

灵活组合的操作系统模块和框架 ArceOS

队员：郑友捷、朱懿

学校：清华大学

指导老师：杨金博

项目导师：陈渝

灵活组合的操作系统模块和框架 ArceOS

摘要

模块化内核介绍

ArceOS 介绍

Starry 介绍

reL4 介绍

组件化内核的创新性：内核多架构支持

原有 ArceOS 模块的扩展

宏内核

微内核

效果展示

宏内核

微内核

多架构意义

组件化内核的泛用性：多指令架构支持

组件化内核的实用性：Dora 支持机器人运行

Dora 介绍

背景

Dora 运行流程简要说明

工作内容

ArceOS 工作

Starry 工作

工作成果

Starry 工作的其他实用成果

ZLMediaKit 支持

Tokio 支持

ext4 文件系统支持

Starry 工作量说明

组件化开发模式

对组件化内核的参考

单向依赖的支持

单向依赖说明

单向依赖工作

单向依赖的意义

鸣谢

摘要

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。

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，我们决定基于模块化操作系统 ArceOS，进一步开发完善模块化的系统，组合出微内核、宏内核的内核架构，并且通过支持实际的应用证明其模块化的可行性和意义。

模块化内核介绍

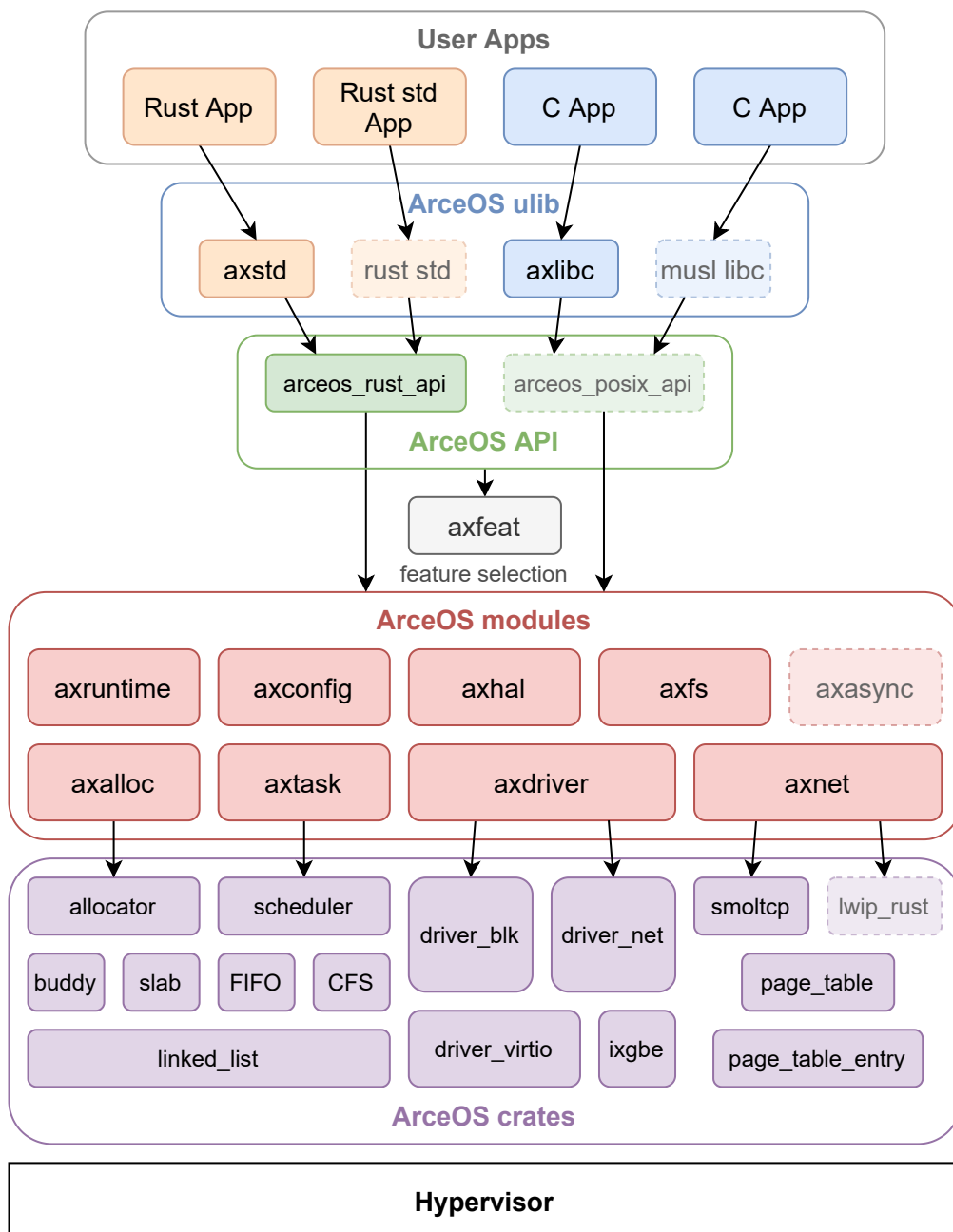
ArceOS 介绍

ArceOS 是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：



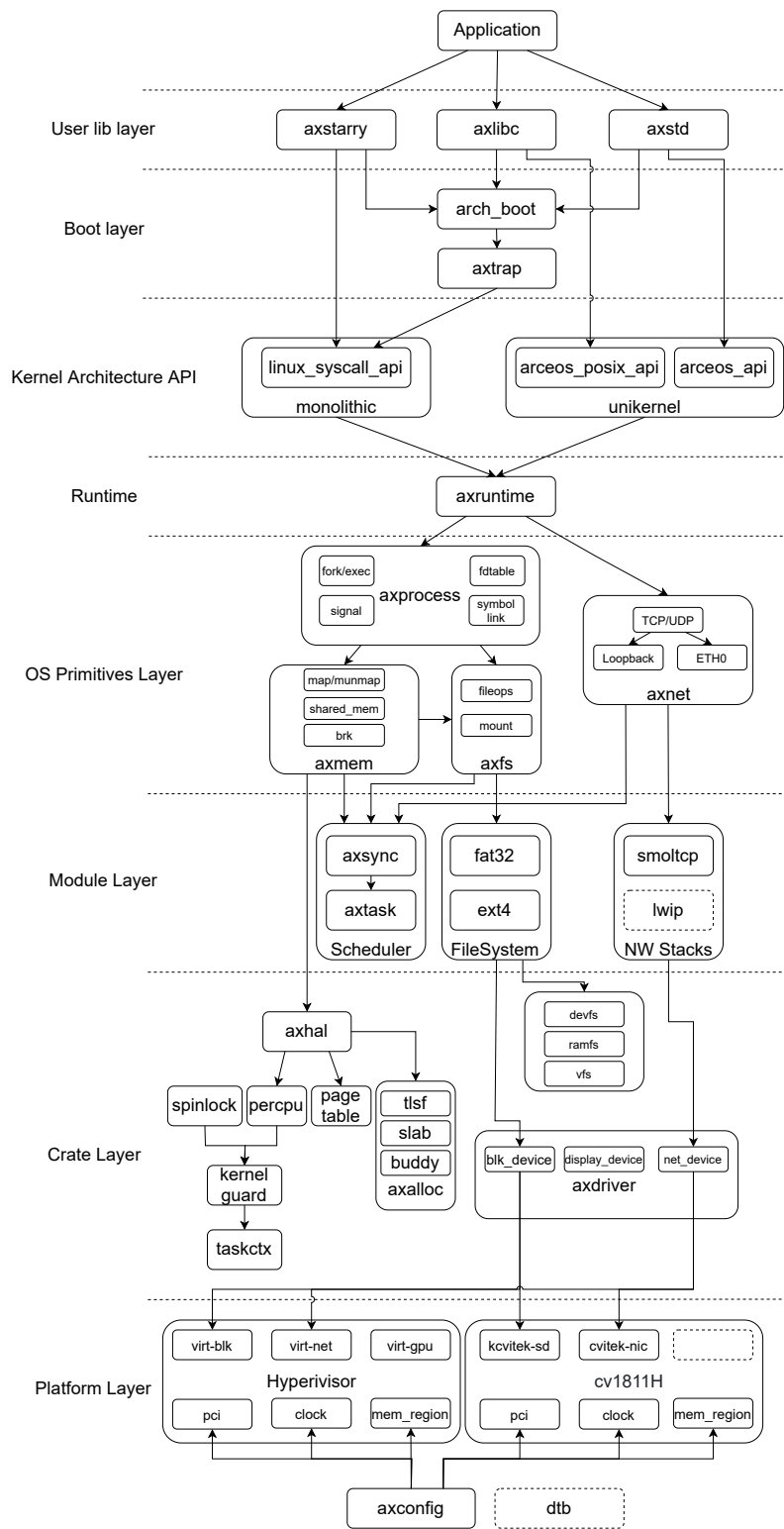
ArceOS 本体目前支持的较为完善的是 Unikernel 结构，但由于其设计的优势，理论上我们可以通过简单的复用其组件组合出宏内核、微内核、hypervisor 等不同架构，从而将架构区别对内核开发带来的影响降到较低的程度。

Starry 介绍

ArceOS 本身虽然是 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。

作为基于 ArceOS 的宏内核，Starry 保持了 ArceOS 本身的功能，通过条件编译等手段可以选择以宏内核还是 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，从而体现了组件的复用性。

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。

reL4 介绍

reL4 微内核是基于 seL4 架构，使用模块化的方式及 rust 语言实现的微内核，旨在通过 seL4 本身的形式化验证以及 rust 语言的安全性，形成新一代的安全微内核。目前已经支持 riscv64 及 aarch64 两个架构在 qemu 上的运行。相关的代码开发组织为：

<https://github.com/reL4team/>

作为基于 seL4 的微内核，鉴于 seL4 内核编译工具链复杂的问题，reL4 使用了模块化的方式，使用 rust 重构了其代码。

seL4 微内核最终的运行镜像，除了需要将内核放入之外，还需要放入设备树，以及相应的用户态的 lib，和用户 rootserver 进程，在原本的代码仓库中，这些代码都糅杂在一起，并且为了能够通过形式化证明，加入了相当多的内容，虽然是微内核，但是代码冗余复杂。

因此 reL4 将其中的内核代码抽象出来，并拆分成 **rel4_kernel**，**rel4_ipc**，**rel4_common**，**rel4_vspace**，**rel4_cspace** 等五个模块。并且在每个模块中，剥离了每个模块与具体架构相关的代码放入各个模块的 arch 文件夹下，最大程度的保留了与架构无关，以及与操作系统内核设计无关的部分，从而实现良好的可迁移性，以便于组件的复用。

除此之外，reL4 还做到了和原生 seL4 的良好兼容性，reL4 完整的实现了 seL4 的 syscall，以及实现了在此基础上作为真正系统调用的，向内核申请分配对象的 invocation 机制，实现了内核间通信的 fastpath 以及 notification 等机制，并完整的通过了 seL4 的所有 test 测试。

参赛队员朱懿作为 reL4 内核开发的主力队员，在使用模块化技术开发 reL4 的工作中做了突出贡献。

组件化内核的创新性：内核多架构支持

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。

原有 ArceOS 模块的扩展

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。

效果展示

宏内核

Starry 基于 ArceOS 开发，保留了 Unikernel 通过 APP 启动的方式。在 <https://github.com/Starry-OS/Starry/tree/main/apps> 中提供了多个启动 APP。其中选定 `Monolithic_userboot` 即可使用宏内核启动，并运行 `testcases` 文件夹下指定的测例内容。[这里](#) 对多架构支持做了详细的说明。

以 C 语言为例，我们希望**通过将代码链接到不同的库**来启动不同架构的内容：

- 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。

<https://github.com/Starry-OS/Starry/actions/runs/10132474884> 中记录了 Starry 的 CI/CD 情况。可以看到 Starry 目前的 CI/CD 均是同时测试 Unikernel 和 宏内核架构的测例，保证多架构支持的功能。

jobs

- ### Run details

- Triggered via push 2 days ago

Azure-stars pushed dab9c01 main

Status

Success

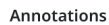
Total duration

4m 29s

Artifacts

1000

on: push



8 warnings

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 =====
█
```

多架构意义

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**。

同一个实际应用运行在不同架构下的例子详见下文的 [Starry 支持 Dora 运行](#)。

组件化内核的泛用性：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 CI/CD 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。

组件化内核的实用性: Dora 支持机器人运行

Dora 介绍

背景

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 运行流程简要说明

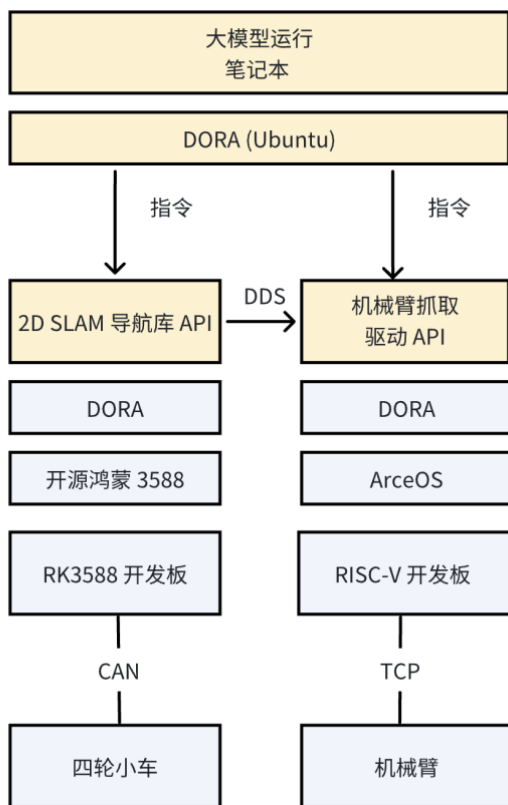
Dora 分为三部分, 分别是 `coordinator`, `daemon`, `cli`

- `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。

工作内容

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 888 888 888 888 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。

工作成果

- 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:

```
build_mode = release
log_level = off

Running testcase: ./dora up
started dora coordinator
started dora daemon
Testcase ./dora up finished with exit code 0
Running testcase: ./dora start dataflow.yml
00000000-12c4-71da-af3d-c380d5075317
attaching to dataflow (use `--detach` to run in background)
Latency:
size 0x0      : 2.675384ms
size 0x8      : 2.711931ms
size 0x40     : 2.61393ms
size 0x200    : 2.925648ms
size 0x800    : 2.832945ms
size 0x1000   : 3.035694ms
size 0x4000   : 3.539478ms
size 0xa000   : 6.038062ms
size 0x64000  : 112.375499ms
size 0x3e8000: 1.386603547s
Throughput:
size 0x0      : 595 messages per second
size 0x8      : 550 messages per second
size 0x40     : 423 messages per second
size 0x200    : 315 messages per second
size 0x800    : 279 messages per second
size 0x1000   : 288 messages per second
size 0x4000   : 77 messages per second
size 0xa000   : 65 messages per second
size 0x64000  : 14 messages per second
Input `latency` was closed
Input `throughput` was closed
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](#)

Starry 工作的其他实用成果

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。

Tokio 支持

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。

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 OSCP
$ make A=apps/monolithic_userboot FEATURES=ext4_rs,img LOG=off NET=y BLK=y ACCEL=n
run APP_FEATURES=batch
```

Starry 工作量说明

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

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负责。

组件化开发模式

为了便于组件化模式的开发，我们在实际工作中也探索了一些方法和开发模式。以下是组件化开发过程中的一些工作范式的整理，也有一些内容体现在了 Starry 的开发中。

- 1、使用 repo 工具便于多仓的开发：多个git仓库以及多人进行开发的时候，需要进行同步协作，使用 repo 而不是 git 的 submodule，可以便于多人本地同时调整多个仓库。
- 2、使用一个大仓库加上多个小仓库的模式进行开发：在合并过程中，如果同时改动了多个小仓库，那么在合并的时候，不仅耗费时间，如果只合并了部分的时候，其他人又进行了合并，这会造成多个仓库的难以兼容。因此采用一个大仓库，并分发到多个小仓库的方式，减少工作量以及冲突的可能。
- 3、使用规范的合并方式：在实际过程中遇到过非主分支的两个分支互相合并导致最后极为复杂的合并提交。因此，强制每个分支都必须只能合并主分支的代码。并且推荐使用rebase 而不是 merge，以及推荐使用 squash 的方式保证分支历史的干净。
- 4、代码合并责任明确化：在多个对等的程序员的协作开发过程中，不同于一个 reviewer 加上多个提交者的方式，如果每次都让某个人进行 review，会对其造成巨大的负担。因此在合并到主分支的过程中，最后我们使用一个提交者并找到一个 reviewer 共同对该次提交负责的方式，进行代码开发。

5、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。

对组件化内核的参考

在 Starry 适配 Dora 的过程中，ArceOS 本身的文件系统、网络模块等模块的功能基本都是可以直接复用的，另外需要在原有模块的基础上完善一些功能，包括共享内存、网桥等。其余的主要开发工作均是对 syscall 兼容层的语义检查和适配。这里体现了 Starry 模块化开发的一些优势之处：

- 在基本不改变底层模块的情况下**以多种内核架构**支持应用

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。

- 提供一个 **syscall 兼容层**，用于处理各个 syscall 的语义检查和实现

Starry 使用的底层模块原来是为了 Unikernel ArceOS 服务的。为了支持宏内核功能，Starry 添加了 linux_syscall_api 兼容层，用来进行 posix syscall 的相关语义检查和实现。这一层可以在将来通过进一步抽象，定义出其所需要的底层内核提供的接口，**从而让更多的内核可以复用这个模块**，省去复杂且易错的类型定义和检查过程。

单向依赖的支持

在模块化的基础上，我们对 Starry 的结构设计提出了更加深入的要求，即模块间依赖关系尽可能形成一个 DAG（有向无环图），避免不安全的符号链接操作，从而更加方便模块的管理以及深入的形式化验证等工作。

单向依赖说明

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。

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。不过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，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。

单向依赖工作

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

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。但 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。

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，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。

单向依赖的意义

单向依赖的内核有如下的优势：

- 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。这里的关键在于验证工作仅与底层身模块的验证组成。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。

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 内核做出贡献的各位协作者。