操作系统内核设计

RocketOS

哈尔滨工业大学

2025 年 6 月

目录

第 1 章　概述

1.1　RocketOS 内核整体设计

RocketOS 是一款采用 Rust 语言开发的现代化宏内核操作系统，专为 RISC-V 和 LoongArch 架构深度优化。该系统采用同步有栈式设计架构，集成了完整的中断处理机制、进程管理系统、内存管理模块、文件系统以及网络协议栈等核心组件，通过精心设计的系统调用接口为用户程序提供高效可靠的服务支持，详细结构如图 1 所示。RocketOS 充分利用 Rust 语言的内存安全特性和零成本抽象能力，在保证系统稳定性的同时实现了出色的内核性能表现，为现代计算环境提供了一个安全、高效、可扩展的操作系统解决方案。

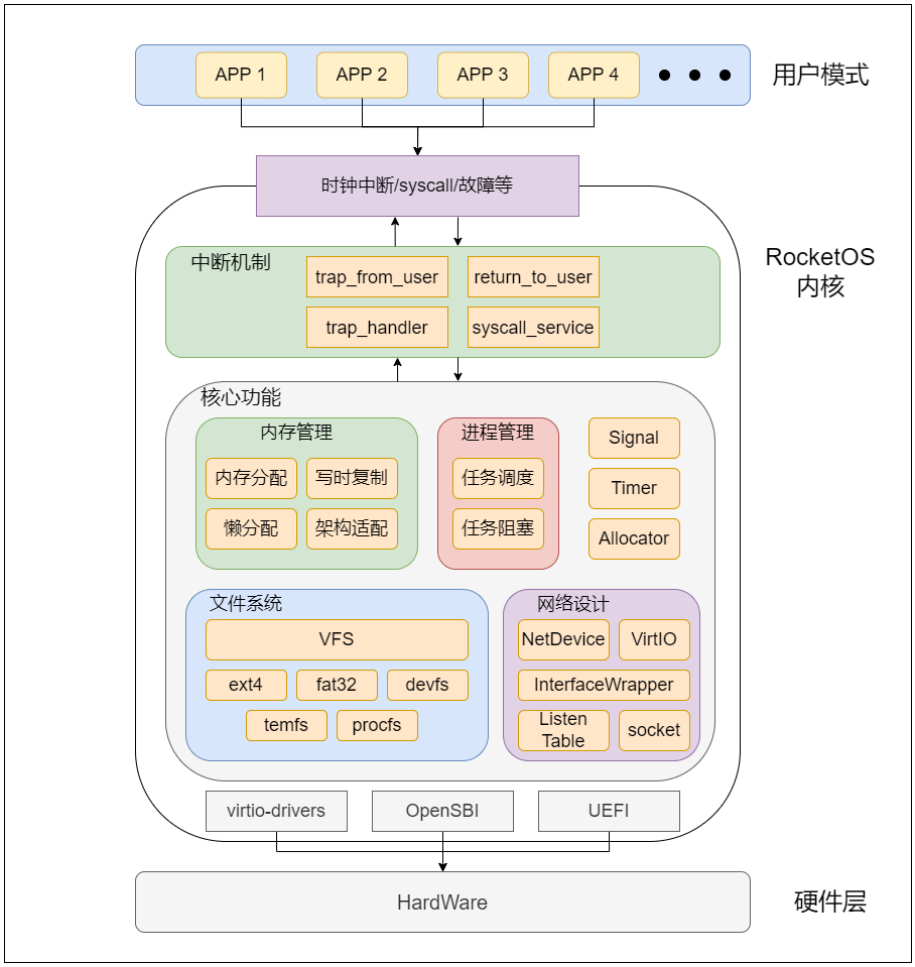

图 1-1　RocketOS 内核架构图

1.2 RocketOS 当前工作

截至初赛结束前，RocketOS 已经实现了以下核心功能模块：

- 内存管理模块，提供高效的内存分配和虚拟内存
- 文件系统，支持多种文件操作和目录管理
- 完善的信号机制，支持进程间的信号传递和处理
- 进程管理系统，支持多任务调度和进程间通信
- 网络协议栈，支持 TCP/IP 协议和基本网络通信

第 2 章　架构适配

2.1　架构适配的总体设计理念

RocketOS 在多架构支持方面采用了"条件编译驱动的架构特化"设计理念，通过编译时的架构选择机制实现针对不同目标架构的深度优化。该设计理念的核心在于充分利用编译器的条件编译特性，在保持单一代码库的前提下，为 RISC-V 和 LoongArch 两种架构提供完全针对性的实现路径。整个适配策略遵循"一次设计，多重实现"的工程原则，既保证了代码架构的优雅性，又确保了在不同硬件平台上的最优性能表现。

这种设计理念的优势在于：

1. 代码复用性：通过条件编译，RocketOS 能够在不同架构之间共享大量的代码，减少重复工作，提高开发效率。
2. 可维护性：通过清晰的架构分层和条件编译，RocketOS 的代码结构更加清晰，易于理解和维护。
3. 灵活性：这种设计允许在不影响整体架构的情况下，快速适配新的硬件平台或架构变更，降低了系统的适应成本。

2.2　适配工作的技术框架

RocketOS 的多架构适配基于条件编译驱动的源码组织框架，通过编译配置系统和源码结构实现架构特化的代码生成。整个技术框架围绕三个核心维度展开，形成了完整的多架构支持体系。

源码组织维度采用"共享核心逻辑、分离架构实现"的文件结构设计理念。系统将通用的算法实现、数据结构操作和业务流程控制等核心逻辑放置在架构无关的源文件中，确保代码的复用性和维护性。而涉及硬件直接交互的关键代码则按架构分别组织，通过精心设计的条件编译指令，编译器能够根据目标架构自动选择正确的源文件和代码段，生成针对特定架构优化的内核镜像。

编译配置维度实现了基于 Makefile 的灵活配置机制。顶层 Makefile 通过 ARCH 环境变量和 MODE 变量实现架构和构建模式的统一控制。系统支持单架构构建和

多架构并行构建，构建系统根据架构参数自动设置相应的编译工具链、目标三元组和链接脚本，确保生成的内核镜像完全符合目标架构的 ABI 规范。

代码实现维度充分利用 Rust 语言的 cfg 属性实现架构特化编程。与传统 C 预处理器相比，Rust 的 cfg 属性能够在保持语法完整性的前提下进行条件编译，编译器对未激活的代码路径进行语法检查但不生成目标代码。这种机制不仅支持粗粒度的架构选择，更支持细粒度的硬件特性配置。

2.3 适配工作的覆盖范围

RocketOS 的多架构适配工作涵盖了操作系统的多个核心子系统，形成了从底层硬件接口到上层系统服务的完整适配体系。每个子系统都遵循"架构无关接口、架构相关实现"的设计原则，确保了系统功能的完整性和性能的最优化。

启动引导子系统负责系统的初始化和硬件检测工作。该子系统实现了架构特定的启动流程，包括 RISC-V 的 SBI 引导机制和 LoongArch 的 UEFI/Legacy 启动支持。通过统一的启动抽象接口，系统能够在不同的固件环境下完成硬件初始化、内存检测和内核加载等关键任务。

内存管理子系统提供了完整的虚拟内存管理功能，包括页表管理、地址空间分配、内存映射和缓存控制等核心服务。该子系统针对两种架构的页表格式差异和 MMU 特性，实现了高效的内存分配算法和页面置换策略，同时通过架构特定的 TLB 管理机制优化内存访问性能。详细内容请参见：RocketOS 的内存设计

中断处理子系统负责系统的中断管理和异常处理工作。该子系统根据两种架构的中断控制器设计和异常处理模型，实现了统一的中断注册接口和处理框架。通过架构特定的中断向量表和优先级管理机制，系统能够高效处理各类硬件中断和软件异常。详细内容请参见：RocketOS 的中断设计

文件系统子系统提供了完整的文件管理和存储服务功能。该子系统基于 VFS 架构实现了文件系统的抽象和管理，支持多种文件系统格式和存储设备。通过架构无关的文件操作接口和缓存机制，系统确保了文件系统服务的高性能和可靠性。详细内容请参见：RocketOS 的 VFS 设计

网络系统子系统提供了完整的网络通信与设备管理功能。该子系统基于 smoltcp 协议栈构建，抽象了网络设备、接口与套接字管理，支持 IPv4/IPv6、TCP/

UDP 等多种协议族。通过统一的设备封装与轮询机制，系统实现了跨架构的高效网络支持，并兼容 RISC-V 与 LoongArch 平台的虚拟网络设备初始化与数据传输。详细内容请参见：RocketOS 的网络设计

2.4 技术挑战与解决思路

RocketOS 的多架构适配工程面临着复杂的技术挑战，这些挑战不仅源于两种目标架构在硬件层面的根本性差异，更体现在如何在保持系统统一性的前提下充分发挥各架构的独特优势。系统通过创新的设计理念和工程实践，形成了一套完整的挑战应对体系。

代码统一性与架构优化的平衡挑战是多架构适配的核心矛盾。传统的解决方案往往在追求代码复用时牺牲性能，或在追求性能时增加维护复杂度。RocketOS 通过构建分层的抽象体系和基于 Rust cfg 属性的条件编译机制，实现了"一次编写，多重优化"的解决思路。系统将通用逻辑与架构特定实现在源码层面进行清晰分离，通过编译时的代码生成确保每个架构都获得最优化的执行路径，同时保持源码的统一管理和维护。

功能兼容性与 ABI 稳定性的维护挑战关系到系统的长期可维护性和生态兼容性。不同架构的调用约定、数据对齐要求和异常处理机制差异可能导致微妙的兼容性问题。系统构建了严格的 ABI 规范和兼容性测试体系，确保系统接口的稳定性和一致性。同时，系统设计了向前兼容的接口演进策略，为未来的架构扩展和功能增强预留了充分的扩展空间。

开发调试复杂度的管理挑战体现在多架构环境下的开发效率和问题定位难度。传统的单架构开发模式难以适应多架构并行开发的需求，调试和测试工作量成倍增长。RocketOS 通过构建统一的开发工具链和分层的调试支持体系，有效缓解了多架构开发的复杂性。系统采用了基于 Makefile 的统一构建接口，通过简单的 ARCH 参数切换就能完成不同架构的编译和测试工作。在调试支持方面，系统集成了 GDB 调试器的多架构支持，通过 gdbserver 和 gdbclient 目标提供了远程调试能力。

第 3 章　内存管理模块实现

3.1　内存布局

RocketOS 内核的内存管理模块负责实现虚拟内存管理、物理内存分配和地址空间隔离等功能。内存布局设计遵循 RISC-V 和 Loongarch 架构的规范，确保内核与用户空间的有效隔离和高效访问。

在 RISC-V 架构下，内核例程与用户例程均采用 Sv39 分页机制进行地址映射，内核与用户共享同一个页表结构（即用户映射与内核映射共存于同一页表, 通过 PTEFlags 的 U 位进行隔离）。

在 RISC-V 架构下，内核例程与用户例程均采用 Sv39 分页机制进行地址映射，内核与用户共享同一个页表结构（即用户映射与内核映射共存于同一页表, 通过 PTEFlags 的 U 位进行隔离）。

- 内核空间 位于虚拟地址的高地址段，内核镜像起始处至可用内存的最高物理地址，按照固定偏移（KERNEL_OFFSET）映射至虚拟地址空间，便于内核直接通过已知偏移进行物理地址访问。
- 用户空间 位于虚拟地址的低地址段，用户程序镜像从低地址起映射, 根据 ELF 文件加载。

该设计的一大优势在于：对于内核态代码而言，物理页帧号（PPN）与虚拟地址之间存在确定性偏移关系，内核访问物理页帧时无需额外查询页表，直接计算得到虚拟地址。这种机制极大地简化了页目录帧构建与写时复制（Copy-On-Write, COW）等内存管理操作。

在 Loongarch 架构下, 内核例程与用户例程处于不同的映射模式下：

- 内核地址空间 通过 CSR_DMW0 窗口，采用直接映射（Direct Map Window, DMW），虚拟地址高段直接映射到物理地址，无需经过页表翻译，访问高性能。
- 用户地址空间 通过分页机制映射（通过设置 PWCL 和 PWCH 寄存器, 使页表映射模式为 Sv39, 与 riscv 一致, 实现上层用户地址翻译的统一），虚拟地址低段需通过页表完成地址转换，支持动态映射、权限隔离等机制。

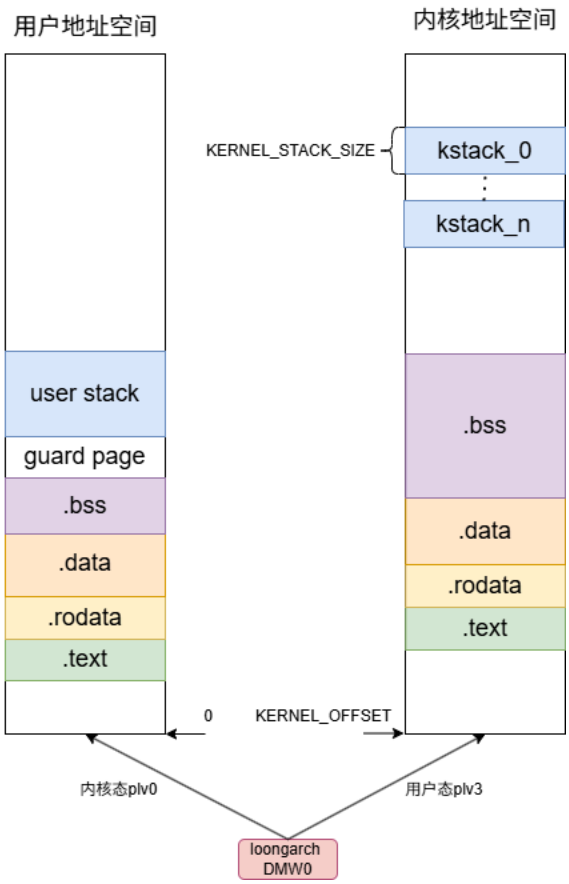

图 3-1 RocketOS 内存布局

3.2 堆分配器和用户页帧分配器

RocketOS 内核设计中，存在两个内存分配器，分别为堆分配器和页帧分配器，其职责范围和实现机制有所不同：

- 堆分配器 负责内核动态内存的分配，管理位于内核镜像区域的内存空间，主要用于 kmalloc、对象创建等需要动态大小内存的场景。堆分配器对 buddy_system_allocator 进行封装，基于 Buddy System 算法实现，支持不同大小块的合并与拆分，具备较好的内存利用率和碎片控制能力（如 代码 1 所示）。
- 页帧分配器 负责管理可用物理内存页帧，支持为页表映射、用户空间分配等场景按页分配物理内存。页帧分配器采用 stack-based bump allocator with recycling 策略，简洁高效，适合内核启动阶段以及高频页帧分配场景（如 代码 2 所示）。

两者的分配器设计均支持 RAII 资源管理，通过封装的跟踪器数据结构在对象生命周期结束时自动回收所占用的内存，简化了内存回收过程，提升了内核代码的健壮性和可靠性。

```
pub struct Heap<const ORDER: usize> {
    free_list: [linked_list::LinkedList; ORDER],
    user: usize,
    allocated: usize,
    total: usize,
}
```

代码 3-1　堆分配器结构

```
pub struct StackFrameAllocator {
    current: usize,
    end: usize,
    recycled: Vec<usize>,
}
```

代码 3-2　页帧分配器结构

3.3　虚拟内存数据结构

一个虚拟地址空间由一个 `MemorySet` 对象描述（如代码 3 所示）。`MemorySet` 对象包含了唯一的地址空间标识、区域映射表和页表帧数组等。区域集合 `areas` 是 RocketOS 虚拟内存系统的核心数据结构，描述了虚拟地址空间的区域。不同的区域可能有不同的映射方式，例如线性映射、共享内存、写时复制等。页表帧数组用于存储页表帧的跟踪器。

```rust
pub struct MemorySet {
    /// 堆顶
    pub brk: usize,
    /// 堆底
    pub heap_bottom: usize,
    /// mmap 的起始地址, 用于用户态 mmap
    pub mmap_start: usize,
    /// 页表
    pub page_table: PageTable,
    /// Elf, Stack, Heap, 匿名私有映射, 匿名共享映射, 文件私有/共享映射
    /// BTreeMap key 是 vpn_range 起始虚拟地址
    pub areas: BTreeMap<VirtPageNum, MapArea>,
    /// System V shared memory
    /// 映射: shm_start_address -> shmid
    pub addr2shmid: BTreeMap<usize, usize>,
}
```

代码 3-3　MemorySet 对象

3.4　虚拟地址空间区域实现

在 RocketOS 中，虚拟地址空间的区域管理由 MapArea 结构体承担(如[代码 4]所示)。它统一表示一段虚拟地址空间的区域，包括起始地址范围、权限、映射类型（如线性映射、按页分配、文件映射等），并提供了映射、取消映射、复制、扩展、分割等操作接口。

当前 RocketOS 支持的区域类型包括：

1. Linear：用于内核空间的线性映射；
2. Framed：按页分配物理页帧，适用于用户数据段等；
3. Stack：用户栈空间，通常按需分配；
4. Heap：用户堆空间，支持动态扩展；
5. Filebe：用户文件映射区域。

每个 MapArea 还包含一个页权限字段（MapPermission），支持读/写/执行/用户等权限标志，并可标记共享（S）或写时复制（COW）等属性。此外，MapArea 还支持关联文件映射及偏移量，便于实现 mmap 功能。

与整体地址空间解耦是 RocketOS 的一项重要设计。MapArea 并不持有页表根或页表帧等资源，而是通过传入的 PageTable 参数在映射和取消映射时完成操作。

页帧的具体分配由上层的 MemorySet 管理。这种设计使得一个 MapArea 可被复制、移动并重用于不同的地址空间实例，有利于实现如 fork 中的地址空间复制、写时复制、懒分配高级功能。

```rust
pub struct MapArea {
    pub vpn_range: VPNRange,
    pub map_perm: MapPermission,
    pub pages: BTreeMap<VirtPageNum, Arc<Page>>,
    pub map_type: MapType,
    /// 文件映射
    pub backend_file: Option<Arc<dyn FileOp>>,
    pub offset: usize,
    pub locked: bool, // 是否被锁定, 用于文件映射
}
```

代码 3-4　MapArea

3.4.1　线性映射区域

主要用于 内核地址空间 的直接映射，比如将物理内存映射到虚拟地址（通常是 KERNEL_BASE 起始）。

特点：

1. 虚拟页号（VPN）和物理页号（PPN）之间保持一个固定的偏移。(riscv 线性偏移，loongarch 直接映射)
2. 没有 Page 结构和物理页帧 Frame 的动态分配。
3. 权限固定、不可更改。
4. 映射时通过 `map_range_continuous` 一次性完成, 占用连续的物理空间。

3.4.2　独占物理页帧映射

用于用户进程的私有映射，如代码段、数据段、匿名映射等。

特点：

1. 每个页都由 `Page::new_framed` 分配一个新页帧。
2. 拥有独立物理页帧的写权限。
3. 可在 fork 时转变为 CopyOnWrite（写时复制）逻辑。

3.4.3　栈区域

表示用户进程的栈内存区域，支持懒分配和向下增长。

特点：

1. 栈空间按页分配，初始时不分配物理页帧 o
2. 当访问未映射页时触发 page fault，缺页异常处理器会分配一页新帧，并更新页表。
3. 当访问栈空间下的保护页时，，会将整个区域的 vpn_range.start 向下扩展一页，从而“向下增长”栈空间。
4. 为防止栈无限扩张，会检查其与前一个区域之间的页数是否大于设定的 栈保护间隙（STACK_GUARD_GAP_PAGES），若间隙不足则终止扩张并触发 SIGSEGV。

3.4.4 堆区域

用户堆映射，支持扩展, 堆顶为 `brk` ，堆底为 `heap_bottom` 。与 `Framed` 共享很多行为，但更强调向上的扩展性

3.4.5 文件映射区域

用户通过 mmap 创建的文件映射区域。

特点：

1. 访问时延迟加载页帧（除非 MAP_POPULATE 会预分配）由 `backend_file.get_pages()` 获取物理页帧。
2. 页帧内容可能会被写回文件。

3.4.6 共享内存区域

RocketOS 支持两种类型的共享内存：System V 共享内存和匿名共享内存。System V 共享内存由 `shmat` 系列系统调用产生，匿名共享内存由 `mmap` 系统调用产生。System V 共享内存通过 `ShmSegment` 对象管理，`SysVShm` 对象包含了共享内存的 id 和对应分配的页帧，如代码 5 所示。使用 Weak 而非 Arc 是为了将物理页的生命周期交由进程 attach/detach 控制，体现了“共享但非所有”的语义。

```rust
pub struct ShmSegment {
    pub id: ShmId,            // 段的元数据
    pub pages: Vec<Weak<Page>>, // 共享页帧列表（使用 Weak 管理引用）
}
```

代码 3-5　ShmSegment

3.5　缺页处理

RocketOS 支持两种可恢复的缺页异常处理机制：写时复制（Copy-on-Write, COW）和懒分配（Lazy Allocation）。

3.5.1　写时复制

写时复制机制用于进程创建后共享只读内存页（例如通过 fork），其页表项包含 COW 标志。当子进程试图写该页时：若该页只被一个进程引用，直接清除 COW 标志，赋予写权限；否则，分配一个新物理页，将原页内容复制到新页，并更新页表项。

算法 3-1: 写时复制页故障处理逻辑

input: va, page_table, area
output: Updated page table and area.pages

```
pte ← page_table.find_pte(va)
if pte.flags has COW then
    page ← area.pages[va]
    if refcount(page) == 1 then
        pte.flags.remove(COW)
        pte.flags.insert(W)
        page_table.update(va, pte)
    else
        new_page ← alloc_frame()
        memcpy(new_page, page)
        pte.flags.remove(COW)
        pte.flags.insert(W)
        page_table.update(va, new_page)
        area.pages[va] ← new_page
    end
else return SIGSEGV
```

3.5.2　懒分配

懒分配机制适用于匿名区域（如堆、栈）和文件映射区域（如 `mmap`），延迟到首次访问再分配实际物理页。

- 对于文件映射区域（Filebe）：
  - 若为共享或只读映射，直接通过 backend 提供物理页；
  - 若为私有写映射，执行写时复制操作；
- 对于匿名区域，如 Heap：
  - 缺页时分配页帧，并更新页表；
  - 为优化效率，会批量分配最多 4 页；
- 对于 Stack 区域：
  - 当访问的是栈底 VPN，会尝试向下增长一页，并更新 `vpn_range`。

算法 3-2: 懒分配页故障处理逻辑

```
    input: va, area, cause
    output: Updated page table and area.pages
1   vpn ← floor(va)
2   if area.type = Filebe then
3       offset ← area.offset + (vpn - area.start) * PAGE_SIZE
4       page ← area.backend.get_page(offset)
5       if page exists then
6           if cause = STORE and !area.is_shared then
7               new_page ← copy_frame(page)
8               map(va, new_page, Writable)
9               area.pages[vpn] ← new_page
10          else
11              map(va, page, area.perm)
12              area.pages[vpn] ← page
13          end
14      else return SIGBUS
15  else if area.type = Stack then
16      if vpn = area.start then
17          if prev_area exists and !guard_gap_ok then return SIGSEGV
18          area.start ← area.start - 1
19          self.areas.update_key(area)
20      new_page ← alloc_frame()
21      map(va, new_page, Writable)
22      area.pages[vpn] ← new_page
```

```
23  else
24      for i in [vpn .. vpn+4] do
25          if !area.pages.contains(i) then
26              page ← alloc_frame()
27              map(i, page, Writable)
28              area.pages[i] ← page
29          end
30  end
```

通过上述设计，RocketOS 能够在创建进程时实现写时复制，减少复制开销；同时允许了程序申请巨大的内存空间，而不会立即分配物理页帧，提高了内存分配的效率。

第 4 章　进程调度模块

4.1　内核同步有栈式模型

4.1.1　异步无栈式设计存在的问题

在分析往届优秀作品时，我们观察到大部分团队采用了异步无栈协程的调度架构。这种设计通过将 async 函数编译为状态机，实现了轻量级的任务调度机制。运行时系统仅需根据保存的状态信息推进执行，在代码编写层面具有显著优势。然而，从操作系统设计角度审视，这种架构引入了几个值得深入探讨的关键问题。

1. “虚拟栈帧”重建与性能问题

异步架构从根本上改变了传统的函数执行模型。在传统栈式执行中，函数调用关系通过栈帧自然维护，而异步中的执行上下文则完全依赖堆分配的 Future 对象。当任务在 await 点挂起并返回 Poll::Pending 时，整个调用栈被销毁，所有执行状态必须序列化保存至堆内存中。任务恢复时，调度器通过重新 poll 相应对象来重建执行环境，这个过程本质上是在堆结构中重构"虚拟栈帧"链。由于 async 函数间的相互 await 调用会形成深层嵌套的状态结构，单次任务调度往往涉及十余层的 poll 函数链式调用。每层调用都需要进行状态检查、分支判断和上下文切换，这些看似微小的开销在高并发场景下会产生可观的性能影响。

2. 内存管理的复杂性

异步任务缺乏真实栈空间支持，所有局部变量和中间状态都必须在堆上显式分配和管理。这种设计带来两个层面的挑战：首先是内存使用模式的改变，大量并发任务的状态结构同时存在于堆内存中，可能导致内存使用量快速增长；其次是生命周期管理的复杂化，这些堆分配结构的回收时机难以精确控制，在系统负载较高或开发者使用不当的情况下，容易出现悬挂 Future 对象或内存泄漏问题。

4.1.2　为何选择同步有栈式设计

同步有栈式设计最重要的优势在于其自然而直观的内存管理模式。所有的局部变量、函数参数和中间状态都自动分配在栈上，这些内存资源的生命周期与函数调用的生命周期完全一致，当函数返回时所有相关内存会被自动回收。这种栈式内存管理完全避免了异步无栈式设计中大量堆分配状态结构带来的内存碎片化问题，也消除了复杂的手动内存生命周期管理需求。更为重要的是，栈内存的分配和释放是常数时间复杂度的简单指针操作，相比异步模式中需要进行的堆内存分配器调用，具有显著的性能优势。

同步有栈式架构具有高度可预测的性能特征。每次函数调用的开销是固定的，主要包括栈帧的建立、参数传递和返回值处理，这些操作的时间复杂度都是常数级别的。而异步无栈式设计中的每次状态推进都可能涉及复杂的状态检查、分支判断和多层嵌套的 poll 调用链，其性能开销随着异步调用的嵌套深度呈现出不可预测的增长趋势。在需要严格性能保证的系统中，有栈式设计的确定性执行时间是一个重要优势。

4.2 任务调度设计

4.2.1 任务切换设计

RocketOS 的任务切换机制是系统核心组件中的杰出设计，与 trap 处理中 tp 寄存器管理构成了紧密集成的架构体系。在任务切换的状态保存阶段，`__switch` 函数实现了一种高效的设计策略,即通过直接在当前任务的内核栈上内嵌 TaskContext 存储空间的方式来保存完整的寄存器状态。TaskContext 中涵盖了关键的寄存器组合：返回地址寄存器 ra、线程指针寄存器 tp，以及 s0 至 s11 共计 12 个调用者保存寄存器，确保了任务切换过程中上下文信息的完整性和一致性。

算法 4-1: 任务切换算法

```
Input: a0 ← next_task_kernel_stack
Output: null

# 保存 taskContext 到内核栈
1: sp ← sp - 16 × 8
2: mem[sp + 0] ← ra
```

```
    mem[sp + 8] ⟵ tp
3:  for i ∈ {0..11} do
      mem[sp + (2 + i) × 8] ⟵ x[Callee[i]]  # Callee ={s0~s11}
4:  mem[sp + 14 × 8] ⟵ satp/pgdl # loongarch 对应 csr 为 pgdl
5:  tp ⟵ sp

# 从内核栈恢复 taskContext
6:  ra ⟵ mem[a0 + 0]
    tp ⟵ mem[a0 + 8]
7:  for i ∈ {0..11} do
      x[Callee[i]] ⟵ mem[a0 + (2 + i) × 8]
8:  satp/pgdl ⟵ mem[a0 + 14 × 8]
    sfence.vma/invtlb  # 清除 TLB
9:  a0 ⟵ a0 + 16 × 8
    sp ⟵ a0
    return/jr
```

在 RocketOS 任务切换过程中，内核栈的定位机制是通过 Task 结构体（任务控制块 TCB）的首字段 kstack 实现的。该 kstack 字段作为记录指针，标记着对应任务内核栈的栈顶位置。

相较于传统的任务切换实现方案，RocketOS 通过直接操作内核栈和 Task 结构中 kstack 字段的创新设计，实现了任务上下文保存与恢复的显著性能优化。传统方案往往需要在堆上分配复杂的状态结构与加锁来维护任务上下文，这不仅增加了内存分配的开销，还引入了额外的内存管理复杂性。而 RocketOS 的方案通过将 TaskContext 直接嵌入到内核栈中，利用栈的天然 LIFO 特性和局部性原理，不仅消除了动态内存分配的开销，还提高了缓存命中率，从而在任务切换这一高频操作上获得了显著的性能提升。

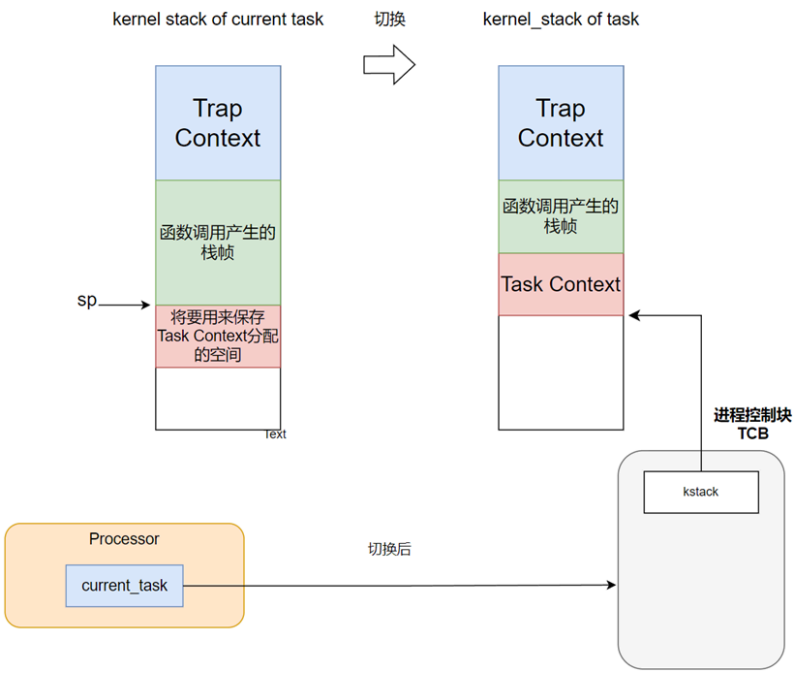

图 4-1 任务切换示意图

4.2.2 任务调度策略

RocketOS 的任务调度系统采用了简洁而高效的 FIFO（First In First Out）调度策略。Scheduler 结构使用 VecDeque 作为就绪队列的底层数据结构，这种选择既保证了队列操作的高效性，又提供了双端队列的灵活性，使得任务的入队和出队操作都能在常数时间内完成。

```
pub struct Scheduler {
  ready_queue: VecDeque<Arc<Task>>,
}
```

代码 4-1 调度队列设计

在 RocketOS 中，任务切换机制通过 yield 和 schedule 两个核心函数实现了不同场景下的调度需求，两个函数在功能定位上存在本质区别，主要体现在对当前任务状态的不同处理方式上，这种差异化设计使得系统能够在保证调度效率的同时，准确响应各种任务状态变化。

yield 函数设计用于处理协作式任务切换场景，主要服务于那些主动放弃 CPU 使用权但仍需要继续执行的任务。当任务调用 yield 时，系统会将当前任务重新加入到就绪队列的末尾。

相对而言，schedule 函数承担着更为复杂的调度职责，主要处理任务生命周期管理和状态转换的关键时刻。当任务因为等待 I/O 操作完成、等待锁资源释放、或者因为其他阻塞条件而无法继续执行时，schedule 函数会被调用来寻找下一个可执行的任务。在这种情况下，当前任务不会被重新加入到就绪队列中，而是根据具体的阻塞原因被转移到相应的等待队列或直接标记为阻塞状态。同时，当任务正常终止或异常退出时，schedule 函数也承担了将任务移除调度队列的工作，从而确保系统资源得到正确释放。

4.2.3 任务阻塞策略

在 RocketOS 的阻塞机制设计中，任务阻塞是由一个全局性的阻塞管理器来实现的，其中的底层实现同样是基于 VecDeque 数据结构。这个设计允许系统在处理任务阻塞时，能够高效地管理和调度等待资源的任务。每当一个任务因为等待某个资源而无法继续执行时，它会被从就绪队列中移除，并加入到全局的阻塞队列中。

```
pub struct WaitManager {
    pub wait_queue: Mutex<WaitQueue>,
}
pub struct WaitQueue {
    queue: VecDeque<Arc<Task>>,
}
```

代码 4-2 阻塞队列设计

阻塞任务的唤醒机制同样构成了整个任务调度系统的重要组成部分，它确保了被阻塞的任务能够在适当的时机重新获得执行机会。这种唤醒机制根据触发条件的不同可以分为三种主要类型，每种类型都对应着不同的系统事件和应用场景。

1. 正常唤醒

   正常唤醒是最常见的唤醒方式，它发生在任务等待的资源或条件变为可用时。当任务因为等待某个特定资源而进入阻塞状态后，系统会持续监控该资源的状态变化。一旦资源变为可用，系统会立即触发唤醒操作，将对应的任务从阻塞队列中移除并重新加入就绪队列。典型的正常唤醒场景包括文件 I/O 操作的完成等。这种唤醒方式保证任务只有在真正需要执行时才会被唤醒，同时也减轻了任务调度器的压力，极大地提高了系统的运行效率。

2. 信号中断唤醒

信号中断唤醒机制则提供了一种异步通信的手段，允许外部事件或其他任务通过发送信号来中断正在等待的任务。这种唤醒方式的特点是具有较高的优先级，能够打断任务的正常等待流程。当任务接收到信号时，无论其等待的原始条件是否满足，都会被强制唤醒并进入信号处理流程。信号唤醒的应用场景非常广泛，包括进程间通信中的自定义信号，定时器到期信号，以及系统异常信号等。这种机制为系统提供了强大的灵活性，使得任务能够响应各种异步事件，实现复杂的控制逻辑和错误处理机制。

算法 4-2: 常规阻塞算法

```
Input: null
Output: If interrupted, returns -1,
        If wakeup normally returns 0.

1: task.state ← INTERRUPTIBLE
2: WAIT_MANAGER.add(task)
3: schedule() # 执行任务切换
4: task.state ← READY
5: if task.is_interrupted == true then
     return -1   # 表示阻塞被信号打断
   end if
6: return 0  # 表示正常唤醒
```

3. 超时唤醒

超时唤醒机制则是为了防止任务无限期等待而设计的保护机制。在实际的系统运行中，某些资源可能长时间不可用，或者某些条件可能永远不会满足，如果没有超时机制，任务将会永远阻塞下去，导致系统资源的浪费和潜在的死锁问题。超时唤醒通过设置一个预定的时间限制，当等待时间超过这个限制时，系统会触发超时回调函数，向对应任务发送定时器到期信号来自动唤醒任务，让任务有机会重新评估情况或采取替代措施。

算法 4-3: 超时阻塞算法

```
Input: dur, clock_id
Output: If interrupted by a signal, returns -1,
        If timeout, returns -2,
        If wakeup normally returns 0.

1: task.state ← INTERRUPTIBLE
2: deadline ← set_wait_alarm(dur, tid, clock_id)
3: WAIT_MANAGER.add(task)
4: schedule() # 执行任务切换
5: task.state ← READY
6: clear_wait_alarm(tid)
7: if task.interrupted == true then
     return -1  # 表示阻塞被信号打断
   end if
8: if current_time() ≥ deadline then
     return -2  # 表示阻塞超时
   end if
9: return 0  # 正常唤醒
```

无论采用哪种唤醒方式，被唤醒的任务都会经历一个标准的状态转换过程。首先，任务会从阻塞队列中被移除，任务的状态会从阻塞状态转换为就绪状态，这标志着任务已经具备了再次执行的条件。接下来，任务会被重新加入到就绪队列中，等待调度器的下一次调度。当任务再度被执行时，通过检查函数调用的返回值，任务能够准确判断自己究竟是因为正常资源可用、信号中断还是超时而被唤醒的。这种判断机制的存在使得任务能够采取相应的后续行动：如果是正常唤醒，任务可以继续执行原本被阻塞的操作，比如读取已经就绪的文件数据或获取已经释放的锁资源；如果是信号唤醒，任务可能需要先处理信号相关的逻辑，然后决定是否重新尝试之前的操作；如果是超时唤醒，任务则需要评估是否应该放弃当前操作，或者调整策略后重新尝试。

4.3 任务结构设计

4.3.1 任务控制块设计

进程作为操作系统中资源管理的基本抽象单元，拥有独立的虚拟地址空间、文件描述符表以及其他系统资源的所有权。线程则代表了处理器调度执行的最小单元，它们在共享宿主进程资源的基础上维护各自独立的执行状态和调用栈。

在关于进程与线程的问题设计上，RocketOS 采取的是与 Linux 相类似的设计思想，Linux 内核通过 sys_clone 系统调用的灵活标志位机制，实现了对不同类型任务创建的统一管理。通过传递不同的 flags 参数组合，同一个系统调用既可以创建拥有完全独立资源的传统进程，也可以生成与父任务共享特定资源的轻量级线程，这种设计体现了"机制与策略分离"的核心思想。基于这一认知，我们采用了统一的 Task 结构体设计，通过 Arc 与 Mutex 的组合特性实现了线程间的资源共享问题，Task 结构设计如下：

```rust
pub struct Task {
  kstack: KernelStack, // 内核栈
  tid: RwLock<TidHandle>,                    // 线程 id
  tgid: AtomicUsize,                         // 线程组 id
  tid_address: Mutex<TidAddress>,            // 线程 id 地址
  status: Mutex<TaskStatus>,                 // 任务状态
  time_stat: SyncUnsafeCell<TimeStat>,       // 任务时间统计
  parent: Arc<Mutex<Option<Weak<Task>>>>,    // 父任务
  children: Arc<Mutex<BTreeMap<Tid, Arc<Task>>>>, // 子任务
  thread_group: Arc<Mutex<ThreadGroup>>,     // 线程组
  exit_code: AtomicI32,                      // 退出码
  exe_path: Arc<RwLock<String>>,             // 执行路径
  memory_set: RwLock<Arc<RwLock<MemorySet>>>,  // 地址空间
  robust_list_head: AtomicUsize,             // 稳健性链表
  fd_table: Mutex<Arc<FdTable>>,             // 文件描述符表
  root: Arc<Mutex<Arc<Path>>>,               // 根路径
  pwd: Arc<Mutex<Arc<Path>>>,                // 当前路径
  umask: AtomicU16,                          // 文件权限掩码
  sig_pending: Mutex<SigPending>,            // 待处理信号
  sig_handler: Arc<Mutex<SigHandler>>,       // 信号处理函数
  sig_stack: Mutex<Option<SignalStack>>,     // 额外信号栈
  itimerval: Arc<RwLock<[ITimerVal; 3]>>,    // 定时器
  rlimit: Arc<RwLock<[RLimit; 16]>>,         // 资源限制
  cpu_mask: Mutex<CpuMask>,                  // CPU 掩码
  pgid: AtomicUsize,                         // 进程组 id
  uid: AtomicU32,                            // 用户 id
  euid: AtomicU32,                           // 有效用户 id
  suid: AtomicU32,                           // 保存用户 id
  fsuid: AtomicU32,                          // 文件系统用户 id
  gid: AtomicU32,                            // 组 id
  egid: AtomicU32,                           // 有效组 id
  sgid: AtomicU32,                           // 保存组 id
  fsgid: AtomicU32,                          // 文件系统组 id
  sup_groups: RwLock<Vec<u32>>,              // 附加组列表
}
```

代码 4-3　任务控制块

在 RocketOS 的 Task 结构设计中，身份标识与层次关系通过三层标识体系得以实现，这一体系以 tid、tgid 和 pgid 为核心，确保了系统在支持现代多线程编程模型的同时，依然兼容传统的进程间关系。tid 作为线程的唯一标识符，能够精确区分系统中每一个独立的线程执行单元；tgid 则用于标识线程组，相当于传统意义上的进程概

念，将属于同一进程的多个线程关联起来；而 pgid 则负责进程组的管理，用于协调一组相关进程的行为，例如信号传递或作业控制。此外，thread_group 字段被设计用于管理同一进程内的所有线程，通过高效的线程组织方式，支持多线程任务的高效协同。

```
pub struct ThreadGroup {
  member: BTreeMap<Tid, Weak<Task>>,
}
```

代码 4-4　线程组结构

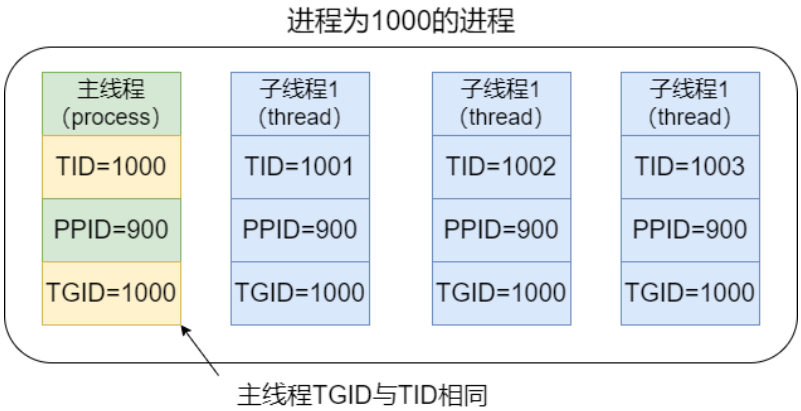

图 4-2　线程组结构图

在进程间继承关系方面，Task 结构通过 parent 和 children 字段维护了一棵完整的进程树，其中 parent 指向父进程，children 则记录子进程集合。为了避免循环引用问题，RocketOS 采用了 Rust 语言的 Arc（原子引用计数）和 Weak 智能指针，通过引用计数的动态管理，确保进程树结构的稳定性和内存安全。

内存管理作为 Task 结构的核心组成部分，体现了 RocketOS 在高效性和安全性上的深思熟虑。memory_set 字段采用了双重 RwLock（读写锁）的嵌套结构，外层 RwLock 负责控制地址空间的切换，例如在 exec 系统调用时对整个地址空间的重新配置；内层 RwLock 则专注于保护具体的内存映射操作，例如页面分配或释放。这种嵌套锁机制允许多个线程在同一地址空间内安全地并发操作，同时支持写时复制（Copy-on-Write）等高级内存管理特性，从而在性能与内存效率之间取得平衡。此外，每个线程通过 kstack 字段维护独立的内核栈，这一设计确保了线程在内核态执行时的隔离性和安全性，避免了因共享内核栈可能导致的竞争条件或数据损坏问题。

在用户权限管理方面，RocketOS 沿用了 Linux 的完整权限模型，通过实际用户 ID（uid）、有效用户 ID（euid）、保存用户 ID（suid）以及文件系统用户 ID（fsuid）等字段，结合对应的组权限字段（gid、egid、sgid、fsgid），实现了细粒度的权限控制。这种设计支持了复杂的权限提升和降级操作，例如在执行 setuid 程序时，能够根据上下文动态调整权限以确保安全性。sup_groups 字段进一步扩展了权限管理功能，允许任务关联多个附加组，从而支持更灵活的权限分配策略，例如在需要跨多个用户组协作的场景下，提供精确的权限控制。

在资源管理和任务调度方面，rlimit 数组用于管理任务对系统资源的访问限制，例如文件描述符数量、堆栈大小或内存使用量，从而防止资源滥用并保证系统的稳定性。itimerval 字段支持定时器功能，允许任务设置周期性或单次触发的定时器，用于实现精确的时间管理或事件触发。cpu_mask 字段则提供了对任务 CPU 亲和性的控制，允许系统将任务绑定到特定的 CPU 核心上运行，以优化性能或降低调度开销。此外，time_stat 字段记录了任务的运行时间统计信息，包括用户态和内核态的执行时间、上下文切换次数等。

4.3.2 任务状态设计

RocketOS 的任务状态设计采用了经典的五状态模型，通过 TaskStatus 枚举类型精确定义了任务在其生命周期中可能处于的各种状态，系统中的每个任务都必然处于这五种状态中的一种:

- **就绪（Ready）**：任务已准备好运行，等待调度器分配 CPU 时间片。
- **运行（Running）**：任务正在 CPU 上执行。
- **可中断阻塞（Interruptable）**：任务因等待某些资源而阻塞，且可以被信号中断。
- **不可中断阻塞（UnInterruptable）**：任务因某些资源而阻塞，但不可被信号中断。
- **僵尸（Zombie）**：任务已终止，但其父进程尚未调用 wait 系统调用获取其退出状态，仍保留在系统中以供父进程查询。

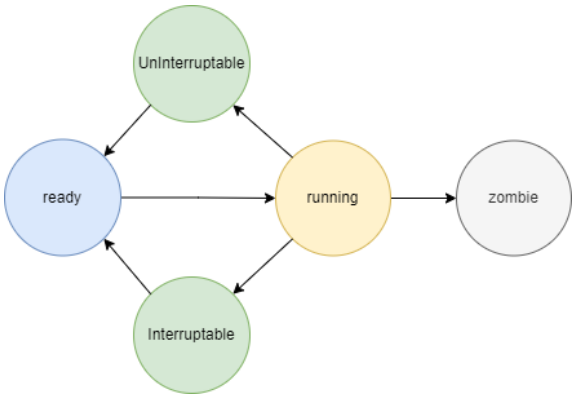

图 4-3 任务状态切换图

从 Running 状态出发存在多种转换路径。当任务主动调用 yield 让权或时间片耗尽时，会重新回到 Ready 状态等待下次调度。如果任务因为 waitpid、文件 I/O 等操作需要等待资源时，会根据等待类型进入相应的阻塞状态：对于可以被信号中断的等待操作（如 waitpid），任务进入 Interruptable 状态；对于关键的系统操作（如磁盘写入），任务进入 UnInterruptable 状态以确保操作的原子性。阻塞状态的唤醒机制体现了不同阻塞类型的特性差异。

Interruptable 状态的任务可以通过两种方式返回 Ready 状态：等待的资源变为可用，或者接收到信号中断。而 UnInterruptable 状态的任务只能等待特定的资源条件满足才能返回 Ready 状态，这种设计保证了系统关键操作不会被意外中断。

当任务执行完毕或异常终止时，会从 Running 状态直接转换到 Zombie 状态。Zombie 任务已经释放了大部分系统资源，只保留基本的进程控制信息等待父进程收集。当父进程调用 wait 系列系统调用获取子进程的退出状态后，Zombie 任务才会被彻底清理，完成整个生命周期。这种设计确保了父子进程间退出状态信息的可靠传递，同时避免了过早资源回收可能导致的信息丢失。

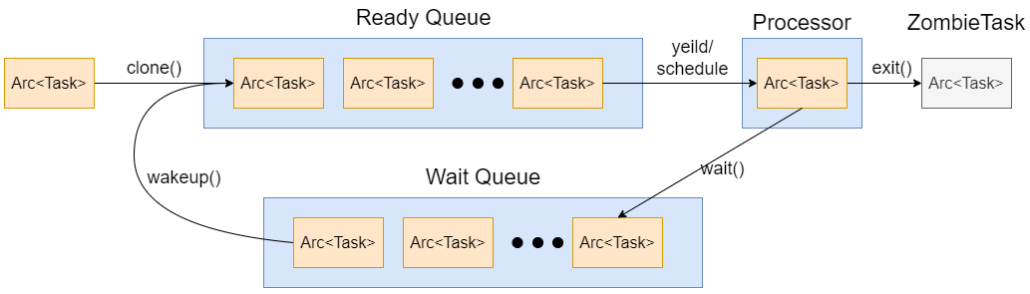

图 4-4 任务流程图

4.4 中断机制设计

中断机制是操作系统中用于处理硬件或软件事件的响应机制，允许系统在特定事件发生时暂停当前任务，快速切换到中断处理程序以执行紧急或高优先级操作。中断通常分为硬件中断（如 I/O 设备信号、时钟中断）和软件中断（如系统调用、异常）。

在 RocketOS 中，从用户态切换到内核态的场景主要包括三种：

- **系统调用**，即用户程序主动请求内核提供的服务；
- **中断**，由硬件设备触发，需内核进行处理；
- **异常**，当用户程序执行非法操作时发生。

4.4.1 用户态 ⟶ 内核态切换

每次从用户态陷入内核态，系统会跳转到由 __trap_from_user 标签定义的汇编代码段。这段代码负责保存用户态的运行上下文，并为内核态的执行环境做好准备。随后，trap_handler 函数会根据具体的陷阱类型（如系统调用、中断或异常）进行针对性处理，确保系统能够高效、正确地响应不同的事件。

算法 4-4: trap_from_user 算法

Input: user_context
Output: processed_context

1: sp ⟵ sscratch/CSR_SAVE0 ⟵ sp # 保存原用户态栈指针
2: sp ⟵ sp - 36 × 8
3: for i ∈ SaveSet do
    mem[sp + i × 8] ⟵ x[i] #SaveSet = {x0~x31(除 sp)}
4: mem[sp + 32 × 8] ⟵ sstatus/CSR_PRMD
   mem[sp + 33 × 8] ⟵ sepc/CSR_ERA
   mem[sp + 2 × 8] ⟵ sscratch/CSR_SAVE0
5: mem[sp + 34 × 8] ⟵ a0 # 保存用户 a0 参数到 last_a0
6: tp ⟵ mem[sp + 35 × 8] # 加载内核 tp
7: stvec/CSR_EENTRY ⟵ &__trap_from_user

```
8:  call/bl trap_handler(a0)
9:  jump __return_to_user
```

4.4.2 TrapContext 设计

在 RocketOS 的内核态用户态切换中，TrapContext 是一个关键的数据结构，用于保存从用户态切换到内核态，以及从内核态切换回用户态时需要恢复的上下文信息。这个结构体的设计保证了用户态和内核态之间的切换能够正确地进行，不会丢失任何重要的状态信息。

在 trap 机制的核心设计中，我们将 trap_context 结构始终固定保存在内核栈的顶端位置，这种设计选择带来了显著的架构优势。通过这种固定位置的布局，当系统通过 CSR 寄存器完成用户栈与内核栈之间的快速切换后，我们能够以确定的偏移量直接访问所有保存的上下文信息。

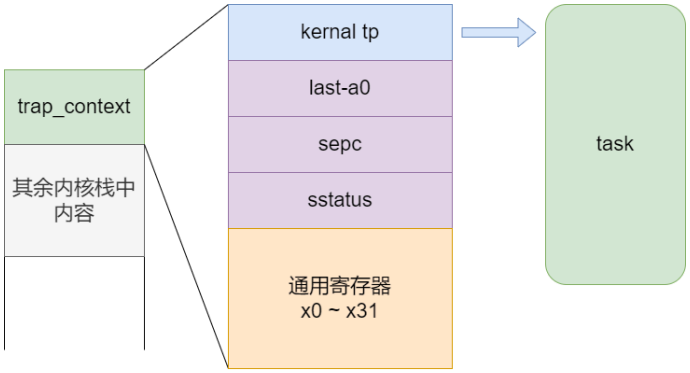

图 4-5 trap_context 结构示意图

如图 7 所示，trap_context 的结构设计，特别是 tp 寄存器的处理策略，主要是针对前述任务切换机制进行的专门优化。RocketOS 在 TrapContext 结构体中引入了 kernel_tp 字段这一创新设计，虽然会产生轻微的存储开销，但实现了任务切换流程对传统堆内存分配和锁竞争机制的彻底规避。这一权衡设计的核心优势在于将任务上下文的访问模式从堆访问重构为栈访问，有效消除了动态内存分配、互斥锁获取与释放等高延迟操作，其性能收益远超额外存储开销所带来的成本，实现了整体系统效率的显著提升。

4.4.3 内核态 ⟶ 用户态切换

当内核完成相应的 trap 处理后，系统需要从内核态安全地返回到用户态，这个过程由 __restore_to_user 标签定义的汇编代码段精确控制。这段代码承担着恢复用户态执行环境的关键职责，它会从之前保存的 TrapContext 中逐一恢复用户态的寄存器状态，包括通用寄存器、程序计数器以及 CSR 寄存器。

算法 4-5: return_to_user 算法

```
Input: processed_context
Output: null

1: stvec/CSR_EENTRY ← &__trap_from_user
2:  t0 ← mem[sp + 32 × 8]
    t1 ← mem[sp + 33 × 8]
    t2 ← mem[sp + 2  × 8]
    sstatus/CSR_PRMD  ← t0
    sepc/CSR_ERA      ← t1
    sscratch/CSR_SAVE0 ← t2
3:  for n ∈ SaveSet do
      x[n] ← mem[sp + n × 8]  #SaveSet = {x0~x31(除 tp)}
4:  mem[sp + 35 × 8] ← tp # 保存内核 tp
5:  tp ← mem[sp + 4 × 8]
6:  sp ← sscratch/CSR_SAVE0  ← sp
7:  sret
```

4.4.4 Riscv 与 LoongArch 的中断兼容性设计

为了实现跨架构兼容性，系统针对 RISC-V 和 LoongArch 架构分别实现了专门的 trap_from_user 和 return_to_user 汇编代码段。在 RISC-V 架构实现中，trap_from_user 段使用 csrr 指令读取 sstatus 和 sepc 等 CSR 寄存器的值，并通过统一的寄存器编号约定（x0 到 x31）来保存通用寄存器状态。相对应的 return_to_user 段则使用 csrw 指令恢复这些 CSR 寄存器，最终通过 sret 指令完成从内核态到用户态的特权级别切换。

LoongArch 架构的实现则采用了该架构特有的指令集和寄存器约定。在 LoongArch 版本的 trap_from_user 中，系统使用 csrrd 指令读取 PRMD（替代 sstatus 的功能）和 ERA（替代 sepc 的功能）寄存器，通用寄存器的保存遵循 r0 到 r31 的

编号约定。return_to_user 的 LoongArch 实现使用 csrwr 指令恢复控制寄存器状态，并通过 ertn 指令执行特权级别的返回操作。

同时，TrapContext 结构的设计也进行了跨架构兼容性的考虑。通用寄存器数组 x[32]统一了 RISC-V 和 LoongArch 两种架构的寄存器保存方式，尽管两种架构在寄存器编号约定上存在差异（RISC-V 中 x[4]为 tp 寄存器，x[10]为 a0 寄存器，而 LoongArch 中 r[2]为 tp 寄存器，r[4]为 a0 寄存器），但通过统一的数组索引机制实现了代码的架构无关性。而针对架构的不同，分别使用了 sepc 和 ERA 寄存器（RISC-V）或 PRMD 寄存器（LoongArch）来保存程序计数器和处理器状态寄存器的值。

这种架构特定的实现确保了在不同硬件平台上良好的兼容性，同时通过条件编译机制使得同一份内核源代码能够根据目标架构自动选择合适的汇编实现。两种架构实现的共同特点是都严格遵循了 TrapContext 结构的统一布局，使得上层的 trap_handler 函数能够以架构无关的方式处理各种 trap 事件，真正实现了"一次编写，多架构运行"的设计目标。

第 5 章　文件系统模块

RocketOS 的文件系统模块构建在一套抽象统一、模块解耦的 VFS 框架之上，支持挂载多个后端文件系统（如 Ext4、Tmpfs、Devfs 等），并通过统一的 Dentry、Inode、File 抽象，协调路径解析、权限控制与文件访问等操作流程。下图展示了 RocketOS 文件系统在内核中的总体架构，体现了从系统调用到具体后端文件系统的协作流程。

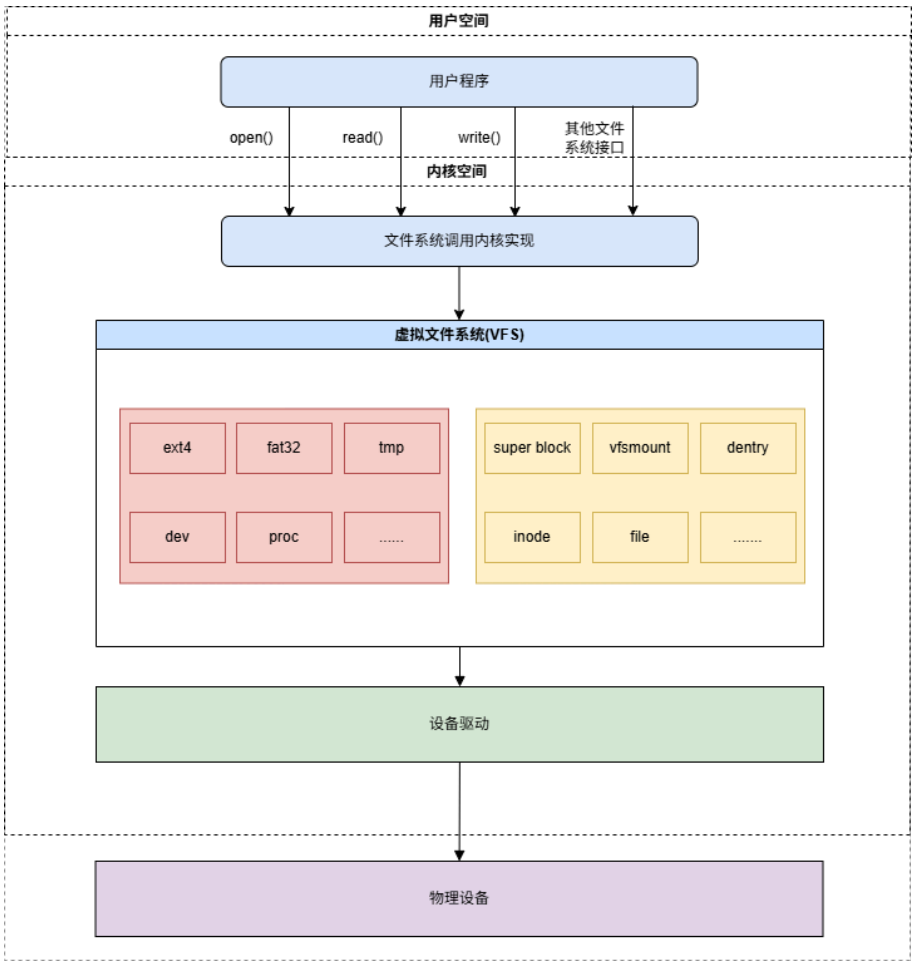

图 5-1　文件系统总体框架

5.1　挂载子系统

RocketOS 的挂载子系统由三类核心结构组成：VfsMount 表示一次具体的文件系统挂载，Mount 表示一个挂载点的元数据，MountTree 维护所有挂载关系的全局视图。三者协同构成一棵挂载树，实现文件系统命名空间的统一视图。

VfsMount 封装了文件系统挂载的核心信息，包括挂载点的根目录、挂载的文件系统（超级块）以及挂载标志。Mount 则表示一个具体的挂载点及其拓扑位置，mountpoint 表示挂载操作发生的目录项，vfs_mount 指向被挂载的文件系统实例，parent 指向父挂载点，children 存储子挂载点。MountTree 则是一个全局的挂载点列表，维护所有 Mount 对象并提供必要的插入与查询接口。该结点通过全局静态变量 MOUNT_TREE 暴露，具备线程安全性。

```
pub struct VfsMount {
    root: Arc<Dentry>,          // 挂载点的根目录
    fs: Arc<dyn FileSystem>,    // 挂载的文件系统(超级块)
    flags: i32,                 // 挂载标志
}
pub struct Mount {
    mountpoint: Arc<Dentry>,    // 挂载点目录
    vfs_mount : Arc<VfsMount>,  // 挂载的文件系统
    parent: Option<Weak<Mount>>, // 父挂载点
    children: Vec<Arc<Mount>>,  // 子挂载点
}
struct MountTree {
    mount_table: Vec<Arc<Mount>>, // 挂载点列表
}
```

代码 5-1　挂载命名空间和挂载树

5.1.1　根文件系统初始化流程

系统启动时，通过 do_ext4_mount 函数挂载根文件系统。该函数执行以下步骤：

1. 打开指定的块设备，并初始化 Ext4FileSystem 实例；
2. 从根块组构造根目录对应的 Inode；
3. 创建根目录的 Dentry，设置自身为其父项以形成闭环；
4. 构建 VfsMount 和对应的 Mount；
5. 将根挂载点加入全局 MountTree；
6. 初始化 /dev, /proc, /tmp 等虚拟文件系统。

5.2 VFS 模块设计

RocketOS 的 VFS 模块设计借鉴了 Linux VFS 架构，采用 Rust 的类型系统与 trait 机制实现了高效、安全的文件系统抽象。VFS 层的核心组件包括 Inode、Dentry、File、MountTree 等，它们通过 trait 定义了统一的操作接口，支持多种后端文件系统的透明接入。

VFS 层的设计目标是提供一个统一的文件命名空间，支持跨文件系统的路径解析、权限控制与文件访问，同时确保高性能与安全性。RocketOS 的 VFS 设计充分利用 Rust 的所有权与生命周期管理特性，避免了传统 C/C++ 文件系统实现中的许多常见错误，如内存泄漏、数据竞争等。

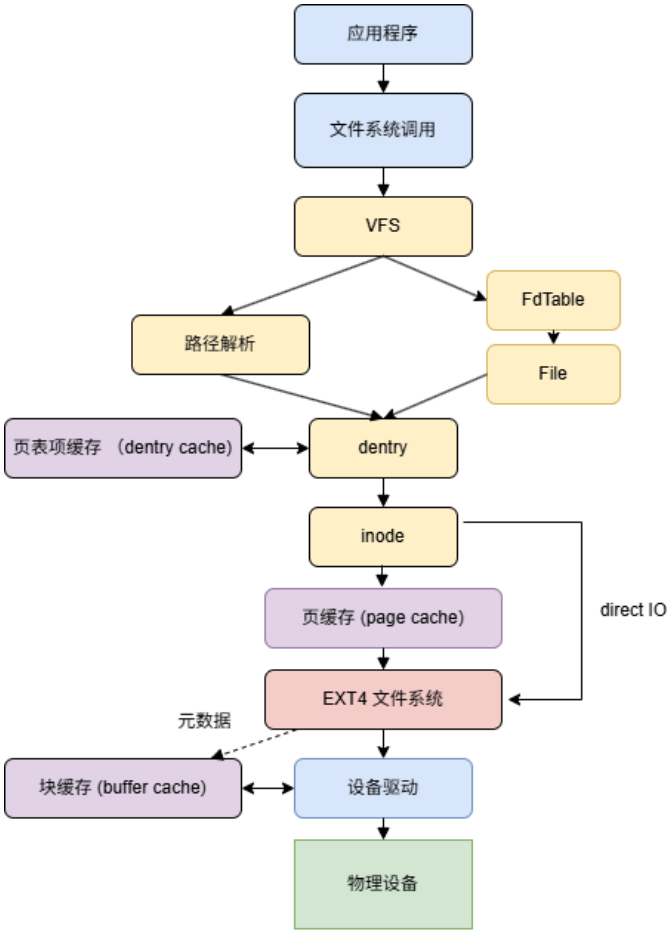

图 5-2 VFS 层结构与核心组件关系

如上图所示，RocketOS 的 VFS 层以 Inode、Dentry、File 为核心抽象，通过 MountTree 管理挂载关系，形成统一的文件命名空间。该设计借鉴了 Linux VFS 架

构，同时引入 Rust 类型系统以强化安全性与并发管理，支持多个后端文件系统的透明接入。

5.3 Inode 与页缓存

在 RocketOS 中，`inode` 抽象的核心体现在 `InodeOp` trait 接口之中。该接口定义了文件与目录的基本操作语义，包括读写、截断、页缓存交互、目录项查找与创建、符号链接、权限修改、文件状态查询等。每个具体的文件系统（如 Ext4）通过实现 `InodeOp` 接口，使其能以统一方式与 VFS 层交互，从而实现跨文件系统的透明访问能力。

为了支持运行时的类型动态调度与多文件系统扩展，`InodeOp` 被设计为 trait 对象（`dyn InodeOp`），并通过 `Arc<dyn InodeOp>` 进行引用计数与共享，确保线程安全与生命周期管理。

```
pub trait InodeOp: Any + Send + Sync {
  fn read(&self, offset: usize, buf: &mut [u8]) -> usize;
  fn write(&self, offset: usize, buf: &[u8]) -> usize;
  fn get_page(&self, page_index: usize) -> Option<Arc<Page>>;
  fn truncate(&self, size: usize) -> SyscallRet;
  fn fsync(&self) -> SyscallRet;
  fn lookup(&self, name: &str, parent: Arc<Dentry>) -> Arc<Dentry>;
  fn create(&self, dentry: Arc<Dentry>, mode: u16);
  ...
}
```

代码 5-2 InodeOp

5.3.1 与其他组件的协作

1. 页缓存与块映射

- InodeOp 提供了 get_page 和 lookup_extent 等接口，用于访问页缓存与磁盘块映射逻辑。该设计体现了块设备抽象与页缓存机制的耦合：
  - get_page: 返回指定页索引对应的缓存页，如无缓存则从磁盘加载或分配新页。
  - lookup_extent: 查找给定页是否被分配块，并返回物理块号区间。
  - write_dio: 支持绕过页缓存的直接写入（Direct IO）。

2. 目录与路径

- lookup, create, unlink, rename, mkdir, symlink, link 提供了标准 POSIX 文件系统语义的目录操作。lookup 的语义约定如下：
  - 若目标存在，则返回包含有效 inode 的 dentry；
  - 若目标不存在，则返回负目录项（inode = None）；
  - 上层调用者负责缓存管理（如放入 dentry cache）。

3. 临时与特殊文件支持
   - tmpfile: 创建匿名临时 inode，不注册 dentry，适用于 tmpfs 等场景。
   - mknod: 创建字符/块设备文件，支持设置设备号（dev_t）。
4. 属性访问与修改
   - 类型与权限：get_mode, set_mode, set_perm
   - UID/GID：get_uid, set_uid, ...
   - 时间戳：get_atime, set_mtime, ...
   - 文件大小与页数：get_size, get_resident_page_count

5.3.2 具体实现

以 Ext4 为例，Ext4Inode 实现了 InodeOp trait，并结合地址空间与块设备访问能力，具有如下核心特性

- 页缓存集成与统一地址空间管理： `Ext4Inode` 内含的 `address_space` 字段管理与 inode 关联的所有内存页缓存。该地址空间支持以下功能：
  - 按页粒度加载磁盘数据，实现懒加载与访问局部性优化；
  - 页面替换与回写策略的挂接点；
  - 支持透明地将逻辑页索引映射到物理块（通过 extent tree）；
  - 提供 Direct IO（ `write_dio` ）能力以绕过缓存写入，适用于性能敏感路径。
- 块设备访问桥接能力： 结构中的 `block_device: Arc<dyn BlockDevice>` 提供了统一的块设备访问通道，使 inode 在页缓存失效、回写等场景中可以直接触发块级 IO 请求，从而与存储设备紧密耦合。
- 文件系统耦合与元信息结构共享： 借助 `ext4_fs: Weak<Ext4FileSystem>` 字段，`Ext4Inode` 能够访问所属文件系统的共享信息，如块组描述符、inode bitmap、

数据块分配器等。该引用也是 inode 实现 extent tree 查找、块分配等操作的基础。

```
pub struct Ext4Inode {
    pub ext4_fs: Weak<Ext4FileSystem>,
    pub block_device: Arc<dyn BlockDevice>,
    pub address_space: Mutex<AddressSpace>,
    pub inode_num: usize,
    pub link: RwLock<Option<String>>,
    pub inner: FSMutex<Ext4InodeInner>,
    pub self_weak: Weak<Self>,
}
```

代码 5-3　Ext4Inode

RocketOS 中的 `Inode` 抽象设计充分借鉴了 Linux 虚拟文件系统（VFS）架构，同时结合 Rust 的语言特性与现代内核设计理念，形成了清晰、模块化、并具备良好扩展性的实现框架。通过 `InodeOp` trait 明确定义 inode 的操作语义，使具体文件系统（如 Ext4）可以在不侵入 VFS 核心的前提下实现定制逻辑，从而支持多种后端并存。通过 `address_space` 与 `block_device` 字段，`Inode` 成为连接缓存管理层与块存储层的重要桥梁，便于实现回写策略、直接 IO、预读优化等机制。

5.4　Dentry 与目录树

Dentry（目录项）是文件系统中用于表示目录结构的核心数据结构。它将文件名与对应的 Inode 关联起来，形成一个树形结构。Dentry 的主要作用是加速文件路径解析和目录项查找。

```
pub struct Dentry {
    pub absolute_path: String,
    pub flags: RwLock<DentryFlags>,
    inner: Mutex<DentryInner>,
}

pub struct DentryInner {
    inode: Option<Arc<dyn InodeOp>>,
    parent: Option<Arc<Dentry>>,
    children: HashMap<String, Weak<Dentry>>,
}
```

代码 5-4　Dentry

2em 在路径解析过程中，文件系统需要将用户输入的路径字符串（如 /usr/bin/bash）逐级解析为实际的 inode 对象。这一解析过程中的每一级目录组件（如 usr、bin）都会对应一个 Dentry 实例，从而构成路径节点链表（Dentry chain）。该链条从根目录向下逐级查找，通过每级 Dentry 提供的父子关联关系，结合 children 完成快速定位。

为提高路径查找的性能，RocketOS 在 VFS 层引入 DentryCache。该缓存以路径字符串为键，存储解析过的 Dentry 对象，并通过引用计数与负目录项机制（用于标识不存在的路径），标识路径查找失败的目录项，用于避免重复错误查找，提升路径解析性能。在绝大多数文件访问场景中，路径查找都可在 DENTRY_CACHE 命中，大幅减少 inode 查找开销。

同时为了确保 VFS 操作的安全性与合规性，Dentry 提供了一套完整的权限检查逻辑：

- dentry_check_access：依据当前任务用户 ID（UID）与组 ID（GID）判断是否具备指定的读、写、执行权限；
- dentry_check_open：结合 OpenFlags 检查文件类型兼容性、写权限、是否允许截断、是否允许创建等；
- chown：实现 POSIX 要求的 chown 逻辑，支持 root 修改权限与 setuid/setgid 位逻辑处理。

RocketOS 的 Dentry 模块是其 VFS 层的关键组成，旨在高效管理路径命名空间、提升路径查找性能并实现访问控制。其设计充分借鉴了 Linux VFS 架构，在保持功能通用性的同时融合 Rust 的类型安全与并发管理优势，达到了良好的系统可维护性、扩展性与运行效率。通过引入 DentryCache、负目录项、引用计数与类型标志机制，RocketOS 的路径解析框架具备现代操作系统所需的高性能路径解析能力。

5.5 FileOp 与文件

在 RocketOS 的虚拟文件系统（VFS）框架中，File 结构体承担着进程级别的打开文件表示，是系统中从用户态调用（如 read, write, lseek, ioctl 等）到内核态 inode 操作之间的关键桥接抽象。其设计参考了 Linux 中 struct file 的语义，在类型安全与并发访问控制方面结合了 Rust 的所有权模型与同步原语。每一个打开的文件描

述符在内核中都映射到一个唯一的 File 对象，封装了文件状态、访问语义与当前偏移量。

File 结构的核心包括一个受互斥锁保护的 FileInner 成员，后者保存了当前访问的 inode 引用、打开路径对应的 Path 结构体、打开标志（OpenFlags），以及动态维护的读写偏移量（offset）。该偏移量以字节为单位，反映了当前进程视角下对该文件的访问位置。在具有 O_APPEND 标志的场景下，偏移量初始化时即被设置为文件大小，以模拟追加写入的语义行为；否则偏移量初始化为零。

RocketOS 中的 File 实现了 FileOp trait，该 trait 统一了所有类型文件所需支持的操作接口，如 read, write, pread, pwrite, seek, truncate, fallocate 等。针对不同语义特征的文件类型（如常规文件、目录、字符设备、命名管道等），系统可通过实现该 trait 的多态对象（trait object）以适配各自行为。对于常规文件的实现而言，read 和 write 操作均基于当前偏移量进行，而 pread 与 pwrite 提供了带偏移量的无状态访问能力。此种抽象设计允许内核模块在实现如文件映射、direct I/O（write_dio）、文件同步（fsync）等功能时，不依赖具体文件类型，从而提升了模块解耦性。

File 抽象在路径语义上也具备重要角色。每个 File 持有其打开路径对应的 Path 引用，而 Path 内部又可追溯至打开文件的 Dentry。通过此设计，File 可在运行时访问其路径相关元信息，如判断是否为目录（通过 is_dir 查询 Dentry 类型标志），执行目录遍历（如 readdir）等。此外，系统也提供了权限检查函数（如 readable, writable），用于根据打开标志与 O_PATH 模式等判断用户对文件的操作权限。

偏移量管理亦为 File 抽象中一项关键机制。除基本的 get_offset 与 add_offset 操作外，系统支持标准化的文件指针移动接口 seek，并扩展支持 `SEEK_DATA` 与 `SEEK_HOLE` 等现代语义，便于处理稀疏文件。在这类语义下，RocketOS 通过 inode 提供的页级映射信息（如 lookup_extent）判断实际存在的数据区域或空洞位置，从而实现对非连续数据布局的有效识别。

综合来看，File 抽象在 RocketOS 中不仅仅是打开文件的简单句柄，更是一个承载访问语义、权限标志、偏移控制与路径绑定的中间层，其设计显著提高了文件系统的通用性与模块化程度。通过与 inode、dentry、path 等子系统的紧密协作，File

为内核提供了一个精确、可扩展且高度类型安全的文件访问模型，适配多种文件类型与访问模式，为构建现代类 UNIX 操作系统内核的文件语义奠定了坚实基础。

5.6 总结

RocketOS 的文件系统模块在设计上深度借鉴了 Linux VFS 架构，通过 Dentry、InodeOp、File 等抽象结构，实现了统一而灵活的文件访问语义。在挂载子系统方面，VfsMount、Mount 与 MountTree 协同构建全局命名空间，使多个后端文件系统得以共存，并通过路径解析机制高效完成用户请求的映射。在访问路径中，Dentry 提供了结构化的目录项管理与缓存机制，有效提升查找性能；而 InodeOp 则定义了文件系统核心操作接口，使具体文件系统如 Ext4 可在不修改 VFS 核心的前提下平滑接入。在此基础上，File 抽象则作为用户态系统调用与内核态文件操作之间的桥梁，承担偏移管理、访问控制与文件类型多态分发等职责。

整个文件系统框架强调模块解耦、抽象统一与类型安全，充分利用 Rust 所提供的 is 内存管理特性，在确保安全性的同时提升了系统可维护性与扩展性。通过这一体系，RocketOS 构建起一套具有现代操作系统特征的文件系统模型，为支持设备文件、网络文件系统、文件权限模型与高性能 I/O 提供了坚实基础。

第 6 章　信号处理模块

信号是操作系统向进程传递事件通知的一种机制，主要用来通知进程某个特定事件的发生，或者是让进程执行某个特定的处理函数。

6.1　信号结构

Sig 结构是信号在内核中的基本承载体，设计简洁高效，专门用于内核内部的信号管理和处理。它包含了信号识别、分发和处理过程中的核心信息，去除了冗余字段，确保在信号处理的关键路径上保持最佳性能。

SigInfo 结构则为信号传递提供了更丰富的信息描述。在 RocketOS 中，SigInfo 包含了信号处理中最常用的关键字段：signo 标识具体信号类型，code 提供信号产生的原因和上下文，fields 根据不同信号类型承载相应的附加数据。这种精简设计既满足了大部分信号处理需求，又避免了结构过于庞大影响系统性能。

```
pub struct SigInfo {
    pub signo: i32,     // 信号值
    pub code: i32,      // 信号产生原因
    pub fields: SiField, // 额外信息
}
```

代码 6-1　SigInfo 结构

当涉及用户态和内核态之间的信号信息传递时，RocketOS 采用 LinuxSigInfo 结构作为标准化接口。LinuxSigInfo 严格遵循 POSIX 标准规范，包含标准要求的所有字段和数据布局，确保与现有 Linux 应用程序的完全兼容。这种设计不仅体现在数据结构字段对应上，还包括内存布局、字节对齐以及各种信号类型特定信息的组织方式

通过内核内部使用精简 Sig 和 SigInfo 结构，而在系统调用接口使用标准 LinuxSigInfo 结构的双重设计，RocketOS 实现了内核效率与标准兼容性的平衡，既保证了内核高效运行，又确保了应用程序的无缝迁移和正确执行。

6.2　信号发送

对于线程级信号处理，系统采用精确定向策略，借助任务管理器结构获取指定任务结构，直接操作目标任务的信号待处理队列，将新接收的信号信息按序插入队

列结构中。这一机制通过队列缓冲确保了信号的可靠传递，即便目标线程当前处于繁忙执行状态，信号也不会因为时序冲突而丢失，为异步通信提供了坚实的可靠性保障。

进程级信号的处理逻辑则更为复杂，因为此类信号具有广播特性，需要同时影响整个线程组内的所有线程实体。系统通过遍历线程组中的每个线程，并对每个线程执行与线程级信号完全一致的入队操作序列，从而确保进程级事件能够均匀传播至所有相关执行单元。

由于信号具有异步中断特性，原本处于阻塞等待状态的任务需要具备被强制唤醒的能力。因此，在信号发送流程中，系统会主动检测目标任务的当前阻塞状态。当检测到任务正处于阻塞等待中，且接收到的信号类型具有阻塞中断权限时，系统会立即执行解除阻塞操作：将任务从相应的阻塞等待队列中移除，同时在任务状态字段中设置"信号中断唤醒"标记，并将该任务重新插入就绪队列等待调度。这种抢占式唤醒机制保障了信号驱动事件的实时性，确保被阻塞任务能够突破原有等待条件的束缚，及时响应关键系统事件或用户请求。

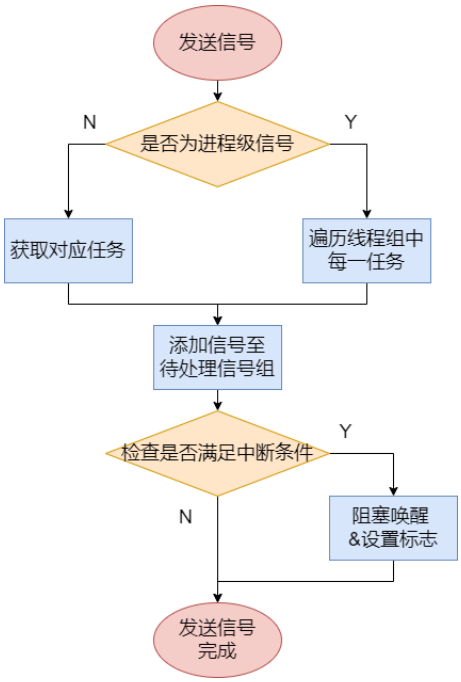

图 6-1　信号发送流程图

6.3　信号处理流程

6.3.1 基本信号处理流程

信号的处理流程是操作系统中一个精密的异步事件处理机制。当系统事件发生时，内核首先在目标进程的进程控制块中设置相应的信号标志位，将信号标记为待处理状态。信号不会立即处理，而是等待特定时机，主要是进程从内核态返回用户态时、被唤醒时或系统调用返回时。

内核检查待处理信号时，会先查看进程的信号屏蔽字，确定哪些信号被阻塞。对于未阻塞的信号，根据处理方式分别执行：默认处理直接执行系统预定义操作；忽略信号则清除待处理标志；自定义处理最为复杂。

执行自定义信号处理时，内核先保存进程当前的执行上下文，在用户栈上构造特殊栈帧，修改程序计数器指向处理函数，并传递信号参数。进程在特殊环境中执行处理函数，期间通常会自动阻塞同类型信号防止重入。处理函数应保持简单快速，避免调用不安全函数。

函数返回后，控制权转移到内核预设的返回代码，触发系统调用通知处理完成。内核随后执行清理工作，恢复保存的执行上下文、清理栈帧、恢复信号屏蔽字，最终将进程执行流程恢复到信号发生前的状态，确保进程继续正常执行。

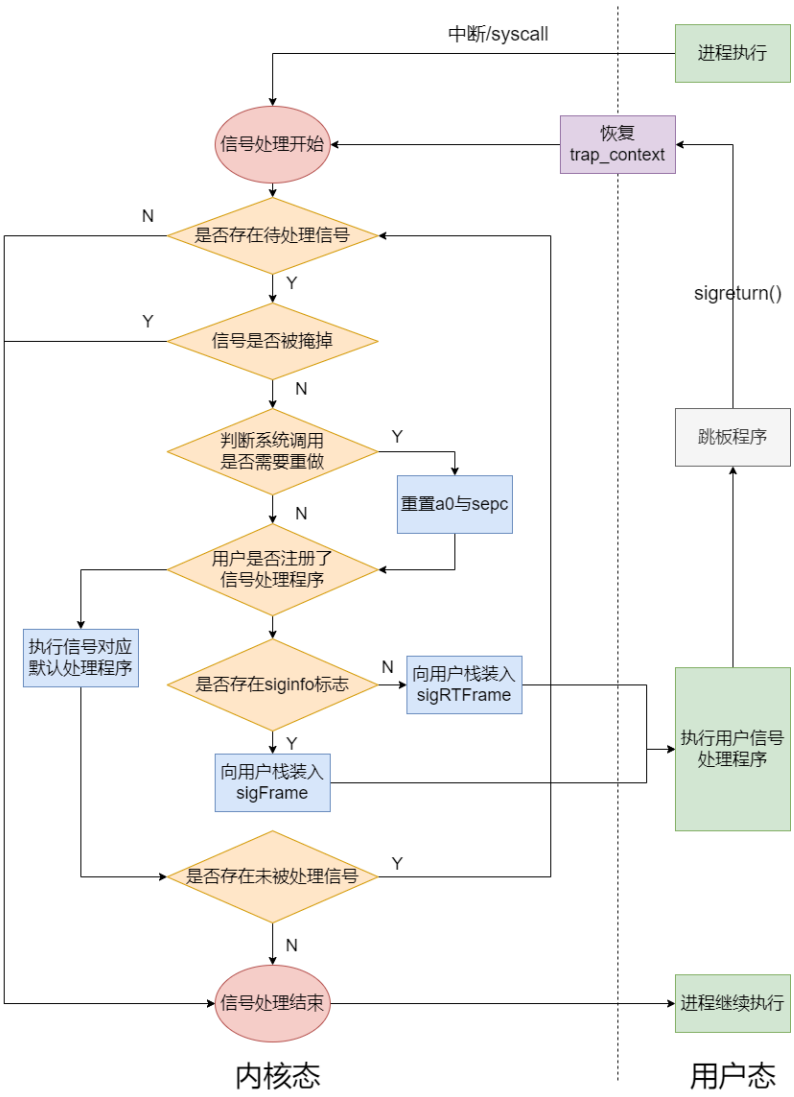

图 6-2　信号处理流程图

6.3.2　对 SA_RESTART 的特别处理

SA_RESTART 是 Linux 信号处理架构中的关键标志位，专门用于解决信号中断与系统调用执行之间的协调问题，保障系统调用的执行连续性和操作可靠性。

在传统 Linux 实现中，SA_RESTART 标志位采用"先重置后撤销"的处理策略，虽然这种方式能够确保信号处理逻辑的正确性，但会产生不必要的状态保存开销，降低了系统调用的执行效率。而在 RocketOS 中，我们采用了更为精细化的优化策略：通过多重条件判断机制来决定是否执行重置操作，避免了盲目的状态保存。这一判断机制基于四个关键条件进行综合评估，只有在确实需要重置的情况下才执行相应操作，从而显著减少了不必要的系统开销，提升了信号处理的整体性能。

1. 需要重启的系统调用是否可以重启
2. 信号处理函数是否存在 SA_RESTART 标志位
3. 信号处理函数是否被被用户注册
4. 任务是否被信号中断阻塞

```
if task.can_restart()
    && action.flags.contains(SigActionFlag::SA_RESTART)
    && task.is_interrupted()
    && action.sa_handler != SIG_DFL
    && action.sa_handler != SIG_IGN
{
    // 回到用户调用 ecall 的指令
    trap_cx.set_sepc(trap_cx.sepc - 4);
    trap_cx.restore_a0(); // 从 last_a0 中恢复 a0
}
```

代码 6-2　SA_RESTART 判断条件

6.3.3　sigframe 结构设计

在用户态进程的信号处理机制中，SA_SIGINFO 标志位扮演着关键的角色，它决定了内核在信号处理过程中采用何种策略来构造和管理信号上下文信息。这个标志位的存在与否直接影响着信号处理函数的调用方式、参数传递机制以及上下文保存恢复的具体实现。当用户态进程通过 sigaction 系统调用注册自定义信号处理函数时，如果在 sa_flags 字段中设置了 SA_SIGINFO 标志位，这向内核表明该信号处理函数需要接收详细的信号信息。内核会根据这个标志位的存在来选择合适的信号帧结构体类型，确保为信号处理提供正确的上下文环境和参数传递机制。

对于未设置 SA_SIGINFO 标志位的情况，内核采用传统的信号处理方式。在这种模式下，内核会直接调用用户态的信号处理函数，仅将信号编号作为单一参数传递给处理函数。这种简化的处理方式对应着较为精简的上下文保存需求，因此内核选择构造常规的 SigFrame 结构体来满足这一需求。SigFrame 结构体设计简洁明了，包含了一个 FrameFlags 类型的 flag 字段用于标识结构体类型，以及一个 SigContext 结构体用于保存处理器的执行上下文信息。

```rust
pub struct SigFrame {
    pub flag: FrameFlags,     // 标志位
    pub sigcontext: SigContext, // 上下文信息
}
pub struct SigContext {
    pub sepc: usize,
    pub x: [usize; 32],
    pub last_a0: usize,
    pub kernel_tp: usize,
    pub mask: SigSet, // 记录原先的 mask
}
```

代码 6-3　SigContext 结构

当用户信号处理程序注册时设置了 SA_SIGINFO 标志位，情况变得更加复杂和功能丰富。在这种情况下，信号处理函数不仅需要接收信号编号，还需要获得包含详细信号信息的 siginfo 结构体以及完整的用户上下文信息。为了满足这些额外的需求，内核必须构造更为完整和复杂的 SigRTFrame 结构体。SigRTFrame 结构体代表了信号处理机制的高级形态，它不仅包含了用于标识结构类型的 flag 字段，还整合了 UContext 结构体来提供完整的用户上下文信息，以及 LinuxSigInfo 结构体来传递详细的信号相关信息。这种设计使得信号处理函数能够获得更丰富的上下文信息，从而实现更精细化的信号处理逻辑。

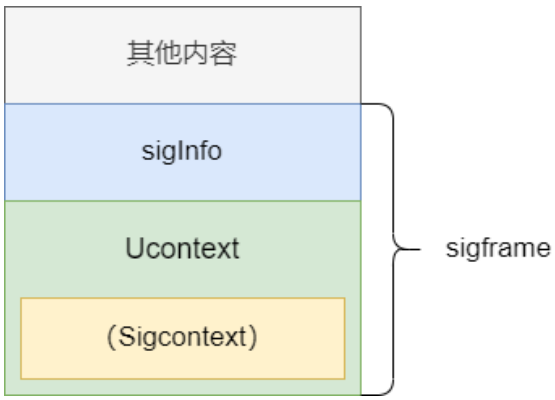

图 6-3　SigRTFrame 结构图

在用户信号处理流程结束后，会自动通过预先设置的跳板平台来调用 sigreturn 返回内核态，在 sigreturn 中通过在信号处理完成后检查 sigframe 中的 flag 字段，内核能够准确识别当前使用的结构体类型，从而选择相应的恢复策略来提取正确的上

下文信息。这种设计不仅保证了不同信号处理模式下的功能正确性，还优化了系统资源的使用效率，避免了在简单信号处理场景中构造过于复杂的上下文结构。

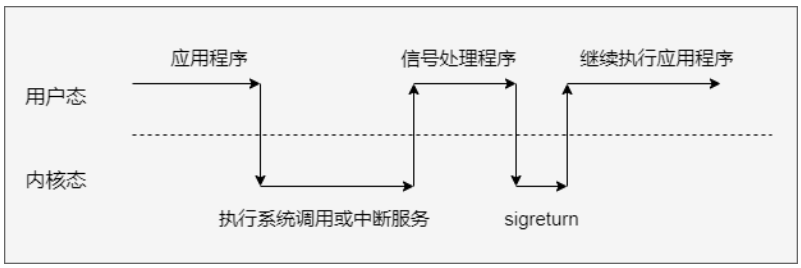

图 6-4 信号处理流程图

第 7 章　网络系统

网络系统基于 smoltcp 协议栈实现，旨在构建高效灵活的网络通信能力。本系统支持 AF_INET、AF_INET6、AF_UNIX 及 AF_ALG 等多种地址族的套接字操作，具备 IPv4 与 IPv6 双协议栈处理能力，并完整实现了 TCP 与 UDP 传输协议。其功能性已通过 iperf、netperf 及 LTP 相关测试验证。网络系统通过统一的抽象接口管理所有网络设备与套接字资源。

7.1　网络系统概述

网络工作模式如下:

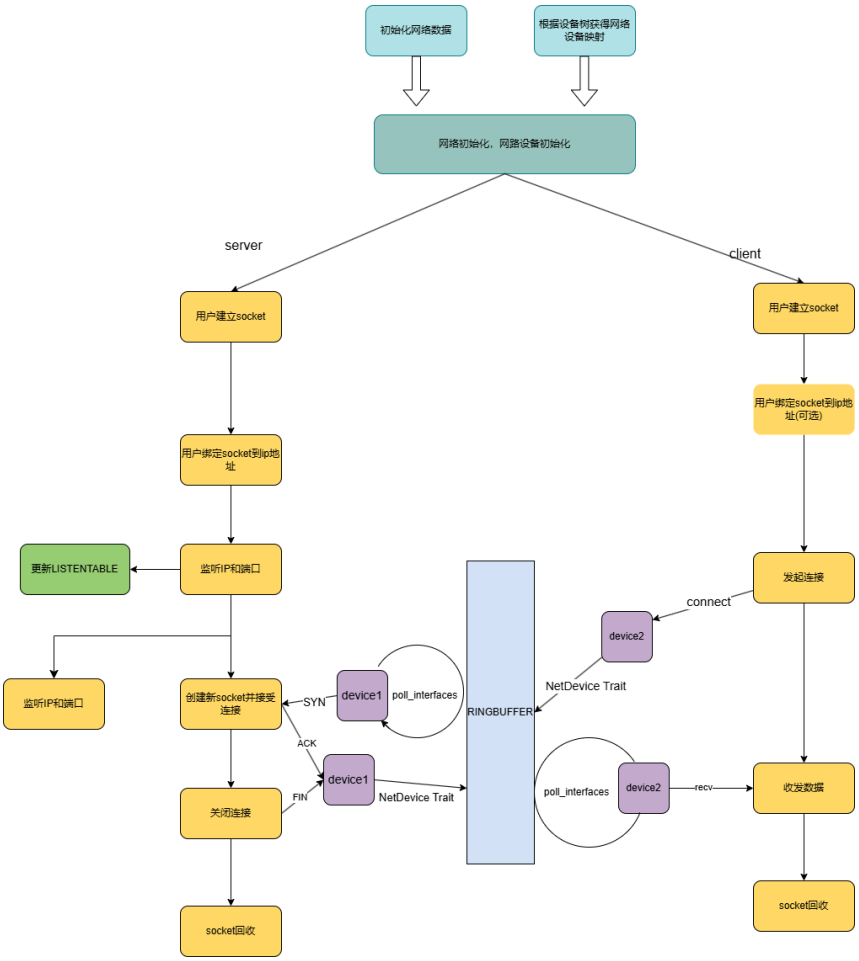

图 7-1　net 工作模式

网络系统包括以下几个主要组件:

- **NetDevice**: 定义网络设备基本操作与特性的抽象接口 NetDevice 。基于该接口可实现异构网络设备，包括虚拟本地设备 VirtioNetDevice 和虚拟本地回环设备 LoopbackDev 。
- **InterfaceWrapper**: 网络系统的网卡抽象，该组件封装 *smoltcp* 的 Interface 接口和 NetDeviceWrapper ，提供对网卡设备的统一管理机制，支持多硬件网卡在操作系统内的映射实例化。与 Linux 架构类似，系统分别管理 lo 回环设备与 eth33 虚拟网卡设备。其中 eth33 通过 QEMU 的 10.0.2.15 地址映射至主机 10.0.2.2 接口。
- **ListenTable**: 全局端口监听表。维护端口至关联套接字的映射关系，通过访问套接字状态判定连接许可性。
- **Socket**: Socket 内核套接字的封装实现，遵循 FileOp 接口规范，支持通过文件描述符进行访问控制与操作。

7.2 网络 Device 设备–物理层

7.2.1 多架构适配机制

网络系统支持 LoongArch 与 RISC-V 双架构，通过设备树解析实现设备至内核地址空间的映射。在 RISC-V 架构中，网络设备通过内存映射 I/O（MMIO）映射至设备地址空间；在 LoongArch 架构中，则通过 PCI 总线挂载设备。网络系统采用条件编译策略实现架构差异的设备树解析。

```
#riscv qemu 网络配置
-device virtio-net-device, netdev=net -netdev user, id=net, hostfwd=tcp::5555-:5555,
hostfwd=udp::5555-:5555\
#loongarch qemu 网络配置
-device virtio-net-pci, netdev=net -netdev user, id=net, hostfwd=tcp::5556-:5555,
hostfwd=udp::5556-:5555 \
```

代码 7-1 网络配置

在 RISC-V 中，通过 rust_main 入口的 dtb_address 参数定位设备树基址，遍历设备树节点并筛选 compatible 属性为 virtio-net 的节点，解析其 reg 属性获取 MMIO 地址完成内核映射。

算法 7-1: riscv 网络设备初始化流程

```
    input: dtb_addr
    output: initialized net device
1   let dev_tree ← Fdt::from_ptr(dtb_addr + KERNEL_BASE)
2   address_cells ← dev_tree.root().prop("address-cells").value[3]
3   size_cells ← dev_tree.root().prop("size-cells").value[3]
4   for node in dev_tree.all_nodes() do
5     for prop in node.properties() do
6       log(prop.name)
7   for node in dev_tree.all_nodes() do
8     if node.name == "soc" then
9       for child in node.children() do
10        if child.name == "virtio_mmio@10008000" then
11          reg ← parse_reg(child, address_cells, size_cells)
12          mmio_base ← reg[0].start
13          mmio_size ← reg[0].length
14          map_area ← MapArea::new( VPNRange(KERNEL_BASE+mmio_base,
              KERNEL_BASE+mmio_base+mmio_size), Linear, R|W )
15          KERNEL_SPACE.lock().push(map_area)
16          sfence.vma()
17          NET_DEVICE_ADDR.lock().replace(KERNEL_BASE+mmio_base)
18          header ← NonNull((KERNEL_BASE+mmio_base) as mut VirtIOHeader)
19          transport ← MmioTransport::new(header)
20          log("vendor=", transport.vendor_id(), "version=", transport.version(),
              "type=", transport.device_type())
21          dev ← VirtioNetDevice::new(transport)
22          net::init(Some(dev))
23          return
24  log("not find a net device")
```

而在 Loongarch 中，遍历 PCI 总线设备，筛选 `device_type` 为 `network` 的节点，解析其 BAR 寄存器完成设备地址映射。

算法 7-2: 基于 PCI 的 VirtIO 设备初始化流程

```
    input: pci_root, allocator
    output: 初始化并启动 VirtIO 设备
1   for (device_fn, info) in pci_root.enumerate_bus(0) do
2     status, command ← pci_root.get_status_command(device_fn)
3     log("Found", info, "at", device_fn, "status", status, "command", command)
4     if virtio_device_type(&info) then virtio_type
```

```
5       log(“ VirtIO”, virtio_type)
6       allocate_bars(&mut pci_root, device_fn, &mut allocator)
7       dump_bar_contents(&mut pci_root, device_fn, 4)
8       transport ← PciTransport::new::(&mut pci_root, device_fn).unwrap()
9       log( “Detected virtio PCI device with type”, transport.device_type(), “features”,
          transport.read_device_features() )
10      virtio_device(transport)
11  fn virtio_device(transport) do
12    match transport.device_type() with
13      DeviceType::Block => virtio_blk(transport)
14      DeviceType::Network =>
15        log(“[initialize net]”)
16        virtio_net(transport)
17      t => log(“Unsupported VirtIO device type”, t)
```

7.2.2 NetDevice 封装

网络设备封装层实现 *smoltcp* 的 Device 接口，通过 NetDeviceWrapper 完成对底层设备的统一抽象，支持虚拟网卡与回环设备等异构设备接入，其逻辑关系如下：

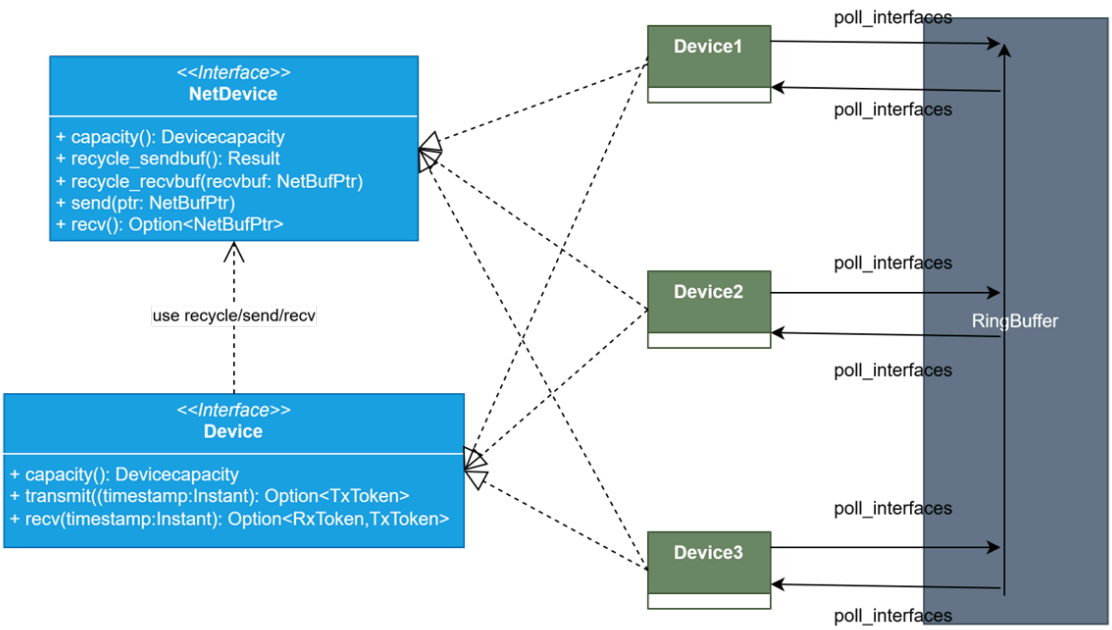

图 7-2 Netdeivce

为实现与 *smoltcp* Device 接口的兼容性，定义 NetDeviceWrapper 结构体。该结构体通过 RefCell 封装 Box<dyn NetDevice>，在实现 trait 时提供内部可变性访问支持。

```rust
pub struct NetDeviceWrapper {
    inner: RefCell<Box<dyn NetDevice>>,
}
// 网络设备管理，实现 sync 和 send 特性，以便在多线程环境中安全使用
pub trait NetDevice:Sync + Send {
    //获取设备容量
    fn capabilities(&self)->smoltcp::phy::DeviceCapabilities;
    //获取设备 mac 地址
    fn mac_address(&self)->EthernetAddress;
    //是否可以发送数据
    fn isok_send(&self)->bool;
    //是否可以接收数据
    fn isok_recv(&self)->bool;
    //一次最多可以发送报文数量
    fn max_send_buf_num(&self)->usize;
    //一次最多可以发送报文数量
    fn max_recv_buf_num(&self)->usize;
    //回收接收 buffer
    fn recycle_recv_buffer(&mut self, recv_buf:NetBufPtr);
    //回收发送 buffer
    fn recycle_send_buffer(&mut self)->Result<(), ()>;
    //发送数据
    fn send(&mut self, ptr:NetBufPtr);
    //接收数据
    fn recv(&mut self)->Option<NetBufPtr>;
    //分配一个发送的网络缓冲区
    fn alloc_send_buffer(&mut self, size:usize)->NetBufPtr;
}
```

代码 7-2 NetDeviceWrapper

系统通过 NetBufPool 实现网络缓冲区的统一管理，采用预分配策略优化动态内存分配效率。设备在 recycle_recv_buffer 与 recycle_send_buffer 操作中调用 alloc 与 dealloc 方法，显著提升网络通信效率并降低内存碎片化概率。

```rust
/// A pool of [`NetBuf`]s to speed up buffer allocation.
///
/// It divides a large memory into several equal parts for each buffer.
pub struct NetBufPool {
    //可以存储的 netbuf 个数
    capacity: usize,
    //每个 netbuf 的长度
    buf_len: usize,
    pool: Vec<u8>,
    //用于存储每个待分配的 netbuf 的 offset
    free_list: Mutex<Vec<usize>>,
}
```

代码 7-3　NetBufPool

算法 7-3: NetBuf 缓冲区分配与回收

fn alloc(self: Arc) ⟶ NetBuf
output: 新分配的 NetBuf
1 offset ← self.free_list.lock().pop().unwrap()
2 buf_ptr ← NonNull(self.pool.as_ptr().add(offset) as **mut u8**)
3 **return** NetBuf { header_len: 0, packet_len: 0, capacity: self.buf_len, buf_ptr: buf_ptr, pool_offset: offset, pool: Arc::clone(self), }
4
fn dealloc(self, offset: usize) ⟶ ()
precondition: offset % self.buf_len == 0
5 assert(offset % self.buf_len == 0)
6 self.free_list.lock().push(offset)

系统为具体网络设备实现 *smoltcp* 的 `Device` trait。该实现使得 *smoltcp* 的 `poll` 轮询机制可通过 trait 方法调用 `NetDeviceWrapper` 的底层操作，完成网络数据包的收发处理。

如代码与图示，*smoltcp* 采用**环形令牌网络**实现设备轮询。实现的 `Device trait` 在轮询过程中管理令牌分配，当 `NetDeviceWrapper` 持有令牌时，通过 `NetDevice` 接口触发数据包处理。

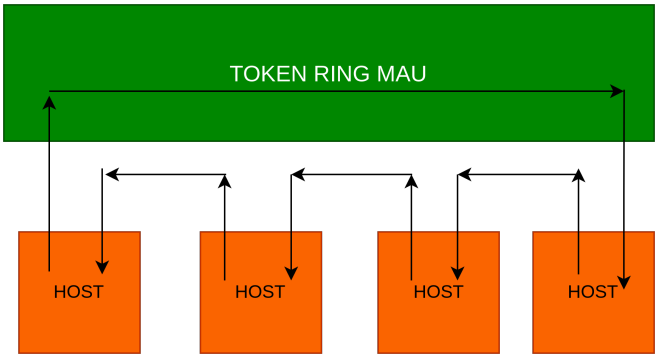

图 7-3　环形令牌网络

算法 7-4: NetDeviceWrapper 驱动接口实现伪代码

```
    impl Device for NetDeviceWrapper
    output: Rx/Tx 令牌或设备能力
1   fn receive(self, timestamp: Instant) → Option<(RxToken, TxToken)>
2     dev ← self.inner.borrow_mut()
3     if let Err(e) = dev.recycle_tx_buffers() then
4       warn("recycle_tx_buffers failed:", e)
5       return None
6     if ¬dev.can_transmit() then return None
7     match dev.receive() with
8       Ok(buf) => rx_buf ← buf
9       Err(DevError::Again) => return None
10      Err(err) =>
11        warn("receive failed:", err)
12        return None
13    return Some((NetRxToken(&self.inner, rx_buf), NetTxToken(&self.inner)))
14
15  fn transmit(self, timestamp: Instant) → Option
16    dev ← self.inner.borrow_mut()
17    if let Err(e) = dev.recycle_tx_buffers() then
18      warn("recycle_tx_buffers failed:", e)
19      return None
20    if dev.can_transmit() then
21      return Some(AxNetTxToken(&self.inner))
22    else return None
23
24  fn capabilities(self) → DeviceCapabilities
25    caps ← DeviceCapabilities::default()
26    caps.max_transmission_unit ← 1514
```

```
27    caps.max_burst_size ← None
28    caps.medium ← Medium::Ethernet
29    return caps
```

7.3 Interface 设备–数据链路层

网络接口设备通过 InterfaceWrapper 封装 *smoltcp* 的 Interface 与 NetDeviceWrapper，提供网卡设备的统一管理接口。该封装实现双重功能：

- 通过 *smoltcp* 的 poll 轮询机制监听网络事件;
- 当事件触发时，通过 NetDevice 接口执行数据包收发操作。

```rust
pub struct InterfaceWrapper {
    //smoltcp 网卡抽象
    iface: Mutex<Interface>,
    //网卡 ethenet 地址
    address: EthernetAddress,
    //名字 eth0
    name: &'static str,
    dev: Mutex<NetDeviceWrapper>,
}
```

代码 7-4 InterfaceWrapper 结构体

系统通过 poll_interfaces 方法实现多网卡设备的协同轮询。轮询过程中，依托实现的 Device trait 对存在网络事件的设备执行数据收发操作。

```rust
pub fn poll_interfaces(&self) {
    //对本地回环设备轮询
    LOOPBACK.lock().poll(
        Instant::from_micros_const((current_time_nanos() / NANOS_PER_MICROS) as i64),
        LOOPBACK_DEV.lock().deref_mut(),
        &mut self.0.lock(),
    );
    //对 ens0 设备轮询
    ETH0.poll(&self.0);
}
```

代码 7-5 poll_interfaces

7.4 ListenTable 监听表–网络层

网络系统通过全局单例 LISTENTABLE 管理所有监听端口及其关联套接字。该表维护端口到套接字的映射关系，并通过套接字状态机实施连接许可控制，其逻辑架构如下：

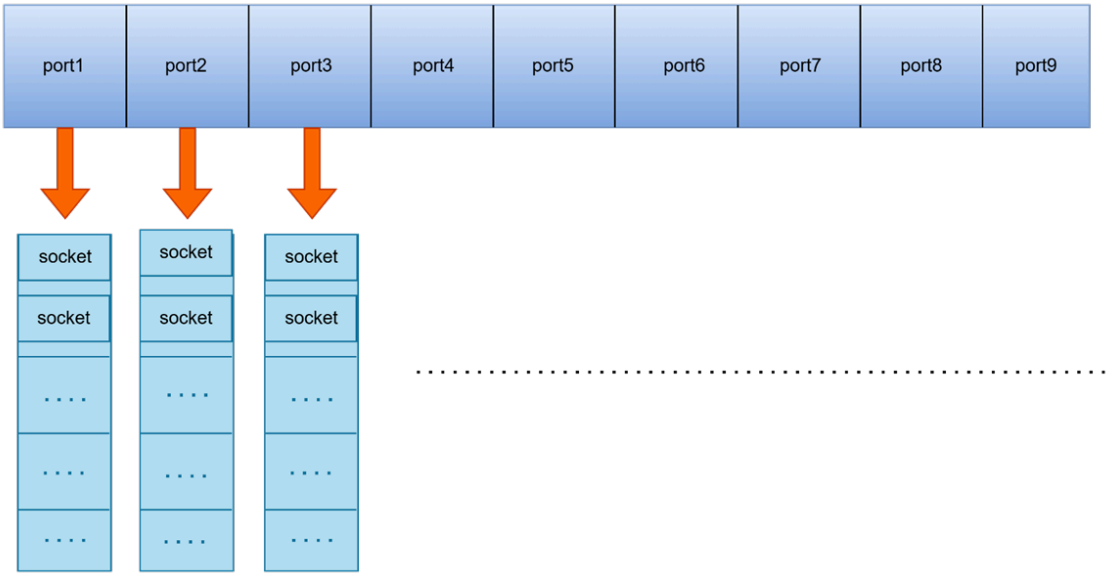

图 7-4 listentable

```rust
static LISTEN_TABLE: LazyInit<ListenTable> = LazyInit::new();
pub struct ListenTable{
    //是由 listenentry 构建的监听表
    //监听表的表项个数与端口个数有关，每个端口只允许一个地址使用
    table:Box<[Mutex<Option<Box<ListenTableEntry>>>]>,
}
#[derive(Clone)]
struct ListenTableEntry{
    //表示监听的 server 地址 addr
    listen_endpoint:IpListenEndpoint,
    task_id:usize,
    //这里由于 sockethandle 与 socket 存在 RAII 特性，因而可以保存 sockethandle
    syn_queue:VecDeque<SocketHandle>
}
```

代码 7-6 ListenTablei 结构体

7.5 Socket 封装–传输层

网络系统采用三层封装实现套接字管理，支持 AF_UNIX、AF_INET、AF_ALG 及 AF_INET6 地址族，兼容 TCP/UDP 协议。套接字实现 FileOp 接口，支持通过文件描述符进行标准化访问。

内核套接字定义如下。Socket 结构体封装协议类 SocketInner 、套接字类型及具体实现，包含状态信息与缓冲区元数据。所有字段均通过原子操作或互斥锁（Mutex）保护，确保多线程环境下的状态一致性。SocketInner 进一步封装 TCP、UDP、UNIX 及 ALG 等协议的具体实现。

```
pub struct Socket {
    pub domain: Domain,
    pub socket_type: SocketType,
    inner: SocketInner,
    recvtimeout: Mutex<Option<TimeSpec>>,
    dont_route: bool,
    ...
}
pub enum SocketInner {
    Tcp(TcpSocket),
    Udp(UdpSocket),
    Unix(UnixSocket),
    Alg(AlgSocket),
}
```

代码 7-7　InterfaceWrapper

套接字作为文件描述符的载体，通过实现 FileOp 接口支持标准文件操作语义。这使得套接字可无缝集成至文件描述符系统调用（如 read/write）的上下文中。

网络系统的套接字管理遵循**资源获取即初始化**（RAII）原则。为 Socket 实现 Drop trait，确保当套接字关闭（shutdown）时，自动释放关联资源并从全局 SocketSetWrapper 中移除其句柄。

```
pub fn remove(&self,  handle: SocketHandle) {
    let socket=self.0.lock().remove(handle);
    drop(socket);
}
```

代码 7-8　Socket remove

通过上述设计，网络系统实现以下核心特性：

1. 异构设备统一封装：支持物理与虚拟网卡（Virtio、回环设备）的透明接入
2. 多协议栈支持：完整实现 IPv4/IPv6、TCP/UDP、AF_UNIX、AF_ALG 协议族
3. 跨架构兼容：在 RISC-V64 与 LoongArch 平台高效运行
4. 分层管理机制：严格分离设备轮询、端口监听与套接字操作关注点
5. 资源安全模型：基于 RAII 与原子操作保障资源生命周期与线程安全

第 8 章　总结与展望

8.1　当前工作总结

截至初赛结束，RocketOS 已构建起完整的操作系统核心架构，成功实现了内存管理模块的高效内存分配与虚拟内存机制，建立了支持多种文件操作和目录管理的文件系统，完善了进程间信号传递与处理的信号机制，部署了具备多任务调度和进程间通信能力的进程管理系统，并集成了支持 TCP/IP 协议的网络协议栈以实现基本网络通信功能。这些核心模块的成功实现为 RocketOS 奠定了坚实的系统基础，展现了其作为现代化操作系统的完整性和实用性。

8.2　经验与教训

在 RocketOS 的开发过程中，我们遇到了多项具体的技术挑战和困难。

首先，双架构适配成为最大的技术难点，RISC-V 和 LoongArch 在指令集、内存模型和中断处理机制上存在显著差异，这要求我们重新设计硬件抽象层，并为每个架构编写专门的底层代码。

其次，内存管理模块的实现比预期复杂，特别是在处理不同架构的页表结构和虚拟内存映射时，需要大量的调试工作来确保内存分配的正确性和效率。在 RocketOS 的内存管理优化中，我们实施了批处理、懒分配和写时复制等关键策略。批处理分配通过预分配内存池减少了频繁分配的开销。懒分配策略延迟物理内存分配直到实际使用，有效降低了内存占用，但增加了页面错误处理的复杂性，特别是在不同架构间的页表管理差异使实现变得困难。写时复制机制在进程创建时显著减少了内存复制开销，但其实现需要复杂的页面权限管理和写保护异常处理。

同时，调试工具的缺乏使得问题定位变得困难，特别是在处理底层硬件相关的错误时，往往需要依靠日志输出和手动分析来排查问题。这些困难虽然增加了开发复杂度，但也促使团队在系统架构设计和问题解决能力方面获得了显著提升。

8.3　未来工作展望

在之后的工作中，我们计划继续完善 RocketOS 的功能和性能，主要包括以下几个方面：

1. 修改调度策略，适配多核调度。
2. 适配 riscv 与 loongarch 开发板，完善相关驱动。
3. 支持更多 ltp 测例，修复更多内核不稳定的 bug。
4. 完善文件系统，支持更多文件操作。
5. 完善网络协议栈，支持更多网络功能。
6. 支持图形界面