺点：性能较低，因为服务间的通信需要通过 IPC，这涉及到用户态和内核态的频繁切换，开销较大。

Nighthawk 采用了宏内核架构。这主要是考虑到教学目的和性能要求。宏内核的设计相对直接，能让开发者更专注于核心功能的实现。同时，通过将所有服务放在内核态，可以避免微内核架构中频繁的上下文切换带来的性能开销，从而在系统调用等场景下获得更好的性能。

2 多架构设计

系统采用了多架构设计，目前通过硬件抽象层实现了对 RISC-V 和 LoongArch 两种主流架构的支持。

具体而言，Nighthawk 使用 rust 语言的 cfg 库和 polyhal-macro 宏等工具，屏蔽了不同架构的底层差异，核心设计理念是通过宏系统和条件编译实现对应架构代码的自动选择，从而将不同架构硬件的功能抽象出来，供内核统一使用。

2.1 RISC-V 架构

由加州大学伯克利分校的研究团队于 2010 年基于精简指令集计算（RISC）原则推出的一个简单、可扩展且灵活的指令集，适用于从微控制器到高性能计算在内的广泛应用领域。

SBI 作为 RISC-V 架构的标准化接口，为操作系统提供了访问机器模式特权功能的标准方法。Nighthawk OS 对 RISC-V 的抽象充分利用了 SBI（System Binary Interface）来实现对底层硬件的访问和控制。 在硬件线程管理、定时器管理、TLB 同步、控制台 I/O 等方面都使用了 SBI 接口，保证硬件操作的标准化。

2.2 LoongArch 架构

LoongArch 是一种全新的 RISC 指令集架构（ISA），由龙芯中科自主研发，具有一定程度上类似于 MIPS 和 RISC-V 的特性。

Nighthawk OS 对 LoongArch 的抽象参考了 PolyHAL 进行设计，由于 LoongArch 与 RISC-V 均为精简指令集架构，设计的时候力求与 RISC-V 行为相同，以保证系统运行的稳定性。

3 系统调用

3.1 系统调用接口

系统调用是操作系统提供给应用程序的接口 ,允许用户态程序请求内核提供的服务。本内核遵循类 UNIX 设计，提供了与 Linux 兼容的系统调用接口。

当应用程序执行系统调用时，它会通过特定的指令（如 `ecall`）陷入内核。在陷入内核之前，应用程序会将系统调用号和相关参数放入约定的寄存器中。例如，在 RISC-V 架构中，系统调用号通常放在 `a7` 寄存器，而参数则依次放在 `a0` 到 `a5` 寄存器中。

内核在接收到 `ecall` 指令后，会从用户态切换到内核态，并跳转到预设的陷阱处理程序。该程序会根据 `a7` 寄存器中的系统调用号 ,分发到对应的内核函数进行处理。处理完成后 ,内核会将返回值放入 `a0` 寄存器，并切换回用户态，返回到应用程序继续执行。

这种机制有效地隔离了用户程序和内核 ,保证了系统的稳定性和安全性。应用程序不能直接访问内核数据结构或执行特权指令 ,所有需要内核权限的操作都必须通过系统调用来完成。

3.2 异步协程调度与任务执行

本内核采用了基于异步协程的任务调度机制，每个用户任务都在一个异步执行单元中运行。`Task` 的生命周期由 `task_executor_unit` 函数管理，该函数是异步协程调度的核心组件。

异步任务执行单元（`task_executor_unit`）是内核调度的基本用户单位。当通过 `spawn_user_task` 生成一个 `UserFuture` 时，它会捕获一个新的 `task` 并成为存储在执行器任务队列中的 `UserFuture`。

任务执行的核心流程包括：

1. 初始化阶段：设置任务的唤醒器（Waker）和定时器中断
2. 用户态执行：通过 `trap_return` 返回用户态执行用户程序
3. 陷阱处理：通过 `trap_handler` 处理来自用户态的异常和中断
4. 异步系统调用：通过 `async_syscall` 处理系统调用请求
5. 调度决策：根据任务状态和调度策略决定是否让出 CPU
6. 信号处理：通过 `sig_check` 处理待处理的信号

当任务被设置为 `Zombie` 状态时，会跳出执行循环并调用 `task.exit()` 来回收部分资源，完整的资源回收会在父进程调用 `wait4` 系统调用时完成。

3.3 用户态与内核态切换

3.3.1 陷阱处理机制

内核通过 `trap_handler` 函数处理来自用户态的异常和中断。该函数首先读取 `stval` 和 `scause` 寄存器来确定异常类型和相关信息，然后根据异常类型分发到相应的处理函数。

在处理过程中 ,内核会同时更新全局定时器管理器并检查是否有过期的定时器 ,确保定时器检查能够稳定调用。

3.3.2 异常处理

用户态异常处理涵盖了多种情况：

- 系统调用异常：`UserEnvCall` 异常会标记任务需要处理系统调用，设置 `is_syscall` 标志
- 页错误异常：包括指令页错误、加载页错误和存储页错误。内核会根据错误类型确定访问权限要求 (执行、读取或写入) , 然后调用地址空间的页错误处理器。如果处理失败，会向任务发送 `SIGSEGV` 信号
- 非法指令异常：当执行非法指令时，内核会向任务发送 `SIGILL` 信号

3.3.3 中断处理

中断处理主要包括：

- 定时器中断：设置下一次定时器中断 , 并根据当前调度策略决定是否设置任务的 `yield` 标志
- 外部中断：处理来自外部设备的中断请求

3.3.4 内核态 → 用户态

`trap_return` 函数负责从内核态返回用户态。该函数会禁用中断，设置用户陷阱入口点，更新任务的定时器状态，然后通过汇编代码 `__return_to_user` 切换到用户态执行。

3.4 异步系统调用处理

本内核的一个重要特性是支持异步系统调用处理，这使得系统调用可以在不阻塞整个内核的情况下等待 I/O 操作完成。

`async_syscall` 函数负责处理异步系统调用：

- 检查任务是否有待处理的系统调用
- 提取系统调用号和参数

- 调用异步系统调用分发函数
- 处理返回值和中断标志

异步系统调用的优势：

- 非阻塞 **I/O**：I/O 密集型系统调用不会阻塞其他任务的执行
- 更好的并发性：多个任务可以同时等待不同的 I/O 操作
- 资源利用率：CPU 可以在等待 I/O 时执行其他任务

当系统调用返回 `EINTR` 错误时，会设置中断标志，以便后续的信号处理能够正确响应。

3.5 系统调用分发与错误处理

系统调用通过统一的 `syscall` 函数进行分发。该函数首先将系统调用号转换为枚举类型，然后通过模式匹配分发到对应的处理函数。每个系统调用都可能是异步的，支持在等待 I/O 时让出 CPU。

错误处理采用 Rust 的 `Result` 类型，成功时返回实际值，失败时返回负的错误码，与 Linux 系统调用的错误处理语义保持一致。

3.6 系统调用列表

本内核实现了大量的 Linux 兼容系统调用，支持超过 100 个系统调用，主要分为以下几类：

类别	系统调用	功能描述
进程管理	`clone`, `clone3`, `execve`, `exit`, `exit_group`, `wait4`, `getpid`, `gettid`, `getppid`, `sched_yield`, `setsid`, `setpgid`, `getpgid`	进程和线程的创建、执行、退出、等待及会话管理
内存管理	`mmap`, `munmap`, `brk`, `mprotect`, `madvise`, `mremap`, `mlock`, `munlock`, `msync`	虚拟内存映射、堆管理、内存保护和同步
文件系统	`openat`, `close`, `read`, `write`, `readv`, `writev`, `lseek`, `fstat`, `fstatat`, `statx`, `dup`, `dup3`, `pipe2`, `sendfile`	文件和目录的打开、读写、状态查询及描述符管理
目录操作	`mkdir`, `chdir`, `getcwd`, `unlinkat`, `getdents64`, `renameat2`, `linkat`, `symlinkat`, `readlinkat`	目录创建、切换、遍历及文件链接管理

文件系统挂载	mount, umount2, statfs, sync, fsync	文件系统挂载、卸载和同步操作
网络通信	socket, bind, listen, accept, connect, sendto, recvfrom, setsockopt, getsockopt, socketpair, shutdown, getpeername, getsockname	套接字创建、连接建立和数据传输
信号处理	rt_sigaction, rt_sigprocmask, rt_sigreturn, rt_sigtimedwait, kill, tgkill, tkill	信号处理函数设置、信号屏蔽和信号发送
时间管理	gettimeofday, clock_gettime, clock_getres, nanosleep, clock_nanosleep, times, setitimer, getitimer	系统时间获取、高精度休眠和定时器管理
同步原语	futex, shmget, shmat, shmdt, shmctl	用户态互斥锁和共享内存管理
I/O 控制	ioctl, fcntl, ppoll, pselect6, pread64, pwrite64	设备控制、文件控制和 I/O 多路复用
用户与权限	getuid, getgid, geteuid, getegid, setuid, setgid, umask	用户身份获取、设置和权限管理
系统信息	uname, sysinfo, syslog, prlimit64, getrusage	系统信息查询、日志记录和资源限制
调度控制	sched_getaffinity, sched_setaffinity, sched_getscheduler, sched_getparam, sched_setscheduler	CPU 亲和性和调度策略管理
其他	getrandom, membarrier, set_tid_address, set_robust_list, get_robust_list, faccessat, utimensat, ftruncate, getmempolicy	随机数生成、内存屏障、线程管理和文件访问控制

这种全面的系统调用支持确保了与现有 Linux 应用程序的高度兼容性，使得大多数用户态程序都能在本内核上正常运行。内核的异步设计进一步提升了系统的并发性能和响应能力。

4 进程设计与管理

4.1 多核无栈协程设计

4.1.1 传统有栈协程设计

传统的有栈协程设计中 ,每个用户线程都有自己的内核栈 ,线程的上下文切换需要保存和恢复栈指针。

以 rCore 为例，如下图，上下文切换需要首先保存当前线程的 CPU 寄存器快照到当前线程的栈上 ,并根据目标线程栈上保存的内容来恢复相应的寄存器 ,最后切换栈指针。这样的设计调度任务的开销大，且容易造成死锁问题。

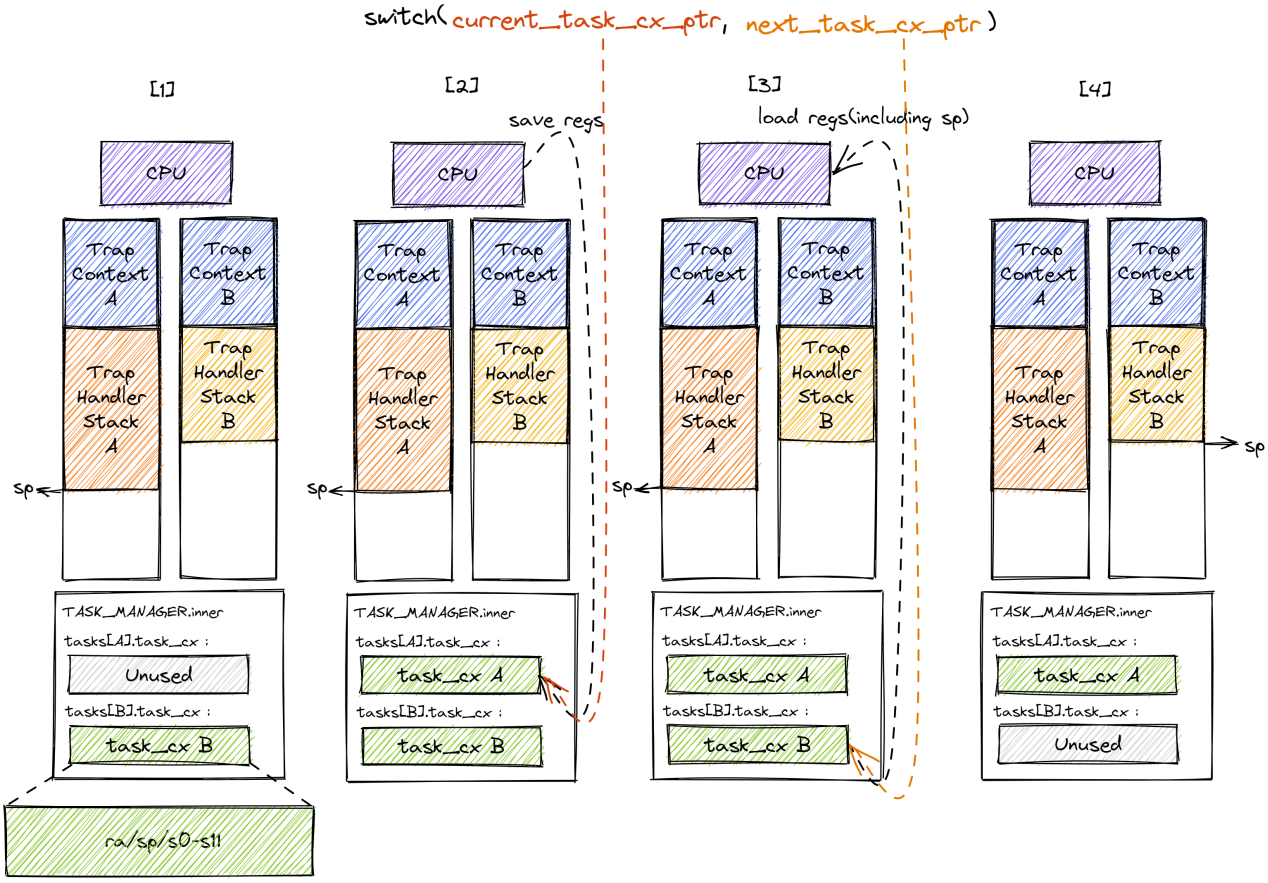

图 1: switch

4.1.2 无栈协程设计

4.1.2.1 语言支持

Rust 在 2018 年稳定版本中正式引入了 `Future` trait，并在 2019 年进一步完善了异步编程生态系统。这一特性在语言层面为异步编程提供了强有力的支持。

`Future` 是 Rust 中异步计算的核心抽象，它代表了一个可能尚未完成的异步计算。与其他编程语言中的异步模型不同，Rust 的 `Future` 采用了惰性求值的设计理念——只有在主动轮询（polling）时才会推进计算。这种设计通过实现 `Future` trait 的 `poll` 方法来定义异步计算的具体行为。

当执行器（executor）轮询 `Future` 时，`poll` 方法会返回一个 `Poll<T>` 枚举值来表示当前的计算状态：

- `Poll::Ready(T)` 表示异步计算已经完成，并包含计算结果
- `Poll::Pending` 表示异步计算仍在进行中，需要稍后再次轮询

为了简化异步编程的复杂性，Rust 提供了 `async` 和 `await` 关键字作为语法糖。`async` 关键字用于声明异步函数，这些函数的返回值会被自动包装成 `Future`；`await` 关键字则用于等待异步计算的完成，它会暂停当前函数的执行，将控制权交还给执行器，直到被等待的 `Future` 完成。

在编译器实现层面，`async` 函数会被转换为一个高效的状态机。编译器会分析函数中的 `await` 点，将整个函数拆分为多个状态，每个状态对应不同的执行阶段。这种状态机的实现方式是 Rust 异步模型的关键优势之一。

Rust 异步编程的最大特点是其零成本抽象（zero-cost abstraction）特性。这意味着：

- 无需额外的堆内存分配
- 避免了动态分发的开销
- 不需要传统的栈切换操作

状态机的设计使得异步任务间的切换仅仅是函数调用和返回的开销，而非昂贵的上下文切换。这种无栈协程（stackless coroutine）的实现方式使得 Rust 的异步模型特别适合构建高并发、低延迟的系统，尤其是在操作系统内核等需要处理大量异步 I/O 操作的场景中表现出色。

此外，Rust 活跃的社区和丰富的生态系统也提供了大量用于异步编程的库和工具。Nighthawk 在无标准库 no-std 的环境下开发，因此可以利用 async-task 库提供的关于 `Future` 的抽象。

4.1.2.2 代码实现

调度代码的主要实现有调度器 Executor , cpu 抽象 Hart 和 调度基本单位 TaskFuture 。

4.1.2.2.1 Executor

由于 Rust 没有内置异步调用所必需的运行时，而大部分库都不支持裸机环境，且异步组件往往强依赖于运行时，因此 Nighthawk 需要自行实现异步运行时及其组件，这包括 executor、异步锁和各种类型的 Future 。

Nighthawk 的异步执行器基于 async_task 库实现。async_task 库提供了构建执行器的基本抽象，包括 Runnable 和 Task 。一个 Runnable 对象持有一个 Future 句柄，在运行时，会对 Future 轮询一次。然后，Runnable 会消失，直到 Future 被唤醒才再次进入调度。一个 Task 对象用于获取 Future 的结果，通过 detach 方法将任务移入后台执行。async_task 提供了 spawn 方法，传入 Future 和调度器，用于创建 Runnable 和 Task 对象。然后，通过调度器将 Runnable 加入调度队列。显而易见的优点是，I/O 阻塞的任务不会进入调度序列，避免了忙等待。

Nighthawk 采用多核感知的任务调度策略。系统为每个 CPU 核心维护独立的任务队列，每个队列包含普通任务队列和优先级队列，以减少核心间的锁竞争。调度开始时，CPU 核心首先尝试从本地优先级队列中取出 Runnable 运行，若优先级队列为空，则从本地普通队列中获取任务。当本地队列为空时，实现了工作窃取机制，核心会尝试从其他活跃核心的队列中窃取任务，提高整体 CPU 利用率。

在任务分配策略上，新产生的任务会根据负载均衡算法分配到任务数最少的核心。若 Runnable 在运行时被唤醒（通常因为任务执行了 yield 让出时间片），则将其放入普通队列；若 Runnable 在其他任务运行时被唤醒（通常因为异步 I/O 完成的中断），则将其放入优先队列，确保 I/O 密集型任务能够及时响应。

```
pub struct TaskLine {
    tasks: SpinNoIrqLock<VecDeque<Runnable>>,
    pritasks: SpinNoIrqLock<VecDeque<Runnable>>,
}

pub static mut HART_TASKS_LINES: [TaskLine; MAX_HARTS] = [HART_TASKS_LINE; MAX_HARTS];

pub fn spawn<F>(future: F) -> (Runnable, Task<F::Output>)
    where
        F: Future + Send + 'static,
        F::Output: Send + 'static,
{
    let schedule = move |runnable: Runnable, info: ScheduleInfo| {
```

```
        push_in_available_line(runnable, info);
    };
    async_task::spawn(future, WithInfo(schedule))
}

pub fn push_in_available_line(runnable: Runnable, info: ScheduleInfo) {
    // 负载均衡：选择任务数最少的核心
    let available_line_id = find_least_loaded_hart();
    unsafe {
        if info.woken_while_running {
            HART_TASKS_LINES[available_line_id].push(runnable);
        } else {
            HART_TASKS_LINES[available_line_id].push_prio(runnable);
        }
    }
}
```

4.1.2.2.2 Hart

Nighthawk 支持多个 CPU 核心的运行。每个核心可以平等地从调度队列中取出 Runnable 运行。为了实现这个目的，每个 CPU 核心都有一个 thread local 的数据结构 Hart ，包含核心 ID、当前运行的任务和处理器特权状态。 tp 寄存器保存了当前核心的 ID，通过这个 ID 可以获取到当前核心的 Hart 。核心上下文保存了当前核心正在运行的任务引用和处理器特权状态（PPS），其中 PPS 包含了 CPU 的关键寄存器状态如 sstatus、sepc 和 satp 等。

```
pub static mut HARTS: [Hart; MAX_HARTS] = [HART_ONE; MAX_HARTS];

pub struct Hart {
    pub id: usize,
    task: Option<Arc<Task>>,
    pps: ProcessorPrivilegeState,
}

pub struct ProcessorPrivilegeState {
    // 包含 sstatus, sepc, satp 等关键寄存器状态
    // 以及中断计数器等处理器状态信息
}
```

内核线程和用户线程使用不同的 Future 封装结构：KernelFuture 和 UserFuture 。这两种结构都包含了处理器特权状态和具体的异步任务 Future ，其中 KernelFuture 不包含用户任务信息，而 UserFuture 持有任务的 Arc 引用。这些 Future 的 poll 过程会首先通过 switch_in 方法加载对应的处理器状态，然后 poll 保存的内部 Future 执行异步任务，最后通过 switch_out 方法保存当前状态。上下文切换遵循下面的步骤：

用户任务切换（**user_switch_in/out**）：（1）禁用中断并自动更新 PPS 状态；（2）交换 Hart 的 PPS 和任务的 PPS；（3）切换到任务的地址空间；（4）记录任务调度时间；（5）设置 Hart 的当前任务引用；

内核任务切换（**kernel_switch_in/out**）：（1）禁用中断并自动更新 PPS 状态；（2）交换 Hart 的 PPS 和任务的 PPS；（3）确保使用内核页表；

与有栈协程相比，上下文切换主要涉及处理器特权状态的交换（通过 `core::mem::swap` 实现），不需要保存和恢复大量的寄存器状态或切换栈指针，因此开销更小。同时，由于异步任务的执行是在异步执行器的控制下，任务的状态完全由编译器生成的状态机管理，不需要考虑栈溢出、互斥锁未释放等传统多线程问题。这样的设计使得 Nighthawk 能够充分利用多核特性，实现高效的内核异步模型。

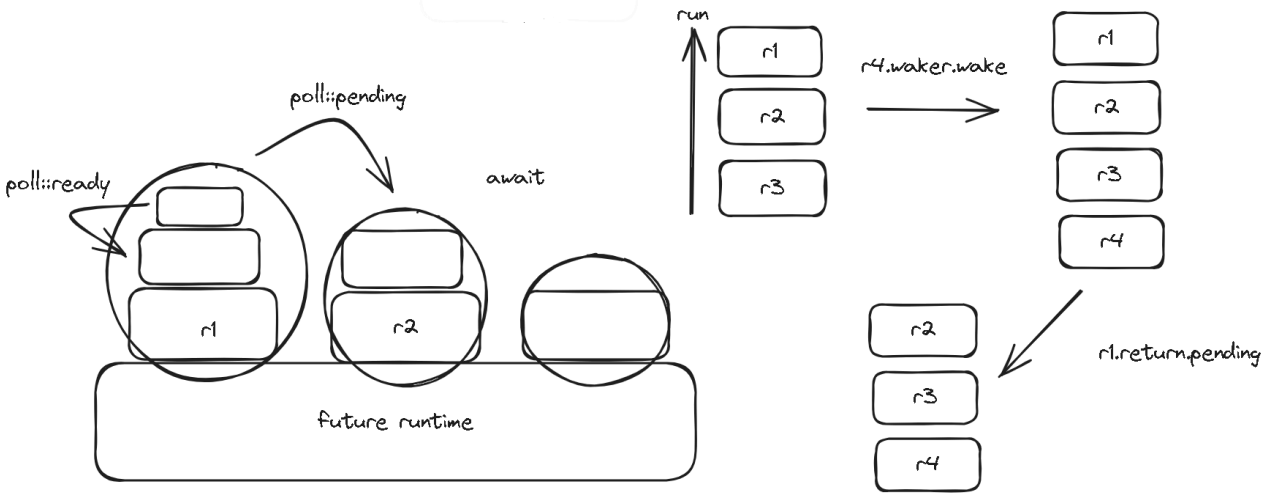

图 2: switch-no-stack

4.1.2.2.3 TaskFuture

TaskFuture 是 Nighthawk 中任务调度的基本单位，它将具体的异步任务与处理器状态管理结合起来，使得异步任务能够在多核环境中正确调度和执行。根据任务类型的不同，Nighthawk 提供了两种 TaskFuture 实现：`UserFuture` 和 `KernelFuture`。

UserFuture 用于封装用户态任务，它包含三个核心组件：任务控制块（`Arc<Task>`）、处理器特权状态（`ProcessorPrivilegeState`）和具体的异步 Future。任务控制块存储了线程/进程的所有状态信息，如 TID、内存空间、调度状态等；处理器特权状态保存了任务的执行环境，包括关键寄存器状态；异步 Future 则是任务的主体逻辑，通常是 `task_executor_unit` 函数。

KernelFuture 用于封装内核态任务，相比 UserFuture 更加简单，只包含处理器特权状态和异步 Future，不需要用户态的任务控制信息。它主要用于处理内核内部的异步事件，如定时器更新、内核服务等。

```
pub struct UserFuture<F: Future + Send + 'static> {
    task: Arc<Task>,
    pps: ProcessorPrivilegeState,
    future: F,
}

pub struct KernelFuture<F: Future + Send + 'static> {
    pps: ProcessorPrivilegeState,
    future: F,
}

impl<F: Future + Send + 'static> Future for UserFuture<F> {
    type Output = F::Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        let future = unsafe { Pin::get_unchecked_mut(self) };
        let hart = current_hart();

        // 尝试切换到用户态环境
        let r = block_on_with_result(async {
            hart.user_switch_in(&mut future.task, &mut future.pps).await
        });

        // 执行内部 Future 或处理切换失败
        let ret = if r.is_ok() {
            unsafe { Pin::new_unchecked(&mut future.future).poll(cx) }
        } else {
            cx.waker().wake_by_ref();
            Poll::Pending
        };

        // 切换回内核态环境
        hart.user_switch_out(&mut future.pps);
        ret
    }
}
```

TaskFuture 的关键创建函数包括 spawn_user_task 和 spawn_kernel_task。spawn_user_task 将任务控制块和 task_executor_unit 封装成 UserFuture，然后通过执行器调度；spawn_kernel_task 直接将异步函数封装成 KernelFuture 进行调度。

任务执行循环 :对于用户任务，其核心逻辑在 task_executor_unit 中实现。该函数首先获取任务的唤醒器(Waker)，然后进入用户态执行循环 :trap_return → 用户态执行 → trap_handler → 系统调用处理 → 信号检查。这个循环会持续进行，直到任务状态变为 Zombie。

状态管理：TaskFuture 通过处理器特权状态的交换实现了无栈上下文切换。当任务被调度时，其 PPS 会与 Hart 的 PPS 交换，从而恢复任务的执行环境；当任务让出 CPU 时，再次交换 PPS 以保存当前状态。这种设计避免了传统栈切换的开销，同时确保了任务状态的正确保存和恢复。

错误处理：对于 UserFuture，如果在 user_switch_in 过程中遇到地址空间锁定等问题，会通过唤醒器重新调度该任务，避免了阻塞整个调度器。KernelFuture 由于不涉及用户态切换，其实现更为简洁直接。

这种 TaskFuture 的设计使得 Nighthawk 能够统一处理用户态和内核态的异步任务，同时保持了高效的调度性能和良好的隔离性。

4.2 进程控制块

进程与线程的统一管理是现代操作系统设计的重要特征之一。在传统设计中，进程和线程往往使用不同的数据结构和管理机制，这不仅增加了系统的复杂性，也带来了维护和扩展的困难。我们采用统一的 `Task` 结构体来表示进程和线程，这种设计具有以下显著优势。

首先，统一的数据结构避免了重复的代码实现，减少了系统的复杂性。调度器、信号处理、内存管理等子系统只需要处理一种任务类型，大大简化了系统设计。

其次，进程和线程共享相同的基础操作（如调度、信号处理、资源管理），统一设计使得这些操作的实现可以被完全复用，减少了代码冗余。

再者，当需要添加新的任务特性或优化现有功能时，只需要修改一个数据结构和相关的操作函数，而不是分别维护进程和线程的两套代码。

并且，进程和线程可以使用相同的调度算法和优先级管理机制，使得系统调度更加公平和高效。

最后，系统调用和内核 API 可以统一处理进程和线程，减少了接口的复杂性，也降低了用户空间程序的使用难度。

这种设计理念与 Linux 内核的做法一致，体现了"一切皆任务"的设计哲学，为构建高效、可维护的操作系统奠定了坚实基础。

4.2.1 Task 结构体设计

基于上述分析，我们对进程和线程统一使用 `Task` 结构体表示，其具体设计如下：

```
pub struct Task {
    // 不可变字段 - 在任务创建时确定，生命周期内不变
    tid: TidHandle,
    process: Option<Weak<Task>>,
    is_process: bool,

    // 线程组管理
    threadgroup: ShareMutex<ThreadGroup>,

    // 任务执行上下文
    trap_context: SyncUnsafeCell<TrapContext>,
    timer: SyncUnsafeCell<TaskTimeStat>,
    waker: SyncUnsafeCell<Option<Waker>>,

    // 任务状态控制
    state: SpinNoIrqLock<TaskState>,

    // 内存管理
    addr_space: SyncUnsafeCell<Arc<AddrSpace>>,
    shm_maps: ShareMutex<BTreeMap<VirtAddr, usize>>,

    // 进程关系管理
    parent: ShareMutex<Option<Weak<Task>>>,
    children: ShareMutex<BTreeMap<Tid, Arc<Task>>>,
    pgid: ShareMutex<PGid>,
    exit_code: SpinNoIrqLock<i32>,

    // 信号处理
    sig_mask: SyncUnsafeCell<SigSet>,
    sig_handlers: ShareMutex<SigHandlers>,
    sig_manager: SyncUnsafeCell<SigManager>,
    sig_stack: SyncUnsafeCell<Option<SignalStack>>,
    sig_cx_ptr: AtomicUsize,

    // 文件系统接口
    fd_table: ShareMutex<FdTable>,
    cwd: ShareMutex<Arc<dyn Dentry>>,
    elf: SyncUnsafeCell<Arc<dyn File>>,

    // 线程特定信息
    tid_address: SyncUnsafeCell<TidAddress>,
    cpus_on: SyncUnsafeCell<CpuMask>,

    // 调度和同步控制
    is_syscall: AtomicBool,
    is_yield: AtomicBool,

    // 定时器和调试信息
    itimers: ShareMutex<[ITimer; 3]>,
    name: SyncUnsafeCell<String>,
}
```

下面介绍一下 `Task` 结构体各个字段的含义。

首先 Task 结构体被大致划分为不可变和可变两部分：

- 不可变部分：这些字段在任务创建时确定，生命周期内不会改变，因此可以保证数据的一致性。多线程环境下访问这些字段时不需要加锁，可以提高访问效率和安全性。

字段名	含义
tid	任务的唯一标识符（任务 ID）
process	对所属进程任务的弱引用。如果该任务本身是进程，则为 None
is_process	标识任务是否是进程（而非线程）

- 可变部分：可变字段涉及任务的状态和资源管理，这些字段在任务的生命周期内可能会改变，需要使用不同的同步机制来保证线程安全。

对于 Task 的可变部分，我们进行了精心设计。调研其他队伍时我们发现，很多队都用了 Inner 结构体存储可变字段，并在外面加上自旋锁的方式。此种方式虽然能在并发时保证数据的安全性，但是用一把大锁来锁定可变字段并不高效。因此，我们根据可变字段的访问模式和共享特性，分别使用不同的同步原语进行包裹：

- **SyncUnsafeCell<T>**：用于任务独有的字段，这些字段只会被拥有该任务的线程访问。SyncUnsafeCell 提供了内部可变性，同时避免了不必要的锁开销。使用时需要保证访问的独占性。
- **ShareMutex<T>**（即 **Arc<SpinNoIrqLock<T>>**）：用于同一进程内多个线程需要共享访问的字段。SpinNoIrqLock 是一种自旋锁，适用于短时间的锁定操作，不会引发中断，适用于内核环境。
- **SpinNoIrqLock<T>**：用于可能被多个任务并发访问的关键状态字段，如任务状态和退出码。
- **AtomicBool** / **AtomicUsize**：用于简单的原子操作字段，提供无锁的并发访问。

这种细粒度的锁设计允许在多个任务之间安全地共享和修改数据，显著提高了系统的并发性和性能，同时保证了线程安全性。通过这些机制，操作系统能够更加高效地管理任务和资源。

4.2.2 任务的状态

在调研其他操作系统时，我们发现部分往届作品只是简单区分了 Running 和 Zombie 状态，并不支持当 SIGSTOP 信号到来时进程暂停执行，此外，任务在阻塞的过程中能否被信

号打断也支持得并不是很好。Linux 关于信号的手册中提到，部分系统调用在阻塞等待的过程中可以被信号打断停止执行，返回 EINTR 错误，如果打断系统调用的信号的 flag 标志中含有 SA_RESTART 可以重启系统调用。基于以上考虑，我们将 Task 的状态分为以下六种：

状态	含义
Running	任务正在 task_executor_unit 循环中运行，占用 CPU 执行其代码
Zombie	任务已终止，会在主循环中检查此状态并进入 exit 处理
WaitForRecycle	任务已退出，等待其父进程回收
Sleeping	任务处于长时间等待状态，类似于等待但等待时间更长
Interruptable	任务处于可中断的等待状态，等待长时间事件（如 I/O）。此状态下，任务可以被信号中断并唤醒
UnInterruptable	任务处于不可中断的等待状态。此状态下，任务不会被信号中断，以确保某些关键操作的完整性和原子性

任务间的状态转换情况如下：

1. **Running ↔ Interruptable**：当任务需要等待某个事件时，从 Running 转换到 Interruptable 状态。例如，在实现 sys_wait4 系统调用时，Task 调用 suspend_now() 将自己从任务调度队列中移除，进入 Interruptable 状态，可以被信号中断。如果等待的子进程退出，返回子进程的 pid；如果被信号中断，返回 EINTR 错误。
2. **Running ↔ UnInterruptable**：当任务需要等待某个关键事件且不希望被信号中断时，从 Running 转换到 UnInterruptable 状态。当等待的事件发生时，恢复到 Running 状态。
3. **Running ↔ Sleeping**：当任务需要进入长时间等待状态时，从 Running 转换到 Sleeping 状态。与 Interruptable 类似，但通常用于更长时间的等待场景。
4. **Running → Zombie**：当任务执行结束并退出时，从 Running 转换到 Zombie 状态，会进入任务的 exit 处理函数。
5. **Running → WaitForRecycle**：当任务退出时，从 Running 转换到 WaitForRecycle 状态，等待父进程通过 wait4 等系统调用回收。

这种状态设计充分考虑了信号处理的复杂性和任务回收的不同场景，使得系统能够正确处理各种任务状态转换，提供了与 Linux 兼容的信号处理机制。

5 设备驱动

外设管理模块在操作系统中具有至关重要的作用，其主要目的是管理和协调系统中的各种外设，确保它们能够高效、稳定地运行。外设管理模块包括设备的发现、初始化、驱动程序加载以及中断处理等功能，这些功能的实现直接影响到整个系统的性能和稳定性。

本内核目前支持块设备（Block Device）、网络设备（Network Device）和字符设备（Char Device），采用基于设备树的设备发现机制，支持 RISC-V 和 LoongArch 两种架构。

5.1 设备树解析

操作系统内核获取设备树地址的流程从固件初始化开始。当系统启动时，OpenSBI 固件（RISC-V）或其他固件首先运行，完成基础的硬件初始化。对于 riscv 架构，固件初始化完成后，将控制权传递给内核的入口点，并传递必要的参数，包括硬件线程 ID 和设备树地址。而对于 loongarch 架构，它的设备树地址则被设为一个固定的值 0x100000.

内核通过 `probe_tree` 函数开始设备树解析过程，该函数会根据目标架构选择不同的解析策略。RISC-V 架构使用 MMIO 方式解析设备，LoongArch 架构使用 PCI 方式解析设备。内核使用 `flat_device_tree` crate 解析设备树，支持从设备树中提取启动参数等重要信息。

```
pub fn probe_tree() {
    let device_tree = unsafe {
        Fdt::from_ptr((DTB_ADDR + KERNEL_MAP_OFFSET) as *const u8)
            .expect("Parse DTB failed")
    };

    #[cfg(target_arch = "riscv64")]
    probe_mmio(&device_tree);

    #[cfg(target_arch = "loongarch64")]
    probe_pci(&device_tree);
}
```

5.1.1 RISC-V MMIO 设备解析

在 RISC-V 架构下，内核通过 `probe_mmio` 函数解析基于内存映射 I/O 的设备。块设备发现过程中，内核遍历设备树中所有 `/soc/virtio_mmio` 节点，读取寄存器基地址和大小信息，通过 `ioremap` 建立物理地址到虚拟地址的映射，使用 `probe_mmio_device` 检测具体的设备类型。

字符设备发现采用多级回退机制。内核首先从 `/chosen` 节点获取标准输出设备路径，支持多种兼容的串口控制器（NS16550A、DW APB UART、SiFive UART），解析寄存器参数如 `reg-io-width` 和 `reg-shift`，最终创建内存映射串口设备。

```
pub fn probe_char_device(root: &Fdt) -> Option<MmioSerialPort> {
    let chosen = root.chosen().unwrap();
    let mut stdout = chosen.stdout().map(|n| n.node());

    if stdout.is_none() {
        // 尝试解析 stdout-path 属性
        let stdout_path = /* 从 chosen 节点解析路径 */;
        stdout = root.find_node(stdout_path);
    }

    if stdout.is_none() {
        // 回退到搜索兼容设备
        stdout = root.find_compatible(&[
            "ns16550a", "snps,dw-apb-uart", "sifive,uart0"
        ])
    }

    Some(probe_serial_console(&stdout.expect("Unable to get stdout device")))
}
```

网络设备发现过程中，内核检测 VirtIO 网络设备，如果未找到硬件网络设备，则初始化回环设备作为备选方案。

5.1.2 LoongArch PCI 设备解析

在 LoongArch 架构下，内核通过 PCI 总线发现和管理设备。PCI 根复合体发现过程中，内核在设备树中查找 PCI 主机控制器节点，支持 CAM 和 ECAM 两种配置访问方式，建立 PCI 配置空间的内存映射。

PCI 设备枚举过程遍历所有 PCI 总线、设备和功能号，检测 VirtIO PCI 设备并创建相应的传输层对象，为设备分配 BAR 空间，启用设备的内存空间和总线主控功能。内核实现了 `PciMemory32Allocator` 来管理 PCI 设备的 32 位内存地址分配，基于设备树的 ranges 属性确定可用的内存区域，确保 BAR 分配满足大小和对齐要求。

5.2 块设备驱动

块设备是计算机中一类能够以固定大小的数据块进行随机访问的存储设备。本内核主要支持基于 VirtIO 标准的块设备，支持 MMIO 和 PCI 两种传输方式，使用 `VirtBlkDevice` 封装底层的 VirtIO 传输协议，提供异步 I/O 接口，支持非阻塞的读写操作。

```
pub fn probe_virtio_blk(root: &Fdt) -> Option<Arc<VirtBlkDevice>> {
    for node in device_tree.find_all_nodes("/soc/virtio_mmio") {
        for reg in node.reg() {
            let mmio_base_paddr = PhysAddr::new(reg.starting_address as usize);
            let mmio_size = reg.size?;

            ioremap(mmio_base_paddr.to_usize(), mmio_size).expect("can not ioremap");

            if let Some(transport) = probe_mmio_device(/* ... */, Some(DeviceType::Block)) {
                let dev = Arc::new(VirtBlkDevice::new_from_mmio(transport));
                BLOCK_DEVICE.call_once(|| dev);
                return Some(dev);
            }
        }
    }
    None
}
```

块设备在系统启动时进行初始化，并执行基本的读取测试以验证设备功能。初始化成功后，设备即可为文件系统提供底层存储服务。RISC-V 架构通过 MMIO 方式访问 VirtIO 块设备，LoongArch 架构通过 PCI 方式访问 VirtIO 块设备。

5.3 字符设备驱动

字符设备以字节流的方式进行访问，不支持随机寻址。本内核主要支持串口设备作为系统控制台，兼容多种串口控制器标准，支持可配置的寄存器宽度和偏移量，实现基于 MMIO 的串口访问接口。

字符设备初始化成功后，通过全局 `CHAR_DEVICE` 单例提供系统控制台功能，支持内核日志输出和用户交互。串口设备的初始化过程包括解析设备树中的寄存器配置、中断号等参数，建立内存映射，最终创建可用的串口设备实例。

5.4 网络设备驱动

网络设备负责处理网络数据包的发送和接收。本内核支持 VirtIO 网络设备和回环设备，在 RISC-V 平台上通过 MMIO 方式访问，提供高性能的虚拟化网络传输能力。

```
pub fn init_net(root: &Fdt) {
    let netmeta = probe_virtio_net(root);

    if let Some(net_meta) = netmeta {
        let transport = probe_mmio_device(/* ... */, Some(DeviceType::Network)).unwrap();
        let dev = create_virt_net_dev(transport).expect("create virt net failed");
        init_network(dev, false);
```

```
    } else {
        log::info!("can't find qemu virtio-net, using loopback");
        init_network(LoopbackDev::new(), true);
    }
}
```

当无法检测到硬件网络设备时 ,内核会自动启用回环设备作为备选方案 ,用于本地网络通信和测试。网络设备初始化完成后，内核会调用 `init_network` 函数启动网络协议栈，为上层应用提供网络通信服务。

5.5 设备管理架构

本内核的设备管理采用了模块化的架构设计，通过条件编译支持不同的硬件架构。RISC-V 使用 MMIO 模块，LoongArch 使用 PCI 模块，统一的设备抽象接口屏蔽了底层差异。

```
#[cfg(target_arch = "riscv64")]
pub mod mmio;
#[cfg(target_arch = "loongarch64")]
pub mod pci;
```

资源管理方面，内核使用 `ioremap` 函数管理设备的内存映射，支持动态的地址空间分配和回收 ,确保设备访问的内存安全性。错误处理机制完善 ,当主要设备不可用时提供备选方案 ,详细的调试日志记录便于问题诊断。

全局设备管理使用 `once_cell` 提供全局设备单例 ,确保设备的线程安全访问 ,支持设备的延迟初始化。本内核的设备驱动框架虽然相对简化 ,但已能满足基本的系统需求 ,支持在虚拟化环境中稳定运行，为上层应用提供可靠的硬件抽象服务。

6 文件系统设计

6.1 虚拟文件系统 (VFS)

虚拟文件系统（Virtual File System，简称 VFS）是内核中负责与各种字符流（如磁盘文件，IO 设备等等）对接，并对外提供操作接口的子系统。它为用户程序提供了一个统一的文件和文件系统操作接口，屏蔽了不同文件系统之间的差异和操作细节。这意味着，用户程序可以使用标准的系统调用，如 `open()`、`read()`、`write()` 来操作文件，而无需关心文件实际存储在哪种类型的文件系统或存储介质上。

Nighthawk OS 的虚拟文件系统以 Linux 为师，并充分结合 Rust 语言的特性，从面向对象的角度出发对虚拟文件系统进行了设计和优化。

目前虚拟文件系统包含 `SuperBlock`，`Inode`，`Dentry`，`File` 等数据结构。

6.1.1 VFS 核心抽象

6.1.1.1 SuperBlock

超级块对象用于存储特定文件系统的信息，通常对应于存放在磁盘特定扇区中的文件系统超级块。超级块是对文件系统的具象，换句话说，一个超级块对应一个文件系统的实例。对于基于磁盘上的文件系统，当文件系统被挂载内核时，内核需要读取文件系统位于磁盘上的超级块，并在内存中构造超级块对象；当文件系统卸载时，需要将超级块对象释放，并将内存中的被修改的数据写回到磁盘。对于并非基于磁盘上的文件系统（如基于内存的文件系统，比如 sysfs），就只需要在内存构造独立的超级块。

超级块由 `SuperBlock` trait 定义，如下：

```
pub trait SuperBlock: Send + Sync {
    /// Get metadata of this super block.
    fn meta(&self) -> &SuperBlockMeta;

    /// Get filesystem statistics.
    fn stat_fs(&self) -> SysResult<StatFs>;

    /// Called when VFS is writing out all dirty data associated with a
    /// superblock.
    fn sync_fs(&self, wait: isize) -> SysResult<()>;
}
```

与传统的面向对象编程语言（如 Java 或 C++）不同，Rust 没有内置的类继承机制，而是鼓励使用组合和 trait 来实现代码复用和抽象。如果要模拟继承特性，就需要设计 Meta 结构体来表示对基类的抽象，为了使用继承来简化设计，减少冗余代码，超级块基类对象的设计由 `SuperBlockMeta` 结构体表示。

```
pub struct SuperBlockMeta {
    /// Block device that hold this file system.
    pub device: Option<Arc<dyn BlockDevice>>,
    /// File system type.
    pub fs_type: Weak<dyn FileSystemType>,
    /// Root dentry points to the mount point.
    pub root_dentry: Once<Arc<dyn Dentry>>,
}
```

对于具体的文件系统，只需要实现自己的超级块对象，其中包含 `SuperBlockMeta` 的字段，就能完成继承对超级块基类的继承。比如对 FAT32 文件系统，我们只需要构造这样一个 `FatSuperBlock` 对象就能完成对 VFS `SuperBlockMeta` 的继承，同时，只需要为 `FatSuperBlock` 实现 `SuperBlock` trait 就能实现对接口方法的多态行为。这样就能在 Rust 语言中使用面向对象的设计来大大简化具体文件系统与 VFS 层接口对接的代码量。

```
pub struct FatSuperBlock {
    meta: SuperBlockMeta,
    fs: Arc<FatFs>,
}
```

6.1.1.2 Inode

索引节点是对文件系统中文件信息的抽象。对于文件系统中的文件来说，文件名可以随时更改，但是索引节点对文件一定是唯一的，并且随文件的存在而存在。

索引节点由 `Inode` trait 表示，如下：

```
pub trait Inode: Send + Sync + DowncastSync {
    /// Get metadata of this Inode
    fn meta(&self) -> &InodeMeta;

    /// Get attributes of this file
    fn get_attr(&self) -> SysResult<Stat>;
}
```

索引节点对象由 `InodeMeta` 结构体表示，下面给出它的结构和描述：

```rust
pub struct InodeMeta {
    /// Inode number.
    pub ino: usize,
    /// Mode of inode.
    pub mode: InodeMode,
    /// Device id for device inodes, e.g. tty device inode.
    pub dev_id: Option<DevId>,
    /// Super block this inode belongs to.
    pub super_block: Weak<dyn SuperBlock>,
    /// File page cache.
    pub page_cache: Option<PageCache>,
    /// Mutable date with mutex protection.
    pub inner: Mutex<InodeMetaInner>,
}

pub struct InodeMetaInner {
    /// Size of a file in bytes.
    pub size: usize,
    /// Last access time.
    pub atime: TimeSpec,
    /// Last modification time.
    pub mtime: TimeSpec,
    /// Last status change time.
    pub ctime: TimeSpec,
    /// State of the underlying file.
    pub state: InodeState,
}
```

6.1.1.3 Dentry

目录项是管理文件在目录树中的信息的结构体，是对文件路径的抽象。在文件系统中，以挂载点，即文件系统的根目录为根节点，按照文件夹与下属文件的父子关系逐级向下，形成一个目录树的结构。目录树的每个节点对应一个目录项，每一个目录项都指向一个文件的索引节点。

Dentry 存在的必要性源于 Unix 将文件本身与文件名解耦合的设计，这使得不同的目录项可以指向相同的索引节点（即硬链接）。虽然竞赛规定使用的 FAT32 文件系统在设计上是将路径与文件本身耦合的，这也导致其不支持硬链接技术，因而往届很多作品并没有 Dentry 这个结构，而是将路径解析的功能保存在 Inode 结构体中，这样的做法是针对竞赛的简化，然而，这并不符合 Unix 哲学，这种 VFS 设计并不能扩展到其他文件系统上。而 Nighthawk 认为遵守 Unix 设计哲学能有更好的扩展性，因此，Nighthawk 选择遵守 Unix 设计规范，将路径与文件本身相分离，形成了 Dentry 和 Inode 这两者的抽象。

目录项与索引节点是多对一的映射关系，因此文件系统只需要缓存目录项就能缓存对应的索引节点。而目录项的状态分为两种，一种是被使用的，即正常指向 Inode 的目录项，一种

是负状态，即没有对应 Inode 的目录项。负目录项的存在是因为文件系统试图访问不存在的路径，或者文件被删除了。如果没有负目录项，文件系统会到磁盘上遍历目录结构体并检查这个文件的确不存在，这样的失败查找非常浪费资源，为了尽量减少对磁盘的 IO 访问，Nighthawk 的文件系统会缓存这些负目录项，以便快速解析这些路径。

目录项的操作由 `Dentry` trait 描述，定义如下：

```
pub trait Dentry: Send + Sync {
    /// Get metadata of this Dentry
    fn meta(&self) -> &DentryMeta;

    /// Open a file associated with the inode that this dentry points to.
    fn base_open(self: Arc<Self>) -> SysResult<Arc<dyn File>>;

    /// Look up in a directory inode and find file with `name`.
    ///
    /// If the named inode does not exist, a negative dentry will be created
    /// as a child and returned. Returning an error code from this routine
    /// must only be done on a real error.
    fn base_lookup(
        self: Arc<Self>,
        name: &str,
    ) -> SysResult<Arc<dyn Dentry>>;

    /// Called by the open(2) and creat(2) system calls. Create an inode for
    /// a dentry in the directory inode.
    ///
    /// If the dentry itself has a negative child with `name`, it will
    /// create an inode for the negative child and return the child.
    fn base_create(
        self: Arc<Self>,
        name: &str,
        mode: InodeMode,
    ) -> SysResult<Arc<dyn Dentry>>;

    /// Called by the unlink(2) system call. Delete a file inode in a
    /// directory inode.
    fn base_unlink(self: Arc<Self>, name: &str) -> SyscallResult;

    /// Called by the rmdir(2) system call. Delete a dir inode in a
    /// directory inode.
    fn base_rmdir(self: Arc<Self>, name: &str) -> SyscallResult;
}
```

目录项对象由 `DentryMeta` 结构体表示：

```
pub struct DentryMeta {
    /// Name of this file or directory.
    pub name: String,
```

```
    /// Super block this dentry belongs to
    pub super_block: Weak<dyn SuperBlock>,
    /// Parent dentry. `None` if root dentry.
    pub parent: Option<Weak<dyn Dentry>>,
    /// Inode it points to. May be `None`, which is called negative dentry.
    pub inode: Mutex<Option<Arc<dyn Inode>>>,
    /// Children dentries. Key value pair is <name, dentry>.
    pub children: Mutex<BTreeMap<String, Arc<dyn Dentry>>>,
}
```

6.1.1.4 File

文件对象是进程已打开的文件在内存中的表示。文件对象由系统调用 `open()` 创建，由系统调用 `close()` 撤销，所有文件相关的系统调用实际上都是文件对象定义的操作。文件对象与文件系统中的文件并不是一一对应的关系，因为多个进程可能会同时打开同一个文件，也就会创建多个文件对象，但这些文件对象指向的索引节点都是同一个索引节点，即同一个文件。

文件对象的操作由 `File` 描述，其形式如下：

```
pub trait File: Send + Sync {
    /// Get metadata of this file
    fn meta(&self) -> &FileMeta;

    /// Called by read(2) and related system calls.
    ///
    /// On success, the number of bytes read is returned (zero indicates
    /// end of file), and the file position is advanced by this number.
    async fn read(&self, offset: usize, buf: &mut [u8]) -> SyscallResult;

    /// Called by write(2) and related system calls.
    ///

    /// On success, the number of bytes written is returned, and the file
    /// offset is incremented by the number of bytes actually written.
    async fn write(&self, offset: usize, buf: &[u8]) -> SyscallResult;

    /// Read directory entries. This is called by the getdents(2) system
    /// call.
    ///
    /// For every call, this function will return an valid entry, or an
    /// error. If it read to the end of directory, it will return an empty
    /// entry.
    fn base_read_dir(&self) -> SysResult<Option<DirEntry>>;

    /// Called by the close(2) system call to flush a file
    fn flush(&self) -> SysResult<usize>;

    /// Called by the ioctl(2) system call.
    fn ioctl(&self, cmd: usize, arg: usize) -> SyscallResult;
```

```
    /// Called when a process wants to check if there is activity on this
    /// file and (optionally) go to sleep until there is activity.
    async fn poll(&self, events: PollEvents) -> SysResult<PollEvents>;

    /// Called when the VFS needs to move the file position index.
    ///
    /// Return the result offset.
    fn seek(&self, pos: SeekFrom) -> SysResult<usize>;
}
```

文件对象的设计由 `FileMeta` 结构体表示，下面给出它的结构和描述：

```
pub struct FileMeta {
    /// Dentry which points to this file.
    pub dentry: Arc<dyn Dentry>,
    /// Inode which points to this file
    pub inode: Arc<dyn Inode>,
    /// Offset position of this file.
    pub pos: AtomicUsize,
    /// File mode
    pub flags: Mutex<OpenFlags>,
}
```

6.1.1.5 FileSystemType

`FileSystemType` 用来描述各种特定文件系统类型的功能和行为，并负责管理每种文件系统下的所有文件系统实例以及对应的超级块。

`FileSystemType` trait 的定义如下：

```
pub trait FileSystemType: Send + Sync {
    fn meta(&self) -> &FileSystemTypeMeta;

    /// Call when a new instance of this filesystem should be mounted.
    fn base_mount(
        self: Arc<Self>,
        name: &str,
        parent: Option<Arc<dyn Dentry>>,
        flags: MountFlags,
        dev: Option<Arc<dyn BlockDevice>>,
    ) -> SysResult<Arc<dyn Dentry>>;

    /// Call when an instance of this filesystem should be shut down.
    fn kill_sb(&self, sb: Arc<dyn SuperBlock>) -> SysResult<()>;
}
```

`FileSystemType` 的设计由 `FileSystemTypeMeta` 结构体表示，下面给出它的结构和描述：

```
pub struct FileSystemTypeMeta {
    /// Name of this file system type.
    name: String,
    /// Super blocks.
    supers: Mutex<BTreeMap<String, Arc<dyn SuperBlock>>>,
}
```

6.1.1.6 Path

`Path` 结构体的主要用来实现路径解析，由于我们在 `DentryMeta` 中使用 `BTreeMap` 来对缓存一个文件夹下的所有子目录项 ,因此我们能够在内存中快速进行路径解析 ,而无需重复访问磁盘进行耗时的 IO 操作。

```
pub struct Path {
    /// The root of the file system
    root: Arc<dyn Dentry>,
    /// The directory to start searching from
    start: Arc<dyn Dentry>,
    /// The path to search for
    path: String,
}
```

由于我们已经通过 Dentry 实现了对目录树的抽象，路径解析的实现非常简单，只需要判断传入路径为绝对路径或相对路径，然后逐级对目录进行查找即可。

```
impl Path {
    /// Walk until path has been resolved.
    pub fn walk(&self) -> SysResult<Arc<dyn Dentry>> {
        let path = self.path.as_str();
        let mut dentry = if is_absolute_path(path) {
            self.root.clone()
        } else {
            self.start.clone()
        };
        for p in split_path(path) {
            match p {
                ".." => {
                    dentry = dentry.parent().ok_or(SysError::ENOENT)?;
                }
                name => match dentry.lookup(name) {
                    Ok(sub_dentry) => {
                        dentry = sub_dentry
                    }
                    Err(e) => {
                        return Err(e);
                    }
                },
```

```
            }
        }
        Ok(dentry)
    }
}
```

6.2 文件描述符与文件表

6.2.1 FdTable

Unix 设计哲学将文件本身抽象成 Inode，其保存了文件的元数据；将内核打开的文件抽象成 File，其保存了当前读写文件的偏移量以及文件打开的标志；进程只能看见文件描述符，文件描述符由进程结构体中的文件描述符表进行处理。

当一个进程调用 `open()` 系统调用，内核会创建一个文件对象来维护被进程打开的文件的信息，但是内核并不会将这个文件对象返回给进程，而是将一个非负整数返回，即 `open()` 系统调用的返回值是一个非负整数，这个整数称作文件描述符。文件描述符和文件对象一一对应，而维护二者对应关系的数据结构，就是文件描述符表。在实现细节中，文件描述符表本质是一个数组，数组中每一个元素就是文件对象，而元素下标就是文件对象对应的文件描述符。

Nighthawk 将 FdTable 定义成一个 `Vec<Option<FdInfo>>`，支持动态增长长度，在 fork 复制 `FdTable` 时比固定大小的数组时间开销更小，并且可以满足 Linux 系统中 RLimit 的限制。

```
#[derive(Clone)]
pub struct FdTable {
    table: Vec<Option<FdInfo>>,
    rlimit: RLimit,
}

#[derive(Clone)]
pub struct FdInfo {
    /// File.
    file: Arc<dyn File>,
    /// File descriptor flags.
    flags: FdFlags,
}
```

6.3 磁盘文件系统实现

6.3.1 FAT32 文件系统

FAT32，全称为 File Allocation Table 32，是一种文件系统格式，用于在各种存储设备上存储和管理文件和目录。它是 FAT 文件系统的一个版本，最初由微软在 1996 年引入，主要是为了解决 FAT16 在处理大容量存储设备时的限制问题。FAT32 文件系统在 Windows 操作系统以及许多其他设备和媒体中得到了广泛应用。

作为为 Windows 设计的文件系统，FAT32 并没有采取 UNIX 系列文件系统的设计范式。相比于 UNIX 系列的文件系统，FAT32 缺少 UNIX 规定的 `rwx` 权限管理，也没有提供硬链接功能或可以实现硬链接功能的模块。虽然要使内核支持 FAT32，只需实现对应的 VFS 接口，但是具体实现仍需要采取一些特殊机制。

Nighthawk 使用了开源的 `rust-fatfs` 库，并在其基础上添加了多核的支持。通过实现 FAT32 的 VFS 层接口完成了 FAT32 的对接。

6.3.2 EXT4 文件系统

Ext4（第四代扩展文件系统）是 Ext3 文件系统的继承者，主要用于 Linux 操作系统。与前代文件系统相比，Ext4 在性能、可靠性和容量方面都有显著改进。相比于初赛要求的 FAT32 文件系统，Ext4 文件系统对 Unix 操作系统适配性更好，支持硬链接等操作。

Nighthawk 使用了开源的 `lwext4-rust` 库，并修改其代码以支持链接功能，以及根据文件偏移获取对应磁盘块号的功能，通过实现 EXT4 的 VFS 层接口完成对接。

6.4 特殊文件系统

在 Nighthawk 中，非磁盘文件系统用于指代所有不需要从磁盘上读取数据的文件系统，包括 procfs, devfs, tmpfs 等。这些文件系统的数据从不落盘，按需从内核中查询。 由于其无需与磁盘交互，因此它们不需要经过常见的为了提高磁盘访问效率而使用的缓存机 制，可以直接实现 VFS 顶层的文件接口，从而减少不必要的性能开销。

6.4.1 procfs

procfs 是一种特殊的文件系统，它不是从磁盘上的文件系统中读取数据，而是从内核中 读取数据。Nighthawk 的 procfs 包括：

- `/proc/mounts`：显示当前挂载的文件系统
- `/proc/meminfo`：提供关于系统内存使用情况的信息，包括总内存、可用内存、缓存和缓冲区等详细数据

6.4.2 devfs

devfs 中的文件代表一些具体的设备，比如终端、硬盘等。Nighthawk 的 devfs 内包含：

- `/dev/zero`：一个无限长的全 0 文件
- `/dev/null`：用于丢弃所有写入的数据，并且读取时会立即返回 EOF（文件结束）
- `/dev/urandom`：一个伪随机数生成器，提供随机数据流
- `/dev/cpu_dma_latency`：控制 CPU 的 DMA 延迟设置，用于调整系统性能
- `/dev/rtc`：实时时钟设备，提供日期和时间
- `/dev/tty`：终端设备，能支持 ioctl 中的特定命令

6.4.3 tmpfs

tmpfs 文件系统中的所有文件和文件夹仅存在于内存中，并在系统重启时被清空。在 Nighthawk 系统中，tmpfs 中的文件内容存储在页缓存中，这使得它们与 mmap (内存映射) 无缝集成，提供了高效的文件操作性能。

6.5 页缓存与块缓存

为了减少读写磁盘的次数，最大化磁盘 IO 性能，Nighthawk 在 Page 模块中实现了页缓存与块缓存，以及二者的统一。

用户程序通过 read/write 系统调用或 mmap 对文件进行读写或内存映射操作，由于内存映射操作以页为单位，因此内核总是按页来存储文件内容，也被称为页缓存。Nighthawk 通过调用外部文件系统库来获取文件内容，外部文件系统库通过调用 Nighthawk 提供的磁盘驱动获取磁盘块的内容。Nighthawk 在磁盘驱动层实现了块缓存。Nighthawk 为每个磁盘块维护一个 `BufferHead` 结构体，每个 `BufferHead` 结构体不存储实际内容，只存储一些元数据和指向存储实际内容的页的指针。

6.5.1 页缓存

页缓存（Page Cache）以页为单位缓存文件内容。被缓存在页缓存中的文件数据能够更快速地被用户读取。对于带有缓冲的写入操作，数据在写入到页缓存中后即可立即返回，而不需等待数据被实际持久化到磁盘，从而提高了上层应用读写文件的整体性能。

页缓存是连接内存模块与文件系统模块桥梁，以页为单位对文件内容的缓存能够与 mmap 等页分配加载机制深度融合，使得文件内容在内存中的映射和缓存更加高效。

Nighthawk 使用哈希表将文件以页为单位的偏移与缓存页对应起来。`PageCache` 被 `InodeMeta` 持有，在文件初次读写时将文件内容从磁盘加载到内存中，并使用`PageCache`结构体统一管理。

```
pub struct PageCache {
    /// Map from aligned file offset to page cache.
    pages: SpinNoIrqLock<HashMap<usize, Arc<Page>>>,
}
```

6.5.2 块缓存

磁盘的最小数据单位是扇区（sector），每次读写磁盘都是以扇区为单位进行操作。扇区大小取决于具体的磁盘类型，有的为 512 字节，有的为 4K 字节。无论用户希望读取 1 个字节，还是 10 个字节，最终访问磁盘时，都必须以扇区为单位读取。如果直接访问裸磁盘，那数据读取的效率会非常低。

同样，如果用户希望向磁盘某个位置写入（更新）1 个字节的数据，他也必须刷新整个扇区。言下之意，就是在写入这 1 个字节之前，我们需要先将该 1 字节所在的磁盘扇区数据全部读出来，在内存中修改对应的这个字节数据，然后再将整个修改后的扇区数据一口气写入磁盘。

为了降低这种低效访问，尽可能提升磁盘访问性能，Nighthawk 实现了块缓存，将频繁访问的磁盘块缓存到内存中，当有数据读取请求时，能够直接从内存中将对应数据读出。当有数据写入时，它可以直接在内存中更新指定部分的数据，然后再通过异步方式，把更新后的数据写回到对应磁盘的扇区中。

QEMU 中 virtio 磁盘块大小为 512B，为一页的八分之一。虽然磁盘块的大小通常不等于页大小，但其缓存内容同样存储在页上。Nighthawk 使用 `BufferCache` 结构体对块缓存进行统一管理，并使用 LRU 算法淘汰最近未访问的块以及存储其内容的页，避免占用过大的内存空间。`BufferHead` 结构体负责存储磁盘块的元信息，包括块偏移、访问次数、块状态，并包括指向实际存储块内容的`Page`结构体指针以及其在页面上的偏移。

```
pub struct BufferCache {
    /// Underlying block device.
    device: Option<Weak<dyn BlockDevice>>,
    /// Block page id to `Page`.
    pub pages: LruCache<usize, Arc<Page>>,
    /// Block idx to `BufferHead`.
    pub buffer_heads: LruCache<usize, Arc<BufferHead>>,
}
```

```
pub struct BufferHead {
    /// Block index on the device.
    block_id: usize,
    page_link: LinkedListAtomicLink,
    inner: SpinNoIrqLock<BufferHeadInner>,
}

pub struct BufferHeadInner {
    /// Count of access before cached.
    acc_cnt: usize,
    /// Buffer state.
    bstate: BufferState,
    /// Page cache which holds the actual buffer data.
    page: Weak<Page>,
    /// Offset in page, aligned with `BLOCK_SIZE`.
    offset: usize,
}
```

6.5.3 页缓存与块缓存的统一

往届作品中，对于页缓存与块缓存的处理，要么像 Alien 那样不考虑页缓存与块缓存的统一，只是简单粗暴的在驱动层面实现块缓存，在 inode 层面实现页缓存，而这种策略会使得一个文件，可能同时存在页缓存和块缓存，不仅导致数据冗余浪费内存空间，并且无法保证页缓存与块缓存数据的同步性；要么像 Titanix 那样不考虑块缓存，只有文件页缓存，对于磁盘上的非文件的频繁访问的块，比如 FAT32 的 FAT 表所在磁盘块，单独做缓存处理，这种策略的缺点在与要求内核自己实现文件系统，因此不能使用外部库提供的高级抽象。 而 Nighthawk 在不自己实现文件系统的情况下将页缓存与块缓存统一了起来，页缓存和块缓存是一个事物的两种表现：对于一个缓存页而言，对上，它是某个文件的一个页缓存，而对下，它同样是一个块设备上的一组块缓存。

页缓存与块缓存统一的难点在于，块设备上的文件是以块为单位存储的，而内核希望按页为单位缓存文件。并且块缓存可以分为两类：基于文件的块缓存和不基于文件的块缓存。基于文件的块缓存以页为粒度进行访问和映射，仅在写回时以块为粒度进行操作； 而不基于文件的块缓存，例如 EXT4 文件系统中的 inode bitmap，我们希望以块为粒度进行访问，但内容仍然需要存储在页上。Nighthawk 使用外部文件系统库提供的对文件的高级抽象，导致 Nighthawk 无法在第一时间准确分辨一个磁盘块到底是基于文件还是不基于文件的，因此我们没有办法预先根据磁盘块号区域划分这两种块缓存，相反我们实现了对磁盘块的访问计数，并根据计数动态判断磁盘块缓存的种类。

Nighthawk 实现块缓存分类的算法是，为磁盘访问计数设置一个阈值，当访问计数超过阈值时，将其视作不基于文件的磁盘块并将其缓存在驱动层。对于基于文件的磁盘块，Nighthawk 调用外部库进行处理，而外部库会调用 Nighthawk 提供的磁盘驱动对磁盘进行访问，最后外部库返回文件的高级抽象，这时 Nighthawk 获取到这个文件后就会立刻将其内容放入页缓存，并调用获取文件偏移对应磁盘块的方法，将对应块号的 `BufferHead` 链接到页缓存上。由于 Nighthawk 在获取文件内容后就立即将对应块识别为基于文件的块，并将其内容存储在页缓存中，因此对该块的访问会通过 `BufferHead` 直接指向页缓存中，不会访问底层磁盘块。

页缓存与块缓存统一不仅可以减少冗余数据的存在，节省内存空间，也能简化管理逻辑，减少开发和维护的难度。在页缓存存在的情况下，对块的访问可以直接获取对应页上的具体内容，无需对磁盘进行读写操作，在页缓存析构时，再将块内容直接写回磁盘，省去了调用外部文件系统写文件接口的时间。

7 进程间通信

7.1 信号机制

信号是操作系统向进程传递事件通知的一种机制，主要用于通知进程发生了异步事件。Nighthawk 与往届作品 Titanix 的信号机制相比，信号机制更加完善。Titanix 的信号队列中只有信号编号，Nighthawk 参考了 Linux 的实现，使用 `SigInfo` 结构体代替，除了能表示信号编号外，还能携带更多的信息：

```
pub struct SigInfo {
    pub sig: Sig,
    pub code: i32,
    pub details: SigDetails,
}
```

这种设计与 POSIX 标准中的 `siginfo_t` 结构体相似，增强了 Nighthawk 系统与标准 POSIX 接口的兼容性。并且使得 Nighthawk 的信号机制具备了更强的表达能力和灵活性。携带附加信息可以是信号的来源、产生原因和相关的上下文数据。例如，`code` 字段可以用于区分不同类型的信号或事件，`details` 字段则可以包含更详细的上下文信息：

```
pub enum SigDetails {
    None,
    Kill {
```

```
        /// sender's pid
        pid: usize,
    },
    CHLD {
        /// which child
        pid: usize,
        /// exit code
        status: i32,
        utime: Duration,
        stime: Duration,
    },
}
```

例如，当子进程状态变为 Zombie 时需要向父进程发送 SIGCHLD 信号告知父进程状态改变，此时将 SigInfo 中的 SigDetails 字段设为 CHLD ，并且告诉父进程自己的 pid 、状态码 status 、用户态运行时间 utime 和内核态运行时间 stime ，这些信息可以有助于父进程在 Wait4 系统调用时掌握子进程的数据。

7.1.1 信号处理函数

在任务由内核态返回到用户态之前，往往需要执行信号处理函数，检查和处理挂起的信号，调用适当的信号处理程序或执行默认行为，确保进程能够正确响应和处理信号。这一机制是操作系统处理异步事件、进程控制和进程间通信的重要组成部分。

这里涉及到一个细节，就是如果此时信号待处理队列中有多个需要处理的信号同时到来，那么应该以什么顺序处理。大部分往届作品都是按照信号到来顺序依次处理的，这种 FIFO(先入先出) 方法虽然简单直接，但在处理高优先级和紧急事件时可能存在不足。例如，当一个需要立即响应的紧急信号和一个普通信号同时到达时，按照到达顺序处理可能导致紧急信号的响应延迟。Nighthawk 在设计信号处理机制时，参考了 Linux 内核的实现，引入了信号优先级的概念，这样的好处是紧急信号可以立即得到处理，减少了响应延迟，提升了系统对关键事件的响应能力。例如，当发生非法内存访问时，会触发 SIGSEGV (Segmentation Fault) 信号，指示程序运行中出现严重问题，这种信号在 Nighthawk 中会优先处理。

用户可以使用 sigaction 系统调用为某些信号自定义信号处理函数，操作系统内核会按照如下步骤来调用用户自定义的信号处理函数：

1. 保存上下文：内核会保存当前的进程执行上下文，包括寄存器状态、堆栈指针、程序计数器等，以便在信号处理完成后恢复进程的执行。
2. 切换到用户态：内核切换到用户态，并开始执行用户自定义的信号处理函数

3. 恢复上下文：使用 `sigreturn` 函数返回到内核态并且恢复之前保存的进程上下文，如寄存器状态、堆栈指针、程序计数器等。

这里同样存在一个问题，即切换到用户态需要保存上下文，那么应该将上下文保存在哪里？部分作品如往届一等奖作品 Titanix 将上下文记录在内核态，这当然也可以，但是对于 POSIX 规范中一些标志位就难以支持，例如设置了 `SA_SIGINFO` 标志位的信号处理程序的函数签名会变成如下形式：

```
void handler(int sig, siginfo_t *info, void *ucontext);
```

这里 `ucontext` 是指向 `ucontext_t` 结构体的指针，提供接收信号时进程的上下文信息，如果将进程上下文保存在了内核态，那么用户将有机会访问到内核中的信息，这无疑是非常不安全的。虽然竞赛并未有测试程序需要用到该标志位，但是 Nighthawk 的目标是实现符合 POSIX 规范的操作系统，因此依然实现了该标志位。

Nighthawk 参考了 Linux 的设计，将信号处理时需要保存的进程上下文保存在了用户栈中，这样 `ucontext` 指向用户栈就非常安全。

7.1.2 系统调用的打断与恢复

在往届参赛作品中，几乎都不支持被含有 `SA_RESTART` 标志的信号打断的系统调用的恢复功能。在类 Unix 操作系统中，如果慢系统调用（如 `sys_read`、`sys_pselect6`）在执行期间被信号打断并且该信号的处理程序（signal handler）被触发，那么：

1. 带有 **`SA_RESTART`** 标志位的信号：信号处理程序返回后，该系统调用会被自动重新启动，而不是直接返回错误。这对某些慢系统调用（如 read、write、select、pselect 等）尤为重要，因为这些调用可能会因为等待外部事件而阻塞很长时间。
2. 没有 **`SA_RESTART`** 标志位的信号：系统调用会被打断并返回一个错误代码，通常是 EINTR（表示系统调用被中断）。调用者需要检查返回值并决定是否重新执行该系统调用

Nighthawk 的设计目标是实现功能完善的操作系统，因此支持了被打断的系统调用的恢复功能。

Nighthawk 采用如下的方案：如果是系统调用陷入内核，并且系统调用返回的是 `SysError::EINTR` 错误，`trap_handler` 函数会返回 `true` 表示系统调用被信号打断了，

```
pub async fn trap_handler(task: &Arc<Task>) -> bool {
    /* skip */
    match cause {
        Trap::Exception(e) => {
            match e {
                Exception::UserEnvCall => {
                    let syscall_no = cx.syscall_no();
                    cx.set_user_pc_to_next();
                    // get system call return value
                    let ret = Syscall::new(task)
                        .syscall(syscall_no, cx.syscall_args())
                        .await;
                    cx.save_last_user_a0();
                    cx.set_user_a0(ret);
                    if ret == -(SysError::EINTR as isize) as usize {
                        return true;
                    }
                }
                /* skip */
            }
        }
        /* skip */
    }
    false
}
```

trap_handler 函数的返回值会传递给 do_signal 函数，在 do_signal 中检测信号是否含有 SA_RESTART 标志位，如果发现需要重启系统调用，会将 TrapContext 中的 sepc 寄存器-4，表示重新执行，由于用户自定义的信号处理程序 signal handler 函数执行完的返回值会将 a0 寄存器覆盖掉，这里使用 restore_last_user_a0 函数进行备份。

```
pub fn do_signal(task: &Arc<Task>, mut intr: bool) -> SysResult<()> {
    /* skip */

    while /* skip */ {
        let action = /* skip */;
        if intr && action.flags.contains(SigActionFlag::SA_RESTART) {
            cx.sepc -= 4;
            cx.restore_last_user_a0();
            intr = false;
        }
        /* skip */
    }
    /* skip */
}
```

7.2 管道机制

7.3 共享内存机制

Nighthawk 通过 `Task` 结构中的 `shm_ids` 字段来支持共享内存。

```
pub struct Task {
    // ...
    /// Map of start address of shared memory areas to their keys in the
    /// shared memory manager.
    shm_ids: Shared<BTreeMap<VirtAddr, usize>>,
    // ...
}
```

在实现写时复制（Copy-on-Write）时，共享内存区域不会被设置为写时复制，以保证多个进程可以真正地共享和修改同一块内存区域。

8 内存管理

8.1 物理内存管理

8.1.1 物理页帧分配器

内核需要管理全部的空闲物理内存，Nighthawk 为此使用了来自 rCore 的仓库的 `bitmap-allocator`。Nighthawk 在内核初始化时，会将所有内核未占用的物理内存加入物理页分配器。

Bitmap allocator 的主要原理是通过一个位图来管理一段连续的内存空间。这个位图中的每一位代表一块内存，如果该位为 0，说明对应的内存块空闲；如果该位为 1，说明对应的内存块已经被分配出去。当需要分配一个指定大小的内存时，bitmap allocator 首先检查位图中是否有足够的连续空闲内存块可以满足分配请求。如果有，就将对应的位图标记为已分配，并返回该内存块的起始地址；如果没有，就返回空指针，表示分配失败。当需要释放已经分配出去的内存时，bitmap allocator 将对应位图标记为未分配。这样，已经释放的内存块就可以被下一次分配请求使用了。

此外，Nighthawk 将物理页帧抽象成 `FrameTracker` 结构体，并结合 RAII 的思想，在结构体析构时自动调用 `dealloc_frame` 函数将页帧释放。

```
/// Manage a frame which has the same lifecycle as the tracker.
pub struct FrameTracker {
    /// PPN of the frame.
    pub ppn: PhysPageNum,
```

```
}

impl Drop for FrameTracker {
    fn drop(&mut self) {
        dealloc_frame(self.ppn);
    }
}
```

8.2 内核动态内存分配

Nighthawk 使用伙伴分配器管理内核所需的动态内存结构，来自 crate `buddy_system_allocator`。

伙伴分配器（Buddy Allocator）是一种内存分配算法，常用于操作系统内核和高性能应用程序中，通过分配和管理内存块来满足不同大小的内存请求，并进行高效的合并和分割操作。其工作原理是将内存块按照 2 的幂次方大小分为多个层级，当需要分配特定大小的内存时，从最小适合该请求的层级开始查找。每个内存块都有一个“伙伴”块，如果块大小是 2^k，那么它的伙伴块也是 2^k 且紧挨着它。通过检查和计算伙伴块的地址，可以快速地进行内存分割与合并。当需要分配的内存块小于当前可用最小块时，将当前块一分为二，直到找到合适大小的块为止；当释放一个内存块时，若其伙伴块也空闲，则将两个块合并为一个更大的块，递归进行直到不能再合并为止。伙伴分配器的优点在于分配和释放内存块的操作非常快速，且通过内存块大小的选择和合并操作，有效减少了外部碎片。

8.3 虚拟地址空间

Nighthawk 采用了与 Linux 类似的虚拟地址空间布局。其中，虚拟地址空间被分为两部分：

1. 用户地址空间：0x0 - 0x0000_ffff_bfff_ffff
2. 内核地址空间：0xffff_fffe_8000_0000 - 0xffff_ffff_ffff_ffff

每个进程都拥有自己独立的用户地址空间。当进程切换时，内核会更新 satp 寄存器，指向当前进程的页表。这种设计确保了进程间的隔离性，一个进程无法直接访问另一个进程的内存，从而提高了系统的安全性。

与 xv6 等教学性内核不同，Nighthawk 的内核地址空间与用户地址空间共享同一个地址空间。这意味着内核代码可以直接访问用户态内存，而无需像一些传统设计那样进行复杂的页表切换或数据拷贝。这种设计的优势在于：

1. 零拷贝：内核可以直接读写用户态的缓冲区，避免了在用户态和内核态之间复制数据，显著提升了 I/O 操作的性能。
2. 高效的系统调用：在处理系统调用时，内核可以直接访问用户传递的指针，简化了实现逻辑。

当然，这种共享地址空间的设计也对安全性提出了更高的要求。为了防止内核意外地访问无效的用户内存，Nighthawk 实现了一套严格的用户指针检查机制。在访问任何用户指针之前，内核都会验证其合法性，确保指针指向有效的、已映射的用户内存区域。

8.4 按需分页与缺页异常

Nighthawk 目前能够利用缺页异常处理来实现写时复制（Copy on write）、懒分配（Lazy page allocation）以及用户地址检查机制和零拷贝技术。

当用户程序因缺页异常返回内核时，内核异常处理函数能够从 `stval` 寄存器读取异常发生的地址，并交给 `VmArea::handle_page_fault` 函数进行处理。

8.4.1 懒分配技术

懒分配技术主要用于堆栈分配以及 mmap 匿名映射或文件映射。在传统的内存分配方法中，操作系统在进程请求内存时会立即为其分配实际的物理内存。然而，这种方法在某些情况下可能导致资源的浪费，因为进程可能并不会立即使用全部分配的内存。

懒分配技术的核心思想是推迟实际物理内存的分配，直到进程真正访问到该内存区域。这样可以优化内存使用，提高系统性能。

对于内存的懒分配，比如堆栈分配，mmap 匿名内存分配，Nighthawk 将许可分配的范围记录下来，但并不进行实际分配操作，当用户访问到许诺分配但未分配的页面时会触发缺页异常，缺页异常处理函数会进行实际的分配操作。

对于 mmap 文件的懒分配，Nighthawk 将其与页缓存机制深度融合，Nighthawk 同样执行懒分配操作，当缺页异常时再从页缓存中获取页面。

8.5 写时复制 (Copy-on-Write)

在 `fork` 进程时，Nighthawk 会将原 `MemorySpace` 中的除共享内存外每一个已分配页的 PTE 都删除写标志位，打上 COW 标志位，然后重新映射到页表中，并将。在用户向 COW 页写入时会触发缺页异常陷入内核，在 `VmArea::handle_page_fault` 函数中，内核会根据 COW 标志位转发给 COW 缺页异常处理函数，缺页异常处理函数会根据 `Arc<Page>` 的原子持有计数判

断是否为最后一个持有者 ,如果不是最后一个持有者 ,会新分配一个页并复制原始页的数据并恢复写标志位重新映射，如果是最后一个持有者，直接恢复写标志位。

```
impl MemorySpace {
    /// Clone a same `MemorySpace` lazily.
    pub fn from_user_lazily(user_space: &mut Self) -> Self {
        let mut memory_space = Self::new_user();
        for (range, area) in user_space.areas().iter() {
            let mut new_area = area.clone();
            for vpn in area.range_vpn() {
                if let Some(page) = area.pages.get(&vpn) {
                    let pte = user_space
                        .page_table_mut()
                        .find_leaf_pte(vpn)
                        .unwrap();
                    let (pte_flags, ppn) = match area.vma_type {
                        VmAreaType::Shm => {
                            // no cow for shared memory
                            new_area.pages.insert(vpn, page.clone());
                            (pte.flags(), page.ppn())
                        }
                        _ => {
                            // copy on write
                            let mut new_flags = pte.flags() | PTEFlags::COW;
                            new_flags.remove(PTEFlags::W);
                            pte.set_flags(new_flags);
                            (new_flags, page.ppn())
                        }
                    };
                    memory_space.page_table_mut().map(vpn, ppn, pte_flags);
                } else {
                    // do nothing for lazy allocated area
                }
            }
            memory_space.push_vma_lazily(new_area);
        }
        memory_space
    }
}
```

9 网络模块

本操作系统的网络模块参考了 Arceos 的实现 ,但 Arceos 的网络模块并不支持异步特性与 IPv6 协议 ,因此我们对其进行了较大改造 ,使其很好地兼容异步无栈协程架构的操作系统 ,并且顺利通过了 libctest、 netperf 测试。

本内核使用了 smoltcp 库作为网络模块的基石，这是一个面向嵌入式设备的 TCP/IP 协议栈库。smoltcp 以 Rust 语言编写，旨在提供高效、可扩展且易于集成的网络堆栈，非常适合资源受限的环境，不依赖于标准库，因此可以在无操作系统的 `no_std` 中运行。

9.1 数据链路层设备

本内核支持本地回环网络设备 `Loopback` 与 `VirtIoNet` 设备。所有的网络设备都需要实现 `NetDevice` trait，这个 trait 为网络设备提供一个统一的接口，使得不同类型的网络设备可以通过相同的接口进行操作和管理。

```
pub trait NetDevice: Sync + Send {
    fn capabilities(&self) -> DeviceCapabilities;
    fn mac_address(&self) -> EthernetAddress;
    fn can_transmit(&self) -> bool;
    fn can_receive(&self) -> bool;
    fn transmit(&mut self, tx_buf: Box<dyn NetBufPtrOps>) -> DevResult;
    fn receive(&mut self) -> DevResult<Box<dyn NetBufPtrOps>>;
    fn alloc_tx_buffer(&mut self, size: usize) -> DevResult<Box<dyn NetBufPtrOps>>;
}
```

为了使得网络设备符合 `smoltcp` 的接口需求 ,定义了 `DeviceWrapper` 结构体 ,它通过 `RefCell` 包装 `Box<dyn NetDevice>` ，允许内部的可变访问，以便在实现 `Device` trait 时提供对底层设备的操作。

为了提供一个高层次的网络接口封装，定义了 `InterfaceWrapper` 结构体，包含设备和接口的详细信息，并实现相关的操作方法。它通过 `Mutex` 来保护对设备和接口的并发访问，确保线程安全。

```
struct InterfaceWrapper {
    name: &'static str,
    ether_addr: EthernetAddress,
    dev: Mutex<DeviceWrapper>,
    iface: Mutex<Interface>,
}
```

9.2 网络层 IP 协议

本内核支持 IPv4 与 IPv6 两种地址，在如 `sys_bind` 等系统调用中，操作系统需要接受用户传入的 IP 地址。在 POSIX 规范中，系统调用传入的 IPv4 address 参数的端口为网络字节序，即大端序，而 RISC-V 指令集为小端序，本内核在内核中使用 `smoltcp` 库提供的 `IpEndpoint` 结构体存储网络地址，以小端序存储。

```
pub struct SockAddrIn {
    pub family: u16,
    pub port: [u8; 2],
    pub addr: [u8; 4],
    pub zero: [u8; 8],
}

pub struct IpEndpoint {
    pub addr: Address,
    pub port: u16,
}
```

本内核为结构体实现了 `From` trait 便于这些结构体进行转换。IP 协议是网络层的核心协议，负责在不同网络之间传输数据包。数据包的接受、处理、发送以及路由处理的逻辑已经由 `smoltcp` 模块封装好了。

9.3 传输层 UDP 与 TCP

`smoltcp` 库本身在 tcp 模块和 udp 模块就提供了相应的 `Socket` 的实现，但本内核重新封装了一遍 `UdpSocket` 和 `TcpSocket`。这是为了提供更高层的抽象和接口，虽然 `smoltcp` 的 `udp::Socket` 和 `tcp::Socket` 提供了基本的 UDP 与 TCP 功能，但它的接口不完全符合 Unix 操作系统的需求。

重新定义的 `UdpSocket` 与 `TcpSocket` 结构体，用 `async` 块封装实现了异步特性，并且提供了与 POSIX 标准类似的接口，使得基于 POSIX API 设计的应用程序更容易移植和使用。在内核的 `UdpSocket` 与 `TcpSocket` 结构体中，使用 `handle` 字段表示套接字在全局 `SOCKET_SET` 中的句柄，便于在各种操作中快速访问和管理。

9.3.1 UDP 套接字

内核中的 UDP 结构体除了存储 `SocketHandle` 外，还存储了本地地址 `local_addr` 与远程地址 `peer_addr`，均使用 `RwLock` 提供并发读写保护，`Option` 类型允许本地地址未绑定与远程地址未连接时为 `None`。`nonblock` 用于指示套接字是否处于非阻塞模式，`AtomicBool` 提供原子操作，确保在多线程环境下对非阻塞模式标志进行安全的读写操作。

```
pub struct UdpSocket {
    handle: SocketHandle,
    local_addr: RwLock<Option<IpListenEndpoint>>,
    peer_addr: RwLock<Option<IpEndpoint>>,
    nonblock: AtomicBool,
}
```

9.3.2 TCP 套接字

`TcpSocket` 结构体大体上与 `UdpSocket` 类似，但采用了更加高效的无锁设计。与往届作品不同的创新点主要体现在以下几个方面：

无锁状态管理：往届作品中基本都是使用 `Mutex` 这种自旋锁对 TCP 套接字进行并发安全处理，本内核使用 `AtomicU8` 原子变量表示套接字的状态，允许多个线程在不使用锁的情况下安全地对套接字状态进行读取和修改。相比传统的锁机制，原子操作更轻量级，减少了上下文切换和线程阻塞，从而提高了并发性能。

```
pub struct TcpSocket {
    state: AtomicU8,
    shutdown: UnsafeCell<u8>,
    handle: UnsafeCell<Option<SocketHandle>>,
    local_addr: UnsafeCell<IpEndpoint>,
    peer_addr: UnsafeCell<IpEndpoint>,
    nonblock: AtomicBool,
}

fn update_state<F, T>(&self, expect: u8, new: u8, f: F) -> Result<SysResult<T>, u8>
where
    F: FnOnce() -> SysResult<T>,
{
    match self.state.compare_exchange(expect, STATE_BUSY, Ordering::Acquire, Ordering::Acquire) {
        Ok(_) => {
            let res = f();
            if res.is_ok() {
                self.set_state(new);
            } else {
                self.set_state(expect);
            }
            Ok(res)
        }
        Err(old) => Err(old),
    }
}
```

高效端口管理：`ListenTable` 使用一个大小为 65536 的数组来管理每个可能的 TCP 端口，每个端口对应一个 `ListenTableEntry`。这种设计通过数组的索引来直接访问监听条目，使得查找和操作非常高效。

```
pub struct ListenTable {
    tcp: Box<[Mutex<Option<Box<ListenTableEntry>>>]>,
}

struct ListenTableEntry {
    listen_endpoint: IpListenEndpoint,
```

```
    syn_queue: VecDeque<SocketHandle>,
    waker: Waker,
}
```

`ListenTableEntry` 中的 `syn_queue` 用于管理在三次握手过程中收到的 SYN 包，等待其连接建立完成。`waker` 用于在有新连接到来时唤醒对应的监听套接字，从而处理新连接请求。

IPv4 和 IPv6 的灵活监听： `ListenTableEntry` 的 `can_accept` 方法支持 IPv4-mapped IPv6 addresses，使得在特殊情况下 IPv6 套接字可以接受 IPv4 的连接。这种设计提供了更大的灵活性，允许在同一个端口上同时监听 IPv4 和 IPv6 的连接。

9.4 套接字 API

为了统一 UDP 套接字、TCP 套接字和 Unix 套接字，本内核对其再次进行了封装。文件和套接字通常都作为文件描述符处理，以统一的方式进行读写和管理。为 `Socket` 结构体实现 `File` trait，套接字可以像普通文件一样进行读写操作，方便管理和使用。

```
pub enum Sock {
    Tcp(TcpSocket),
    Udp(UdpSocket),
    Unix(UnixSocket),
}

pub struct Socket {
    pub types: SocketType,
    pub sk: Sock,
    pub meta: FileMeta,
}
```

本内核的 `File` trait 为异步函数，通过实现异步读写方法，可以利用 Rust 的异步特性，提高套接字操作的效率。本内核对网络模块中的资源管理充分使用了 RAII 思想，确保在创建和销毁时正确地分配和释放资源，当 `Socket` 被 `drop` 时，会通过 `shutdown` 关闭套接字并从全局的 `SOCKET_SET` 中移除套接字句柄，确保系统不再持有对该套接字的引用，防止资源泄漏。

9.5 网络数据包处理

本内核使用了最新版本的 smoltcp，去除了接受信息数据旁路 preprocess 处理，改为提前分配一些监听 handle，然后在接受的地方进行处理。网络数据包的处理通过 `NetRxToken` 和 `NetTxToken` 实现，这两个结构体分别实现了 smoltcp 的 `RxToken` 和 `TxToken` trait。

在接收数据包时，NetRxToken 会调用 snoop_tcp_packet 函数对 TCP 数据包进行预处理，检查是否为新的连接请求。对于 SYN 包，系统会自动创建相应的套接字并加入到监听队列中，实现了高效的连接管理。

```
pub(crate) struct NetRxToken<'a>(
    pub(crate) &'a RefCell<Box<dyn NetDevice>>,
    pub(crate) Box<dyn NetBufPtrOps>,
);

impl RxToken for NetRxToken<'_> {
    fn consume<R, F>(self, f: F) -> R
    where
        F: FnOnce(&[u8]) -> R,
    {
        let medium = self.0.borrow().capabilities().medium;
        let is_ethernet = medium == Medium::Ethernet;
        crate::tcp::snoop_tcp_packet(self.1.packet(), is_ethernet).ok();

        let mut rx_buf = self.1;
        let result = f(rx_buf.packet_mut());
        self.0.borrow_mut().recycle_rx_buffer(rx_buf).unwrap();
        result
    }
}
```

与往届一等奖参赛作品相比，本内核提供了更加高效的对套接字的 Poll 操作。通过调用 smoltcp 的 poll_delay 方法检查多个条件来决定下次调用 poll 的时间，得到时间后，新构建一个定时器 PollTimer 放到 TimerManager 中，当时钟中断到来时检查如果定时器超时了，调用回调函数 callback 自动进行 poll 操作，实现了高效的网络事件处理机制。

本内核的网络模块设计充分考虑了异步特性和高并发场景，通过无锁设计、高效的端口管理和智能的数据包处理，为上层应用提供了高性能的网络通信服务。

10 附录

（可以写调试和通过测例的过程，参考 Titanix 文档）

11 总结与展望

本项目成功地基于 Rust 语言设计并实现了一个功能较为完备的宏内核操作系统。在开发过程中，我们深入探索了现代操作系统设计的核心技术，采用异步协程架构和跨架构设计，支持 RISC-V 和 LoongArch 两种指令集架构，并通过实践加深了对理论知识的理解。

11.1 主要工作成果

本项目在多个核心领域取得了显著成果，构建了一个现代化的操作系统内核。

内核架构：确立了宏内核的设计路线，采用模块化的代码结构和异步协程架构。通过条件编译实现跨架构支持，同一份代码可以在不同硬件平台上运行，体现了良好的可移植性和可维护性。 进程管理：实现了完整的进程/线程模型和基于异步协程的任务调度器。每个用户任务在 task_executor_unit 中运行，通过 trap_return、 trap_handler 和 async_syscall 实现用户态与内核态的高效切换。支持进程创建、退出、等待等完整的生命周期管理。 内存管理：构建了支持分页、虚拟内存、按需加载和写时复制的内存管理系统。实现了基于硬件 MMU 的高效用户指针检查机制，支持零拷贝技术，避免了用户态数据到内核态数据的不必要拷贝，提升了系统性能。 异步系统调用：实现了支持异步处理的系统调用框架，提供超过 100 个与 Linux 兼容的系统调用接口。异步系统调用使得 I/O 密集型操作不会阻塞整个内核，显著提高了系统的并发性能和响应能力。 设备驱动：设计了跨架构的设备驱动框架，支持块设备、字符设备和网络设备。通过基于设备树的设备发现机制，实现了硬件抽象和驱动程序的统一管理，为不同硬件平台提供了一致的访问接口。网络协议栈：基于 smoltcp 库实现了完整的网络功能，支持 IPv4 和 IPv6 协议。采用无锁设计的 TCP 套接字管理，实现了高效的端口管理和连接处理机制，提供了与 POSIX 兼容的套接字 API。

我们充分利用了 Rust 语言的安全性、并发性和高性能等优势，在没有垃圾回收的条件下，构建了一个健壮、高效的内核。通过类型系统保证内存安全，通过所有权机制防止数据竞争，实现了系统级编程的安全性和性能的统一。

11.2 技术创新与特色

本项目在多个方面体现了技术创新：

异步协程架构：采用基于 Rust async/await 的异步协程调度模型，每个用户任务运行在独立的异步执行单元中，提供了比传统线程模型更高的并发性能和更低的资源消耗。 跨架构设计：通过模块化设计和条件编译，实现了对 RISC-V 和 LoongArch 架构的统一支持。同一份内核代码可以在不同硬件平台上编译运行，大大提高了代码的复用性和可维护性。 零拷贝用户指针处理：基于硬件 MMU 实现了高效的用户指针合法性检查，支持内核态直接访问用户地址空间，避免了传统的软件地址翻译和数据拷贝开销。

11.3 开发经验总结

在项目开发过程中，我们积累了宝贵的经验：

系统调用实现：严格参考 System Calls Manual 手册，确保与 Linux raw syscall 规范的兼容性。通过详细的系统调用文档和测试用例，保证了接口的正确性和完整性。 调试策略：多核环境下的调试极具挑战性，我们建立了完善的日志系统，通过运行时信息收集和分析来定位问题。对于用户态程序调试，需要紧密结合 libc 实现，根据系统调用序列推断程序执行状态。代码质量：坚持多重构代码的原则，保持代码的简洁性和可读性，这大大降低了后期维护和功能扩展的难度。广泛使用断言机制，在问题发生的早期阶段就能够发现并定位，避免了后续难以追踪的复杂问题。 性能优化：通过性能分析工具和基准测试，识别系统瓶颈并进行针对性优化。异步架构和零拷贝技术的应用显著提升了系统的整体性能。

11.4 未来发展方向

尽管本系统已经初具规模，但仍有许多可以改进和扩展的方向：

硬件适配：计划适配开发板，完善相关硬件驱动程序，扩大系统的硬件支持范围。这将验证系统的跨平台能力，并为更多硬件平台提供支持。 测试覆盖：支持更多 LTP 测例，修复更多内核不稳定的 bug，提高系统的稳定性和兼容性。通过扩大测试覆盖面，确保系统在各种场景下的可靠性。 网络性能优化：进一步提升网络协议栈的性能，优化数据包处理路径，支持更高的网络吞吐量。计划支持 SSH 服务，为远程管理和开发提供便利。 图形界面支持：构建基础的图形用户界面（GUI）系统，为用户提供更直观的交互体验。这将包括窗口管理、图形渲染和输入设备支持等功能。

11.5 项目意义与影响

本项目不仅是一个技术实现 ,更是对现代操作系统设计理念的实践和验证。通过使用 Rust 语言开发操作系统内核 ,我们验证了内存安全语言在系统级编程中的可行性和优势。异步协程架构的成功应用为操作系统设计提供了新的思路和方法。

作为一个开源项目 ,本系统为操作系统领域的学习者和研究者提供了有价值的参考实现。清晰的代码结构、详细的文档和完整的测试用例，为相关研究和教学提供了良好的基础。