Starry Mix 设计文档

清华大学 Starry Mix 队

郭士尧 王铮

2025 年 8 月

目录

第 1 章 概述

1.1 简介

Starry Mix 基于 ArceOS 和 StarryOS 项目。ArceOS 是一个 Rust 语言组件化 unikernel 操作系统，而 StarryOS 是在 ArceOS 基础上开发的宏内核操作系统。我们在这两个项目的基础上继续开发了 Starry Mix。

1.2 系统架构

Starry Mix 的架构可以分为四个层级：

1. ArceOS crates: 支撑 ArceOS 的与 OS 无关的组件。该类组件的实现不需要依赖于具体接入的 OS，在用户态、内核态均可以运行。任意内核都可以尝试接入这些组件，而几乎不需要对组件本身的内容进行修改。许多 `#![no_std]` 的 Rust 第三方库都可以作为这样的组件使用，而我们在开发过程中新设计或重新设计了如下的这些组件：

   axbacktrace 基于 DWARF 的栈回溯

   extern-trait 类型擦除的可拓展接口

   lwext4_rust 对 C 语言 lwext4 库的 Rust 封装

   scope-local 作用域共享资源管理

   simple-sdmmc 基于 SDIO 的 SD 卡驱动

   weak-map 弱引用表

2. ArceOS modules: ArceOS 的模块。该类组件的实现与具体接入的 OS 相关，一般涉及到内核的核心功能，很难从一个内核中剥离并给其他内核使用。

   axalloc 内存分配

   axconfig 提供设备树等架构、平台相关配置

   axdisplay 图形显示支持

   axdriver 多种底层驱动（块设备、网络设备、显示设备等）的统一接口

   axhal 硬件抽象层

   axinput 输入设备支持

   axlog 日志打印

   axmm 内存管理

   axnet 基于 smoltcp 的 TCP/IP 协议栈

axruntime 系统初始化

axsync 提供同步原语（Mutex）

axtask 任务调度与管理

此外我们新设计了以下这些模块：

axfs-ng 文件系统

axinput 输入设备

3. Starry crates: 与宏内核相关的组件。这些组件也不依赖于具体的 OS，但是专为在宏内核的内核态中使用而设计，特别是与 POSIX 标准相关。这些组件包括：

   starry-process 会话 / 进程组 / 进程管理和操作

   starry-signal 系统信号处理相关逻辑

   starry-vm 虚拟内存（用户空间）访存

4. Starry modules: Starry Mix 的模块，由以下几个模块通过工作区的形式组成：

   starry-core 宏内核基础功能，包括进程管理，加载用户程序 ELF，虚拟文件系统抽象等

   starry-api 宏内核接口，主要是 POSIX 相关行为，包括文件描述符、系统调用实现、虚拟文件系统实现（procfs、devfs 等）

   starry 宏内核入口，负责加载初始化进程和退出时清理资源

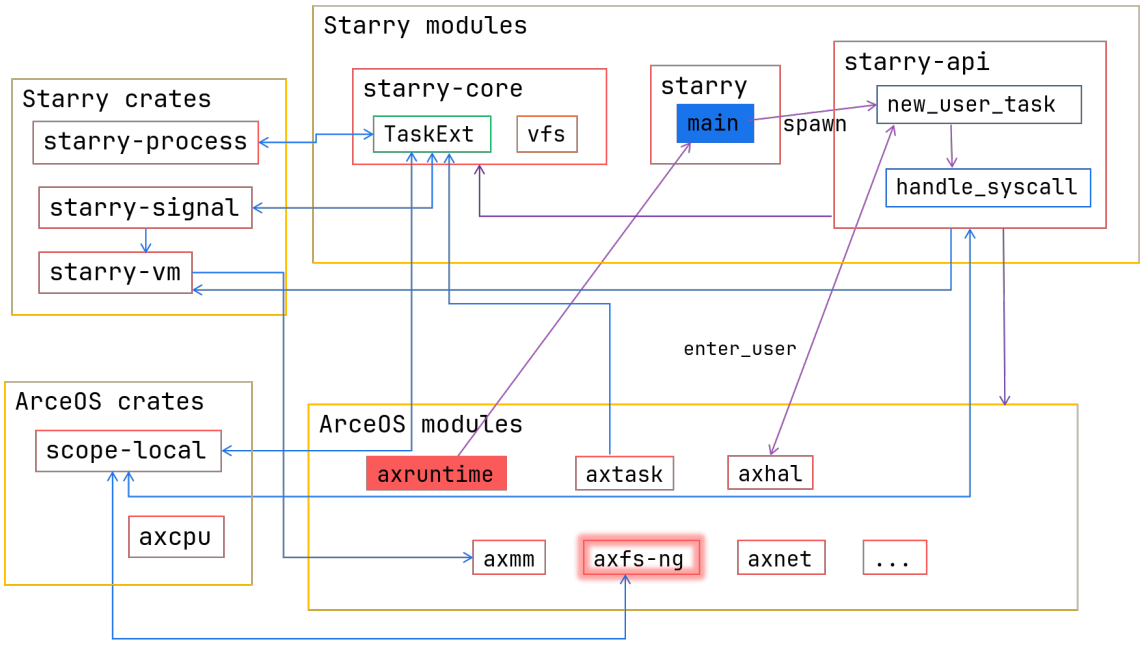

图 1　架构示意图

1.3 完成情况

相对于其他队伍，我们接触比赛与开始开发的时间较晚（今年三月），但依旧取得了较好的完成情况。

Starry Mix 共实现 193 项系统调用，涵盖 **文件系统、IO、内存管理、网络、进程管理** 等多种系统调用类型，能够运行官方测例中的大量 LTP 测例以及 LTP 之外的所有测例。

Starry Mix 各项子模块的完成情况如下表所示：

子模块	完成情况
应用支持	可以运行 **Alpine Linux** 镜像 支持 `rustc`、`gcc`、`nano` 等流行应用 支持 **X11 图形环境**
文件系统	UNIX-Like 的 **VFS 设计** 软/硬链接、套接字、管道 支持 EXT4 与 FAT 文件系统 **页缓存**、**块缓存** 机制加速
内存管理	统一的用户缓存抽象 **懒分配、写时复制** 完善的 **页缓存** 机制，支持文件内存映射
外设支持	**输入设备** 支持（evdev） **显示设备** 支持（fbdev） **TTY** 设备支持
异步机制	基于 `Pollable` 的 **异步同步混合机制** 兼容 UNIX poll API
网络模块	支持 TCP 与 UDP 支持 **Unix Domain Socket** 支持 **cmsg**、**PEERINFO** 等高级特性

表 1　子模块完成情况

第 2 章 组件化设计

ArceOS 为了提升组件可重用性，从操作系统模块中拆分出独立功能单元，并通过 `Cargo.toml` 约束依赖关系，避免循环依赖。然而由于操作系统设计的复杂性，有些时候仍然难以避免一些反向的功能依赖。

为了解决循环依赖关系，ArceOS 设计并使用了 arceos-org/crate_interface，通过类似外部函数接口（FFI）的方式，把上层模块的功能在链接期暴露给下层组件。这种做法虽然解决了部分问题，但是会阻止 Rust 的编译期优化，严重影响性能，需要手动启用链接期优化（LTO）。实际上 Rust 官方也在尝试解决这一问题 rust-lang/rust#125418 中可以看到目前仍处于 “experiment” 阶段。

2.1 extern-trait

crate_interface 仅能够把上层的函数实现暴露给下层，而在开发过程中我们遇到的更复杂的问题是，有些时候甚至需要把上层某个类型提供给下层使用。

通常情况下这一问题有两种解决方案，一种是使用**泛型**，即在下层的数据结构定义中预留一个泛型的位置来存储上层的数据，必要的话还可以通过 trait 对类型进行约束。然而在我们的情况里，涉及到的相关结构体需要存储在全局静态变量里，就使得泛型成为不可能。另一种方法是使用**动态分发**，即用一个 `Box<dyn Trait>` 或 `Arc<dyn Trait>` 来代替实际的类型。然而动态分发也有各种各样的问题，应用范围很有限，下面会详细对比它与我们的设计的区别。

为了解决这个问题，我们参考 crate_interface 的思路，设计了一个全新的 extern-trait。和泛型在编译时确定类型以及动态分发在运行时确定类型不同，我们的设计是在**链接时**确定类型，通过在**函数调用约定**上做了一些巧妙的设计，实现了类型擦除。

这种设计与动态分发的区别在于：（以 `Box<dyn Trait>` 为例）

- `Box` 需要在堆上进行内存分配，`#[extern_trait]` 可以不分配堆内存
- `dyn` 需要虚表，有一定运行时开销；`#[extern_trait]` 的方法调用都是静态的
- `dyn Trait` 要求 `Trait` 是 `dyn-compatible` 的，而 `#[extern_trait]` 限制更少（例如方法可以返回 `Self`）
- `Box<dyn Trait>` 是动态类型，在运行时可能会有多个实现，而 `#[extern_trait]` 约束只能有一个实现

在 小节 4.1 中，我们会展示一个使用 `#[extern_trait]` 的例子。

通过这样的方式，我们可以做到完全零开销地在下层组件中使用上层的未知类型数据。

2.2 scope-local

组件化设计的另一个问题是需要统一组织分布在不同组件中的资源，例如内存地址空间、文件系统信息（包括根目录、当前工作目录、umask 等）、文件描述符表等。如果把他们手动地一个个存储在任务数据块或进程数据块中并手动维护，就丧失了模块化的灵活性，不宜于拓展，并且会让代码变得臃肿。

并且我们面临的另一个特殊的挑战是，在 unikernel 中同样需要使用文件系统等资源，它们不需要把资源与任务关联，只需要单一的全局资源即可。况且 ArceOS 有大量代码是基于 unikernel 设计的，需要尽可能地重用代码逻辑，在设计上进行兼顾。

先前在 ArceOS 中引入了一个 `axns` 模块，把对这些资源的管理抽象为“命名空间”，如在 unikernel 中就只需要一个全局的命名空间，而在宏内核中可以为每个进程分配一个独立的命名空间。不过之前的设计较为简单，也存在内存泄露等问题。

我们继承了 `axns` 的思想，重新设计了 scope-local。

类似标准库中的 `thread_local!`，它也提供了一个 `scope_local!` 宏，并定义了以下几个概念：

- `Item`：资源定义，包含内存布局、初始化函数和回收函数，类型擦除；类似于通过 `thread_local!` 定义的一个全局变量
- `ItemBox`：单个资源的容器，类似于标准库中的 `Box`，通过 `Item` 提供的定义进行内存管理
- `Scope`：作用域，存储一系列资源，是 `ItemBox` 的容器
- `ActiveScope`：当前激活的作用域，初始为自动创建的全局作用域
- `LocalItem<T>`：资源访问代理，保留类型，访问当前激活的作用域的资源；类似于 `thread_local` 返回的 `LocalKey<T>` 类型访问接口
- `ScopeItem<T>`：绑定到某个特定 `Scope` 的资源访问代理

它们的关系如下图所示：

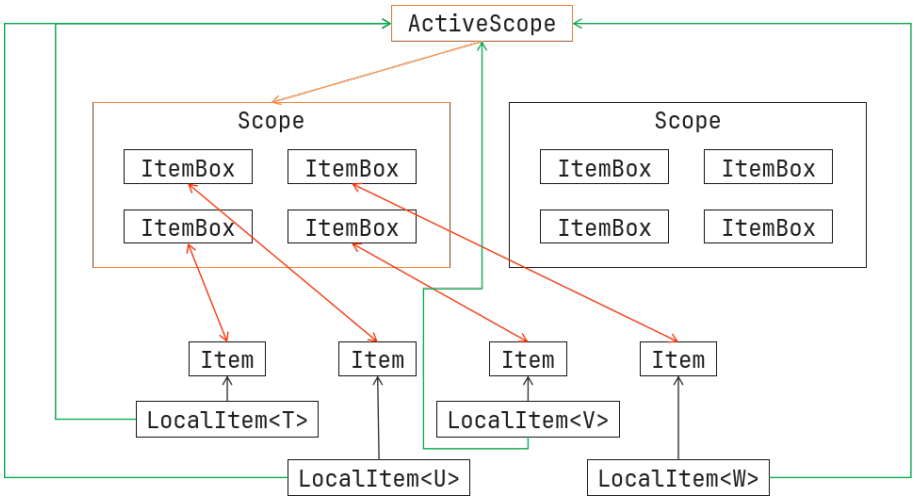

图 2　scope-local

它的使用方式也和 `thread_local!` 类似，例如：

```
/// 定义文件描述符表
scope_local::scope_local! {
    /// The file descriptor table.
    pub static FD_TABLE: Arc<RwLock<SomeType>> = ...;
}

/// 在 clone 系统调用中实现 CLONE_FILES
if flags.contains(CloneFlags::FILES) {
    FD_TABLE.scope_mut(&mut scope).clone_from(&FD_TABLE);
} else {
    FD_TABLE
        .scope_mut(&mut scope)
        .write()
        .clone_from(&FD_TABLE.read());
}
```

第 3 章 内存管理

3.1 内存布局

Starry Mix 使用单一页表架构，内核与用户共享地址空间。为了灵活适配不同的底层硬件架构，我们将外部的配置文件（`toml`）通过 `axconfig` 库集成进操作系统，在其中可以动态配置操作系统的内存布局。

Starry Mix 的内存空间主要由如下部分构成（由低到高）：

- **用户映射空间**：存储用户代码和静态数据
- **用户堆空间**：为用户堆预留的地址范围
- **用户栈空间**：自高向低分配
- **信号跳板地址**：见 小节 6.2.2
- **物理地址范围**：包含内核代码、内核栈与内核堆

虚拟页表的高地址范围被线性映射至物理地址，但只有内核可以访问。这部分的内存布局并不固定（除了内核代码），由内核动态分配。

3.2 内存分配

内存分配的相关逻辑实现在 `axalloc` 模块中，其核心是 `GlobalAllocator` 结构，其定义如下：

```
pub struct GlobalAllocator {
    balloc: SpinNoIrq<DefaultByteAllocator>,
    palloc: SpinNoIrq<BitmapPageAllocator<PAGE_SIZE>>,
    stats: SpinNoIrq<UsageStats>,
}
```

其中，最底层的 `palloc`，基于位图分配器管理未使用的页并将其分配；而 `balloc` 基于 `palloc`，提供了常用的以字节范围为单位的分配接口。其实现根据 `axalloc` 启用的 features 决定，可选的有 Slab 算法、Buddy 算法与 Tlsf 算法。

`stats` 用于记录内存分配的相关统计数据。我们将内存分配分为如下几种类型：

- `RustHeap`：Rust 内核中分配的堆内存
- `UserMem`：用户使用 `mmap` 等接口时动态申请的内存

- `PageCache`：用于页缓存的内存
- `PageTable`：用于存储页表项的内存
- `Dma`：用于 DMA（直接硬件访问）的内存

我们记录这些内存类型对应的总字节数，便于后期我们调试诊断内存泄露问题。

3.3 虚拟页表管理

Starry Mix 中的虚拟页表由 `AddrSpace` 对象管理，其内部封装了 `memory_set::MemorySet<Backend>`。`MemorySet` 是一个抽象的内存范围管理结构，由内部的泛型对象（即我们实现的 `Backend`）来管理具体的内存分配，需要提供 `map`、`unmap`、`protect` 三种接口，分别对应 POSIX 中 `mmap`、`munmap`、`mprotect` 三种系统调用。

`Backend` 取决于该内存范围所采用的内存模式，分为四种：

- `Backend::Linear`：**线性映射**
- `Backend::Cow`：**动态分配映射**，支持**写时复制**
- `Backend::Shared`：**共享内存映射**
- `Backend::File`：基于**文件**的可读写映射

在执行上述系统调用时会调用 `Backend` 上的对应接口。此外，在用户态程序触发缺页异常时，也会根据对应的 `Backend` 决定处理策略。

3.3.1 线性映射

线性映射将虚拟内存范围线性映射到特定的物理内存范围，其实现也相对直接，直接调用由 `page_table_multiarch` 提供的页表接口即可。

内核并不直接面向用户提供线性映射的接口，其只用于内核内部对内存空间的初始化。

3.3.2 动态分配映射

动态分配映射，即在初始 `map` 时并不分配物理页，只在发生缺页异常后才分配对应页并无缝返回到用户态；整个过程对用户是透明的。该映射还提供可选的文件支持，如果映射对应了一个文件，则新创建的页默认会设置为读取的文件内容。

该映射类型采用写时复制策略，即在 `clone` 后子进程直接继承父进程映射的页，只在写入后触发权限异常才重新分配新页。这种设计保证了操作系统的高效性。

该映射类型对应 `mmap` 调用中的 `MAP_PRIVATE` 类型。

3.3.3 共享内存映射

如果 `mmap` 的参数中包含 `MAP_SHARED` 标志位，代表这块内存是共享内存，这意味这不同的进程可以共享同一个内存范围。我们采用的实现策略相对简单：在分配时将对应的物理页全部分配，并将物理页地址存储在 `SharedPages` 中（物理页数组）。此后，如果其他进程想映射相同的共享内存，只需要根据键值拿到对应的 `Arc<SharedPages>` 并映射即可。

3.3.4 基于文件的可读写映射

当 `mmap` 中 `fd > 0` 且类型为 `MAP_SHARED` 时，代表该内存映射对应到特定文件且需要与其他进程共享。为了实现这一行为，我们对照 Linux 实现了页缓存机制。基于文件系统的数据绑定机制（小节 5.1.8），我们可以将页缓存信息绑定到一个目录项上。

```
struct CachedFileShared {
    page_cache: Mutex<LruCache<u32, PageCache>>,
    evict_listeners: Mutex<LinkedList<EvictListenerAdapter>>,
}
```

文件缓存维护两项成员：页缓存列表与逐出监听。页缓存列表使用 LRU 队列进行维护。当页缓存过多，需要逐出旧的页缓存时，直接释放该页往往是不够的，因为该页可能已经被映射到多个任务的地址空间中。如果这些任务继续对该页进行读写，可能造成不可预期的后果。为此，每一个使用该文件缓存的任务都会注册一个逐出监听，当监听当对应页逐出时会对应将指定页从自身的地址空间 unmap，从而确保了文件映射的正确性。

3.4 用户地址访问

原 ArceOS 实现中，由于用户态内核态共享地址空间，用户态指针仅表示为裸指针，存在如下问题：

- 发生缺页异常时，故障处理函数会因为故障来自于内核而主动崩溃（因为在内核发生缺页是致命的，可能已经发生严重错误）

- 无法主动探测地址异常并返回错误（POSIX 规范要求，用户提供的地址非法时应返回 EFAULT）
- 需要频繁使用 unsafe 解引用，不美观
- 对 slice 切片和字符串的处理繁琐

为此我们设计了 starry-vm 组件，把对用户空间的访存抽象化。在这一模块中，我们设计了 `VmPtr` 和 `VmMutPtr` 两个 trait，代表一个指向用户空间的指针，并为 `*const T` / `*mut T` / `NonNull<T>` 等指针类型实现安全的访存方法。

它依赖于 小节 2.1 中设计的 `#[extern_trait]`，上层操作系统通过 `VmIo` trait 定义基本的访存办法：

```
#[extern_trait(VmImpl)]
pub unsafe trait VmIo {
    /// Creates an instance of [`VmIo`].
    /// ...
    fn new() -> Self;
    /// Reads data from the virtual memory ...
    fn read(&mut self, start: usize, buf: &mut [MaybeUninit<u8>]) ->
VmResult;
    /// Writes data to the virtual memory ...
    fn write(&mut self, start: usize, buf: &[u8]) -> VmResult;
}
```

具体到 Starry Mix 中，实现会检查用户对指定的内存区域是否有权限读或写，并将其中还未分配或者 COW 的页进行分配，避免发生 page fault 产生大量性能开销。

然后在 starry-vm 中，提供一系列安全的访存封装：

```
/// 读写 slice
pub fn vm_read_slice<T>(ptr: *const T, buf: &mut [MaybeUninit<T>]) -
> VmResult;
pub fn vm_write_slice<T>(ptr: *mut T, buf: &[T]) -> VmResult;

/// 加载到 Vec
pub unsafe fn vm_load_any<T>(ptr: *const T, len: usize) ->
VmResult<Vec<T>>;
pub fn vm_load<T: AnyBitPattern>(ptr: *const T, len: usize) ->
```

```rust
VmResult<Vec<T>>;

/// 加载 C 风格字符串和数组，如 execve 时的 argv
pub fn vm_load_until_nul<T: Pod>(ptr: *const T) -> VmResult<Vec<T>>;

/// 指针读
pub trait VmPtr: Copy {
  ...
  fn vm_read_uninit(self) -> VmResult<MaybeUninit<Self::Target>>;
  fn vm_read(self) -> VmResult<Self::Target>
    where
        Self::Target: AnyBitPattern;
}

/// 指针写
pub trait VmMutPtr: VmPtr {
    fn vm_write(self, value: Self::Target) -> VmResult;
}
```

通过把用户访存抽象为一个独立组件，我们就可以在其他组件中访问用户内存，如 starry-signal。

第 4 章 进程管理

4.1 拓展任务数据

由于 ArceOS 是为 unikernel 设计的，为了支持宏内核进程，需要在原有的任务上关联宏内核需要的信息和相关的方法。为此我们使用在 小节 2.1 中设计的 `#[extern_trait]`，在 ArceOS 中实现对拓展数据类型无感知的情况下就能够保存、回收和调用拓展方法。

在 ArceOS 的 axtask 模块中定义了一个这样的 trait 来支持拓展数据和接口：

```rust
/// User-defined task extended data.
#[extern_trait::extern_trait(pub TaskExtProxy)]
pub unsafe trait TaskExt {
    fn on_enter(&self) {}
    fn on_leave(&self) {}
}
```

随后在任务数据结构中，只需要使用 `TaskExtProxy` 作为拓展数据类型即可，而无需关心用户具体使用什么类型实现了 `TaskExt`。

4.2 数据结构

Starry Mix 的进程数据结构分为两部分：一部分是与内核实现无关的满足 POSIX 标准的字段和结构，另一部分则与具体实现关联。

4.2.1 `starry-process`

`starry-process` 是我们实现的一个通用的进程管理模块，实现了符合 POSIX 标准的会话、进程组、进程到线程：

```rust
// 会话
struct Session {
    sid: Pid, // 会话ID
    process_groups: ... // 所有进程组
    terminal: ... // 控制终端
}

// 进程组
```

```rust
struct ProcessGroup {
    pgid: Pid, // 进程组ID
    session: Arc<Session>, // 所属会话
    processes: ... // 所有进程
}

// 线程组
struct ThreadGroup {
    threads: ... // 所有线程
    exit_code: i32, // 退出码
    group_exited: bool, // 是否已退出全部线程，对应 exit_group 系统调用
}

// 进程
struct Process {
    pid: Pid, // 进程ID
    is_zombie: AtomicBool, // 是否为僵尸进程
    tg: ... // 线程组
    children: ... // 子进程
    parent: ... // 父进程
    group: ... // 所属进程组
}
```

在功能上，提供了一套原子化的访问和管理接口，主要包括：

```rust
impl Process {
    // 创建会话，对应 setsid 系统调用
    pub fn create_session(self: &Arc<Self>) -> ...;
    // 创建进程组，对应 setpgid 系统调用
    pub fn create_group(self: &Arc<Self>) -> ...;
    // 移动进程到进程组，对应 setpgid 系统调用
    pub fn move_to_group(self: &Arc<Self>, group: &...) -> bool;

    // 退出线程，返回值表示是否为线程组中最后一个线程
    pub fn exit_thread(&self, tid: Pid, exit_code: i32) -> bool;

    // 退出进程，包括标记为僵尸进程、子进程移交收割者等操作，等待父进程 wait
    pub fn exit(self: &Arc<Self>);
    // 释放资源，用于 wait 系统调用
```

```
    pub fn free(&self);
}
```

4.3 自动回收资源

一方面，对于用户程序来说，进程组、会话等功能是可选的，如果不需要管理进程组等资源就可以不用调用相关系统调用，内核会自动负责分配和回收相关资源。而想要尽可能零开销地回收这些“所有进程都退出了”的进程组或者“所有进程组都退出了”的会话，难免要使用引用计数。另一方面，我们在某些功能里需要遍历当前所有进程或者某个进程组中的所有进程，需要维护进程的存活状态，因此也需要对进程进行引用计数。

引用计数在 Rust 语言中有内置的的数据类型支持，包括强引用类型 `Rc` 和 `Arc`，以及对应的弱引用 `Weak`。我们自然想尽可能地利用 Rust 本身的语言特性，而避免编写手动管理的代码。然而如果按照常规的存储树状结构的方法，即在父节点中保存子节点的强引用，最多只能做到自动回收作为叶子节点的进程的程度，仍然需要手动判断进程组是否为空再进行回收。我们创新性地设计了一种从子节点保存父节点强引用的办法，如下图所示：

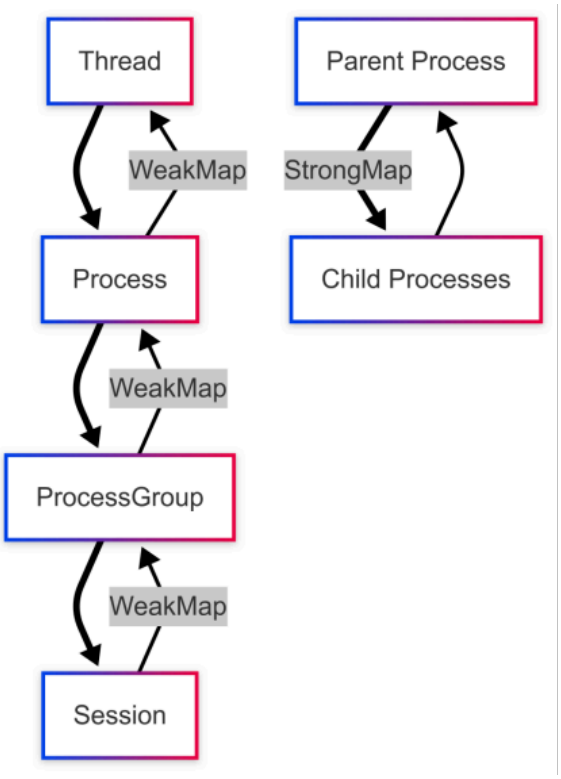

图 3　进程数据结构

4.3.1 `weak-map`

该结构的核心在于我们设计的`weak-map`。这个 crate 提供了一个`WeakMap`类型，接口与标准库`BTreeMap`一致，但是使用弱引用存储值。当弱引用失效时，访问`WeakMap`会“如同”对应条目不存在。为了避免调用时产生大量重复代码，它的全部接口的参数和返回值反而都是强引用。通过这样的设计，它提供高效迭代和访问策略，避免对已失效数据的访问。

这种设计的关键就在于`WeakMap`。我们用一个例子来说明：

```
// 进程表
let mut procs = WeakMap::new();
// 创建一个进程
let proc = Arc::new(Process:new());
// 插入到进程表中
procs.insert(0, &proc); // 直接插入强引用
// 之后从进程表中获取
procs.get(&0); // 返回类型为 Option<Arc<Process>>
// 释放进程，模拟进程退出后父进程通过 wait 回收进程资源
drop(proc);
// 之后再尝试获取这个进程
procs.get(&0); // 返回值为 None
// 统计总进程数
procs.len(); // 返回值为 0
```

而如果朴素地使用`BTreeMap`存储：

```
let mut procs = BTreeMap::new();
let proc = Arc::new(Process:new());
procs.insert(0, Arc::downgrade(&proc)); // 需要手动 downgrade
procs.get(&0); // 返回类型为 Option<&Weak<Process>>，还需要进一步 upgrade
drop(proc);
procs.get(&0); // 返回值仍然为 Some(_)，但是里面实际上是已经过期的弱引用
procs.len(); // 返回值为 1
```

可以看到使用`WeakMap`极大程度简化了处理流程，避免了大量重复代码和由于可能的程序员编码疏忽而产生的错误。

`WeakMap`的内部实现使用一个操作计数器懒惰地清理过期引用，在避免内存泄露的同时，相比积极的清理策略可以产生更小的内存开销并避免破坏所有权。

```rust
// 计数器
#[derive(Default)]
struct OpsCounter(AtomicUsize);
// 累积 1000 次操作后尝试清理
const OPS_THRESHOLD: usize = 1000;

// 一些关键操作（节选）
impl<K, V> WeakMap<K, V> where ... {
    fn cleanup(&mut self) {
        self.ops.reset();
        // 进行清理，只保留还未过期的引用
        self.inner.retain(|_, v| !v.is_expired());
    }
    fn try_bump(&mut self) {
        // 增加引用计数
        self.ops.bump();
        // 如果到达阈值就进行清理
        if self.ops.reach_threshold() {
            self.cleanup();
        }
    }
    // 查询时没有可变借用，只增长计数，不尝试清理
    pub fn get<Q>(&self, key: &Q) -> ... {
        self.ops.bump();
        ...
    }
    // 删除时拿到可变借用，可以尝试清理
    pub fn remove<Q>(&mut self, key: &Q) -> ... {
        self.try_bump();
        ...
    }
}
```

第 5 章 文件系统

原有的 ArceOS 文件系统已经有 `axfs` 作为文件系统的实现，但其中存在几个较大的问题影响到我们操作系统的实现：

- 所有操作基于绝对路径，效率不高的同时语义与 UNIX 不匹配，导致一些系统调用 (openat、fstatat 等）实现繁琐
- 不提供 inode、owner、permission (mode)、soft/hard link 等接口
- 并发访问文件系统存在问题

为此，我们从头实现了新的文件系统模块：`axfs-ng`。类似于原 `axfs-vfs` 与 `axfs`，我们也分为 `axfs-ng-vfs` 和 `axfs-ng` 两个库。`axfs-ng-vfs` 提供文件系统的抽象，和基础的操作接口。`axfs-ng` 则包含了包括路径解析、当前路径（thread-local）以及具体文件系统的实现（如 ext4、vfat 等）。

5.1 结构设计

5.1.1 节点特征 (Node Traits)

文件系统由普通文件、文件夹以及其他特殊文件构成。在 POSIX 的文件系统操作中，有一部分是针对文件的（如 `read`、`write`、`truncate` 等），有一部分是针对文件夹的（如 `read_dir`、`lookup` 等）。常规的做法是将所有操作整合到统一对象上（Inode），但在 Rust 中这样做可能导致不必要的冗余代码。

我们基于 Rust 1.86 版本新稳定的 Trait Upcasting 特性，编写了 `NodeOps`、`FileNodeOps`、`DirNodeOps` 三种 traits，其中后两者均继承自 `NodeOps`。这种设计不仅方便实现（只需实现必须的方法），也有效避免了误用，因为必须要将 `NodeOps` 下转为对应的 `FileNodeOps` 或 `DirNodeOps` 才可使用对应方法，相当于将文件夹 / 文件的检查从运行时通过类型这一工具搬到了编译时。

三种 traits 定义如下：

```rust
/// 通用节点接口
pub trait NodeOps: Send + Sync {
    /// 获取 inode 编号
    fn inode(&self) -> u64;
    /// 获取节点元数据
    fn metadata(&self) -> VfsResult<Metadata>;
```

```rust
    /// 更新节点元数据
    fn update_metadata(&self, update: MetadataUpdate) ->
VfsResult<()>;
    /// 获取节点的文件系统操作接口
    fn filesystem(&self) -> &dyn FilesystemOps;
    /// 获取节点大小
    fn len(&self) -> VfsResult<u64>;
    /// 同步节点到存储介质
    fn sync(&self, data_only: bool) -> VfsResult<()>;
    /// 将节点转换为任意类型
    fn into_any(self: Arc<Self>) -> Arc<dyn core::any::Any + Send +
Sync>;
}

/// 文件节点特化的操作接口
pub trait FileNodeOps: NodeOps {
    /// 带偏移读取
    fn read_at(&self, buf: &mut [u8], offset: u64) ->
VfsResult<usize>;
    /// 带偏移写入
    fn write_at(&self, buf: &[u8], offset: u64) -> VfsResult<usize>;
    /// 追加写入，返回写入的字节数和新的文件大小
    fn append(&self, buf: &[u8]) -> VfsResult<(usize, u64)>;
    /// 设置文件大小
    fn set_len(&self, len: u64) -> VfsResult<()>;
    /// 设置文件软链接目标
    fn set_symlink(&self, target: &str) -> VfsResult<()>;
}

/// 目录节点特化的操作接口
pub trait DirNodeOps<M: RawMutex>: NodeOps {
    /// 读取目录内容，返回读取的条目数
    fn read_dir(&self, offset: u64, sink: &mut dyn DirEntrySink) ->
VfsResult<usize>;
    /// 按名字查找目录条目
    fn lookup(&self, name: &str) -> VfsResult<DirEntry>;
    /// 创建子节点
    fn create(
        &self,
        name: &str,
        node_type: NodeType,
```

```
        permission: NodePermission,
    ) -> VfsResult<DirEntry>;
    /// 链接一个现有节点到目录
    fn link(&self, name: &str, node: &DirEntry) ->
VfsResult<DirEntry>;
    /// 删除目录条目
    fn unlink(&self, name: &str) -> VfsResult<()>;
    /// 重命名
    fn rename(&self, src_name: &str, dst_dir: &DirNode, dst_name:
&str) -> VfsResult<()>;
}
```

5.1.2 节点封装

上面三种 trait 面向文件系统实现者提供了必须实现的函数，但我们实际的文件系统构建中并不会直接使用这三种 trait。我们分别实现了节点对应的封装对象 `FileNode` 和 `DirNode`。这两种节点内部不仅包含对应的动态 trait 对象（`dyn FileNodeOps` 或 `dyn DirNodeOps`），还包含节点必需的额外信息（如 `DirNode` 还包含目录项缓存 dentry）。

虽然有 `FileNode` 与 `DirNode` 的包装，但他们类型并不兼容。`DirEntry` 将他们二者聚合为统一类型，并向外暴露了接口。此外，`DirEntry` 还包含了节点的具体类型（如常规文件或软链接）以及节点的名字和父节点，从而支持了遍历文件树、获取绝对路径等功能。

5.1.3 挂载支持

但在实际的文件系统操作中，由于挂载（mount）功能的存在，单个 `DirEntry` 并不能完全描述一个文件或目录的位置。一个系统中可能同时存在多个文件系统的多个节点，因此我们引入了 `Location` 类型来描述一个节点的完整位置。其定义如下：

```
pub struct Location {
    mountpoint: Arc<Mountpoint>,
    entry: DirEntry,
}
```

在此基础上，`Location` 封装提供了多种操作接口。在大多数情况下，都应该只使用 `Location` 来操作文件。如果例外，可以通过 `Location::entry` 获取

`DirEntry` 来进行更底层的操作。这样做一方面是为了避免内部 `DirEntry` 被不可预料地复制，从而导致内存泄露；另一方面是确保一些底层的一些操作不暴露给外层导致误用（例如 `Location::lookup_no_follow` 和 `DirEntry::lookup` 行为并不一致，前者在后者的基础上还会处理 `.` 和 `..`，同时还会处理目标是挂载点的情况）。

5.1.4 上下文

`FsContext` 是解析路径的上下文，其包含两个字段：根目录与当前目录；二者类型均为 `Location`。在实际的操作系统中，可以通过 `chroot` 来改变根目录、`chdir` 来改变当前目录。

有了上下文后，我们便可以提供路径解析的功能（包括绝对和相对路径）。同时我们也在该类型中继承了类似 `std::fs` 的方法接口，便于实现系统调用时调用。如下所示：

```
let cx: FsContext<RawMutex> = todo!();

let content = "Hello, World!";
cx.write("test.txt", content)?;
assert_eq!(
  cx.read("test.txt")?,
  content.as_bytes(),
);
```

5.1.5 文件对象

文件对象 `File` 在 `Location` 的基础上，包装了文件当前偏移量和文件打开权限，从而提供了 UNIX 风格的文件操作接口，如 `read`、`write`、`seek` 等。在 POSIX 操作系统有 fd (File Descriptor) 的概念。在 axfs-ng 中，一个磁盘文件对应一个 `DirEntry`，一个 `DirEntry` 对应一个 `Location`，一个 `Location` 对应多个 `File`，一个 `File` 对应多个 fds。

5.1.6 打开选项

在打开文件方面，我们设计了与 Rust std 相似的接口。我们定义了打开选项 `OpenOptions` 类型，与 `std::fs::OpenOptions` 基本相同，用于配置打开文件的选项。不同之处在于：

- `OpenOptions` 包含更多底层选项，如配置打开文件时使用的用户身份、权限等
- 由于 `OpenOptions` 也支持打开文件夹，因此 `OpenOptions::open` 返回一个 `OpenResult` 而非 `File`

  这里 `OpenResult` 定义如下：

```
pub enum OpenResult {
    File(File),
    Dir(Location),
}
```

5.1.7 路径操作

路径操作借鉴了 `std::path` 的设计，提供了 `Path` 和 `PathBuf` 两个类型。与 `std::path` 不同的是我们不需要考虑非 UTF-8 以及 Windows 盘符的问题，因此设计上更简单。

实现了一些统一方便的接口，如 `Path::components` 方法来获取路径的各个组件、`Path::join` 方法来拼接路径、`Path::file_name` 来获取路径的文件名等。避免了像原 `axfs` 那样需要在几乎每个文件系统操作中都手动解析路径。

5.1.8 数据绑定

有时，外部的操作系统可能需要在特定的文件上绑定额外的数据，例如命名管道、UDS、文件页缓存等。传统的宏内核不采用模块化设计，需要手动在目录项缓存结构体中添加成员。在 Starry Mix 中，遵循模块化的设计，我们在每个目录项缓存中添加了 `TypeMap` 字段，允许外部使用者在其中存储自定义类型的数据。

`TypeMap` 本质是一个 `TypeId -> Arc<dyn Any>` 的映射，向外部暴露了 `get<T>` 和 `get_or_insert<T>` 等类似传统 map 的接口；方便使用、扩展。具体实现中，为了限制这部分的额外开销，`TypeMap` 没有使用 `BTreeMap` 或是 `HashMap`，而是使用小的 `Vec` 来存储内容。这样不仅节省空间，而且在内容数目不大的情况下省去了额外的开销（如哈希、内存分配），加速了查找。

5.2 实现细节

5.2.1 目录项缓存

类似于 Linux 中的 dentry，我们实现了目录项的缓存机制。其目的一方面在于加快目录项的查找速度，另一方面是确保单个目录项在内存中只存在一份，从而避免多个重复目录项存在情况下可能的并发错误。

其具体实现在 `DirNode` 中。`DirNode` 里包含 `cache: Mutex<BTreeMap<String, DirEntry>>` 成员，用互斥锁保护了一个缓存的目录项表。`DirNode::lookup` 方法会先在缓存中查找目录项，如果未找到再调用 `DirNodeOps::lookup` 方法生成对应的目录项，并将其加入缓存。`DirNode::rename`、`DirNode::unlink` 等方法也会在调用底层 `DirNodeOps` 的基础上清除对应的目录项缓存，保证了缓存的一致性。

但也需要注意的是，一些动态的文件系统（如 `/proc` 对应的 procfs），可能会出现，一个 inode 在没有 unlink 等主动调用的情况下就自动消失，这时候我们的 dentry 是无法自动去捕捉这些信息的，甚至反而会因为 dentry 的存在导致对应的目录项无法正确回收。这里我们为 `DirNodeOps` 增加了额外的方法：`is_cacheable`。当该函数返回 `false` 时，代表该文件夹对应的目录项是动态的、无法被缓存；`DirNode` 在这种情况下便会禁用 dentry 机制。

5.2.2 挂载点

UNIX 系统支持将文件系统挂载到另一文件系统的子目录下，当在访问该子目录时，实际上访问的是挂载的文件系统。我们在 `axfs-ng` 中也实现了类似的功能。具体是定义了 `Mountpoint` 类型包含了挂载点的相关信息（如文件系统实例、根目录等），并在 `DirEntry` 中记录了 `mountpoint: Mutex<Option<Arc<Mountpoint>>>` 字段来记录该 `DirEntry` 上是否挂载了文件系统。

在 `Location::lookup_no_follow` 中，在 `DirEntry::lookup` 的基础上，还会额外检查查找结果的目录项是否是挂载点。如果是，则会返回该挂载点的根目录作为查找结果。我们还正确处理了同一目录被挂载多次的情况，实现的部分均符合 UNIX 规范。

5.2.3 vfat 支持（fat16、fat32）

与 `axfs` 一样，我们基于 `rust-fatfs` 实现了对 vfat 的支持。此外，由于原 `rust-fatfs` 缺少对应实现，我们需要手动添加如获取文件更改时间、获取文件夹相关元数据之类的接口。

由于 `rust-fatfs` 的设计与我们类似，将文件和文件夹区分开来，因此对应实现也是相当直接的。

`rust-fatfs` 的接口只考虑了单线程访问，例如 `fatfs::Dir` 包含了对 `fatfs::FileSystem` 的引用，因此较难扩展到多线程。为了确保并发安全性，我们在全局的 `fatfs::FileSystem` 上添加了互斥锁，并定义了 `FsRef<T>` 类型用于包装 `fatfs::Dir` 和 `fatfs::File`，要求外部提供 `fatfs::FileSystem` 的引用来获取 `T` 的引用，从而确保了多线程访问的安全性。

5.2.4 Ext4 支持

我们基于 `lwext4` 实现对 Ext4 的支持。但其原有的 Rust 封装，也是 `axfs` 使用的 `lwext4_rust` 继承了 `axfs` 的设计，将文件系统操作与路径解析混合，因此完全重写了 `lwext4_rust`，提供了基于 inode 的文件系统操作接口。

此外，由于原 `lwext4` 部分操作强依赖于全局变量（如 `read`、`write` 等），无法同时创建多个实例，因此也需要重写这些接口。这部分完全在 Rust 中完成，避免了改变 `lwext4` 暴露的接口。

5.3 特殊文件系统

在内核中，除了 Ext4、vFAT 这类完整的文件系统，还有一些为内核功能服务的特殊文件系统。基于 Unix 万物皆文件的理念，这类文件系统为应用程序提供了便捷的接口，也是 Unix-Like 操作系统中不可或缺的部分。

5.3.1 tmpfs

tmpfs 一般挂载在 `/tmp`，是存在于系统内存中的文件系统。Starry Mix 中实现了该文件系统，命名为 `MemoryFs`。`MemoryFs` 内部维护了一个 inode 的 arena allocator，文件系统节点根据 inode 编号访问对应的资源，并使用 `InodeRef` 类型维护这些引用。

我们使用 Rust 的 RAII 模式，将 inode 引用的生命周期绑定在 `InodeRef` 类型上。但 `InodeRef` 被 drop 时，其对应的 inode 的引用数也对应减一。这样不仅简化了代码的编写，也利于我们诊断维护对 inode 的引用泄露问题。

对文件夹节点，我们使用 `BTreeMap` 来维护目录项信息；对文件节点，我们定义了结构体 `FileContent`。`FileContent` 支持两种模式：

- 密集模式 `Dense`：使用 `Vec<u8>` 存储文件内容
- 稀疏模式 `Sparse`：使用 `BTreeMap<u64, Page>` 存储文件的非空块

这种设计对应了 POSIX 中对文件空洞的需求。在文件读写没有造成太大空洞时，文件会一直使用密集模式；但如果用户通过 `fallocate` 或 `seek` + `write` 形成较大的空洞，则会自动转换为 `Sparse` 模式。这样既兼顾了内存，也不会造成严重的性能退化。

5.3.2 procfs

procfs 挂载在 `/proc`，提供了任务相关的信息。实现时，我们覆写了文件系统的 `read_dir` 与 `lookup` 方法，使其能将文件系统操作动态映射到当前系统的所有任务上。具体实现中，我们避免了在对应的文件系统节点实现中保存对 `Thread` 或 `Process` 的强引用，因为这可能会导致内存泄露的问题。

此外需要注意的是，为 `/proc` 启用目录项缓存会存在问题，需要将其禁止，详见 小节 5.2.1。

5.3.3 devfs

devfs 挂载在 `/dev`，提供了可供用户访问的设备。为了便于实现简单的设备，我们抽象了新的 trait `DeviceOps`，其定义如下：

```
pub trait DeviceOps: Send + Sync {
    /// 在指定位置读取内容
    fn read_at(&self, buf: &mut [u8], offset: u64) ->
VfsResult<usize>;
    /// 在指定位置写入内容
    fn write_at(&self, buf: &[u8], offset: u64) -> VfsResult<usize>;
    /// 转换为任意类型
    fn as_any(&self) -> &dyn Any;
}
```

在外层我们使用 `Device` 对象包装 `DeviceOps`，在 `DeviceOps` 的基础上还提供了设备号、设备类型的信息。之后实现里直接将 `Device` 对象提供给上层文件系统即可。

5.4 使用举例

下面给出使用 `axfs-ng` 的一些代码样例：

```rust
/// 创建文件系统
let disk = RamDisk::from(&std::fs::read("resources/ext4.img")?);
let fs = fs::ext4::Ext4Filesystem::<RawMutex>::new(disk)?;

/// 创建挂载点对象（根目录也算挂载点）
let mount = Mountpoint::new_root(&fs);

/// 根据 mountpoint 创建文件系统上下文
let cx: FsContext<RawMutex> = FsContext::new(mount.root_location());

/// 列举根目录内容
for entry in cx.read_dir("/")? {
    let entry = entry?;
    println!("- {:?} ({:?})", entry.name, entry.node_type);
}

/// 打开文件
let mut file = File::create(&cx, "test.txt")?;
file.write_all(b"Hello, world!")?;
drop(file);

/// 创建文件夹
let mode = NodePermission::from_bits(0o766).unwrap();
cx.create_dir("temp", mode)?;

/// 挂载子文件系统
let disk = RamDisk::from(&std::fs::read("resources/fat16.img")?);
let sub_fs = fs::fat::FatFilesystem::<RawMutex>::new(disk);
cx.resolve("temp")?.mount(&sub_fs);
```

第 6 章 信号系统

信号作为 POSIX 中经典的机制，是一种基础的跨进程通信方式，用于通知进程某种事件的发生。Starry Mix 的信号核心逻辑实现在独立的库 `starry-signal` 中，可以被其他操作系统复用；Starry Mix 自身基于 `starry-signal` 实现了与 Linux 兼容的信号相关系统调用。

Starry Mix 同时支持标准信号（又称 POSIX reliable signals）和实时信号（POSIX real-time signals）。

6.1 信号队列

每个线程和进程都维护了信号队列 `PendingSignals`，其定义如下：

```
pub struct PendingSignals {
    pub set: SignalSet,
    info_std: [Option<SignalInfo>; 32],
    info_rt: [VecDeque<SignalInfo>; 33],
}
```

其中 `set` 以位图形式存储了所有待处理的信号（至多 64 种），而 `info_std` 与 `info_rt` 分别存储了标准信号和实时信号的附属信号信息。标准信号和实时信号的区别在于，当单个标准信号被递送多次而进程没有处理时，会发生信号丢失；而实时信号会按先入先出的顺序存储所有信号递送信息。

使用 `kill`、`rt_sigqueueinfo` 等系统调用向进程发送信号时，对应的信号信息会被加入线程或进程的信号队列。由于信号的处理是异步的，这些信号并不会立即打断正在执行的线程，而是在之后的某个时间被线程检查并处理。

6.2 信号处理

在 POSIX 系统中，信号有多种处理方式。具体而言，用户可以通过 `sigaction` 系统调用设置信号的处理方式，其中参数 `struct sigaction` 结构体中的 `sa_handler` 项定义了对信号的处理方式，包括如下三种：

- `SIG_DFL`：使用 POSIX 中定义的对信号的默认处理方式
- `SIG_IGN`：忽略该信号，不进行任何处理
- 函数地址：自定义信号处理函数

系统对信号默认的处理方式包括终止进程、生成 core dump、忽略信号等。

6.2.1 信号栈

处理用户自定义信号处理函数时，内核需要为其分配专门的信号栈，以避免其和被打断前的栈空间冲突。栈的相关配置可以通过 `sigaltstack` 系统调用进行设置。具体实现中，该栈不仅会包含必要的信号相关信息，还会包含被打断前的栈帧信息，以便在信号处理完成后能够正确恢复到被打断的状态。

在 `starry-signal` 中，栈的相关设置包含在 `SignalStack` 结构体中，其定义如下：

```rust
#[repr(C)]
#[derive(Clone)]
pub struct SignalStack {
    /// 栈顶指针
    pub sp: usize,
    /// 相关 flags
    pub flags: u32,
    /// 栈的大小
    pub size: usize,
}
```

根据 `SignalStack`， `starry-signal` 可以完成在用户空间分配信号栈的工作，避免了上层操作系统重复实现这一逻辑。

6.2.2 跳板函数

在执行用户自定义的处理函数时，我们需要实现信号跳板（signal trampoline）的机制。具体而言，处理用户自定义信号处理函数时，内核需要临时跳转到用户提供的函数地址，这里跳转后的返回地址是由操作系统决定的，而这个返回地址就称为信号跳板。其内容一般是调用 `sigreturn` 系统调用来将控制权移交回内核。但由于信号跳板必须对用户空间可见，这导致我们无法直接在内核程序中编写信号跳板。

Linux 对此的解决方案是使用 vdso 机制（虚拟动态共享对象）。通过在用户空间映射一个共享库，内核可以在信号处理过程中使用这个库提供的函数，从而实现信号跳板的功能。我们的解决方案与此相似，在内核代码中编写了信号跳板并为将其嵌入到空白页中，在创建用户任务时将其映射到用户空间上的固定地址，这样用户空间的代码就可以直接调用这个信号跳板。

6.3 操作系统解耦

我们将关于信号的大部分逻辑实现在了独立的库 `starry-signal` 中，包括信号结构定义、信号队列、信号处理、信号栈构建等。为了可以在不同的操作系统中复用这些逻辑，`starry-signal` 并不强依赖于具体的操作系统任务实现，而是提供了 `ThreadSignalManager` 和 `ProcessSignalManager` 两个结构体。操作系统将这两个结构体加入到对应的线程存储块和进程存储块中，并在需要时调用其方法来处理信号。同时为了通用与简洁，`starry-signal` 并不提供具体的系统调用实现，而是提供了基础的信号操作原语（如 `PendingSignals` 结构、`send_signal` 函数等），这些原语可以被操作系统的系统调用实现所调用。

第 7 章 异步设计

在原本的 ArceOS 中，只支持同步任务模型，使用基于有栈协程切换。我们为了保证性能和兼容性，仍然沿用了这一设计，但是在此基础上创新性地设计了支持异步任务与运行异步函数的混合调度模式。

7.1 为什么要引入异步

之所以要加入异步支持，一方面是为了复用 Rust 提供的丰富异步生态，包括 `event-listener` 事件监听、`async-channel` 异步管道等。这些组件可以被用于高效地实现 UDS、匿名管道等组件。另一方面，我们可以基于异步优雅方便地实现信号打断机制。这基于 Rust 中 `Future` 基于轮询的特性，在一次 `poll` 结束后我们可以去主动检查信号状态并做出合理的回应。

7.2 `Future` 执行

在原有同步任务调度的基础上，我们实现了一个轻量级的 `Future` 执行器，通过 `block_on` 和 `block_on_interruptible` 两个函数将异步世界与同步世界连接起来。调用后，他们会将当前一个能唤醒当前任务的回调包装为 `Waker` 传入 `Future::poll` 中，并在返回 `Poll::Pending` 的情况下陷入休眠，直到 `Waker` 被调用使得任务被重新放入调度队列中。这样，当调度器下次选择该任务运行时，它就可以从上次暂停的地方继续执行 Future。

`block_on` 与 `block_on_interruptible` 的区别在于，`block_on_interruptible` 会额外在每次休眠前检查任务当前的信号状态，如果被打断则提前返回。为了防止任务永久陷入睡眠无法唤醒，上层在大多数情况下使用的都是 `block_on_interruptible`；`block_on` 只在少数需要特别处理信号，例如 `sigtimedwait`、`sigsuspend` 等系统调用的实现中使用。

7.3 poll 机制与 I/O 多路复用

UNIX 规范中有 poll 的机制，其核心在于对文件描述符的特定事件（比如可读、可写）进行监听，并在事件发生之前休眠，以实现高效的 I/O 操作。为了在 Rust 中实现此机制，但同时又保证同步接口的兼容性（即不大肆改造代码，全部使用 async），我们设计了包含 `Pollable`、`Poller` 的一套异步核心框架。

7.3.1 `Pollable`

`Pollable` 代表一个可以被 poll 的对象，如文件、管道等。

```rust
pub trait Pollable {
    fn poll(&self) -> IoEvents;
    fn register(&self, context: &mut Context<'_>, events: IoEvents);
}
```

其中 `IoEvents` 类型表示一个事件集合，对应 UNIX 中 `POLLIN`、`POLLOUT` 等标志位。

与 UNIX 中的 poll 接口不同，我们设计的 poll 有两个接口：

- `poll`：处理设备事件，返回当前设备状态；
- `register`：注册事件监听，在下次设备发生指定事件时唤醒 `Context` 对应的 `Waker`

UNIX 中的 poll 本质上也是在做这两件事，将其分离后将有助于代码解耦与灵活复用。实际上，Linux 的新版 poll API 也遵循这样的设计。

7.3.2 `Poller`

`Pollable` 的设计相对底层。在其基础上，我们又引入 `Poller` 类型来简化 `Pollable` 的使用。

```rust
impl<'a, P: Pollable> Poller<'a, P> {
    pub fn new(pollable: &'a P, events: IoEvents) -> Self { .. }

    pub fn non_blocking(self, non_blocking: bool) -> Self { .. }
    pub fn timeout(self, timeout: Option<Duration>) -> Self { .. }

    pub fn poll<T>(self, f: impl FnMut() -> LinuxResult<T>) ->
LinuxResult<T> { .. }
}
```

当系统要执行可能需要等待的操作时（如 `read`、`send` 等），会基于操作的目标对象（通常是文件、Socket 等）以及感兴趣的事件创建新的 `Poller` 对象，并配置响应的参数（是否立即返回、超时），然后调用其 `poll` 方法来获取结果。传递给 `poll` 的函数是一个通常立即返回的函数，其在资源充足时可以返回结果，在资源未就绪时返回 `EAGAIN`。`Poller` 在得到 `EAGAIN` 的结果后会最终调用到 `Pollable::register` 方法，将当前任务休眠，并在下一次被唤醒后重新调用 `f`。

7.3.3 对代码逻辑的优化

基于 `Pollable` 与 `Poller` 这两项有利的工具，我们可以优雅且正确地实现事件驱动的 I/O 操作。例如，下面给出了 eventfd 的部分实现：

```
impl FileLike for EventFd {
    fn read(&self, buf: &mut [u8]) -> axio::Result<usize> {
        // ...
        Poller::new(self, IoEvents::IN)
            .non_blocking(self.nonblocking())
            .poll(|| {
                // ...
                match result {
                    Ok(count) => {
                        // ...
                        self.poll_tx.wake();
                        Ok(data.len())
                    }
                    Err(_) => Err(LinuxError::EAGAIN),
                }
            })
    }
}

impl Pollable for EventFd {
    fn poll(&self) -> IoEvents {
        let mut events = IoEvents::empty();
        // ...
        events.set(IoEvents::IN, count > 0);
        events.set(IoEvents::OUT, u64::MAX - 1 > count);
        events
    }

    fn register(&self, cx: &mut Context<'_>, events: IoEvents) {
        if events.contains(IoEvents::IN) {
            self.poll_rx.register(cx.waker());
        }
        if events.contains(IoEvents::OUT) {
            self.poll_tx.register(cx.waker());
        }
```

```
    }
}
```

在引入 `Pollable` 接口前，`read` 部分需要手动与 `WaitQueue` 交互、手动设计循环逻辑、同时要小心处理进入 `WaitQueue` 前后就绪状态发生变动的问题。使用 `Pollable` 接口重构后，代码逻辑明晰许多，避免了大量的重复代码，同时也为上层兼容 UNIX 的 poll 接口提供了基础。

7.4 `PollSet`

在具体实现中常常会遇到如下场景：多个任务监听同一个事件，在事件触发后这些任务需要被全部唤醒。如果使用 Rust 生态中的 `event-listener` 来实现，需要将等待的过程额外包装为 Future、较为麻烦，同时 `event-listener` 包含公平性、重复唤醒规避等高级特性，会带来额外的开销。因此我们设计了 `PollSet` 结构，内部使用原子性的无锁单向列表存储监听在该事件上的任务列表（以 `Waker` 的形式存储）。在 `PollSet::wake` 被调用或 `PollSet` 被 drop 时，这些 `Waker` 会被一并调用并移除，从而满足了我们的需求。

第 8 章 网络

Starry Mix 的网络子系统基于 `smoltcp` 库构建，实现了兼容 POSIX 的 Socket API。系统采用分层设计，在 IP 层实现了**路由**和**转发**功能，上层支持 **TCP**、**UDP** 以及 **UDS (Unix Domain Socket)**。

8.1 底层网络系统

底层网络系统的核心是 `Router`，它扮演了 IP 层数据包转发的角色。

8.1.1 `Router` 的实现

`Router` 实现 `smoltcp::phy::Device` trait，并将其 `medium` 设置为 `Ip`。这意味着 `Router` 直接处理 IP 数据包，而不是更底层的以太网帧。其内部包含一个 `RouteTable`，用于存储路由规则。

```
pub struct Router {
    rx_buffer: PacketBuffer,
    tx_buffer: PacketBuffer,
    devices: Vec<Box<dyn Device>>,
    pub(crate) table: RouteTable,
}
```

当 `smoltcp` 的网络接口需要发送一个 IP 包时，它会调用 `Router` 的 `transmit` 方法。`Router` 会在 `dispatch` 的时候，根据目标 IP 地址查询 `RouteTable`，找到最匹配的路由规则，然后将数据包交给该规则指定的设备（`Device` trait 的实现者）来发送。

8.1.2 设备抽象（`Device`）

我们将底层设备抽象为 `Device`，其负责发送与接受 IP 数据包。

```
pub trait Device: Send + Sync {
    #[allow(unused)]
    fn name(&self) -> &str;

    fn recv(&mut self, buffer: &mut PacketBuffer<()>, timestamp:
Instant) -> bool;
    fn send(&mut self, next_hop: IpAddress, packet: &[u8],
```

```
timestamp: Instant) -> bool;
}
```

其实现包括 `LoopbackDevice` 与 `EthernetDevice`。

- **本地回环设备**（`LoopbackDevice`）：内部维护一个缓冲区，发出的数据包直接由自身接收。
- **以太网设备**（`EthernetDevice`）：负责将底层的以太网卡封装为 IP 包设备，实现了 ARP 邻居协议。

8.1.3 路由表

`RouteTable` 包含一组 `Rule`，每个 `Rule` 定义了一个 IP 地址范围（CIDR）、一个可选的网关（`via`）、一个目标设备以及源 IP 地址。

```
pub struct Rule {
    pub filter: IpCidr,
    pub via: Option<IpAddress>,
    pub dev: usize,
    pub src: IpAddress,
}
```

在系统初始化时，会为环回设备（`lo`）和每个物理网卡（`eth0`）配置相应的路由规则。之后的数据包会根据默认的路由规则流入不同的设备。

8.2 TCP

TCP Socket 的实现位于 `axnet::tcp::TcpSocket`，它封装了 `smoltcp::socket::tcp::Socket`，并提供了面向流的、可靠的通信接口。

8.2.1 状态管理

`TcpSocket` 内部使用一个 `StateLock` 来管理套接字的状态，确保状态转换的原子性和正确性。`State` 枚举定义了套接字可能处于的几种状态：

```
#[repr(u8)]
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
enum State {
    Idle,       // 初始状态，未绑定或连接
    Busy,       // 临时状态，表示正在进行状态转换
```

```
    Connecting, // 正在尝试建立连接
    Connected,  // 已建立连接
    Listening,  // 正在监听传入连接
    Closed,     // 已关闭
}
```

`StateLock` 使用 `AtomicU8` 来存储当前状态，并通过 `compare_exchange` 实现无锁的状态转换。

8.2.2 监听与接受连接

当一个 TCP Socket 调用 `listen` 时，它会向全局的 `LISTEN_TABLE` 注册自己，并开始监听指定的端口。

`LISTEN_TABLE`（`axnet::listen_table::ListenTable`）负责管理所有处于监听状态的 TCP 套接字。当 `snoop_tcp_packet` 侦测到新的连接请求时，`LISTEN_TABLE` 会将该请求加入到相应监听套接字的内部队列中。

当用户调用 `accept` 时，`LISTEN_TABLE` 会从等待队列中取出一个新的连接，创建一个新的 `TcpSocket` 并返回给用户。

8.3 UDP

UDP Socket 的实现位于 `axnet::udp::UdpSocket`，它封装了 `smoltcp::socket::udp::Socket`，提供无连接的、不可靠的数据报服务。

UDP 的实现相对简单。`bind` 会将 Socket 绑定到一个本地地址，`connect` 会设置一个默认的远程地址。`send` 和 `recv` 方法则负责与 `smoltcp` 的 UDP Socket 交互，完成数据的收发。

8.4 Socket 选项

Socket 选项允许用户配置套接字的行为和属性，是 POSIX Socket API 的重要部分。下面列出了一些常用且在 Starry Mix 中实现的 Socket 选项：

选项层级	选项名称	描述
通用	SO_REUSEADDR	允许重用本地地址
通用	SO_RCVTIMEO	接收超时
通用	SO_SNDTIMEO	发送超时
通用	SO_NONBLOCK	设置/获取非阻塞模式
通用	SO_SNDBUF	设置发送缓冲区大小
通用	SO_RCVBUF	设置接收缓冲区大小
IP	IP_TTL	IP 数据包的生存时间（TTL）
TCP	TCP_NODELAY	禁用 Nagle 算法（TCP）
UDP	SO_BROADCAST	允许发送广播数据报（UDP）

表 2　Socket 选项列表

在 UNIX 规范中，程序通过 `setsockopt` 和 `getsockopt` 系统调用来设置和获取 Socket 选项。不同的选项可能有不同类型的参数，在上述接口中它们统一通过一个 `arg` 字段传递。但如果要在 Rust 中实现同样的接口，则会涉及到 `unsafe` 操作且不够类型安全。为了避免这种情况，Starry Mix 秉承模块化的思想，将具体的访存检验操作移交给上层，底层使用安全的接口。

Starry Mix 定义了两种枚举类型：`GetSocketOption` 和 `SetSocketOption`，分别用于获取和设置 Socket 选项。选项的参数位于成员中。其定义大致如下：

```
pub enum GetSocketOption<'a> {
    ReuseAddress(&'a mut bool),
    // ...
}

pub enum SetSocketOption<'a> {
    ReuseAddress(&'a bool),
    // ...
}
```

实际代码中我们使用宏来简化了定义：

```
define_options! {
    // ---- Socket level options (SO_*) ----
    ReuseAddress(bool),
    Error(i32),
```

```
    DontRoute(bool),
    SendBuffer(usize),
    ReceiveBuffer(usize),
    KeepAlive(bool),
    SendTimeout(Duration),
    ReceiveTimeout(Duration),
    SendBufferForce(usize),
    PassCredentials(bool),
    PeerCredentials(UnixCredentials),

    // --- TCP level options (TCP_*) ----
    NoDelay(bool),
    MaxSegment(usize),
    TcpInfo(()),

    // ---- IP level options (IP_*) ----
    Ttl(u8),

    // ---- Extra options ----
    NonBlocking(bool),
}
```

Socket 选项的设置和获取通过 `Configurable` trait 实现，其定义如下：

```
pub trait Configurable {
    fn get_option_inner(&self, opt: &mut GetSocketOption) ->
LinuxResult<bool>;
    fn set_option_inner(&self, opt: SetSocketOption) ->
LinuxResult<bool>;

    // ...
}
```

由此，底层实现便可以通过枚举匹配安全地实现 API。系统调用层（`api::src/syscall/net/opt.rs`）会将来自用户态的 `setsockopt` 和 `getsockopt` 请求转换为对 `Configurable` trait 的调用。

8.5 Unix Domain Socket

Unix Domain Socket (UDS) 是一种在同一台机器上的进程间进行通信的机制。它使用文件系统路径或抽象命名空间中的地址，而不是网络地址。Starry Mix 内核实现了一套兼容 UDS 的接口，支持流式（`SOCK_STREAM`）和数据报（`SOCK_DGRAM`）两种套接字。

8.5.1 核心抽象

UDS 的实现核心是 `TransportOps` trait。这一抽象将具体的传输类型（字节流或数据包）与上层实现（UDS 系统调用：绑定、连接等）分离开来，实现清晰的代码逻辑。

```
#[async_trait]
pub trait TransportOps: Configurable + Pollable + Send + Sync {
    fn bind(&self, slot: &BindSlot, local_addr: &UnixSocketAddr) ->
LinuxResult<()>;
    fn connect(&self, slot: &BindSlot, local_addr: &UnixSocketAddr)
-> LinuxResult<()>;

    async fn accept(&self) -> LinuxResult<(Transport,
UnixSocketAddr)>;

    fn send(&self, src: &mut impl Buf, options: SendOptions) ->
LinuxResult<usize>;
    fn recv(&self, dst: &mut impl BufMut, options: RecvOptions<'_'>)
-> LinuxResult<usize>;

    fn shutdown(&self, _how: Shutdown) -> LinuxResult<()> {
        Ok(())
    }
}
```

这个 trait 被 `StreamTransport` 和 `DgramTransport` 分别实现，并被进一步封装为 `Transport`，同时使用 `enum_dispatch` 静态分发，减少了动态分发的开销。

```
#[enum_dispatch(Configurable, TransportOps)]
pub enum Transport {
    Stream(StreamTransport),
```

```
    Dgram(DgramTransport),
}
```

8.5.2 地址类型

UDS 使用 `UnixSocketAddr` 枚举来表示地址，它包含三种变体：

- `Path(Arc<str>)`：使用文件系统中的一个路径作为套接字地址。这会在文件系统中创建一个类型为 `Socket` 的 VFS 节点。
- `Abstract(Arc<[u8]>)`：使用一个抽象的、与文件系统无关的名称。这部分实现通过一个全局的 `HashMap` 来维护地址和 `BindSlot` 的映射。
- `Unnamed`：代表一个未命名的套接字，通常在 `socketpair` 或未 `bind` 的套接字中出现。

8.5.3 流式套接字（`SOCK_STREAM`）

流式套接字提供可靠的、双向的字节流通信。

8.5.3.1 实现机制

`StreamTransport` 的核心是一个 `Channel` 结构，它包含两个单向的环形缓冲区（`ringbuf::HeapRb`），每个方向一个，构成了全双工的字节流。

```
struct Channel {
    tx: HeapProd<u8>,
    rx: HeapCons<u8>,
    poll_update: Arc<PollSet>,
    peer_pid: u32,
}
```

当两个套接字连接时，它们会交换对方的 `Channel`，从而建立一个通信会话。

8.5.3.2 连接过程

. `bind()`：服务器端调用 `bind` 时，会创建一个 `Bind` 结构，其中包含一个 `async_channel`，用于接收新的连接请求。 . `listen()`：`listen` 在当前的实现中是一个空操作，因为 `bind` 已经完成了监听的准备。 . `connect()`：客户端调用 `connect` 时，会创建一个新的 `Channel` 对，将其中一个 `Channel`（服务器端）和本地地址信息打包成一个 `ConnRequest`，并通过 `async_channel` 发送给监听

中的服务器。 . `accept()`: 服务器端调用 `accept` 时，会异步地从 `async_channel` 中接收一个 `ConnRequest`，并用其中的 `Channel` 创建一个新的 `StreamTransport`，从而完成连接的建立。

8.5.4 数据报套接字 (`SOCK_DGRAM`)

数据报套接字提供可靠的、无连接的数据报通信。与网络上的 UDP 不同，Starry Mix 中的 Unix 数据报套接字是可靠的。这是因为其实现基于内核内的 `async_channel`，它充当了发送方和接收方之间的缓冲区，确保了消息的有序和不丢失（除非缓冲区溢出）。

8.5.4.1 实现机制

`DgramTransport` 使用 `async_channel` 来实现。每个绑定的数据报套接字都有一个 `Channel`，其中包含一个 `Sender<Packet>`，用于发送数据包。

```
struct Packet {
    data: Vec<u8>,
    cmsg: Vec<CMsgData>,
    sender: UnixSocketAddr,
}

struct Channel {
    data_tx: async_channel::Sender<Packet>,
    poll_update: Arc<PollSet>,
}
```

8.5.4.2 通信方式

- **无连接**: 可以直接调用 `sendto`（通过 `send` 方法的 `options.to` 参数实现）向一个指定的地址发送数据。`send` 会查找到目标地址对应的 `BindSlot`，并将数据包发送到其 `async_channel` 中。
- **面向连接**: 也可以先 `connect` 到一个地址，之后就可以直接使用 `send` 和 `recv`，而无需每次都指定目标地址。

8.5.5 命名与绑定

`BindSlot` 结构是实现地址绑定的关键。

```
#[derive(Default)]
pub struct BindSlot {
```

```
    stream: Mutex<Option<stream::Bind>>,
    dgram: Mutex<Option<dgram::Bind>>,
}
```

它包含了分别用于流式和数据报套接字的 `Bind` 结构。对于 `Path` 类型的地址，`BindSlot` 被存储在 VFS 节点的 `user_data` 中。对于 `Abstract` 类型的地址，则存储在一个全局的 `HashMap` `ABSTRACT_BINDS` 中。这种设计确保了地址的唯一性，并允许多种类型的套接字绑定到同一个地址。

第 9 章 物理设备支持

截止编写时，我们还未完成对 2k1000 开发版的支持，所以这里暂时只有 VisionFive 2 的部分。

9.1 VisionFive 2

VisionFive 2 开发版使用 SD 卡作为存储设备。我们参考 os-module/visionfive2-sd、rust-embedded-community/embedded-sdmmc-rs、https://codeberg.org/weathered-steel/sdmmc-driver 等项目，设计了一个独立的 SD/MMC 驱动组件 simple-sdmmc，可以被其他操作系统复用。

在文件系统上，我们复用了 google/gpt-disk-rs，实现了 **GPT 分区**，从而仅使用 SD 卡上的特定分区作为根文件系统，同时也不影响 U-Boot 正常加载操作系统内核。

对 VisionFive 2 开发版的支持单独放在了 axplat-riscv64-visionfive2 仓库中，可供其他基于 ArceOS 的操作系统复用。

由于 ArceOS 缺少对设备树的支持，只能手动维护外部设备，所以目前我们还没有实现网络等外设，后续我们会尝试为 ArceOS 添加设备树解析的支持以更好地在物理设备上工作。

第 10 章 开源协作

Starry Mix 基于的 ArceOS 和 starry-next(StarryOS) 作为开源项目，离不开开源社区诸多贡献者的参与和贡献。

10.1 代码来源

我们在开发过程初期为上游仓库贡献了许多内容，因此其他同样基于 starry-next 进行开发的同学都或多或少地用了一些我们写的代码。可以通过下面的两个链接查看我们贡献的 Pull Request：

- Issues · oscomp/starry-next
- Issues · oscomp/arceos

Starry Mix 的主要代码仓库位于 Starry-Mix 和 arceos，通过 GitHub Actions 自动进行 vendor 等操作并同步至比赛仓库 Starry Mix / starry-mix · GitLab。

arceos 仓库保持了对上游仓库 oscomp/arceos 的 fork 状态，可以直接在 GitHub 上直观地看到我们新增的提交。

Starry Mix 仓库从 starry-next 的 c5942de 提交处分叉，之后新增的代码基本由我们独立完成，但以下内容参考或直接使用了他人的工作：

- `SockAddr` 封装: oscomp/starry-next #56

10.2 第三方依赖

在开发过程中，我们 fork 并使用了以下开源项目：

lwext4 Ext4 文件系统（C 语言）

lwext4_rust 对 `lwext4` 的 Rust 封装

rust-fatfs vFAT 文件系统

kernel_elf_parser ELF 加载

allocator 内存分配算法

smoltcp TCP/IP 协议栈实现

axerrno 错误类型定义

axsched 调度器实现

axio I/O traits 定义

page_table_multiarch 页表管理

axcpu 架构相关硬件抽象层

axplat_crates 平台相关硬件抽象层

axdriver_crates 设备驱动

以及贡献了以下这些 PR:

- arceos-org/axio #1
- arceos-org/flatten_objects #2
- arceos-org/page_table_multiarch #17
- arceos-org/page_table_multiarch #20
- arceos-org/page_table_multiarch #22
- arceos-org/flatten_objects #3
- arceos-org/page_table_multiarch #21
- arceos-org/axcpu #11
- arceos-org/axcpu #12
- arceos-org/axcpu #14
- arceos-org/axplat_crates #17
- arceos-org/axplat_crates #19
- garro95/priority-queue #73

此外我们也使用了许多 crates.io 上的第三方库，这里不详细列举。