Starry Mix 设计文档

清华大学 Starry Mix 队

郭士尧 王铮

2025 年 6 月

目录

第 1 章 概述

进展汇报幻灯片：https://cloud.tsinghua.edu.cn/f/924d8221719a49618ea0/

比赛演示视频：https://cloud.tsinghua.edu.cn/f/e96b194f650d4101a15b/

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。

我们直接使用了 ArceOS 的如下模块并加以改进：

`axalloc` 处理内核内存分配

`axconfig` 统一提供系统相关配置

`axdriver` 提供多种底层驱动（块设备、网络设备、显示设备等）的对外接口

`axhal` 提供多种底层架构的统一硬件抽象层

`axlog` 负责日志打印

`axmm` 处理内存管理与页表相关

`axnet` 提供基于 smoltcp 的 TCP/IP 协议栈

`axruntime` 提供系统入口与运行时

`axsync` 提供同步原语（Mutex）

`axtask` 负责任务调度与管理

我们重新设计了如下模块：

`axfs-ng` 重构的文件系统模块，支持更全面的文件系统操作

我们新增了以下与 ArceOS 无关的独立模块，可以被其他操作系统重用：

`axprocess` 会话 / 进程组 / 进程 / 线程管理和操作

`axsignal` 系统信号处理相关逻辑

`extern-trait` 类型擦除的可拓展接口

`scope-local` 基于作用域的系统共享资源管理和自动回收

Starry Mix 则作为运行在 ArceOS 上的一个“内核应用”，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 ELF，虚拟文件系统（procfs、devfs 等）

`starry-api` POSIX 相关行为（文件描述符）和系统调用实现

`starry` 入口程序，负责加载初始化进程，处理系统调用和缺页等异常

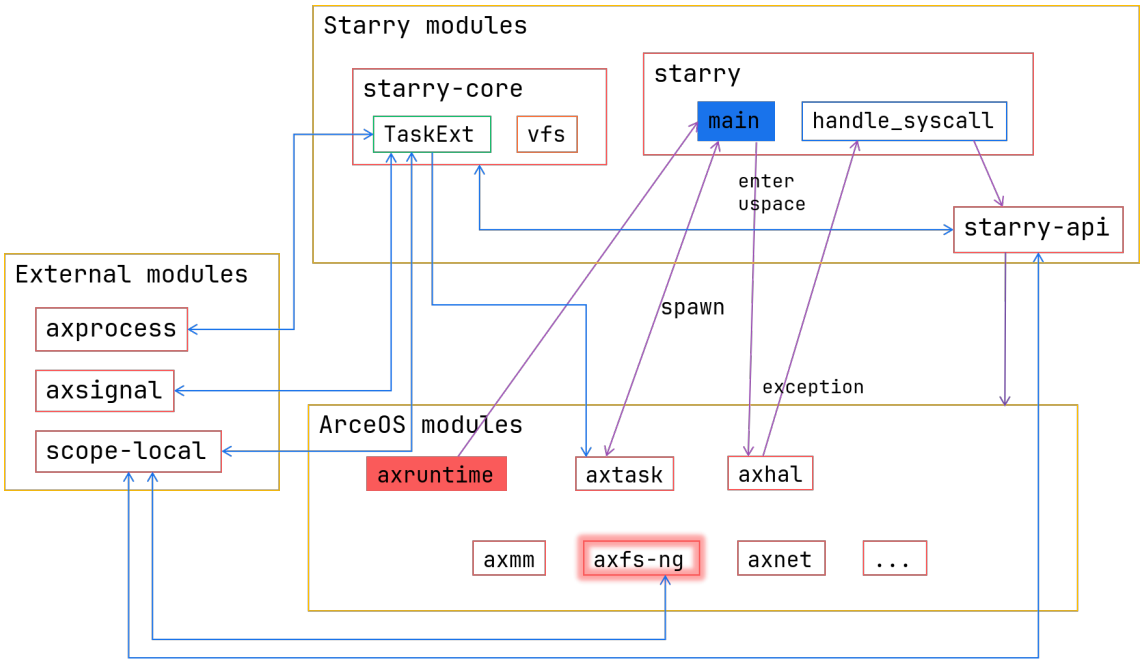

图 1　架构示意图

第 2 章 内存管理

2.1 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 5.2.2
- 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。

2.2 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

- `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。

2.3 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 syscalls。

`Backend` 取决于该内存范围所采用的内存模式，分为三种：

- `Backend::Linear`：线性映射
- `Backend::Alloc`：动态分配映射
- `Backend::Shared`：共享内存映射

在执行上述 syscalls 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。

2.3.1 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。

2.3.2 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。这个过程对用户是透明的。

`mmap` syscall 大部分情况会使用这种内存分配后端。但同时，为了处理 `mmap` 将文件映射到内存的调用类型（`fd > 0`），我们需要额外提供 `populate` 参数。若 `populate` 为 `true`，则该后端在创建时会一次性分配好所有的页。在 `mmap` 中，我们会将文件内容拷贝到已经分配的内存中，供用户使用。

2.3.3 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。

2.4 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 `UserPtr<T>` / `UserConstPtr<T>` 类型。这两种类型是 `*const T` 或 `*mut T` 的封装，提供了更安全的访问方式。提供的接口如下（以 `UserConstPtr<T>` 为例）：

```
impl<T> UserConstPtr<T> {
    /// 获取指针的虚拟地址
    pub fn address(&self) -> VirtAddr;
    /// 类型转换
    pub fn cast<U>(self) -> UserConstPtr<U>;

    /// 将指针转换为引用
    pub fn get_as_ref(self) -> LinuxResult<&'static T>;
    /// 将指针转换为指定长度的切片
    pub fn get_as_slice(self, len: usize) -> LinuxResult<&'static
[T]>;

    /// 将指针转换为以 null 结尾的切片
    pub fn get_as_null_terminated(self) -> LinuxResult<&'static [T]>
    where
```

```
        T: PartialEq + Default;
}
```

此外对 `UserConstPtr<c_char>` 还提供了 `get_as_str` 方法，用于解析用户传递的字符串类型指针。

在将指针转换为引用的过程中，我们做了如下操作：

- 检查地址是否满足对应类型的对齐条件
- 检查对应内存权限是否可供用户读（或写）
- 将范围内未分配的页全部分配

这样就解决了上面提到的几项不足。

需要注意的是 `get_as_null_terminated` 和 `get_as_str` 两种方法，其对应的地址范围大小无法在函数调用时被确定。如果不在访问前将未分配的页分配，则依旧会产生内核缺页的问题。为此，我们添加了全局标志位 `ACCESSING_USER_MEM`，表示当前内核是否正在访问用户内存。如果该标志位为 `true`，则处理缺页时不会拒绝来自内核的异常。这样我们就完成了不定长数据的验证，同时也没有妥协原有的设计（即将一般的内核缺页认为是致命错误）。

第 3 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 syscall (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）。

3.1 结构设计

3.1.1 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<M>: 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<M>;
    /// 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<M>: NodeOps<M> {
    /// 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<M> {
    /// 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<M>>;
    /// 创建子节点
    fn create(
        &self,
        name: &str,
        node_type: NodeType,
```

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

3.1.2 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。

3.1.3 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<M> {
    mountpoint: Arc<Mountpoint<M>>,
    entry: DirEntry<M>,
}
```

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）。

3.1.4 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 syscalls 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(),
);
```

3.1.5 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。

3.1.6 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<M> {
    File(File<M>),
    Dir(Location<M>),
}
```

3.1.7 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。

3.2 实现细节

3.2.1 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。

3.2.2 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。

3.2.3 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。

3.2.4 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。

3.2.5 同步操作解耦

文件系统的实现中不可避免会涉及到同步操作（这里只用到互斥锁 `Mutex`）。为了避免对 `axsync` 的强耦合，我们对关键 trait 均添加了 `M: lock_api::RawMutex` 泛型参数（`<M>`）。`lock_api` 基于 `RawMutex` trait 提供了具体的 `Mutex` 类型，可以根据需要选择不同的锁实现，达成了同步操作的解耦。

3.3 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。

3.3.1 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。

3.3.2 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 3.2.1。

3.3.3 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。

3.4 使用举例

下面给出使用 `axfs-ng` 的一些代码样例：

```
/// 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
```

```rust
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 4 章 进程管理

4.1 任务调度

ArceOS 使用有栈的上下文切换的方式进行任务调度，并支持 FIFO、Round-robin、Completely Fair Scheduler 三种调度算法，其中后两者支持抢占式调度。

4.2 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。我们为此单独设计了一个 `#[extern_trait]` crate，用来在 ArceOS 中实现在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.
/// # Safety
/// See [`extern_trait`].
#[cfg(feature = "task-ext")]
#[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`。

这看起来似乎与 `Box<dyn Trait>` 类似，并且也有类似的接口，但是它们的实现是完全不同的：

- `Box` 需要在堆上进行内存分配，`#[extern_trait]` 可以不分配堆内存
- `dyn` 需要动态分发，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]` 在编译时要求只能有一个实现

通过这样的方式，`#[extern_trait]` 可以做到几乎零开销拓展任务数据。

4.3 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。

4.3.1 `axprocess`

`axprocess` 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：

```
// 会话
struct Session {
    sid: Pid, // 会话ID
    process_groups: ... // 所有进程组
}

// 进程组
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
}

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

```rust
// 线程
struct Thread {
    tid: Pid, // 线程ID
    process: Arc<Process>, // 进程
    data: ... // 拓展数据
}
```

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 Thread {
    // 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(&self, exit_code: i32) -> bool;
}

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);
}
```

4.3.2 自动回收

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：

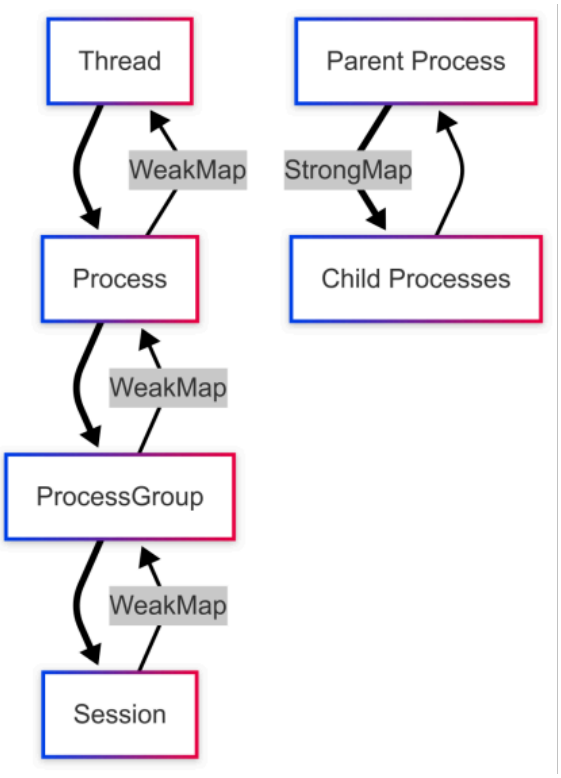

图 2　进程数据结构

4.3.3 `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。

```
// 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();
        ...
    }
}
```

4.4 资源隔离与共享

文件描述符表、文件系统信息（包括根目录、当前工作目录、umask 等）等资源通常在进程间隔离，也可在 `clone` 系统调用时通过设置一些标志进行共享。在通常的类 UNIX 操作系统内核设计中，只需要把这些资源存储在与进程数据中，再使用 `Arc` 和 `Mutex` 等实现一下共享即可。

然而我们面临的一个特殊的挑战是，在 unikernel 程序中同样需要使用文件系统等资源，需要在设计上进行兼顾，况且 ArceOS 有大量代码是基于 unikernel 的单一全局资源模式设计的，需要尽可能地重用代码逻辑。

在 ArceOS 中先前引入了一个 `axns` 模块，把对这些资源的管理抽象为命名空间，在 unikernel 中就只需要维护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命名空间。然而之前的设计较为简单，也存在内存泄露等问题，并且也没有做成独立的内核无关组件。

我们继承了 `axns` 的思想，重新设计了一个名为 `scope-local` 的 crate。

4.4.1 `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：

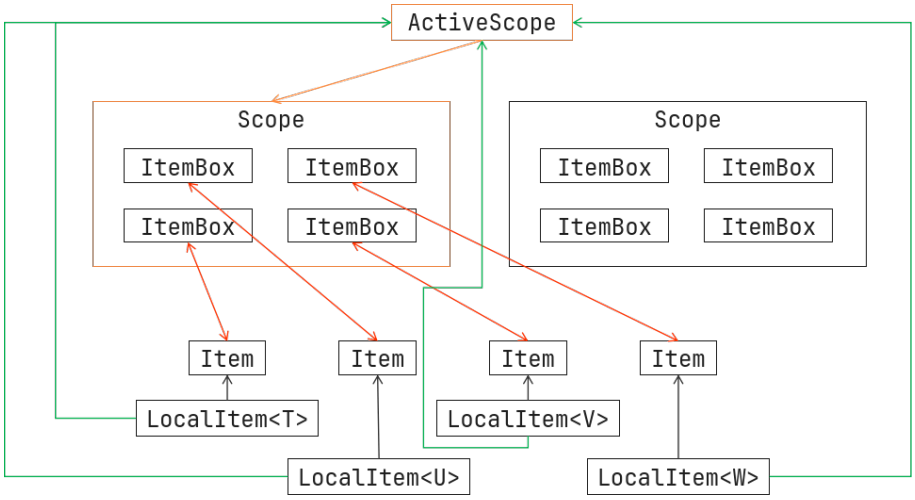

图 3　scope-local

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>> = ...;
}
```

对应 `CLONE_FILES` 的实现：

```rust
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());
}
```

目前这一套系统还只管理上面提到的两种资源，但是这样的动态可拓展设计可以应用到更多资源的管理上，例如地址空间和页表等。

第 5 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 `axsignal` 中，可以被其他操作系统复用；Starry Mix 自身基于 `axsignal` 实现了与 Linux 兼容的信号相关 syscalls。

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

5.1 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 syscall 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。

5.2 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。

5.2.1 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 `axsignal` 中，栈的相关设置包含在 `SignalStack` 结构体中，其定义如下：

```
#[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`， `axsignal` 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。

5.2.2 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。

5.3 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 `axsignal` 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，`axsignal` 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，`axsignal` 并不提供具体的 syscall 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 syscall 实现所调用。

第 6 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。

6.1 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-THU/starry-mix 和 Starry-Mix-THU/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。这其中包括一个来自 linyc1111 的提交 8a20005，用于改进 `axnet` 模块。

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：

- 共享内存实现: AsakuraMizu/starry-next #2
- `SockAddr` 封装: oscomp/starry-next #56

6.2 第三方依赖

在开发过程中，我们 fork 并使用了以下开源项目：

gkostka/lwext4 Ext4 文件系统（C 语言）

elliott10/lwext4_rust 对 `lwext4` 的 Rust 封装

rafalh/rust-fatfs vFAT 文件系统

Azure-stars/kernel_elf_parser ELF 加载

arceos-org/allocator 内存分配算法

smoltcp-rs/smoltcp TCP/IP 协议栈实现

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/flatten_objects #3
- arceos-org/page_table_multiarch #21

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。