NexusOS

一个多核异步、基于框内核架构的 OS

张家文、张昊冉、阎彭旭

2025-07-26

NexusOS

- **基于 Rust 编写**：
  Rust 语言提供内存与并发安全，减少相关 bug 与安全漏洞
- **使用 Rust 异步语法编写异步**：
  异步操作编写更简单、OS 运行更高效
- **框内核架构**：
  最大化 Rust 提供的安全保障，兼具“微内核的安全性”与“单体内核的性能”
- **支持 RISC-V 和 LoongArch 架构**

框内核架构

框内核（Framekernel）是一种将“微内核的安全性”与“单体内核的性能”结合的新型操作系统架构。其核心思想体现在以下几点：

- **单地址空间**：整个操作系统都运行在同一地址空间中，像单体内核一样可以通过普通函数调用和共享内存实现最快速的内部通信。
- **Rust 全栈实现，分区管理安全性**：
  - **OS Framework（操作系统框架）**
    - 体量小，允许使用 **unsafe Rust** 处理极少量底层代码。
    - 其职责是把不安全代码封装为一组 **安全 API**，对内核内存安全负责。
  - **OS Services（操作系统服务）**
    - 体量大（系统调用、文件系统、设备驱动等均在此实现），只能用 **safe Rust** 编写。

框内核架构

- **最小可信基（TCB）**：由于只有 OS Framework 中的代码可使用不安全 Rust，系统整体的内存安全性可以“归约”到这部分极小的 TCB 上，从而显著降低出错面[1]。

- **四大设计要求**：
  1. **Soundness（健全性）**：框架提供的安全 API 结合 Rust 工具链应保证使用者无法触发未定义行为。
  2. **Expressiveness（表达力）**：API 足够丰富，使开发者能在 **safe Rust** 中实现绝大多数 OS 功能，尤其是设备驱动。
  3. **Minimalism（极简性）**：框架体量越小越好；凡是可在外部实现的功能，不放进框架。
  4. **Efficiency（高效性）**：API 应为零开销抽象或仅引入极小开销。

异步与同步的区别

内核异步化的关键点之一是内核控制流的隔离与切换

内核控制流的隔离

同步时，内核控制流是通过多个内核栈以及存储的寄存器值来隔离的。

异步时，内核控制流通过 future 自动生成的状态机来隔离。

内核控制流的切换

同步时，内核控制流通过切换内核栈和寄存器的值(即`context_switch`函数)来切换。

异步时，内核控制流通过切换状态机来切换。或者是通过`future::poll`函数来切换。

异步与同步的区别

对于实现的影响

同步时，

1. 内核任务需要分配独立的内核栈，
2. 并且需要保存和恢复寄存器的值，即需要 task_context 字段来保存和恢复寄存器的值

异步时，

1. 内核任务则不需要独立的内核栈，
2. 无需保存和恢复寄存器，不需要 task_context 字段（对于内核线程与用户线程的切换来说，相应的寄存器数据保存到每个 cpu 都有的内核栈中即可，不需要单独用 task_context 字段来保存）

异步与同步的区别

3. 没有显式的任务切换（即没有显式的 context_switch 函数，而是可能在某一个 await 点切换），且此时一个内核任务对应一个 future 状态机，所以可以把运行的任务存到相应的 future 状态机中（通过将要运行的任务作为 async 函数的参数或 async 闭包的捕获变量，然后把 spawn 此 future）。

进程模型-核心理念

异步与解耦

- **解耦: 可调度实体 (Task) 与 进程语义 (ThreadGroup) 分离。**
- **异步化:**
  - 内核控制流由 `Future` 状态机隔离与切换。
  - 抛弃了传统的独立内核栈与显式上下文切换 (`context_switch`)。

状态隔离

- **信息与状态分离:**
  - **信息 (ThreadSharedInfo)**: 对外共享，可被其他任务观测。
  - **状态 (ThreadState)**: 对内私有，仅任务自身访问。

进程模型-关键数据结构

结构	关键字段	作用
Task	data(跨任务共享只读数据)，local_data(仅当前 Task 可写)，user_space	纯调度实体；不直接持有任何内核资源
CurrentTask	包装 NonNull，并显式 ! Send + !Sync	保证只在当前执行栈使用本地数据，防止逃逸
TaskOptions	builder模式，允许一次性注入 data / local_data / user_space	构建 Task 对象
ThreadSharedInfo	tid、parent/children、lifecycle、cpu_times	进程/线程号及可被其他实体观测的公共状态；跨task共享，可被父子关系、wait4等使用

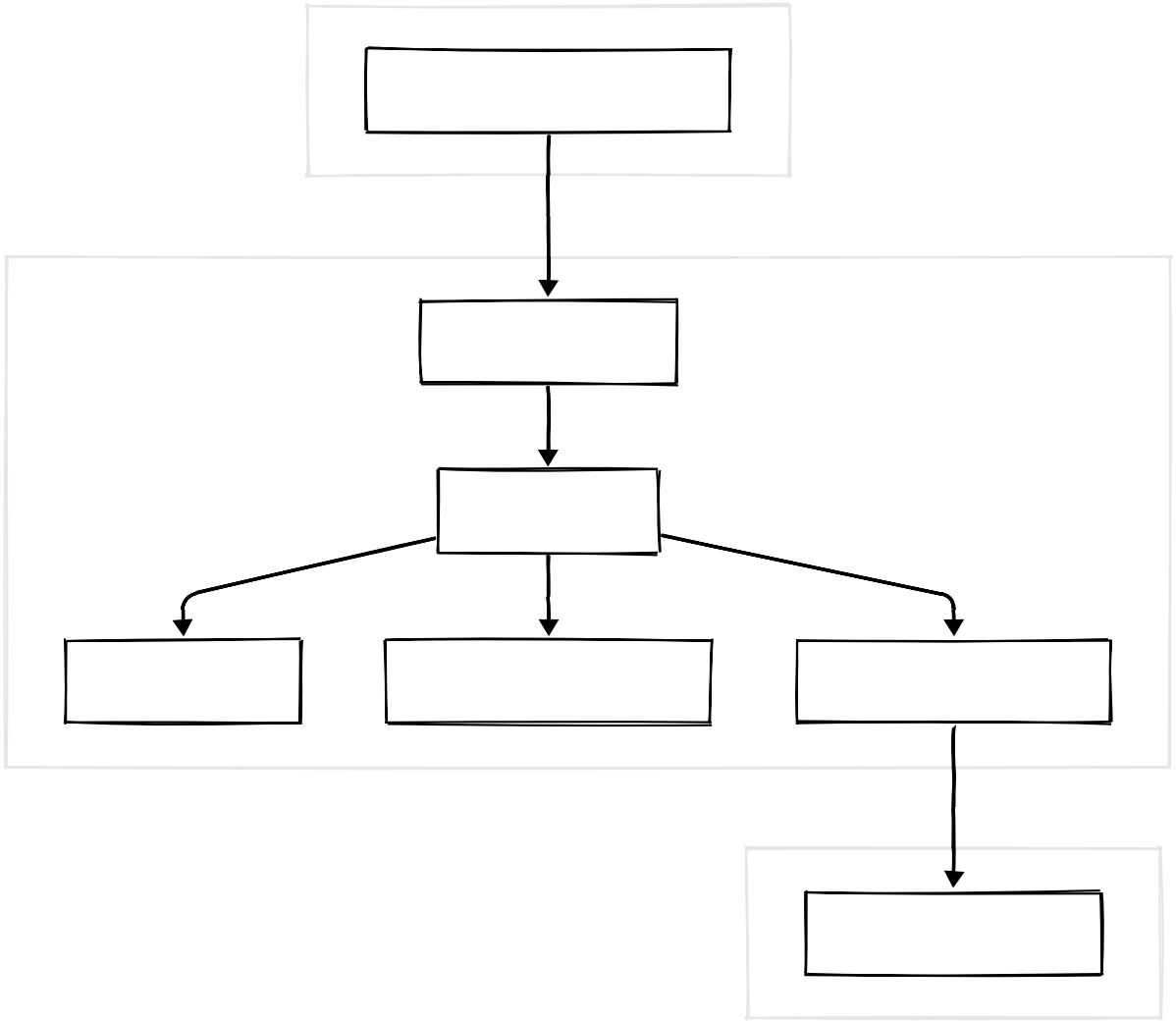
进程模型-关键数据结构

Lifecycle	state(原子枚举) + exit_wait_queue	线程生命周期状态机（Running/Zombie）和等待队列
ThreadGroup	id (=leader tid) + members	线程组容器；实现与 Linux tgid 等价的进程标识
ThreadState	task、thread_group、process_vm、fd_table、cwd、user_brk	系统调用路径的**单一入口参数**，屏蔽细节；sys-call 处理与用户态切换时的工作集

VFS 设计：整体架构

- **目标**: 纯异步、高性能、易扩展的虚拟文件系统。
- **特点**:
  - **全异步接口**: 所有文件操作均返回 `Future`。
  - **静态分发**: 避免 `dyn Trait` 开销，使用 `enum` 在编译期确定调用路径。
  - **分层设计**: 系统调用适配层、VFS 核心层、具体文件系统实现层。

VFS 设计：整体架构



VFS 设计：整体架构

- **Syscall Adapter** 位于 `kernel/src/syscall/fs.rs`，负责把用户态 open/read/write 等调用翻译为对 `SFileHandle` / `SDirHandle` 的异步操作。
- **VfsManager** – “资源目录”+“挂载点”总管

  维护 ProviderRegistry（文件系统类型 → provider 实例）
  维护 **MountRegistry**（挂载路径 → `MountInfo { id, fs }`） **提供** `mount()/unmount()`、`locate_mount()` **等核心服务** 内部使用 ID 池 `IdPool` 发放 `MountId` / `FilesystemId`

- **Static Dispatch** 为了在 `#![no_std]` 环境中避免 traitobject 的动态分发开销，以枚举包裹两种（目前）文件系统实现：`Ext4`、`Dev`。

  `SVnode`、`SFileSystem`、`SProvider` **三个总汇类型** 每个具体能力（文件、目录、符号链接）再细分 `SFile`、`SDir`、`SSymlink` 等枚举

VFS 设计：整体架构

- 缓存

  `VnodeCache`：`<VnodeId, SVnode>`，**容量可配置，当前仅插入不淘汰**

  `DentryCache`：`<(PathBuf, name), SVnode>`，在 PathResolver 中使用

- **路径子系统** 零依赖 POSIX-style 规范化与切片：`PathBuf` / `PathSlice` (path/{buf.rs, slice.rs}) 支持 `.`、`..`、去重 `/` 等逻辑；`PathResolver` 负责跨挂载遍历与符号链接解析。

VFS 设计：整体架构

核心抽象（traits.rs）

Trait	关键方法	备注
AsyncBlockDevice	read_blocks / write_blocks / flush	面向块设备；DevFS 不依赖它
FileSystemProvider	mount()	把块设备+选项→文件系统实例
FileSystem	root_vnode() / statfs() / sync() 等	Arc 参与异步生命周期
Vnode	metadata() / set_metadata() / cap_type()	最小公共能力
FileCap / DirCap …	open() / lookup() / create() / link() …	按需实现的“扩展能力”

VFS 设计：整体架构

Path & Resolver

规范化算法（`path/normalize.rs`）

单遍扫描实现：

1. 拆分 `/`
2. 过滤空段和 `.`
3. 处理 `..`（相对路径下溢时保留 `".."`）
4. 重新拼接

PathResolver

此前用递归实现，但由于递归的异步函数无法确定其 future 的大小，所以返回值需要用 Box 来包裹，这导致性能较差，因此最终采用迭代实现。

VFS 设计：整体架构

外层 `loop`**：处理符号链接重启** 内层 `walk_one_mount()`：

根据最长前缀锁定挂载 → 获取 FS root 逐段遍历目录 命中 DentryCache 后直接返回 碰到符号链接：返回 `Step::Restart(new_abs)` 给外层

最大跟随深度 `max_symlink = 40`。

VFS 设计：整体架构

缓存

```
pub struct VnodeCache {
    cache: RwLock<BTreeMap<VnodeId, SVnode>>,
    capacity: usize,
}

pub struct DentryCache {
    cache: RwLock<BTreeMap<(PathBuf, AllocString), SVnode>>,
    capacity: usize,
}
```

- 读写均为 **async RwLock**；多读单写
- 未实现 LRU，容量超限时直接拒绝插入
- `DentryCache::invalidate_dir()` 提供目录级失效接口

VFS 设计：整体架构

Static Dispatch

- `SVnode` → 按 **类型** 决定分派 (`File / Dir / Symlink`)
- `SFileSystem` / `SProvider` → 按 **文件系统实现** 分派 (`Ext4 / Dev`)
- 每个分派层都把异步方法“手写”封装(预计后面会写宏来避免手动)，每个分派层都实现都会实现其分派类型都有的方法。

具体文件系统实现

DevFS (`impls/dev_fs/`)

- 内存中的 **设备文件系统**，挂载点 `/dev`
- 仅支持 **字符设备文件** 与 **目录**
- `AsyncCharDevice` trait：默认 `read()` / `write()` 返回 `ENODEV`
- 通过 `DevFs::register_char_device(name, dev, perm)` 动态创建 `/dev/<name>`

VFS 设计：整体架构

- 已内置 `StdOutDevice` → 将写入转发到 `ostd::early_print!`

Ext4 (`impls/ext4_fs/`)

- 依赖第三方库 **another_ext4**

Pipe

- `RingPipe` + `PipeReader` / `PipeWriter`
- 在 `SFileHandle` 中实现其静态分发，这是由于 fd 表中存储的是 `SFileHandle`。
- 无文件系统挂载；由 `syscall::do_pipe2` 创建为匿名 `SFileHandle`
- 简易 **阻塞读 + 唤醒** 机制：空读时 `WaitCell::wait()`，写端 `wake()`

SMP 与异步化改造

1. **RISC-V 多核启动**：实现了完整的 RISC-V 多核启动流程。这包括修改启动汇编以区分 BSP (引导处理器) 和 AP (应用处理器)、为每个核心设置独立栈空间、通过 SBI 调用唤醒 AP，并确保每个核心都能正确初始化其本地环境。

2. **异步化内核**：引入了 `maitake` 异步调度器，将内核的任务模型从传统的基于独立内核栈和显式上下文切换的同步模型，转变为基于 `Future` 状态机和调度器轮询的异步模型。 控制流变革：**内核任务的执行逻辑由 `Future` 状态机隐式管理，取代了原有的独立内核栈和 `context_switch` 机制。同步原语调整**：建立了新的同步原语使用策略：对于必须禁用中断/抢占的底层临界区，沿用 `Guard*` 系列锁；对于其他通用场景，则全面采用 `maitake` 提供的异步锁和阻塞锁。

SMP 与异步化改造

3. **核间中断 (IPI)**：为了支持多核间的协作，实现了一套源自 x86 的 IPI 机制。该机制采用 per-CPU 的无锁队列来传递带有类型和参数的命令，实现了高效、类型安全的核间通信。

错误处理设计

受“《Rust 错误处理在 GreptimeDB 的实践》”启发，设计了一套分层错误处理机制，采用 `error-stack` crate 作为基础，并在编写 VFS 首次使用。

最终的效果类似这样：

```
could not parse configuration file
├- at libs/error-stack/src/lib.rs:25:10
├- could not read file "test.txt"
├- 1 additional opaque attachment
|
└─▶ No such file or directory (os error 2)
    ├- at libs/error-stack/src/lib.rs:25:10
    └- backtrace (1)
```

1. **内部错误丰富化**：在 VFS 等核心组件内部，使用 `error-stack` 来构建错误报告。当错误发生时，`report!` 宏会自动捕获源码位置，并通过

错误处理设计

`change_context()` 和 `attach_printable()` 等方法在错误传播链上附加详尽的、结构化的上下文信息。这构成了一个逻辑上的“错误栈”，清晰地展示了错误的根本原因和演变路径。

2. **统一外部接口**：在 VFS 的公共 API 边界，所有内部的 `error-stack::Report` 都会被转换为统一的 `nexus_error::Error` 类型。这个转换过程会首先使用 `tracing` 记录下完整的内部错误报告（包含所有上下文和附件），然后提取出一个最符合 POSIX 语义的 `Errno` 值暴露给系统调用层。

3. **轻量化与高效**：此机制避免了生成高开销的系统级 `Backtrace`，仅依赖 `error-stack` 的源码位置捕获和逻辑上下文链，对正常执行路径的性能影响极小。

遇到的困难

物理地址与虚拟地址的混淆

- **问题**：在尝试启动 AP (应用处理器) 时，最初传递给 SBI `hart_start` 调用的是 `_start` 符号的虚拟地址。然而，此时 AP 的 MMU (内存管理单元) 尚未启用或配置，它需要的是物理地址才能正确定位启动代码。这导致 AP 无法启动，且难以调试。
- **解决方案**：通过分析 QEMU 的启动日志和链接器脚本，确定内核加载的物理基地址。在调用 `hart_start` 时，显式地将启动符号的虚拟地址转换为物理地址。认识到，在与固件和底层硬件交互时，必须严格区分和管理不同阶段的地址空间。

遇到的困难

隐式的初始化顺序依赖

- **问题**：系统在多核启动后发生 Panic，原因是 `timer` 模块在初始化时需要通过 `smp::inter_processor_call` 发送 IPI 以进行 TLB shootdown，但此时 `smp` 模块自身尚未完成 IPI 中断线的分配（`INTER_PROCESSOR_CALL_IRQ`）。这是一个典型的隐式初始化顺序依赖问题。
- **解决方案**：调整了 `ostd/src/lib.rs` 中的模块初始化顺序，确保 `smp::init()` 在所有可能依赖 IPI 的模块（如 `arch::init_on_bsp()`）之前被调用。这要求必须仔细梳理模块间的依赖关系，并使其显式化。

遇到的困难

AP 启动栈与 BSP 栈溢出

- **问题**：在开发初期，单核启动偶尔会发生异常，表现为 `riscv_boot` 函数被两次调用且传入错误参数。经过排查，最终定位到是 BSP (引导处理器) 的启动栈空间不足（仅 64KiB），在解析设备树或进行早期内存分配时发生了栈溢出，破坏了返回地址。同时，AP 也需要自己独立的、安全的栈空间。
- **解决方案**：
  1. 将 BSP 的启动栈空间扩大至 256KiB，解决了栈溢出问题。
  2. 为 AP 设计了独立的启动入口 `_start_ap`。BSP 在唤醒 AP 时，会为其分配独立的栈空间，并将栈顶地址通过 `hart_start` 的 `opaque` 参数（写入 `a1` 寄存器）传递给 AP。AP 从 `_start_ap` 开始执行时，直接从 `a1` 寄存器获取并设置自己的栈指针。

遇到的困难

页表相关

- 启动页表映射冲突：“boot.S”里早期设置的启动页表，和后面代码里操作启动页表的地址搞混了，导致起不来。
- S 模式访问 U 位页表项限制：ostd 创建页表时会默认给加上 User 位（U 位）。但在 RISC-V 的 S 模式下，CPU 不让访问 U 位是 1 的页表项映射的内存，结果内核访问内存时就出错了。
- 非叶子页表项 U 位保留问题：为了解决上面那个问题，给用户空间映射设置了 U 位，但又有新的问题。RISC-V 手册规定，非叶子页表项（就是目录项）的 U、D、A 位是保留的，软件得把它们清零。我在中间层页表设置了 U 位，导致访问异常。

遇到的困难

伪时钟中断风暴

- **问题**: 在某些时候，会进入一种停止运行，只有时钟中断的状态
- **解决方案**:
  1. 排查发现是在只有在进入用户程序时，可能会触发这种状态
  2. 排查发现是增加时钟中断间隔，可以降低进入这个状态的概率，且关闭时钟中断，就不会进入这种状态
  3. 检查发现，相邻时钟中断并没有紧邻，中间有间隔，可以用来运行触发中断周围的代码
  4. 最终发现在切换进入用户程序时，进入中断就可以了。不然，在切换时，如果触发中断，可能会导致种种冲突。

遇到的困难

qemu 版本问题

- 问题: 无法 0x80000000