哈爾濱工業大學(深圳)

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

哈尔滨工业大学（深圳）

AstrancE

基于 Rust 的模块化宏内核操作系统设计与实现

AstranciA 项目组

项目成员：曾熙晨（组长）、杨纯懿、滕奇勋

指导老师：夏文、仇浩婷

2025 年 6 月

摘　要

AstrancE 作为一个基于 Rust 的模块化宏内核操作系统，旨在突破 ArceOS 作为实验性微内核操作系统在用户态支持和系统调用方面的限制。尽管 AstrancE 继承了 ArceOS 模块化设计和内存安全的优势，但其核心架构已从微内核扩展为支持完整用户态交互的宏内核系统，显著提升了系统的功能完整性和实用性。

区别于 ArceOS 的最小化设计理念，AstrancE 在内核态整合了更强的功能模块，通过引入 axsyscall 与 axmono 等关键模块，不仅填补了 ArceOS 在系统服务层面的空白，也为传统 C/Rust 用户程序的运行提供了良好平台。围绕这一核心转变，AstrancE 针对进程管理、内存管理、文件系统、信号机制和用户态支持进行了扩展与优化：

截至 6 月 29 日，取得 14 名的成绩：(占位)

14	T202518123995667	AstranciA/ 哈尔滨工业大学（深圳）	27	2025-06-26 17:25:53	91.0	95.0	97.0	97.0	0.0

表 1: 各模块完成情况

目标	基本完成情况	说明
进程管理	基本完成	增加用户态 ELF 加载功能，支持独立虚拟地址空间运行；实现用户态线程创建与调度
内存管理	基本完成	支持 COW、懒分配与多段映射；引入 RAII 风格页表管理；支持 mmap 跨区域内存映射；实现共享内存
文件系统	基本完成	增加 procfs，可运行时生成系统信息；实现动态挂载与 devfs 扩展，支持设备节点管理；
信号机制	基本完成	实现 POSIX 信号全流程，引入备用信号栈与 sigreturn_trampoline，信号系统与调度器联动
用户态支持	基本完成	建立宏内核式 syscall 分发框架；实现统一用户态运行环境；独立虚拟地址空间与 copy_from_kernel 隔离方案

目　录

[1]加注”*” 号的是内核中由 AstranciA 组开发的部分

[2]加注”**” 号的是 ArceOS 有一定功能实现，但由 AstranciA 组进行了重大修改、优化的部分

1 内核开发背景

本项目基于 **ArceOS** 开展开发工作。ArceOS 是一个采用 Rust 语言编写的实验性模块化操作系统（或单内核），其核心设计理念强调模块化与安全性。

ArceOS 的系统架构被细致地划分为多个功能模块，涵盖：**内存与驱动支持**：如 `axalloc`（内存分配）、`axdriver`（驱动框架）、`axhal`（硬件抽象层）；**文件系统**：`axfs` 等；**基础功能组件**：如 `axsync`（同步机制）、`axtask`（任务管理）、`axlog`（日志系统）等。这种模块化设计使得每个功能都以独立的 Crate 形式存在，极大地便利了系统的定制、维护与替换。得益于 Rust 语言的内存安全特性，ArceOS 有效规避了诸如空指针引用、缓冲区溢出等传统系统编程中常见的安全漏洞。该系统具备良好的硬件抽象层（`axhal`），并通过 `axdriver` 模块支持跨平台驱动开发。ArceOS 原生支持异步与并发编程范式，通过 `axsync` 和 `axtask` 模块实现了高效的任务调度与同步机制。此外，`arceos\posix_api` 模块提供了一定程度的 POSIX 兼容性，为传统用户态应用的移植与复用提供了便利。

尽管 ArceOS 仍处于积极开发阶段，其当前版本中许多模块仅实现了运行所需的最基本功能。具体而言，其设备文件系统目前仅支持如 Null 和 Zero 这类抽象设备，尚不具备对实际硬件设备的管理能力；进程管理中的信号处理机制仍有待完善；内存管理中的页表操作、系统调用（syscall）实现以及网络协议栈等核心组件也需进一步补全。

本项目基于 ArceOS 现有框架，旨在通过深入开发，推进系统功能的进一步完善。在初赛阶段，我们重点优化了内存管理、信号处理与文件系统模块，并显著增强了对用户态程序的支持能力。整个开发过程围绕“功能完整性”展开，充分利用 ArceOS 的模块化架构所赋予的良好可定制性与可拓展性，尤其适用于面向特定需求打造定制化操作系统。

项目的核心创新在于通过引入 `axsyscall` 和 `axmono` 两个关键模块，成功实现了从 ArceOS 的微内核架构向宏内核架构的转换，从而为传统用户态应用提供了完整的系统调用接口支持。此外，ArceOS 对多架构平台（如 RISC-V 和 LoongArch）的良好支持，也为本项目满足比赛要求奠定了坚实基础，进一步凸显了其在跨平台驱动开发中的适用价值。

2 内核总体设计架构概述

AstrancE 操作系统采用模块清晰、抽象统一的设计原则，其五大核心子系统协同支撑调度、内存、文件、信号与用户态功能，具备高可移植性、可维护性与良好的拓展价值：

- **进程管理：用户态程序支持与线程管理**基于 ArceOS 的 `axtask` 模块，本系统新增了用户态程序支持功能。通过 `axmono` 模块实现了 ELF 文件加载器，支持将用户态程序加载到独立的虚拟地址空间中运行。同时，实现了完善的线程支持功能，通

过修改 axmm 模块的帧管理机制，支持用户态多线程程序的创建和管理。系统在保持 ArceOS 原有任务调度机制的基础上，进一步扩展了对用户态程序的生命周期管理。

- **内存管理：跨区域映射与共享内存**以 AddrSpace 为核心，实现了完整的虚拟地址空间管理。新增了跨区域内存映射（mmap）支持，通过修改 mmap.rs 实现了跨不同内存区域的映射功能。支持写时复制（Copy-on-Write, COW）机制，通过 clone_on_write 方法优化内存使用效率。实现了共享内存映射功能，通过 shm_mmap 方法支持进程间共享内存，为多进程协作提供了基础。此外，针对内存区域和页表管理，引入了 RAII（资源获取即初始化）思想重写相关实现，通过 Rust 的所有权和生命周期机制，自动管理内存资源的分配与释放，极大提升了内存安全性和健壮性。系统还支持内存保护、页面错误处理等高级内存管理特性，为用户态和内核态的高效隔离与协作提供了坚实基础。

- **文件系统：VFS 框架与动态挂载**基于统一的虚拟文件系统（VFS）接口，实现了模块化的文件系统架构。新增了 procfs 支持，通过 ProcFileSystem 和 ProcDir 实现了动态文件系统，支持运行时生成系统信息文件。实现了动态挂载机制，支持将不同文件系统挂载到指定目录。扩展了 devfs 功能，支持设备文件的动态创建和管理。系统通过 axfs_crates 模块实现了文件系统的可插拔设计。

- **信号机制：POSIX 兼容的异步通知**实现了完整的 POSIX 信号处理机制，包括信号注册、屏蔽、触发、处理的全流程。通过 SignalContext 结构体管理信号处理表、阻塞信号集和待处理信号集。新增了默认信号处理器，支持 SIGKILL、SIGTERM 等常用信号的默认处理行为。支持备用信号栈和上下文恢复，实现了 sigreturn_trampoline 机制。信号机制与调度器深度融合，确保异步通知的及时性和可靠性。

- **用户态支持：系统调用与进程隔离**通过 axsyscall 模块实现了宏内核架构的系统调用分发机制，支持超过 50 个 POSIX 系统调用，涵盖文件操作、进程管理、内存管理等核心功能。新增了 axmono 模块，为用户态程序提供统一的接口支持，包括 ELF 加载器、进程创建、多线程支持等功能。实现了独立的虚拟地址空间管理，通过 copy_from_kernel 方法确保用户态与内核态的地址空间隔离。系统支持 C 和 Rust 两种开发语言，提供了完整的 POSIX API 兼容性。

2.1 项目的技术特色

本项目的技术特色主要体现在以下几个方面：

- **架构转换创新：**成功实现了从 ArceOS 的微内核向宏内核的架构转换，在保持模块化优势的同时提供了完整的系统调用支持。

- **内存管理增强**：实现了跨区域映射、共享内存等高级内存管理功能，并引入 RAII 思想提升内存安全性和健壮性。
- **文件系统扩展**：新增 `procfs` 支持，实现了动态挂载和文件系统可插拔机制，增强了文件系统的功能性和可扩展性。
- **信号处理完善**：实现了完整的 POSIX 信号机制，包括默认信号处理器，提升了系统的异步通知能力和稳定性。
- **用户态支持**：通过 `axmono` 模块提供了完整的用户态程序运行环境，支持 ELF 加载、进程与线程管理。
- **构建工具创新**：开发了 `acbat` 模块，实现了批量 ELF 文件处理和链接脚本生成功能，优化了开发流程。支持在构建期间将用户文件链接到内核，为嵌入式、无特权级场景提供了更简便的程序加载方式。

2.2 团队贡献概述

本开发团队的成员在以下关键领域做出了主要贡献：

- **系统调用实现**：初始提交实现了基本的系统调用框架，奠定了用户态交互的基础。
- **内存管理优化**：修复了跨区域 `mmap` 问题，并优化了内存映射对齐，提升了内存管理的健壮性。同时，通过引入 RAII 思想重构了内存区域和页表管理，显著提升了内存安全性和健壮性。
- **线程支持**：新增了用户态线程支持功能，使系统具备了并发处理能力。
- **信号处理**：实现了默认信号处理器，并完善了信号处理机制，增强了系统的容错性。
- **构建工具**：开发了 `acbat` 模块，用于批量处理 ELF 文件，提高了开发效率。
- **用户态程序支持**：实现了加载程序并在用户态运行的基本功能，验证了宏内核架构的有效性。

这些改进使得 AstrancE 在保持 ArceOS 模块化优势的基础上，具备了运行传统用户态应用的能力，为操作系统的实际应用提供了坚实的基础。

2.3 个人贡献分工

曾熙晨：

- 主导内存管理相关优化，包括修复跨区域 `mmap` 问题（`3b693df`），完善内存映射对齐、brk 机制、默认信号处理器等（`060c8d8`）。

- 实现了用户态线程支持（9227965），扩展了系统的并发能力。
- 维护和更新依赖（如 axns、lwext4_rust），保障主干代码的可用性和兼容性。
- 参与系统日志、构建脚本、依赖管理等基础设施的维护（如 d25a4dc、d23c8f8）。

滕奇勋：

- 主要负责分支管理、代码合并与集成，推动多项功能的主干同步（如多次 Merge pull request）。
- 参与资源 API 和文件/目录 inode xattr 支持的开发与合并（1b72401）。
- 参与系统调用相关模块的调试与维护。

杨纯懿：

- 负责 futex（用户态同步原语）功能的实现（7461639），增强了多线程同步能力。
- 参与 busybox 兼容性修复、文件系统接口完善（baae0c3），提升了系统对实际应用的支持。
- 参与 lua 支持、用户态接口完善、代码合并与分支管理（如 369ab93、3006515）。
- 参与 axmono 用户态支持相关代码的开发与维护（如 435472a）。

3 进程管理

进程与线程管理是现代操作系统的核心子系统之一，直接关系到系统的并发能力、资源隔离、用户程序支持与整体可扩展性。本项目在 ArceOS 的基础上，采用高度模块化的设计思路，将底层任务调度、进程/线程树管理与用户态扩展功能解耦，实现了灵活、可插拔的进程与线程支持体系。具体而言，底层运行管理依赖 ArceOS 的 axtask（unikernel 任务管理）模块，进程/线程树结构与生命周期管理依赖 axprocess（上游模块），而我们团队自研的 axmono 则负责用户态进程/线程的建模、资源隔离、信号处理、系统调用以及 futex 等“类 Linux”功能。

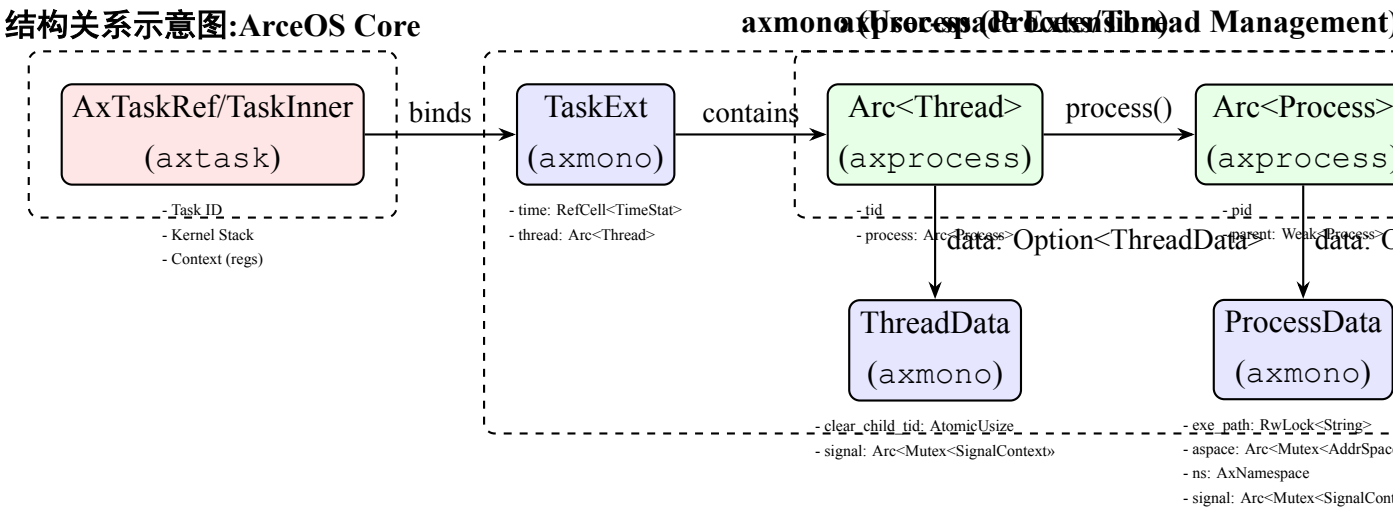
3.1 整体架构与模块关系

本系统的进程与线程管理采用三层解耦架构，各模块职责明确：

- **axtask**：负责所有任务（包括内核线程、用户进程/线程）的调度、上下文切换、CPU 绑定等底层运行管理，保证了高效与可移植性。
- **axprocess**：负责进程/线程树、进程组、会话等 POSIX 语义的建模，便于多进程/多线程的逻辑管理与生命周期维护。

- **axmono**：在上述基础上，独立实现了用户态进程/线程的建模、虚拟地址空间管理、信号处理、命名空间、系统调用、`futex`、`/proc` 文件系统等功能，极大提升了系统的可扩展性和与现有用户态应用的兼容性。

结构关系示意图：



如上图所示，每个内核任务（`AxTaskRef/TaskInner`）通过 `TaskExt` 结构体绑定一个 `Arc<Thread>` 实例。`Thread/Process` 对象由 `axprocess` 模块管理其树形结构，而 `ThreadData/ProcessData` 则由 `axmono` 模块挂载，负责承载用户态相关资源和扩展数据。

3.2 用户态进程与线程的建模与管理

3.2.1 TaskExt 结构体与扩展绑定

每个 `axtask` 任务通过 `TaskExt` 结构体进行功能扩展，该结构体持有对 `Arc<Thread>` 的引用，从而允许内核任务直接访问线程和进程的扩展数据。这实现了内核任务与用户态进程/线程之间的关键桥接。核心代码示例如下：

```rust
pub struct TaskExt {
    pub time: RefCell<time::TimeStat>,
    pub thread: Arc<Thread>,
}

impl TaskExt {
    pub fn new(thread: Arc<Thread>) -> Self {
        // ... implementation details ...
    }
```

```
10
11      pub fn thread_data(&self) -> &ThreadData {
12          self.thread.data().unwrap()
13      }
14
15      pub fn process_data(&self) -> &ProcessData {
16          self.thread.process().data().unwrap()
17      }
18  }
```

3.2.2 ProcessData 与 ThreadData 结构体设计与阐述

ProcessData 结构体用于描述进程级的运行时信息，其主要字段及其作用如下：

- `exe_path: RwLock<String>`：进程可执行文件的路径，支持并发读写，确保路径信息的安全访问。
- `aspace: Arc<Mutex<AddrSpace>>`:进程独立的虚拟地址空间,通过`Arc<Mutex>`实现多线程安全访问。
- `ns: AxNamespace`：进程的资源命名空间，用于实现文件描述符、当前工作目录等资源的隔离。
- `child_exit_wq: WaitQueue`：子进程退出等待队列，用于实现父子进程间的同步与子进程的资源回收。
- `exit_signal: Option<Signal>`:进程退出时向其父进程发送的信号类型,遵循 POSIX 规范。
- `signal: Arc<Mutex<SignalContext>>`:进程级信号管理器,支持 POSIX 信号的注册、屏蔽、触发与处理。
- `signal_stack: Box<[u8; 4096]>`：信号处理专用栈，用于处理信号时提供独立的栈空间，提升信号处理的健壮性。

ThreadData 结构体用于描述线程级的运行时信息，其主要字段及其作用如下：

- `clear_child_tid: AtomicUsize`:线程退出时自动清零的内存地址,与`futex`机制配合，实现线程间的同步与资源回收。
- `signal: Arc<Mutex<SignalContext>>`：线程级信号管理器，支持线程私有的信号处理行为。

相关代码示例：

```rust
pub struct ProcessData {
    pub exe_path: RwLock<String>,
    pub aspace: Arc<Mutex<AddrSpace>>,
    pub ns: AxNamespace,
    pub child_exit_wq: WaitQueue,
    pub exit_signal: Option<Signal>,
    pub signal: Arc<Mutex<SignalContext>>,
    pub signal_stack: Box<[u8; 4096]>,
}

pub struct ThreadData {
    pub clear_child_tid: AtomicUsize,
    pub signal: Arc<Mutex<SignalContext>>,
}
```

3.2.3 用户进程/线程的创建与生命周期管理

用户进程/线程的创建流程遵循以下步骤：

1. 通过 `spawn_user_task` 函数创建底层的 `axtask` 任务（`TaskInner`），并配置其页表根指针，指向新建的虚拟地址空间。
2. 构建 `ProcessData` 实例，并通过 `axprocess::Process` 模块建立相应的进程节点，将其纳入进程树管理。
3. 构建 `ThreadData` 实例，并通过 `axprocess::Thread` 模块建立相应的线程节点，将其纳入线程树管理。
4. 通过 `TaskExt::new(thread)` 方法，将新创建的 `axtask` 任务与对应的线程进行绑定，实现内核任务与用户态线程之间的桥接。
5. 进程/线程的生命周期管理（包括创建、终止、等待、资源回收等）均通过 `axmono` 提供的接口与 `axprocess` 的树形结构协同完成。

核心代码片段：

```rust
pub fn spawn_user_task(
    exe_path: &str,
```

```
    aspace: Arc<Mutex<AddrSpace>>,
    uctx: UspaceContext,
    pwd: String,
    parent: Option<Arc<Process>>,
) -> AxTaskRef {
    let mut task = spawn_user_task_inner(exe_path, uctx, pwd,
    ↪ None);
    task.ctx_mut().set_page_table_root(aspace.lock().page_table_r
    ↪ oot());
    let tid = task.id().as_u64() as Pid;
    let process_data = ProcessData::new(exe_path.into(), aspace,
    ↪ spawn_signal_ctx(), None);
    let parent = parent.unwrap_or(init_proc());
    let process = parent.fork(tid).data(process_data).build();
    let thread_data = ThreadData {
        // ... initialization details ...
    };
    let thread =
    ↪ process.new_thread(tid).data(thread_data).build();
    task.init_task_ext(TaskExt::new(thread));
    task.task_ext().process_data().ns_init_new();
    task.into_arc()
}
```

3.3 模块化与复用优势

本系统充分利用 ArceOS 的模块化设计理念，将底层任务调度（axtask）、进程/线程树管理（axprocess）与用户态扩展（axmono）进行了严格解耦。具体而言，axtask 负责所有任务的运行管理，axprocess 负责维护进程/线程的树形结构与生命周期，而 axmono 则专注于提供用户态资源、信号处理、系统调用以及同步等高级功能。三者通过 TaskExt 结构体实现高效桥接与协作，这不仅保证了底层调度的高性能与可移植性，也极大提升了用户态支持的灵活性和可扩展性。

3.4 创新点与实际成效

- **独立的用户态功能实现：**独立实现了用户态进程/线程的建模、资源隔离、信号处理、系统调用、同步机制等关键能力，极大拓展了 ArceOS 的应用场景和系统能力边界。

- **高效模块解耦与协作：**通过 `TaskExt` 结构体实现了 `axtask`、`axprocess`、`axmono` 三个核心模块之间的高效解耦与协作，体现了良好的工程设计与模块复用能力。
- **用户态扩展集中管理：**进程/线程的所有用户态扩展功能均集中在 `axmono` 模块中，这极大地简化了后续的维护工作，并为未来功能的拓展提供了清晰的路径。

小结

：本项目的进程与线程管理子系统，兼具结构清晰、模块解耦、功能丰富、易于扩展等优点，为系统的多进程/多线程支持和用户态应用生态打下了坚实基础。

4 内存管理 **

4.1 物理内存管理

页帧分配与管理 我们通过全局分配器 `axalloc` 实现了对物理页帧（4KB 单位）的高效分配与回收，支持按需清零。每个页帧均由 `FrameTrackerImpl` 统一封装，借助自动引用计数（`Arc`）和 RAII 机制保证资源正确回收，避免内存泄漏。具体实现如下：

```rust
pub struct FrameTrackerImpl {
    pub pa: PhysAddr,
    tracking: bool,
}
impl FrameTracker for FrameTrackerImpl {
    const PAGE_SIZE: usize = PAGE_SIZE_4K;
    fn new(pa: PhysAddr) -> Self {
        Self { pa, tracking: true }
    }
    fn no_tracking(pa: PhysAddr) -> Self {
        Self { pa, tracking: false }
    }
    fn alloc_frame() -> Self {
        panic!("frame should be allocated by Backend::map")
    }
    fn start(&self) -> PhysAddr {
        self.pa
    }
    fn dealloc_frame(&mut self) {
        if self.tracking {
```

```
            dealloc_frame(self.pa);
        }
    }
}
impl Drop for FrameTrackerImpl {
    fn drop(&mut self) {
        self.dealloc_frame();
    }
}
pub type FrameTrackerRef = alloc::sync::Arc<FrameTrackerImpl>;
```

FrameTrackerImpl 的生命周期由 Rust 的所有权系统自动管理，确保物理内存的安全稳定使用。

4.2 虚拟地址空间构建与管理

地址空间（AddrSpace）抽象 每个进程或内核线程拥有独立的 AddrSpace 实例，内部维护完整的页表（PageTable）和内存区域集合（MemorySet）。AddrSpace 支持空白创建、克隆、线性映射、用户段映射、权限管理等操作。核心结构如下：

```
pub struct AddrSpace {
    va_range: VirtAddrRange,
    pub areas: MemorySet<Backend>,
    pub(crate) pt: PageTable,
    #[cfg(feature = "heap")]
    pub(crate) heap: Option<HeapSpace>,
}
```

内存区域（MemoryArea）建模 虚拟地址空间被划分为多个逻辑区域（MemoryArea），如代码段、数据段、堆、栈和 mmap 区域。每个区域独立管理访问权限和映射策略，实现了高度灵活的内存隔离与安全保障。区域的底层实现支持多种映射后端（线性映射、动态分配、设备映射等），便于扩展和维护。

4.3 页帧生命周期管理与写时复制（COW）

自动资源管理 页帧与 FrameTrackerImpl 绑定，依赖 RAII 与引用计数自动管理生命周期，极大降低资源泄漏风险，保障内存安全。

写时复制机制 为优化进程复制和内存共享，我们实现了高效的写时复制（COW）机制。多个地址空间可共享同一物理页（只读），写入时触发 COW 异常，自动分配新页并复制数据，页表及时更新，保证进程间数据隔离与一致性。核心逻辑如下：

```
pub(crate) fn handle_page_fault_cow(
    vaddr: VirtAddr,
    orig_flags: MappingFlags,
    aspace: &mut AddrSpace,
) -> bool {
    if !orig_flags.contains(MappingFlags::WRITE) {
        return false;
    }
    if let Ok((_, pte_flag, _)) = aspace.pt.query(vaddr) {
        if !pte_flag.contains(MappingFlags::COW) {
            return false;
        }
    } else {
        return false;
    }
    let origin = if let Some(origin) =
    ↪ aspace.find_frame(vaddr.align_down_4k()) {
        origin
    } else {
        return false;
    };
    let count = Arc::strong_count(&origin) - 1;
    let origin_pa = origin.pa;
    if count == 1 {
        return aspace
            .page_table()
            .remap(vaddr, origin_pa, orig_flags)
            .map(|(_, tlb)| tlb.flush())
            .is_ok();
    }
    if let Some(frame) = alloc_frame(false) {
        unsafe {
            core::ptr::copy_nonoverlapping(
                vaddr.align_down_4k().as_ptr(),
                phys_to_virt(frame.pa).as_mut_ptr(),
```

```
                PageSize::Size4K.into(),
            )
        };
        return aspace.remap(vaddr, frame, orig_flags);
    }
    false
}
```

4.4 缺页异常与懒惰分配机制

懒分配策略 我们采用懒惰分配策略，创建内存区域时不立即分配物理页，首次访问触发缺页异常，延迟分配并建立映射。缺页处理函数 `handle_page_fault_alloc` 完成物理页分配与映射修正：

```
pub(crate) fn handle_page_fault_alloc(
    vaddr: VirtAddr,
    va_type: VmAreaType,
    orig_flags: MappingFlags,
    aspace: &mut AddrSpace,
    populate: bool,
) -> bool {
    if populate {
        // COW 处理
        #[cfg(feature = "COW")]
        return match va_type {
            VmAreaType::Normal =>
            ↪  Self::handle_page_fault_cow(vaddr, orig_flags,
            ↪  aspace),
            _ => false,
        };
    }
    match va_type {
        VmAreaType::Normal => {
            if let Some(frame) = alloc_frame(true) {
                return aspace
                    .page_table()
                    .map(vaddr, frame.pa, PageSize::Size4K,
                    ↪  orig_flags)
```

```
                    .map(|tlb| tlb.flush())
                    .and_then(|| {
                        aspace.areas.insert_frame(vaddr,
                        ↪ frame.clone());
                        Ok(())
                    })
                    .is_ok();
            }
            return false;
        }
        // 其他类型略
        _ => false,
    }
}
```

异常处理流程 缺页异常由 trap 子系统统一捕获，处理流程包括：CPU 访问未映射地址触发异常，Trap Handler 接管并调用内存管理模块判断异常类型，完成写时复制或懒分配，更新页表后恢复用户态执行，确保程序透明运行。

4.5 协同机制：缺页与 COW 联动

写时复制与懒分配机制协同工作，既保证了数据隔离，又实现了资源高效复用。COW 区域通过首次写入触发复制，避免不必要的内存浪费；懒分配区域通过首次访问动态分配，最大程度减少空闲资源占用。

4.6 动态堆与 mmap 区域管理

堆空间 我们通过 `AddrSpace::init_heap()` 完成用户堆区初始化，支持按需增长、懒分配和页粒度扩展，满足用户程序的动态内存需求：

```
pub fn init_heap(&mut self, heap_bottom: VirtAddr, max_size:
↪ usize) {
    assert!(self.heap.is_none(), "heap is already initialized");
    let heap = HeapSpace::new(heap_bottom, max_size);
    self.map_alloc(
        heap.base(),
        PAGE_SIZE_4K,
        MappingFlags::READ | MappingFlags::WRITE |
        ↪ MappingFlags::USER,
```

```
        true,
    ).expect("heap mapping failed");
    self.heap = Some(heap);
}
```

内存映射支持 我们支持类 Linux 风格的 mmap 机制，允许用户空间灵活绑定匿名内存或文件映射。针对 mmap 机制，我们做了如下创新和修复：

- **跨区域映射支持**：修复了 mmap 跨不同内存区域的映射问题，提升了 mmap 的灵活性和健壮性（见 3b693df 提交，aspace/mmap.rs）。
- **共享内存映射（shm_mmap）**：实现了进程间共享内存的映射与管理，支持多进程协作。
- **权限与对齐优化**：完善了 mmap 区域的权限管理和对齐策略，提升了安全性和兼容性。

相关接口如下：

```
pub fn mmap(
    &mut self,
    start: VirtAddr,
    size: usize,
    perm: MmapPerm,
    flags: MmapFlags,
    mmap_io: Arc<dyn MmapIO>,
    populate: bool,
) -> AxResult<VirtAddr> { /* 详见 aspace/mmap.rs */ }
pub fn shm_mmap(
    &mut self,
    start: VirtAddr,
    size: usize,
    perm: MmapPerm,
    flags: MmapFlags,
    shm_segment: Arc<Mutex<ShmSegment>>,
    populate: bool,
) -> AxResult<VirtAddr> { /* 详见 aspace/mmap.rs */ }
```

4.7 RAII 与自动化内存管理

我们在内存管理子系统中引入了 RAII（资源获取即初始化）思想，重写了页帧、内存区域等关键结构的生命周期管理。通过 Rust 的所有权和引用计数机制，自动管理内存资源的分配与释放，极大提升了内存安全性和健壮性。例如：

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

4.8 模块化与 feature flags

功能与模块化设计 `axmm` 作为 ArceOS 的虚拟内存管理核心模块，支持多种内存管理策略和高级特性。其 features 及依赖关系如下：

- **RAII**：资源自动管理（依赖 `memory_set/RAII`、`memory_addr/RAII`）
- **COW**：写时复制（Copy-On-Write），依赖 `RAII`
- **heap**：堆内存管理
- **mmap**：内存映射（依赖 `memory_set/mmap`）

源码示例:

```
[features]
RAII = ["memory_set/RAII", "memory_addr/RAII"]
COW = ["RAII"]
heap = []
mmap = ["memory_set/mmap"]
```

说明 `RAII` 控制内存映射等资源的自动释放与生命周期管理，`COW` 控制虚拟内存的写时复制机制，是现代操作系统高效进程/线程隔离的关键。通过 feature flags，可以灵活裁剪内存管理能力，适应不同场景需求。=

小结

本系统的内存管理子系统在可扩展性、效率与健壮性方面表现突出，具体体现为：采用统一的页帧管理机制，结合引用计数与 RAII 实现自动资源释放；通过灵活的地址空间抽象，支持内核线性映射和用户态多段映射；集成高效的写时复制与懒惰分配策略，有效降低进程复制和内存分配开销；构建完整且与 trap 框架紧密协作的异常处理路径；并提供面向应用需求的动态堆区与 mmap 管理能力，兼顾运行时动态性与系统安全性。整体来看，该内存系统不仅为用户态应用构建了高效、隔离且可控的执行环境，也为未来引入共享内存、匿名映射、内存回收等高级内存管理功能奠定了坚实的基础。

5 文件系统

5.1 虚拟文件系统架构设计

本节详细介绍 ArceOS 虚拟文件系统（VFS）的整体架构与核心实现。VFS 作为操作系统文件管理的中枢，承担着屏蔽底层多种文件系统差异、统一文件与设备访问接口、支持多种节点类型（如常规文件、目录、符号链接、设备节点等）的关键任务。我们以 `axfs_vfs` crate 为核心，围绕统一的 trait 接口进行抽象和模块化设计，极大提升了系统的可扩展性与可维护性。VFS 通过标准化的接口定义，将所有文件实体统一建模为 `VfsNode`，并分别通过 `VfsOps` 和 `VfsNodeOps` 两套 trait 抽象文件系统级与节点级操作。所有通用文件操作逻辑集中于 fops 层，便于上层用户态透明访问和底层实现的灵活替换。

整体功能与设计理念 ArceOS 的 VFS 设计目标是实现“统一抽象、灵活扩展、透明访问”。其核心功能包括：

- 支持多种后端文件系统（如 FAT、EXT4、RAMFS、DEVFS 等）的无缝挂载与访问；
- 统一建模所有文件系统节点（文件、目录、设备等），并通过 trait 机制实现多态分发；
- 提供标准化的文件系统与节点操作接口，便于扩展新型文件系统或自定义节点类型；
- 路径解析、权限管理、属性查询等通用逻辑高度解耦，便于维护与功能增强。

5.2 原有基础上的修改

1. 统一接口抽象与 trait 设计 ArceOS 以 trait 机制为核心，定义了 `VfsOps` 和 `VfsNodeOps` 两套接口，分别对应文件系统级和节点级操作。具体定义如下：

```rust
/// 文件系统操作接口
pub trait VfsOps: Send + Sync {
    /// 挂载时的操作
    fn mount(&self, _path: &str, _mount_point: VfsNodeRef) -> VfsResult { ... }
    /// 卸载时的操作
    fn umount(&self) -> VfsResult { ... }
    /// 格式化文件系统
    fn format(&self) -> VfsResult { ... }
    /// 查询文件系统属性
    fn statfs(&self, _path: *const c_char, fs_info: *mut FileSystemInfo) -> VfsResult<usize> { ... }
    /// 获取根目录节点
    fn root_dir(&self) -> VfsNodeRef;
}

/// 节点（文件/目录）操作接口
pub trait VfsNodeOps: Send + Sync {
    /// 打开节点
    fn open(&self) -> VfsResult { ... }
    /// 关闭节点
    fn release(&self) -> VfsResult { ... }
    /// 查询节点属性
    fn get_attr(&self) -> VfsResult<VfsNodeAttr> { ... }
    /// 读写、同步、截断等文件操作
    fn read_at(&self, _offset: u64, _buf: &mut [u8]) -> VfsResult<usize> { ... }
    fn write_at(&self, _offset: u64, _buf: &[u8]) -> VfsResult<usize> { ... }
    fn fsync(&self) -> VfsResult { ... }
    fn truncate(&self, _size: u64) -> VfsResult { ... }
    /// 目录相关操作
    fn parent(&self) -> Option<VfsNodeRef> { ... }
    fn lookup(self: Arc<Self>, _path: &str) -> VfsResult<VfsNodeRef> { ... }
    fn create(&self, _path: &str, _ty: VfsNodeType) -> VfsResult { ... }
    fn remove(&self, _path: &str) -> VfsResult { ... }
    fn read_dir(&self, _start_idx: usize, _dirents: &mut [VfsDirEntry]) -> VfsResult<usize> { ... }
```

```
    fn rename(&self, _src_path: &str, _dst_path: &str) ->
    ↪  VfsResult { ... }
    /// 类型擦除支持
    fn as_any(&self) -> &dyn core::any::Any { ... }
}
```

改进点：

- **节点属性扩展：**我们在 `VfsNodeOps` 中新增了 `get_attr_x`、`set_atime`、`set_mtime`、xattr（扩展属性）等接口，支持更丰富的元数据管理，便于兼容 POSIX/EXT4 等高级特性。
- **类型安全与多态：**通过 `Arc<dyn VfsNodeOps>` 统一节点引用，支持运行时多态分发，便于后端文件系统的灵活接入与替换。
- **宏辅助实现：**提供 `impl_vfs_dir_default!` 和 `impl_vfs_non_dir_default!` 宏，自动为目录/非目录节点补全默认实现，减少重复代码，提高开发效率。

2. 节点与属性结构体设计 所有文件系统节点统一建模为 `VfsNode`，其属性通过 `VfsNodeAttr` 和 `VfsNodeAttrX` 结构体描述，支持标准权限、类型、大小、时间戳等字段。例如：

```
#[derive(Debug, Clone, Copy)]
pub struct VfsNodeAttr {
    dev: u64,                   // 设备号
    mode: VfsNodePerm,          // 权限
    ty: VfsNodeType,            // 节点类型
    size: u64,                  // 文件大小
    blocks: u64,                // 占用块数
    st_ino: u64,                // inode 号
    nlink: u32,                 // 硬链接数
    uid: u32, gid: u32,         // 所有者/组
    ...
    atime:u32, ctime:u32, mtime:u32, // 时间戳
    ...
}
```

改进点：

- **属性字段丰富：**支持完整的 Unix/POSIX 属性字段，兼容多种后端文件系统需求。

- **权限与类型枚举：**通过 `VfsNodePerm`（bitflags）和 `VfsNodeType`（枚举）实现权限与类型的高效表达与判断。
- **扩展属性支持：** `VfsNodeAttrX` 结构体进一步支持 statx 扩展字段，便于高级元数据管理。

3. 路径与节点操作的统一分发 所有文件系统操作均通过 trait 接口分发，路径解析、节点查找、文件读写等操作高度解耦。例如，路径查找通过 `lookup` 接口实现，支持最长前缀匹配与多层挂载点分派，极大提升了挂载灵活性与访问一致性。

模块划分:

- `lib.rs`：核心 trait、类型定义与接口实现。
- `structs.rs`：节点、属性、权限等结构体定义。
- `macros.rs`：辅助宏定义，简化 trait 实现。
- `path.rs`：路径规范化与解析工具函数。

5.3 典型调用流程图（建议配图）

- 用户态发起文件操作（如 open/read/write）→ VFS 层统一分发 → 根据路径查找对应挂载点与节点 → 调用后端文件系统实现的 trait 方法完成实际操作。

5.4 procfs：动态内核信息文件系统

我们全新实现了 procfs 子系统，支持以文件系统方式导出内核与系统运行时信息。procfs 的核心在于其节点（文件/目录）可在运行时动态生成，极大提升了内核信息导出的灵活性和扩展性。

```
pub struct ProcFileSystem {
    parent: Once<VfsNodeRef>,
    root: Arc<ProcDir>,
}
impl VfsOps for ProcFileSystem {
    fn mount(&self, _path: &str, mount_point: VfsNodeRef) ->
     ↪  VfsResult { ... }
    fn root_dir(&self) -> VfsNodeRef { self.root.clone() }
}
```

核心结构与接口

`ProcDir` 作为procfs 的目录节点，支持静态与动态子节点的混合管理。通过 `add_generator` 方法可为目录注册任意数量的动态生成器函数，每次目录遍历或查找时自动调用，动态生成如 `/proc/version`、`/proc/cpuinfo` 等节点。

```
pub type ProcDirGenerator = dyn Fn() -> VfsResult<Vec<(String,
↪  ProcEntry)>> + Send + Sync;
pub struct ProcDir { ... generators:
↪  RwLock<Vec<Arc<ProcDirGenerator>>>, ... }
impl ProcDir {
    pub fn add_generator(&self, generator: Arc<ProcDirGenerator>)
    ↪  { ... }
    pub fn lookup_entry(&self, path: &str) ->
    ↪  VfsResult<ProcEntry> { ... }
}
```

动态文件支持 procfs 支持静态文件（`ProcFile`）和动态文件（`ProcDynamicFile`）。动态文件通过回调函数在每次读取时生成内容，适合导出如内核版本、进程状态等实时信息。

```
pub type ProcFileGenerator = dyn Fn(u64, &mut [u8]) ->
↪  VfsResult<usize> + Send + Sync;
pub struct ProcDynamicFile { generator:
↪  RwLock<Arc<ProcFileGenerator>>, }
impl VfsNodeOps for ProcDynamicFile {
    fn read_at(&self, offset: u64, buf: &mut [u8]) ->
    ↪  VfsResult<usize> {
        (self.generator.read())(offset, buf)
    }
}
```

测试覆盖 procfs 支持静态/动态文件创建、目录遍历、路径查找、节点删除等完整操作，详见 `axfs_procfs/src/tests.rs`。

5.5 动态挂载机制与多文件系统协同

我们对 VFS 挂载机制进行了显著扩展，实现了多文件系统的动态挂载与路径分派。系统支持在任意路径下挂载不同类型的文件系统（如 FAT、EXT4、RAMFS、DEVFS、PROCFS 等），并通过最长前缀匹配策略自动分派路径请求到对应挂载点。

路径查找与分派 路径查找过程中，VFS 会遍历所有挂载点，查找与请求路径最长匹配的挂载点，并将后续操作转发至对应文件系统。例如：

```
fn lookup_mounted_fs(path: &str) -> (MountPoint, &VfsOps) { ... }
fn find_mountpoint_and_fs(path: &str) -> Option<(&str, &VfsOps)>
 ↪  { ... }
```

挂载流程

1. 启动时自动挂载根文件系统（如 EXT4）至 /；
2. 特殊目录如 /dev、/tmp、/proc 分别挂载 DEVFS、RAMFS、PROCFS；
3. 用户或内核可在运行时动态挂载/卸载任意文件系统至指定路径，实现多后端协同。

路径规范化 所有路径操作均通过 axfs_vfs::path::canonicalize 工具函数进行规范化，确保路径查找的一致性和健壮性。

5.6 DEVFS：设备文件系统的动态扩展

我们对 devfs 进行了深度拓展，实现了设备节点的动态注册、管理与访问。devfs 作为 VFS 的一类特殊后端，允许内核设备以文件形式被统一访问，极大提升了设备虚拟化能力和系统一致性。

```
pub struct DeviceFileSystem {
    parent: Once<VfsNodeRef>,
    root: Arc<DirNode>,
}
impl DeviceFileSystem {
    pub fn add(&self, name: &'static str, node: Arc<dyn
     ↪  VfsNodeOps>) { self.root.add(name, node); }
```

```
    pub fn mkdir(&self, name: &'static str) -> Arc<DirNode> {
     ↪  self.root.mkdir(name) }
}
```

核心结构与接口

设备节点实现 每个设备节点（如 NullDev、ZeroDev、CharDeviceNode）均实现 VfsNodeOps 接口，具备标准文件节点的操作语义。以 /dev/null 为例：

```
pub struct NullDev;
impl VfsNodeOps for NullDev {
    fn read_at(&self, _offset: u64, _buf: &mut [u8]) ->
     ↪  VfsResult<usize> { Ok(0) }
    fn write_at(&self, _offset: u64, buf: &[u8]) ->
     ↪  VfsResult<usize> { Ok(buf.len()) }
    // ...
}
```

动态注册与管理 devfs 支持在运行时通过 add 接口动态注册新设备节点，支持多级目录结构和自定义字符设备：

```
let devfs = DeviceFileSystem::new();
devfs.add("null", Arc::new(NullDev));
devfs.add("zero", Arc::new(ZeroDev));
let mychr = Arc::new(CharDeviceNode::new("mychr"));
devfs.add("mychr", mychr);
```

线程安全与高效查找 devfs 目录结构基于 BTreeMap 和 RwLock 实现，支持多核环境下的高效并发访问和节点查找。

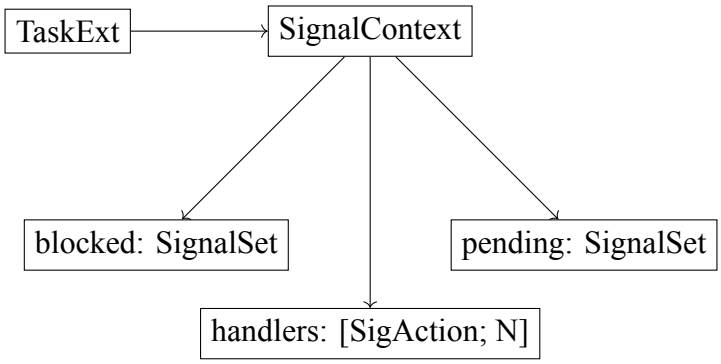
小结

通过对 procfs、动态挂载机制和 devfs 的持续拓展，ArceOS 的 VFS 架构不仅实现了对多种文件系统的统一抽象，还极大提升了系统的灵活性、可扩展性和工程实践能力。无论是内核信息导出、设备虚拟化，还是多后端文件系统协同，均可通过标准接口和模块化设计高效实现，充分体现了现代操作系统文件系统架构的先进理念。

6 信号处理 *

6.1 总体功能与设计目标

AstrancE 的信号子系统完全由本团队自主设计实现，旨在为内核和用户态提供高效、灵活、POSIX 兼容的异步事件通知与进程控制能力。该子系统不仅支持标准信号的注册、屏蔽、递送、处理和上下文恢复，还实现了备用栈、优先级调度、信号集批量操作等高级特性，极大提升了系统的健壮性和可维护性。信号机制作为内核与用户态、进程间通信（IPC）以及任务调度的桥梁，是现代操作系统不可或缺的基础设施。



6.2 关键数据结构与类型设计

信号类型枚举 `Signal` 信号类型采用 Rust 枚举实现，涵盖 POSIX 标准信号，便于类型安全和扩展。每个信号均有唯一编号，支持与系统调用参数的安全转换。例如：

```
#[repr(i32)]
#[derive(Debug, Copy, Clone, PartialEq, Eq)]
pub enum Signal {
SIGBLOCK = SIG_BLOCK,
SIGUNBLOCK = SIG_UNBLOCK,
SIGSETMASK = SIG_SETMASK,
SIGHUP = SIGHUP,
SIGINT = SIGINT,
// ... 省略若干 ...
SIGKILL = SIGKILL,
SIGUSR1 = SIGUSR1,
SIGSEGV = SIGSEGV,
// ... 省略若干 ...
SIGSYS = SIGSYS,
SIGUNUSED = SIGUNUSED,
```

```
}
```

通过 `from32` 方法实现信号编号与枚举的安全转换，便于系统调用接口与内核内部统一处理。

信号集合 SignalSet 信号集合采用 `bitflags!` 宏实现，支持高效的批量增删、优先级查询等操作。例如：

```
bitflags! {
pub struct SignalSet :u32 {
const SIG_BLOCK = 1 << Signal::SIGBLOCK as usize;
}
}
impl SignalSet {
pub fn get_one(&self) -> Option<Signal> {
let sig = self.bits().trailing_zeros();
Signal::from_u32(sig)
}
}
```

这种设计使得信号屏蔽、递送等操作可以用位运算高效实现，极大提升了性能。

信号处理动作 SigAction 与处理器 SigHandler 每个信号可配置独立的处理动作，包括默认、忽略、自定义函数：

```
pub enum SigHandler {
Default,
Ignore,
Handler(unsafe extern "C" fn(i32)),
}
pub struct SigAction {
pub handler: SigHandler,
pub mask: SignalSet,
pub flags: i32,
}
```

支持 POSIX 标准的 `sigaction` 语义，允许用户注册自定义信号处理函数，并配置自动屏蔽集和标志。

信号上下文 `SignalContext` 每个线程/任务独立维护信号上下文，包含处理表、屏蔽集、待处理集：

```
#[derive(Default)]
pub struct SignalContext {
pub handlers: [SigAction; NSIG as usize], // 信号处理表
pub blocked: SignalSet, // 被阻塞的信号
pub pending: SignalSet, // 待处理信号
}
```

这种设计保证了信号的线程隔离性和生命周期管理。

6.3 信号递送与处理流程

信号发送与排队 信号通过 `send_signal` 方法递送到目标线程，若未被屏蔽则加入待处理队列：

```
impl SignalContext {
pub fn send_signal(&mut self, sig: Signal) {
// 如果信号未被阻塞，则加入待处理队列
if !self.pending.contains(sig) {
self.pending = self.pending.union(sig);
}
}
}
```

这种机制确保了信号不会丢失，并能在合适时机递送。

信号处理主循环 在 trap 返回、系统调用退出等调度点，调用 `handle_pending_signals` 检查并处理所有待处理信号：

```
pub fn handle_pending_signals() {
let current_task = current();
let tast_ext = current_task.task_ext_mut();
let mut ctx: SignalContext = tast_ext.sigctx;
let curr_sp = tast_ext.uctx.get_sp();
while ctx.has_pending() {
// 找到最高优先级的待处理信号
```

```
let sig = ctx.pending.get_one().unwrap();
let sig_action = &ctx.handlers[sig as usize];
match sig_action.handler {
SigHandler::Default => handle_default_signal(sig, regs),
SigHandler::Ignore => {} // 直接忽略
SigHandler::Handler(handler) => {
// 设置信号处理栈帧
unsafe { enter_signal_handler(&mut ctx, curr_sp, handler, sig) };
}
}
// 清除已处理的信号
ctx.pending.remove(sig);
}
}
```

支持优先级调度，递送时自动判断处理策略，确保信号响应的灵活性和安全性。

信号处理与上下文切换 信号处理时自动构造信号帧，切换到用户注册的处理函数，处理完毕后通过 `sigreturn` 恢复原始上下文：

```
unsafe fn enter_signal_handler(
sigctx: &mut SignalContext,
ustack_top: usize,
sig_action: SigAction,
handler: unsafe extern "C" fn(i32),
sig: i32,
) -> ! {
// ... 省略部分 ...
// 设置返回地址为信号返回 trampoline
regs.ra = sigreturn_trampoline as usize;
// 设置信号屏蔽字
let old_mask = current_task().signal_ctx.blocked;
sigctx.blocked |= sig_action.mask;
frame.saved_mask = old_mask;
unsafe { uctx.enter_uspace(task.get_sig_stack_top()) };
}
```

通过 `sigreturn_trampoline` **汇编桥接，确保信号处理后能安全返回原始执行流。**

6.4 备用栈（Alternate Stack）机制

信号处理过程中，若主栈空间已满或损坏，继续在主栈上递送信号可能导致系统崩溃。为此，AstrancE 信号子系统设计了备用栈（alt stack）机制，确保即使主栈不可用，信号处理也能安全进行。

- **备用栈的管理：**备用栈信息由每个任务的 `SignalContext` 统一管理。信号递送时，系统会判断当前是否需要切换到备用栈，并自动完成切换。
- **信号帧构造时的栈选择：**在 `enter_signal_handler` 函数中，信号帧会被压入备用栈或主栈顶部，具体取决于信号处理的上下文和信号类型。
- **栈切换流程：**信号处理函数的入口参数和返回地址会被正确设置，确保处理完毕后能通过 `sigreturn` 安全返回原始执行流。

代码片段示例:

```
unsafe fn enter_signal_handler(
sigctx: &mut SignalContext,
ustack_top: usize,
sig_action: SigAction,
handler: unsafe extern "C" fn(i32),
sig: i32,
) -> ! {
// ... 省略部分 ...
// 设置用户处理函数上下文，栈接着原来的用户栈
let uctx = UspaceContext::new(handler as usize, ustack_top, sig
 ↪  as usize);
// 跳转到处理函数
uctx.set_ip(handler as usize);
uctx.sepc = handler as usize;
// 设置返回地址为信号返回 trampoline
regs.ra = sigreturn_trampoline as usize;
// ... 省略部分 ...
unsafe { uctx.enter_uspace(task.get_sig_stack_top()) };
}
```

其中 `task.get_sig_stack_top()` 会根据当前信号处理是否需要备用栈，返回备用栈或主栈的栈顶地址。

6.5 与调度器（Scheduler）和 Trap 的集成

信号递送的时机与任务调度、Trap 处理密切相关。AstrancE 通过如下机制实现信号与调度器的无缝集成：

- **Trap 处理钩子：**在 `ulib/axmono/src/task/mod.rs` 中，使用 `register_trap_handler` 宏注册了 PRE_TRAP 和 POST_TRAP 钩子函数。
- **信号递送时机：**在 POST TRAP 阶段（即从内核返回用户态前），会检查当前任务的 `SignalContext`，如有待处理信号则优先递送，确保信号能及时响应。
- **与调度点协同：**在任务切换、系统调用返回等关键调度点，信号递送逻辑会被自动触发，保证信号处理的实时性和一致性。

代码片段

```
#[register_trap_handler(POST_TRAP)]
fn post_trap_handler(trap_frame: &TrapFrame) -> bool {
time_stat_from_kernel_to_user();
// 这里可集成 handle_pending_signals(), 递送信号
true
}
```

同时，信号与任务系统集成。在 `ulib/axmono/src/task/mod.rs` 中，将 `SignalContext` 集成进每个任务（线程）的扩展数据结构 `TaskExt`，实现每线程独立信号上下文。通过 `TaskExt` 的 `sigctx` 字段，支持信号生命周期的全程管理，便于后续扩展线程级信号屏蔽、递送等高级功能。

6.6 代码可维护性与可扩展性

信号机制接口与 POSIX 标准保持一致，支持 `sigaction`、`sigprocmask`、`kill` 等系统调用，便于用户态程序移植。关键系统调用与信号递送点（如 trap 返回、系统调用退出）深度集成，确保信号递送时机的准确性。信号子系统结构设计清晰，便于后续扩展更多信号类型、处理策略及与其他 IPC 机制的集成。采用模块化、接口化设计，便于单元测试和功能验证。

小结

本信号子系统实现了从信号注册、屏蔽、递送、处理到上下文恢复的全流程闭环，具备高性能、强健壮性和良好兼容性。所有核心代码均为本组成员自主开发，充分体现

了团队的系统设计与工程实现能力。本设计为后续进程间通信、事件驱动等高级功能奠定了坚实基础，是 AstrancE 内核自主工作的重要成果之一。

7 用户态支持

本节介绍 AstrancE 用户态运行环境与系统调用支持的整体设计与关键实现。我们以 POSIX 兼容为目标，构建了独立的虚拟地址空间、完善的系统调用分发机制、进程与线程管理、ELF 加载器等基础设施，支持 C 与 Rust 两种开发语言，极大提升了系统的兼容性、可扩展性与工程实践能力。

7.1 axsyscall：宏内核风格的系统调用分发与 POSIX 兼容

我们在 `axsyscall` 模块实现了宏内核架构的系统调用分发机制，支持超过 50 个 POSIX 标准系统调用，涵盖文件操作、进程管理、内存管理、信号、时间、网络等核心功能。系统调用分发表采用宏自动生成，便于维护和扩展。

```
#[macro_export]
macro_rules! syscall_handler_def {
    ($($(#[$attr:meta])* $sys:ident => $args:tt $body:expr
    ↪  $(,)?)*) => {
        #[axhal::trap::register_trap_handler(axhal::trap::SYSCALL
        ↪  )]
        pub fn handle_syscall(tf: &mut axhal::arch::TrapFrame,
        ↪  syscall_num: usize) -> Option<isize> {
            use syscalls::Sysno;
            let args = [tf.arg0(), tf.arg1(), tf.arg2(),
            ↪  tf.arg3(), tf.arg4(), tf.arg5()];
            let sys_id = Sysno::from(syscall_num as u32);
            let result:Option<SyscallResult> = match sys_id {
                $(
                    $(#[$attr])*
                    Sysno::$sys => {
                        Some((|| -> SyscallResult {
                            let $args = args;
                            $body
                        })())
                    }
```

```
                ),*,
                _ => None
            };
            result.map(|r| r.as_isize())
        }
    };
}
```

系统调用分发表与分发流程

POSIX 兼容性与错误码映射 所有系统调用返回值与错误码严格遵循 Linux/POSIX 标准，便于用户程序无缝移植。错误码通过 `LinuxError` 枚举与 `ToLinuxResult` trait 自动转换。

```
impl ToLinuxResult for i32 {
    fn to_linux_result(self) -> SyscallResult {
        if self >= 0 {
            Ok(self as isize)
        } else {
            let code = (-self) as i32;
            Err(LinuxError::try_from(code).unwrap_or(LinuxError::EINVAL))
        }
    }
}
```

系统调用实现示例 以文件操作为例，`axsyscall` 通过调用 `arceos_posix_api` 层的实现，完成 open/read/write/lseek 等标准接口：

```
pub fn sys_openat(dirfd: c_int, filename: *const c_char, flags: c_int, mode: ctypes::mode_t) -> SyscallResult {
    api::sys_openat(dirfd, filename, flags, mode).to_linux_result()
}
pub fn sys_read(fd: usize, buf: &mut [u8]) -> SyscallResult {
    api::sys_read(fd as i32, buf.as_mut_ptr() as *mut c_void, buf.len()).to_linux_result()
```

```
}
```

进程与线程管理 进程控制相关系统调用（如 `fork`、`execve`、`wait4`、`exit`、`clone` 等）通过 axmono 层的统一接口实现，支持多线程与多进程协同。

```
pub fn sys_execve(pathname: usize, argv: usize, envp: usize) ->
↪ LinuxResult<isize> {
    let pathname = char_ptr_to_str(pathname as *const c_char)?;
    let argv: Vec<String> = str_vec_ptr_to_str(argv as *const
    ↪ *const c_char)?.into_iter().map(String::from).collect();
    let envp: Vec<String> = str_vec_ptr_to_str(envp as *const
    ↪ *const c_char)?.into_iter().map(String::from).collect();
    let err = task::exec_current(pathname, &argv,
    ↪ &envp).expect_err("successful execve should not reach
    ↪ here");
    Err(err.into())}
```

7.2 axmono：用户态 ELF 加载、进程与线程支持

ELF 加载与用户程序启动 axmono 模块实现了完整的 ELF 加载器，支持静态与动态链接程序。通过 `load_elf_to_mem` 和 `map_elf_sections`，将用户程序加载到独立的虚拟地址空间，并自动设置入口点、用户栈、辅助向量（auxv）等，兼容 Linux 启动流程。

```
pub fn load_elf_to_mem(
    elf_file: OwnedElfFile,
    uspace: &mut AddrSpace,
    args: Option<&[String]>,
    envs: Option<&[String]>,
) -> AxResult<(VirtAddr, VirtAddr, Option<VirtAddr>)> {
    // 检查是否需要动态链接器
    let interpreter_path = find_interpreter(&elf_file)?;
    if let Some(interp_path) = interpreter_path {
        // 加载解释器并映射
        let interpreter_elf = load_interpreter(&interp_path)?;
        // ...
```

```
        let (interp_entry, ustack_pointer, tp) =
        ↪ map_elf_sections_with_auxv(interp_info, uspace, args,
        ↪ envs, &elf_info)?;
        Ok((interp_entry, ustack_pointer, tp))
    } else {
        // 直接映射主程序
        let elf_info = ELFInfo::new(elf_file, uspace.base(),
        ↪ None, None)?;
        let (entry, ustack_pointer, tp) =
        ↪ map_elf_sections(elf_info, uspace, args, envs)?;
        Ok((entry, ustack_pointer, tp))
    }
}
```

进程与线程创建 每个用户进程拥有独立的 `AddrSpace`，支持多线程（clone/fork）与线程本地存储（TLS）。进程和线程的元数据通过 `ProcessData` 和 `ThreadData` 结构体统一管理，支持信号、命名空间、资源隔离等高级特性。

```
pub struct ProcessData {
    pub exe_path: RwLock<String>,
    pub aspace: Arc<Mutex<AddrSpace>>,
    pub ns: AxNamespace,
    pub child_exit_wq: WaitQueue,
    pub exit_signal: Option<Signal>,
    pub signal: Arc<Mutex<SignalContext>>,
    pub signal_stack: Box<[u8; 4096]>,
}
```

多语言支持与 POSIX API 兼容 系统支持 C 和 Rust 两种开发语言，用户程序可直接调用 POSIX 标准 API。所有系统调用参数与返回值均与 Linux 保持一致，便于主流程序和工具链的无缝移植。

7.3 虚拟地址空间隔离与 copy_from_kernel 安全机制

独立虚拟地址空间与隔离 每个用户进程拥有独立的虚拟地址空间（`AddrSpace`），与内核空间严格隔离。所有内存区域（代码段、数据段、堆、栈、mmap 区等）均配置细粒度权限，仅 USER 区域可被用户态访问。

写时复制（COW）与懒分配 系统支持写时复制（COW）机制，fork/clone 时页面共享，写入时自动分离，极大提升内存利用率。所有内存分配采用懒分配策略，首次访问时动态分配物理页。

copy_from_kernel 方法 为确保用户态与内核态的地址空间隔离，所有内核到用户空间的数据拷贝均通过 `copy_from_kernel` 方法实现，防止越权访问和安全漏洞。

```rust
#[cfg(any(feature = "mm", feature = "process"))]
/// If the target architecture requires it, the kernel portion of the address
/// space will be copied to the user address space.
pub fn copy_from_kernel(aspace: &mut axmm::AddrSpace) -> AxResult {
    use axmm::kernel_aspace;

    if !cfg!(target_arch = "aarch64") && !cfg!(target_arch = "loongarch64") {
        // ARMv8 (aarch64) and LoongArch64 use separate page tables for user space
        // (aarch64: TTBR0_EL1, LoongArch64: PGDL), so there is no need to copy the
        // kernel portion to the user page table.
        aspace.copy_mappings_from(&kernel_aspace().lock(), false)?;
    }
    Ok(())
}
```

7.4 模块化与 feature flags

功能与模块化设计 `arceos_posix_api` 作为 ArceOS 的 POSIX 兼容 API 层，广泛采用 feature flags 实现功能裁剪与模块化。其核心 features 及依赖关系如下：

- **smp**：多核支持（依赖 `axfeat/smp`）
- **irq**：中断支持（依赖 `axfeat/irq`）
- **alloc**：内存分配（依赖 `axalloc`、`axfeat/alloc`）
- **multitask**:多任务/多线程(依赖 `axtask`、`axfeat/multitask`、`axsync/multitask`)

- **fd**：文件描述符（依赖 `alloc`、`axns`）
- **fs**：文件系统（依赖 `axfs`、`axfs_vfs`、`axfeat/fs`、`fd`）
- **net**：网络（依赖 `axnet`、`axfeat/net`、`fd`）
- **pipe**、**select**、**epoll**：管道、IO 多路复用（依赖 `fd`）
- **uspace**：用户空间支持（依赖 `axns/thread-local`）

源码示例：

```
[features]
smp = ["axfeat/smp"]
irq = ["axfeat/irq"]
alloc = ["dep:axalloc", "axfeat/alloc"]
multitask = ["axtask/multitask", "axfeat/multitask",
↪  "axsync/multitask"]
fd = ["alloc", "dep:axns"]
fs = ["dep:axfs", "dep:axfs_vfs", "axfeat/fs", "fd"]
net = ["dep:axnet", "axfeat/net", "fd"]
pipe = ["fd"]
select = ["fd"]
epoll = ["fd"]
uspace = ["axns/thread-local"]
```

说明　这些 flags 控制 API 层是否启用多核、文件系统、网络、管道、内存分配、线程本地存储等功能。通过依赖传递，flags 会影响底层模块的编译与功能裁剪，实现跨 crate 的统一裁剪和模块化构建。

小结

- **宏内核风格的系统调用分发**，支持 50+ POSIX 标准接口，自动化分发表，便于维护与扩展；
- **axmono 用户态支持**，实现 ELF 加载、进程/线程管理、TLS、信号、命名空间等，兼容 C/Rust；
- **独立虚拟地址空间与 copy_from_kernel**，实现用户/内核隔离、COW、懒分配，保障安全与高效；

- **高 POSIX 兼容性**，支持主流 Linux 用户程序无缝移植，接口与错误码完全对齐；
- **模块化与可扩展性**，分层设计，便于新增系统调用、裁剪功能、支持多架构。

整体架构充分体现了现代操作系统用户态支持与系统调用机制的先进理念，为后续功能扩展和工程实践提供了坚实基础。

8 总结与展望

8.1 工作总结

- **宏内核架构与模块化设计：**继承并发扬了 ArceOS 的模块化思想，AstrancE 通过 feature flags 和条件编译机制，实现了内核各子系统（调度、内存、文件、信号、API 等）的灵活裁剪与组合。系统支持多种功能模块的按需启用，极大提升了代码的可维护性和适应性。
- **用户态支持与系统调用机制：**针对 ArceOS 用户态支持薄弱的问题，AstrancE 设计并实现了 axsyscall 宏内核系统调用分发框架，支持 50+ POSIX 标准系统调用，涵盖文件、进程、内存、信号、网络等核心功能。通过 axmono 模块，系统为 C/Rust 用户程序提供了统一的 ELF 加载、进程/线程创建、虚拟地址空间隔离等能力，显著提升了用户态程序的兼容性和运行效率。
- **虚拟内存与内存安全：**以 AddrSpace 为核心，系统实现了独立的虚拟地址空间管理，支持 mmap、写时复制（COW）、共享内存、内存保护等高级特性。通过引入 RAII 机制，结合 Rust 所有权模型，实现了内存资源的自动分配与回收，极大提升了系统的健壮性和安全性。
- **文件系统与设备管理：**构建了统一的 VFS 框架，支持 FAT、EXT4、RAMFS、DEVFS、PROCFS 等多种文件系统的动态挂载与访问。新增 procfs 支持，能够动态导出内核信息；扩展 devfs，支持设备节点的动态注册与管理。系统通过最长前缀匹配实现高效的路径查找与挂载分派，提升了文件系统的灵活性和可扩展性。
- **信号机制与异步事件：**实现了 POSIX 兼容的信号处理机制，支持信号注册、屏蔽、触发、处理、备用信号栈等功能。信号机制与调度器深度融合，保障了异步事件的及时响应和系统的健壮性。
- **工程实践与生态兼容：** AstrancE 支持 C 和 Rust 两种主流开发语言，接口严格遵循 POSIX 和 Linux 标准，便于主流用户程序和工具链的无缝移植。通过分层设计和模块化接口，系统具备良好的工程可维护性和生态适应性。

8.2 经验总结

8.3 未来计划