框内核与框固件

洛佳

2024/11/20

本次分享……

- 框内核（如Asterinas）和OS HALs
  - 框内核设计的极简原则和五条规则；为什么用Rust编写内核?
  - 框内核的架构
  - Asterinas中如何运用框内核?
  - OS HALs：比较ArceOS axhal和ByteOS polyhal；OS HALs是框内核的框吗?
- 作为引导程序固件的“框固件”
  - RISC-V M/S/U模型引导程序固件的解决方案
  - 框固件的架构
  - RustSBI框固件引导程序
- 框固件实现的注意事项
  - 固件组件化的粒度在哪里?
  - 安全固件与安全操作系的分发问题
  - 外设驱动：厂家SoC时钟与IP核设备抽象

框内核与Asterinas项目概述

- 内核 = 框 + 服务
  - 框：安全抽象；服务：设备驱动，操作系统功能
- 调用成本低：框与服务在同一地址空间
- 框和服务如何隔离？使用Rust语言的特性

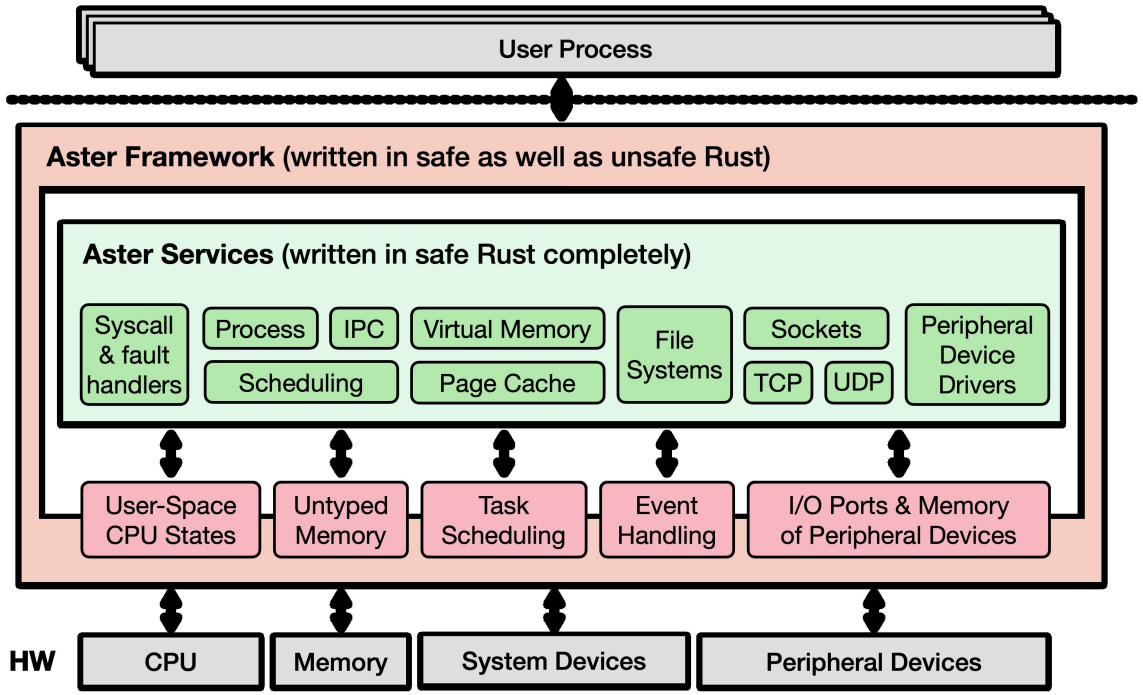
	rCore	RedLeaf	Tock	**Asterinas**
Peripheral drivers	●	◐	◐	○
File systems	●	○	○	○
Network stacks	●	○	○	○
Schedulers	●	●	●	○
IPC and Signals	●	●	●	○
TCB	100%	60.34%	72.68%	20.89%

Table 2: A comparison between the TCB size of different Rust kernels. ●: all crates within the subsystem are included in the TCB; ◐ only some crates are included in the TCB; ○ no crate is included.

- 降低TCB尺寸（TCB即可信计算基，Trusted Computing Base，而不是“线程控制块”Thread Control Block)。框内核中，“框”即可信计算基
  - Asterinas运用了框内核思想，它的TCB（即“框”）占20%。这一数值显著低于其它内核（RedLeaf：60%），且随着设备驱动增加，框的比例会逐渐下降
- 框内核例子的完成度
  - Asterinas具有ext2、TCP/UDP、VirtIO-blk等功能，具有ostd标准库和osdk内核开发工具

框内核原则：极简原则、五条规则

- 极简原则：模块应位于框内，当且仅当模块位于服务时，服务无法仅用安全操作实现所需功能
- 五条规则
  - CPU寄存器：内核寄存器——框，用户寄存器——服务
  - 内存：非类型化内存——框，类型化内存——服务
  - 任务调度：上下文切换——框，调度算法——服务
  - 事件调度：内核态异常——框，用户态异常与外部中断——服务
  - 设备：系统设备（中断控制器等）——框，外围设备（网络、磁盘、图形设备等）——服务
    - 为了避免DMA破坏类型化内存的约束，必须使用IOMMU



为什么用Rust编写内核?

- 高效、内存安全和线程安全
  - （我的理解）高效：编译型语言。内存安全：借用检查、无UB语法。线程安全：Send和Sync traits。
- 所有权模型
  - 我的看法：借用检查永远是存在的，unsafe只是一种着色，允许解引用裸指针等功能。unsafe是人类专家对代码的标注（如Vec::set_len是unsafe fn，但里面没有任何解引用裸指针等参数）。我认为并没有绕过所有权模型

框内核架构

- 威胁模型
  - 框内核认为硬件是可信的，即CPU和系统核心设备是可信的；而外围设备是不可信的
  - 软件可信基（工具链、引导程序等）是可信的；而非特权用户服务、用户程序是不可信的
- 框内核的设计目标：稳健、表达能力、极简、高效
- 类型化内存、非类型化内存
  - 类型化内存：可能危及系统安全的物理内存区域
  - 示例。如MMU建立新映射时，从非类型化内存中分配一块区域；解除映射时，将它恢复为非类型化。
  - 为了确保类型化，框内核具有适当的编译时、运行时保证

Asterinas中如何运用框内核

- 访问非类型化内存：struct Frame（页帧？）
  - 框内核的“框”在运行时分配和释放内存，分配操作将Frame转换为类型化的内存，释放操作将类型化的内存释放回Frame
  - IOMMU只映射非类型化内存（Frame），避免破坏类型化约束
  - MMU只向用户空间分配非类型化内存
  - “框”只允许“服务”将非类型化内存映射到虚拟地址
- 修改用户态CPU寄存器：struct UserMode
  - 内核CPU状态位于类型化内存中
- 任务调度：提供trait以便选择调度算法
- 事件处理：严格区分异常和中断
- I/O操作：系统加载时，从I/O分配器中移除系统端口。MMIO必须排除物理内存（Frame）
- 异步性：使用Send/Sync，UnsafeCell等

框内核思考

- 为什么不能用C语言实现框内核？无法从编程语言约束层面良好隔离框和服务
- 如果为RISC-V架构实现框内核，需要做哪些额外考虑？
  - 必须为外设实现RISC-V IOMMU
  - RISC-V硬件架构可理解为“S态环境”，包括处理器、外设与SBI环境
- “框”和“服务”必须在同一个特权级吗？
  - 是的！否则框和服务间的调用将变得低效
- 用Rust编写的安全固件是否可以拆成框和服务？
  - 可以！拆分使用Rust编写的固件，将能良好划出unsafe和业务部分。安全固件中同样面临用户内存的访问问题（此时“用户”即固件支持的内核），除了内存分配的情况相比内核更少。
- 有没有更好的“Frame”实现？
  - 如果“框”环境指定了全局的页帧分配器组，可从物理页号取得对应分配器，无需在Frame中另行引用

OS HALs：分析ArceOS axhal和ByteOS PolyHAL

- axhal的完成度高
  - 具有task调度上下文
- polyhal具有一些值得参考的地方，如过程宏组成的运行环境
- 值得使用的库：linkin、handler_table
- TLS数据、percpu数据使用了tp和gp寄存器

项目	axhal(arceos-org/arceos)	polyhal(Byte-OS/polyhal)	注
架构支持	aarch64, riscv, x86_64	aarch64, riscv, x86_64, loongarch64	
• 上下文结构	TrapFrame(整数,sepc,sstatus), TaskContext(ra,sp,s0-s11)	TrapFrame(整数,sepc,sstatus,fsx), KernelToken(ra,sp,gp,tp,s0-s11, a0,未使用)	
• TLS定义	读/写tp寄存器	定义tp寄存器存储TLS	
• 杂项抽象	中断（启/停/等待、写入口）、页表（读根/写根/刷新）、CPU编号	中断（启/停）、页表（刷页/刷全部）、CPU编号	
• 宏支持	寄存器读写（仅用于支持内部实现）	寄存器读写、生成宏（arch_entry, arch_interrupt, def_percpu）	
板级平台支持	virtio, raspi, bsta1000b, x86pc	virtio, k210, cv1811h	
• 裸机启动	支持（汇编代码，主从核），初始页表（Sv39线性映射）。调用axtask	支持（汇编代码，主从核），初始页表（Sv39线性映射）	
• 控制台	转发到SBI DBCN/legacy console	转发到SBI DBCN/legacy console	
• 电源操作	关机（转发到SBI SRST）	关机（转发到SBI SRST）	
• 中断与分发	全局处理函数表（HandlerTable，根据平台决定irq上限）,distributed slice，时钟中断	暂无中断分发功能，时钟中断	
• 多核唤醒	使用SBI HSM扩展	使用SBI HSM扩展	
• 时钟	irq时使用SBI TIME，rtc时使用goldfish外设	转发到SBI TIME	
内存布局	内核内存块（含权限管理）、MMIO区域	由fdt决定	
全局数据与TLS	可支持TCB, tdata, tbss	可支持TCB	
• percpu数据（硬件线程级）	gp寄存器，静态决定cpu数量	gp寄存器(cpu数量存疑)	
• TLS数据（任务级）	tp寄存器	暂未使用	
系统调用参数	distributed slice, 静态分发	静态分发	

OS HALs是框内核的“框”吗？

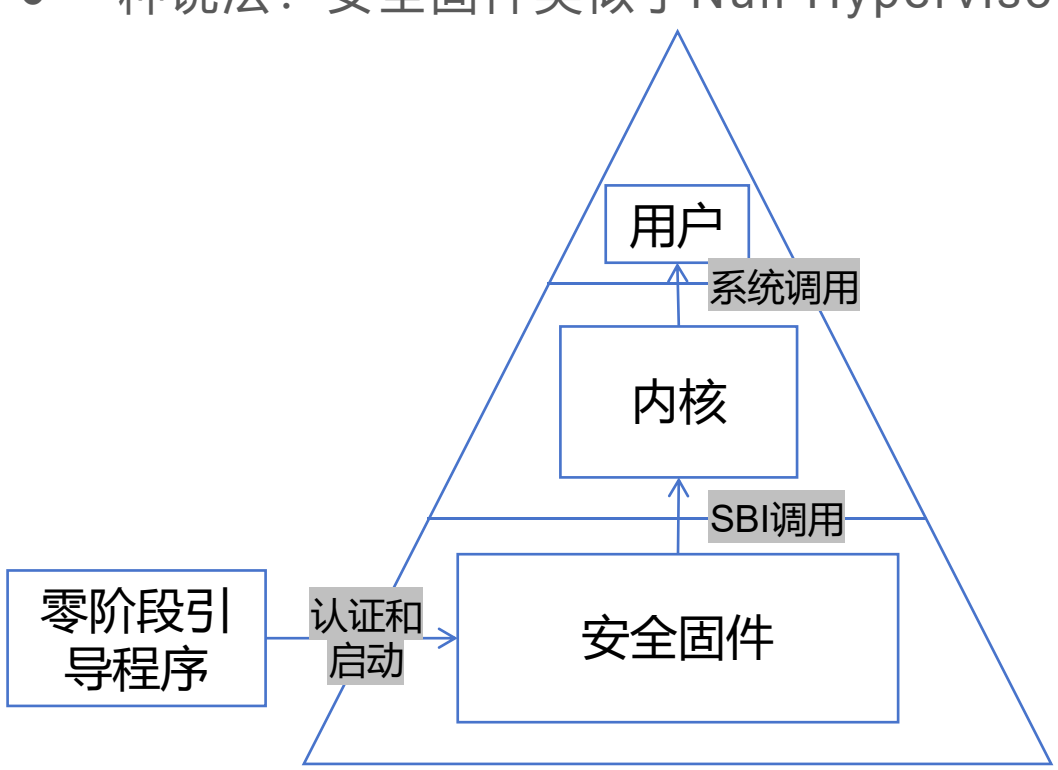
- 非类型化内存目前使用&[u8]、&mut [u8]，而不是Frame等特殊类型
- 用户态寄存器位于TrapFrame中
- 仍需要包含核心系统设备驱动。如何从DTB/ACPI中区分“核心”和“非核心”设备？
- 上下文切换被包含在HAL内，而HAL通常不包含调度算法
- 并未限制MMIO的映射范围
- 思考
  - ostd或axstd是服务吗？两者都包含内核功能、用户态异常等，ostd在设计上包含调度算法，两者都不包含外围设备驱动
  - OS HAL与Embedded-hal：它们都是硬件抽象层，前者抽象内核运行所需的CPU与环境，后者抽象各类嵌入式设备。每个项目对OS HAL的理解各不相同（可能最早源于zCore等项目）

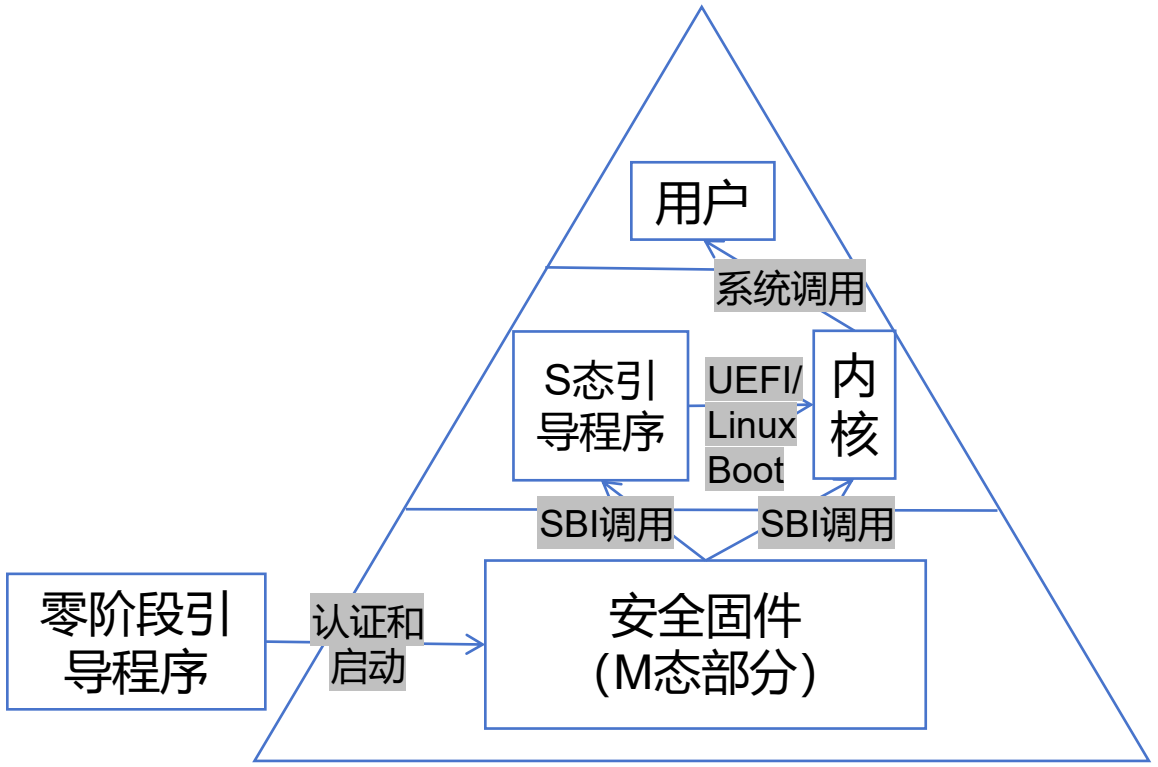
框内核+固件=“框固件”？

- 引导程序固件与框内核面临相似的安全威胁，也就是威胁模型相似
  - 假设：硬件方面CPU和系统核心设备可信，软件方面编译工具和ROM代码可信
  - 为什么我们不提“引导程序可信”？因为固件本身构成引导程序，来证明下一信任环节的可信性
- 固件的设计需求：安全性、性能、可复用性和可靠性
- 即使裸机固件提供的服务较少，引导程序固件要求多功能（TEE等），对内存管理仍有较高要求
  - 如：M态SBI服务的部分模块通过共享内存和S态内核沟通
  - 随着引导程序固件复杂性的增加，它可能成为一个微型内核
- “固件框”+“固件服务”的模型如何分配内存？
  - 将内核、应用都看作非安全的“非类型化内存”，只有安全的“类型化内存”可以被固件使用
- 多阶段引导程序下，每个环节具有各自的“框”和“服务”
  - 其中，S态的“固件框”和“固件服务”架构类似于精简版的“框内核”，精简了U态和系统调用，但增加了引导程序的S/S态接口（如UEFI、LinuxBoot等）

RISC-V M/S/U模型引导程序固件的解决方案

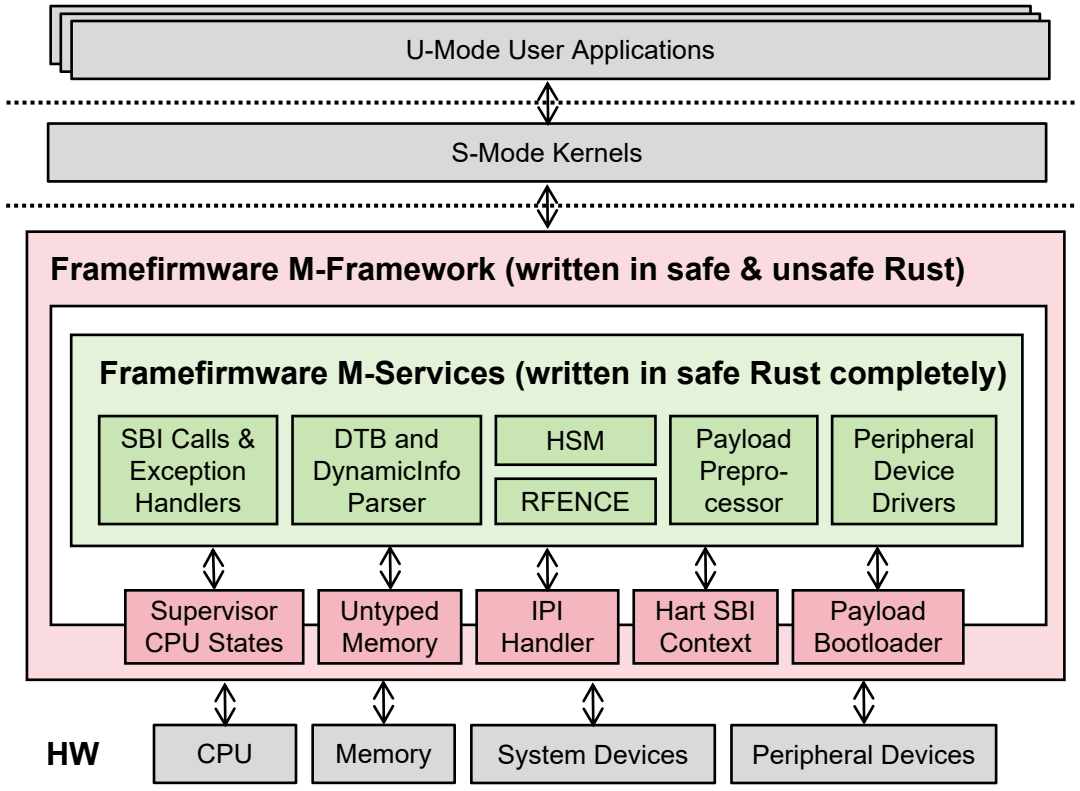
- 首先被零阶段引导程序（ZSBL）启动，然后启动M态的安全固件
  - 从M态启动之后，分为纯M态方案（SBI only）和M/S态混合方案（UEFI，LinuxBoot或U-Boot）
- 一种说法：安全固件类似于Null Hypervisor



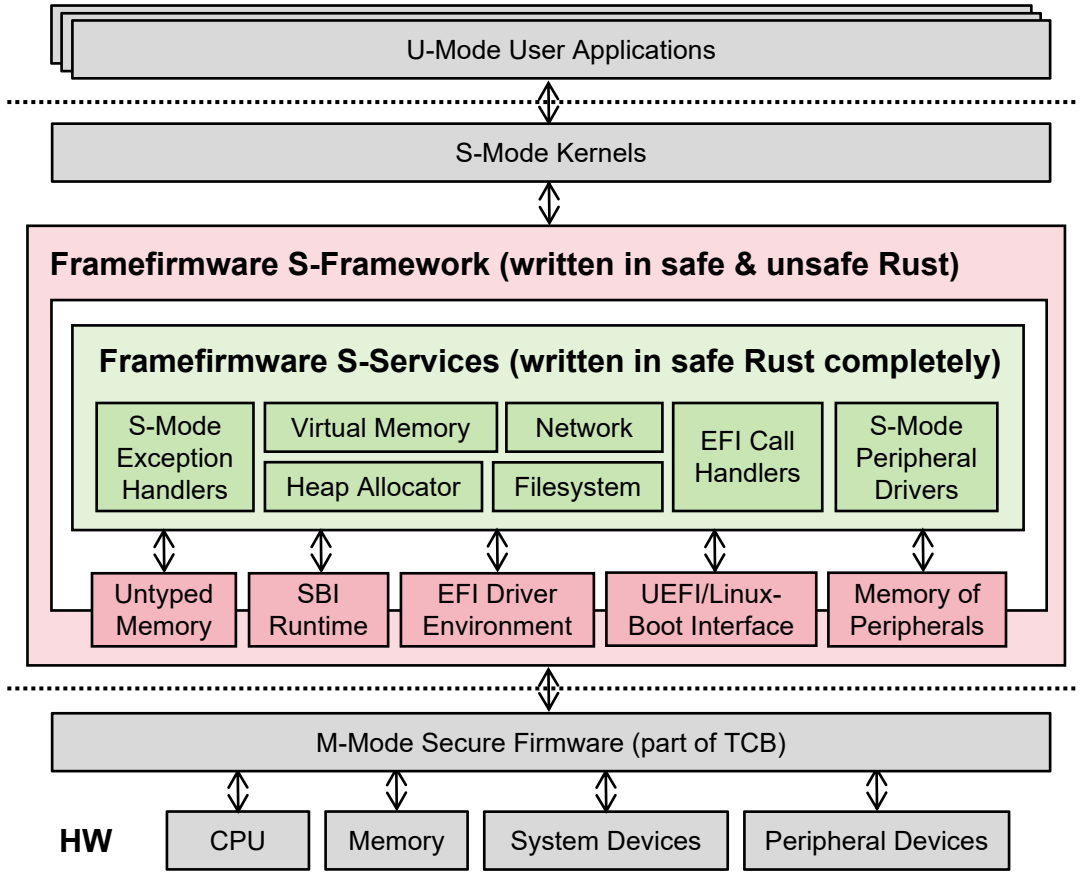


框固件的架构

- 纯M态方案下



- M/S态方案下



RustSBI框固件引导程序

- RustSBI同时适用于纯M态和M/S态混合的引导方案
- 纯M态：RustSBI Prototyper项目（M态模块）支持动态固件和固定载荷两种设计模型
  - “框”包含上下文切换与核的SBI上下文、机器态异常（M态软件中断）和下阶段程序的启动方法等，“服务”包含多核管理、核间中断等SBI服务、M态设备驱动。TCB即“固件框”，M态“固件服务”不属于TCB
- M/S态混合：开发中的RustSBI S态模块
  - 如果S态模块支持UEFI，它将包含网络启动等复杂功能，接近内核的复杂度
  - 和框内核的“框”相似，“固件框”包含系统设备和非类型化内存，“固件服务”包含文件子系统、网络子系统和虚拟内存等基础功能，以及UEFI的EFI调用处理器
  - 出于引导程序固件的功能，“固件框”也包含SBI运行环境、EFI驱动环境和UEFI/LinuxBoot启动方法
  - TCB包含S态的“固件框”和整个M态，而S态的“固件服务”、S态内核不是TCB的一部分
- 即使引导方案同时具有多个阶段，M/S态共同分发仍然是可行的，因为我们认定的TCB占比较小（S态仅包含UEFI固件部分）注：TCB占比是框内核的主要安全性指标

组件化固件的粒度分界在哪里？

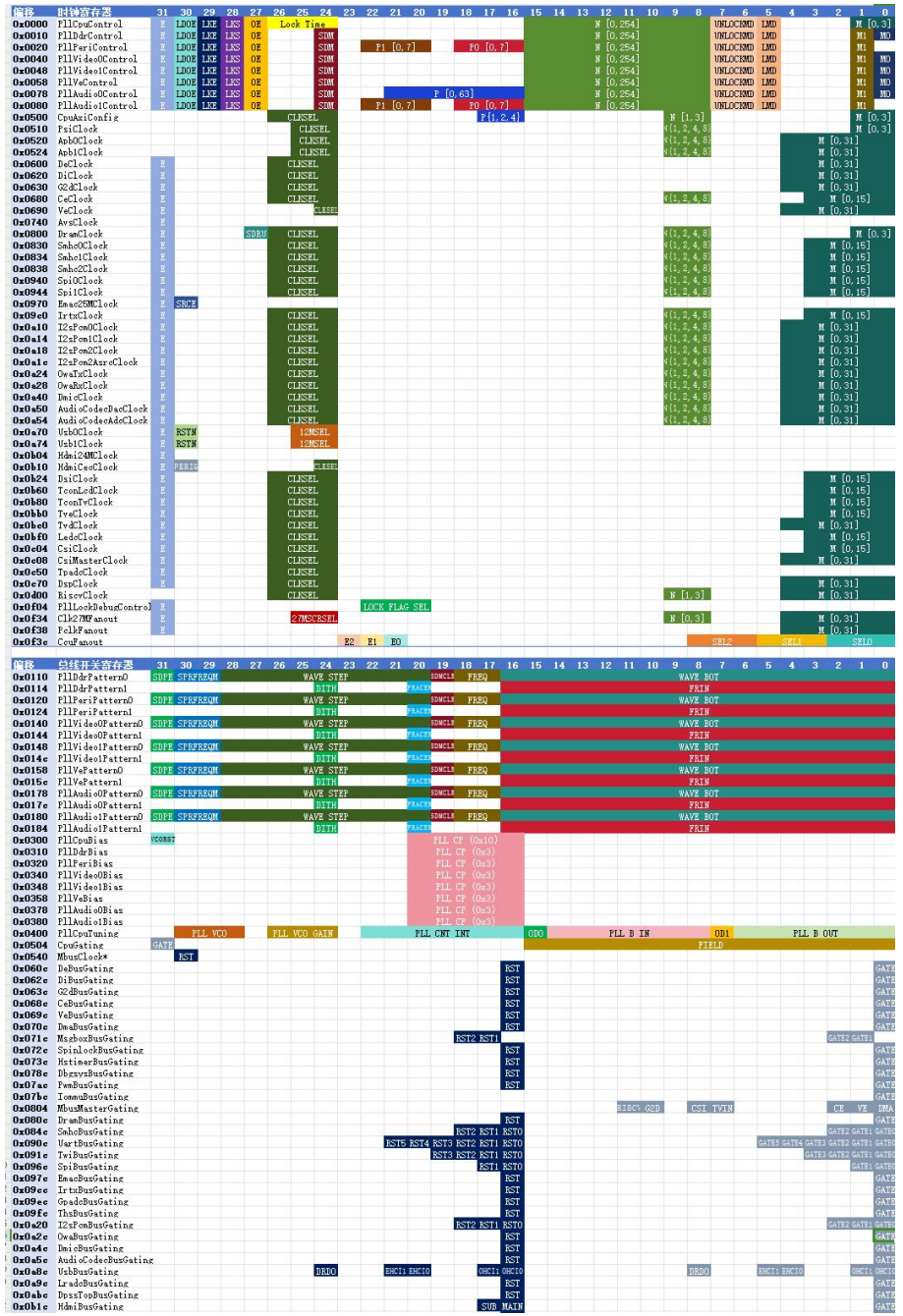
- 为了增强内核的可复用性和内核生态的强健性，构造“组件化内核”（参考ArceOS项目）
- 组件化是否意味着拆得越细越好？并不是
  - 考虑软件工程因素，组件化具有粒度下限：过于彻底的组件化可能导致代码复用性差
  - 早期的RustSBI项目将每个平台的支持拆为一个项目，代码在平台间复制粘贴，违背了可复用性原则
  - 极端化思想：如果每行代码都是一个组件，无法定义组件的“功能”，从而无法测试单个组件，导致生态复用更困难
- 系统软件组件化的粒度分界，在于代码是相对独立的功能集合，易于独立管理和测试
- 组件化固件的粒度有上限，因为对固件的设计目标，安全性和可靠性是重要的
  - 一组经过安全性论证和实际生产环境验证的组件，能降低整个固件生态的开发成本
  - 分立元件的安全并不意味着系统安全，因为此时元件间的组合方式也会影响安全性
  - 因此考虑系统安全因素，组件化固件的粒度应当在降低到可复用性最好的同时，也提高到安全性论证的成本最低（需要验证的组件数最少、组件的组合方式不影响安全性）

安全固件与安全操作系统的分发问题

- 操作系统分发时，应当附带或捆绑固件吗？
  - 不应当附带固件。因为从软件供应链角度，此时固件和操作系统位于同一信任级别，固件的特权边界安全保证失去了保护作用
  - 举例：操作系统镜像作为发行版安装包，给出了哈希值或签名，它以相同的安全算法验证了固件和操作系的未被篡改，然而也意味着固件和操作系统的信任级别是相同的
- 安全固件具有多个引导级别时，应当一同分发吗？
  - 可以共同分发。因为安全固件的多个引导级别体积均较小，合并后TCB仍小于操作系统
  - 对TCB体积有更高要求的用户，可以不使用S态引导程序而直接以安全固件启动，损失可移植性但是提高了安全性
- 虚拟机可以将安全固件以宿主环境的指令集运行吗？
  - 可以，例如：RISC-V SBI也存在于HS和VS态之间，可将RustSBI编译到其它架构，支持RISC-V系统
  - 开发者可使用宿主架构的其它安全机制验证宿主指令集安全固件的安全性，使得虚拟系统的信任链连续

外设驱动：厂家SoC时钟与IP核设备抽象

- SoC时钟的支持方法
  - ROM启动时提供Clocks结构体，表示ROM结束后的时钟配置，节省重配置时间
  - 封装所有的时钟结构，提供手动调节PLL参数的功能
  - 将时钟结构引入到embedded-hal结构化设备的初始化过程中
  - 若硬件没有自动控制时钟依赖的功能，软件应管理依赖的时钟
- IP核设备的支持方法
  - 基于设备类型，选择embedded-hal或embedded-io等库的抽象（USB、SD/MMC和网络有不同的抽象可供选择）
  - 将设备与中断、DMA共同初始化后，提供相同的抽象，使得用户轻松配置更高速度的外设
  - 实现IP核初始化时和时钟结构共同计算波特率等重要频率



结论

- “框内核”是通过编程语言特性将内核划分为“框”和“服务”两部分的隔离机构
- 框内核的思想可以引入固件开发，形成“框固件”
- 以组件化形式设计“框固件”时，组件划分粒度、外设驱动的编写方法需要谨慎考虑

引用文献

- Peng, Yuke, et al. "Framekernel: A Safe and Efficient Kernel Architecture via Rust-based Intra-kernel Privilege Separation." Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems. 2024.
- ArceOS Project. https://github.com/arceos-org/arceos
- RustSBI Prototyper Project. https://github.com/rustsbi/prototyper
- riscv.org. (2023) Risc-v supervisor binary interface specification. RISC-V Platform Specification Task Group. [Online]. Available: https://github.com/riscv-non-isa/riscv-sbi-doc/releases/download/v1.0.0/riscv-sbi.pdf
- Embedded-HAL Project. https://github.com/rust-embedded/embedded-hal

感谢各位

框内核与框固件