

多构型组件化内核的研究与实现

队员：郑友捷、朱懿

学校：清华大学

指导老师：杨金博

选择赛题：灵活组合的操作系统模块和框架 ArceOS

项目导师：陈渝

多构型组件化内核的研究与实现

概述

研究背景和意义

工作内容说明

本文组织架构

组件化内核的概念

组件的概念

组件化应用在内核

探索组件化开发内核的必要性

组件化内核的设计和开发原则

组件化内核的设计原则

设计目标

设计原则

组件化内核的开发模式

简单的 git 及其协作

repo 工具下的协作

git-subrepo 工具下大仓库和多个小仓库的协作

Issue 项目管理

总结

组件化内核的探索成果

实践内核介绍

Starry 介绍

reL4 介绍

实践内核突出点介绍

创新性：内核多构型支持

泛用性：多指令架构支持

实用性：复杂 APP 支持

组件化内核优点介绍

组件便于复用

没有明显的额外性能开销

分层解耦与可替换性

效果展示

Starry

reL4

总结

鸣谢

概述

研究背景和意义

内核开发一直都是重要且有意义的工作。但是由于内核本身的大量代码及不同功能模块之间错综复杂的依赖关系，导致**开发、维护一个成熟的内核非常的困难**。根据

[The Linux Foundation 的报告](#) 以及一些开源社区的统计数据，截至 2023 年初，Linux 内核的代码量大约在 2800 万行左右。具体的代码量可能会因为版本的更新和新功能的添加而有所增加。如此庞大的代码对维护人提出了非常苛刻的要求。为了解决这个问题，许多专家提出了模块化内核的思想：**开发人员维护各自负责的内核模块。在一个统合框架的基础上，通过配置选项选择不同的模块进行组合，从而形成一个可运行的整体内核**。通过将每个内核模块打磨地足够精细可靠，期望让模块组合时出现的 bug 和问题尽可能地少，从而降低内核开发的难度和复杂性。

传统的内核如 Linux 多是使用 C 语言编写，但是 C 语言对模块的管理比较薄弱，允许通过符号链接等形式随意跨越模块来进行函数的调用组合。在缺乏限制的情况下，不同模块之间的依赖关系并不是一个 DAG（单向无环图），而会是复杂的环状、网状关系。为了保证模块化的开发，对编程语言的限制也是一个需要考虑的因素。而 Rust 语言在包、模块上的管理便做得更加成熟，以 Cargo 项目的形式来保证依赖关系的成立。当出现循环依赖时便无法通过编译。因此使用 Rust 开发模块化的操作系统相比于 C 语言会更加具有可行性。

随着近年来使用 Rust 编写新内核或者重写旧内核的实例越来越多，Rust 语言的特性对操作系统编写者的影响也越来越显著。Rust 相比于 C 的一个突出优点便是通过 Cargo 对包进行有序规范的管理。由于 Cargo 对包的严格限制，开发者需要慎重考虑每一个包的功能划分和接口设计，潜移默化地要求了内核开发者需要具有一定的模块化开发思想。如何设计、打磨好一个模块，让其接口尽可能清晰简明，并复用到其他尽可能多的内核中，是模块开发者需要深思熟虑的内容。

在上述分析的基础和清华大学陈渝老师、向勇老师的指导下，我们决定基于操作系统 ArceOS（Unikernel 架构）和 seL4（微内核），进一步使用 Rust 开发完善模块化的系统，组合出微内核、宏内核的内核架构，提出了数条模块化系统的设计原则，并且将其实践在 Starry 等 OS 中，验证了将内核模块化的可行性。我们还进一步完善了 Starry 等 OS，支持 Dora Tokyo 等实际应用，并通过复用组件带来的工作量减负，展示了模块化系统的实用性。

工作内容说明

1. 我们总结、提出了数条模块化系统的设计、开发原则，并且将其实践在 Starry, reL4 和其他内核上。
2. 我们基于已有的操作系统，按照模块化的思想进一步开发扩展，在 ArceOS（Unikernel 架构）的基础上开发了 Starry，适配宏内核和 Unikernel 架构，用 Rust 重写了 seL4（微内核）得到了 reL4，适配微内核架构。在开发这些内核的过程中，我们也在贯彻组件化的思想，有意识地去划分模块。
3. 我们进一步完善 Starry 等内核，适配更多应用，从而打磨模块的功能和正确性，让其更具备被其他内核复用的价值。
4. 我们利用已有的 Starry 模块尝试开发新内核，实现了部分模块的复用，证明了模块的可复用性。
5. 我们对 Starry 使用相同的测例进行了不同架构（宏内核和 Unikernel）下的性能测试，其性能差距与传统的宏内核和 Unikernel 差距相近，侧面验证组件化并不会对性能带来明细的影响。

本文组织架构

- 第一章是概述，介绍本文的研究背景、工作内容和意义。
- 第二、三章介绍组件化内核的概念和探索组件化内核的必要性。
- 第四章总结了组件化内核在设计和实际开发过程的一些实用原则。以这些原则指导组件化内核开发可以明细架构设计，减少开发阻力。
- 第五章介绍了我们在组件化内核上的实际探索成果，包括其工作量、成果特点以及反映出来的组件化内核的优势。
- 第六章是对我们的探索成果的效果展示。
- 第七章是对我们工作的总结和展望。

组件化内核的概念

组件的概念

组件化，作为**软件工程领域的一项核心理念**，已在众多实践场景中展现出其非凡的成效与深远影响。起初，组件化的构想聚焦于代码的复用性，旨在通过共享代码片段来加速开发进程。然而，随着这一理念的深化与拓展，每个精心设计的组件不仅承载了特定的功能实现，还配备了标准化的接口，这一转变极大地促进了开发效率的提升、维护成本的降低，以及系统可扩展性和可重用性的显著增强。

组件化的实施，为软件开发带来了前所未有的并行处理能力。它鼓励并促进了不同开发者之间的协同作业，使他们能够并行不悖地专注于各自负责的组件开发，有效减少了团队间的依赖与等待时间。这种高度的独立性不仅加速了项目的整体进度，还显著降低了因组件间复杂依赖关系而可能引发的风险。

更为重要的是，一个经过精心定义与验证的组件，如同构建软件大厦的预制砖块，能够在多个项目或系统中灵活复用，极大地避免了重复劳动与资源浪费，真正实现了“一次编写，到处运行”的愿景。这种高度的可重用性，不仅提升了开发效率，还确保了代码质量的一致性与稳定性。

在后续的维护阶段，组件化的优势同样显著。通过将系统划分为清晰界定的组件，开发者能够更加精准地定位问题所在，从而实施更加高效、精准的修复与升级策略。此外，由于组件间的松耦合设计，对某一组件的单独升级或改造几乎不会影响到系统的其他部分，这极大地降低了维护的复杂性与成本，确保了系统的持续稳定运行。

总之，组件化开发方式以其独特的优势，在大型软件工程的构建中发挥着不可替代的作用。

组件化应用在内核

组件化内核，顾名思义，就是**使用组件化的方式来构造内核**。它将操作系统的核心——内核，视为一项复杂的软件工程并使用组件化的思想进行编写。在这一过程中，我们打破了传统内核开发的界限，让**各个组件独立开来编写，由专门的人负责维护**，使得程序员能够如同**编写普通程序般自如地编写内核模块**，极大地提升了开发的灵活性与效率。

现在也常有内核宣称自己是以组件化的思想来进行开发的。但我们认为**本文提到的组件化的思想和传统的“组件化”不一样的地方**在于：传统的内核组件一般是从内核中分离出去的，更多情况下还是适用于自己开发的内核，但我们希望内核开发者可以像**选择第三方库一样选择各种组件**，从而按照自己需求快速地组装出一个可用的内核。这就要求组件可以更加广泛地复用到各个内核中，对组件的粒度、功能划分和模块设计提出了很高的要求。

我们的长远愿景，是通过 **深入探索与实践组件化内核的构建之道，提炼并总结出一套系统化、可复用的方法论原则**。这些原则将不仅指导我们如何高效、有序地组织内核的各个组件，更将作为宝贵的经验财富，为后来者提供明确的指引。同时，我们致力于通过实际案例的展示，向业界证明：在多元化的系统构型环境中，组件化内核设计方法展现出其无与伦比的普适性与强大生命力。无论是面对何种复杂的系统需求，组件化内核都能以其独特的优势，助力我们打造出更加健壮、可扩展且易于维护的操作系统核心。

探索组件化开发内核的必要性

正如 Unikraft 对 linux 的批评 提到的内容，类似于 Linux 这样的宏内核，各个部分之间的依赖相当紧密（如下图），以至于单独替换掉其中一个部分是一项令人生畏的工作。

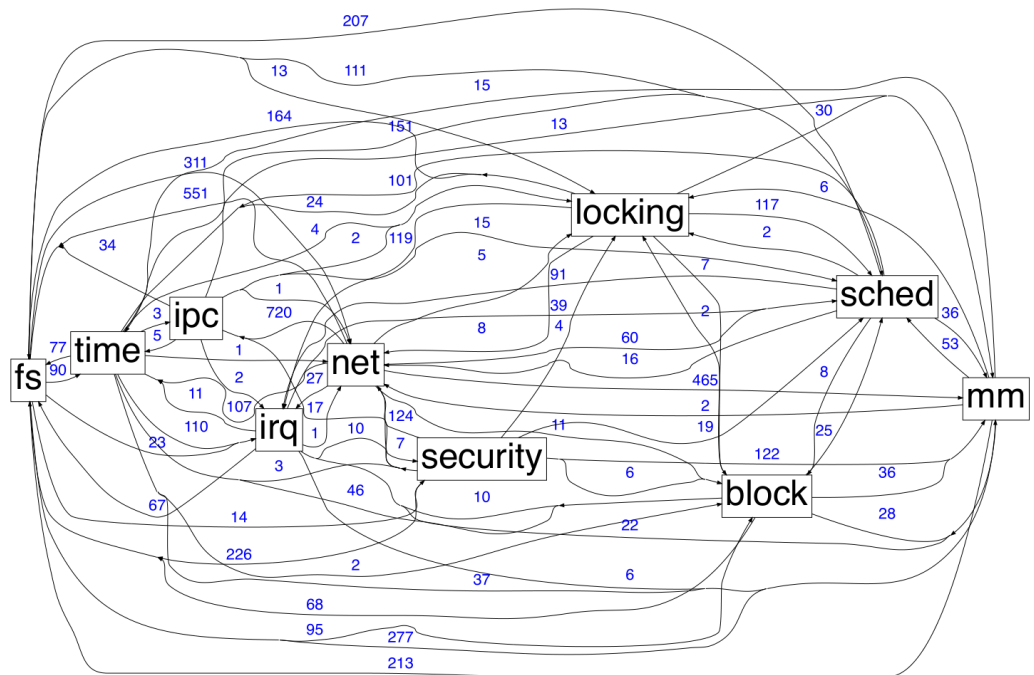


Figure 1. Linux kernel components have strong inter-dependencies making it difficult to remove or replace them.

- 鉴于此，Unikraft 率先踏入了完全模块化内核的探索之旅，做出了如下工作：
- 1. 创新性地**以微库形式呈现一系列高度可配置的操作系统功能模块**。
 - 2. 支持**自由配置**所需要的内核功能模块。这一设计赋予了用户前所未有的灵活性，能够轻松地构建基线中添加或移除所需模块，通过精细组合这些微库，打造出专为特定应用需求量身定制的Unikraft。
 - 3. 为这些模块精心设计了清晰定义的 API 接口，让用户能够基于性能优化、资源最小化等多样化需求，在**同一组件的不同实现间做出选择并灵活组合**。

在当前的IoT（物联网）、智能自动驾驶等前沿领域，实际需求往往聚焦于运行少数几个高度定制化的软件，而非全面依赖一个泛用型操作系统。这些场景常伴随着资源限制、对系统调用速度的高要求以及对实时性的严苛标准。在此背景下，传统Linux宏内核的“大而全”模式显得愈发笨重，难以满足快速变化且多样化的需求。相比之下，组件化操作系统的优势凸显无遗，其卓越的组件可替换性和模块化设计，为精准匹配各类特定场景需求提供了强有力的支持。

可以预见的，内核包含的功能会越来越复杂，涉及的代码量会越来越大（参考概述中提到的研究背景）。如果不加管理的话，模块之间的依赖关系会越来越复杂。我们认为，未来的内核必然不会是一个完全的自包含系统，而是由诸多可替换组件所构成的灵活的动态的内核。因此，进行多构型组件化内核的研究和实践在我们看来有着长远的意义。

■ 组件化内核的设计和开发原则

在上一个章节，我们讨论了组件化内核开发的必要性和意义。组件化内核开发作为一个新的开发模式，相比于传统内核开发模式提出了许多全新的需求。如何根据这些新的需求确定合适的设计模式、开发模式，是组件化内核需要着重考虑的问题。

组件化内核开发模式相比于传统开发模式有许多不同的要求：

1. 模块需要考虑复用性。传统的内核虽然也有考虑所谓“组件”，但是这些一般是从软件工程的角度的上考虑，比较方便自身内核的功能的管理，而不是考虑设计服务更多内核。为了保证模块具有复用性，我们需要着重考虑模块本身包含的功能以及对外提供的接口，这涉及到内核组件设计上的问题。
2. 支持多人开发各自的模块：组件化内核的一个很重要的特征就是支持多人同时开发，每一个人各自维护自己的模块，减少内核维护的压力。结合上一点的模块复用性，这就要求要多人同时在多个仓库上进行开发，这与传统的单仓库开发内核不同，对开发模式和开发规范也提出了挑战，需要我们提出一些开发规范。

依据上面的分析，我们学习其他的内核和复杂应用（如 Android）的经验，并整合自身的开发历程，在**组件化内核设计和实际开发**上分别提出了数条参考原则，用来帮助更好地推进组件化内核的实现。

■ 组件化内核的设计原则

■ 设计目标

组件化内核的一个很重要的问题是组件的界定范围，即什么样的功能可以被划分为组件模块。传统对可分离组件的划分有如下两类：功能固定的代码与接口规范明确的代码。前者表现的形式常为一些固定的算法、数据结构或者较为简单的功能包装，如排序算法、红黑树等，这些多以库的形式被包装起来为他人使用。另外还有一类，即存在明确的接口规范的代码。这类代码本身可能功能较为复杂且难以实现，但是对外有一套统一的接口。调用这个模块的用户只关心接口函数以及模块本身是否正确实现了接口语义，至于如何实现在多数情况下并不会为用户所关注。这类模块的实例即文件系统、驱动等模块，在 Linux 中有 `vfs` 等接口规范作为参考，在接口差距不大的时候可以通过简单适配互相兼容。

在 Linux 等内核中，已经对上述提到的两类模块做了功能划分和一定程度的解耦合，但是其只占据了内核中的一部分内容。内核中的地址空间管理、调度模块等内容与具体内核实现关联较大，在传统内核中常与众多模块彼此调用而互相耦合，导致抽象、维护的成本非常高。目前的模块化工作对这部分功能的尝试实践相比于前者的驱动、文件系统等模块还比较少。

我们开发 Starry 时期望不仅是划分小的功能模块（如一些简单的 `bitset` 算法）、传统的文件、网络、驱动、硬件抽象层模块，还希望更进一步，尝试将调度模块、地址空间等模块尽可能解耦出来，通过定义一套接口规范，让各个内核实现指定接口，从而能够复用这些核心模块，减少内核开发、维护的工作量，更好地打磨内核的具体细节和结构。我们期望最终形态下，OS 内部仅保留一个运行时，选择所需要的模块并且实现其需要的接口，从而组合成一个完整的可运行的内核。

为了实现这一个目标，我们分析了已有的一些内核的设计，总结提出了**数条组件化开发的设计原则**。我们将这些原则应用在自己的内核中，从而尽可能地降低内核模块间的耦合度，实现模块化内核的开发目标。

设计原则

在划分模块的时候，如何确定模块需要实现的功能、如何确定模块对外提供的接口，是模块设计者需要深思熟虑的内容。前者决定了模块需要依赖的子模块的内容，是向底层的考虑，后者决定了模块是否能够较为方便地和其他同类型模块一起被复用，是向顶层的考虑。

因此 Rust 组件化内核的设计原则需要同时考虑两个方面，分别是模块功能划分和接口设计。为此我们提出了 6 条设计原则。

底层或固定算法模块独立

底层（未被其他模块依赖）模块和其他功能固定的模块可以独立，如 bitmap、排序算法等模块。

这类模块由于比较简单，其对外提供接口时较为自由，按照自身功能提供即可。内核在调用的时候按需调用即可。在 Starry 中这类模块的例子有 `handle_table` 模块，它对裸的函数指针进行包装，用来为中断处理函数向量或者其他内容提供便捷的类型。

```
use handler_table::HandlerTable;

static TABLE: HandlerTable<8> = HandlerTable::new();

TABLE.register_handler(0, || {
    println!("Hello, event 0!");
});
TABLE.register_handler(1, || {
    println!("Hello, event 1!");
});

assert!(TABLE.handle(0)); // print "Hello, event 0!"
assert!(TABLE.handle(2)); // unregistered
```

功能明确且与 OS 松耦合模块可以独立

对于功能简单、明确且与内核松耦合（需要的交互接口较少）的模块可以独立。它通过向外界提供一个 trait，描述自身需要调用者实现的接口。调用者只要实现了这个 trait，便可以使用模块的功能。这类模块由于功能比较明确简单，一般不存在多种类替换的需求，对上层提供的接口也不需要明确的规范限制。

一个例子是 `page_table`，他需要内核提供分配物理页以及虚实地址转化两个接口函数即可。然后本身 `page_table` 的实现有较为固定的算法，因此可以直接独立出来。对于不同的内核来说只要实现了前两个给定的接口函数，就可以自由调用页表提供的功能。在 Starry 中，相关的 trait 定义与实现如下：

```
/// The low-level **OS-dependent** helpers that must be provided for
/// [PageTable64].
pub trait PagingIf: Sized {
    /// Request to allocate a 4K-sized physical frame.
    fn alloc_frame() -> Option<PhysAddr>;
    /// Request to free a allocated physical frame.
```

```

fn dealloc_frame(paddr: PhysAddr);
/// Returns a virtual address that maps to the given physical address.
///
/// Used to access the physical memory directly in page table implementation.
fn phys_to_virt(paddr: PhysAddr) -> VirtAddr;
}

/// Implementation of [PagingIf], to provide physical memory manipulation to
/// the [page_table] crate.
pub struct PagingIfImpl;

impl PagingIf for PagingIfImpl {
    fn alloc_frame() -> Option<PhysAddr> {
        global_allocator()
            .alloc_pages(1, PAGE_SIZE_4K)
            .map(|vaddr| virt_to_phys(vaddr.into()))
            .ok()
    }

    fn dealloc_frame(paddr: PhysAddr) {
        global_allocator().dealloc_pages(phys_to_virt(paddr).as_usize(), 1)
    }

    #[inline]
    fn phys_to_virt(paddr: PhysAddr) -> VirtAddr {
        phys_to_virt(paddr)
    }
}

```

功能复杂，但与 OS 无关的模块可以独立

对于与 OS 无关的组件，如文件系统、驱动、网络协议栈等模块，他们本身的实现较为复杂，但与具体 OS 本身（何种架构、同步/异步协程）等实现关联不大或基本无关。

这类模块由于与 OS 无关，一般存在着多种实现方式，如驱动、文件系统等。内核在调用这类模块的时候，由于有多种选择，对每一个驱动都需要做适配，因此我们需要定义一个通用的内核接口规范，要求驱动在实现了这些接口之后，就可以直接接入内核使用。

以文件系统为例，Linux 提出了 vfs 等文件系统接口规范，当文件系统实现了 vfs 接口规范，就可以直接接入内核。

同样以文件系统为例，Starry 支持了四个不同的文件系统实现，分别是 fat32，lwext4 和两个用 Rust 编写的两个不同实现的 ext4，具体的链接如下：

模块名称	文件系统类型	仓库	备注
fatfs	fat32	https://github.com/rafaelh/rust-fatfs	
lwext4	ext4	https://github.com/elliott10/lwext4_rust	原先是用 C 编写，改动提供了 Rust 的接口
another_ext4	ext4	https://github.com/LJxTHUCS/another_ext4	Rust 实现的 ext4
ext4_rs	ext4	https://github.com/yuoo655/ext4_rs.git	Rust 实现的 ext4

为了便捷地让不同文件系统的开发者能够迅速接入 ArceOS/Starry，我们仿照 vfs 的形式，指定了文件系统需要在 Starry 中实现的接口。当文件实现了这些接口，就可以快速地接入到 Starry 中进行使用。具体接口定义如下：

```

/// A wrapper of [`Arc<dyn VfsNodeOps>`].
pub type VfsNodeRef = Arc<dyn VfsNodeOps>;

/// Alias of [`AxError`].
pub type VfsError = AxError;

/// Alias of [`AxResult`].
pub type VfsResult<T = ()> = AxResult<T>;

/// Filesystem operations.
pub trait VfsOps: Send + Sync {
    /// Do something when the filesystem is mounted.
    fn mount(&self, _path: &str, _mount_point: VfsNodeRef) -> VfsResult {
        Ok(())
    }

    /// Do something when the filesystem is unmounted.
    fn umount(&self) -> VfsResult {
        Ok(())
    }

    /// Format the filesystem.
    fn format(&self) -> VfsResult {
        ax_err!(Unsupported)
    }

    /// Get the attributes of the filesystem.
    fn statfs(&self) -> VfsResult<FileSystemInfo> {
        ax_err!(Unsupported)
    }

    /// Get the root directory of the filesystem.
    fn root_dir(&self) -> VfsNodeRef;
}

/// Node (file/directory) operations.

```

```

pub trait VfsNodeOps: Send + Sync {
    /// Do something when the node is opened.
    fn open(&self) -> VfsResult {
        Ok(())
    }

    /// Do something when the node is closed.
    fn release(&self) -> VfsResult {
        Ok(())
    }

    /// Get the attributes of the node.
    fn get_attr(&self) -> VfsResult<VfsNodeAttr> {
        ax_err!(Unsupported)
    }

    // file operations:

    /// Read data from the file at the given offset.
    fn read_at(&self, _offset: u64, _buf: &mut [u8]) -> VfsResult<usize> {
        ax_err!(InvalidInput)
    }

    /// Write data to the file at the given offset.
    fn write_at(&self, _offset: u64, _buf: &[u8]) -> VfsResult<usize> {
        ax_err!(InvalidInput)
    }

    /// Flush the file, synchronize the data to disk.
    fn fsync(&self) -> VfsResult {
        ax_err!(InvalidInput)
    }

    /// Truncate the file to the given size.
    fn truncate(&self, _size: u64) -> VfsResult {
        ax_err!(InvalidInput)
    }

    // directory operations:

    /// Get the parent directory of this directory.
    ///
    /// Return `None` if the node is a file.
    fn parent(&self) -> Option<VfsNodeRef> {
        None
    }

    /// Lookup the node with given `path` in the directory.
    ///
    /// Return the node if found.
    fn lookup(self: Arc<Self>, _path: &str) -> VfsResult<VfsNodeRef> {
        ax_err!(Unsupported)
    }

    /// Create a new node with the given `path` in the directory
    ///

```

```

/// Return [Ok()](Ok) if it already exists.
fn create(&self, _path: &str, _ty: VfsNodeType) -> VfsResult {
    ax_err!(Unsupported)
}

/// Remove the node with the given path in the directory.
fn remove(&self, _path: &str) -> VfsResult {
    ax_err!(Unsupported)
}

/// Read directory entries into dirents, starting from start_idx.
fn read_dir(&self, _start_idx: usize, _dirents: &mut [VfsDirEntry]) -> VfsResult<usize> {
    ax_err!(Unsupported)
}

/// Renames or moves existing file or directory.
fn rename(&self, _src_path: &str, _dst_path: &str) -> VfsResult {
    ax_err!(Unsupported)
}

/// Convert &self to [&dyn Any][1] that can use
/// [Any::downcast_ref][2].
///
/// [1]: core::any::Any
/// [2]: core::any::Any#method.downcast_ref
fn as_any(&self) -> &dyn core::any::Any {
    unimplemented!()
}
}

```

对于其他内核，只要指定自己内核需要的接口的规范格式，同时这类接口较为规范（如与 Linux 的 `vfs` 语义较为接近），那么文件系统开发者便可以以较高的效率实现这些接口，从而快速接入到内核中使用。

OS 相关的模块：设计对应向下向上接口

对于与 OS 具体实现相关的模块，比如任务调度、地址空间管理模块，它们与内核具体的实现细节有关。以任务调度模块为例，它需要访问、操控内核线程任务的资源，并且受锁、信号和系统调用等操作的影响，如互斥锁阻塞任务、信号杀死任务、系统调用调度任务等。对于这类模块，在传统的成熟内核中耦合程度已经非常地高，想要将其涉及的内容抽象出来相当地困难。

但对于 Starry 来说目前的资源依赖关系还没有那么复杂，因此我们可以梳理模块之间的依赖关系，完成以下两类工作：

1. **抽象出调度模块需要内核实现的接口**：当 Starry 实现了这个 trait，就可以复用这个模块。这里的接口与前面提到的页表等模块的接口类似，当内核实现了接口提供的 trait 的时候，便可以将模块接进来，从而正常调用模块提供的功能。

以下为调度模块需要内核实现的接口：

```

trait SchedTask {
    fn current_task() -> Self;

    /// get the task context for task switch
    fn get_ctx(&self) -> TaskContext;
}

```

```
#[cfg(feature = "preempt")]
fn need_preempt(&self) -> bool ;

#[cfg(feature = "monolithic")]
/// get the page table token of the process which the task belongs to
fn get_page_table_token(&self) -> usize ;

#[cfg(feature = "monolithic")]
/// force to set the page table token of the process UNSAFELY
fn set_page_table_token(&self, token: usize);

/// update the time information when the task is switched out
fn time_stat_when_switch_from(&self, current_tick: usize);

/// get the policy of the scheduler
#[cfg(feature = "monolithic")]
fn get_sched_policy(&self) -> SchedStatus;

/// get the cpu set for the task
#[cfg(feature = "monolithic")]
fn get_cpu_set(&self) -> usize;
}
```

需要强调的是，宏内核关于调度策略可能会有很多更多复杂的操作，目前记录的 `cpu_set` 和 `sched_policy` 对应的是 `sched_setscheduler` 和 `sched_setaffinity` 相关的系统调用需要用到的方法。因此内核提供的 trait 可能会随着内核的要求升级而逐渐变得复杂。但无论 trait 如何变迁，我们应当保证当内核正确实现了接口的语义时，可以快速地服用这个模块，就像接入一个文件系统一样便捷。

2. **规定内核需要模块实现的接口**：当模块实现了这个接口，就可以让上层调用这类模块的功能，而不需要关心具体是哪个模块在发挥作用。

这部分内容和上文提到的文件系统的 `vfs` 类似。当我们为外部模块定义了需要实现的接口的时候，则上层模块可以根据这个接口较为方便地进行调用，而不用关心底层的调度模块具体是如何实现的。包装这一层的意义在于 **为存在多种实现可能性的模块提供替换的可能**。以调度模块为例，它存在着多种调度算法，我们希望在更换不同调度算法的模块的时候，**上层模块不会感受到模块的变化，也不需要做出源代码级别的修改**。而想实现这一步，就需要在引入模块之后，对模块向上层提供的函数做一层统一的封装。

这里相比于 `vfs` 的问题在于 **并没有一个明确的、为较多人接受的接口规范制定**。由于不同的内核实现侧重点不同，对任务调度的要求不同，因此所需要的接口很多情况下也是不同的。为了尽可能保证规范性，我们 **仿照 `rust-std` 的接口语义** 定义了调度模块向上提供的接口。

Starry 要求调度模块实现的接口如下。

```
/// Initializes the task scheduler (for the primary CPU).
pub fn init_scheduler();

/// Initializes the task scheduler for secondary CPUs.
pub fn init_scheduler_secondary();

/// Spawns a new task with the given parameters.
///
/// Returns the task reference.
pub fn spawn_raw<F>(f: F, name: String, stack_size: usize) -> AxTaskRef
where
```

```

F: FnOnce() + Send + 'static;

/// Spawns a new task with the default parameters.
///
/// The default task name is an empty string. The default task stack size is
/// [ axconfig::TASK_STACK_SIZE ].
///
/// Returns the task reference.
pub fn spawn<F>(f: F) -> AxTaskRef
where
    F: FnOnce() + Send + 'static;

/// Set the priority for current task.
///
/// The range of the priority is dependent on the underlying scheduler. For
/// example, in the [CFS] scheduler, the priority is the nice value, ranging from
/// -20 to 19.
///
/// Returns `true` if the priority is set successfully.
///
/// [CFS]: https://en.wikipedia.org/wiki/Completely\_Fair\_Scheduler
pub fn set_priority(prio: isize) -> bool;

/// Current task gives up the CPU time voluntarily, and switches to another
/// ready task.
pub fn yield_now();

/// Current task is going to sleep for the given duration.
///
/// If the feature `irq` is not enabled, it uses busy-wait instead.
pub fn sleep(dur: core::time::Duration);

/// Current task is going to sleep, it will be woken up at the given deadline.
///
/// If the feature `irq` is not enabled, it uses busy-wait instead.
pub fn sleep_until(deadline: axhal::time::TimeValue);

/// wake up task
pub fn wakeup_task(task: AxTaskRef);

/// Current task is going to sleep, it will be woken up when the given task exits.
///
/// If the given task is already exited, it will return immediately.
pub fn join(task: &AxTaskRef) -> Option<i32>;

/// Exits the current task.
pub fn exit(exit_code: i32) -> !;

```

对于具体的线程任务等定义，为了统一起见，我们也抽象为了单独的模块，向上提供资源的访问和修改方法。但是对于其他内核开发者来说，他们完全可以绕过 Starry 的线程任务定义，自己定义适合内核的任务结构，并且按照我们定义的接口规范实现对应的语义，从而复用我们提供的模块。

模块功能归属原则

当一个模块设计出来的时候，为了保证可以被各种类型的内核共享，它应当不包含过多的与内核架构、指令架构相关的特殊操作，每一个模块**只实现各自需要的功能以及子模块中需要扩展的功能**。

举个例子，Starry 是基于 ArceOS 进行设计的，而 ArceOS 作为一个 Unikernel，其模块一开始设计时仅满足 Unikernel 的必要需求，而对于宏内核的文件系统初始化（如初始软连接建立、meminfo 等文件信息的初始化）都没有考虑。当将 ArceOS 扩展为宏内核时，Starry 需要进行宏内核的文件系统初始化。这时候这部分修改不应该添加在 axfs 中，而应当添加在 Unix 宏内核启动库 axstarry 中与文件系统相对应的 mod 中。若是将这部分工作移动到 axfs 中进行，那么由于宏内核文件系统初始化的时候需要写入地址布局、进程信息等一系列内容，就需要依赖进程控制、地址空间管理等模块，而这些是原先的文件系统不应该依赖的，就会导致引入不必要的依赖甚至导致循环依赖的问题。因此这里便体现了上面提到的原则，axstarry 除去实现本身需要的宏内核 Unix 启动命令读取的功能，还需要实现文件系统等子模块中需要在 Unix 宏内核中扩展的功能。

循环依赖解决

我们需要先解释一下循环依赖的概念：在 Rust 中，循环依赖（circular dependency）指的是两个或多个模块（或 crate）之间互相依赖，导致编译器无法确定哪个模块应该先编译。这种情况会引发编译错误，因为 Rust 的编译器需要确保依赖关系是有序的，以便能够正确地解析和编译代码。

假设有两个模块 **A** 和 **B**，如果模块 **A** 依赖于模块 **B**，同时模块 **B** 也依赖于模块 **A**，就会形成一个循环依赖：

```
// mod_a.rs
mod mod_b;

pub struct A;

impl A {
    pub fn new() -> Self {
        B::use_b(); // 使用 B 中的函数
        A
    }
}
```

```
// mod_b.rs
mod mod_a;

pub struct B;

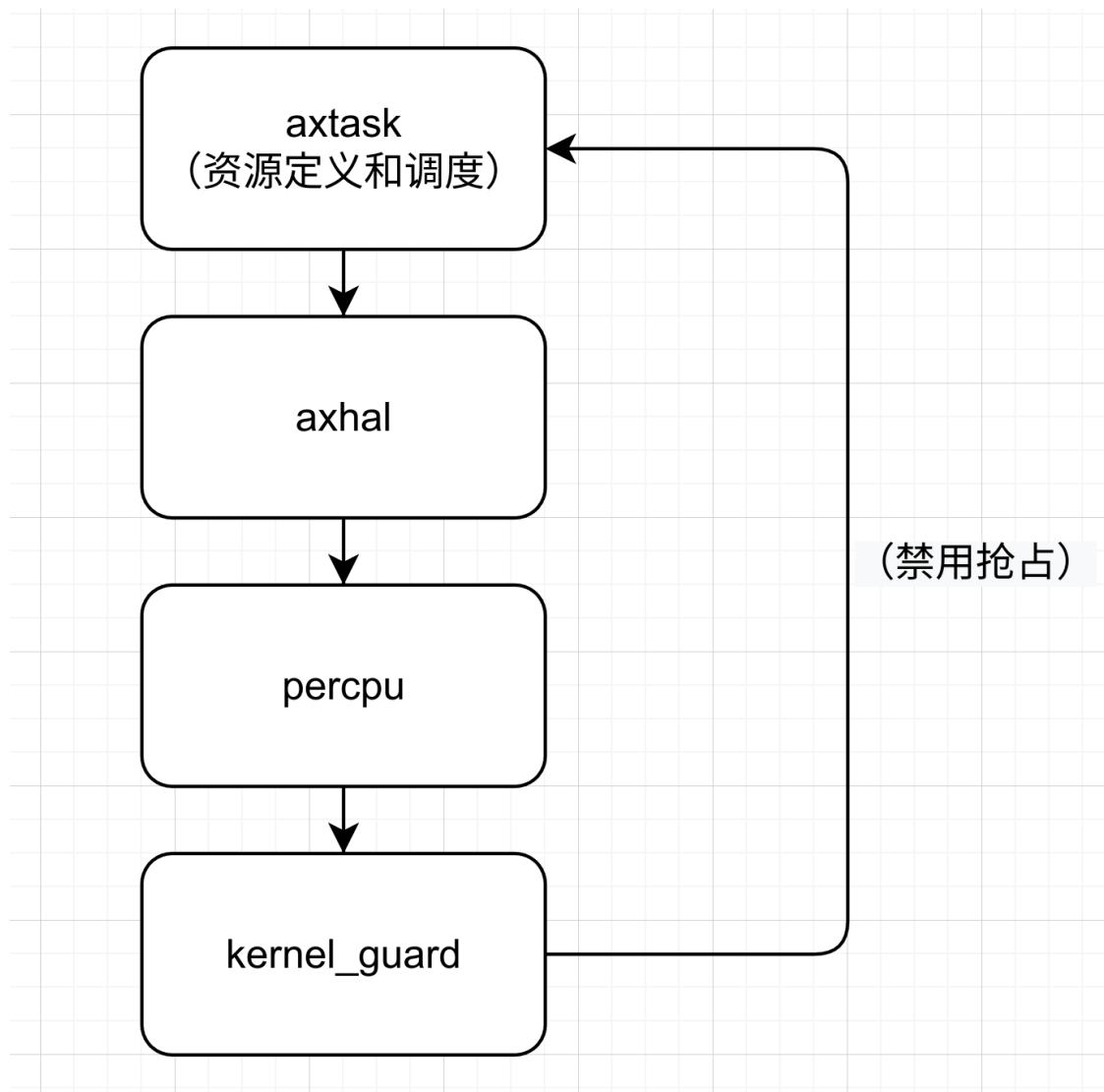
impl B {
    pub fn use_b() {
        A::new(); // 使用 A 中的函数
    }
}
```

在这个例子中，**mod_a** 依赖 **mod_b**，而 **mod_b** 也依赖 **mod_a**，这就形成了循环依赖。Rust 编译器会因为无法决定哪个模块应该先编译而报错。

解决循环依赖的方法如下：

1. **重构代码**：将共享逻辑提取到一个第三方模块中，避免直接依赖。例如，将 **A** 和 **B** 的共享部分提取到一个新的模块 **C** 中，然后 **A** 和 **B** 都依赖 **C**。
2. **使用接口 (trait)**：将依赖关系抽象成 trait，然后在需要的地方实现这个 trait，这样可以打破直接的依赖关系。
3. **利用惰性初始化**：有时候可以通过推迟依赖的使用，使用引用 (**&**)、指针 (**Box** 或 **Rc**) 等手段来避免循环依赖。
4. **分离声明与实现**：将依赖的声明放在一个地方，具体的实现放在另一个地方，以减少模块之间的直接依赖。

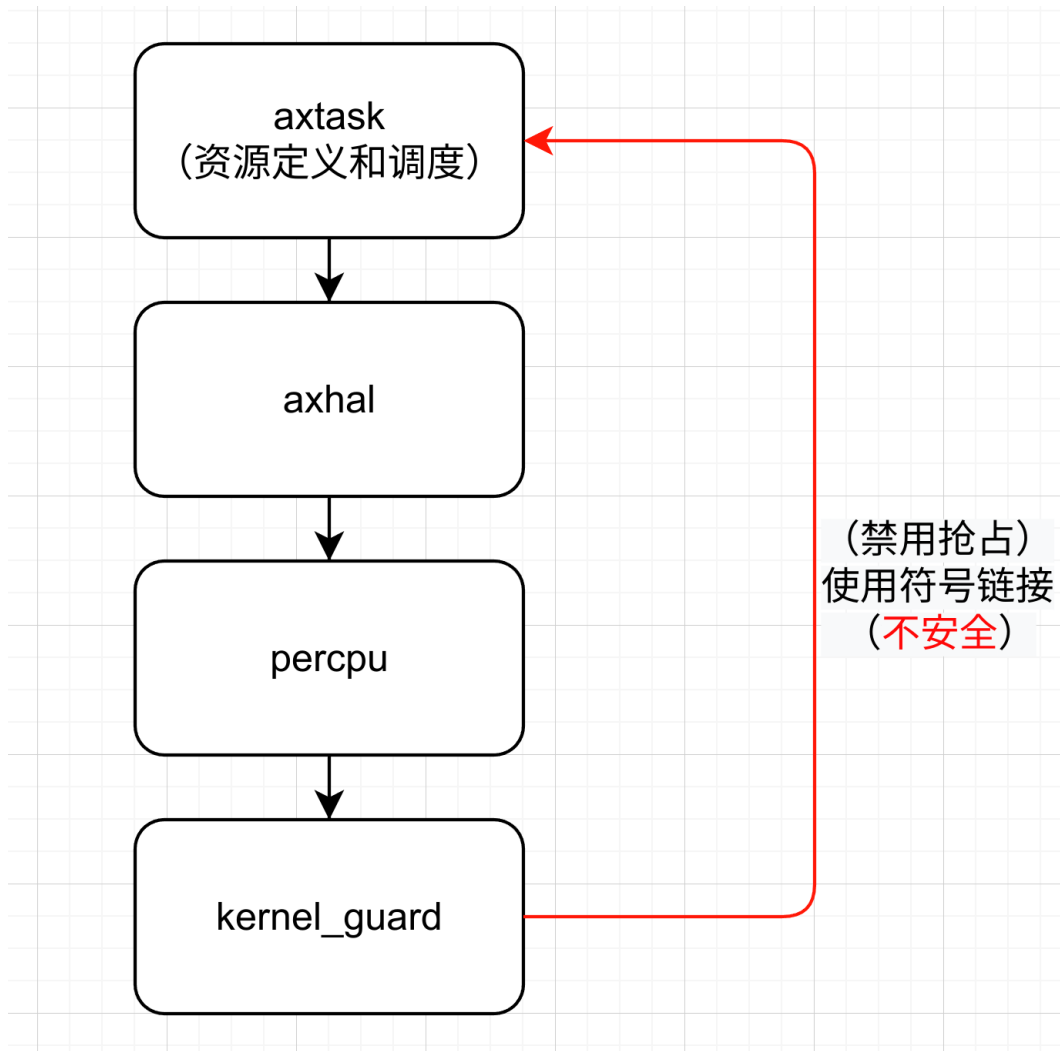
在内核中，当模块划分的粒度逐渐变细，模块数量也会开始上升。Starry 目前已经拆分出来了 61 个模块，不同模块之间的数目上升会导致依赖关系的进一步复杂化，由于不合理划分功能而出现循环依赖的可能性会越来越高。事实上，就算进行了合理的功能划分，也仍然会有循环依赖的问题。Starry 中的一个较为典型的循环依赖问题如下：**axhal** 硬件抽象层依赖于 **percpu** 读取每一个 CPU 的信息，而 **percpu** 读取需要禁用抢占，抢占需要修改任务资源属性，任务资源属性定义在 **axtask** 模块下，与 **axtask** 调度模块有关，**axtask** 调度模块又依赖于 **axhal** 的硬件接口，从而形成循环依赖。



为了解决循环依赖问题，我们提出了如下两种解决思路：

- **通过不安全的符号链接绕过去 (类似 C)**：在原生的 ArceOS 中使用了这种方法，具体使用 https://github.com/arceos-org/crate_interface 模块，通过标注过程宏利用符号链接来进行函数的调用，打破了 Cargo 对包的依赖关系的限制，从而解决了循环依赖的问题。

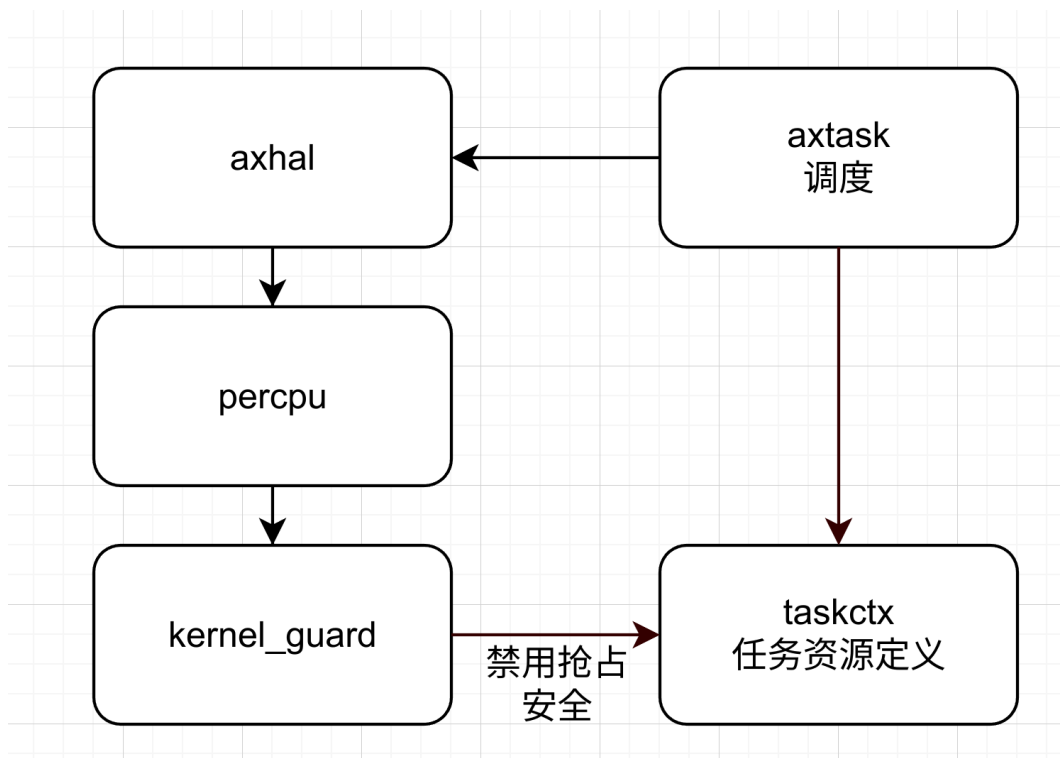
在上面的例子中，ArceOS 取消了 `percpu` 在 Cargo 上对 `axtask` 的依赖，在 `percpu` 禁用抢占这里使用了符号链接，通过符号链接直接调用 `axtask` 的禁用抢占函数，从而打破了循环依赖。



这种方法可以快速地解决问题，并且保持原来各个模块的功能一致，不需要对模块内容做大的修改。但缺点是违背了 Rust 的安全保证以及模块依赖关系，本质上和 C 语言直接通过函数进行链接的方法是类似的，存在隐藏的安全问题，所以需要尽可能减少使用。

- **资源定义和实现分开**：资源定义和实现分开原则是我们提出的一种用于划分模块、处理循环依赖的新方法。在传统的模块划分方法下，我们习惯将资源定义和对它的操作放在一个模块下，因此上述提到的抢占锁、调度模块和任务本身的资源在传统意义上都是定义在一起的，可以集中在一个模块中进行处理。对于 Linux 等 C 语言编写的内核，整个内核就是一个大模块，所以不会有这个问题。但是为了划分具体的模块，减少单个模块的代码量和复杂度，我们希望将调度、互斥锁和任务资源定义分开。我们视任务结构定义为**资源的定义**，而互斥锁引起的调度、调度模块本体都是对任务结构中**与调度相关的资源的操作**。体现在 Starry 中，`taskctx` 模块是我们的任务结构的定义，而 `kernel_guard`，`axtask` 分别是对抢占锁、调度模块模块的定义。这两个模块会引用 `taskctx` 提供的资源并且进行操作。

在这个定义和实现分开的方式下，我们之前提到的将调度模块独立为单个模块的操作进行也会更加方便，可以和**内核强相关的任务结构定义解耦合**。另外上面提到的**循环依赖问题也被打破了**。`axhal` 依赖于 `percpu`，而 `percpu` 依赖于 `taskctx`，直接修改任务的资源。`axtask` 进行调度的时候读取 `taskctx` 的资源并且进行操作，两个模块之间进行共享。



第二种解决方法可以在保证 Rust 语言安全性的前提下解决循环依赖问题，还能更进一步追求内核模块间的 [单向依赖关系](#)，理清楚模块之间自上而下的依赖关系。但是它的不足之处在于需要进一步拆分模块内容，有时候并不是所有的模块资源拆分都是合理的，过多地拆分会导致模块过于细碎而难于管理。

以上提到的两种解决循环依赖的方法各自有其优势和不足之处，在 ArceOS 中更多采用第一种方法解决，在 Starry 中更多地采用的是第二种方法来解决循环依赖问题，从而尽可能实现单向依赖的结构。实际可以根据不同内核的情况各自选择。

组件化内核的开发模式

由于每个模块都是具有自己功能的，特定的，独立的仓库，因此，在整个内核的开发过程中，需要同时对多个仓库进行处理。然而，在内核的实际开发过程中，如何进行多个仓库之间的同步，如何解决多人代码 review 问题，如何进行有效的 CI 测试都是我们在进行多构型组件化内核实践中需要解决的问题。

为了便于多仓开发，以及针对使用的工具存在的不便捷性，我们前后更换了几种具体的工具来进行管理。我们的模块化仓库的开发方式也是一定程度上基于我们使用的多仓开发工具而确定下来的。在实际中，依据我们对其他仓库的学习以及自身开发经验的总结，我们提出了如下的开发模式，用于帮助进一步规范组件化多仓内核开发。

简单的 git 及其协作

最开始，我们选择使用简单的 git，具体来说我们维护了一个大仓库，在这个大仓库中，又包含了若干个文件夹（common/cspace/vspace/ipc/task/kernel）用于表示我们的模块。

在这种情况下的协作，最开始是每个人独立的选择一个分支，每个人都维护自己的分支和主分支的同步。

但是这很快出现了问题，具体来说，是由于本身的多人协作，导致若干分支之间互相 merge，最终合并到主分支上之后，出现大量的merge冲突，并且曾经出现过的merge代码会重复的 merge。同时，在这种情况下，对于主分支的维护者来说也是一个灾难。

为了解决这个问题，我们试图规范化我们的协作。做出如下约定

- git 分支不再以每个人维护自己的代码的方式，而是按照某个 issue 的方式去维护
- 禁止非主分支之间的 merge
- 对于其他分支和主分支之间的合并，每次需要提交之前都需要拉取主分支最新的更改，然后合并好之后，再合并回主分支
- 每个人都对各自的提交负责，并完善 github 的 ci。

在这种协作方式下，相比于最开始的无序，协作效率有了一定的提高。

但是对于模块化的目的来说，这种简单的 git 仓库，并不能很清楚明白的让其他人快速地复用我们的代码，也无法达成我们模块化的目的。

因此很快，我们出于模块化的考虑，将整个代码拆分到了多个子仓库中去。

repo 工具下的协作

在这种方式下，我们试图构建多个小仓库，来达到模块化的目的。同时，为了管理多个小仓库的情况，我们开始寻求别的工具来替代简单的git来进行代码的组织管理，因为这已经涉及到了多个仓库了。

最开始我们瞄准的是 github 的 submodule 功能。但是 submodule 存在以下的局限性：我们的代码本身是在快速演进当中，需要对多个模块的代码进行大规模的调整，使用 submodule 之后，将我们的更改分发到多个小仓库颇为不便，当然这个问题在各个子模块稳定下来之后会相对好一些。

因此我们调研一圈之后，转而使用 repo 工具进行仓库的构建

repo 本身是 google 为了开发 Android 而开发的一个工具，本身也是依托于 git，因此并不是对上述简单 git 仓库的一个直接替换。

对于我们的仓库，只需要

```
repo init -u You_git_addr
repo sync
```

即可快速的同步

然后配置了 default.xml 文件中配置好每个小仓库的源位置，源分支，目标文件夹，即可实现自动的把小仓库进行拉取，如下所示（限于篇幅只列出部分）：

```
<?xml version="1.0" encoding="UTF-8"?>

<manifest>
    .....
    <project name="seL4_libs.git" remote="seL4" path="projects/seL4_libs"
revision="2ca525429e7b4f5abbc1fc694d3aeaff5eb6e7db" upstream="master" dest-branch="master"/>
    .....
</manifest>
```

这个时候，每个小仓库内部仍然是一个个独立的 git 文件夹。

但是，在具体使用的时候，我们仍然出现了诸多问题，针对这些问题，我们也继续演进了协作的方式。

最大的问题同样来自于将本地更改后的代码和远程的多个小仓库进行同步。

由于我们的各个模块仍然还在快速演进当中，所以当某个模块发生变更的时候，必然会对其他诸多模块产生影响，因此一次改动往往涉及多个小仓库。而按照之前约定的协作规范，通常不应该直接向主分支进行 commit，而是一如之前约定的那样，push 到其他的分支之后，进行合并再提交到主分支。

这导致了一个问题，如果我试图修改多个模块，我就必须在本地调整过代码并验证通过之后，把其他多个小仓库都新建一个分支，进行提交，然后合并到对应小仓库的主分支里面去。这一步，涉及到多个仓库的多个分支以及他们的合并和提交，需要的操作数量过多，导致人类很容易在操作过程中因为漏掉步骤或者敲错代码导致出错。

而一旦出错，需要修复这些个错误，又需要重新涉及到同样类似的操作数量用于修复。这已经变成了一种灾难。

另一个问题是，如果某个人做了一些修改，那么他必须让涉及到多个仓库的事情变成原子的。也就是说，如果某个人 A 已经提交了一部分子仓库的更新，但是与此同时，另一个人 B 试图往别的仓库提交，那么这时候，A 再试图提交到那些仓库就是错误的行为，因为此时该模块的主分支的最新提交已经合并了 A 的提交，但是 B 没有合并进来，就会引起灾难性的问题。

还有一个是关于 CI，由于提交了之后，会立即触发某个仓库的 CI，然后拉取其他模块的仓库。但是这时候，并不是所有的模块仓库，都已经完成了更新，因此，就存在 CI 的报错，这个报错是由于提交的同步导致的。所以并不是真正的 CI 的错误，此时，我们需要手动的让 CI 重新运行。

为此我们提出了更多的代码协作约定来避免这个问题。

- 不再是单个人负责某个提交，而是需要两个人共同对某个提交进行负责，减少因为一个人的失误导致的问题。
- 由于我们协作的人数非常少，因此，对于一次提交只需要在工作群里说一声，即可让其他人知道某人想要提交，这时候其他人会暂时停止提交。不过这个方法在更大规模下则难以适用。
- 在本地做好测试。

git-subrepo 工具下大仓库和多个小仓库的协作

在这一步，我们并不是完全废弃了 repo 的工具，而是在此基础上进行了补充

如之前所说，多个模块的问题在于多个小仓库的同时提交存在复杂性。而提交更为简单则是一个简单的大仓库的优点，因此我们再次寻找到 git-subrepo 工具

我们设计了一个大仓库和对应同步的多个小仓库的方式，用于灵活开发和模块化之间的权衡。

使用这个工具，可以通过简单的

```
git subrepo push sub-module-dir
```

单独 push 某个模块

也可以正常的跟使用单个大仓库一样使用 git 方式提交。

同时，我们基于这种工具构建了 CI，当在大仓库提交了之后，可以自动的向小仓库进行同步，而如果在小仓库单独进行了修改，也会同步到大仓库中去。

Issue 项目管理

我们使用 github 的 issue 等方式进行项目的管理：在组织的 github 中增加一个 project，在其中将任务分配到每个人，并存在相应的 issue，如果存在问题，可以在 issue 下进行讨论。每次pr的过程中必须提及该 issue，并在合并后关闭 issue。每个分支都必须对应一个 issue，而不是按照个人划分，每个分支以个人为单位。一个例子是：<https://github.com/Starry-OS/Starry/issues/13>

总结

我们分别在组件化内核的模块设计和实际开发上提出了数条原则。

在内核的模块设计上，我们提出：

1. 底层模块或者是固定算法的模块可以独立，其不需要考虑和内核之间的接口设计，内核一般按需调用即可。
2. 功能明确且与 OS 松耦合的模块可以独立，其通过 Rust-trait 规定需要内核实现的接口。
3. 功能复杂，但与 OS 无关的模块可以独立。对内核来说存在着多种可选模块，可以定义一个接口规范，要求驱动在实现这些接口之后可以立即接入内核使用。一个例子是 Linux 的 vfs 接口规范。
4. OS 相关的模块：我们需要考虑向上向下设计接口。对上，我们需要抽象出模块需要内核实现的接口，以 trait 的形式呈现。对下，我们规定内核需要模块实现的接口。如果模块实现了这些接口，就可以直接接入内核使用，这是为了方便内核复用不同种类的模块，如多种调度算法模块。
5. 模块功能归属原则：每一个模块 **只实现各自需要的功能以及子模块中需要扩展的功能**，不应该为了实现额外的功能而引入不必要的依赖。
6. 循环依赖解决：当模块化出现循环依赖的问题的时候，我们可以通过符号链接（不安全）或者 **将资源定义和实现分开** 的方式解决循环依赖。

在内核的实际开发上，我们提出：

1. 使用简单的 git 协作开发
2. 仿照 Android 使用 repo 工具进行同等级多仓管理和提交
3. 使用 git-subrepo 工具管理大仓库和小仓库之间的协作问题
4. 通过 github 的 project 和 issue 来跟踪提交，方便查看同一批提交，进行提交的回滚。

以上原则是我们在开发过程中总结得到的内容，这些原则有些是 **开发者已经熟悉的开发规范**，有些是我们分析了已有的问题提出来的 **适配解决方案**。它们也体现在了 [Starry](#)，[QuadOS](#) 等模块化内核上，在实践中不断完善，验证自身的实用性。我们认为这些内容有助于帮助更好地进行组件化内核的开发。

组件化内核的探索成果

在上述思想指导下，我们在组件化内核上进行了探索和实践，得到了实际的工作成果。

实践内核介绍

参赛选手郑友捷在 ArceOS 的基础上扩展实现了 Starry，同时支持宏内核和 Unikernel 两种架构，并且运行在实体开发板上，适配了 tokio, dora 等复杂应用。参赛选手朱懿在 seL4 内核的基础上用 Rust 进行重构重写，得到了 Rust 模块化内核 reL4。以下介绍 Starry 和 reL4 这两个内核的组成和细节。

Starry 介绍

ArceOS 本身是一个 Unikernel，但是可以通过组合其组件形成宏内核的架构。在这个思想的指导下，参赛选手郑友捷与其他同学在 2023 年开发了 StarryOS。Starry 通过复用 ArceOS 的模块，并且加入宏内核所需要的支持（如进程管理、多地址空间等），从而形成了基于 ArceOS 模块的宏内核。

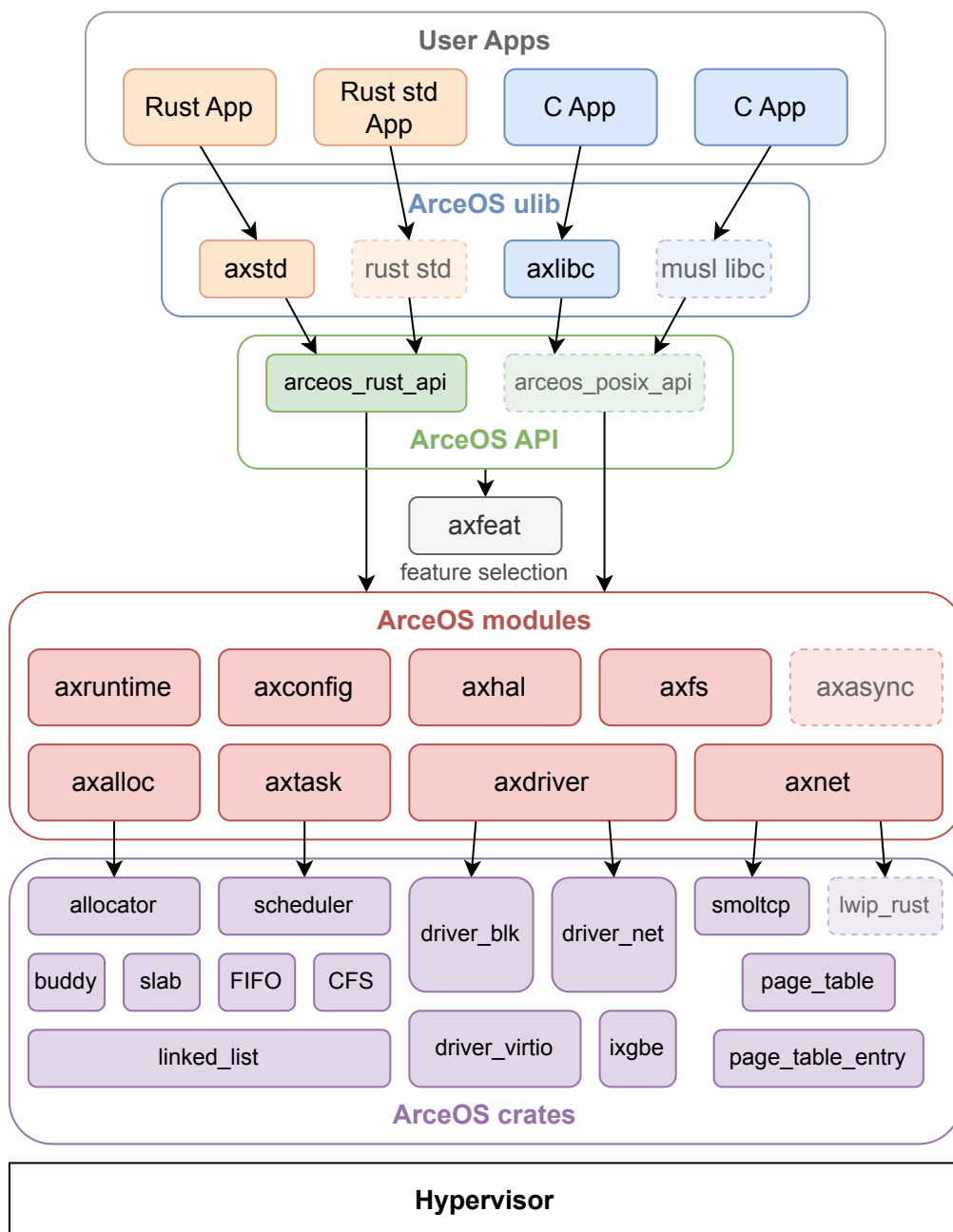
ArceOS 介绍

ArceOS 是以以组件复用化为宗旨，用 Rust 语言开发的一个 Unikernel 架构的模块化内核。它从 2022 年开始开发，目前已经拥有文件系统、网络等模块，支持 tokio, rust-std 等经典应用。ArceOS 作为模块化内核，其与功能相关的子模块（如页表、分配器等）可以较为方便地复用到其他内核中，并且也已经发布到 crates.io 上供其他人使用。

ArceOS 采用模块化组件化的设计思维，通过使用 **内核模块 + 组件化的OS框架** 来得到不同形态的 OS kernel。

- 提供了一套组件化的操作系统框架
- 提供各种内核组件的实现，各种内核组件可在没有 OS kernel 的情况下独立运行
 - 如 filesystem, network stack 等内核组件可以在裸机或用户态以库的形式运行或测试
 - 各种设备驱动等内核组件可以在裸机上运行
- 理想情况下可以通过选择组件构成 Unikernel / 宏内核 / 微内核
- Unikernel 特性如下
 - 只运行一个用户程序
 - 用户程序与内核链接为同一镜像
 - 不区分地址空间与特权级
 - 安全性由底层 hypervisor 保证
 - 适合用于嵌入式系统

ArceOS 的架构图如下：



ArceOS 本体目前支持的较为完善的是 Unikernel 结构，但由于其设计的优势，理论上我们可以通过简单的复用其组件组合出宏内核、微内核、hypervisor 等不同架构，从而将架构区别对内核开发带来的影响降到较低的程度。

Starry 组件介绍

作为基于 ArceOS 的宏内核，Starry 保持了 ArceOS 本身的功能，探索构建了一个**多构型的组件化内核**，并且**最大程度地保留了两种架构使用到的公共组件**，从而体现了组件的复用性。

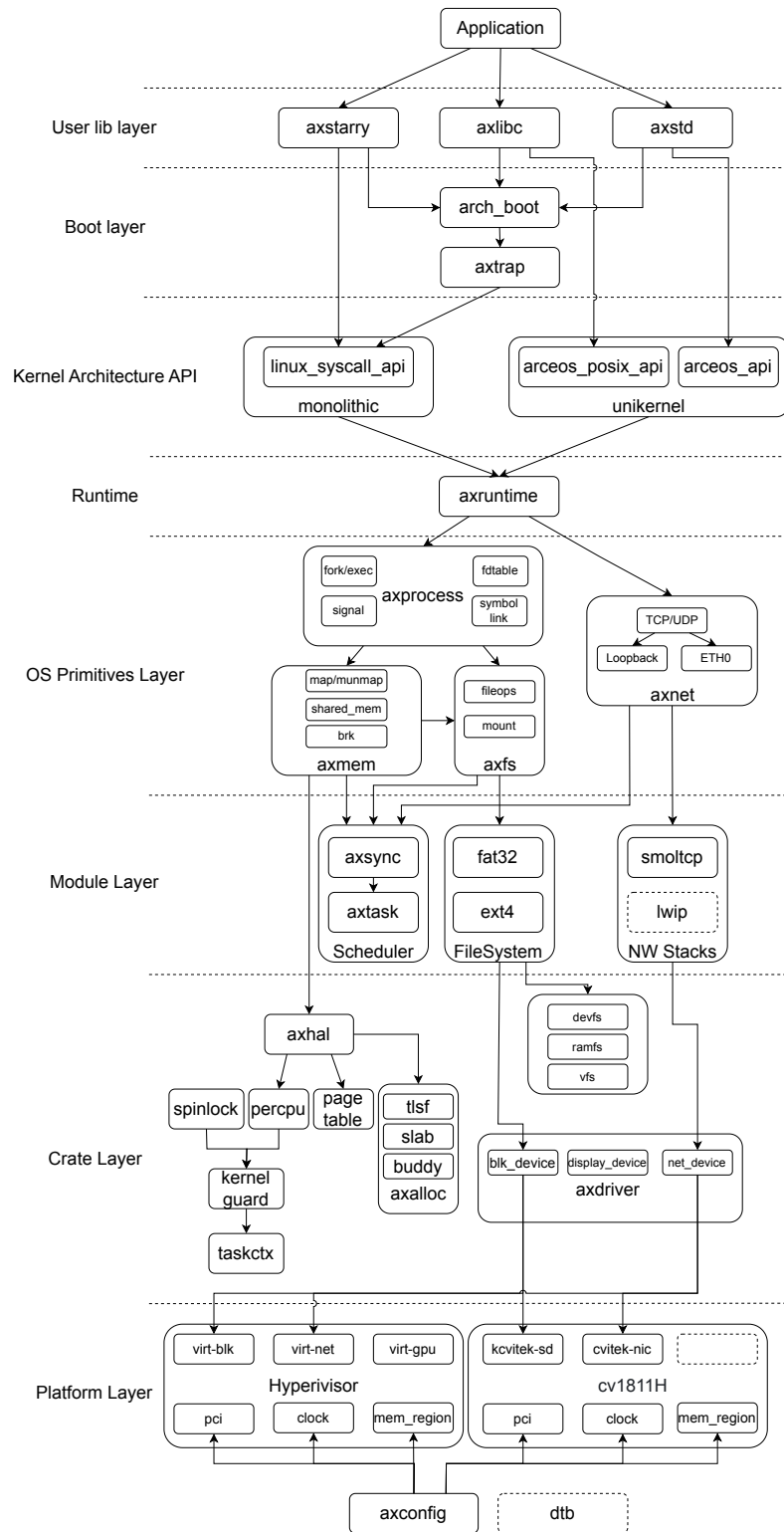
- Unikernel：提供一个 axlibc 库，底层将 syscall 改为 Unikernel 提供的接口，用户通过链接 axlibc 库可以直接将 Unikernel 和用户程序打包链接为一个可执行文件进行执行
- 宏内核：应用程序和传统的 Linux 用户库链接成一个可执行文件，加入到文件镜像中供宏内核读取执行

执行方式如下：

```
# 以 宏内核 启动
$ make A=apps/c/helloworld STRUCT=Monolithic run
# 以 Unikernel 启动
$ make A=apps/c/helloworld STRUCT=Monolithic run
```

通过控制编译条件，可以在 **不改动或者基本不改动源代码** 的情况下，将程序运行在不同架构的内核上，从而获得不同的运行效率和反馈。

Starry 的架构图如下图所示：



各个模块说明如下：

模块名称	URL	功能	复用性
Starry	https://github.com/Starry-OS/Starry	Starry 主仓，通过配置选择不同组件启动内核	共用
axstd	https://github.com/Starry-OS/axstd	Unikernel rust-std 支持	Unikernel
axlibc	https://github.com/Starry-OS/axlibc	Unikernel libc 支持	Unikernel
axstarry	https://github.com/Starry-OS/axstarry	宏内核启动初始化配置	宏内核
arch_boot	https://github.com/Starry-OS/arch_boot	支持内核 boot 启动	共用
axtrap	https://github.com/Starry-OS/axtrap	内核 trap 集中处理入口	共用
linux_syscall_api	https://github.com/Starry-OS/linux_syscall_api	宏内核 posix syscall 处理	宏内核
arceos_posix_api	https://github.com/Starry-OS/arceos_posix_api	Unikernel posix syscall 处理	Unikernel
arceos_api	https://github.com/Starry-OS/arceos_api	Unikernel 提供的通用接口	Unikernel
axruntime	https://github.com/Starry-OS/axruntime	内核运行时	共用
axprocess	https://github.com/Starry-OS/axprocess	进程管理	宏内核
axnet	https://github.com/Starry-OS/axnet	网络模块	共用
axmem	https://github.com/Starry-OS/axmem	地址空间管理	宏内核
axfs	https://github.com/Starry-OS/axfs	文件系统管理	共用
scheduler	https://github.com/Starry-OS/scheduler	调度模块	共用
axhal	https://github.com/Starry-OS/axhal	硬件抽象层，提供平台的硬件接口	共用
axdriver	https://github.com/Starry-OS/axdriver	驱动管理	共用

Starry 作为参赛作品参加了 2023 年全国大学生计算机系统能力大赛内核赛道，通过了内核赛道的所有测例，包括 musl-libc-test, lua, busybox, lmbench, iperf/netperf, UnixBench, libc-bench, cyclicttest 等，获得了线下决赛的第六名，最终获得全国二等奖的成绩。在比赛结束之后，Starry 依然在不断地维护、更新，并作为组件化内核的成果参加了本届大赛的功能赛道。

工作量描述

Starry 在开发维护的过程中，本队伍参赛选手 郑友捷 作为主要维护人，另外也有其他的实习生、工程师以协作者的形式向仓库提起 PR，协助完善 Starry 的各项功能。Starry 的所有模块存放在 <https://github.com/Starry-OS> 组织中。其中各个仓库的个人 commit（即郑友捷的提交）和协作 commit（其他协作者的 commit）细节如下：

仓库名称	URL	功能	个人 commit	协作 commit
Starry	https://github.com/Starry-OS/Starry/	Starry 主仓，配置组合各个内核组件形成完整内核	16	2
linux_syscall_api	https://github.com/Starry-OS/linux_syscall_api/	linux posix syscall 兼容实现层	46	19
axprocess	https://github.com/Starry-OS/axprocess/	进程管理	13	5
axnet	https://github.com/Starry-OS/axnet/	网络模块	4	2
axfs	https://github.com/Starry-OS/axfs/	文件系统模块	7	4
other		其他各种模块仓库，由于涉及数目较多不一列举	58	5
总计			124	37

上图统计的是 6-7 月份 StarryOS 的 commit 记录，其中 contributors 除去参赛选手郑友捷还有 5 位协作者。从上图可以看出，虽然在开发过程中其他协作者也协助完成了一些功能，但是 Starry 的主体开发维护仍然由参赛选手郑友捷负责。

目前支持情况

对于 2024 年 OS 比赛内核实现赛的 RISC-V 中测例支持情况如下：

- musl-libc-test, lua, busybox, lmbench, iperf/netperf, UnixBench, libc-bench, cyclicttest：均可正常运行
- LTP 测例：选择测试了常见的 syscall，如 execve, clone, close 等测例，共测试了 30 个测例，通过 24 个，通过率较高。

其他常见应用支持如下：

- [Tokio](#) : examples 已经在本机测试通过
- [Redis](#) : 局域网内测试 redis 完成
- [Dora](#) : examples 和 benchmark 测试通过, 可以通过 qemu 走局域网链接上机械臂并且进行操作

reL4 介绍

seL4 微内核简介

seL4 微内核 ([Home](#) | [seL4](#)) 是在 L4 微内核的基础上增加了安全性而开发的, 其具有以下的特点

- 轻量化

轻量化是微内核天然具有的一个特点, seL4 微内核最初版本的代码仅仅只有8700行。轻量化带来的优势一方面更易于读懂, 便于开发者学习, 另一方面也减少了运行时的开销。

- 安全性

seL4 微内核的安全性是通过形式化验证的方式证明的。其首先写出全部微内核对象的抽象规范, 将其转换为可执行规范, 随后将可执行规范转为 C 代码, C 代码需要转为二进制代码, 在这整个过程中, 都seL4都保证了其正确性, 从而保证了最终生成的二进制可执行文件的无 bug 特性。

同时, seL4 微内核实现了一个称为能力空间 (CSpace) 的 capability-based 的模型, 基于这个模型, 实现了对每个内核对象的细粒度的控制, 同时通过能力派生, 以及能力检查的控制, 保证每个内核对象操作的安全性。

- 高性能

传统的微内核存在 IPC 性能损失过大, 而微内核架构意味着多个服务模块之间的调用都通过 IPC 实现, 过长的 IPC 调用链加之单个 IPC 的较长通信时间, 导致微内核存在整体性能不高的问题。

而 L4 系列的微内核从提升性能本身出发, 提出了一系列更为贴合微内核哲学的设计思想, 只保留了地址空间, 进程, 进程间通信等基本内容在内核中, 并实现了基于寄存器的进程间通信。从而大幅度减少了内核态尤其是 IPC 的开销。

seL4 微内核作为 L4 系列微内核的继承者, 同样继承了 L4 系列微内核的高性能的特点, 同时, 针对于现代 CPU 的发展, 也利用了更多的现代 CPU 的优势。从而实现了更高的 IPC 性能。

- 高度可定制化

除了微内核最为核心的功能之外, 其他的系统资源都交给了可定制化的 root server 进程来分配。而内核也仅仅只负责最为基本的功能, 因此, 内核的其他模块都可以按照用户的需求进行更改并在用户态运行。

使用 Rust 对 seL4 微内核进行改造

基于以上的介绍, 综合考虑到改造的安全性和性能, 我们决定将 seL4 的 aarch64 和 riscv64 部分的代码改造成 Rust 版本。并在此基础上对 seL4 进行模块化的划分。reL4 的仓库地址在:

https://github.com/reL4team/reL4_kernel/。本节介绍对 seL4 进行 Rust 改造的内容。

1. 嵌入Rust代码

在改造过程中，我们的做法是，将一部分的 Rust 代码嵌入到 seL4 代码中，进行逐步的替换，从而避免完全性的重构，在这个过程中一步一步验证我们嵌入的代码的正确性，最终替换掉整个 seL4 的内核代码。

为此，首先我们修改了部分 seL4 的 CMake 构建脚本。

在 `cmakelists.txt` 文件中，我们增添了如下代码

```
if(KernelArchRiscV)
    message(STATUS "ARCH ${KernelArchRiscV}")
    link_directories(../reL4_kernel/target/riscv64imac-unknown-none-elf/release)
endif()
if(KernelArchARM)
    message(STATUS "ARCH ${KernelArchARM}")
    link_directories(../reL4_kernel/target/aarch64-unknown-none-softwarefloat/release)
endif()

target_link_libraries(kernel.elf PRIVATE kernel_Config kernel_autoconf rustlib)
```

这两段代码的作用，一个是增加这两个路径，另一方面，我们的 Rust 版本的代码本身在项目的 `Cargo.toml` 文件中就声明了

```
[lib]
name = "rustlib"
path = "src/lib.rs"
crate-type = ["staticlib"]
```

换言之，这会将自己生成成为一个名为 rustlib 的静态链接库。

随后，我们逐步的使用 Rust 代码替换掉C中的诸多函数即可。

2. 使用 ffi

我们使用ffi的接口来进行C和Rust的兼容。FFI 是一种编程接口，可以让 C 代码和 Rust 代码能够相互调用。对于 Rust 一端调用 C 代码，只需要简单的使用

```
extern 'C' {
    fn function(args:type)
}
```

等方式即可。

而在 Rust 端提供 C 端所需要的代码，则需要在 Rust 端定义的函数前面加上

```
#[no_mangle]
```

用于禁止编译器对函数的重命名

最后，为了便于后期对这些侵入式代码的去除，我们统一将所有这些 ffi 函数都放在了每个模块中的 `ffi.rs` 文件中，并在此约定好，所有使用 ffi 的函数均进行统一的归纳

C和Rust的不同语法导致的代码调整

C 和 Rust 具有不同的语言风格，我们不能简单的使用 `unsafe` 块对 seL4 的代码转译成 Rust。以下是一些我们遇到的问题和相应的处理。

1. 使用面向对象的设计

C 语言本身是面向过程的一种方法，虽然可以在结构体中加入函数指针等方法来实现面向对象的设计，但其本身的语言设计并不存在面向对象代码模式，更缺少 Rust 引入生命周期、所有权机制而增加的语法。Rust 则不同，为了安全性等原因，其引入了一堆与 C 不完全一致的代码规则，这些代码规则不仅仅是为了安全性的检查，也不仅仅是可读性的区别，还会一定程度上影响编译器的行为。

以 `capability` 中的最基本单元 `cte_t` 为例：

在 C 代码中。定义如下

```
/* Capability table entry (CTE) */
struct cte {
    cap_t cap;
    mdb_node_t cteMDBNode;
};
typedef struct cte cte_t;
```

然而，使用 Rust 面向对象的方法去重构相关的代码，则将 `cte_t` 相关的方法均做一定的封装。

```
pub struct cte_t {
    pub cap: cap_t,
    pub cteMDBNode: mdb_node_t,
}

impl cte_t {
    /*functions*/
}
```

而在调用相关函数的时候，则不再填入函数的第一个参数，而是使用该对象的某个方法。

以上只是一个简单的例子，对于其他的 C 中的函数，我们也做了归类整理，尽量将他们使用面向对象的方法重写。

2. ffi 符号兼容

在改造成和 C 代码兼容的过程中，存在以下问题，部分 C 代码中的函数已经按照 Rust 版本的设计，已经变成了某个对象的某个方法（并非某个 `#[no_mangle]` 的函数），而在 C 代码中，又试图使用该符号，这就会导致报错缺少符号定义，这本来是由于没有完全实现 Rust 代码改造，而又必须使得 C 和 Rust 代码兼容导致的问题。

对于这个问题，我们额外补足了这些在链接器中会报符号缺失错误的函数。同时，为了未来的扩展，我们在所有这些地方均增加了注释。

3. Rust 未定义行为和 `unsafe` 的注意事项

在 Rust 重构 C 代码的过程中，由于 FFI 的存在以及对于操作系统和系统底层细节的控制要求，引入 `unsafe` 代码是必然的行为，但是在引入 `unsafe` 代码的时候，对于其真实的可能行为必须加以了解，这样才能避免由于 `unsafe` 的引入产生 bug。

以下是一个例子：

在使用 Rust 重构 C 代码的过程中，使用裸指针 `*mut T` 和 `*const T` 是非常常见的行为。但是，该行为在 Rust 中是一个未定义的行为。可以参考

[Warn that `*const T as *mut T` is Undefined Behavior · Issue #66136 · rust-lang/rust \(github.com\)](#)

它的某种可能的导致不安全行为的方式（及其原因解释）如下：

这种操作通常来自于对于某个不可变引用，试图为该引用写入某些值。因此需要将其转为 `const` 的裸指针，再转为 `mut` 的裸指针。

而如果不小心的在某个较短的生命周期的函数中，试图写入这个值，由于他被认为是不可变引用，所以编译器会在某种时候把它优化到栈上，最终把短生命周期中的栈上的内容写入了，随着退栈，最终导致了出错。

在 rust 的规范中，唯一允许这种情况的是使用 `UnsafeCell`，使用其来封装某个类型来达成内部可变量性。（因为对这个类型，做了一定的编译器开窗）

这个问题的模式，一方面要满足需要将某种情况下的裸指针转为 `const` 再转为 `mut` 类型，同时还需要满足他可能是一个短的生命周期——以至于编译器会在某些优化的情况下将其放置在栈上。

以上只是其中一个 Rust 未定义行为以及 `unsafe` 代码中不小心的操作所引入的问题，同时使用 C 和 Rust 代码，意味着不得不使用大量的 `unsafe` 代码，seL4 中又使用了大量的 64 位的 `usize` 类型进行转换，这既可以表示为某种指针，也可以表示为页表中的某一个 `entry` 等等其他类型。在这个转换过程中必须考虑到 Rust 代码的语义翻译不出错，也要考虑到不能触发 Rust 中的不安全行为。

aarch64 和 riscv64 的多架构支持

根据我们改造的目标，我们需要同时支持满足 aarch64 和 riscv64，在原有的 seL4 代码中，也将跟架构相关的代码放置在了一起。

由于我们划分出来的模块没有 `arch` 模块，而每个模块中都有大量跟架构相关的代码，因此我们选择在每个模块的文件夹下面增加了一个 `arch` 的模块。在这里面集中放置跟 `arch` 相关的代码。

一个常见的目录树如下

```
arch
├── aarch64
│   └── some files under aarch64
├── mod.rs
├── riscv64
│   └── some files under riscv64
```

而在 `mod.rs` 中使用如下代码进行包装

```
#[cfg(target_arch = "aarch64")]
mod aarch64;

#[cfg(target_arch = "aarch64")]
pub use aarch64::*;

#[cfg(target_arch = "riscv64")]
mod riscv64;

#[cfg(target_arch = "riscv64")]
pub use riscv64::*;
```

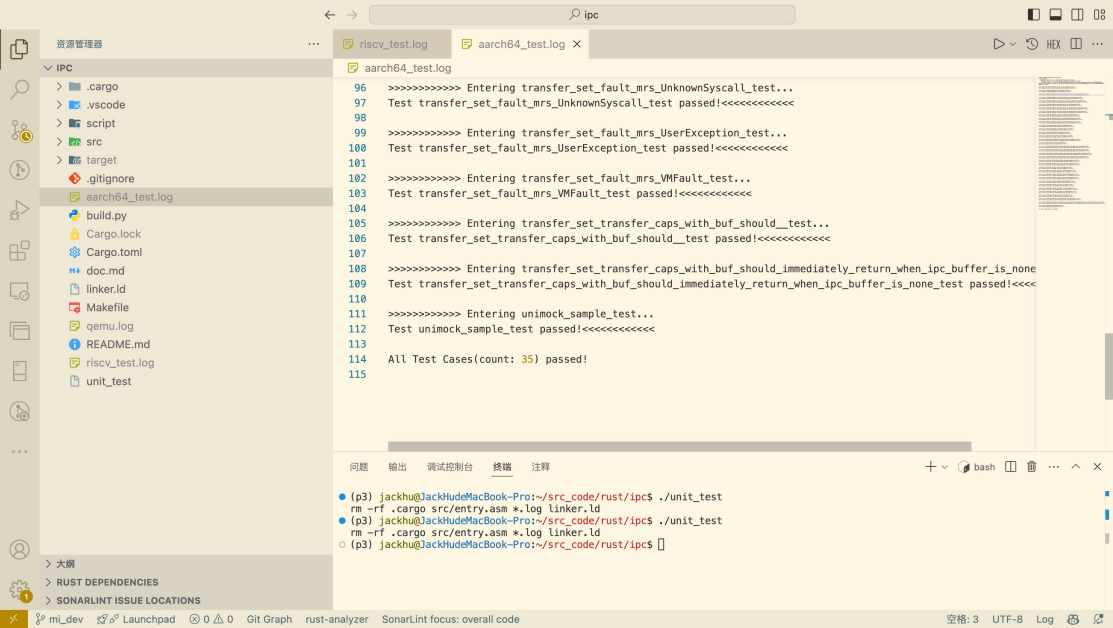
向上提供在某种架构下的代码，供其他模块使用放在某个架构下的代码。

同时，尽管这么做了，但是并不能完全避免在其他部分的代码中包含arch相关的代码，因此，在其他地方我们也会选择使用

```
#[cfg(target_arch = "aarch64")]
#[cfg(target_arch = "riscv64")]
```

来定义专属于某个架构的代码。

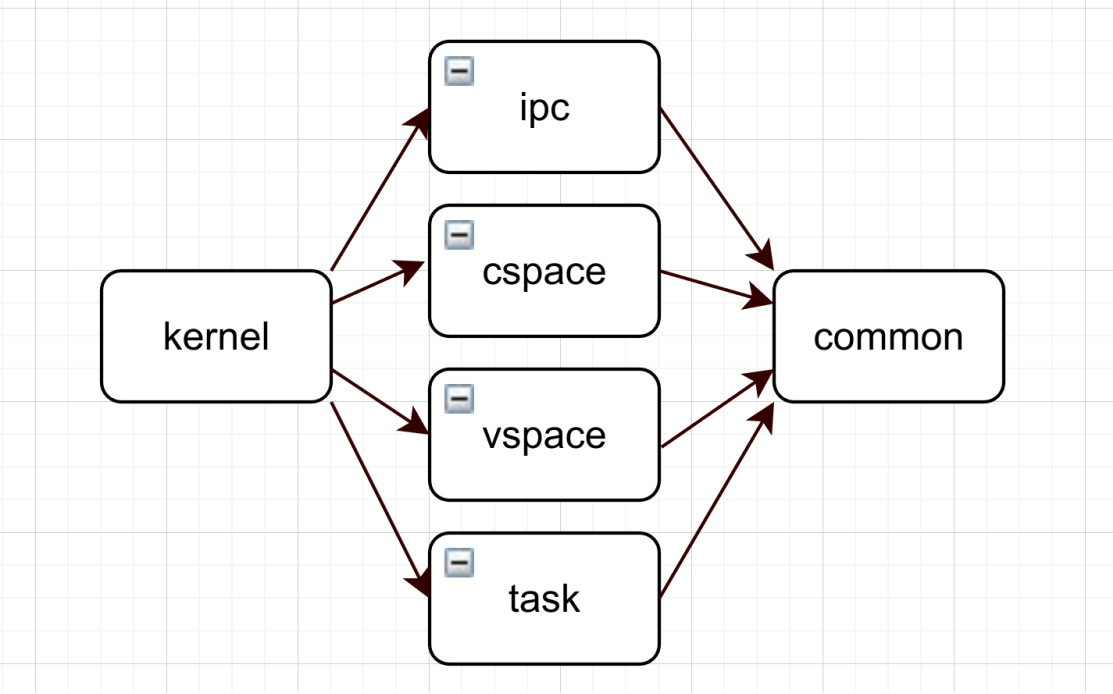
相关的指令架构的单元测试截图如下（以 aarch64 为例）：



reL4 组件介绍

在上述基础上对微内核进行模块化改造，以及相关的模块化设计的基础之后，开始对reL4微内核进行模块的划分

根据原本seL4的特点，我们划分并实现了如下的几个模块：common，task，cspace，vspace，ipc，kernel。模块间的依赖关系如下



组件文档详见 [sel4_vspace - Rust \(rel4team.github.io\)](https://rel4team.github.io)，该文档下也可以找到其他模块的文档介绍。各个组件介绍如下：

1. common 模块：

这部分的代码主要包括所有模块都依赖的代码，主要还是各种宏，以及各个模块都需要使用到的内容，比如定义 seL4 内核中的通用错误码和对应的描述。

2. cspace 模块

这部分主要用于描述 reL4 的capability的部分。相比之下较为简单。它包括compatibility、deps、interface三个模块，其中 compatibility 和 deps 子模块是通过 FFI 实现的，实现了 reL4 的 capability 功能，之后会逐渐改造为纯 Rust 实现。而 interface 模块包含了暴露给外部的接口。

3. vspace 模块

这部分主要用于描述reL4的地址空间控制相关的内容，是一个比较常见的地址空间管理模块。

4. ipc 模块

这部分主要包括reL4的ipc部分。主要包括endpoint，notification，transfer三种机制

- endpoint：这是一种基本的通信机制，用于线程（或进程）之间的同步和异步消息传递。endpoint允许线程以发送（Send）、接收（Recv）或回复（Reply）的形式进行通信。这些操作可以通过IPC（Inter-Process Communication）机制实现，允许线程之间安全地交换信息。

在reL4_kernel中，endpoint被表示为一个结构体（通过plus_define_bitfield!宏定义），其中包含了队列头部、队列尾部和状态等字段。这些字段用于管理通过endpoint发送和接收消息的tcb队列。EPState枚举定义了endpoint可能的状态，包括：

- Idle：表示endpoint当前没有进行任何消息传递操作。
- Send：表示有线程正在尝试通过endpoint发送消息。
- Recv：表示有线程正在尝试从endpoint接收消息。

- notification：这是一种用于线程间通信（Inter-Process Communication, IPC）和同步的机制。notification对象可以被视为一种轻量级的信号量，它允许一个线程向一个或多个等待的线程发送信号，从而通知它们某个事件的发生或者某种条件已经满足。

具体来说，notification对象可以处于以下几种状态之一：

- Idle（空闲）：notification对象当前没有被任何线程等待或激活。
- Waiting（等待）：至少有一个线程正在等待notification对象的信号。
- Active（激活）：notification对象已经被激活，但尚未被任何等待线程接收。notification对象通过signal操作被激活，当线程执行wait操作时，如果notification已经被激活，则线程会继续执行；如果notification未被激活，则线程会进入等待状态，直到notification被另一个线程激活。

- transfer：这是在不同线程或进程之间传输信息、能力（capabilities）或者处理fault（faults）的过程。在给定的代码片段中，transfer是一个trait，定义了一系列与信息传输、能力传递、fault处理和信号完成相关的函数。这些函数允许线程（通过tcb_t结构体表示）之间进行通信和交互。

5. task 模块：这部分主要包括了reL4的进程管理方面的内容，实现了微内核架构下的任务资源定义和调度的功能。

6. kernel 模块：除了上述模块中所描述的代码均放置在此，细致的分类包括系统boot的代码，中断代码，内核对象，syscall的解析等。由于其不是一个模块，而是内核，因此不在此赘述。

实践内核突出点介绍

我们在对多构型组件化内核进行开发实践的过程中，不仅仅是开发了一个 demo 内核，还做了许多工作去尽力地完善内核本体以及它们所涉及的组件，逐步打磨我们的组件，更加值得被其他内核复用。

创新性：内核多构型支持

作为基于 ArceOS 的内核，Starry 不仅 **保留了 ArceOS 本身的 Unikernel 架构**，还通过添加额外的模块，让 Starry **可以以宏内核的架构启动**。

在这个基础上，Starry 可以通过条件编译等手段选择 **以宏内核还是 Unikernel 架构**的形式启动内核，并且 **最大程度地保留了两种架构使用到的公共组件**，并且在此基础上添加了所需的模块扩展。

宏内核

为了支持宏内核，需要在原先 ArceOS 的模块上添加如下模块：

- axmem：引入地址空间
- axprocess：引入进程概念，支持多进程多线程运行，添加了很多与Linux系统调用相关的内容
- linux_syscall_api：Linux系统调用用户库，包装了作为Linux兼容层的众多对外接口

这些模块虽然不适用于 Unikernel，但是适用于其他的宏内核库，尤其是 linux_syscall_api 模块，包装了对 posix syscall 语义的检查和类型的定义，这个对其他的宏内核也会有参考意义。

Starry 对原有 ArceOS 的模块利用可以参考上面的架构图，**基本会复用 Unikernel 涉及的所有功能模块**，这也体现了 Unikernel 本身模块化设计的优势所在，为宏内核的许多模块开发带来了可以复用的功能。

微内核

对于组件化微内核的实践，主要着眼于现有的 seL4 微内核入手，以 seL4 微内核为基础，开发了 rust 版本的 reL4。并对 reL4 的进行了组件化的实践。

对于微内核的组件，包括了4个部分

- cspace：引入 capability 机制，对每个内核对象都有相应的权限进行控制
- vspace：引入了进程地址空间相关的方法。
- ipc：引入了对于微内核进程间通信相关的方法，包括了使用endpoint作为进程间通信的实体，notification 机制进行快速轻量级通信等。
- common：对于和架构相关的一些宏定义，以及多个模块所公用的宏定义和方法进行抽象和封装

这些模块在已有的操作系统的基础上，实现了微内核的一些特性，包括通过 capability 机制对内核对象的保护，通过优化过的 ipc 方式进行进程间通信等。

多架构支持的意义

讨论一个内核兼容多种内核架构的意义时需要先明确不同内核架构彼此的适用环境。以宏内核和 Unikernel 为例：宏内核作为一个尽可能完善的内核整体，对应用功能、安全性检查等方面都做了许多工作，适用于通用操作系统等；而 Unikernel 本身要求尽可能地精简，以性能为重要衡量标准，适用于各种嵌入式场景。

<https://github.com/Azure-stars/Starry/blob/feat/module/doc/OS-Train-Repo/Week4.md> 对 Starry 和 ArceOS 基于同样的功能调用做了性能上的对比测试。结果表明：基于**同样的底层模块**，宏内核相比于 Unikernel 架构，会花费更多的时间在特权级切换、trap 上下文的存储和恢复、参数安全性检查上，因此性能会有一定下降。但由于宏内核支持的多进程和信号捕获机制，他能够在任务发生异常的时候捕获并进行对应的处理，而不至于直接使得整个内核终止。

这一点结合上面的自选架构启动，可以帮助应用在开发过程中较为方便地进行调试，即只需要将应用作为子进程进行启动，而父进程**在宏内核下可以较为方便地捕获子进程的错误信息**。当子进程开发完毕，只需要简单的改变链接库，同时脱离父进程单独启动，就可以在**Unikernel 架构下以较高的性能运行**。我们可以

另外，由于应用场景的不同，同一个应用可能在不同场景下适合运行在不同架构上。下文即将介绍的 **Dora** 就是一个例子。当场景适合嵌入式内核的时候，就可以 Unikernel 的形式运行程序，而当场景适合于多进程多应用的时候，就适合以宏内核形式运行。上述两种场景在 Dora 的例子中甚至是分布式应用的两个不同节点的表现，也就是说在一个应用环境下便可能使用同一套组件以不同架构同时运行，从而在复用组件的同时尽可能地贴近使用场景，提高运行效率。

总结上述内容，多架构支持主要意义体现在两个方面：

1. 便于开发：程序开发可以先在宏内核上进行，通过完善的错误捕获处理机制解决问题之后，再运行在 Unikernel 上，获得足够的运行性能。
2. 根据场景选择运行架构，以开发同一套组件的成本获取不同架构带来的最高性能。

泛用性：多指令架构支持

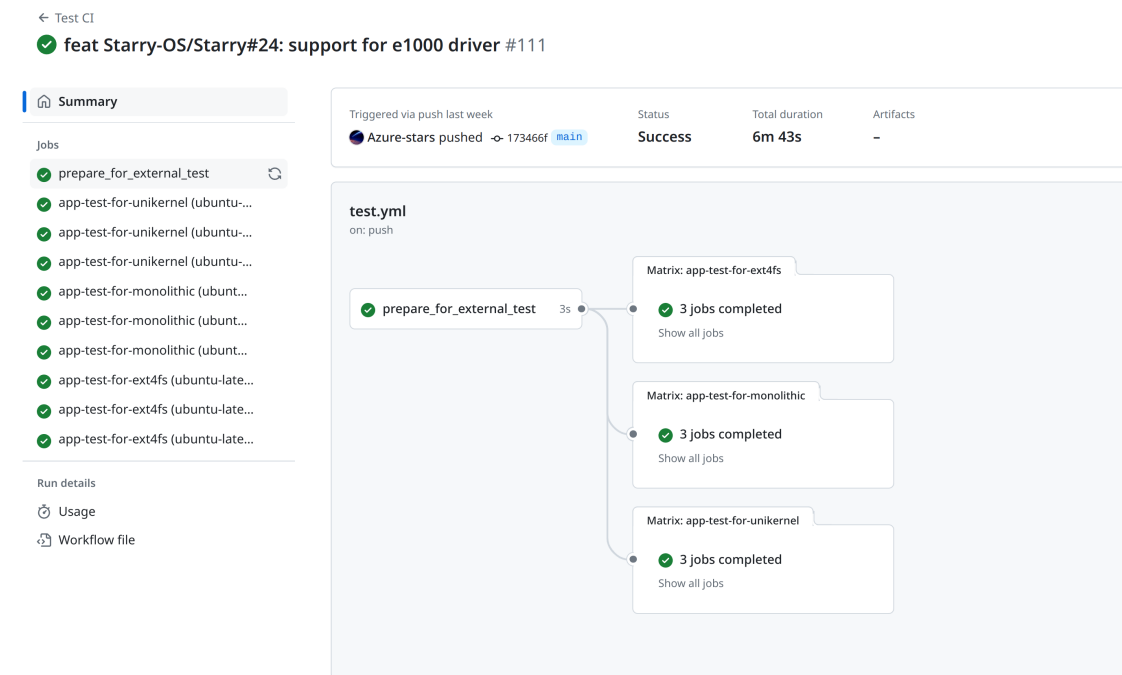
组件化的内核设计将与硬件相关的设计集中到了硬件抽象层上。这一层在 ArceOS 表现为 **axhal** 层，而在 Starry 中由于**单向依赖的设计**拆分为三层：**axhal**，**axtrap** 与 **arch_boot**。三个模块各自的功能如下：

- axhal：提供平台相关的接口函数，如时间、内存区域等；提供与指令架构相关的函数，如开关中断、寄存器交换保存
- axtrap：提供 trap 入口和处理函数，包括 syscall ‘interrupt ‘exception 等
- arch_boot：提供内核启动相关汇编，支持从 multiboot ‘SBI 等启动内核

其中 axhal 和 arch_boot 对于宏内核和 Unikernel 是通用的，axtrap 中需要添加宏内核相关的异常处理，如 syscall、缺页异常等。

在模块化的设计思想下，原生的 ArceOS 支持 **x86_64**，**riscv64**，**aarch64** 三种指令架构的启动。Starry 在其基础上添加了宏内核相关的启动支持（如 x86_64 对 rflags 等寄存器的设置）等，使其可以通过复用原来的硬件抽象层，完成对内核态、用户态等多特权级的支持，也同时支持 x86_64，riscv64，aarch64 指令架构的启动。

<https://github.com/Starry-OS/Starry/actions/runs/10132474885/job/28016482503> 等三个 job 也体现了对不同指令架构的测试内容。



Starry 不仅通过了 2023 年的内核赛道的所有测例的三个指令架构的版本，还将他们存放在 <https://github.com/Starry-OS/testcases> 供 CICD 测试使用。

上述的支持均在 QEMU 上运行测试。另外 Starry 也支持在实际开发板上运行。目前 Starry 已经**成功**在 **riscv 星光二代开发板**与 **x86_64 工控机**上启动运行，并且可以加载指定测例。支持 x86_64 工控机运行的代码存放在 https://github.com/Starry-OS/Starry/tree/support_intel_i219 上。

在 reL4 方面，他们也开始支持多指令架构，目前已经成功适配了 riscv64 和 aarch64 指令架构。

实用性：复杂 APP 支持

Dora

面向数据流的机器人应用程序 Dora-rs 是一个中间件应用。他们通过使用共享内存和 Apache Arrow 实现内存零拷贝，降低传统的机器人框架中间件如 ROS2 的通信开销，在性能上表现出了较大的优势。

以下为 Dora 的相关资料：

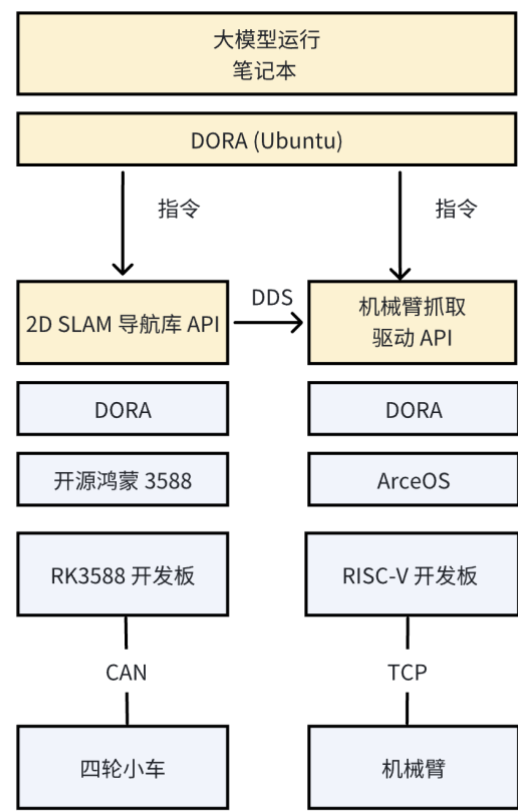
- 官网：<https://dora-rs.ai/>
- 代码仓库：<https://github.com/dora-rs/dora>

Dora 分为三部分，分别是 coordinator，daemon，cli

- coordinator 启动之后，自动监听 53290 端口，等待 daemon 连入
- daemon 启动之后，向 53290 端口发送注册信息，告知自己所在的 IP 和端口，由 coordinator 确认注册成功
- coordinator 还会监听 6012 端口，等待接受 CLI 信息
- 用户通过 CLI 向 coordinator 发送一个 start 数据流，其中包括了需要启动的多个 node 的信息和他们所在的可执行文件路径
- coordinator 接收到数据流信息之后，搜索 node 运行在哪个 daemon 上，将需要启动的 node 信息和所在路径传递给对应的 daemon，称为 spawn dataflow
- daemon 接收到 spawn 数据之后，找到 node 所需要的可执行文件，并且启动，若启动成功则向 coordinator 回传成功信息和 node 数据
- coordinator 接收到 All finished 信息之后向 CLI 传递数据，完成一次交互

通过这个运行流程可以看出 Dora 主要起了一个中间件的作用。通过 Dora 可以完成信息的交流和指令的执行。当上层用户通过 CLI 下达指令之后，便可以通过 Dora 传递到指定的 node 启动对应的程序进行运行，并且接受程序的反馈传给用户。

我们的工作是基于 Dora 中间件实现人工智能小车的指令接收和下达。具体功能如下：



在以上内容中，ArceOS/Starry 承担的工作主要是对机械臂的操作，包括如下功能：

- 1. 接收来自小车的命令
- 2. 通过 TCP 链接机械臂，以固定指令格式下达操作机械臂的指令

以上的通信功能均是由 Dora 完成通信和协作，因此对于 ArceOS/Starry 的支持要求分为两部分：

- 1. Dora 本体支持，包括不同节点通信
- 2. 机械臂 SDK 支持，包括通信和报文格式转化

ArceOS 工作

Dora 本身设计开发时是针对通用操作系统，即宏内核架构设计的。因此本体启动时会同时启动 coordinator、daemon 和多个 node，每一个都作为单独的进程拥有各自的地址空间。但对于 Unikernel 来说不支持多进程和多地址空间，因此不能直接运行 Dora 在 Unikernel 上，需要对 Dora 的源码做出调整适配。

在我们的应用场景下，单个开发板上仅会运行一个机械臂 SDK node，因此在单个开发板上不需要完成 Dora 的全部功能支持，只要支持单个 Node 即可。因此便可以采用嵌入式开发的思想，将 Dora 的 Node 单独抽象出来，并且运行在 ArceOS Unikernel 上。

以下关于 ArceOS on Dora 工作并非由参赛选手完成，而是由 ArceOS 的维护者贾越凯学长、胡柯洋学长完成。我们将他们的工作在这里简单阐释，用于和 Starry 的工作进行对比呼应，从而突出组件化的优势。

- Port nodes in [dora-rs/rs-latency](#) to dynamic Node：将 Dora 的 node 节点修改为 dynamic Node 属性，从而支持在运行了 daemon 的情况下可以动态接入新的节点
- Dora Nodes (sink or node or both) run on ArceOS：在 ArceOS 上运行单个 node。Unikernel 不支持多进程，但可以支持单进程多线程的任务。对于我们的应用场景，一个机械臂 SDK 就是一个 node，ArceOS 只需要支持单个 Node 节点即可。
- ArceOS run on QEMU/KVM：即将 ArceOS 运行在 QEMU 或者 KVM 虚拟机上，实现裸机 or Guest OS 启动。
- Dora Daemon & Coordinator run on host Linux：对于我们的应用场景，让单个 Node 运行在 ArceOS 上，而 Daemon/coordinator 等任务作为用户下达指令的途径则运行在其所在的通用操作系统上，如 Linux 等

通过修改 Dora 的源码，分离出单个 Node 之后，两位学长成功让单个 Node 运行在了 ArceOS Unikernel 上。

```
SeaBIOS (version rel-1.16.3-0-ga6ed6b701f0a-prebuilt.qemu.org)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+06FD0AA0+06F30AA0 CA00

Booting from ROM..
Initialize IDT & GDT...

      d8888      .d88888b. .d8888b.
      d88888      d88P" "Y88b d88P Y88b
      d88P888      888      888 Y88b.
      d88P 888 888d888 .d8888b .d88b. 888      888 "Y888b.
      d88P 888 888P" d88P" d8P Y8b 888      888 "Y88b.
      d88P 888 888      888 88888888 888      888 "888
      d8888888888 888 Y88b. Y8b. Y88b. .d88P Y88b d88P
      d88P      888 888 "Y8888P "Y8888 "Y88888P" "Y8888P"

arch = x86_64
platform = x86_64-qemu-q35
target = x86_64-unknown-arceos
smp = 1
build_mode = release
log_level = off

Dora sink-dynamic on ArceOS booted at time 2024-07-09 12:15:36.215625224 +00:00
DaemonChannel: try to connect to 10.0.2.2:53291
DaemonChannel: try to connect to 10.0.2.2:33057
DaemonChannel: try to connect to 10.0.2.2:33057
DaemonChannel: try to connect to 10.0.2.2:33057
DaemonChannel: try to connect to 10.0.2.2:33057
Input `latency` was closed
Input `throughput` was closed
finished
→ arceos-org git:(dora-wip) █
```

通过 benchmark 测试可以发现 Unikernel node 的性能较为优秀。这也体现了嵌入式系统在合适场景下的优秀之处。

Starry 工作

Dora 在 Starry 上的运行符合其一开始针对通用式操作系统的设计思路，因此基本不需要对源码进行修改，但对 Starry 的功能完善性提出了较高的要求。Dora 本身使用 tokio 库进行编程，并且使用共享内存降低传输开销，对宏内核的网络通信、内存管理和进程管理均有要求。

Starry 通过复用 ArceOS 的组件，拥有了基本的网络驱动、文件系统等模块的支持。为了支持 Dora 还做了如下工作：

1. syscall 兼容层支持

Dora 本身用到了更多复杂的 syscall，如 socketpair、sigaltstack 等，并且涉及更多的 syscall flags，需要对 syscall 的语义做更为仔细的检查。

Dora benchmark 本身使用到的 syscall 共有 60 个，具体如下：

futex	sched_getaffinity	brk	getrandom	clone
madvise	write	sched_yield	clock_gettime	epoll_create1
exit	arch_prctl	fcntl	rt_sigaction	unlink
sendto	wait4	poll	setsockopt	socketpair
epoll_pwait	ftruncate	fstat	read	socket
chdir	pipe2	fork	listen	rt_sigreturn
getsockopt	bind	nanosleep	mkdir	set_tid_address
recvfrom	ioctl	open	getsockname	pidfd_open
mprotect	close	getpeername	eventfd2	stat
gettid	connect	epoll_ctl	execve	rt_sigprocmask
sigaltstack	munmap	fsync	readlink	prctl
mmap	accept4	getcwd	exit_group	dup2

具体的 syscall flags 内容详见：<https://github.com/Starry-OS/Dora-analysis/tree/main/dora-log>

我们通过使用 [LTP 测例集](#) 来辅助进行 syscall 语义的检查。目前已经通过了数十个测例的 LTP 测例集，并且完成了语义的检查。这个 syscall 兼容层的语义检查可以在将来抽象为一个单独的模块，供给不同宏内核以复用。

2. 网桥搭建

Starry 使用的是 smoltcp 协议栈，在原先的 Unikernel 架构下仅使用以太网作为 iface 进行网络通信，未考虑本地回环情况。在 2023 年参加内核赛道比赛时，为了通过网络相关的测例，需要启动本地回环 LOOPBACK iface。当时的实现情况是将以太网和 LOOPBACK iface 通过编译选项进行二选一启动，也通过了对应的测例。但是在 Dora 支持时，我们需要**同时满足 LOOPBACK 通信和以太网通信**。为了实现这一点，需要以太网设备和本地回环设备可以互相通信，即搭建一个网桥。

目前 Starry 使用了一个简易的网桥实现，能够满足 Dora 的需求，但是在网络负载提高时会导致丢包问题出现。接下来 Starry 会参考 lwip 协议栈对网桥的实现，进一步修改 smoltcp，实现更加完善的网桥。

3. 网卡驱动支持

项目的目标是在实际开发板上运行 Starry-Dora，并且能够通过网络与机械臂进行通信。因此 Starry 需要支持运行在开发板上，并利用开发板的网卡和外部进行通信。我们选用了一台 x86_64 的工控机，网卡设备为 e1000。

目前 e1000 驱动移植到 Starry 的测试已经在 QEMU 上完成，并且也已经在工控机上完成了设备感知和识别，下一步会将其与 Dora 结合起来，从而真正实现开发板上的 Dora 运行。

4. 其他 Unix 相关支持

为了实现 Dora 的功能，还需要用到其他的支持，如 `dev/shm` 共享内存支持等。这部分工作可以通过研究 Linux 相关语义，尝试复用已有组件（如 tmpfs），从而尽可能地减少工作量。这也体现了模块化操作系统的优势。

多架构支持对 Dora 意义

上面提到的 Dora 场景下，对 Dora 的使用分为了两类：

1. 需要多进程多任务：如用户端运行大模型，通过 Dora 下达指令、接收反馈传达给大模型，小车端通过 Dora 接收模型 Dora 的指令并下达指令给机械臂端的 Dora
2. 单进程单任务：Dora 接收命令并且通过 TCP 形式发送信息给机械臂

对于多进程多任务，需要**使用宏内核架构**来运行包括守护进程、协作任务和单个任务节点等多个进程任务。而对于单进程单任务端，只需要运行一个 Dora 节点，并且通过动态注册接入机制接入到宏内核下的 Dora 程序中，**这个任务就适合使用 Unikernel 架构来完成**。通过这样的优化方法，我们在**同一个业务场景下，用同一套组件在不同节点运行了不同架构的内核**，相比于原先完全使用宏内核而言，加快了运行效率，提高了性能。

Tokio

在开发 Dora 的过程中已经对 Tokio 做了基本的支持，但是后续运行 Tokio 的一些 examples 时发现仍然需要额外支持，具体需要做如下支持：

1. 终端输入输出缓冲区支持：Starry 使用 busybox 作为终端，其输入输出缓冲区默认长度仅为 1。在运行 Tokio 等测例的时候需要发送一段完整的报文给后端时，仅能将报文拆成多个字符逐次发送。这种情况下可能会导致报文格式识别错误，从而导致测例无法通过。因此我们需要规范终端的输入输出，按照标准终端的格式做好缓冲区、回显以及字符转义。
2. 网桥的完善：之前支持 Dora 的网桥模式并不是规范的实现，在实际测试的时候会有丢包的情况出现。因此需要参考如 lwip 等协议栈的网桥实现，将以太网设备和本地回环设备真正沟通起来。

目前 Starry 已经成功运行了 Tokio examples 的绝大多数测例，代表着 Starry 对 tokio 已经有了初步的支持。

ZLMediaKit 支持

ZLMediaKit 是一个基于 C++11 的高性能运营级流媒体服务框架，其官网首页为 <https://docs.zlmediakit.com/zh/>。其编译需要 x86_64 指令架构的 glibc 支持。对 ZLMediaKit 的支持与 Dora 支持均代表着对复杂 glibc 应用的支持。

ZLMediaKit 支持所需功能详见 [ZLMediaKit 分析](#)。目前对 ZLMediaKit 支持已经完成了 server 端的支持，下一步需要将 ffmpeg 启动在 Starry 上，并且通过局域网通信向 ZLMediaKit 进行推流。

ext4 文件系统支持

目前 Starry 已经支持的文件系统如下：

- fatfs：原生 rust-fatfs 支持
- lwext4 文件系统：以 C 语言转 Rust 支持
- ext4fs：一个由 Rust 编写的 ext4 文件系统，支持 ext4 的各种功能
- another_ext4fs：由 Rust 编写的 ext4 文件系统，用于进行文件系统 model check

这些文件系统底层模块由各自负责的人员开发完成之后，通过模块化的思想接入到 Starry 中。以 ext4fs 运行的指令如下：

```
# Build the image in ext4fs
$ ./build_img.sh -a x86_64 -fs ext4
```

```
# Run in the lwext4fs with Rust interface, whose url is https://github.com/elliott10/lwext4_rust.
$ make A=apps/monolithic_userboot FEATURES=lwext4_rust LOG=off NET=y BLK=y ACCEL=n run

# Run in a new ext4fs written in Rust, whose url is https://github.com/yuoo655/ext4_rs.
$ make A=apps/monolithic_userboot FEATURES=ext4_rs LOG=off NET=y BLK=y ACCEL=n run

# Replace virt-io with ram-disk
$ make A=apps/monolithic_userboot FEATURES=ext4_rs,img LOG=error NET=y BLK=y ARCH=x86_64 ACCEL=n
run

# Run testcases for OSCOMP
$ make A=apps/monolithic_userboot FEATURES=ext4_rs,img LOG=off NET=y BLK=y ACCEL=n run
APP_FEATURES=batch
```

组件化内核优点介绍

在进行多构型组件化内核开发的过程中，我们逐步感受到组件化内核的优点。

组件便于复用

组件化内核的一个重要要求便是组件的复用性。组件的复用性带来的是**开发效率的提高以及后续维护工作压力的降低**。在我们的工作中，组件的复用主要包括两个方面：多架构的复用和不同内核的复用。

多架构的复用

多架构的复用是指使用分布的组件，快速地搭建不同架构的内核，从而应对不同的应用场景。以下是复用组件开发不同架构内核的一些例子：

- Starry 通过复用原先的组件，并添加所需要的宏内核组件，可以支持选择宏内核或者 Unikernel 启动。在支持 Dora 时，对底层网络栈模块的修改可以同时作用到两种内核架构上，从而同步改进工作
- Hera OS 是同类型的一个参考例子，由我们的指导教师杨金博老师开发，他复用了宏内核 ByteOS 独立出来的 polyhal、驱动、文件系统等模块，仅改动核心部分，使用 1100 行内核主体代码便完成了一个微内核的构建，极大提高了开发效率

不同内核的复用

组件复用性的一个更加常见的场景是应用在不同内核中。证明组件划分、实现规范的一个很重要的因素便是组件被足够多且不同类型的内核复用。复用的内核包括成熟的内核和正在丰富完善内容的新内核。以下是一些典型模块的复用例子：

- crate_interface：这是一个 ArceOS 中用于打破依赖关系的模块。在之前的[组件化内核设计原则](#)中提到了当模块越来越多、依赖关系越来越复杂的时候，循环依赖的问题出现的可能性会越来越高。因此 ArceOS 中采用了类似于 C 的符号跳转形式来打破依赖关系的限制，将其封装在了 crate_interface 模块中。这个模块也被用在了 ByteOS，lk_model(<https://github.com/shilei-massclouds/lkmodel>) 等内核上，是一个较为方便地实现跨组件函数调用的模块
- elf_parser：这是 Starry 中实现的 ELF 解析模块，可以将 rust 中提供的第三方库 xmas-elf 进一步封装，提供便于内核页表分析的地址区段，对动态链接字段进行重定位等处理，从而快速完成 ELF 文件的分析。这个模块也用在在了新内核 arceos-monolithic 的开发中，较快地完成了 ELF 文件的加载解析。

- polyhal 是同类型的一个参考例子，由我们的指导教师杨金博老师开发，他是从 ByteOS 中独立出来的硬件抽象层，目前可以用在 rCore, ByteOS, hera, qurdos 等多个内核上，提供泛用硬件抽象层支持。

当对内核组件进行拆分的时候，我们发现越是接口多的模块，**适配到一个成熟的内核的难度就越高**。以上文的 polyhal 为例，它已经适配到数个内核中，但是当我们在尝试适配到 Starry 时，发现他的接口设计和 Starry 原生的硬件抽象层 axhal 差距较大，想要直接适配需要做的修改会很多，相反，我们独立出来的模块用来**组装新内核**就会较为方便，可以较大程度地减小新内核开发的压力，如 hera, arceos-monolithic 内核均是如此。因此这也是组件化的一个很有意义的应用场景，即为**新内核的组装**提供多样且快速的选择。

在支持 Dora 前，ArceOS 便已经有了较为成熟的底层模块。Starry 通过复用这些模块，可以跳过复杂的网络、文件系统等模块和驱动的开发，更加专注于 Dora 本身涉及到的 syscall 的分析和实现，从而做好 syscall 兼容层的打磨和适配。

这一点很好地回应了摘要中提到的复杂代码维护问题，可以**让负责各部分代码的人更加专注地实现自己的功能，从而提高内核的开发效率，方便内核模块和功能的维护**。这在之前介绍 Dora 时已经有了体现。在进行 Dora 的适配的时候，我们需要对网络栈进行修改。这部分的修改不仅可以适用在宏内核，也可以体现在 Unikernel 中。同样的组件修改可以同时体现在不同架构下，降低了开发和维护的成本。当不同的内核共享相同的组件时，对组件的 bug 修复或者新特性的引入可以让所有内核都可选地进行更新，从而丰富引用该模块的内核的功能。它为内核开发带来的是开发效率的提高（引入合适的模块比自己重写一个要方便地多）与维护成本的降低（模块维护者的更新可以让所有引用该模块的内核都看到）。

没有明显的额外性能开销

组件化内核一个重要的问题即是其性能问题。为了通用，我们需要对模块进行拆分，这是否会在性能上带来明显的开销呢？为了验证这一点，我们分别测试了相近组件不同架构和不同内核在同一批测例下的性能表现，以此说明组件化对性能并不会带来明显的开销。

相近组件不同架构

Starry 本身是基于 ArceOS 开发的，复用了 ArceOS 的许多组件，但是也在这个基础上添加了宏内核所需要的扩展，同时适配了 Unikernel 和宏内核。我们手动了测例，对**宏内核和 Unikernel 的 Starry**进行了测试。

具体测试内容在：

<https://github.com/Azure-stars/Starry/blob/feat/module/doc/OS-Train-Repo/Week4.md>，测例地址为：<https://github.com/Azure-stars/Starry/tree/feat/module/apps/ostrain>

根据宏内核和 Unikernel 的异同之处，考虑从以下几个方面来进行测例性能测试：

- 文件：包括读写等内容
- 任务：新建线程、任务切换、睡眠
- 内存分配：动态分配堆内存、对内存空间的读写操作

文件 IO 测试

反复对一个文件进行打开、读或者写操作 50 次，计算消耗的时间，并重复该操作 10 次计算平均值。

测例	Unikernel	Monolithic
Fileopen	0.55s	1.066s
Fileread	0.61s	1.07s
Filewrite	1.69s	2.21s

内存分配测试

测试操作分为两种：

A：连续申请内存，并一次性全部释放。

B：申请一次内存，并立即释放，然后继续申请下一次内存，不断重复。

进行 Num 次操作 A 和 操作 B，计算消耗的时间。

Num	10	50	100	500	1000	5000	10000
Unikernel	0.4ms	1.8ms	3.74ms	19.11ms	39.34ms	0.185s	0.377s
Monolithic	0.81ms	2.8ms	5.69ms	28.51ms	57.33ms	0.335s	0.557s

任务调度

新建 50 个线程，每一个线程执行 Num 次 yield 函数，计算最后所有线程释放时经过的时间：

Num	Monolithic	Unikernel
1	54.76ms	3.44ms
10	56.68ms	0.707ms
15	58.62ms	0.92ms
20	59.65ms	1.195ms
25	60.81ms	1.33ms
50	67.46ms	2/25ms
60	70.33ms	2.86ms
70	72.74ms	3.28ms
80	75.32ms	3.65ms
90	77.63ms	4.03ms
100	80.40ms	4.20ms

根据上述测试可以看出，在使用组件构建内核的时候，即使这些组件一开始是面向 ArceOS Unikernel 设计的，但是在扩展成为宏内核之后，他们的性能表现差距也在合理的范围内（1倍到20倍左右），和正常的 Unikernel（如 Unikraft）相比宏内核（Linux）的差距类似，说明组件化模块设计在进行多架构扩展的时候对性能的影响不会特别明显。

不同内核

这里的测试期望是将我们的组件化内核和其他不考虑组件化划分的内核进行对比测试，从而验证组件化的拆分设计对于内核的性能没有明显的性能开销影响。不过这个验证的困难之处在于：我们的内核组件一开始便是以组件化为目标设计的接口，而不是从一个整体的内核开始拆分。其他的内核的组件完善程度和 Starry 也不完全一致，组件所做的优化也可能影响最终的性能测试结果。

Starry 参与了 2023 年全国大学生计算机系统能力大赛内核赛道决赛，相关性能测试位于 <https://github.com/oscomp/testsuite-for-oskernel/tree/final-2023>，在线下赛决赛中，**性能测试总分排在第 6 名（现场共有超过 40 支队伍参与评分）**，和部分一等奖内核差距不大。同时后续研究发现，Starry 在实现宏内核扩展时更多考虑了功能的正确性，对性能优化没有做特别多的考虑，如 COW 等优化暂时还未实现。考虑这些因素在内，我们认为组件化的模块设计并没有在很大程度上影响内核的性能。

不过上述分析也反映了组件化的一个好处：即我们可以**对单个组件进行更加针对性的优化**，这个优化工作可以体现在引用了该模块的各个内核上，工作成果更为显著。

分层解耦与可替换性

所谓组件分层解耦与可替换性，即在组件开发过程中，**单向依赖**性质保证下的内核组件具有明显的分层特征，如上文中的 [Starry 架构图](#) 中的组件关系介绍图。复杂的内核通过组件化的划分变成了多个独立的组件，**彼此之间解耦**且有上下层的依赖关系，还可以通过实现指定的接口来**完成内核模块的替换**，如文件系统、调度模块的替换（详见 [模块替换原则](#)）。这部分性质的保证不仅与模块本身的划分和接口设计有关，还与模块间的**单向依赖关系**有关。

单向依赖介绍

在使用 Rust 开发组件化内核的过程中，Cargo 会严格保证组件之间的单向依赖特性。所谓单向依赖，即模块间的调用关系满足单向性，只能由上层调用下层，而尽量避免下层模块调用上层实现。之前的章节已经提到过 [循环依赖](#) 的问题，实际上在模块化的过程中，随着模块的数量和依赖关系越来越复杂，循环依赖的问题出现的可能性会越来越高。

虽然 Rust 的 Cargo 管理要求模块之间不允许存在显式的环形依赖，但是可以通过符号链接等方式避开这一限制，从而实现下层调用上层内容。Rust core 库中的 `panic_handler` 等即是类似的形式，而 ArceOS 的 `crate_interface` 也是通过符号链接的形式来实现这一功能。但这一方式需要用到不安全的符号链接机制，相当于绕过了 Rust 的安全检查，可能会导致潜在的安全问题，并且使得模块调用关系更加复杂。

我们期望的单向依赖不仅是模块依赖上的显式关系，还有函数调用关系上的单向，即需要**尽可能避免不安全的符号链接**，或者将这些操作的频次与重要性降得尽可能低，保证代码流运行上的有序性。

Starry 的单向依赖工作

我们以 trap 处理这一个内核常见的功能为例，解释我们对循环依赖的改动方式和对单向依赖的实现方式。

ArceOS 原生的硬件抽象层 axhal 实现了对硬件平台和指令架构的一系列操作，包括启动、trap 处理和对上接口提供。其中 trap 处理由于需要涉及汇编中保存任务上下文等操作，在不同指令架构下有不同的实现内容，一个常见的做法便是放在了硬件抽象层（如 axhal）中。

在 trap 处理的时候，需要根据异常类型进行分发，即有如下伪代码：

```
fn trap_handler(tf: Trapframe) {
    match cause {
        page_fault => handle_page_fault(tf),
        syscall => handle_syscall(tf),
        irq => handle_irq(tf),
        ...
    }
}
```

因此 trap 处理需要调用 syscall 的处理函数。但是 syscall 一般需要处于较高的层级，在完成了各个模块的功能之后，通过 posix syscall 的语义来实现每一个 syscall 的功能并且进行分发，而 axhal 本身属于较为底层的模块。

如果将 trap 处理放在较为底层的模块，而 syscall 具体实现放在上层，则会导致频繁的底层调用上层的情况，出现循环依赖的问题。

为了实现单向依赖这一目标，Starry 不仅仅复用了 ArceOS 的模块，还对他们进行了改造，具体包括如下方面：

1. 分拆硬件抽象层

为了解决这一问题，Starry 将 axhal 分拆为三个模块：`axhal`，`axtrap` 与 `arch_boot`。三个模块各自的功能如下：

- `axhal`：提供平台相关的接口函数，如时间、内存区域等；提供与指令架构相关的函数，如开关中断、寄存器交换保存
- `axtrap`：提供 trap 入口和处理函数，包括 syscall ‘interrupt ‘exception 等
- `arch_boot`：提供内核启动相关汇编，支持从 multiboot ‘SBI 等启动内核

根据上文提到的 [架构图](#) 可以看出，`arch_boot` 和 `axtrap` 均处于较高层次，用于进行内核的启动和 trap 的集中入口，而底层的 `axhal` 则仍然保持自身提供对上接口的功能。`axtrap` 需要调用的 syscall 函数可以从下一层的 `linux_syscall_api` 层获取，从而解决了反向依赖的问题。

2. 分拆任务模块

ArceOS 实现了任务互斥锁，即当不同互斥锁同时申请同一资源时，会触发任务的等待和调度。相比于传统的自旋锁，互斥锁很好地提升了效率。但是 ArceOS 中在 `axhal` 层定义了较多 `percpu` 变量，对 `percpu` 变量的访问需要使用互斥锁来进行保护（禁止抢占）。依据 ArceOS 的架构图可以看出：互斥锁的实现需要调用 `axtask` 的相关功能（任务的调度），而 `axtask` 本身又依赖于携带了互斥锁的 `axhal` 层，从而导致了环形依赖的出现。

为了解决这一问题，我们采用了 **资源定义和实现分开存放** 的思想，新定义了 `taskctx` 模块，用于存放任务上下文信息。互斥锁对任务调度状态的改变可以直接作用在 `taskctx` 上，而 `axhal` 可以继续采用 `kernel_guard` 中定义的互斥锁。`axtask` 也会引用 `taskctx` 的内容，根据其存放的任务上下文信息进行实际的任务调度操作，从而完成单向依赖目标的实现。

依据上述思想，我们还对其他的部分模块做了改动，尽可能去除各种存在的环形依赖的情况。目前 Starry 保留的单向依赖情况如下：

需要依赖的函数	函数功能	保留的原因
axlog	log 日志输出	这里的单向依赖是为了更好地获取日志的信息，包括当前运行所在核和任务 ID。由于是调试模块，我们认为其不影响正常功能的实现，故将其保留。
panic_handler、 global_allocator 等	Rust core 库必需功能	这是 Rust core 库的必需功能，是形成一个完整可执行文件的必需内容，由编译器保证其安全性，因此我们予以保留。

分层解耦可替换性的应用优势

1. 模块引用关系清晰简明，通过编译期进一步保证内核的安全性：在保证单向依赖的情况下，不同函数的引用关系和模块的显式依赖关系是匹配的，而不会出现以前的 `extern "C"` 不安全符号引用，从而通过 Rust 语言的编译器进一步保证内核的安全性。
2. 便于通过配置文件选择内核部件启动：在单向依赖的情况下，可以清晰地画出模块的依赖图，从而通过条件编译等形式方便地选择需要启动的模块。这一点不仅可以用在定制化的 Unikernel 中，也可以用于宏内核开发。以上文 [Starry 的架构图](#) 为例，假设我需要开发并且测试 `axfs` 模块，目前会有三种方式：

1. 在 `axfs` 模块中单独编写单元测试用例，通过 `cargo test` 进行测试。这个测试方法最为简单，但是无法在裸机环境下真正测试和文件系统驱动的交互。
2. 将 `axfs` 模块和其他所有模块打包起来作为一个完整的宏内核，运行给定的可执行文件测例，类似于内核赛道的测试方法。这个方法可以测试真实环境的文件系统表现，但是编译、启动以及运行都会很花时间。在文件系统开发初期接口尚未完善的时候，为了能够打包成为一个完整的宏内核，还需要伪造实现各个未实现的接口，才能通过整个内核的编译，不利于我们进行便捷的测试。
3. 通过配置文件，选择将 `axfs` 与其子模块打包在一起，配合上最小内核启动必须的 `arch_boot` 组件，抛弃其他兄弟和父组件。这样会形成一个能够运行 `axfs` 模块功能的最小内核（默认是 Unikernel）。

我们可以将这个内核运行在 QEMU 上，从而完成裸机测试文件系统的目的。

另外由于 `axfs` 不需要被父组件所引用（`arch_boot` 只需要一个 `main` 函数入口即可），因此它无需伪造实现目前暂时还没有实现的接口，从而保证测试的便捷性。

上面提到的第 3 点测试方式在单向依赖保证下能够较为顺利的编译、运行，也能够让开发人员真正地像开发一个应用程序一样开发一个内核模块。

3. 便于进行安全验证：近年来关于内核的安全性讨论十分热烈。许多内核都提出了自己保证安全性的思路，如限制不安全代码所在范围、形式化验证内核模块等。对于形式化验证，以前的工作的一大痛点便是验证工作量巨大。形式化验证要求对于指定验证的内核代码需要编写对应的验证语句，当内核代码量较为庞大时，这个验证工作会很复杂。因此如何将验证工作细化拆分为多个模块，能够复用已有的验证成果便显得十分重要。

在组件化模块化系统设计的思想下，我们期望每一个模块的验证工作由底层模块验证和自身模块内容验证组成，通过对每一个模块都进行各自的形式化验证，上层复用下层模块的验证成果，从而降低上层模块的验证难度。在之前的 ArceOS 中，由于反向依赖的存在，底层模块需要依赖上层模块的实现，在没有完成所依赖的上层实现的验证前，对底层模块的验证将是不完整的。而在单向依赖的内核下将不会有这个问题，对组件化的形式化验证也会更加便利。

<https://gitlab.eduxiji.net/1202410003993297/project2210132-239820/-/tree/verifyngkernel>，它将 ArceOS/Starry 的其中两个模块 `bitmap_allocator` 和 `memory_addr`，分别是内存分配器和地址管理封装模块，均是两个处于底层、不依赖于其他内核模块的模块。在完成了这两个模块的验证之后，我们通过单向依赖的特性，可以保证当引入的模块正常工作时（即不会被其他模块非法访问修改等原因被影响），其表现的形式和语义规定是一致的，即引入模块实现和工作的正确性，从而在证明上层模块的时候减少证明的复杂度，促进验证工作的开展。

效果展示

<https://github.com/Starry-OS/Starry/actions/runs/10132474884> 中记录了 Starry 的 CICD 情况。可以看到 Starry 目前的 CICD 均是同时**测试 Unikernel 和 宏内核架构**的测例，且对于每一种内核构型均**同时测试 x86 64, riscv64, aarch64 三种指令架构**，保证多指令架构支持的功能。

在复杂 APP 支持方面，我们支持了 Dora, tokio, redis 等复杂应用。对于 tokio, redis 等多是基础应用，展示性不强。对于 Dora，我们的工作成果如下：

- Starry 在 Qemu 上成功启动 Dora，可以在同一台机器上运行 daemon 'coordinator 和多个 node
- 完成了 Dora 的 benchmark 测试，测试截图如下：

```

9   build_mode = release
10  log_level = off
11
12  Running testcase: ./dora up
13  started dora coordinator
14  started dora daemon
15  Testcase ./dora up finished with exit code 0
16  Running testcase: ./dora start dataflow.yml
17  00000000-12c4-71da-af3d-c380d5075317
18  attaching to dataflow (use `--detach` to run in background)
19  Latency:
20  size 0x0      : 2.675384ms
21  size 0x8      : 2.711931ms
22  size 0x40     : 2.61393ms
23  size 0x200    : 2.925648ms
24  size 0x800    : 2.832945ms
25  size 0x1000   : 3.035694ms
26  size 0x4000   : 3.539478ms
27  size 0xa000   : 6.038062ms
28  size 0x64000  : 112.375499ms
29  size 0x3e8000: 1.386603547s
30  Throughput:
31  size 0x0      : 595 messages per second
32  size 0x8      : 550 messages per second
33  size 0x40     : 423 messages per second
34  size 0x200    : 315 messages per second
35  size 0x800    : 279 messages per second
36  size 0x1000   : 288 messages per second
37  size 0x4000   : 77 messages per second
38  size 0xa000   : 65 messages per second
39  size 0x64000  : 14 messages per second
40  Input `latency` was closed
41  Input `throughput` was closed
42  size 0x3e8000: 1 messages per second

```

- QEMU 上成功连接上局域网的机械臂主机，并且可以通过 Dora 下达指令启动/关闭机械臂。这是 [相关演示视频链接](#)

在性能测试方面，具体测试内容在：

<https://github.com/Azure-stars/Starry/blob/feat/module/doc/OS-Train-Repo/Week4.md>，测例地址为：
<https://github.com/Azure-stars/Starry/tree/feat/module/apps/ostrain>。相关的对测例的分析详见：
 性能测试结果。

reL4

reL4基于seL4开发

执行方式如下：

```

mkdir reL4
cd reL4
repo init -u git@github.com:reL4team/mi-dev-repo.git
repo sync
cd build-scripts
make ARCH=riscv64 seL4-test #使用riscv64架构进行构建，并运行seL4的测例
make ARCH=aarch64 seL4-test #使用aarch64架构进行构建，并运行seL4的测例

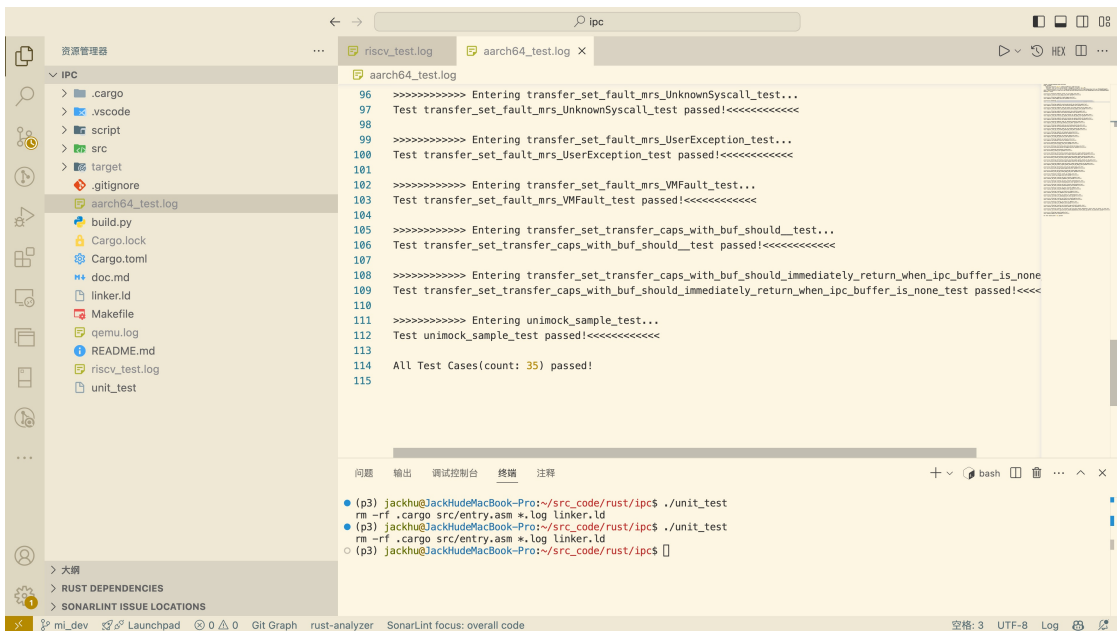
```

由于reL4本身兼容seL4，因此我们可以运行所有的seL4原生的测例，运行结果如下

```
Ubuntu 22.04.3 LTS
Test TRIVIAL0002 passed
Starting test 111: VSPACE0000
Running test VSPACE0000 (Test threads in different cspace/vspace)
test_name: VSPACE0000, Test cost: 0
Test VSPACE0000 passed
Starting test 112: VSPACE0001
Running test VSPACE0001 (Test unmapping a page after deleting the PD)
test_name: VSPACE0001, Test cost: 0
Test VSPACE0001 passed
Starting test 113: VSPACE0002
Running test VSPACE0002 (Test create ASID pool)
test_name: VSPACE0002, Test cost: 0
Test VSPACE0002 passed
Starting test 114: VSPACE0003
Running test VSPACE0003 (Test create multiple ASID pools)
test_name: VSPACE0003, Test cost: 0
Test VSPACE0003 passed
Starting test 115: VSPACE0004
Running test VSPACE0004 (Test running out of ASID pools)
test_name: VSPACE0004, Test cost: 0
Test VSPACE0004 passed
Starting test 116: VSPACE0005
Running test VSPACE0005 (Test overassigning ASID pool)
test_name: VSPACE0005, Test cost: 0
Test VSPACE0005 passed
Starting test 117: VSPACE0006
Running test VSPACE0006 (Test touching all available ASID pools)
test_name: VSPACE0006, Test cost: 0
Test VSPACE0006 passed
Starting test 119: Test all tests ran
Test suite passed. 119 tests passed. 52 tests disabled.
All is well in the universe

Test end =====
```

reL4 目前支持 riscv64 和 aarch64 指令架构，通过采用单元测试保证功能的正常运行。以下为 reL4 对于 aarch64 架构的测试运行截图：



总结

本文先提出了组件化内核的概念，分析为何需要使用组件化模式进行内核开发。在此基础上，我们依照组件化内核的开发思想，分析现有的内核存在的特征和问题，提出了数条[组件化内核的设计和开发原则](#)。接下来我们在这些原则的指导下开发了模块化操作系统 Starry 和 reL4。这些实践内核本身也处于积极维护的状态，具有**创新性**（多架构支持）、**泛用性**（多指令架构支持）以及**实用性**（支持多种常见且复杂的应用）。接下来我们根据这些实践内核的表现，分析了组件化内核的复用性和性能，认为如果模块划分得当，我们可以让组件被更多的内核复用，并通过组件的专门维护者精心打磨优化组件，从而为更多内核带来性能上的进步。

我们还分析了组件化的另外一个优势，即有利于梳理、规范化模块之间的依赖关系，并且通过合理的划分形成**单向依赖的关系**，从而拥有解耦和可替换性。这个性质可以帮助我们实现**可选模块启动内核**，做到像**开发应用一样开发内核**，也有利于**形式化验证的逐层进行**。

回顾概述中的问题，我们认为多构型组件化内核的设计思想相比传统内核设计而言，可以减小成熟内核开发、维护的压力，也可以通过组装组件快速地构建起来一个合适的新内核（如 [Hera OS](#)），是一种值得继续深入探索的内核开发模式。

目前我们对模块的划分设计仍然具有一定的局限之处，在开发模式上也有不甚便利的地方。在将来，我们也会继续在多构型组件化内核开发上的探索，总结更为简明有效的设计原则，探索更为高效的开发模式，进一步展示组件化内核的优势。

鸣谢

感谢清华大学的陈渝老师、向勇老师、李明老师、杨金博老师、泉城实验室的石磊老师、CICV 的郭伟康老师对我们的指导。感谢贾越凯学长、闭浩扬学长、胡柯洋学长、田凯夫学长、萧络元工程师、陈乐工程师、李扬工程师、胡志文学长、李龙昊学长在我们的开发过程中给予的帮助和支持。另外也感谢为 Starry 和 reL4 内核做出贡献的柏乔森、李昕昊、杨金全等各位协作者。