摘　要

为某个特定应用或者场景定制操作系统是一个热门且常见的工作。但定制内核一般都是面向某一个特定的内核架构，无法充分适配到不同内核架构对应的场景，比如微内核的安全隔离、Hypervisor 的虚拟化等。当需求发生改变时（这在当前时代并不罕见），为了适配到新的内核架构，要么重新定制开发，要么采用已有的通用内核，无法利用已有的定制工作带来的好处，如通过形式化验证的模块或是统一的通信协议。

因此我们希望能快速地将定制工作迁移到不同内核架构上（即快速定制不同形态的内核）。为了达成这个目标，我们研究了不同内核形态的异同点，并且尝试在看似无关的多种内核架构之间搭起一座桥梁，从而以较小的代价实现不同内核架构的转化。我们提出了灵活内核的设计原则，将不同架构共享的内核基础服务功能视为“内核意义上的库”，而由内核在完成初始化后的执行逻辑决定最终表现出来的内核形态。ArceOS 是我们验证灵活内核设计原则的实验内核，它是一个以组件化的思想设计开发的 Unikernel。目前 ArceOS 可以用较小的代码量快速构建宏内核、Unikernel、Hypervisor 等多种内核形态。通过复用同一套基础内核服务，我们发现新内核形态开发的工作量和难度显著降低，也为快速定制不同形态内核提供了新的实现思路。

关键词：组件化；操作系统；异构

Abstract

Customizing an operating system for a specific application or scenario is a popular and common task. However, custom kernel development typically targets a specific kernel architecture, making it difficult to fully adapt to various kernel architectures for different scenarios, such as microkernel security isolation or Hypervisor-based virtualization. When requirements change (which is not uncommon in today's world), adapting to a new kernel architecture often requires either redeveloping the kernel or adopting an existing generic kernel, which means losing the benefits derived from previous customization efforts, such as modules verified through formal methods or unified communication protocols.

To address this issue, we aim to quickly migrate custom work across different kernel architectures (i.e., rapidly customize different kernel architectures). In order to achieve this goal, we investigate the similarities and differences between various kernel architectures and attempt to bridge seemingly unrelated kernel architectures, thereby enabling transformation between different kernel forms with minimal cost. We propose the design principle of a flexible kernel, viewing the kernel's shared basic services across different architectures as "kernel libraries." The final form of the kernel is determined by the execution logic after initialization, driven by the specific kernel design. ArceOS is an experimental kernel that validates this flexible kernel design principle. It is a unikernel designed with a modular approach. Currently, ArceOS can quickly construct various kernel forms such as monolithic kernels, unikernels, and hypervisors with minimal code. By reusing the same basic kernel services, we find that the workload and complexity of developing new kernel forms are significantly reduced, offering a new approach to rapidly customizing kernel architectures.

Keywords: componentization; operating system; heterogeneous

目 录

插图清单

附表清单

符号和缩略语说明

POSIX	可移植操作系统接口（The Portable Operating System Interface）
DPDK	数据平面开发工具包（Data Plane Development Kit）
RDMA	远程直接内存访问（Remote Direct Memory Access）
SPDK	存储性能开发工具包（Storage Performance Development Kit）
IPC	进程间通信（Inter-Process Communication）
vCPU	虚拟 CPU（Virtual Central Processing Unit）
Hypervisor	虚拟机监控器
HAL	硬件抽象层（Hardware Abstraction Layer）
qemu	快速仿真器（The Quick Emulator）
ext4	扩展文件系统 4（Fourth Extended File System）
ramfs	内存文件系统（RAM File System）
TLS	线程局部存储（Thread Local Storage）
vfs	虚拟文件系统（Virtual File System）
trait	一种语言概念，表示一组可用于扩展类功能的方法
syscall	系统调用（System Call）

第 1 章　引　言

定制化操作系统是指根据特定的应用场景，定制设计操作系统的结构和功能，从而在性能、安全性等特定方面发挥比通用操作系统更好的表现。定制化操作系统不仅涉及对单个功能组件的优化，还可能对操作系统的整体架构进行创新设计，从而诞生了多种内核架构，如宏内核、微内核、Unikernel、Hypervisor 等。定制操作系统是一个热门且常见的工作，尤其是在云计算、边缘计算、物联网等新兴领域，它可以更好地满足特定应用的需求。

然而，尽管人们在定制化操作系统方面取得了显著的进展，但现有的定制化操作系统往往仅在某个特定的内核架构下进行设计和实现，缺乏跨架构的通用性和灵活性。传统观念认为，不同内核架构之间存在着不可逾越的差异，在一个内核架构上开发的模块或组件无法直接迁移到另一个内核架构上使用。人们通过关注如何去优化某个架构下的内核的表现，但我们希望能够将视野放宽，关注不同内核架构之间的异同点，从而在已有成熟组件的基础上快速构造不同架构的操作系统，将组件的定制特点与内核架构在某个特定场景下的优势结合，从而降低适配新场景的开销。

1.1　研究背景

1.1.1　定制化操作系统

根据某个特定场景，对操作系统进行改造和定制的工作一直是人们关注的热点。通过分析不同的场景，开发者调整内核的架构设计，选择性的强化或者忽略某些部分的功能，从而更好地发挥对应场景下的特点。比如个人设备关注 OS 的通用和安全，物联网设备关注便捷与性能，而云服务器关注性能、资源的利用率和隔离安全。Libra[1]、Arrakis[2]、IX[3]、OSv[4]等是一些定制操作系统的例子，它们在某些应用上拥有着比通用操作系统更佳的表现。

在开发定制操作系统的过程中，人们发现原有的一些内核架构（如单片内核）在某些场景下的表现并不理想，甚至会引入一些不必要的开销，比如在嵌入式设备上有时并不需要引入复杂的调度操作和特权级隔离机制，而对于云服务器来说宏内核的进程隔离机制并不能满足安全容器的需求。因此人们开始考虑拆分和重组内核的结构，提出了许多新的内核架构。下面列举一些经典的内核架构及其适用场景：

- 宏内核（*Monolithic Kernel*）：将所有操作系统服务（如进程管理、内存管理、文件系统等）都放在内核态运行，具有较高的性能和效率，但缺乏模块化和灵活性，典型例子为 Linux。
- 微内核（*Microkernel*）[5]：将最小化的内核功能（如进程管理、内存管理和 IPC）保留在内核态，其余功能则移至用户态运行，提高了系统的模块化和安全性。适用于对可靠性和安全性要求较高的嵌入式系统和安全操作系统。
- *Library OS*[6]：将操作系统服务以库的形式嵌入到应用程序中，使得每个应用拥有独立的运行环境，增强了资源隔离性和可定制性。适用于运行单一任务、需要强隔离和灵活部署的云端服务场景。
- *Unikernel*[7]：将应用和其所需的最小操作系统服务编译为单一镜像，具备极小的攻击面和启动延迟。适合云计算和边缘计算中对启动速度、安全性和资源效率要求极高的环境。
- *Hypervisor*[8]：通过虚拟化技术在底层硬件上运行多个操作系统实例，支持资源隔离与多租户部署。广泛应用于数据中心、云平台和虚拟机管理场景。
- *Hybrid Kernel*[9]：结合微内核与宏内核特点，将性能关键组件保留在内核空间，同时保持一定程度的模块化设计，被许多通用操作系统（如 Windows NT 和 macOS）采用，适用于对性能和兼容性有平衡需求的桌面与服务器系统。
- *Multikernel*[10]：将操作系统设计为分布式系统，内核组件彼此独立运行，彼此通信以协调工作，特别适合在多核或分布式硬件环境下运行，提升可扩展性与并发处理能力。
- *Exokernel*[11]：将传统操作系统的抽象最小化，直接向应用暴露硬件资源访问能力，由用户空间库提供抽象，提升了灵活性和性能。适用于高性能计算或对资源控制精度要求高的研究型系统。

相比于为某个特定应用定制的内核，内核架构的适用场景更加广泛。前文的例子中，Libra 基于 LibraryOS 开发，Arrakis 基于宏内核，而 OSv 则是 Hypervisor 和 Exokernel 的结合实践。可以看出，在合适的内核架构上对应用进行内核定制，能够更好地发挥内核架构的优势。

1.1.2 快速构建操作系统

从零开始构建一个可以用于实际场景的操作系统是高成本且无趣的。无趣是因为绝大部分的工作和已有操作系统的功能重复，创新点可能仅占实际工作量的很小一部分。高成本是指因为操作系统本身的封闭生态和紧耦合性，直接复用其他内核的模块很有可能带来适配问题，需要额外花费时间去调试兼容。为每一个新场景开发一个内核的代价过大。如何快速定制不同功能的内核是一个需要解决

的重要问题。

已有许多工作为快速构建操作系统做出了贡献。ThinkerToy[12] 为物联网设备提供了一组标准操作系统模块，TinyOS[13] 专注于打磨面向传感器网络的操作系统核心组件。这类工作面向某个具体的场景提供了组件化构建操作系统的可能。另外也有许多工作希望提供适合多种场景的通用构造方案。OSKit[14] 使用胶水代码将驱动等各类模块封装为易于调用的组件库的形式，从而简化新操作系统的构建。Anykernel[15] 致力于让成熟的 kernel module 可以运行在任何内核架构上。而 EbbRT[16] 为提供了一组弹性构建块，从而允许不同的应用程序定制适合自己的 LibraryOS。

1.1.3 实验内核 ArceOS 介绍

本文的工作基于组件化操作系统 ArceOS 开发。它由清华大学贾越凯博士主持开发，使用 Rust 语言编写，旨在通过组件化的设计思想，提供一个灵活、可定制的内核，满足不同应用乃至不同场景的需求。项目开源代码可以在 Github 上获取。

图 1.1 展示了 ArceOS 的结构。作为组件化操作系统，它由多层功能组件组成，每层的功能如下：

1. 元件层（ArceOS crates）：由 ArceOS 开发者开发的、与内核无关的组件库。任意内核都可以尝试接入这些组件，而几乎不需要对组件本身的内容进行修改。绝大多数组件均发布到了 crates.io 上，可以便捷地为其他开发者复用。
2. 模块层（ArceOS modules）：与 ArceOS 相关的组件库。它包括了各类内核基本功能组件，如内存管理、调度器、IO 设备驱动等，可认为是传统意义上的“内核”, 用来管理硬件并为应用提供服务。
3. API 层（ArceOS API）：将模块层的功能封装为 API 的形式，以供应用程序调用。ArceOS 为 Rust 和 C 语言编写的应用程序分别提供了不同的 API，并针对语言的特点做了快速路径优化。
4. 用户库层（ArceOS ulib）：进一步封装 API 层提供的功能，提供对 Rust 标准库、libc 库等已有用户库的兼容，以便应用程序的移植。
5. 应用程序层（User Apps）：目前 ArceOS 提供了对 Rust 和 C 这两种语言编写的应用程序源码级别的支持。

另外图中提到的 `axfeat` 模块代表了 ArceOS 的可定制性功能。它本身并没有实际的功能实现，而是利用 Rust 的条件编译机制[17]，完成对 ArceOS 的不同模块的功能定制。通过接收用户传入的 feature 信息，`axfeat` 会将它传递给模块层和元件层的不同模块中，从而实现对整体内核的条件编译，达到 Unikernel 要求的最简化内核镜像的要求。

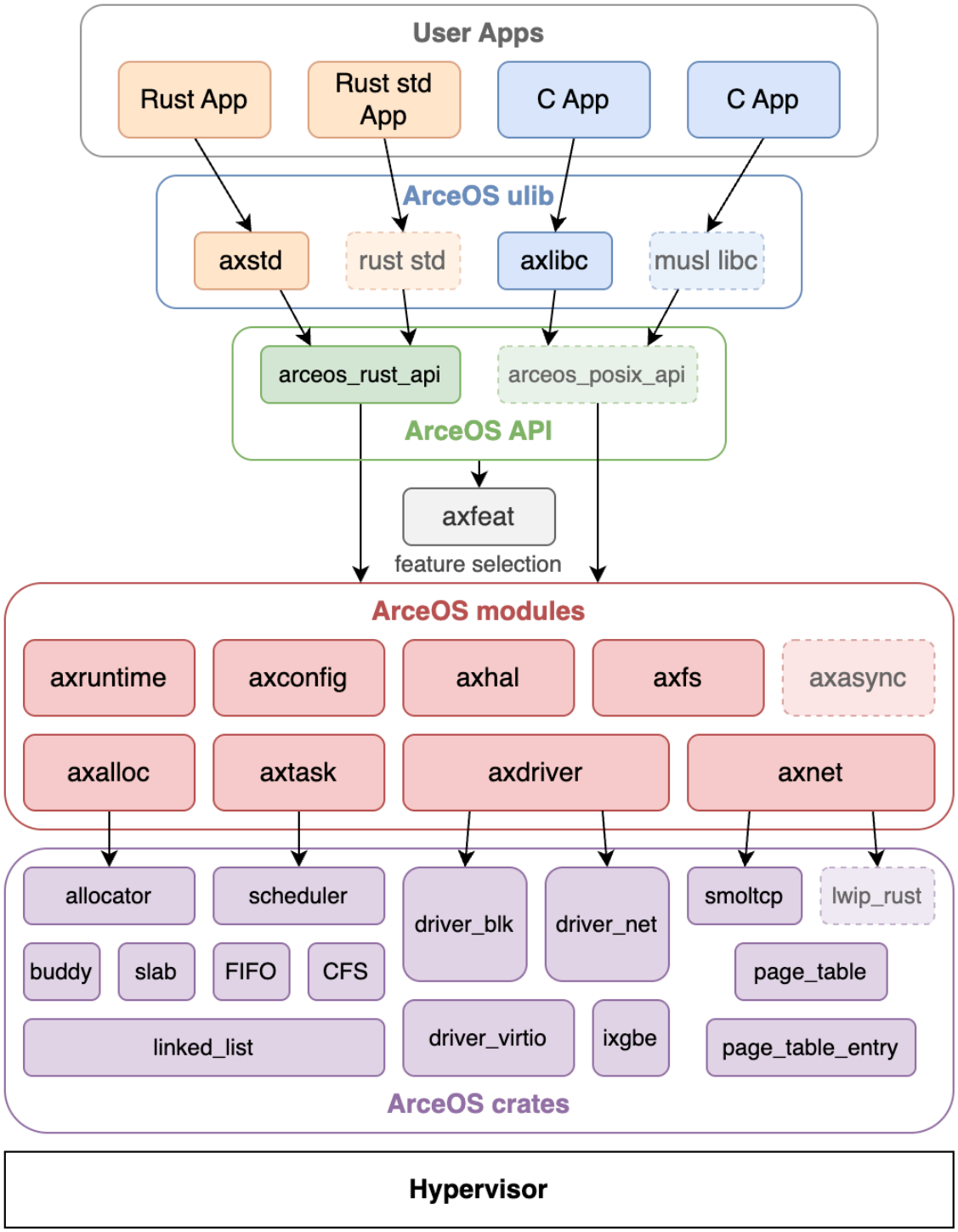

图 1.1　ArceOS 层次结构图

本文基于 ArceOS 实现了一个宏内核扩展应用 Starry[18]。它的前身是笔者在 2023 年 3 月至 8 月期间基于 ArceOS 的早期版本开发的一个宏内核 Starry-Old[19]。它最终获得了 2023 年全国大学生计算机系统能力大赛操作系统内核实现赛道的二等奖。因为是早期版本，Starry-Old 的代码质量较低，且对组件化的思想贯彻地并不彻底。但它仍然是 ArceOS 宏内核扩展的一个重要参考。目前 Starry 的开发也部分参考了 Starry-Old 的设计思路。

1.2 设计动机

1.1节已经介绍了许多定制操作系统的例子。在定制操作系统的发展历程中，许多内核架构被设计出来。而在当前时代下，操作系统的应用场景趋向于异构化，除去 PC、手机等传统应用场景，各种边缘物联网设备也成为了内核的载体，对内核架构的需求也走向多元化[20-21]。运行在 IoT 设备上的边缘节点一般通过 Unikernel 获取快速服务。而手机等中控设备需要微内核、宏内核等架构提供的安全保证。但目前已有的定制内核工作往往是在某一内核架构下面向特定应用程序进行设计。当场景发生改变，如需要特权级的保护或者虚拟化的支持，无法直接将已有的工作迁移到新的内核架构上，而常常需要重新设计开发，从而带来高昂的适配成本。

另外我们常常希望许多成熟的内核模块与其特性可以在新的内核上复用。一个例子是为 seL4 提供 Hypervisor 功能（下称 seL4-hv）[22-23]，在保证自身安全性（由形式化验证确认）的同时提供虚拟化的支持。但这样的适配往往需要与原有的内核架构绑定，如 seL4-hv 仍然需要通过 IPC 来调用系统服务，可能导致通用性或性能的下降[24]。

分析上述情况出现的原因，我们发现尽管对构建某个特定架构的内核的工作已有较多成果，但是对不同内核架构之间的关系的研究却比较少。我们希望能够总结已有的若干个成熟内核形态的异同点，尝试从某个角度将不同的内核架构统一起来，从而以较小的代价实现不同内核架构的转化，最终做到快速定制不同形态的内核，并能将已有的工作迁移到不同内核架构上，从而降低适配新场景的开销。

1.3 本文工作

本文通过对组件化操作系统 ArceOS[25] 的进一步开发，探索使用内核组件快速定制不同内核架构的可能性。具体地，本文为原本为 Unikernel 架构的 ArceOS 实现了宏内核扩展，支持利用 ArceOS 的已有组件快速构建宏内核系统，并且通过

一系列测例和实际应用评估宏内核的完善程度。我们还探索了如何将已有的成熟组件迁移到不同内核架构上，让我们的工作更加具有现实意义。

本文主要分为以下几个部分：

1. 第 2 章介绍使用内核组件快速定制不同形态内核的设计思路。我们提出了灵活内核的设计原则，并介绍了实验内核 ArceOS 的设计背景和实现细节。
2. 第 3 章结合具体代码，介绍了如何在 ArceOS 上快速定制不同形态的内核。我们以宏内核扩展应用 Starry 为例，展示了定制一个新内核形态时的实现思路和细节。
3. 第 4 章对 ArceOS 和 Starry 的实现进行评估。评估包括三个方面：Starry 对 POSIX API 的支持情况、定制新的内核形态的工作量与新的内核模块适配情况，从而展示 Starry 的功能完善性。
4. 第 5 章对本文工作进行总结，并展望本工作在未来的可能应用场景。

第 2 章　快速定制内核形态的设计思路

本章节主要介绍关于快速定制内核架构的设计思路。我们会基于对不同架构的异同点分析，提出适配多种架构的灵活内核的设计原则，并以我们的实际开发内核 ArceOS 为例进行详细介绍。

2.1　灵活内核的设计原则

本节标题的*灵活内核*是我们定义的一种特殊的内核实现。这种内核的特点是 *One Architecture to build all*，即可以使用同一套内核框架快速定制不同内核形态的操作系统，如宏内核、微内核、Hypervisor、Unikernel 等。前文提到，尽管我们对构建某一特定架构的内核已有较多工作成果，但对不同内核架构的关系研究仍然较少。现行的各种流行内核架构多是开发者在适配应用的过程中根据自身经验和需求创新设计出来的方案，而不是某种逻辑推演的结果。寻找他们的异同点、并尝试归纳为某种通用的设计原则是一件不甚直观的工作，也鲜有人进行尝试。为了能够更方便的理解、总结不同内核架构的异同点，我们可以采用*类推*的思维。操作系统本质也是一个软件，只是其功能相比一般的应用程序来说复杂得多。因此我们可以先针对我们熟悉的用户态程序进行分析。

对于不同的用户程序来说：

1. 共同点：它们绝大多数都依赖于一些基础服务库，即用户所需的共同服务，如标准 IO、数学运算、字符串处理等。
2. 不同点：用户程序根据自定义的执行流（在 C 语言下一般由 `main` 函数开始）决定所引用的具体库、实现逻辑和对外提供的服务，如图形界面、网络服务、数据库等。

我们可以将用户程序的设计思路类比到操作系统的设计上来。在近些年来，由于技术的发展，部分应用的性能瓶颈不再是硬件，而逐渐转移到内核因为繁杂、通用的设计带来的不必要开销上。在网络、内存 IO 方面该现象尤为明显。许多用户态框架的出现（如 DPDK、RDMA、SPDK）都是为了绕过内核的通用设计，直接与硬件进行交互，从而提升性能。因此近些年也有技术人员预测：操作系统将逐渐为内核旁路替代，而逐渐消亡。

但操作系统最重要的工作应当是*为用户提供更高级别的抽象*。所谓的抽象即是内核对给用户提供的服务的包装。对不同架构的操作系统来说，共同点就是*内核所提供的基础服务，即内核的执行环境*，包括硬件交互、驱动控制、资源管理等

内容。关于这部分的具体说明详见 3.2 节。而当内核被设计为不同的架构，本质上内核提供的服务内容并没有改变，其区别主要体现在以下两方面：

1. 抽象的封装程度高低：操作系统一般会屏蔽底层的服务实现细节，而通过某些规定好的接口（如 POSIX API、IPC、standard library）与用户程序进行交互。在这种情况下，内核会为用户提供一个尽可能高级的抽象。但对于 Exokernel 等架构，它们希望能够将硬件的细节暴露给用户程序，从而让用户程序自己决定如何使用硬件资源，而内核仅做一些必要的安全检查和低级抽象。
2. 调用内核服务的方式：为了适配场景的需求，不同内核架构调用服务的方式可能有所区别，如宏内核的系统调用、微内核的 IPC、Library OS 的函数调用、Hypervisor 的 VM exit、Multikernel 的分布式请求等。开发者通过封装不同的调用、检查方式，从而满足不同场景下关于安全性、性能、便捷性等方面的需求。

其中主流的内核架构主要是在调用内核服务的方式上有所区别。我们选取了四种较为常见的内核架构，分别是宏内核、微内核、Unikernel 和 Hypervisor，就其关于调用内核服务的方式的区别进行总结，如表 2.1 所示。

表 2.1　不同内核架构对比

	Unikernel	Monolithic Kernel	MicroKernel	Hypervisor
Privilege Mode	Privileged	Privileged + User	Privileged + User	Host privilege + Guest
Address Space	Single	Multiple	Multiple	Host + Multiple Guest
Scheduling Unit	Thread	Thread	Service with thread	vCPU
Resource Ownership	Global	Per-Process	Per-Service	Per-VM
Service Interaction	Function Call	System Call	Message Passing	Hypercall/VM exit

从表 2.1 可以看出，不同的内核架构有着各自的特征功能，如宏内核的 POSIX API 原生实现、进程管理等，微内核的 IPC 通信机制，Hypervisor 的 vCPU 模型、guest Memory 管理，Unikernel 的单进程模型、全局共享资源等。但这些不同的形态也存在着共性的功能。它们都需要基本的 CPU 特权态执行环境，需要基本的操作系统服务，包括引导启动内核、与底层硬件进行交互、进行任务调度等。

进一步总结可以发现，这些共性恰好可以由最简内核形态：*Unikernel* 提供。这个现象的产生并不是没有原因的。Unikernel 在设计时就考虑了尽可能减少复杂性，将内核的功能限制在最小的范围内，因此它便可以成为其他内核形态的基础。因此我们提出了灵活内核的设计原则：*Unikernel* 是各种内核形态的交集。我们将可定制的 *Unikernel* 作为内核的基本框架和执行环境，即 *Kernel Backbone*，而将各种内核形态的特性实现视为 *Unikernel* 上运行的定制 *APP*，即 *Kernel Plugin*。图 2.1

展示了这一原则的大体结构。

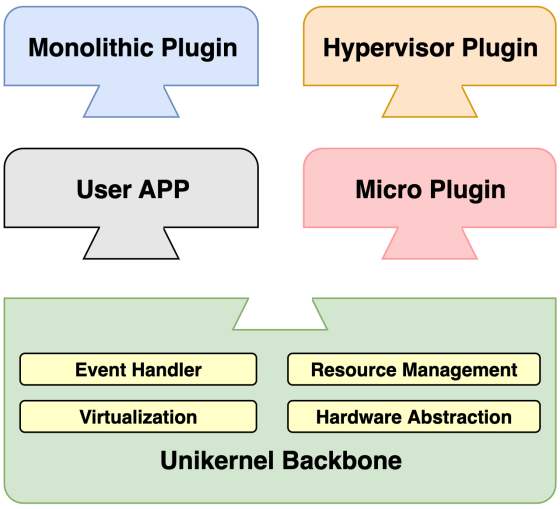

图 2.1　灵活内核架构的设计原则

为了简便称呼，在本文中我们将与 Unikernel 链接的具体应用程序称为内核应用。它和我们常见的用户编写、运行在通用内核上的应用程序的区别在于：内核应用是与内核紧密耦合的，运行在内核态的应用程序。它们可以直接调用内核提供的服务，而不需要通过系统调用的方式进行交互。

接下来我们以宏内核的构建为例讲解我们的灵活内核设计原则。宏内核的特性功能包括完善的进程管理、信号处理、POSIX API 实现等各种功能。这些特性功能将实现在 Monolithic plugin 中，并作为 Unikernel 上定制的内核应用（运行在内核态）一起编译运行。当底层的 Unikernel 完成了内核功能的初始化，便会将执行流交给 Monolithic plugin，运行宏内核的相关功能，如加载用户应用、接收 syscall 并进处理。而这个过程中调用的内核服务由 Unikernel 和 Monolithic Plugin 共同提供，于是便可以以较小的工作量快速构建出一个宏内核。

灵活内核的设计原则要求不同架构复用同一套内核基础服务框架，这对内核组件的功能划分和可复用性提出了较高的要求。因此我们在 2.2 节为内核组件的概念做了进一步的划分，从而保证我们构造的内核组件具有良好的复用性和可扩展性。

2.2　组件的层次划分

使用组件化构建内核并不是一个新说法，许多内核都在声称自己采用组件化或模块化的方式进行构造，但现有工作通常仅仅按照功能对组件进行划分，并将不同组件简单聚合在一起作为组件库向外提供。

但事实上组件之间应当是存在依赖关系的，如地址空间管理的功能模块应当依赖于页表实现，即上层的地址空间管理模块引用了下层的页表模块。当我们想要为地址空间管理模块添加新的页表实现时，一个理想的情况是为下层的页表模块寻找替代实现，而无需影响上层的地址空间管理模块。现有的组件化内核大多没有考虑到这一点，导致了组件之间的耦合性过高，无法灵活地进行替换和扩展。

为了组合不同组件定制具有多样形态和功能的内核，我们进一步分析、总结了组件在不同内核形态下的共性与差异，并依照*底层复用、上层定制*的原则划分组件所属的层次。

1. crates layer：该类组件的实现不需要依赖于具体接入的 OS，在用户态、内核态均可以运行。*任意内核*都可以尝试接入这些组件，而几乎不需要对组件本身的内容进行修改。这也是 OSkit 等工作的组件库提供的组件的主要特点。是否属于 OS 无关组件与组件的粒度无关，页表等数据结构和网卡驱动等功能都属于 OS 无关组件。
2. backbones layer: 该类组件的实现与具体接入的 OS 相关，一般涉及到内核的核心功能，很难从一个内核中剥离并给其他内核使用。一个例子是任务调度队列。调度算法如 CFS、FIFO 早已作为组件被其他内核广泛复用，但想要将 Linux 的调度队列抽象出来给其他内核使用基本是不可能的，因为它早已与 Linux 的许多部分融紧耦合，如中断、系统调用等。但这类组件在重构内核形态时，可以在几乎不做修改的情况下仍然发挥着重要的作用。比如 KVM 仍然借用了 Linux 的调度队列进行虚拟机任务的调度。
3. plugins layer: 这一层代表的是在不同的内核架构特有的 OS 相关组件，如宏内核的 POSIX API 兼容层、微内核的 IPC 实现层等，位于组件层次的最上层。需要注意的是，这类组件并不一定是某个形态特有的，例如 IPC 虽然是微内核的基本功能，但也可能存在于特定需求下的宏内核中。这类组件的功能实现一般都需要内核基本服务的支持，而这已经在 backbones layer 提供。

我们将组件划分为三个层次，从底向上分别是*内核无关、内核相关、架构相关*。这种划分方式不仅关注组件的功能，还关注功能在不同内核形态上的表现。当尝试在上层构建新的内核功能和形态的时候，可以复用下层的组件，从而降低定制的工作量。

2.3 基于 ArceOS 的设计目标

ArceOS 本体是一个 Unikernel，按照灵活内核设计原则，我们希望*通过定制 Plugin 可以实现 ArceOS 对不同内核架构的扩展*。本文章的工作主要聚焦于 ArceOS

的宏内核扩展实现，另外也有工程师参与实现了 ArceOS 在 Hypervisor 上的扩展，从多方面验证了灵活内核架构设计原则的有效性。

本文将基于 ArceOS 与其宏内核扩展的工作，解答如下问题：

1. 哪些组件应当作为内核的基础服务，被不同架构所复用？
2. 如何理解 Kernel Plugin（内核架构扩展应用）的设计思路；它是如何实现将 Unikenrel 的处理逻辑切换到一个新的内核架构处理逻辑的？
3. 如何以较小的难度和工作量开发、完善 Kernel Plugin 的功能？

上述问题是我们在设计灵活内核架构时所考虑的主要内容，而能够为这些问题找到合适的答案即是我们的设计目标。我们希望通过对 ArceOS 的宏内核扩展的实现，能够为读者提供一个清晰的实现流程，帮助读者更加深入地理解灵活内核架构的设计思路。

第 3 章　ArceOS 的异构形态实现

本章节将具体介绍 ArceOS 与其上的不同内核架构扩展的实现细节，从而解答在 2.3 节中提出的问题，展现灵活内核架构的设计思路。我们将以宏内核扩展为例，对 ArceOS 的异构形态实现进行介绍。

3.1　以复用的原则构建不同形态内核

为了同时在 ArceOS 上实现不同的内核架构，一个直观的想法是直接对 ArceOS 的内容进行*侵入式修改*，将宏内核的功能直接嵌入到 ArceOS 的各层代码中。这样做的好处是可以快速实现宏内核的功能，而不需要考虑对其他架构的兼容。事实上，Starry-Old 就是采用这种方法实现的。因为当时它作为参赛作品，实现时间较为紧张。在笔者代码能力有限的情况下，直接对 ArceOS 的代码进行修改是最简单的实现方式。

但这种方法的缺点也很明显：它无法实现内核的基础服务对其他内核架构的兼容性。灵活内核架构的设计原则希望我们能够在不同内核架构之间共享同一套基础服务框架，通过复用代表内核核心功能的大部分组件，从而降低开发新的内核架构的工作量。但侵入式修改无法保证对其他内核架构的复用。一个例子是任务调度机制，对于宏内核来说我们需要在切换到新任务的同时切换页表基址（必要情况下），但 Unikernel 和 Hypervisor 架构下并没有这个需求。如果强行将这个功能加入到 ArceOS 模块中，会导致给 Unikernel 和 Hypervisor 架构引入不必要的内容和检查，从而导致代码的冗余。

因此，我们希望能在进行多内核架构扩展的时候尽可能复用已有的内核功能。我们将基于*求同存异*的态度，从复用的视角看待灵活内核架构的设计。

3.1.1　执行环境复用

灵活内核的设计原则指出：Unikernel 是不同内核形态的交集，也是它们共有的执行环境。对于 ArceOS 来说，它的本体即是一个 Unikenrel，因此可以将 ArceOS 提供的内核基础服务（即元件层和模块层）作为不同内核架构共享的执行环境，即 Kernel Backbone。而不同内核形态的具体实现是 Unikernel 上的定制 App，对应的即是 ArceOS 的应用程序层。因此我们按照 2.2 节重新划分 ArceOS 的架构设计，如图 3.1 所示。

分层设计中，Backbone Layer 对应 ArceOS 的模块层，代表内核提供的基础服

务，可以为不同的内核架构复用。而 Plugins Layer 属于 ArceOS 的应用程序层，代表不同内核架构的特性实现。每一个内核架构对应一个 Kernel Plugin，它们共享着 Backbone Layer 提供的服务，在复用执行环境的基础上降低新内核架构的工作量。

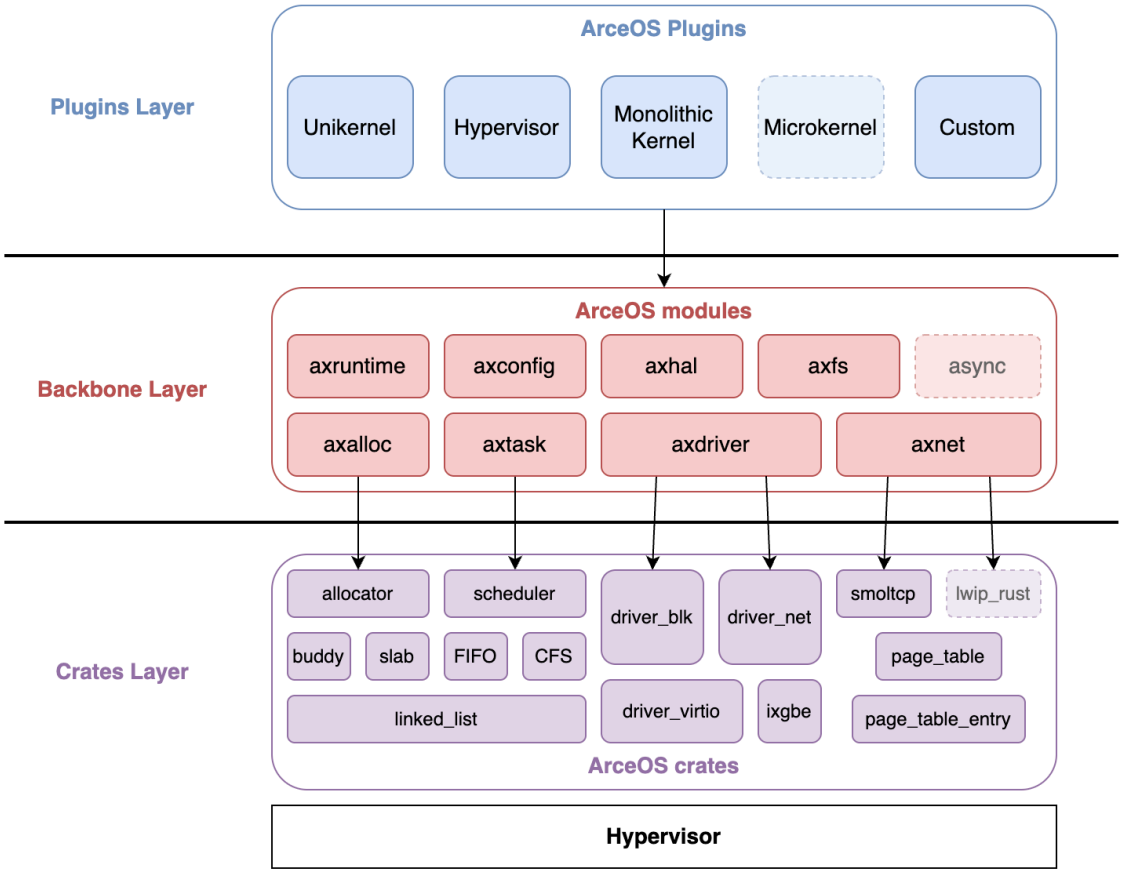

图 3.1　ArceOS 层次结构图

我们将以宏内核扩展的实现进行详细介绍。宏内核的特性功能包括完善的进程管理、信号处理、POSIX API 实现等各种功能。这些特性功能将实现在宏内核扩展中，并作为 ArceOS 上定制的内核应用（运行在特权级）一起编译运行。图 3.2 介绍了宏内核扩展的执行流。具体流程拆分如下：

1. ArceOS Backbone 进行内核功能的初始化①，并将执行流交给内核应用（此处为宏内核扩展）②。
2. 宏内核扩展运行宏内核的相关功能，如加载用户应用③、管理地址空间④并为任务准备初始化上下文⑤。当准备工作完成，便会将执行流交给用户应用，返回到用户态执行用户程序的代码⑥。
3. 当用户应用需要进行系统调用时，会通过 syscall 进入内核态⑦。此时 ArceOS Backbone 的硬件交互模块会捕获异常内容，发现是需要宏内核处理的 syscall，便会转发给宏内核扩展进行处理⑧。

4. 宏内核扩展处理完毕之后，会将结果传给 ArceOS Backbone，检查无误之后通过特权级切换回到用户态⑨，继续执行用户应用的代码。

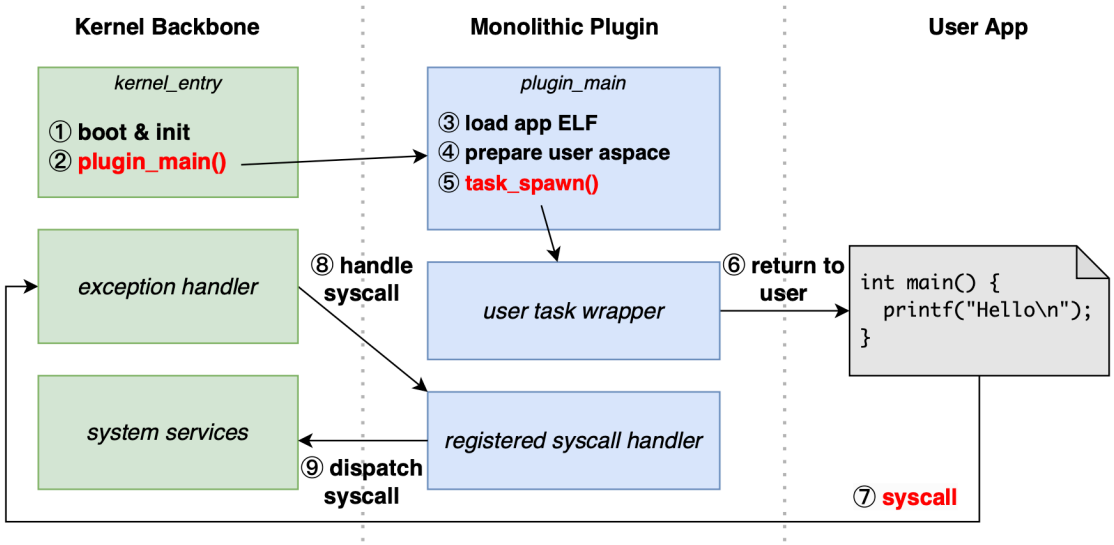

图 3.2　宏内核扩展的执行流

上述执行流展示了如何在 Unikernel 的执行环境上实现一个宏内核。除去 Starry 之外，ArceOS 的 Hypervisor 扩展实现 AxVisor[26] 也采用了类似的逻辑，即在完成了内核初始化之后将执行流交给 Hypervisor 扩展进行处理，并由 Hypervisor 扩展处理用户的特殊请求，如 VM Exit 等。

通过分析执行流可知，在理想情况下，不同内核架构的特性实现基本都位于 Plugins Layer，不需要对 Backbone 做侵入式修改，也可以完成对内核基础服务组件的共享，从而能以较小工作量实现新的内核架构。

但实际情况并没有这么顺利，不同内核架构之间在资源管理、运行模式等多个方面存在区别，我们需要判断这些区别是否需要由 Backbone 来负责，并设计合适的机制来保证不同架构的兼容。我们在表 2.1 中展示了不同内核架构的区别。接下来的两节我们会介绍 Backbone 中为了解决内核架构兼容性而采用的机制。

3.1.2　调度功能复用：Module Extension 机制

Backbone 的共性抽象并不是完全理想的，它无法同时满足各种形态的内核的需求。一个例子是任务调度功能。不同形态内核都需要有任务调度的机制，因此它应当被实现在 Backbone 中。但每一个内核形态的任务调度单元可能包含不同的成员和实现 (见表 2.1)，比如微内核的任务可能包含 IPC 的相关信息，Hypervisor 的 vCPU task 可能包含 guest OS 的相关信息，而宏内核的任务包含了实现 POSIX API 的相关信息，如 CPU 亲和度、调度优先级等。这就导致了 Backbone 中的任务

调度单元需要有一定的扩展性，以适应不同形态内核的需求。

伪代码 1 Module Extension for Task Scheduling

```
1  // Base task in Unikernel backbone
2  struct Task {
3    id: usize,
4    state: AtomicU8,
5    /// other fields
6    ...
7    /// task extended data
8    task_ext_ptr: *mut u8,
9  }
10
11 // Define task extended data for monolithic kernel
12 struct MonolithicTaskExt {
13   proc_id: usize, // process ID
14   uctx: UspaceContext, // user space context
15   aspace: AddrSpace, // virtual memory address space
16 }
17 // Call `def_task_ext!` to define task extended data
18 def_task_ext!(MonolithicTaskExt);
19
20 // Usage in monolithic kernel plugin
21 fn spawn(entry) {
22   let new_task = Task::new(entry);
23   // init task extended data
24   new_task.init_task_ext(
25     MonolithicTaskExt { proc_id, uctx, aspace }
26   );
27   Task::spawn_task(task)
28 }
29 fn mmap(addr, len, prot, flags) {
30   // use task extended data
31   Task::current_task().task_ext().aspace.mmap(addr, len, prot, flags)
32 }
```

为了实现这种扩展性，我们设计了 Module Extension 机制。以任务调度单元为例，伪代码 1 展示了 Backbone 和宏内核 Plugin 中对任务调度单元的实现。我们先在 Backbone 中定义了 Unikernel 的调度单元结构 (lines 2 - 9)，并为其设置了一个 *task_ext_ptr*(line 8)，作为指向扩展区域的指针。当我们仅使用 Backbone 运行 Unikernel 功能时，无需使用这个扩展区域，自然也不需要对其进行初始化。如果引入了宏内核 Plugin 时，我们可以在上层定义了宏内核所需的信息 (lines 12 - 16)，并使用 *def_task_ext!* 宏将其定义为 task 的扩展数据。之后我们可以便可以在 Plugin 中使用这些特性信息，如在 mmap 函数中使用宏内核特有的 aspace 字段（来管理进程的地址空间）(line 29)。这样，我们就扩展了 Backbone 功能，可以适应宏内核形态的需求。对于 Hypervisor、微内核等其他架构，也可以利用 Module Extension 机制，实现自己的特性扩展。

需要注意的是，代码 1 被称为伪代码是因为它仅用于解释 Module Extension 机制的使用方式和宏内核示例。真实的 Starry 主线为了通过编译或者适应模块化等

需求，具体实现代码可能与示例略有差距，但它对 Module Extension 机制的使用方式是类似的。

代码 2 描述了调用了这一宏之后的具体展开代码。当调用了 extended data 的初始化函数 (line 24)，我们会在堆上为扩展数据申请一段空间，并让 *task_ext_ptr* 指针指向这段区域。这段区域会随着任务本身参与到调度中，从而达到了扩展调度单元内容的目的。利用 Module Extension 机制，我们可以让 Kernel Plugin 自行注册特性信息和功能，而不会影响 Backbone 的简洁性。

代码 2 Expansion of *def_task_ext!(MonolithicTaskExt)*

```
1  static __TASK_EXT_SIZE: usize =
2    size_of::<MonolithicTaskExt>();
3  static __TASK_EXT_ALIGN: usize =
4    align_of::<MonolithicTaskExt>();
5
6  impl TaskExtRef<MonolithicTaskExt> for Task {
7    fn task_ext(&self) -> &MonolithicTaskExt {
8      unsafe { &*self.task_ext_ptr
9        as *const MonolithicTaskExt }
10   }
11 }
12
13 pub trait TaskExtRef<T: Sized> {
14   fn task_ext(&self) -> &T;
15 }
16
17 impl Task {
18   fn init_task_ext<T: Sized>(&mut self, data: T) {
19     self.task_ext_ptr =
20       alloc(__TASK_EXT_SIZE, __TASK_EXT_ALIGN);
21     self.task_ext_ptr.write(data);
22   }
23 }
```

3.1.3 资源实现逻辑复用：Namespace 机制

Namespace 机制提出的背景是不同内核架构下资源的所有权管理问题。我们希望能灵活地控制资源的隔离和共享情况，从而做到复用同一套资源的实现、管理逻辑，为不同的架构提供服务。

以文件描述符而言，对于 Unikernel 来说，由于是单地址空间，该资源应当是全局共享，所有任务共享同一个文件描述符表。但对于宏内核等架构来说，不同的进程、服务或者虚拟机之间存在隔离，需要控制文件描述符表针对进程等单位的隔离情况。尽管文件描述符表可能因为隔离而不是同一个，但对文件描述符的操作、管理逻辑却是相同的。表现在开发上即编写的控制代码一致，而操作的数据不一定一致。为了尽可能复用资源的实现逻辑，我们需要解决系统资源的共享问题。

代码 3 ResArc 结构体

```rust
pub struct ResArc<T>(LazyInit<Arc<T>>);
impl<T> ResArc<T> {
    /// Creates a new uninitialized resource.
    pub const fn new() -> Self {
        Self(LazyInit::new())
    }
    /// Returns a shared reference to the resource.
    pub fn share(&self) -> Arc<T> {
        self.0.deref().clone()
    }
    /// Initializes the resource and doesn't share with others.
    pub fn init_new(&self, data: T) {
        self.0.init_once(Arc::new(data));
    }
    /// Initializes the resource with the shared data.
    pub fn init_shared(&self, data: Arc<T>) {
        self.0.init_once(data);
    }
}
```

关于资源共享的设计目标包括以下几个内容：

1. 允许资源分散式定义在各个组件中：由于组件化的设计，内核功能会分散实现在不同组件中，包括文件描述符表、文件系统元数据（如工作目录）等。我们应当允许这些资源分散定义在各个组件中，从而满足组件化设计的要求。
2. 尽量不修改资源的实现、管理代码：我们希望能够尽量复用已有的资源逻辑，仅通过静态配置或动态控制改变实际操作的数据，从而达到调用 global 或者 per-task 的内核资源。
3. 任务间细粒度资源共享：对资源的隔离情况并不是一刀切的，在宏内核架构中可以控制某些进程所拥有的资源彼此隔离，也可以控制资源在某些进程中共享（如 `sys_clone` 系统调用中通过 `CLONE_VM` 共享地址空间，`CLONE_FILE` 共享文件描述符信息等）。我们希望能灵活控制每种资源的隔离和共享情况。

为了方便地控制资源的共享，我们设计了 `ResArc` 结构，用于定义一种类型的资源。代码 3 展示了 `ResArc` 的实现。通过使用 Arc 指针，我们可以为这个类型的资源分配新的独占数据，或者和其他任务进行共享。

在 `ResArc` 结构的基础上，我们设计了 `AxNamespace` 结构，用于定义每一个任务的资源空间，存储其所拥有的资源。对应到代码实现上即是在 `AxNamespace` 中保存所有的 `ResArc`，其中可能部分 `ResArc` 是独占资源，而其他的 `ResArc` 对应的数据与其他任务共享，从而实现了细粒度控制资源隔离情况的目标。

图 3.3 展示了 Namespace 机制在 Unikernel 架构和宏内核架构下的实现和区别。对于 Unikernel 来说，不存在控制资源共享情况的必要性，全局共享同一份资源，

因此仅有一个全局的 Namespace，存储着各类资源数据（自然也是全局共享），所有 Task 直接访问全局 Namespace 获得相关数据。而对于宏内核架构来说，我们需要为每一个进程、服务或者虚拟机分配一个 Namespace，存储它们所拥有的资源。可以按照单个资源的粒度控制 Namespace 的数据共享情况。在图 3.3 的宏内核部分，每一个 Task 拥有自己的 Namespace。Task 3 与 Task 1（或者 Task 2）属于不同的进程，因此两者的 Namespace 中所拥有的资源绝大部分是独占而不共享的（除非通过 `sys_clone` 的参数明确指定共享数据）。而对于 Task 1 和 Task 2 而言，两者均属于同一进程 Process 1，文件描述符表、地址空间等资源默认情况下会进行共享。通过 `ResArc` 和 `AxNamespace` 机制，即可细粒度地控制资源的隔离和共享。

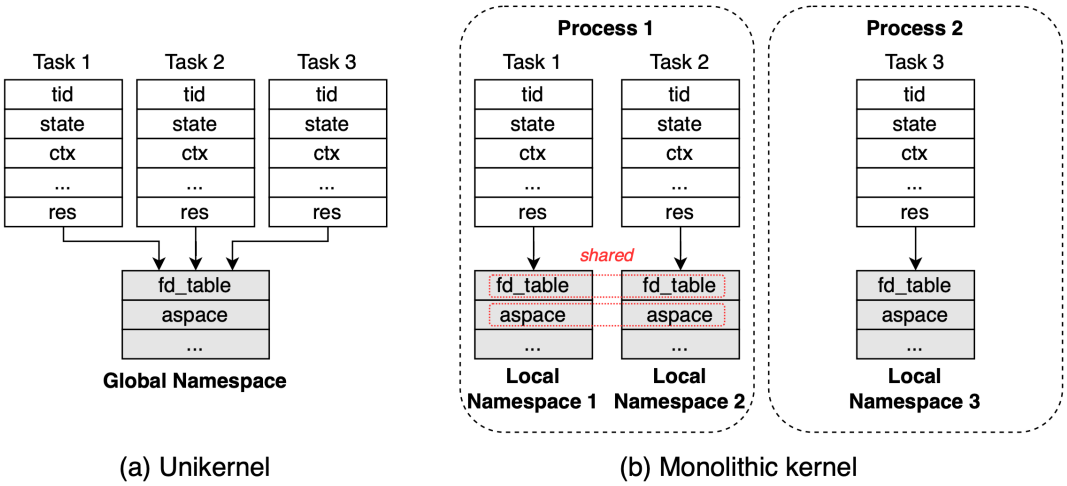

图 3.3　Namespace 机制示意图

接下来以文件描述符表为例，介绍 `Namespace` 的使用方式。代码 4 展示了 ArceOS 原先的资源定义方式。因为 ArceOS 是 Unikernel 架构，其资源均是全局共享，在没有引入 Namespace 机制的时候，文件描述符表是一个全局的静态变量，所有的任务都可以直接访问它。

代码 4 ArceOS 资源原定义（面向 Unikernel）

```
1 static FD_TABLE: FileTable<Arc<dyn FileLike>> = FileTable::new();
2
3 FD_TABLE.get(STDOUT).write(buf);
```

当引入了 Namespace 机制之后，我们可以将文件描述符的资源定义方式改写为代码 5 的形式。其中 `def_resource!` 宏用于定义 `ResArc` 资源的名称和类型。在编译时，我们会将所有通过 `def_resource!` 宏定义的资源进行收集，并

排布在 Namespace 中，得到一个确定的 Namespace 布局。在这种机制下，对资源的使用代码并不需要发生变化，但实际操作的数据已经通过 ResArc 进行了封装和区分。

代码 5 ArceOS 资源定义（兼容不同架构）

```
def_resource! {
static FD_TABLE: ResArc<FileTable<Arc<dyn FileLike>>> = ResArc::new();
}

// 使用资源的代码并未改变
FD_TABLE.get(STDOUT).write(buf);
```

代码 6 宏内核架构下对 Namespace 资源的使用示例

```
struct MonolithicTaskExt {
  ns: AxNamespace,
}
fn sys_clone(clone_flags, newsp, parent_tid, child_tid, tid) {
  let new_task = AxTask::new(entry);
  let ns = AxNamespace::new_local(); // 为每个任务分配 namespace 内存
  new_task.init_task_ext(MonolithicTaskExt { ns });
  if clone_flags.contains(CloneFlags::CLONE_FILES) {
    // 与当前任务共享文件描述符表
    FD_TABLE.deref_from(&ns).init_shared(FD_TABLE.share());
  } else {
    // 使用新的文件描述符表
    FD_TABLE.deref_from(&ns).init_new(FD_TABLE.copy());
  }
  // 是否共享地址空间
  if clone_flags.contains(CloneFlags::CLONE_VM) { ... } else { ... }
  axtask::spawn_task(task)
}
```

代码 5 展示了如何在编译期确定 Namespace 中的 ResArc 类型。接下来我们将以 sys_clone 系统调用为例，展示如何在运行时确定 Namespace 中的 ResArc 数据。代码 6 展示了宏内核架构下对 Namespace 资源的使用方式。我们在 sys_clone 系统调用中创建了一个新的任务，并为其分配了一个新的 Namespace。之后根据 clone_flags 等参数，决定是否共享文件描述符表和地址空间等资源。通过这种方式，我们可以自由确定资源的独占或共享情况。

接下来我们介绍 Namespace 的实现细节，即它如何在不修改资源访问代码的情况下，实现对不同任务的 ResArc 的访问。我们以代码 5 中对资源的定义为例。代码 7 描述了宏展开后的代码。对 ResArc 的访问关键在于 core::ops::Deref 的实现。当完成了代码 5 中对各种资源的定义之后，在 ResArc 的 deref 函数中，会定义一个静态变量 RES，类型为 ResArc 并将其链接到 axns_resource 段中。这个静态变量的地址会在编译时确定，并且在运行时不会发生变化。因此当收集了所有的资源定义之后，axns_resource 段即是确定的 Namespace 的

代码 7 Namespace 机制实现原理

```rust
// Expansion of macro def_resource!
struct FD_TABLE { __value: () }
impl core::ops::Deref for FD_TABLE {
  type Target = ResArc<FileTable>;
  pub fn deref(&self) -> &Self::Target {
    #[link_section = "axns_resource"]
    static RES: Self::Target = ResArc::new();
    let offset =
      &RES as *const _ as usize - __start_axns_resource as usize;
    &*(current_namespace_base().add(offset) as *const _)
  }
}

pub fn current_namespace_base() -> *mut u8 {
  #[cfg(feature = "monolithic")] // current().task_ext().ns.base()
  { crate_interface::call_interface!(
      AxNamespaceIf::current_namespace_base
    ) }
  #[cfg(not(feature = "monolithic"))]
  { AxNamespace::global().base() }
}

// Defined in Monolithic kernel plugin
struct AxNamespaceImpl;
#[crate_interface::impl_interface]
impl AxNamespaceIf for AxNamespaceImpl {
  fn current_namespace_base() -> *mut u8 {
    // All kernel tasks should share the same namespace.
    static KERNEL_NS_BASE: Once<usize> = Once::new();
    let current = axtask::current();
    // Safety: We only check whether the task extended field is null.
    // And we do not access it.
    if unsafe { current.task_ext_ptr() }.is_null() {
      return *(KERNEL_NS_BASE.call_once(|| {
        let global_ns = AxNamespace::global();
        let layout = Layout::from_size_align(
          global_ns.size(), 64
        ).unwrap();
        // Safety: The global namespace is a static readonly variable.
        // And it will not be dropped.
        let dst = unsafe { alloc::alloc::alloc(layout) };
        let src = global_ns.base();
        unsafe {
          core::ptr::copy_nonoverlapping(src, dst, global_ns.size())
        };
        dst as usize
      })) as *mut u8;
    }
    current.task_ext().ns.base()
  }
}
```

布局，且每一个 ResArc 对应在 axns_resource 段的偏移量是确定的（即静态变量 ResArc 相对 axns_resource 段首的偏移）。在运行时，我们可以通过 current_namespace_base 函数获得当前任务的 Namespace 基址，并通过 ResArc 的偏移量计算出当前任务的 Namespace 中的 ResArc 的地址。这样，我

们便可以在不修改资源访问代码的情况下，完成对不同任务的资源访问。

代码 7 中第 23～51 行是宏内核扩展中实现的 `current_namespace_base` 函数。它会根据当前任务的 `task_ext_ptr` 是否为空，判断其是否拥有自己的 Namespace。

1. 如果当前任务的 `task_ext_ptr` 为空，说明当前任务是内核态任务（如 idle、gc 任务等），则它们共享一个内核态的 Namespace。我们会在第一次访问该 Namespace 时分配一段内存，并将其作为内核态 Namespace 的基址。之后的访问均使用这段内存。
2. 如果当前任务的 `task_ext_ptr` 不为空，则返回当前任务的 Namespace 的基址（在创建任务时分配，详见代码 6 的第 6～7 行）。

3.1.4 小结

本章节基于*求同存异*的态度，从复用的视角尝试构造兼容不同架构的操作系统。我们通过共享内核的执行环境，复用了绝大部分的内核服务组件，从而降低了新架构开发的成本。另外关于表 2.1 中展示了不同内核架构的区别，我们分别采用如下方式进行解决：

1. 运行特权级：由 Kernel Plugin 控制实际运行的特权级，如宏内核选择加载应用程序并跳转到用户态。
2. 地址空间管理：由 Backbone 提供通用的方法，如分配页表、映射页表等，而 Kernel Plugin 实现具体的管理机制，如宏内核关于进程的地址空间管理、Hypervisor 关于 vCPU 的地址空间管理等。
3. 调度单元：不同内核架构在调度时需要保存的信息不同，如宏内核的页表基址信息、Hypervisor 的 vCPU 信息等。我们通过设计了 Module Extension 机制（3.1.2 节）来解决这个问题。Extension 机制允许在 Kernel Plugin 为 Backbone 的结构注册额外的信息，从而满足不同内核架构的需求。
4. 资源所有权：内核需要管理资源的共享情况。对于 Unikernel 来说，绝大部分的内核资源需要全局共享。但对宏内核、微内核等架构来说，需要按照进程等单位进行资源等隔离和共享。为了兼容不同架构的区别，我们设计了 Namespace 管理机制（3.1.3 节），在保证资源的实现和管理逻辑不变的情况下，做到灵活控制资源的隔离和共享。

因此，通过将特性实现移动到 Kernel Plugin，或者为它们实现新的机制，我们可以在保持 Backbone 对不同内核架构兼容性的情况下，实现上层的扩展应用。

3.2 ArceOS Backbone 的组成

灵活内核架构的核心是 Backbone Layer，即内核提供的基础服务。那么这一层具体应该包含哪些内容，又是如何划分的呢？依据我们的内核开发经验，对多数内核来说，它们可以总结为如下方面：

1. 驱动控制：兼容适配不同来源的硬件驱动，从而控制对应的设备。
2. 硬件交互：与不同的运行平台的硬件细节进行交互。对内核来说，向下需要适配不同硬件平台，并调用它们所提供的功能支持，如时钟读取、异常捕获等；向上需要屏蔽不同的硬件平台区别，并为它们提供相应的操作硬件的通用接口，如使能中断、读写寄存器等。
3. 资源管理：操作系统管理的资源包括硬件资源（如物理页、内存、IO 设备等）和软件资源（如进程、线程、信号量等）。内核需要对这些资源进行分配、回收和调度，为用户程序提供良好的抽象和稳定的服务。

对应到 ArceOS 的设计中，即是其模块层的内容。对照图 3.1，ArceOS 模块层由如下模块组成：

1. axalloc：实例化内存分配器，负责物理页的分配和回收，支持 Buddy 分配算法、Slab 分配算法等。
2. axhal：硬件抽象层，提供了对硬件平台的抽象和适配，负责 boot 内核、时钟、异常等硬件平台的细节处理，并为上层提供统一的硬件功能接口。
3. axdriver：驱动管理模块，负责接入不同来源的设备，并为用户提供统一的驱动功能调用接口，如块设备、网络设备和显示设备。
4. axnet：网络兼容模块，通过调用网络设备的功能，来实现网络协议栈对内核的兼容，并为上层提供统一的网络功能调用接口。
5. axfs：文件系统模块，负责接入不同来源的文件系统，并为用户提供统一的文件系统功能调用接口，如文件读写、目录操作等。
6. axtask：任务管理模块，定义内核的调度单元结构，并且实现了任务调度的相关功能，如任务创建、任务切换、任务唤醒等。它是内核的核心模块，负责管理和调度所有的任务。
7. axsync：同步原语模块，它基于 axtask 模块，提供了对内核的同步原语的支持，如信号量、互斥锁、读写锁等。
8. axns：Namespace 机制实现，具体的原理参考 3.1.3 节，控制资源的隔离和共享。
9. axruntime：运行时管理。在内核完成基本的 boot 之后，会将执行流交给运行时层，初始化内核的各种功能模块，如驱动注册、任务调度队列初始化等，从

而构造一个完整的内核运行时。

上述内容构成了 ArceOS 所提供的内核基础服务。本文所讨论的四种内核扩展均需要调用这些模块提供的服务。同时它们也代表了内核的核心功能。通过共享同一套基础服务组件，开发新的内核架构的工作量便会大大降低。

3.3 接口定义与实现解耦：以硬件抽象层为例

组件化系统的一个重要目标就是让开发者各自维护属于自己的组件，而不会彼此干扰，从而能够集合众人之力推进内核快速迭代升级。那么应当如何让不同的内核模块，如不同的硬件平台（qemu 与各种实体开发板）、不同的文件系统（fatfs、ramfs、ext4fs）能同时接入到内核中，而不会互相影响呢？在 ArceOS 上，我们以硬件抽象层为例子，展示如何针对模块设计一组通用的功能接口，来实现不同内核模块的接入。

3.3.1 硬件抽象层介绍与对比

硬件抽象层（Hardware Abstraction Layer，简称 HAL）是操作系统与硬件平台之间的接口层，它负责屏蔽不同硬件平台的细节，并为上层提供统一的硬件功能接口。我们目前讨论两种不同的硬件抽象层实现：

1. axhal：这是 ArceOS 自身的硬件抽象层实现。虽然 axhal 可以支持让 ArceOS 构建为宏内核、Unikernel、Hypervisor 等多种内核架构，但因为 ArceOS 本身为 Unikernel 架构，其自带的硬件抽象层实现也保持了 Unikernel 的特点，即最简化设计，比如静态配置、不直接支持动态读取设备信息等。
2. PolyHAL：这是一个通用、轻量级的硬件抽象层实现。它将 HAL 抽象为一个独立的 crate 并发布在 crates.io 上，从而让更多内核能够方便地接入该 HAL 的实现。

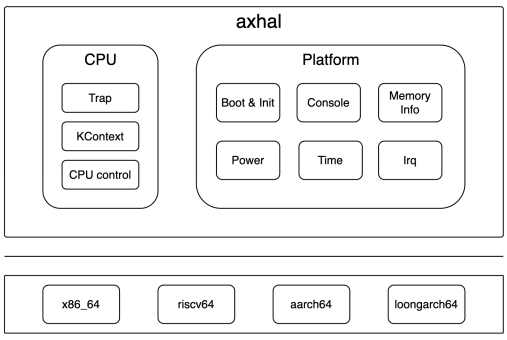

图 3.4 axhal 功能示意图

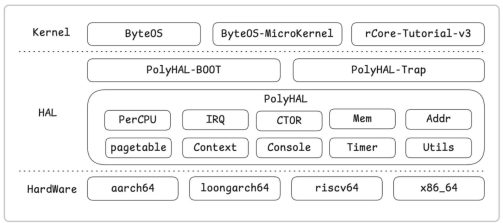

图 3.5 PolyHAL 功能示意图

尽管 PolyHAL 和 axhal 都属于模块化的硬件抽象层，将对不同硬件的操作细

节封装并对外提供服务，但这并不代表着将 PolyHAL 适配到 ArceOS 是简单的工作。同样的，对于 PolyHAL 支持的内核，如 ByteOS[27]、rCore-Tutorial-v3 等，直接将 axhal 替换掉 PolyHAL 并不一定可以正常工作。因为两者设计理念不同，对外提供的接口的语义和实现也不同。我们将通过对 PolyHAL 和 axhal 的接口进行分析对比，展示它们的不同之处。

我们先对 axhal 和 PolyHAL 的相似功能进行分析。图 3.4 和 3.5 展示了 axhal 和 PolyHAL 的功能示意图。整体上两者的功能是相似的，包括了对不同指令架构的 CPU 操作、boot 内核、异常处理、时钟、内存管理等功能。但尽管支持的功能相似，对外提供的调用方式却有许多区别，一个例子是获取当前任务所在的 CPU ID。axhal 通过 *this_cpu_id* 获取当前任务所在的 CPU ID，而 PolyHAL 则通过 *hart_id* 来获取。当内核接入的 HAL 从 axhal 替换为 PolyHAL 时，不同的函数命名会带来额外的胶水代码适配成本。表 3.1 展示了两者部分功能相近但实现不同的接口。由于 axhal 和 PolyHAL 由不同开发者设计，*绝大多数的功能接口都存在细微区别*，尽管功能相似，但直接替换 axhal 为 PolyHAL 并不能保证内核的正常工作。我们需要对内核代码进行适配，才能完成替换工作，而这会带来相当大的工作量。

功能	**axhal 接口**	**PolyHAL 接口**
获取当前 CPU ID	*this_cpu_id*	*hart_id*
启用 Irq	*irqs_enabled*	*irq_enable*
任务上下文结构	*TaskContext*	*KContext*
获取当前页表基地址	*read_page_table_root*	*PageTable::current()*
内核初始化函数	需要命名为 *rust_main*	用 *#[arch_entry]* 标注

表 3.1　axhal 与 PolyHAL 支持功能的偏差

表 3.2 展示了 axhal 和 PolyHAL 支持功能的部分不同点。由于由不同的开发者设计，侧重实现的功能也不一定一致。axhal 为了满足 ArceOS 对 Rust Standard Library 的支持，较好地提供了 TLS（Thread Local Storage）支持，而 PolyHAL 则没有提供。同样的，由于 PolyHAL 希望能尽可能提供针对不同平台的泛用性，提供了包括设备树解析、动态调整设备配置等功能，而 axhal 为了保持 ArceOS Unikernel 的最简化特点，选择了静态配置的方式。在不同场景需求下，我们可能需要做到快速选择不同的 HAL 实现并接入到内核中。

功能	axhal	PolyHAL
TLS(Thread Local Storage)	✓	X
FP/SIMD[a]	x86_64/riscv64/aarch64	x86_64
ACPI[b]	X	x86_64
VGA	X	✓
Device Tree Parser	X	✓
设备内存区域等信息配置	静态配置	动态解析
Page table	由外部 crate page_table_multiarch 实现	自带实现

[a] FP/SIMD: Floating point and SIMD support
[b] ACPI: Advanced Configuration and Power Interface

表 3.2　axhal 与 PolyHAL 支持功能不同点对比

3.3.2　通用接口层定义

为了能够便捷地接入不同 HAL 实现，同时解决表 3.1 提到的接口定义偏差问题，我们希望能够为 HAL 定义一个通用接口层，为各类内核需要使用到的接口给一个标准化的规定。上层内核按照这个接口层提供的接口来完成对应的内核功能，而不需要不关心真实接入的 HAL。对接口层的实现交给具体的 HAL 完成。

这层硬件抽象层接口的好处在于：

1. 让不同内核能避免了解具体 HAL 的实现细节、初始化细节，使用一套统一的接口来完成硬件交互功能
2. 便于根据不同的场景需求，快速选择不同的 HAL 实现并接入到内核中

图 3.6 展示了通用接口层的示意图。我们将计划实现的 HAL 通用接口层和 vfs(Virtual File System) 接口层进行类比。vfs 是由 Linux 中开始定义的一组通用的文件系统操作接口，并也在 ArceOS 中予以支持。它包括打开、读取、写入等各种常见操作。不同的文件系统实现只需要实现这些接口，就可以接入到对应的内核中。vfs 接口也为文件系统开发者所认可，目前 Linux 已经支持了超过十种文件系统，证明了这组接口的通用性和可扩展性。我们认为 HAL 通用接口层也可以借鉴 vfs 接口的设计思路，定义一组通用操作接口。这样，开发者只需要实现这些接口，就可以接入到对应的内核中。

对于 HAL 接口层，我们将其分为 CPU Interface（CPU 相关控制接口）、Platform Interface（平台相关控制接口）两部分。CPU 接口主要负责对内核所运行的 CPU 的操作，如读取当前 CPU ID、设置中断使能等；Platform 接口主要负责对所在硬件平台的操作封装，如 boot 内核、串口输出、设置时钟中断等。表 3.3 展示了我们目

子功能	接口名称	功能
Platform Init	platform_init	为主核初始化平台设备
	platform_init_secondary	为从核初始化平台设备
Console	write_bytes	向串口写入字节数组
	read_bytes	从串口读取字节
Irq	set_enable	设置当前 CPU 的中断使能
	register	注册中断处理函数
	unregister	注销中断处理函数
	handle	接受中断信息并进行处理
Mem	phys_ram_ranges	运行平台的所有物理内存
	reserved_phys_ram_ranges	运行平台的保留物理内存
	mmio_ranges	运行平台的 MMIO 内存信息
Power	cpu_boot	引导指定的 CPU 启动
	system_off	关闭整个系统
Time	current_ticks	返回当前的时钟滴答数
	ticks_to_nanos	将时钟滴答数转换为纳秒
	nanos_to_ticks	将纳秒转换为时钟滴答数
	epochoffset_nanos	RTC wall time 相对于系统全局时钟的偏移
	set_oneshot_timer	设置单次定时器

表 3.3　硬件平台相关的通用控制接口

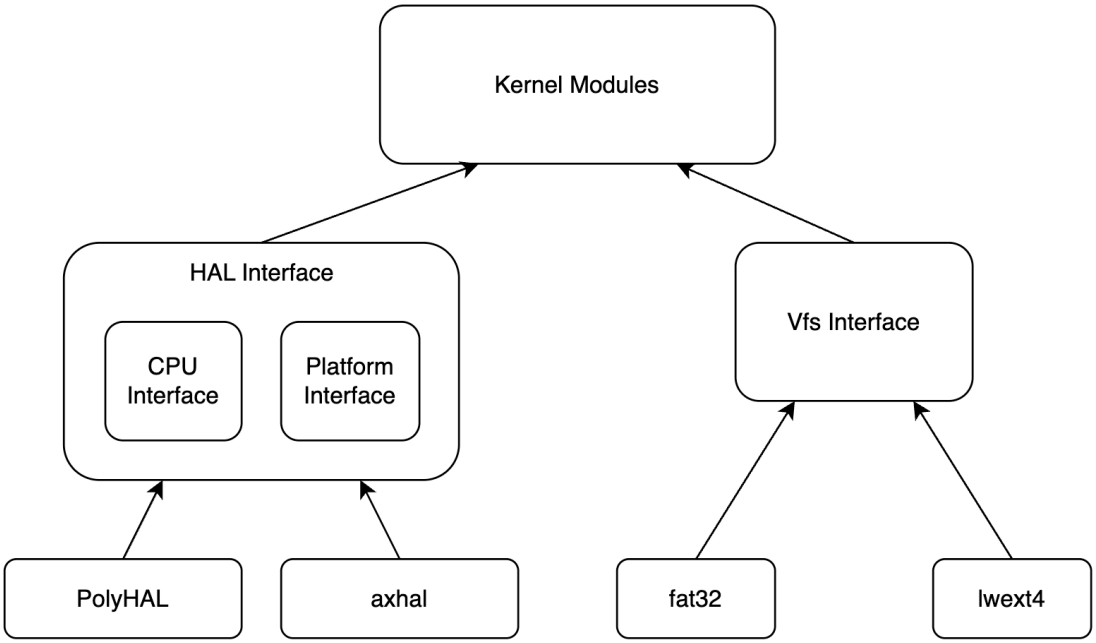

图 3.6　通用接口层示意图

前设计的 HAL 接口层的 Platform Interface 定义。它兼顾了 PolyHAL 和 axhal 对外提供的接口，尽量做到对不同 HAL 实现的兼容。我们希望通过这组接口定义，能够让不同内核在接入 HAL 时，避免了对 HAL 实现细节的了解和适配工作。

axhal_plat 仓库提供了 Platform Interface 接口的具体代码定义。而 axhal_platforms 仓库则提供了对不同硬件平台对这一组通用接口的实现。这些平台都可以接入到 ArceOS 中，让 ArceOS 具备了多平台的兼容性。

ArceOS 也为其他功能模块设置了类似表 3.3 的通用功能接口，包括 axfs（文件系统模块）、axdriver（驱动管理模块）等，从而让开发者可以基于这组接口开发更多实现，不断完善 ArceOS 的功能。

需要强调的是，目前已有的接口定义不一定便是完善的、无需修改的，它仅仅能保证支持目前的 ArceOS 功能。当 ArceOS 需要进一步完善时，有可能对这些接口进行修改与更新。当更新了接口定义，则需要已经接入的模块进行相应的更新。这与 Linux 中的 vfs 接口的要求一致。而且对硬件抽象层等模块的接口设计在此前没有较为成熟的参考，目前仅适用在 ArceOS 上，没有像 vfs 接口等规范一样为广大开发者所认同。但我们认为*接口设计是否得到认可，与 ArceOS 本身的功能完善性是相辅相成的*。当 ArceOS 支持了足够多的内容，拥有了一定知名度，自然能够吸引更多的开发者来为其贡献扩展工作，届时我们设计的这些接口也会为更多人所了解，并被尝试迁移到其他内核上。因此目前对于不同功能模块的接口设计虽然还是一个尝试，不过仍然有坚持开发完善的必要。

3.4 为 ArceOS 适配新的内核模块

3.3 节介绍了 ArceOS 的通用功能接口设计。依据 ArceOS 内核提供的通用接口，开发者可以方便地为其适配新的内核模块。我们以文件系统为例，介绍为 ArceOS 适配 ext4 文件系统的细节。

ArceOS 为文件系统提供了两组接口，分别是负责文件系统操作的 `VfsOps` 和负责文件系统节点操作的 `VfsNodeOps`。接口定义如下：

代码 8 axfs 模块定义的通用接口

```rust
/// Filesystem operations.
pub trait VfsOps: Send + Sync {
    /// Do something when the filesystem is mounted.
    fn mount(&self, _path: &str, _mount_point: VfsNodeRef) ->
     ↪  VfsResult;
    /// Do something when the filesystem is unmounted.
    fn umount(&self) -> VfsResult;
    /// Format the filesystem.
    fn format(&self) -> VfsResult;
    /// Get the attributes of the filesystem.
    fn statfs(&self) -> VfsResult<FileSystemInfo>;
    /// 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;
    /// Do something when the node is closed.
    fn release(&self) -> VfsResult;
    /// Get the attributes of the node.
    fn get_attr(&self) -> VfsResult<VfsNodeAttr>;
    // file operations:
    /// Read data from the file at the given offset.
    fn read_at(&self, _offset: u64, _buf: &mut [u8]) ->
     ↪  VfsResult<usize>;
    /// Write data to the file at the given offset.
    fn write_at(&self, _offset: u64, _buf: &[u8]) -> VfsResult<usize>;

    ...
}
```

ArceOS 原生支持的文件系统为 fatfs。它实现了 `VfsOps` 和 `VfsNodeOps` trait，在内核启动时会以 fatfs 为根文件系统。但为了扩展 ArceOS 功能，支持更多场景（如全国大学生计算机系统能力大赛内核实现赛道），我们需要为 ArceOS 适配 ext4 文件系统。

已有的 ext4 文件系统实现大多是仍使用 C 语言编写，在接入到 ArceOS 时需要先利用 bindgen[28] 等工具对外暴露可调用的 Rust 接口。我们选用 lwext4 文件系统的 rust 实现版本（下称 lwext4_rust[29]）作为 ext4 的 Rust 版本实现。lwext4 是一个轻量级的 ext4 文件系统实现，专门为嵌入式系统和资源有限的设备（如微控制

器、低功耗硬件）设计。它是对传统 ext4 文件系统的简化版本，旨在减少对系统资源（如内存、处理能力）的需求，同时仍然提供 ext4 文件系统的核心功能。因为 ArceOS 本体为 Unikernel，资源有限，所以 lwext4 是一个合适的选择。

lwext4_rust 的实现已经提供了对 ext4 文件系统的基本操作，如创建、删除、读取和写入文件等。为了将 lwext4_rust 接入到 ArceOS 中，我们只需要做如下两个操作：

1. 通过调用 lwext4_rust 提供的服务，实现 `VfsOps` 和 `VfsNodeOps` trait 所要求的方法。
2. 将上述 trait 的实现和 lwext4_rust 加入到构建依赖选项中，并通过 feature 等条件编译机制，让内核可选使用 lwext4_rust 作为根文件系统

对于第一项，ArceOS-lwext4 是 ArceOS 接入 lwext4_rust 文件系统的代码实现。它通过 400 行代码，实现了 `VfsOps` 和 `VfsNodeOps` trait 所要求的方法，完成了 lwext4_rust 的适配工作。

第二项可以通过 Rust 的包管理工具 Cargo 来完成。Cargo 是 Rust 的官方包管理工具，它可以帮助我们管理项目的依赖、构建和发布等工作。通过将 lwext4_rust 作为依赖添加到 Cargo.toml 中，并使用 feature 等条件编译机制，即可自选当前启用的根目录文件系统种类。代码 9 展示了 ArceOS 对根目录文件系统的选择逻辑。我们根据目前是否启动了 `myfs`、`lwext4_rs`、`fatfs` 等 feature，来选择当前的根目录文件系统。其中 myfs 指的是用户自定义的文件系统，它不一定遵循 ext4 或者 fatfs 的规范，而是一个任意的文件系统实现。lwext4_rs 指的是 lwext4_rust 文件系统实现，fatfs 指的是 ArceOS 原生支持的 fatfs 文件系统实现。

代码 9 axfs 定制当前的根目录文件系统

```rust
// axfs/src/fs/mod.rs
cfg_if::cfg_if! {
  if #[cfg(feature = "myfs")] {
    pub mod myfs;
  } else if #[cfg(feature = "lwext4_rs")] {
    pub mod lwext4_rust;
  } else if #[cfg(feature = "fatfs")] {
    pub mod fatfs;
  }
}

```

通过如上分析可知，如果需要适配新的内核模块，开发者只需要额外提供通用接口的实现，并将其加入到内核的构建选项中即可。而通用接口的实现代码和其他内核功能并没有耦合关系，对新模块的实现并不会影响已有的功能。因此对于 ArceOS 来说，适配新的内核模块的工作量和复杂程度都会下降。通过分工合作

适配更多内核模块，可以推进 ArceOS 的快速迭代升级。

3.5 小结

本章节详细介绍了 ArceOS Backbone 和 Starry 扩展应用的实现细节。现在我们回顾一下 2.3 提出的设计目标，并与我们的实现细节进行对应。

1. 3.2 节介绍了 ArceOS Backbone 的组成和具体划分的功能模块。在 ArceOS 中这些模块组成了内核的基础服务，可以为不同内核形态所复用。
2. 3.1.1 介绍了 ArceOS 的执行环境复用模式。它允许不同内核形态复用同一套执行环境，从而降低了内核开发的复杂度和工作量。而其中我们以宏内核扩展的执行流为例，详细分析了 Kernel Plugin 的启动和处理逻辑，解释了如何在复用执行环境的情况下，完成宏内核扩展的功能。
3. 3.3 节引入了通用功能组件接口的概念。让开发者在为内核适配新的模块时，专注于实现接口的功能，而不需要关心其他内核模块的实现细节。我们以 axhal 模块为例，展示了 ArceOS 的通用功能接口设计，并在 3.4 节讲述了如何将新的内核模块适配到 ArceOS 上。通过将接口的定义和实现进行区分，我们可以使得不同内核模块之间互不干扰，从而便于合作，快速完善内核功能。

通过仔细划分组件功能、设计组件接口并引入一系列新机制，我们较好地实现了 ArceOS 的宏内核扩展功能，展示了灵活内核架构设计的可行性和优势。

第 4 章　宏内核扩展应用 Starry 实现

本章节将介绍宏内核扩展应用 Starry 的结构设计和具体实现，从而在更高的角度上理解宏内核扩展的实现。

4.1　Starry 结构设计

图 4.1 展示了 Starry 的结构图。

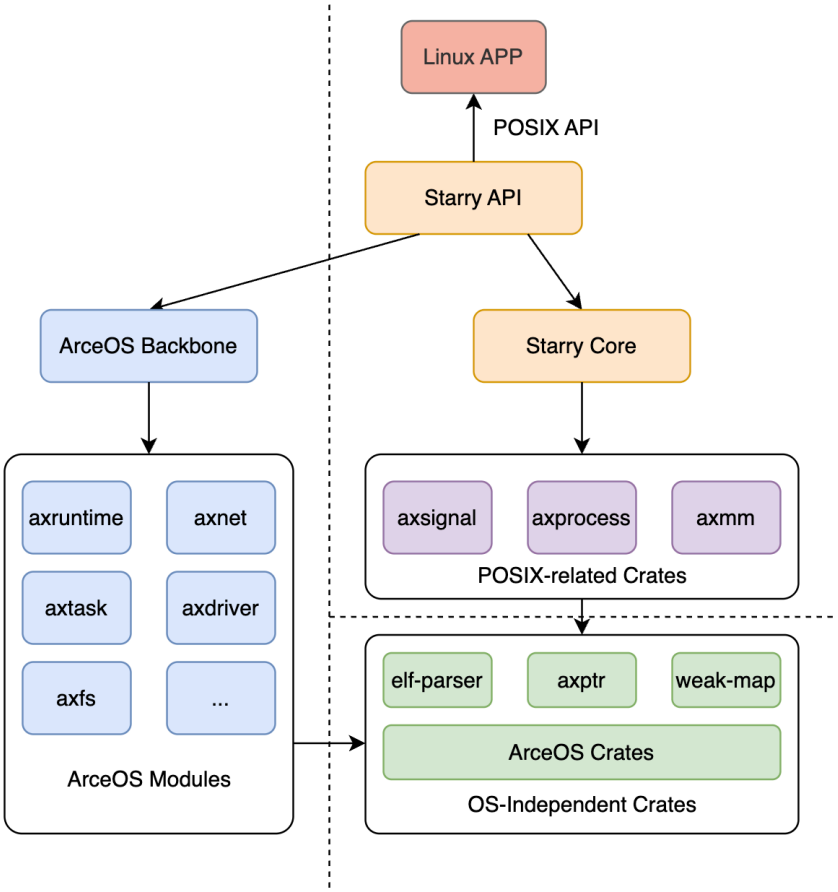

图 4.1　Starry 结构图

我们将 Starry 的实现划分为三大部分：

1. Starry Core：Starry 关于宏内核的核心逻辑实现。在这里我们定义了用户程序加载、进程管理、地址空间管理等一系列功能，并实现了宏内核的初始化逻辑。它与上层实现的接口（POSIX 接口或其他接口）无关，仅负责完成一个基本的宏内核应当具备的功能：特权级切换、进程粒度的资源隔离等。

2. Starry API：Starry 的 API 层，负责实现对上层应用的接口。它会调用底层 ArceOS 和 Starry Core 中的各种功能，并将它们封装为标准的 POSIX 接口。当接收到用户的 syscall 请求时，Starry 会将其转发给 Starry API 层进行处理，从而实现兼容 Linux Application 的目标。
3. Starry Crates：Starry 抽象出来的模块组件。遵循组件化开发的原则，我们希望 Starry 中实现的功能不仅可以被 ArceOS 上的其他内核架构复用，甚至还可以被其他内核复用。我们将这些功能抽象为 Starry-Crates，提供给其他内核使用。目前 Starry-Crates 中已经实现的主要模块如表 4.1 所示。值得一提的是，这些 crates 并没有依赖任何 ArceOS 模块层的实现，也就是说它们也是内核无关模块。其他内核也可以直接复用这些模块，从而降低自身开发宏内核功能的工作量。

名称	功能	可访问链接
axprocess	进程创建、管理、退出等逻辑	Starry-OS/axprocess
axsignal	POSIX 规范下的信号定义和处理逻辑	Starry-OS/axsignal
axptr	封装用户地址空间的访存操作	Starry-OS/axptr

表 4.1　Starry 抽象的部分内核无关组件

4.2　Starry Core 逻辑

Starry Core 实现了宏内核功能的核心逻辑。3.1.1 节介绍了复用执行环境下宏内核扩展的执行流。

我们将以 Starry Core 中的代码为例，展示 Starry 接管 Unikernel 执行流后的具体操作：

1. 当 ArceOS 完成了内核的初始化之后，会调用应用程序的 `main` 入口，将执行流交给应用程序。
2. 此时 Starry 会枚举所有需要运行的用户态应用程序，并串行地为每一个应用程序调用 `run_user_app` 函数。
3. 在这个函数中，我们会为待加载的应用程序创建一个新的用户态地址空间。之后我们会调用 `load_user_app` 函数加载应用程序的 ELF 文件，并将其映射到用户态地址空间中。最后我们会创建一个新的任务，并且初始化任务的上下文信息，将其加入到调度队列中。
4. 最后 Starry 的内核任务主动让出 CPU，等待用户态应用程序的运行。内核会

代码 10 Starry 加载应用程序

```rust
/// This function is called by the kernel to run a user application.
/// Arguments and environment variables are passed to the application.
pub fn run_user_app(args: &[String], envs: &[String]) -> Option<i32> {
  let mut uspace = new_user_aspace_empty()
    .and_then(|mut it| {
        copy_from_kernel(&mut it)?;
        Ok(it)
    })
    .expect("Failed to create user address space");

  let path = FilePath::new(&args[0]).expect("Invalid file path");

  let (entry_vaddr, ustack_top) =
    load_user_app(&mut uspace, args, envs)
    .unwrap_or_else(|e| panic!("Failed to load user app: {}", e));
  let user_task = spawn_user_task(
      Arc::new(Mutex::new(uspace)),
      UspaceContext::new(entry_vaddr.into(), ustack_top, 2333),
      axconfig::plat::USER_HEAP_BASE as _,
  );
  // Wait for the user task to finish.
  user_task.join()
}
```

通过 `join` 函数等待其结束，并返回其退出码。

5. 等当前应用程序运行完成之后，内核会继续调用 `run_user_app` 函数，加载下一个应用程序。这样，Starry 就完成了对用户态应用程序的批处理加载和运行。

可以看到，Starry Core 并不需要为宏内核实现许多内核功能的初始化逻辑，而更多是对加载、运行用户应用程序的支持。因为绝大多数内核功能的初始化逻辑已经在 ArceOS Backbone 中实现，对于 Starry 来说，我们只需要关注宏内核的特性需求，即运行用户应用程序。因此，执行环境的复用有效降低了 Starry 开发的复杂程度和工作量。

另外 Starry Core 中也封装了一系列重要结构，如对标 POSIX 线程的 StarryThread 等，并为他们提供了相关的方法，从而为 Starry API 层的实现提供支持。

4.3 Starry API 层实现

Starry API 层的目标是为 Starry 提供 Linux 兼容的 API 接口。我们会调用 Starry Core 和 ArceOS 中已经提供的功能，并按照 POSIX API 规范的要求进行封装。当封装完毕，Starry API 层会为 Starry 提供一个 syscall 入口。当用户调用了 syscall，会触发 trap 并进入 ArceOS Backbone 的异常处理逻辑。ArceOS Backbone 判断当前异常是否属于 syscall。若是，则转发给 Starry 应用处理。Starry 调用 Starry API 层

提供的 syscall 函数，并传入对应的参数，并将获得的结果回传给用户程序。

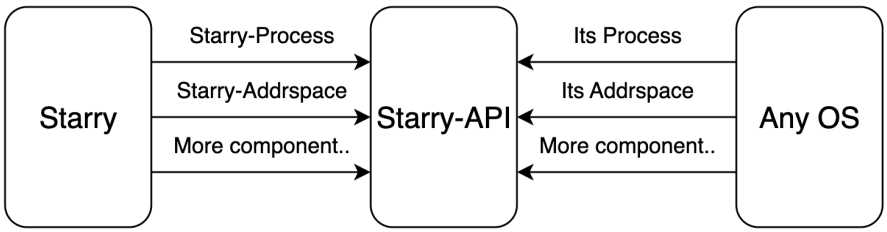

图 4.2　Starry API 复用说明

关于如何将功能封装为 POSIX API 的工作已有许多参考，我们不在此进行赘述。我们将介绍 Starry API 与组件化内核设计的关系。在开发宏内核时，对 POSIX API 的正确封装实现是一个必要但却繁琐的工作，因为 syscall 的数量繁多，且每一个 syscall 的具体功能还可能与传入的参数相关。宏内核还需要对当前调用的 syscall 进行安全检查与错误处理。尽管 POSIX API 有详细的手册规定具体的实现方式和边界情况，但开发经验告诉我们，将手册内容转化为具体代码实现的过程仍然容易出现许多错误。若能存在一个提供了较为完善的 POSIX API 实现的 crate，并可以较为方便地迁移到不同内核上，便是对宏内核乃至其他需要适配 POSIX API 的内核的一个极大帮助。这也是为何 Starry API 会抽象为一个 crate 的原因。目前 Starry API 尽管在 Rust 语义下以 crate 的形式存在，但却仍然直接依赖于 ArceOS 和 Starry Core 的实现，而不是完全解耦的状态。换句话说它仍然和 Starry 紧密耦合。在将来的工作中，我们希望能将 Starry API 独立出来，成为一个内核无关的 crate，从而真正为更多内核提供实际的支持。

Starry API 复用的一个说明图如图 4.2 所示。为了封装 POSIX API，我们需要内核提供多种功能，包括进程管理、文件系统、网络模块等。这些功能应当作为 Starry API 的输入，从而支持 Starry API 完成 POSIX API 的功能要求。具体到实现上，我们会在 Starry API 中将需要内核提供的功能封装为一组 trait，每一个 trait 实现特定范围的功能，如地址空间管理、进程管理等。代码 11 提供了一个简单的地址空间管理 trait 示例。需要适配 Starry API 的内核应当实现这一系列的 trait，并将其作为 Starry API 的输入。Starry API 会调用这些 trait 中的方法，完成对 POSIX API 的封装。这样就可以在不修改 Starry API 的情况下，适配不同内核的实现。

代码 11 Starry API 中的 AddrSpace trait 示例

```
1
2  pub trait AddrSpaceProvider {
3    fn check_region_access(
4      range: VirtAddrRange,
5      access_flags: MappingFlags
6    ) -> bool;
7
8    fn populate_area(start: VirtAddr, size: usize) -> LinuxResult<()>;
9
10   fn mmap(
11     start: VirtAddr,
12     size: usize,
13     prot: MappingFlags,
14     flags: MappingFlags,
15   ) -> LinuxResult<()>;
16
17   fn munmap(start: VirtAddr, size: usize) -> LinuxResult<()>;
18
19   fn mprotect(
20     start: VirtAddr,
21     size: usize,
22     prot: MappingFlags,
23   ) -> LinuxResult<()>;
24 }
```

第 5 章　系统评估

本章节将对目前 Starry 扩展应用的开发情况进行评估。评估内容包括如下方面：

- POSIX API 支持进度
- 扩展应用的开发工作量
- 新的内核模块的适配情况

5.1　POSIX API 支持进度

目前 Starry 目前共支持 54 个 syscall，包括进程管理、文件读取等功能，可以支持加载 Linux APP 并完成相关功能。

为了更好地测试 Starry 的 POSIX API 支持情况，我们使用了全国大学生计算机系统能力大赛操作系统内核实现赛道的测试套件。测试套件的组成如下：

测例集	测例说明
basic	基本系统调用测试
busybox	Busybox 功能正确性测试
libc-test	测试标准 C 库实现的正确性，涵盖各种 libc 接口
iozone	文件系统性能测试工具，测试读写速度、吞吐等
iperf	网络性能测试工具，测量带宽、延迟、丢包等网络指标
netperf	网络基准测试工具，可用于测量 TCP、UDP 等多种网络的性能
libc-bench	评估 libc 实现的运行时性能，如字符串操作、内存函数等
lua	支持 Lua 语言解释器，并解释其语法、语义及运行时行为
lmbench	测量系统性能的微基准测试工具，如延迟、带宽等
ltp	Linux 测试项目，全面验证 Linux 系统的功能和稳定性

表 5.1　内核实现赛道的测试套件说明

相比于系统能力大赛的的要求，我们对测试套件的通过提出了更加严格的要求：

1. 大赛仅要求在 riscv64 和 loongarch64 上进行测试，而我们要求同时通过 x86_64、riscv64、aarch64 和 loongarch64 四种指令架构的测试。
2. 大赛仅要求在单核场景下完成正确性和性能测试，但为了验证 Starry 对多核支持的正确性，我们要求所有的测例运行在多核场景下，更加贴近 Linux 的

真实表现。

为了满足我们对测例镜像的要求，笔者更新了系统能力大赛的测例编译流程，并加入了多指令架构的编译支持，详见 https://github.com/oscomp/testsuits-for-oskernel/tree/2025_multiarch 。为了在开发的过程中能随时保持已通过测例的正确性，我们为开发仓库配置了自动集成测试，同时在多核场景下测试四种指令架构的测例通过情况，以严格的规范来保证 Starry 的稳定性。只有当某个测例同时在四种指令架构的多核场景下通过了测试，才认为 Starry 成功支持了该测例。

在 2025 年 2 月初，Starry 仅仅只能通过 basic 测例中的单核场景部分。但截至本章节撰写时，Starry 对测例的通过情况如下：

测例集	通过数量/总数
basic	32/32
libc	207/217
lua	9/9
busybox	52/54
其他	未测试/未通过

表 5.2　Starry 对测试套件的通过情况

尽管仍有部分套件未通过，但它们多是性能相关的测试。对测试功能正确性的测例的支持情况良好，说明在 2 个月的时间内，Starry 在 POSIX API 的实现上已经有了较大的进展。

5.2　开发工作量

灵活内核架构的一个重要优势是通过复用基础内核服务，降低新的内核架构的开发工作量。工作量的一个重要衡量标准即是代码量。表 5.3 展示了 ArceOS 不同子系统的代码行数。可以看出，在已有的 ArceOS Backbone 基础上，只需要较小的代码量就可以构造出宏内核和 Hypervisor 两种内核形态，并且它们和原有的 Unikernel 架构是相互独立、彼此兼容的，避免了侵入式开发带来的代码耦合和兼容性问题。

需要注意的是，较小的代码量并不等同于支持的功能少。5.1 节展示了 Starry 目前的功能支持情况。为了达到同样的支持情况，传统的从零开始构建的宏内核的代码量因为包含引导内核、驱动、调度队列等内核模块的实现，一般会需要 2 万行以上的代码量。而 Starry 只需要在已有的 ArceOS Backbone 基础上，增加 1980 行

代码即可。我们还可以与笔者之前开发的 Starry-Old 进行对比。同样是在 ArceOS 的基础上进行开发，Starry-Old 需要 8504 行代码来达到等价的支持情况，而 Starry 通过新设计 Module Extension3.1.2 机制和 Namespace 3.1.3 机制，尽可能复用了已有的代码逻辑，有效降低了开发新功能的代码量。

ArceOS 的子系统	LOC	说明
元件层[1]	13368	ArceOS 提供的内核无关组件
模块层	13649	ArceOS 提供的内核相关组件
Starry	1980	宏内核扩展应用
AxVisor[2]	1572	Hypervisor 扩展应用

[1] 不包括其他开发者提供的已发布的组件。
[2] 由其他工程师开发，目前可以支持在直通所需设备等情况下，同时运行 Linux 等多个 Guest OS，有了一定的功能支持。

表 5.3　ArceOS 各子系统代码量

另外复用基础服务还可以让一个组件的新功能同时应用于不同形态的内核。比如 3.4 节提到的对 ext4 文件系统的适配不止可以用于宏内核，对于 Unikernel 和 Hypervisor 都可以同时复用，从而降低了适配新功能的工作量。

5.3　内核模块适配情况

为了进一步完善 Starry 的功能，我们为其适配了一系列内核模块。除去 3.4 节介绍的 ext4 文件系统外，我们还做了如下模块的适配工作：

1. HAL 层支持 loongarch64 指令架构：为了支持系统能力大赛的测例，我们为 ArceOS 提供了 loongarch64 指令架构的适配。目前它仅支持在 loongarch64-qemu 平台上运行，但已经足够完成比赛测例的需求。
2. 适配更多 ext4 文件系统实现：除去 lwext4 文件系统外，我们还接入了另外两个由 Rust 编写的 ext4 文件系统实现，分别是简易 ext4 文件系统 ext4-rs 和通过 Metis Model Checker[30] 检查的实现 ext4-checked，为 ArceOS 提供了更多文件系统支持。

3.3 节介绍了通用功能组件接口的概念。通过抽象出模块需要实现的接口，可以简化新的内核模块接入到 ArceOS 中的工作。更多内核模块适配到 ArceOS 可以不断完善 ArceOS 的功能，扩展 ArceOS 的应用场景，也进一步验证我们的灵活内核设计的可行性。

5.4 小结

本章节对 ArceOS 的宏内核扩展应用 Starry 的开发情况进行了评估。相比于传统的宏内核，Starry 以较小的工作量实现了对 POSIX API 的初步支持，同时适配了许多新的内核模块，扩展了自身的功能和应用场景。对 Starry 的评估，说明灵活内核的设计原则不仅可以降低开发新内核形态的工作量，还让适配新的内核模块变得更加简单，展现了其良好的复用性和可扩展性。通过动员更多技术人员参与到灵活内核的开发中，可以高效地推进开源内核的功能完善和生态构建。

第 6 章　总　结

本文针对定制内核形态这一问题，提出了灵活内核架构的设计原则，并通过对组件化操作系统 ArceOS 的开发实践，完善了宏内核扩展应用 Starry 的设计实现。评估结果表明，相比于传统宏内核来说，Starry 通过复用 ArceOS 的基础内核服务，减少了开发新功能所需的工作量，并通过抽象通用接口降低了适配新的内核模块的难度。目前 Starry 已经成为了全国大学生计算机系统能力大赛内核实现赛道的参考内核，为同学提供了设计操作系统的参考。

ArceOS 作为一个开源在 Github 上的项目，已经受到了众多开发者的关注和参与。通过贯彻组件化的设计原则，不同开发者可以为 ArceOS 开发属于自己的模块，降低彼此干扰的风险，从而能够更加高效地进行协同开发，推进 ArceOS 功能的完善。

在未来操作系统的应用场景将愈发多元化，快速定制内核的形态与功能会是一个长期的需求。我们会进一步完善 ArceOS 与 Starry，为更多操作系统开发者提供定制新内核的参考实现。我们希望通过灵活内核架构的设计原则，能够为技术人员提供一个新的思路，重新思考操作系统的整体架构设计，打破操作系统开发门槛高、难以共享生态等固有障碍，助力开源操作系统生态发展。

参考文献

[1] Ammons G, Appavoo J, Butrico M, et al. Libra: a library operating system for a jvm in a virtualized execution environment[C/OL]//VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments. New York, NY, USA: Association for Computing Machinery, 2007: 44–54. https://doi.org/10.1145/1254810.1254817.

[2] Peter S, Li J, Zhang I, et al. Arrakis: The operating system is the control plane[J/OL]. ACM Trans. Comput. Syst., 2015, 33(4). https://doi.org/10.1145/2812806.

[3] Belay A, Prekas G, Klimovic A, et al. Ix: a protected dataplane operating system for high throughput and low latency[C]//OSDI'14: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. USA: USENIX Association, 2014: 49–65.

[4] Kivity A, Laor D, Costa G, et al. Osv: optimizing the operating system for virtual machines [C]//USENIX ATC'14: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference. USA: USENIX Association, 2014: 61–72.

[5] Black D, Golub D, Julin D, et al. Microkernel operating system architecture and mach[J]. Journal of information processing, 1991, 14(4): 442-453.

[6] Anderson T E. The case for application-specific operating systems[R]. USA: University of California at Berkeley, 1993.

[7] Madhavapeddy A, Mortier R, Rotsos C, et al. Unikernels: library operating systems for the cloud[C/OL]//ASPLOS '13: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2013: 461–472. https://doi.org/10.1145/2451116.2451167.

[8] Popek G J, Goldberg R P. Formal requirements for virtualizable third generation architectures [J/OL]. Commun. ACM, 1974, 17(7): 412–421. https://doi.org/10.1145/361011.361073.

[9] Abdalkarim B A, Akgün D. Analysis of hybrid kernel-based operating systems[C]//2022.

[10] Baumann A, Barham P, Dagand P E, et al. The multikernel: a new os architecture for scalable multicore systems[C/OL]//SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2009: 29–44. https://doi.org/10.1145/1629575.1629579.

[11] Engler D R, Kaashoek M F, O'Toole J. Exokernel: an operating system architecture for application-level resource management[C/OL]//SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 1995: 251–266. https://doi.org/10.1145/224056.224076.

[12] Gutfreund S H. Maniplicons in thinkertoy[C/OL]//OOPSLA '87: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications. New York, NY, USA: Association for Computing Machinery, 1987: 307–317. https://doi.org/10.1145/38765.38835.

[13] Levis P, Madden S, Polastre J, et al. Tinyos: An operating system for sensor networks[M/OL]// Ambient Intelligence. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005: 115-148. https://doi.org/10.1007/3-540-27139-2_7.

[14] Ford B, Back G, Benson G, et al. The flux oskit: a substrate for kernel and language research[C/OL]//SOSP '97: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 1997: 38–51. https://doi.org/10.1145/268998.266642.

[15] Kantee A. Rump kernels: No os? no problem![J/OL]. USENIX ;login:, 2012, 36(6): 35-43. https://www.usenix.org/publications/login/december-2012-volume-37-number-6/rump-kernels-no-os-no-problem.

[16] Schatzberg D, Cadden J, Dong H, et al. Ebbrt: a framework for building per-application library operating systems[C]//OSDI'16: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. USA: USENIX Association, 2016: 671–688.

[17] The Rust Project Developers. Conditional compilation - the rust reference[EB/OL]. 2025. https://doc.rust-lang.org/reference/conditional-compilation.html.

[18] Zheng Y, Jia Y. Starry[EB/OL]. https://github.com/oscomp/starry-next.

[19] Zheng Y, Jia Y. Starry-old: A monolithic kernel based on arceos[EB/OL]. https://github.com/Starry-OS/Starry-Old.

[20] Schatzberg D, Cadden J, Krieger O, et al. A way forward: enabling operating system innovation in the cloud[C]//HotCloud'14: Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing. USA: USENIX Association, 2014: 4.

[21] Chen H, Miao X, Jia N, et al. Microkernel goes general: Performance and compatibility in the HongMeng production microkernel[C/OL]//18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24). Santa Clara, CA: USENIX Association, 2024: 465-485. https://www.usenix.org/conference/osdi24/presentation/chen-haibo.

[22] Klein G, Elphinstone K, Heiser G, et al. sel4: formal verification of an os kernel[C/OL]//SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2009: 207–220. https://doi.org/10.1145/1629575.1629596.

[23] Millwood J, VanVossen R, Elliott L. Performance impacts from the sel4 hypervisor[C]// Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium. 2020: 13-15.

[24] Martins J, Pinto S. Shedding light on static partitioning hypervisors for arm-based mixed-criticality systems[C]//2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS). 2023: 40-53.

[25] Jia Y. Arceos: An experimental modular os written in rust.[EB/OL]. https://github.com/arceos-org/arceos.

[26] Hu K. Axvisor[EB/OL]. https://github.com/arceos-hypervisor/axvisor.

[27] Yang J. A posix-compatible kernel written in rust[EB/OL]. https://github.com/Byte-OS/ByteOS.

[28] You J Y, Álvarez E C, Fitzgerald N, et al. bindgen - rust bindings to c libraries[EB/OL]. 2025. https://crates.io/crates/bindgen.

[29] Xiao L. lwext4_rust - rust implementation of the lightweight ext4 filesystem[EB/OL]. https://github.com/Azure-stars/lwext4_rust.

[30] Liu Y, Adkar M, Holzmann G, et al. Metis: File system model checking via versatile input and state exploration[C/OL]//22nd USENIX Conference on File and Storage Technologies (FAST 24). Santa Clara, CA: USENIX Association, 2024: 123-140. https://www.usenix.org/conference/fast24/presentation/liu-yifei.