摘　要

本文针对组件化操作系统 ArceOS 的宏内核扩展 Starry，旨在设计并实现一个高可靠性且功能完善的文件系统管理组件，以完善 Starry 的功能，增强其安全性、兼容性和可扩展性。研究首先深入分析了 ArceOS 的组件化设计哲学、分层架构及其提供的文件系统基础，并剖析了 Starry 宏内核的整体架构与核心机制，为文件系统组件的构建奠定了坚实基础。核心工作在于文件系统管理组件的设计与实现：构建了清晰的层次化架构，支持多种具体文件系统实现，并在此基础上实现了一系列关键的 POSIX 文件操作相关系统调用。在开发与验证阶段，本文举例说明了具体的实现过程，并通过标准测试套件对组件进行了全面的功能、兼容性与稳定性测试。测试结果表明，所实现的文件系统组件达到了预期目标，显著提升了 Starry 对 Linux 应用程序的兼容性。本文工作不仅为 Starry 贡献了一个关键的核心组件，也验证了在 ArceOS 基础上扩展复杂系统功能的可行性，同时指出了当前实现的局限与未来开发优化的方向。

关键词：ArceOS；Starry；文件系统；组件化操作系统；宏内核

Abstract

This paper addresses Starry, the macro-kernel extension for the component-based operating system ArceOS. It aims to design and implement a highly reliable and fully functional file system management component to enhance Starry's capabilities, security, compatibility, and scalability. The research begins with an in-depth analysis of ArceOS's component-based design philosophy, layered architecture, and its existing file system infrastructure. It also examines the overall architecture and core mechanisms of the Starry macro-kernel, establishing a solid foundation for the development of the file system component. The core contribution is the design and implementation of the file system management component. This involved establishing a clear hierarchical architecture, enabling support for various specific file system implementations, and subsequently implementing key POSIX-compliant system calls for file operations. During the development and verification phase, this paper details the implementation process with examples. The component was comprehensively tested for functionality, compatibility, and stability using standard test suites. The results indicate that the implemented file system component met its design objectives and significantly improved Starry's compatibility with Linux applications. This work not only contributes a crucial core component to Starry but also validates the feasibility of extending complex system functionalities upon ArceOS. Furthermore, it highlights the limitations of the current implementation and suggests directions for future development and optimization.

Keywords: ArceOS; Starry; file system; componentized operating system; monolithic kernel

目 录

插图和附表清单

第 1 章　引　言

1.1　课题背景

1.1.1　操作系统内核

操作系统是用于管理计算机中各类硬件和软件资源的关键软件程序。操作系统内核是操作系统的核心部分，它向下协调和控制各种硬件资源，包括处理器、硬盘、外设、内存等，向上则为用户程序提供基础服务接口，使得用户程序能够正确执行，并且能够较为简单地使用计算机的各种资源。

针对不同的应用场景，操作系统内核有多种架构类型。**宏内核（Monolithic Kernel）**是应用最广泛的内核类型，它将操作系统的全部功能都放在内核空间中，使用内核特权级运行。宏内核的通用性较强，但各个系统功能之间的耦合度较高，对开发和维护带来一定程度的困难。宏内核的各个功能间缺乏隔离，所以系统的可靠性比较差，容易在出现故障时导致系统整体崩溃。Linux 作为宏内核的典型代表，在各个领域和设备类型上都有广泛的应用。

微内核（Microkernel）[1] 则仅在内核中实现操作系统所需的最基本和必要的功能，例如进程管理和基本的内存管理等，而其他功能则以用户态服务的形式实现，并通过用户态进程间的接口进行交互。相比宏内核，微内核的安全性高，不易受到安全攻击，适合用在对安全性要求高的场景，同时也使得内核更容易开发和维护。但因为进程间通信需要频繁进行内核和用户空间之间的上下文切换，产生的代价较高而性能较差。QNX[2] 作为成功的微内核，主要部署在嵌入式系统中，在汽车、军事等高可靠性要求领域有广泛的应用。seL4[3] 是微内核的另一个典型代表，它首次使用了形式化验证的方法验证内核的正确性。

混合内核（Hybrid Kernel）[4] 结合了宏内核和微内核的特点，依然将部分功能放在用户态实现，但考虑了性能影响，将更多服务放在内核态实现，如 Windows NT 和 macOS（XNU）等操作系统。

外核（Exokernel）[5] 是一种极简的内核架构，只提供基本的硬件保护和资源复用功能，而将硬件资源的抽象和管理权限交给应用程序，使其能够直接管理硬件资源并根据自身需求定制资源管理策略。这种架构具有较高的灵活性和性能优势，但也增加了应用程序的开发复杂度，要求开发者深入理解底层硬件。

库操作系统（LibOS）[6] 是一种将传统操作系统服务（如网络栈、文件系统等）作为库直接链接到应用程序中的架构。这种设计旨在通过减少内核抽象和上

下文切换来提升性能和专用性，适用于云端微服务等场景。为 Java 虚拟机实现的 Libra[7] 即为库操作系统的一个代表。

单内核（Unikernel）[8] 是一种较为新兴的内核架构，它将应用程序与所需的内核组件一起编译为一个高度优化的单一镜像，直接运行在裸机或虚拟机管理器上。单内核具有启动迅速、内存占用少、安全性较高等优点，特别适合物联网、云计算等环境。

本文的研究对象即为以组件化操作系统 ArceOS（基础形式为单内核）为基座扩展实现的宏内核操作系统 Starry。

1.1.2 文件系统

在现代计算机系统中，文件系统作为操作系统的核心组件，是控制数据存储与管理的关键机制。它为用户和应用在数据存储设备上组织、存储、命名、共享及保护信息提供了抽象方法，是数据持久化、多用户多任务环境下信息共享与隔离的基石。

文件系统有多种多样的核心职责。**命名与组织**允许用户通过有意义的名称和层级目录（如树状结构）管理文件。**存储管理**负责在物理介质上分配回收空间，将文件逻辑结构映射到物理存储块，并跟踪空间使用情况。**数据访问**提供标准接口（如读写操作）供应用程序使用。**元数据管理**则涉及文件属性（如名称、大小、时间戳、权限）的创建与维护。

文件系统的设计随硬件与需求不断演进。从早期简单的 FAT，到功能更强的 UFS、NTFS、ext 系列（引入日志、权限等），再到应对海量数据与新存储介质的日志结构文件系统（LFS[9]）、写时复制文件系统（CoW，如 ZFS[10]、Btrfs[11]）及分布式文件系统（如 NFS[12]、HDFS[13]）。这种多样性反映了不同场景对文件系统的性能、可靠性、扩展性等方面的不同侧重。

在内核架构中，文件系统模块居于核心，常作为模块存在。现代操作系统多采用经验证可靠的虚拟文件系统（VFS）[14] 作为抽象层，与设备驱动程序交互，转换逻辑 I/O 为硬件操作，通过为具体文件系统实现抽象，完成对多类型文件系统的支持。

1.1.3 ArceOS

ArceOS[15] 是一个基于 Rust 语言开发的操作系统，旨在为微虚拟机等环境快速构建高性能、高安全的专用操作系统。其核心设计目标强调组件的高度可重用性、灵活的按需定制能力以及接口的专用化，从而在提升性能的同时保证良好的兼容性。ArceOS 的核心功能围绕微虚拟机场景进行优化，提供包括多核处理、多

线程、网络协议栈以及文件系统等基础内核服务。

在整体架构上，ArceOS 以 Unikernel 作为其基本运行形态，采用模块化和分层设计。这种设计使其不仅能作为 Unikernel 运行，还能通过组件的不同组合与配置，构建出宏内核或虚拟机管理程序（Hypervisor）等多种内核形态。ArceOS 的核心组件各司其职，组件间的交互主要通过 Rust 语言的 crate 依赖机制以函数调用的方式进行，对于一些难以避免的反向或循环依赖，则通过一个名为 crate_glue 的特殊元件进行接口定义和解耦。

ArceOS 的关键技术特点在于其深度组件化设计和基于 Rust 语言特性的优化。它利用 Rust 的特性（feature）机制，实现了操作系统功能的灵活选择与替换，从而达到高度的专用化。特别地，ArceOS 为 Rust 应用程序设计了 API 快速路径，通过直接借用并实现 Rust 标准库的接口，避免了传统 POSIX 接口在微虚拟机场景下的冗余开销，显著提升了性能，同时也能直接兼容已有的 Rust 应用。对于 C 语言应用，ArceOS 也提供了 POSIX 兼容接口。此外，ArceOS 充分利用了 Rust 语言带来的内存安全、并发安全、强大的包管理（Cargo）和条件编译等优势。用户主要通过 Rust 标准库 API 或 POSIX API 与内核进行交互，资源管理策略（如内存分配算法、任务调度策略）也支持通过编译特性进行定制。

ArceOS 的主要优点在于其高性能（尤其在 API 快速路径和小数据包网络处理上）、高度的可定制性、良好的组件可重用性以及由 Rust 语言带来的内存与并发安全性。这使得开发者能够根据具体应用需求，快速构建出极简且高效的专用操作系统。然而，它也存在一些局限性，例如部分组件间的循环依赖问题虽有标记和管理机制，但尚未被彻底消除；对微内核形态的完整支持仍在探索中。ArceOS 非常适用于对性能、安全性和定制化有较高要求的微虚拟机场景、嵌入式系统以及希望充分发挥 Rust 语言优势的各类应用环境。

1.1.4 Starry

Starry 的设计初衷是借助 ArceOS 所具备的高可复用性与灵活性的内核组件，迅速搭建一个具备较完整功能、兼容 POSIX API 的宏内核系统，以此检验 ArceOS 作为“灵活内核”基础架构对多种内核形态的支持能力。其主要功能聚焦于宏内核的典型特征，包括进程管理、用户态与内核态的分离、系统调用接口的实现，以及用户程序的加载和运行。

从架构层面来看，Starry 作为一层宏内核扩展或“内核插件”，构建于 ArceOS Unikernel 之上。ArceOS 提供了诸如硬件抽象、内存管理、任务调度等底层基础能力，而 Starry 则在此之上实现了宏内核所需的核心机制，例如进程抽象、基于 ArceOS 内存管理的地址空间隔离，以及面向用户应用的系统调用接口。通过这种

分层设计，Starry 能够直接复用 ArceOS 已有的诸多成熟模块，包括设备驱动、文件系统（如 axfs 及其相关模块）和调度器（可通过适配或扩展 axtask 实现）等。

作为 ArceOS “灵活内核”理念的具体落地，Starry 展现了如何在 Unikernel 基础上通过插件式扩展实现宏内核形态，突出体现了内核组件的高度复用性，显著减少了新内核形态开发所需的工作量。用户主要通过 POSIX API 与 Starry 交互，这使得大量现有 C 语言应用能够顺利移植和运行。在资源管理方面，Starry 负责进程级的资源分配与管理（如虚拟地址空间、文件描述符等），而底层的物理资源分配和设备管理则依赖 ArceOS 提供的能力。

Starry 的主要优势在于极大降低了宏内核系统从零开发的难度，并继承了 ArceOS 组件化架构带来的灵活性与易维护性，为特定场景下快速构建定制化、POSIX 兼容的宏内核环境提供了有效方案。但与此同时，Starry 也存在一定局限性，例如由于引入额外抽象层和组件间交互，可能带来一定的性能损耗，其功能完整性和稳定性也尚难与成熟的通用宏内核系统媲美。基于这些特征，Starry 主要适用于教学实验、操作系统原理验证、原型系统开发，以及结合 ArceOS 底层特性实现定制化宏内核的应用场景。

1.2 相关研究工作

1.2.1 BrickOS

BrickOS[16] 提出了“积木式内核”这一全新架构，核心思想在于实现“内核架构可配置性”。该理念打破了传统操作系统在宏内核与微内核之间的固有界限，使开发者能够依据具体应用需求及硬件平台特性，灵活地从极简的“简要内核”到功能完善的宏内核或微内核等多种形态进行定制构建。

推动这一设计的根本动力，源自泛在计算领域所面临的复杂挑战。一方面，硬件平台高度异构，既包括缺乏内存管理单元（MMU）的嵌入式微控制器（如 ARM Cortex-M 系列），也涵盖具备完整功能的服务器级处理器（如 x86、ARM Cortex-A 系列），两者在体系结构和资源规模上差异显著。另一方面，应用需求日趋多样且变化迅速。在这种背景下，传统的单一内核或固定架构难以适应广泛场景，而针对小批量、多样化需求进行内核定制又常常导致高昂的开发成本和低复用性。

为了解决上述问题，BrickOS 将操作系统的核心功能模块（如任务调度、内存管理、文件系统等）拆解为一系列可独立演进、按需组合的“积木”组件。这些组件不仅高度模块化，还支持灵活的运行模式配置：既可在共享地址空间的内核态下运行以获得极致性能和低延迟，也可部署为独立的用户态进程，通过进程隔离机制提升系统安全性与容错能力。正是这种部署方式的灵活性，使 BrickOS 能够

针对不同硬件特性（如是否具备 MMU）及具体应用需求（如强实时控制对低延迟的要求，或高吞吐智能应用对算力的需求）进行深度定制和优化。

为实现上述灵活的组件部署与隔离，BrickOS 特别重视为底层硬件的内存保护机制（如 MPU 或 MMU）提供统一的抽象层。该抽象层成为不同运行模式下组件安全隔离的基础。例如，多个组件在内核态共享地址空间时，该抽象层可保障其内存安全，这对于缺乏完整 MMU 的嵌入式平台尤为关键；而当组件以用户态进程方式运行时，则可借助独立进程地址空间实现更强的隔离。

BrickOS 的主要贡献在于提出并实践了一种面向泛在计算、支持异构硬件且可敏捷定制的领域操作系统内核设计方法。原型系统的开发与测试结果显示，该策略具备实际可行性并能取得良好性能表现。然而，这类高度灵活和可配置的系统在实际应用中也面临诸多挑战：如组件间复杂交互的管理难题、配置空间扩展带来的测试与验证复杂性，以及在生态系统完善度和社区支持方面与主流通用操作系统之间的差距等。

1.2.2 DragonOS

龙操作系统（DragonOS）[①]旨在打造一个高性能且自主可控的操作系统内核，聚焦于云计算的轻量化应用场景，并以实现对 Linux 应用的兼容为核心目标。该项目自 2022 年启动，采用 Rust 作为主要开发语言，借助其内存安全特性提升系统可靠性。DragonOS 社区以开源、商业中立的方式运作，遵循 GPLv2 协议，致力于长期发展成为独立自主、高可靠性的服务器操作系统。

在架构设计上，DragonOS 采用宏内核或类宏内核的模式，将内存管理、多核支持、进程调度、进程间通信（IPC）、文件系统抽象、异常与中断处理以及设备驱动等核心功能模块直接集成于内核层。此种架构有助于实现 Linux 兼容，并提升各模块间的通信效率，从而满足高性能需求。此外，DragonOS 在虚拟化支持和设备模型方面表现出一定优势，显示其在宏内核基础上进一步引入了模块化优化的理念。

DragonOS 的突出意义在于以 Rust 语言为基础，探索构建兼容 Linux 的自主操作系统内核，为操作系统自主可控的发展方向提供了新路径。作为社区主导的开源项目，它为开发者提供了丰富的实践平台。然而，作为新兴系统，DragonOS 仍面临诸多挑战：全面实现 Linux 兼容性难度极大；虽然已完成 GCC 等基础工具链的移植，但生态系统的完善尚需时日；其性能优势也有待实际应用进一步验证。项目规划在五年内实现生产环境的大规模部署，目前开发主要依赖 Rust nightly 工具链及 QEMU 平台进行。

① https://dragonos.org

1.2.3 星绽

星绽（Asterinas）[①] 致力于打造一款兼具安全性、高效性并兼容 Linux ABI 的操作系统。其设计理念主要建立在两大基础之上：一方面，借助创新的框架内核结构与 Rust 语言的深度集成，显著提升内存安全保障；另一方面，注重优化开发体验，力求成为高安全需求领域的理想选择。

该系统的核心特色在于采用独特的框架内核架构，将所有不安全的 Rust 代码严格限定于一个精简的可信计算基（TCB）中。绝大多数内核功能模块，包括系统调用、文件系统、网络协议栈及设备驱动，均以安全的 Rust 代码实现于此架构之上，从而整体提升系统安全性。其底层操作系统框架 OSTD（OS Standard Library and Drivers）作为基础支撑层，结构简明高效，代码规模约为一万行，便于实现高效抽象。所有内核模块（如 Virtio 驱动）均以 OSTD 为基础进行开发。

星绽的内核开发完全基于 Rust 语言，并对 unsafe Rust 代码块的使用加以严格限制。通过 OSDK 工具集提升开发效率，同时采用 MPL 许可证，允许模块开发者自主选择开源或专有授权。对 Linux ABI 的良好兼容性显著降低了应用迁移门槛。星绽优先支持云计算场景下的关键驱动（如 Virtio），主要面向两类高安全需求环境：一是虚拟机可信执行环境（TEE），以增强数据安全；二是安全容器场景，提供更强的操作系统级隔离能力。

星绽的主要创新体现在提出了框架内核架构，并以 Rust 语言实现了以内存安全为核心的内核系统。该项目为高安全性应用提供了可行的 Linux 替代方案，同时改善了内核开发的便捷性。尽管如此，星绽仍面临一些挑战：如对硬件驱动的广泛支持尚需进一步投入，目前主要只支持 x86-64 虚拟机平台；OSTD 的正确性也有赖于社区的持续审查。总体而言，星绽为安全操作系统内核的设计与实现积累了宝贵的经验和实践路径。

1.3 本文工作

本文旨在为组件化操作系统 ArceOS 的宏内核扩展 Starry 提供一个高性能、高可靠性且功能完备的文件系统管理组件。本工作将依托 ArceOS 所提供的底层基础组件以及 Starry 宏内核的现有架构，尤其是层次化的文件系统组件架构，完成对底层各个组件的分析与改进，精细地设计和实现 Starry 中文件系统所需的数据结构，并实现一套与 POSIX 标准兼容的文件系统相关系统调用接口，最终构建出一个经过严格测试的文件系统模块，并将其集成到 Starry 宏内核中，实现对宏内核扩展工作的进一步完善和验证。

① https://github.com/asterinas/asterinas

本文将分为六个章节。第二章将深入剖析基座操作系统 ArceOS 的设计理念、分层架构、核心组件及文件系统相关组件（如 axfs、axfs_vfs），并将阐述其系统应用的构建与执行流程。第三章将详细分析基于 ArceOS 的宏内核扩展 Starry 的架构和实现。第四章将在此基础上，重点阐述文件系统管理组件的设计与实现，包括对类文件对象的抽象和文件描述符表的实现，以及多种文件相关系统调用的具体实现方式。第五章将对实现的文件系统组件的开发过程、测试环境、测试用例及测试结果进行详细说明与分析，并进行性能评估。第六章将对全文工作进行总结，并对未来研究方向进行展望。

第 2 章　基座操作系统 ArceOS 架构分析

本章将深入分析 ArceOS 的总体架构，为后续宏内核扩展奠定基础。

2.1　ArceOS 整体架构

ArceOS 由许多具有依赖关系的组件组成，图 2.1 展示了 ArceOS 的整体架构。自下而上，ArceOS 主要分为以下几个层次：

- **底层平台**：包括裸机和 QEMU 等硬件或虚拟化平台，为操作系统提供运行环境。
- **元件层**：实现上与具体操作系统无关的基础组件，如内存分配器、调度器、网络协议栈、驱动抽象等，这些组件可被不同操作系统重用。
- **模块层**：ArceOS 的核心功能组件，包括 axruntime、axconfig、axalloc、axfs、axsync、axtask、axdriver、axnet 等，负责 ArceOS 作为操作系统内核的基本功能实现。
- **接口层**：为上层应用提供统一的系统接口，包括 arceos_rust_api 和 arceos_posix_api，分别面向 Rust 和 C 语言生态。
- **用户库层**：如 axstd、rust_std、axlibc、musl_libc 等，进一步封装接口层，兼容主流用户库，方便应用程序移植。
- **应用层**：最终运行的系统应用程序，可以为 Rust 应用或 C 应用。

这种分层架构有效地实现了各层之间的解耦，便于功能扩展和模块复用，同时为不同类型的应用提供了良好的兼容性和灵活性。

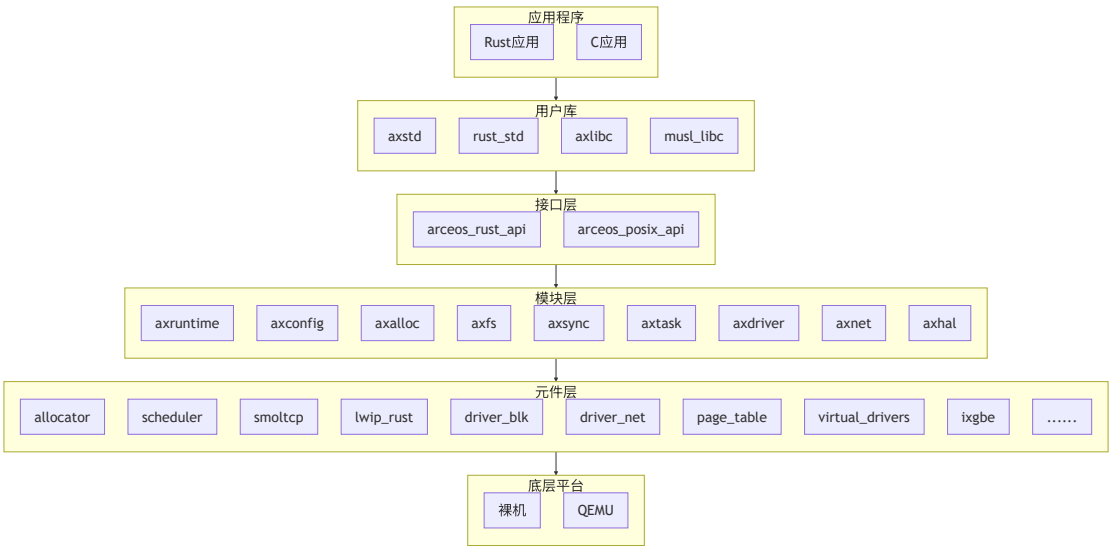

图 2.1　ArceOS 架构图

2.2　ArceOS 组件介绍

本节将简要介绍 ArceOS 中的重要组件。

2.2.1　axhal 组件

axhal 是 ArceOS 的硬件抽象层，主要作用在于为底层硬件资源提供统一的抽象接口。借助 axhal，上层模块能够以统一的方式访问不同平台和架构的硬件，无需关注具体实现细节的差异。

ArceOS 支持 RISC-V、ARM、x86 及 LoongArch 等主流 CPU 架构。针对这些架构，axhal 对中断管理、上下文切换等功能接口进行了统一封装。此外，ArceOS 兼容多种硬件平台（如 QEMU、树莓派、飞腾派等），各平台独特的启动和配置流程均在 axhal 中加以适配并实现。

2.2.2　axalloc 组件

axalloc 负责 ArceOS 的动态内存分配。其内部集成了两类分配器：字节分配器可满足任意大小的内存请求，页分配器则专注于连续内存页的分配。根据不同的 feature 选项，axalloc 可以支持多种分配算法（如 TLSF、伙伴系统、Slab 等），以便灵活适配各种应用场景。

2.2.3 axtask 组件

在 ArceOS 中，线程作为调度的基本单位，被统一称为“任务”。axtask 组件实现了任务的调度与管理功能。在宏内核扩展时，任务机制还可进一步扩展为进程支持。axtask 不仅负责任务的初始化、调度和同步，还内置了多种调度算法（如先入先出、时间片轮转、完全公平调度等），以满足不同类型系统的需求。

2.2.4 axdriver 组件

axdriver 实现了 ArceOS 的设备驱动抽象，涵盖网络、存储、显示等多种设备类型。对于每类设备，axdriver 都提供统一的接口规范，使上层模块无需关心具体硬件型号。例如，存储设备驱动主要负责数据块的读写，而网络设备驱动则专注于数据包的收发。

2.2.5 axruntime 组件

axruntime 是 ArceOS 的核心运行时模块。在系统引导完成并由 axhal 移交控制权后，axruntime 负责后续的初始化流程。在进入应用 main 函数前，axruntime 会统一初始化所有启用的内核模块。其 rust_main 函数会根据系统配置和已启用特性，依次完成日志、内存分配、虚拟内存、任务调度、设备驱动等模块的初始化。此外，axruntime 还负责注册全局处理例程（如中断、异常分发和 Rust panic 处理），以保障系统稳定运行，并实现对应功能。

2.2.6 axconfig 组件

axconfig 是 ArceOS 的配置管理模块，主要负责定义和提供系统运行所需的各类参数。在系统构建阶段，axconfig-gen 工具会生成包含目标平台、架构、内核加载基址、物理内存范围、任务栈大小、MMIO 区域等关键常量的配置文件。编译时，axconfig 通过宏机制将这些配置项集成到内核代码中，使各模块能够据此完成正确的初始化和操作，从而灵活适配不同硬件和应用场景。

2.2.7 axsync 组件

axsync 为 ArceOS 提供了同步与互斥机制。在多任务并发环境下，为保证共享资源的安全访问和任务间的高效协作，axsync 基于 axtask 的底层管理能力（如等待队列），封装并实现了多种常用同步原语，包括互斥锁（Mutex）、信号量（Semaphore）、条件变量（Condvar）等，为系统其他模块构建可靠的并发程序提供了基础支撑。

2.2.8 axnet 组件

axnet 是 ArceOS 的网络协议栈模块，为系统提供网络通信能力。应用程序如果需要进行网络数据交换，则可以通过启用相关特性使用 axnet。该组件依赖 axdriver 提供的网络设备驱动接口以实现底层硬件交互。在此基础上，axnet 负责实现核心网络协议（如部分 TCP/IP 协议族）、数据包的封装与解析、网络接口管理及套接字（Socket）接口等，使 ArceOS 能够接入网络并支持多样化的网络应用。

2.2.9 文件系统相关组件

ArceOS 的文件系统管理功能由多个组件协同实现，包括 axfs、axfs_vfs、axfs_ramfs 和 axfs_devfs 等。这些组件构建在 axdriver 提供的设备驱动基础上，支持多种文件系统类型，并向上层提供简洁易用的接口。整个文件系统架构采用了清晰的分层设计：最底层是设备驱动层（axdriver），上面依次是磁盘抽象层（axfs 中的 dev 模块）、虚拟文件系统层（axfs_vfs）、具体文件系统实现层（如 axfs_ramfs、axfs_devfs 等）以及文件系统接口层（axfs 中的 fops 和 api 模块）。这种层次化结构有效降低了各组件间的耦合度，使系统更易于维护和扩展。

2.2.9.1 磁盘抽象

在 axdriver 的基础上，axfs 组件中的 dev 模块定义了一个磁盘抽象，将底层的 axdriver 所提供的块设备 AxBlockDevice 封装为一个带有游标（cursor）的磁盘抽象 Disk，支持顺序或随机读写。封装后，各文件系统不再需要直接操作底层的块设备，而是使用磁盘抽象所提供的更易于使用的接口。表2.1 展示了磁盘抽象所提供的接口。

表 2.1 磁盘抽象接口

方法	描述
new	创建一个新的 Disk 实例，初始化游标和底层块设备
size	获取磁盘的总字节数（容量）
position	获取当前游标在磁盘中的全局字节偏移
set_position	设置游标到指定的全局字节偏移
read_one	从当前游标位置读取一个块或部分块的数据
write_one	从当前游标位置写入一个块或部分块的数据
read_offset	读取指定偏移处的整个块数据

续表 2.1　磁盘抽象接口

方法	描述
write_offset	向指定偏移处写入一个完整块的数据

在具体的文件系统实现中，由于使用外部元件作为文件系统的基础实现，还需为磁盘抽象实现对应的特质（trait），以适配文件系统所需求的磁盘访问接口。例如 axfs 组件中对 ext4 文件系统的实现中所基于的元件 lwext4_rust 要求的磁盘设备需要实现 KernelDevOps trait，其中只包含 read, write, seek 和 flush 四个接口，所以需要对磁盘抽象进一步封装，以满足底层元件的需求。

2.2.9.2　虚拟文件系统

axfs_vfs 是 ArceOS 中的一个元件层组件，它定义了一套完整的虚拟文件系统（VFS）接口，这套接口具有良好的通用性，可以被多种操作系统内核复用。虚拟文件系统作为一个抽象层，隐藏了不同文件系统实现的具体细节，为上层应用提供了统一的文件操作接口。

axfs_vfs 组件主要通过 trait 定义了文件系统操作的标准接口，包括 VfsOps、VfsNodeOps 等核心特质。其中 VfsOps 特质定义了文件系统级别的操作，如挂载、卸载、格式化等；而 VfsNodeOps 特质则定义了文件节点级别的操作，如打开、关闭、读写等。这种设计使得不同的文件系统实现（如 FAT32、Ext4 等）只需实现这些特质，即可方便地集成到 ArceOS 中。表 2.2 和表 2.3 分别展示了 VfsOps 和 VfsNodeOps 接口中所包含的方法。

表 2.2　VfsOps 接口说明

方法	描述
mount	当文件系统被挂载到指定路径时执行的初始化操作
umount	当文件系统被卸载时执行的清理操作
format	格式化文件系统，清除所有数据并建立初始结构
statfs	获取文件系统的状态信息，如总容量、剩余空间等属性
root_dir	获取文件系统的根目录节点引用

表 2.3　VfsNodeOps 接口说明

方法	描述
open	节点被打开时的初始化操作，如更新访问时间、增加引用计数等
release	节点被关闭时的清理操作，如释放资源、更新元数据等
get_attr	获取节点的属性信息，包括类型、大小、文件权限等
read_at	从文件指定位置读取数据，支持随机读取
write_at	向文件指定位置写入数据，支持随机写入
fsync	将文件的缓存数据同步到存储设备，确保数据持久化
truncate	调整文件大小，可以扩展或截断文件
parent	获取目录的父目录节点引用
lookup	在目录中查找指定路径的子节点
create	在目录中创建新的文件或子目录
remove	从目录中删除指定的文件或子目录
read_dir	读取目录项，用于目录遍历
rename	重命名或移动文件/目录到新的位置
as_any	将对象转换为 Any 类型，用于运行时类型转换

2.2.9.3　文件和目录接口

为了进一步封装文件和目录的操作，ArceOS 在 axfs 组件的文件操作接口 fops 模块中定义了结构，用于表示被打开的文件和目录，如代码2.1所示，可以看到，它们实际上是对 VfsNodeRef（即 Arc<dyn VfsNodeOps>，指向一个实现了虚拟文件系统节点 trait 的对象）的封装，主要是在类型上将文件和目录区分开来，便于上层 API 的使用。

代码 2.1　fops 中文件和目录的结构定义

```
pub struct File {
    node: WithCap<VfsNodeRef>,
    is_append: bool,
    offset: u64,
}

pub struct Directory {
```

```
    node: WithCap<VfsNodeRef>,
    entry_idx: usize,
}
```

在 fops 的基础上，axfs 组件在 api 模块中提供了可供其他组件使用的对外接口。为了接口的通用性，api 模块再次对文件进行封装，并将文件的读写操作改为实现 axio 模块中提供的 Read 和 Write trait，从而使得文件的读写可以与其它对象（如网络数据流）的读写操作一致，便于上层组件的复用。除了针对已打开的文件或目录进行的操作以外，api 模块还提供了大量的实用函数，表 2.4 展示了这些函数的功能。

表 2.4　axfs 组件 api 模块提供的实用函数

函数名	描述
read_dir	返回一个目录的迭代器，用于读取目录中的所有项
canonicalize	将路径转换为规范的绝对路径形式
current_dir	获取当前进程的工作目录路径（cwd）
set_current_dir	更改当前进程的工作目录到指定路径
read	读取文件的全部内容，返回为字节数组
read_to_string	读取文件的全部内容，并转换为字符串
write	将字节数据写入文件，覆盖原有内容
metadata	获取文件或目录的元数据信息（如大小、修改时间等）
create_dir	在指定路径创建一个新的空目录
create_dir_all	创建指定路径的目录及其所有不存在的父目录
remove_dir	删除指定路径的目录（必须为空）
remove_file	删除指定路径的文件
rename	将文件或目录从源路径移动到目标路径
absolute_path_exists	检查指定的绝对路径在文件系统中是否存在

2.3　系统应用执行流程

本节介绍 ArceOS 接入系统应用的标准执行流程，通过该流程，可以了解 ArceOS 的各部分组件如何协同工作，从而组成一个可运行的操作系统。

2.3.1 系统构建

ArceOS 使用 Make 进行整个系统的构建。Make 是一个经典的代码构建工具，基于 Makefile 文件中规定的文件依赖关系和构建规则，以及通过命令行调用时传入的参数，自动地进行配置生成、代码编译、链接和执行等操作。由于项目的复杂性，ArceOS 将构建脚本拆解为主文件夹下的 Makefile 文件和各个子 Makefile 文件，并通过 include 语句将各个子 Makefile 文件包含进来，从而实现完整的构建流程。

ArceOS 的构建过程主要分为生成配置和编译链接两步。当执行 make 命令时，构建系统会首先通过构建目标 defconfig 调用工具 axconfig-gen 生成用于系统启动的配置文件。在调用时，构建系统会生成一系列参数并传入 axconfig-gen，其中包含由目标平台决定的配置文件路径、额外的配置文件路径，以及目标的平台信息、架构信息和启用的 CPU 核数。根据这些参数，axconfig-gen 依次以覆写的方式应用默认配置文件、平台决定的配置文件、额外的配置文件和调用命令中指定的具体参数，并最终将配置信息保存在.axconfig.toml 文件中。该配置文件将在编译时被 axconfig 模块中的宏包含进来，从而使得配置中的配置项能被内核代码使用。表 2.5 列出了.axconfig.toml 文件中的部分配置项，可以看出，它们在系统的各个部分都可能被用到。

表 2.5　ArceOS 配置文件中的部分配置项

配置项	值（举例）	说明
arch	x86_64	指定系统的目标架构，决定了底层硬件抽象层的实现
platform	x86_64-qemu-q35	指定运行平台，影响设备驱动和平台特定代码的选择
kernel-base-vaddr	0xffff_8000_0020_0000	内核镜像的基础虚拟地址，是内存管理的关键参数
mmio-regions	多个地址区间	内存映射 I/O 区域，定义了与外设通信的物理地址空间
task-stack-size	0x40000	每个任务的栈大小，影响系统支持的任务复杂度和嵌套深度

在正式编译前，构建系统还需根据参数计算出系统需要启用的 feature。在 Makefile 的 feature.mk 部分，构建系统会根据系统应用的属性和若干构建选项，包括 LOG 等级、总线类型和是否支持 SMP，选择编译时需要启用的 feature。当系统

应用的语言是 Rust 时，系统可以根据 Cargo 的配置自动推断出模块的依赖关系和需要启用的 feature，而当语言是 C 时，则需要额外指定一系列 C 标准库所依赖的 feature。此后，构建系统将进行系统代码的编译和链接，最终生成一个 ELF 格式的可执行文件，再使用 objcopy 工具将其转化为二进制内核镜像文件。

2.3.2 系统引导和启动

在构建完成后，ArceOS 可以进行系统引导、启动和初始化，其流程如图 2.2 所示。

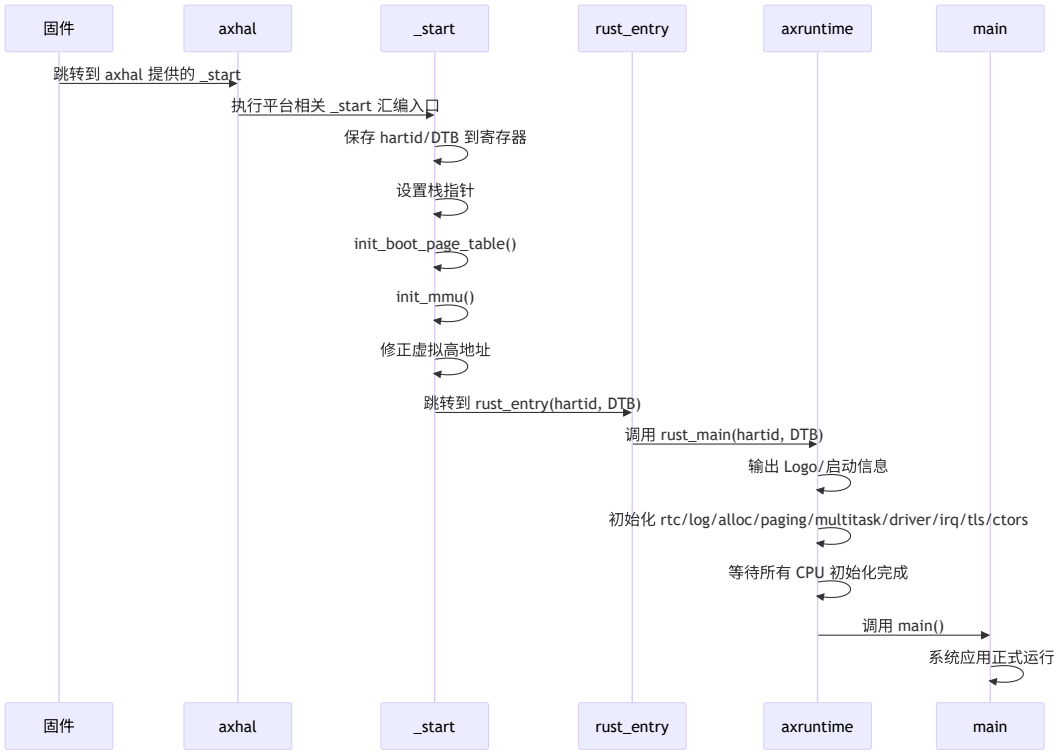

图 2.2 系统引导和初始化的流程图

ArceOS 的系统引导和启动部分主要由硬件抽象层模块 axhal 实现。axhal 为其所支持的每个平台均提供了系统引导代码。在构建时，指定平台所对应的入口函数 _start 和 _start_secondary （如果启用了 SMP）将被链接到.text.boot 段，作为主 CPU 和副 CPU 在启动时最先执行的代码。不同架构的引导代码均使用对应的汇编实现，在实现上不完全相同，但总体功能是类似的。

以 RISC-V 架构为例，_start 函数是系统启动时的第一个执行点。该函数首先将参数 hartid（硬件线程 ID）和设备树指针（DTB）保存到寄存器中，为后续作为参数传递给 Rust 入口函数 rust_entry 做准备。随后，设置启动时使用的栈指针，以支持接下来的函数调用。接下来，调用函数 init_boot_page_table 初始化引导页表，

并调用函数 init_mmu 启用内存管理单元（MMU），为系统的虚拟内存管理打下基础。在启用 MMU 后，函数通过为栈指针增加物理地址和虚拟地址的偏移量来修正虚拟高地址，确保内存访问的正确性。最终，函数将保存的 hartid 和设备树指针作为参数传递给 axruntime 模块中的入口函数 rust_entry，将控制权从汇编代码转移到 Rust 代码，开始系统的高级初始化过程。通过这一系列步骤，ArceOS 能够从最基本的硬件状态过渡到可执行复杂操作系统功能的状态。当启用多核时，副 CPU 将从 _start_secondary 函数开始执行。不同的是，_start_secondary 的参数为 hartid 和副 CPU 的栈地址，并且不需要调用 init_boot_page_table 函数，因为主 CPU 已经完成了初始化页表的操作。

2.3.3 系统初始化

在 ArceOS 系统完成启动引导后，运行时模块 axruntime 开始接管后续的初始化流程，为正式进入系统应用的 main 函数做准备。axruntime 中的初始化流程实现在 rust_main 函数中，接受 CPU 的 hartid 和 DTB 作为参数，被前述的 _start 函数调用。

rust_main 函数首先会向控制台输出 ArceOS 的系统 Logo（以文本形式）和关键启动信息，包括使用的架构、平台、编译目标、日志等级和 SMP 配置，用于提示系统启动成功、已进入初始化阶段，并帮助用户了解当前系统采取的配置。之后，函数开始依据模块的启用情况，即对应 feature 的启用情况，对各个模块依次进行初始化：

- 如果启用了 rtc feature，即启用了真实的时钟，则额外输出启动时的时间戳；
- 调用 axlog::init 初始化 axlog 模块中的日志系统，即根据启动时的参数设置输出的最大日志等级，此后可以调用 axlog 中的宏正常输出日志；
- 如果启用了 alloc feature，即有动态内存分配机制，则会调用 init_allocator 函数，初始化全局内存分配器，并将可用的物理内存区域注册到内存分配器中；
- 如果启用了 paging feature，即启用了虚拟内存管理，则会调用 axmm::init_memory_management 函数，初始化虚拟内存管理，从而支持虚拟地址空间；
- 如果启用了 multitask feature，即多任务功能，则会调用 axtask::init_scheduler 函数，初始化任务调度器，从而支持任务的调度；
- 如果启用了 fs, net, display feature 中的任意一个，则需要调用 axdriver::init_drivers 函数，初始化对应的设备驱动，并且在接下来调用对应模块的初始化函数，从而完成文件系统、网络协议栈和显示设备的初始化；
- 如果启用了 smp feature，则会调用 mp::start_secondary_cpus，依次启动其他

CPU 核；

- 如果启用了 irq feature，即启用了中断处理，则会调用 init_interrupt 函数，初始化中断处理，并注册时钟中断处理函数；
- 如果启用了 tls feature 但没有启用 multitask feature，则调用 init_tls 函数，初始化线程局部存储功能；
- 调用 ctor_bare::call_ctors 函数，对系统中所有全局构造函数进行调用，这些函数均带有 #[ctor] 标记，在编译时会被收集到.init_array 数组中，在初始化时被依次调用；

在上述初始化流程结束后，主 CPU 会进行自旋等待，知道所有 CPU 核均完成初始化。随后，调用 main 函数，正式开始执行系统应用。

第 3 章　宏内核扩展 Starry 分析

3.1　Starry 整体架构

Starry 是 ArceOS 的宏内核扩展，实质上是作为 ArceOS 的系统应用，充分利用 ArceOS 中已有的组件和代码，从而实现宏内核应有的功能特性。Starry 与 ArceOS 相同，遵循了组件化的设计原则，将独立的功能封装在不同的组件中。组件化的架构有助于代码组织，提高可重用性，使得 Starry 的维护、测试和扩展变得更加容易。

整体上，Starry 还呈现一种分层架构，如图 3.1 所示。

图 3.1　Starry 整体架构图

Starry 的最顶层是其中运行的用户应用，它们通过系统调用与 Starry 内核进行交互。Starry 中包含的组件可分为三部分：starry-api、starry-core 和 Starry 扩展组件。

starry-api 是 Starry 所提供的系统调用接口，作为用户应用与内核交互的入口通道，专门接收并处理来自用户态的系统调用请求。在处理系统调用请求时，starry-api 会根据所需进行的操作调用 starry-core 组件或 ArceOS 提供的系列基础组件。

starry-core 组件中主要实现宏内核的地址空间管理、进程管理、用户程序管理等核心功能。starry-core 的功能实现依托于 ArceOS 提供的系列基础组件和为 Starry 开发的独立组件。ArceOS 所提供的基础组件包含但不限于以下内容：axfs 负责文件系统的目录管理，axtask 处理任务的创建与调度流程，axsync 提供锁、信号量等同步原语，axnet 处理网络数据包的收发，axruntime 维护应用程序的运行时环境，axmm 管理物理内存与虚拟地址空间，axhal 则构建了硬件抽象层。在架构底

层，axhal 组件直接与硬件层交互，通过统一的接口抽象屏蔽了不同硬件平台的差异性。

除了 ArceOS 原本提供的基础组件外，Starry 还将宏内核扩展所需要的部分功能抽象成扩展组件，如 axprocess 管理进程生命周期及资源分配，axsignal 实现进程间信号传递机制。这些组件不与其他部分耦合，可以直接被应用到其他内核中，符合 ArceOS 和 Starry 的组件化设计原则，有助于实现内核开发中的代码复用。

3.1.1 starry-core 组件

starry-core 组件是 Starry 宏内核扩展的核心，其组件内部的关键子模块协同工作，以实现宏内核的核心功能，主要包括内存管理、任务管理和时间管理。

内存管理模块 (mm) 负责用户态进程的地址空间。其主要职责涵盖用户地址空间的创建与维护，ELF 可执行文件的加载与映射——包括解析 ELF 格式、映射程序段、设置内存权限、初始化用户栈与堆，以及处理动态链接器。该模块还管理信号处理栈帧的映射，同步内核与用户地址空间的映射（在特定架构），并提供用户内存访问控制机制。

任务管理模块 (task) 负责用户态任务（线程和进程）的创建、管理和调度相关的扩展功能。它支持创建新用户任务，并为任务附加扩展数据（如执行时间统计）。模块内定义了线程和进程的特定数据结构，存储如线程 ID、信号管理器、程序路径、地址空间引用等信息。全局表用于维护所有活跃的线程、进程、进程组和会话，采用弱引用机制。此外，它还实现了命名空间接口和用于信号处理的等待队列封装。

时间管理模块 (time) 定义了与任务执行时间统计相关的数据结构和逻辑。它负责记录进程及其子进程在用户态和内核态的执行时间，支持多种类型的间隔计时器，并精确跟踪单个任务的时间消耗。这些统计数据在任务状态切换或调度时更新，为系统提供准确的时间信息，并被任务管理模块所利用。

3.1.2 starry-api 组件

starry-api 组件的核心目标是为上层应用程序提供一套标准化的系统调用接口，旨在构建一个与 POSIX 标准兼容或高度相似的 API 层，使得 Linux 应用程序能够在其之上运行。starry-api 中的 imp 模块是系统调用具体实现的核心，其下又根据所实现的系统调用功能领域划分为多个子模块，其他子模块主要用于支持系统调用的实现。starry-api 的架构如图 3.2 所示。

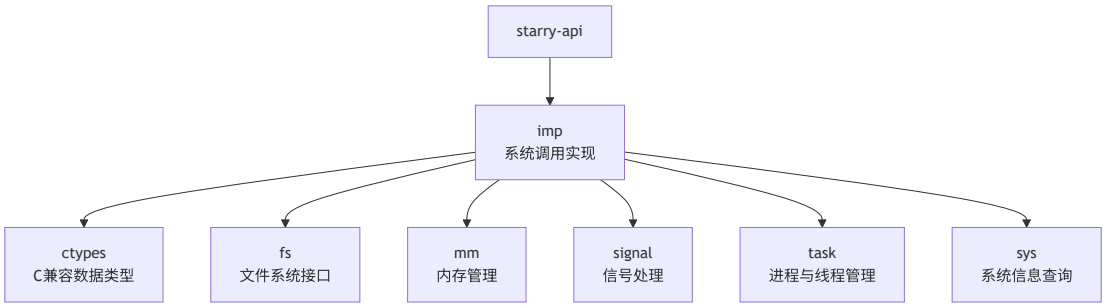

图 3.2　starry-api 架构图

- ctypes 模块扮演着至关重要的角色，它定义了大量与 C 语言兼容的数据类型（如 mode_t, off_t, stat, sockaddr, timespec, iovec 等）。这些类型确保了 API 接口与遵循 C ABI（Application Binary Interface）的应用程序之间的互操作性，是实现跨语言系统调用的基石。
- fs 模块及其子模块共同构成了文件系统的接口。它涵盖了广泛的文件操作，包括但不限于：文件和目录的创建、打开、关闭、读写（sys_read, sys_write, sys_openat, sys_close）、状态获取（sys_stat, sys_fstat, sys_statx）、文件系统挂载与卸载（sys_mount, sys_umount2）、管道创建（sys_pipe）、文件描述符复制（sys_dup）以及文件控制（sys_ioctl, sys_fcntl）等。
- mm 模块及其子模块负责内存管理相关的系统调用。sys_brk 用于调整程序数据段的上界（heap），而 sys_mmap, sys_munmap, sys_mprotect 则提供了内存映射、取消映射以及修改内存区域保护属性的功能，这些是现代操作系统内存管理的核心组成部分。
- signal 模块实现了信号处理机制。它包括了信号动作的注册与修改（sys_rt_sigaction）、信号掩码的控制（sys_rt_sigprocmask）、查询未决信号（sys_rt_sigpending）、发送信号（sys_kill, sys_tkill）、从信号处理函数返回（sys_rt_sigreturn）以及备用信号栈（sys_sigaltstack）等功能，为进程间通信和异步事件处理提供了支持。
- task 模块及其子模块（clone.rs, execve.rs, exit.rs, schedule.rs）负责进程和线程的管理。它实现了进程/线程的创建（sys_clone, sys_fork）、新程序的执行（sys_execve）、进程/线程的终止（sys_exit, sys_exit_group）、调度控制（sys_sched_yield）以及任务休眠（sys_nanosleep）。等待子进程结束的功能（如 sys_wait4）也应属于此模块范畴。
- sys 模块提供了一些通用的系统信息查询调用，例如获取用户和组 ID（sys_getuid, sys_geteuid, sys_getgid, sys_getegid）以及系统名（sys_uname）。

3.2 Starry 执行流程

Starry 的执行流程建立在 ArceOS 的基础之上，并主要在 main 模块和 entry 模块中实现，图 3.3 展示了 Starry 的执行流程。

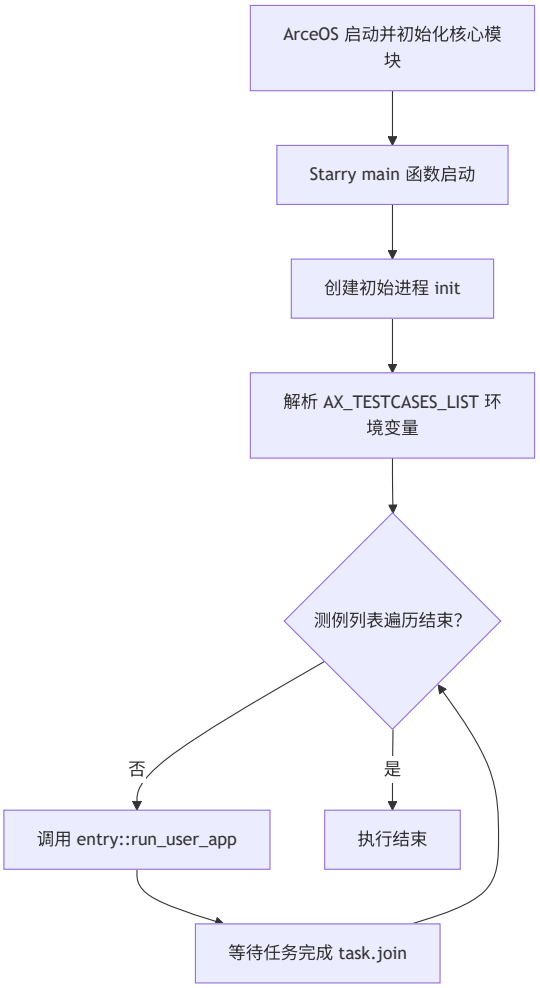

图 3.3 Starry 执行流程图

在 ArceOS 完成其引导启动和核心模块的初始化之后，系统的控制权便移交给 Starry 的主程序，即 main 函数，标志着 Starry 系统执行流程的正式启动。Starry 将待执行的用户态程序组织为一系列” 测例”。这些测例的列表通过环境变量 AX_TESTCASES_LIST 在系统启动时指定，列表中的每一项对应测例在文件系统映像中的具体路径。

Starry 采用批处理的方式执行这些测例。main 函数首先会创建一个初始进程，这是后续所有用户进程的父进程。随后，它解析 AX_TESTCASES_LIST 环境变量，获取测例路径列表。接着，系统会遍历此列表，对于每一个测例路径，它会调用 entry::run_user_app 函数来加载并执行该测例。主执行流程会同步等待当前测例执

行完毕并返回退出码后，再继续处理列表中的下一个测例，直至所有测例执行完成。

entry::run_user_app 函数详细定义了单个用户测例的加载与执行过程，其时序图如图 3.4 所示。首先，它调用 starry_core::mm::new_user_aspace_empty 函数创建一个全新的、独立的用户态地址空间。紧接着，为了确保用户程序能够正确访问内核提供的某些功能或数据（例如信号处理的 trampoline 代码），会调用 starry_core::mm::copy_from_kernel 将必要的内核映射复制到该用户地址空间，并调用 starry_core::mm::map_trampoline 映射信号处理的 trampoline。

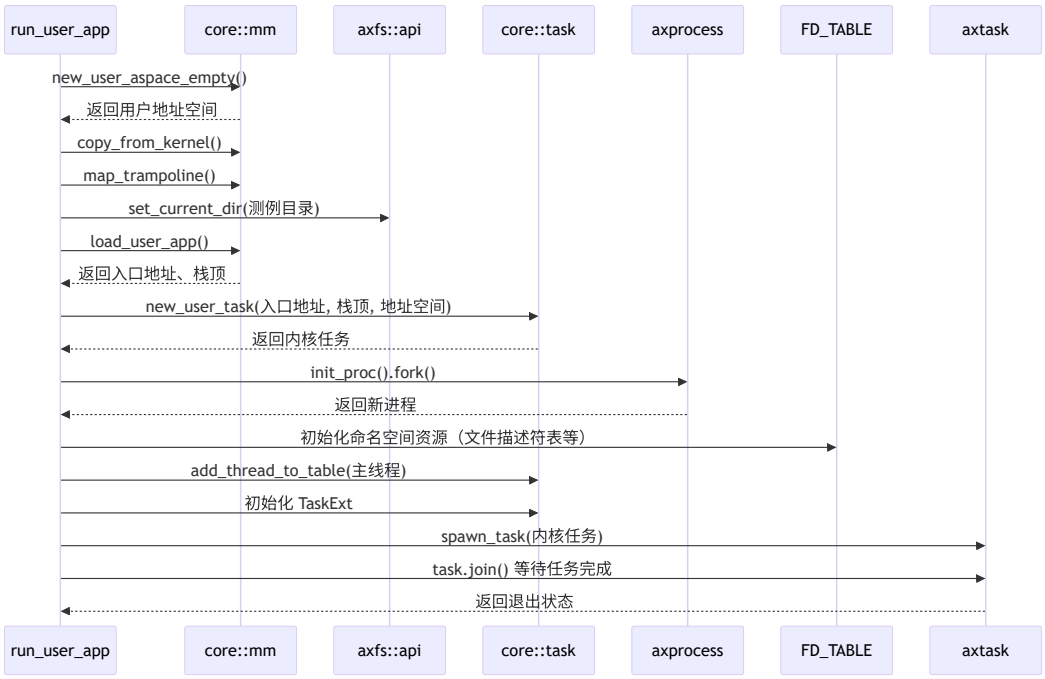

图 3.4　entry::run_user_app 时序图

随后，函数从传入的参数中提取测例的可执行文件路径，并将其所在的目录设置为当前进程的工作目录，这是通过调用 axfs::api::set_current_dir 实现的。核心的加载步骤由 starry_core::mm::load_user_app 函数完成，该函数负责解析测例的 ELF 文件格式，将其代码段和数据段加载到先前创建的用户地址空间中，并初始化用户栈，包括传递命令行参数和环境变量。此函数返回用户程序的入口点虚拟地址和用户栈顶指针。

用户程序加载完毕后，系统会创建一个用户态上下文（UspaceContext），其中包含了程序入口点和栈顶信息。基于此上下文，通过调用 starry_core::task::new_user_task 创建一个新的内核任务，该任务将负责执行用户测例。新任务的页表根会被设置为刚创建的用户地址空间的页表根。

接下来，系统会为这个新任务关联一个进程实体。通过 axprocess::init_proc().fork() 创建一个新的进程，并为其填充 ProcessData，其中包含了可执行文件路径、用户地址空间（包裹在 Arc<Mutex<...>>中以支持共享和同步）、以及用于父进程接收子进程退出信号的配置（如 Signo::SIGCHLD）。同时，为了确保新进程拥有独立的文件描述符表和当前工作目录等命名空间资源，会利用 FD_TABLE.deref_from(&process_data.ns).init_new(...) 等语句，从父进程（此处为 init 进程）的命名空间中复制并初始化这些资源。

然后，为该进程创建一个主线程，并填充 ThreadData。新创建的线程和进程会被注册到全局的线程表和进程表中，这是通过 starry_core::task::add_thread_to_table 函数完成的。内核任务也会被初始化其扩展数据 TaskExt，其中包含了对新创建线程的引用。

最后，调用 axtask::spawn_task 将这个配置完毕的内核任务提交给 ArceOS 的调度器执行。run_user_app 函数会调用 task.join() 方法，阻塞等待该用户任务执行完成，并获取其退出状态。

在用户程序执行期间，内存访问是一个关键环节。Starry 将 mm 模块中的 handle_page_fault 函数注册为页错误（PAGE_FAULT）的陷阱处理程序。当用户程序或内核在访问用户内存时（通过 starry_core::mm::is_accessing_user_memory() 判断）如果发生页错误，此函数会被调用。它会尝试通过用户地址空间的 handle_page_fault 方法来处理这个缺页（例如，按需分配物理页或处理写时复制）。如果无法处理，则判定为段错误（Segmentation Fault），记录错误信息，并通过 starry_api::do_exit 以 SIGSEGV 信号终止当前任务。

3.3 系统调用处理

系统调用（syscall）是操作系统提供给用户程序的一组接口，它允许用户程序请求操作系统内核执行特权操作。用户程序在执行过程中，往往需要访问硬件资源（如磁盘、网络、内存等）、进行文件操作、进程管理、内存分配等一系列需要特权的操作，而这些操作出于安全性和稳定性的考虑，不能直接由用户程序在用户态完成。此时，用户程序需要通过系统调用这一受控的接口，向操作系统内核发出请求，由内核代为完成相关操作。系统调用的实现通常依赖于陷入（trap）或中断机制：当用户程序发起系统调用时，处理器会切换到内核态，转而执行内核中对应的系统调用服务例程，待操作完成后再将结果返回给用户程序。通过系统调用机制，操作系统既为用户程序提供了丰富的功能接口，又有效地隔离了用户态与内核态，保证了系统的安全性和可靠性。

3.3.1 处理流程

Starry 中的系统调用是通过经典的陷入（trap）机制实现的。用户程序执行系统调用指令（如 syscall 或 ecall，具体指令取决于底层硬件架构）时，处理器会捕获这一指令并从用户态切换到内核态，触发一个陷入。ArceOS 内核接收此陷入，并根据陷入类型识别其为系统调用请求。随后，该请求被转发至 Starry 在 src/syscall.rs 中注册的统一系统调用处理入口 handle_syscall 函数。此函数接收一个唯一的系统调用编号（其值遵循特定架构的应用二进制接口 ABI 约定，例如 Linux x86_64 ABI 中的编号规则）以及最多六个 u64 类型的参数。

在接收到系统调用请求后，handle_syscall 会根据系统调用编号，将请求分发给 starry-api 的 imp 模块下各个子模块中定义的具体处理函数。这些具体实现函数会调用 ArceOS 提供的底层核心组件（如任务调度、内存分配、虚拟文件系统接口等），以完成用户程序的系统调用请求。imp 模块的各个子模块负责的部分如下：

- **imp/fs**：负责文件系统相关的系统调用。例如，sys_read 用于从文件描述符读取数据到用户缓冲区，sys_write 用于将用户缓冲区的数据写入到文件描述符指向的文件。
- **imp/mm**：处理内存管理相关的系统调用。例如，sys_mmap 用于在进程地址空间中映射一段新的内存区域，sys_munmap 用于解除之前映射的内存区域。
- **imp/task**：负责进程和线程管理相关的系统调用。例如，sys_clone 用于创建新线程或进程，sys_execve 用于在当前进程中加载并执行新的用户程序。
- **imp/signal**：实现信号处理相关的系统调用。例如，sys_rt_sigaction 用于注册或修改信号处理动作，sys_kill 用于向指定进程发送信号。
- **imp/sys**：提供系统信息和通用功能相关的系统调用。例如，sys_uname 用于获取系统名称和版本信息，sys_getuid 用于获取当前进程的用户 ID。

starry-api/imp 模块中的系统调用处理函数的返回类型为 Result<isize, LinuxError>，用于表示操作可能成功或失败的结果。若系统调用成功执行，返回 Ok(返回值)；若发生错误，则返回 Err(错误类型)。handle_syscall 会对该返回值进行统一处理：如果是 Ok，则直接将 isize 类型的返回值返回给用户程序；如果是 Err，则将错误类型（LinuxError）转换为负数形式的错误码，并返回给用户程序。这样的处理方式符合 POSIX 标准的约定，即 0 或正数表示成功，负数表示错误，绝对值为具体的错误码（如 EBADF、EINVAL 等），这些错误码在 axerrno 模块中统一定义，便于用户程序识别和处理不同的错误类型。

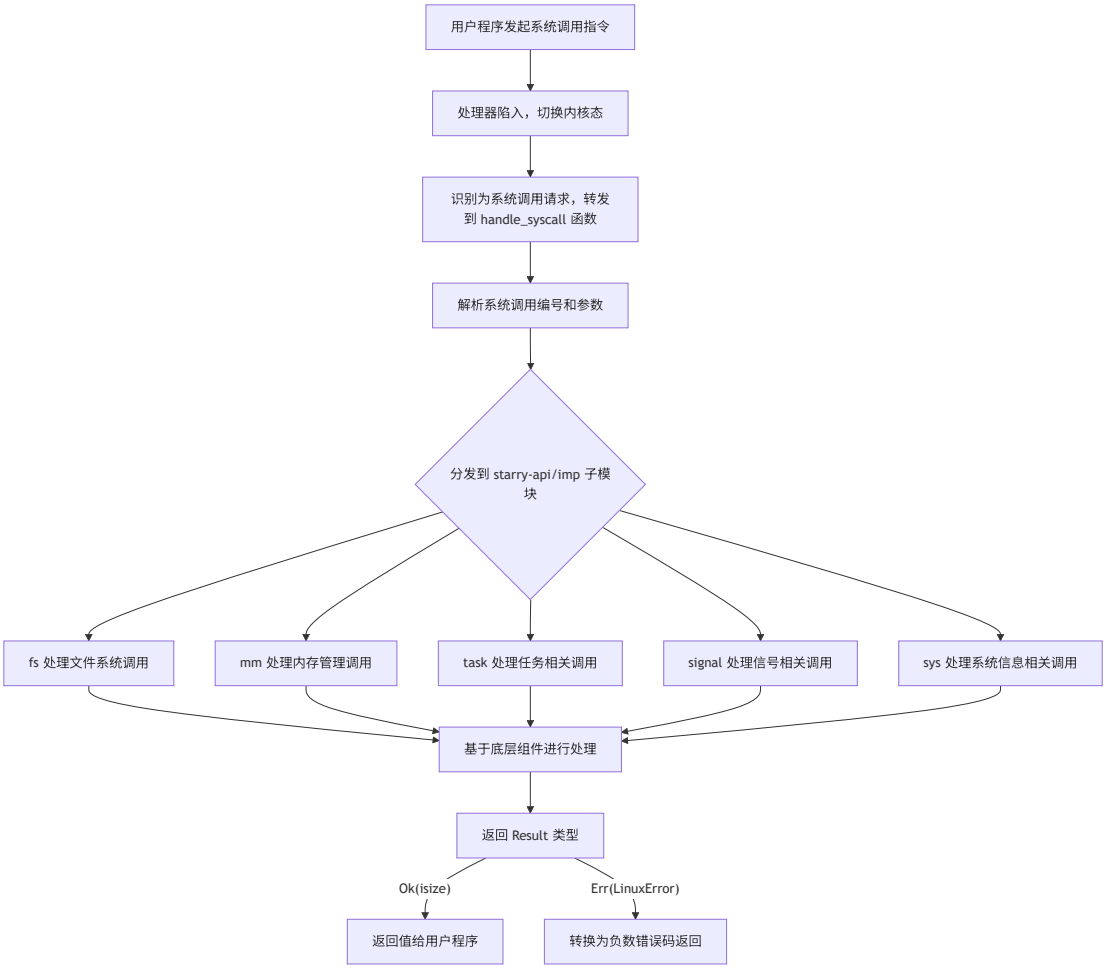

图 3.5　Starry 的系统调用处理流程图

3.3.2　用户指针

许多系统调用的参数或者期望的返回值（例如需要读写用户程序提供的大块缓冲区，或者返回复杂的结构体数据）无法仅用单个 u64 类型的寄存器参数直接传递，因此需要通过指针来传递用户地址空间中的内存地址。为了安全且高效地处理这类指向用户空间的指针，starry-api 的 ptr 模块提供了 UserConstPtr<T> 和 UserPtr<T> 两种核心数据结构。UserConstPtr<T> 用于表示指向用户地址空间的只读指针，确保内核不能意外修改用户数据；相对地，UserPtr<T> 则用于表示可写指针，允许内核向用户空间写入数据。

在各个系统调用处理函数中，从 u64 类型参数（即寄存器传入的地址值）转换而来的用户指针会首先被包装成这两种类型中的一种。这些类型封装了必要的安全性检查逻辑，例如验证所指地址是否确实位于当前进程的用户地址空间范围之内、是否满足必要的内存对齐要求，以及访问权限是否匹配（例如，不能通过

UserConstPtr 写入）。

用户指针类型提供了一系列将用户空间数据安全转换为 Rust 类型的方法，这些方法在实际访问用户内存之前会执行严格的校验，若校验失败（例如指针为 null、越界或权限不足），则会提前返回相应的错误（如 EFAULT），从而有效避免了因非法内存访问可能导致的 panic 或数据损坏，极大地保障了操作系统的整体稳定性。用户指针的常用方法及其用途如表 3.1 所示。

表 3.1　UserConstPtr/UserPtr 常用方法及使用示例

方法名	功能说明	典型使用场景
get_as_str	将用户空间以空字符结尾的 C 字符串安全读取为 Rust 字符串切片	处理文件路径参数（如 open、execve）
get_as_mut_slice	获取用户空间指定长度的可写 Rust 切片	写入数据缓冲区（如 sys_read）
get_as_ref	获取指向用户空间单个结构体的只读引用	读取结构体（如 sys_nanosleep 处理 timespec）
get_as_mut	获取指向用户空间单个结构体的可写引用	填充结构体（如 sys_fstat 处理 stat）
get	获取用户空间裸指针（需配合 unsafe 使用，调用者需保证安全）	低层结构体填充（如 sys_uname 填充 UtsName）

在与用户态程序交互时，尤其是为了兼容期望 POSIX 环境的应用程序，系统调用接口必须遵循标准的 C 语言数据结构和常量定义。starry-api 的 imp/ctypes 模块在此扮演了关键角色，它定义了大量与 Linux 系统调用接口兼容的常量（如文件打开标志 O_CREAT、O_RDONLY，内存保护标志 PROT_READ、PROT_WRITE）和 C 风格的数据结构（如 struct stat、struct timespec、struct sockaddr 等）。这些定义确保了在进行系统调用时，用户程序传递的参数和期望接收的返回数据在结构上与内核的理解一致。例如，在处理 sys_fstatat 这类需要填充 struct stat 的系统调用时，内核会先填充一个 Rust 内部表示（如用于保存文件信息的 Kstat，随后可能转换为 ctypes 中定义的 stat 结构），然后利用 UserPtr 将此结构体的内容安全地复制回用户程序指定的内存地址。因此，ptr 模块提供的 UserConstPtr 和 UserPtr 类型，对于在内核与用户空间之间正确、安全地传递这些由 ctypes 定义的复杂数据结构至关重要，它们有效地避免了直接且不安全的内存访问，保证了跨越用户态与内核态边界数据的一致性和完整性。

第 4 章　文件系统管理组件设计与实现

4.1　整体架构

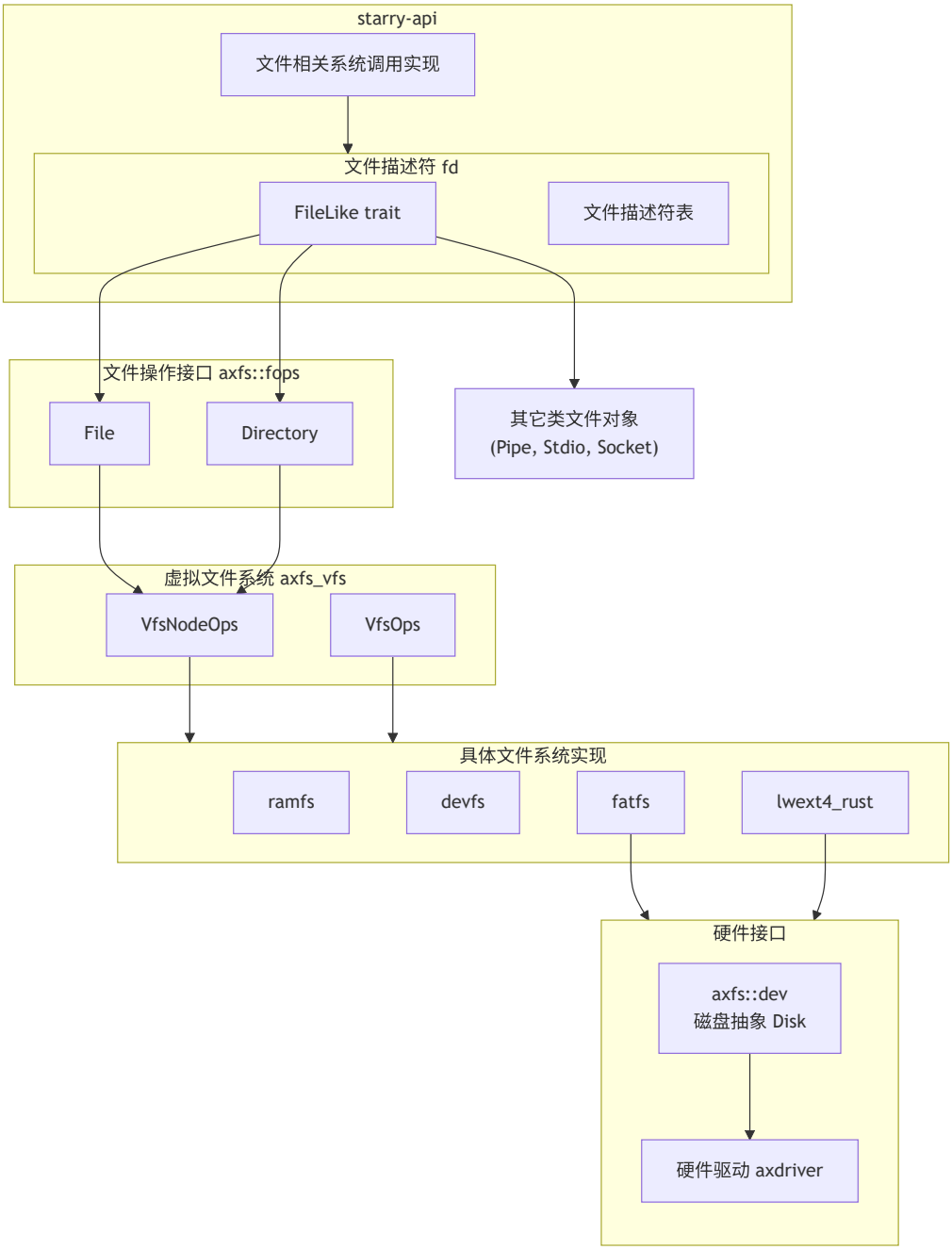

图 4.1　文件系统管理组件架构图

Starry 的文件系统管理组件的整体架构如图 4.1所示，其核心层次自上而下包括：

- **系统调用接口层 (starry-api)**: 这是用户应用程序与文件系统交互的直接入口，负责解析和处理文件相关的系统调用请求。在系统调用的实现过程中，文件描述符 (File Descriptor, FD) 是核心抽象。每个进程都拥有自己的文件描述符表，表中的条目（即文件描述符，通常是一个整数）指向内核中表示已打开的类文件的对象。为统一处理不同类型的可被” 文件化” 访问的资源，引入了通用接口 FileLike trait。无论是普通文件、目录，还是管道、标准输入输出、套接字等其他类文件对象，都通过实现 FileLike trait 来提供一致的操作行为（如读、写、关闭等）。这样，文件描述符机制便可以透明地管理这些不同类型的资源。文件相关的系统调用实现正是基于这一文件描述符机制来运作的。
- **文件操作抽象层 (axfs::fops)**: 该层位于系统调用接口之下，提供了对文件和目录的具体操作封装。它定义了 File 和 Directory 等结构体，这些结构体是对更底层虚拟文件系统节点的进一步封装，并实现了前述的 FileLike trait，从而能够被文件描述符层统一处理。同时，这些结构体内部会调用虚拟文件系统层提供的接口来执行实际的文件操作。
- **虚拟文件系统层 (axfs_vfs)**: 这是整个文件系统管理的核心抽象层，负责屏蔽不同类型文件系统的实现差异。axfs_vfs 定义了一套标准的接口，主要通过 VfsNodeOps 和 VfsOps 两个核心 trait 来实现。
  - VfsNodeOps: 定义了对单个文件或目录节点（Inode）的操作，如读 (read_at)、写 (write_at)、查找 (lookup)、创建 (create)、获取属性 (get_attr) 等。
  - VfsOps: 定义了对整个文件系统实例的操作，如挂载 (mount)、卸载 (umount)、获取文件系统状态 (statfs)、获取根目录节点 (root_dir) 等。

  通过这套统一的接口，上层模块无需关心底层具体文件系统的实现细节，实现了对不同文件系统的透明访问。
- **具体文件系统实现层**: 这一层包含了各种具体文件系统的实现，例如：
  - ramfs: 基于内存的文件系统。
  - devfs: 设备文件系统，将设备暴露为文件。
  - fatfs: FAT 格式的文件系统。
  - lwext4_rust: ext4 文件系统。

  每种文件系统都需要实现 axfs_vfs 定义的 VfsNodeOps 和 VfsOps trait，从而

能够被集成到整个文件系统框架中。

- **硬件接口层**: 该层负责与实际的物理存储设备（如硬盘、SSD）或虚拟存储设备进行交互，并为上层文件系统提供统一的块设备抽象接口。具体来说，axdriver 提供了底层的块设备驱动，而 axfs 组件中的 dev 模块则在其基础上实现了磁盘抽象 Disk，将底层多样化的硬件驱动封装起来，向上层文件系统提供统一的、基于块的读写接口，并管理读写游标等。这样，具体文件系统实现可以基于这一磁盘抽象进行开发，而无需直接处理底层硬件细节，实现了驱动与文件系统的解耦。

综上所述，Starry 的文件系统管理组件通过这种清晰的分层架构，实现了从用户接口到底层硬件驱动的逐级抽象和功能解耦。每一层都专注于特定的功能，并通过明确定义的接口与相邻层交互。这种设计不仅使得系统结构清晰，易于理解和维护，也为未来扩展新的文件系统类型或支持新的存储设备提供了良好的灵活性和可扩展性。这之中，文件操作抽象、虚拟文件系统和磁盘抽象的实现已在第二章对 ArceOS 中文件系统组件的分析中详细介绍，而本章后续将介绍四种具体的文件系统实现、文件描述符的实现和文件相关系统调用的实现。

4.2 具体文件系统实现

4.2.1 ramfs

ramfs 是一种基于内存的文件系统，将其所有数据直接存储在内存中。ArceOS 提供了 axfs_ramfs 组件作为其一种实现，并实现了 axfs_vfs 虚拟文件系统抽象层定义的接口。由于数据存储在内存中，ramfs 的读写速度非常快，但其缺点是当系统关闭或重启时，所有存储的数据都会丢失。在 Starry 中，ramfs 可用于存储临时文件、作为早期启动阶段的根文件系统，或用于测试目的。

axfs_ramfs 组件的核心结构是 RamFileSystem 以及代表文件和目录的 FileNode 和 DirNode。

- **RamFileSystem**: 代表整个 ramfs 文件系统实例，实现了 axfs_vfs 定义的 VfsOps trait。它负责管理文件系统实例，包括初始化根目录节点，并在挂载 (mount) 时处理父目录信息以正确支持”..” 导航。
- **DirNode**: 代表 ramfs 中的目录节点，实现了 axfs_vfs 定义的 VfsNodeOps trait。它内部存储其子节点（其他目录或文件节点），并负责处理如查找 (lookup)、创建 (create)、删除 (remove)、读取目录项 (read_dir) 和获取属性 (get_attr) 等目录操作。
- **FileNode**: 代表 ramfs 中的文件节点，同样实现了 axfs_vfs 定义的 VfsNodeOps

trait。它内部存储文件数据（通常为字节序列），并负责处理如读取 (read_at)、写入 (write_at)、截断 (truncate) 和获取属性 (get_attr) 等文件操作。

4.2.2 devfs

devfs (Device File System) 是一种特殊的虚拟文件系统，其主要功能是在文件系统层级中表示和管理系统中的设备。ArceOS 提供了 axfs_devfs 组件作为 devfs 的实现，并同样实现了 axfs_vfs 虚拟文件系统抽象层的接口，为用户应用程序提供了一个统一的、通过文件操作与设备进行交互的接口。这种设计将设备驱动程序的复杂性封装起来，使得设备可以像普通文件一样被访问。

axfs_devfs 的核心结构是 DeviceFileSystem 和 DirNode。

- **DeviceFileSystem**: 代表整个 devfs 文件系统实例，实现了 axfs_vfs 定义的 VfsOps trait。它管理 devfs 的根目录节点，处理挂载操作，并提供添加设备节点和创建子目录的方法（如 add 和 mkdir）。
- **DirNode**: 代表 devfs 中的目录节点，实现了 axfs_vfs 定义的 VfsNodeOps trait。它内部存储子节点（其他目录或设备节点），这些设备条目通常在系统初始化或驱动加载时静态注册。DirNode 负责节点查找、目录读取等操作。值得注意的是，用户通常不能直接创建或删除 devfs 中的节点，这些操作受限，以保护系统资源。

axfs_devfs 组件还提供了标准设备节点的实现，它们都实现了 VfsNodeOps trait：

- **NullDev**: 模拟 /dev/null 设备。写操作被丢弃，读操作立即返回文件结束符 (EOF)。
- **ZeroDev**: 模拟 /dev/zero 设备。写操作被丢弃，读操作返回零字节。

这些以及其他通过 DirNode::add 方法添加的设备节点，均作为字符设备 (VfsNodeType::CharDevice) 实现 VfsNodeOps，定义了各自的文件操作行为。

在 Starry 中，会创建一个 DeviceFileSystem 实例，并通过其 add 方法注册各种设备节点（包括 NullDev, ZeroDev 和其他驱动提供的设备）。该 devfs 实例随后被挂载到特定路径（如 /dev）。用户应用程序便可通过标准文件系统调用访问这些设备文件，与底层设备驱动交互。

4.2.3 fatfs

rust-fatfs 是一个用 Rust 语言实现的 FAT (File Allocation Table) 文件系统库。它旨在提供一个健壮、灵活且支持 no_std 环境的 FAT 文件系统实现，使其能够用于嵌入式系统以及标准操作系统环境中。该库支持 FAT12、FAT16 和 FAT32 等多种 FAT 变体。

rust-fatfs 的核心设计围绕以下几个主要模块和抽象：

- **文件系统层**: 作为与文件系统交互的主要入口，管理已挂载的 FAT 卷。负责初始化、FAT 表管理、簇分配及根目录访问。用户可通过 FsOptions 配置行为，并可使用 format_volume 格式化卷。
- **引导扇区**: 解析和构建 FAT 引导扇区 (Boot Sector) 和 BIOS 参数块 (BPB)，存储文件系统操作所需的关键参数。
- **文件分配表**: 抽象对不同类型 FAT (FAT12, FAT16, FAT32) 的操作，提供统一的簇读写、查找空闲簇、分配和释放簇链的接口。
- **目录操作**: Dir 代表目录，支持创建、打开、删除子目录和文件，以及通过 DirIter 遍历目录内容。处理长文件名 (LFN) 和短文件名 (SFN)。
- **文件操作**: File 代表打开的文件，提供读、写、寻址 (seek)、截断 (truncate) 等标准操作，并与 FAT 表交互更新元数据。
- **目录条目**: 定义目录条目（短文件名和长文件名）的结构，包含文件名、属性、时间戳、起始簇号和文件大小等信息。
- **时间和错误处理**: TimeProvider trait 允许自定义时间戳获取，Error 枚举定义了库的错误类型。

总体而言，rust-fatfs 通过模块化的设计和清晰的抽象，提供了一个功能相对完善的 FAT 文件系统解决方案。它将文件系统的高层逻辑（如文件和目录操作）与底层的 FAT 表管理和扇区读写分离开来。通过泛型和 trait，它实现了对底层存储和特定环境功能的解耦，例如 I/O 操作和时间戳获取。这使得 rust-fatfs 能够适应多种使用场景，从简单的镜像文件操作到复杂的嵌入式存储设备管理。

为了将 rust-fatfs 集成到 Starry 的虚拟文件系统框架中，我们在 ArceOS 的 axfs 组件中实现了一个适配层。该适配层主要包含以下几个部分：

- **FatFileSystem**: 此结构体实现了 VfsOps trait，代表一个挂载的 FAT 文件系统实例。它封装 fatfs::FileSystem，并配置 Starry 的 Disk (通过实现 rust-fatfs 的 I/O traits，组合为本地 IoTrait) 作为底层 I/O。它还负责根目录节点的延迟初始化。
- **FileWrapper<'a, IO: IoTrait>** 和 **DirWrapper<'a, IO: IoTrait>**: 这两个包装器分别封装 fatfs::File 和 fatfs::Dir，并为它们实现 VfsNodeOps trait。它们将 VFS 的文件和目录操作（如读写、查找、创建、删除等）转换为对底层 fatfs 对象的调用，并使用 axsync::Mutex 保证线程安全。
- **错误转换**: 定义了 as_vfs_err 辅助函数，用于将 fatfs::Error<E> 转换为 axfs_vfs::VfsError，以统一错误处理。

- **I/O 适配**: 为 axfs 中的 Disk 类型实现了 rust-fatfs 所需的 IoBase、Read、Write 和 Seek traits，使得 rust-fatfs 可以直接利用 axfs 提供的块设备抽象进行操作。

通过上述适配层，在启用对应的 feature 时，Starry 能够有效地将 rust-fatfs 库作为一个具体的 VFS 实现集成到其文件系统中，从而支持对 FAT 格式存储设备的访问。当一个 FAT 分区被挂载时，会创建一个 FatFileSystem 实例，其后的文件操作请求会通过 FileWrapper 和 DirWrapper 路由到 rust-fatfs 库进行处理。

4.2.4 lwext4_rust

lwext4_rust 是一个围绕 C 语言实现的轻量级 ext4 文件系统库 lwext4 的 Rust 封装。该库通过 Rust 的 FFI (Foreign Function Interface) 调用底层的 C 函数来执行实际的文件系统操作。

我们同样在 ArceOS 的 axfs 组件中实现了 lwext4_rust 的适配层，主要包含以下几个部分：

- **Ext4FileSystem**: 实现了 VfsOps trait，代表一个挂载的 ext4 文件系统实例。它封装了 lwext4_rust::Ext4BlockWrapper<Disk>，通过实现 KernelDevOp trait 与 lwext4 库交互。
- **FileWrapper**: 实现了 VfsNodeOps trait，代表 ext4 文件系统中的文件或目录。它封装了 lwext4_rust::Ext4File，并通过 axsync::Mutex 保证线程安全。
- **I/O 适配**: 为 Starry 的 Disk 类型实现了 KernelDevOp trait，使 lwext4_rust 库能直接使用 Starry 的块设备抽象。
- **底层交互**: 包括 Ext4BlockWrapper 用于管理文件系统实例，Ext4File 提供文件操作接口，以及通过 FFI 调用 lwext4 C 库函数。
- **错误转换**: 将 lwext4 C 库的错误码转换为 Starry VFS 使用的 VfsError 类型。

我们将 lwext4 文件系统作为 Starry 的默认文件系统，并挂载到根目录 / 下。

4.3 文件描述符模块实现

4.3.1 文件描述符

文件描述符（File Descriptor, FD）是操作系统内核为实现对各类文件对象（如普通文件、管道、目录、网络套接字等）统一管理而引入的一种抽象。文件描述符本质上是一个非负整数，用于唯一标识某个进程当前打开的文件或其他 I/O 资源。进程在打开文件或其他类文件对象时，内核会分配一个文件描述符，后续对该对象的所有操作（如读写、控制等）均基于该文件描述符进行。

文件描述符的作用　文件描述符机制为操作系统和应用程序带来了如下优势：

- **统一抽象接口**：通过文件描述符，操作系统为各种 I/O 资源提供了统一的访问接口。应用程序可以使用相同的 API（如 read、write、close 等）操作不同类型的资源，简化了编程模型。
- **资源管理与隔离**：每个进程拥有独立的文件描述符表，内核能够方便地跟踪和管理进程打开的所有文件及其状态（如读写位置、访问模式等），并在进程退出时自动回收相关资源，保证了资源的隔离性和安全性。
- **灵活性与扩展性**：文件描述符机制支持 I/O 重定向、管道、I/O 多路复用（如 select、poll、epoll）等高级操作，是实现进程间通信和高性能 I/O 的基础。

文件描述符的实现方式　在类 Unix 操作系统中，文件描述符的实现通常包括以下几个核心部分：

- **文件描述符表**：每个进程在内核中维护一个文件描述符表（一般为数组或哈希表），表项的下标即为文件描述符，内容为指向内核全局文件对象的指针。
- **全局文件对象**：内核为每个打开的文件维护一个全局文件对象（如 Linux 中的 struct file），记录文件的状态、读写偏移、访问权限、指向具体文件系统节点的指针等信息。
- **引用计数**：多个文件描述符（如通过 dup、fork 等操作）可以指向同一个全局文件对象，内核通过引用计数管理其生命周期，确保资源的正确释放。
- **操作分发机制**：文件对象内部包含一组操作函数指针（如 read、write、ioctl 等），不同类型的文件（普通文件、目录、管道、套接字等）实现各自的操作逻辑，实现多态分发。

文件描述符表（FD Table）　文件描述符表是操作系统内核为每个进程维护的一张表，用于管理进程可以访问的所有类文件对象。每当进程打开一个文件或创建一个新的 I/O 资源时，内核所返回的文件描述符实际上是在该进程的文件描述符表中分配的空闲条目的下标。当进程进行后续的读写、控制等操作时，内核均会根据文件描述符在表中的位置找到对应的 FileLike 对象，并进行操作。

文件描述符表的主要作用包括：

- **资源隔离与管理**：文件描述符表为每个进程单独维护，确保进程只能访问自身打开的文件等 I/O 资源，实现了进程间的资源隔离和安全管理。
- **高效映射与查找**：文件描述符表通常以数组或哈希表实现，文件描述符作为下标能够快速定位到对应的文件对象，提升了 I/O 操作的效率。
- **支持重定向与继承**：文件描述符表支持文件描述符的复制（如 dup、dup2 系

统调用）和在进程创建（如 fork）时的继承，便于实现输入输出重定向和进程间通信。

- **统一资源管理接口**：通过文件描述符表，内核能够以统一的方式管理和追踪进程打开的所有类文件对象（如普通文件、管道、套接字等），便于资源的分配与回收。

在 Starry 的实现中，我们在 starry_api 中的 fd 模块实现了文件描述符相关的功能，从而为实现和完善文件相关的系统调用打下了基础。具体来说，我们定义了 FileLike trait，作为类文件对象的统一抽象，并实现了文件描述符表 FD_TABLE，作为每个进程的全局资源，用于管理进程打开的文件描述符。

4.3.2 FileLike 接口设计与实现

为了实现 Starry 中系统调用对类文件对象抽象的需求，从而为文件、目录、标准输入输出、管道和网络套接字等不同对象提供统一的接口，我们在 starry_api 模块中定义了 FileLike trait，为不同的类文件创建各自的结构体，并实现了 FileLike trait。如对于文件系统来说，可以认为 Filelike trait 是 axfs 组件中 File 类型的更高层次的抽象，它基于文件的低层接口定义了一系列方法，包括读取、写入、获取文件状态、设置非阻塞模式等。

FileLike 接口定义 FileLike trait 定义了一组类文件操作的方法：

- **read**：从类文件对象中读取数据，将读取到的数据填充到传入的缓冲区（buf）中。返回值为实际读取的字节数，或者错误信息。不同类型的文件对象（如普通文件、管道、套接字等）可以有各自的读取实现。
- **write**：将缓冲区（buf）中的数据写入到类文件对象中。返回值为实际写入的字节数，或者错误信息。不同类型的文件对象可以有不同的写入逻辑。
- **stat**：获取类文件对象的状态信息，并以 Kstat 结构体的形式返回。
- **into_any**：将类文件对象转换为 Any 类型（trait 对象），以便在运行时进行类型转换（downcast），实现类型安全的动态转换。
- **poll**：检查类文件对象当前的可读、可写等状态，通常用于实现 I/O 多路复用（如 select、poll、epoll 等机制），以便高效地等待文件状态变化。
- **set_nonblocking**：设置类文件对象的阻塞/非阻塞模式。非阻塞模式下，I/O 操作不会因为资源不可用而阻塞进程，而是立即返回，这对于实现高性能 I/O 和事件驱动编程非常重要。
- **from_fd**：从文件描述符创建类文件对象，具体实现由 get_file_like 函数完成，将在后文中说明文件描述符表的实现时介绍。

- **add_to_fd_table**：将类文件对象添加到文件描述符表中，具体实现由 add_file_like 函数完成，将在后文中说明文件描述符表的实现时介绍。

FileLike 接口实现 在 starry_api 的 fd 模块中，我们分别对传统文件、目录、标准输入输出、管道和 TCP/UDP 套接字实现了 FileLike trait。

- **普通文件**：普通文件由 File 类型实现 FileLike 接口。File 类型封装了对底层文件系统的操作，提供文件的读写、状态获取等功能。它使用互斥锁保护内部的文件对象（axfs::fops::File），以确保线程安全。文件的读写操作通过调用底层文件系统的相应方法实现，文件状态的获取则通过返回一个包含文件元数据的结构体来实现。
- **目录**：目录由 Directory 类型实现 FileLike 接口。与普通文件不同，目录不支持直接的读写操作，只是提供目录遍历和文件访问的功能，而在试图进行读写操作时会直接返回错误 EBADF。目录的状态获取返回一个表示目录权限和类型的结构体。目录的文件描述符主要用于 AT 系列系统调用时指定相对路径的起始目录，所以 Directory 类型中最常用的方法为 path，即返回目录的绝对路径。
- **标准输入输出**：标准输入输出由 Stdin 和 Stdout 类型实现 FileLike 接口。Stdin 类型提供从控制台读取输入的功能，使用缓冲读取来提高效率，Stdout 类型负责将输出写入控制台。它们通过底层的 axhal 组件所提供的函数 axhal::console::read_bytes 和 axhal::console::write_bytes 来实现实际的读写操作。
- **管道**：管道由 Pipe 类型实现 FileLike 接口。管道实现了进程间通信，使用环形缓冲区存储数据。Pipe 类型提供读写操作，其中读操作从缓冲区中读取数据，写操作将数据写入缓冲区。管道的状态获取返回一个表示管道权限的结构体。管道的实现还包括对缓冲区状态的管理，以支持阻塞和非阻塞的读写操作。
- **TCP/UDP 套接字**：套接字由 Socket 类型实现 FileLike 接口，支持 TCP 和 UDP 两种协议。Socket 类型基于 axnet 组件提供的 UdpSocket 和 TcpSocket 类型实现，提供网络通信的基本操作，包括数据的发送和接收、连接的建立和关闭等。TCP 套接字支持连接的监听和接受，UDP 套接字支持数据报的发送和接收。套接字的状态获取返回一个表示套接字权限的结构体。

4.3.3 文件描述符表 FD_TABLE 实现

在 starry_api 的 fd 模块中，我们通过一个全局的静态资源 FD_TABLE 实现了文件描述符表，其定义如代码4.1所示。

代码 4.1　FD_TABLE 的定义

```
def_resource! {
    pub static FD_TABLE:
    ResArc<RwLock<FlattenObjects<Arc<dyn FileLike>,
    AX_FILE_LIMIT>>> = ResArc::new();
}
```

类型细节　从定义代码中可以看出，文件描述符表 FD_TABLE 的类型为 ResArc<RwLock<FlattenObjects<Arc<dyn FileLike>, AX_FILE_LIMIT> > >

- **ResArc**：ResArc 是 ArceOS 的 axns 组件提供的用于管理资源的智能指针，它提供了引用计数和线程安全的共享所有权，并且支持资源的懒加载，即在进程第一次使用当前资源时才进行初始化。
- **RwLock**：RwLock 是用于实现读写锁的同步原语，它允许多个读者线程同时读取数据，但只允许一个写者线程独占写入，这符合文件描述符表的读写操作需求。
- **FlattenObjects**：FlattenObjects 是一个用于管理固定数量对象的容器（在这里设定为 AX_FILE_LIMIT，即单个进程可打开的最大文件数，通常定义为 1024），它基于 bitmap 标识每个位置的对象是否存在，从而可以高效地进行元素访问和分配新的元素。

全局资源实现　FD_TABLE 属于一种全局资源，在每个进程中只有一个唯一的对象，所以我们在其定义中使用了 axns 提供的宏 def_resource! 进行声明。axns 是 ArceOS 的命名空间组件，它提供了一套进程间独立的资源空间机制。在使用宏声明全局资源时，axns 将其加入到命名空间中全局资源列表中的一个固定偏移位置，每个进程将有一个指针指向其拥有的命名空间，当需要访问该资源时，只需使用宏，组合命名空间指针和偏移量，即可不损耗性能地访问全局资源。使用 axns 提供的机制，可以方便、高性能地实现进程间资源的隔离和共享。

文件描述符表实现　我们实现了几个函数，用于对当前的文件描述符表进行操作：

- **add_file_like**：该函数负责将实现了 FileLike 接口的对象添加到文件描述符表中。其实现通过获取表的写锁（RwLock），调用 FlattenObjects 的 add 方法分配新的文件描述符。若表已满（达到 AX_FILE_LIMIT 限制），则返回 EMFILE 错误。成功时返回新分配的文件描述符（整数索引），该值通过 FlattenObjects 的内部分配策略生成，通常为最小可用索引。

- **get_file_like**：通过文件描述符（整数索引）查找对应的文件对象。该函数首先获取表的读锁，使用 FlattenObjects 的 get 方法进行索引查找。若索引无效或对应位置为空，则返回 EBADF 错误。成功时返回 Arc<dyn FileLike> 的克隆引用，确保对象生命周期与使用场景解耦。此过程通过 Arc 的引用计数机制保证对象在表内外共享时的安全性。
- **close_file_like**：关闭文件并释放相关资源。该函数获取表的写锁后，调用 FlattenObjects 的 remove 方法移除指定索引处的条目。若索引无效，返回 EBADF 错误。成功移除后，通过 ResArc 的自动引用计数机制触发文件对象的析构（若无其他引用）。此操作会隐式触发文件对象的 Drop 实现，完成底层资源释放（如关闭套接字、刷新文件缓冲区等）。

除此之外，为了实现默认的标准输入输出和标准错误输出所对应的文件描述符，我们还实现了一个初始化函数。通过将该函数声明为 #[ctor_bare::register_ctor] 属性，可以实现进程启动时的自动调用，从而在进程启动时初始化文件描述符表，向其中添加标准输入输出和标准错误输出。

4.4 文件相关系统调用实现

4.4.1 文件相关系统调用

Linux 中包含大量的文件相关的系统调用，这些系统调用为用户程序提供了访问和操作文件系统的标准接口，可以分为：基本文件操作类、文件描述符管理类、文件属性和元数据操作类、目录和路径操作类、文件系统操作类和 IO 多路复用类。为了实现对 Linux 应用的兼容，Starry 需要实现这些系统调用的绝大部分。

表 4.1 列出了 Linux 的基本文件操作系统类调用，这些调用主要用于对文件内容的读写操作，包括打开、关闭文件，以及各种读写方式。

表 4.1　Linux 基本文件操作系统类调用

系统调用	功能描述
OPEN	打开或创建文件，返回文件描述符
CREAT	创建新文件，如果已存在则截断为零长度
CLOSE	关闭文件描述符，释放相关资源
READ/WRITE	从文件读取或写入数据
LSEEK	改变文件读写位置

续表 4.1　Linux 基本文件操作系统类调用

系统调用	功能描述
READV/WRITEV	从/向多个缓冲区读写数据
PREAD64/PWRITE64	在指定位置读写数据，不改变文件偏移量
SENDFILE64	在文件描述符间直接传输数据
COPYFILERANGE	在文件间复制数据
FTRUNCATE64	截断文件到指定长度

表 4.2 列出了 Linux 的文件描述符管理类系统调用，这些调用用于管理文件描述符的生命周期和属性，以及创建特殊的文件描述符用于进程间通信。

表 4.2　Linux 文件描述符管理类系统调用

系统调用	功能描述
DUP/DUP2/DUP3	复制文件描述符
PIPE/PIPE2	创建管道用于进程间通信
PIDFD_OPEN	获取进程文件描述符
EVENTFD/EVENTFD2	创建事件通知文件描述符
MKNOD	创建特殊文件
FCNTL64	执行文件描述符控制操作

表 4.3 列出了 Linux 的文件属性和元数据操作类系统调用，这些调用用于查询和修改文件的属性信息，如权限、所有者、时间戳等。

表 4.3　文件属性和元数据操作类系统调用

系统调用	功能描述
STAT/FSTAT/LSTAT	获取文件状态信息
STATX	获取扩展的文件状态信息
CHMOD/CHOWN	修改文件权限和所有者
SYNC/FSYNC	同步文件系统或单个文件到磁盘

表 4.4 列出了 Linux 的目录和路径操作类系统调用，这些调用用于管理文件系统的目录结构，包括创建、删除目录，以及在目录中移动和重命名文件。

表 4.4　目录和路径操作类系统调用

系统调用	功能描述
MKDIR/RMDIR	创建/删除目录
UNLINK	删除文件或符号链接
RENAME	重命名或移动文件
ACCESS	检查文件访问权限
READLINK	读取符号链接
GETCWD	获取当前工作目录
CHDIR	改变当前工作目录
GETDENTS64	读取目录项

文件系统操作系统调用用于管理整个文件系统，如挂载、卸载文件系统，以及获取文件系统信息。

表 4.5　文件系统操作类系统调用

系统调用	功能描述
MOUNT/UNMOUNT	挂载/卸载文件系统
STATFS	获取文件系统信息
IOCTL	设备控制操作

IO 多路复用系统调用用于同时监视多个文件描述符的状态变化，是实现高性能 IO 的重要机制。

表 4.6　IO 多路复用类系统调用

系统调用	功能描述
SELECT/PSELECT6	监视文件描述符状态变化

续表 4.6　IO 多路复用类系统调用

系统调用	功能描述
POLL/PPOLL	改进的文件描述符监视机制
EPOLL 系列	高效的事件驱动 IO 多路复用机制

此外，许多系统调用都有支持相对路径的 AT 变体版本，如 OPENAT、MKDIRAT、UNLINKAT 等，这些系统调用可以基于某个目录的文件描述符，使用相对路径解析需要操作的文件。

4.4.2　实现细节

本节将阐述我们已在 Starry 中实现的与文件系统相关的系统调用。出于篇幅考虑，我们将只介绍部分有代表性的系统调用的实现细节。

4.4.2.1　sys_getdents64

功能　sys_getdents64 用于从一个打开的目录中读取目录项。它是实现如 ls 等目录列表命令的基础。此调用以一种与具体文件系统无关的方式，允许用户程序获取目录内容。其名称中的”64” 表明它使用 64 位的偏移量和 inode 号，以支持大文件系统和大量文件。

参数

- fd (i32): 一个指向已打开目录的文件描述符。
- buf (UserPtr<u8>): 一个指向用户空间缓冲区的指针，内核将把读取到的目录项（struct linux_dirent64 结构序列）填充到此缓冲区。
- len (usize): 用户空间缓冲区 buf 的大小（字节数）。

执行流程　sys_getdents64 首先对用户提供的缓冲区进行校验，确保其有效且大小足够。随后，它从给定的文件描述符中获取内部的目录对象。接着，系统迭代读取底层文件系统的目录条目，将每个条目的信息（如 inode 号、名称、类型和在目录流中的偏移量）格式化为标准的 linux_dirent64 结构，并将其连同文件名一起写入用户缓冲区。此过程会持续进行，直到缓冲区满或目录中所有条目均被读取。内核会维护目录流的当前读取偏移量，以便后续调用可以从上次结束的位置继续。最终返回成功写入用户缓冲区的总字节数。

4.4.2.2 sys_openat

功能 sys_openat 用于打开或创建一个文件，并返回一个新的文件描述符。其 at 后缀表明它可以相对于一个目录文件描述符来解析路径名，增强了路径操作的灵活性和安全性。

参数

- dirfd (c_int): 目录文件描述符。用于解析相对路径，或在路径为绝对路径时被忽略。特殊值 AT_FDCWD 表示相对于当前工作目录。
- path (UserConstPtr<c_char>): 指向以空字符结尾的文件路径字符串的用户空间指针。
- flags (i32): 控制文件打开方式的标志位组合，如 O_RDONLY, O_CREAT 等。
- mode (__kernel_mode_t): 当 flags 中包含 O_CREAT 标志时，指定新建文件的权限模式。

执行流程 sys_openat 开始时，会从用户空间安全地获取路径字符串，并将传入的整数型 flags 和 mode 参数解析为内部更易于处理的打开选项结构。接着，根据 dirfd 的值和路径字符串的类型（绝对或相对）确定路径解析的基准目录。然后，调用底层文件系统接口（通过 axfs 模块）尝试打开或创建指定的文件。如果操作成功，系统会创建一个代表该打开文件的内部对象（如 File 或 Directory），并将其添加到当前进程的文件描述符表中，返回新分配的文件描述符给用户程序。若操作失败（例如文件不存在且未指定创建，或权限不足），则返回相应的错误码。

4.4.2.3 sys_read

功能 sys_read 用于从一个打开的文件描述符所指向的文件中读取数据。

参数

- fd (c_int): 一个已打开并具有读取权限的文件描述符。
- buf (UserPtr<u8>): 一个指向用户空间缓冲区的指针，内核将把从文件中读取到的数据填充到此缓冲区。
- count (usize): 请求读取的最大字节数，也即用户缓冲区 buf 的大小。

执行流程 该调用首先验证文件描述符 fd 的有效性，并获取与之关联的内部文件对象（实现了 FileLike trait）。接着，它从用户空间安全地获取目标缓冲区的引用。然后，调用文件对象的 read 方法，尝试从文件当前偏移量开始读取最多 count 字节

的数据到内核的一个临时缓冲区或直接到用户缓冲区（取决于具体实现和优化）。读取完成后，将实际读取到的数据复制到用户提供的 buf 中，并返回实际读取的字节数。如果到达文件末尾，返回 0。若发生错误（如 I/O 错误、文件描述符无效等），则返回相应的负错误码。

4.4.2.4 sys_sendfile

功能 sys_sendfile 用于在两个已打开的文件描述符之间高效地传输数据，通常用于将数据从一个文件直接发送到另一个文件（如从磁盘文件发送到 socket），避免了用户空间缓冲区的中转，提高了数据传输效率。

参数

- out_fd (c_int): 目标文件描述符，数据将被写入到该文件（如 socket 或普通文件）。
- in_fd (c_int): 源文件描述符，数据将从该文件读取。
- offset (UserPtr<u64>): 指向源文件偏移量的用户空间指针。如果为 NULL，则从当前文件偏移量读取，否则从指定偏移量读取并更新该偏移量。
- len (usize): 需要传输的最大字节数。

执行流程 sys_sendfile 首先验证输入参数的有效性，并通过 get_file_like 获取源和目标文件描述符对应的内部文件对象。随后判断 offset 是否为 NULL：

- 若 offset 非 NULL，则将 in_fd 解析为普通文件（File），并从指定偏移量开始读取数据，并在读取后更新 offset 的值。
- 若 offset 为 NULL，则直接从 in_fd 的当前偏移量读取数据。

数据传输通过内部的 do_sendfile 辅助函数实现。该函数循环从源文件读取数据块，并立即写入目标文件，直到读取完毕、写入不完整或达到指定长度。整个过程无需将数据复制到用户空间，提升了效率。

操作成功时，返回实际传输的字节数。若发生错误（如文件类型不支持、权限不足、I/O 错误等），则返回相应的负错误码。

4.4.2.5 sys_mount

功能 sys_mount 用于将一个文件系统挂载到指定的挂载点。在 Starry 的当前实现中，此功能有所简化，主要支持”vfat” 类型，并且主要以记录逻辑关联的形式实现文件挂载。

参数

- source (UserConstPtr<c_char>): 指向源设备路径或源目录路径的指针。
- target (UserConstPtr<c_char>): 指向挂载点目录路径的指针。
- fs_type (UserConstPtr<c_char>): 指向表示文件系统类型的字符串的指针。
- flags (i32): 挂载标志位。
- _data (UserConstPtr<c_void>): 指向文件系统特定挂载选项数据的指针 (当前被忽略)。

执行流程 sys_mount 首先从用户空间安全地获取源路径、目标路径和文件系统类型字符串。它会进行一系列有效性检查，包括文件系统类型是否为支持的"vfat"，以及目标挂载点路径是否存在且未被其他文件系统" 挂载"。Starry 的" 挂载" 操作主要是将源路径和目标挂载点路径作为一个记录添加到一个全局的、受保护的已挂载文件系统列表中。如果所有检查通过且记录成功添加，则返回成功；否则返回错误码。

4.4.2.6 sys_pipe

功能 sys_pipe 用于创建一个管道（pipe），这是一种单向的字节流通信机制，返回两个文件描述符：一个用于读取，一个用于写入。

参数

- fds (UserPtr<[c_int; 2]>): 一个指向用户空间数组的指针，内核将在此数组中存入读取端和写入端的文件描述符。

执行流程 该调用首先安全地获取用户提供的用于存储文件描述符的数组引用。接着，它在内核中创建一个新的管道对象，该对象内部包含一个共享的缓冲区以及用于同步读写操作的机制，并分离出管道的读取端和写入端。随后，系统分别为管道的读取端和写入端分配新的文件描述符，并将这两个文件描述符添加到当前进程的文件描述符表中。最后，将这两个新分配的文件描述符（通常读取端在前，写入端在后）写入用户提供的数组中。如果任一步骤失败（如文件描述符表已满），则会进行适当的清理（如关闭已分配的一端）并返回错误。

4.4.2.7 sys_statx

功能 sys_statx 用于获取文件的扩展元数据信息，并将其存储在用户提供的 struct statx 结构中。它提供了比传统 stat 系列调用更灵活的方式来指定获取哪些信息和

控制路径解析行为。

参数

- dirfd (c_int): 目录文件描述符，用于解析相对路径。
- path (UserConstPtr<c_char>): 指向文件路径字符串的用户空间指针。
- flags (u32): 控制路径解析和操作行为的标志位，如 AT_SYMLINK_NOFOLLOW, AT_EMPTY_PATH。
- _mask (u32): 指定希望填充 struct statx 中哪些字段的位掩码 (Starry 当前忽略此参数)。
- statxbuf (UserPtr<statx>): 指向用户空间 struct statx 结构体的指针。

执行流程 sys_statx 首先从用户空间安全地获取路径字符串和目标 statx 缓冲区的引用。根据 flags 参数和路径字符串是否为空，确定操作的目标：如果设置了 AT_EMPTY_PATH 且路径为空，则目标是 dirfd 本身；否则，目标是通过 dirfd 和 path 解析得到的具体文件或目录。系统随后获取目标文件的元数据结构体（Kstat）。这可能涉及通过文件描述符直接调用内部对象的 stat 方法，或者通过解析路径名后打开文件/目录再获取其状态。获取到元数据后，系统将其转换为标准的 struct statx 格式，并填充到用户提供的缓冲区中。在目前 Starry 的实现中，会尝试填充所有支持的字段，而忽略 _mask 参数。

第 5 章　文件系统组件开发与测试

5.1　课题开发过程

5.1.1　开发环境与调试方法

如前文所述，本课题的开发工作主要采用 Rust 编程语言实现。笔者的开发环境搭建在运行 Ubuntu 20.04 的 Windows Subsystem for Linux 2 (WSL2) 之上。为了提升开发效率与代码质量，笔者主要使用 Visual Studio Code (VS Code) 作为集成开发环境。VS Code 配合 rust-analyzer 插件，可以实现对 Rust 代码的智能补全、语法高亮、类型推断和跳转等功能，极大地优化了开发体验。

为了确保构建与测试过程的一致性及可复现性，我们广泛采用了 Docker 容器化技术，基于统一的配置文件构建特定的 Docker 镜像，从而实现了对代码编译和运行环境的统一管理。通过在 VSCode 中远程连接 Docker 容器，我们可以在容器内直接进行代码编辑、调试和构建。这样便保证了开发环境与实际运行环境的一致性。

为了支持多人协作开发，我们采用了 Git 作为版本控制工具。所有代码均托管于 GitHub 和 GitLab 等平台，通过分支管理、合并请求（Pull Request/Merge Request）等机制进行协作开发。此外，我们还利用 GitHub 提供的持续集成（CI）服务，在线自动对代码进行风格检查和正确性测试。这些措施确保了开发进度的同步与代码质量的提升。

在调试方面，我们初期常通过引入日志语句（例如 axlog 提供的 debug! 或 info! 宏）来追踪执行流程和观察变量状态。针对更深层次的问题分析与故障排查，我们则采用了 GNU 调试器（GDB）。通过在 QEMU 模拟环境中配置 GDB，我们能够设置断点、单步执行代码，并实时检查内存与寄存器状态。为了优化调试体验，我们在编译时会选择开启调试符号，从而为 GDB 提供更丰富的调试信息。

5.2　开发日程

本课题的开发工作可以分为以下几个主要阶段：

2024 年 12 月，主要进行了 Rust 编程语言及相关工具链的学习，搭建了基于 WSL2 和 Docker 的开发与测试环境，并成功实现了基座系统 ArceOS 的编译运行。期间还调研了操作系统比赛的测例，明确了以对 ArceOS 进行宏内核扩展的方式，支持更多 Linux 系统调用的目标，并开始分析所需实现的系统调用。

2025 年 1 月至 2 月中旬，确定了笔者在设计和完善文件系统管理组件方面的目标，并开始分析所需实现的系统调用。同时，进一步完善了 Docker 开发环境，解决了在不同环境下编译和运行内核时的配置与兼容性问题，实现了对已有的 Starry 系统的复现，并基于测例开展了初步的自动化测试。

2025 年 2 月下旬至 3 月上旬，实现和调试修复了文件系统相关的核心系统调用，如 openat、read、fstat、mkdir 等。并在 RISC-V 平台上通过了大部分 basic 测例。同时，开始在 LoongArch 等其他架构上进行测试和适配。

2025 年 3 月中旬至 3 月下旬，开始针对 libctest 实现更复杂的系统调用，以及配合团队成员优化 Starry 的实现架构。这个阶段中，在 Github 仓库中搭建了持续集成（CI）环境，实现了自动化测试和测例打分，可以直观地看到当前的测例通过情况。

2025 年 4 月至 5 月，工作重心转向提升系统兼容性和稳定性，发现和修复了更多系统调用实现中的问题，以及建立和完善 Starry 和 ArceOS 的技术文档，为其他开发者提供参考。

5.3 系统调用实现过程：以 utimensat 为例

utimensat 系统调用的作用是以纳秒的精度修改文件的最后访问时间（atime）和最后修改时间（mtime），是 utime 系统调用的增强版本。这个系统调用的实现过程，比较典型地反映了我们在实现内核功能扩展时遇到的实际问题。在项目初期，ArceOS 的虚拟文件系统（VFS）层并没有对这些时间戳属性做精确的管理。最简单的做法是采用” 伪实现”，也就是让系统调用表面上返回成功，但实际上并不修改底层文件系统的元数据。这种方式在一些早期或者简化的操作系统实现（比如旧版本的 ArceOS 宏内核扩展 Starry-Old）中比较常见。不过，为了让操作系统能为用户态程序（比如通过 fstat 系统调用）提供准确的文件时间信息，我们决定 Starry 的整个文件系统进行实际的功能增强。

为此，我们首先需要扩展 VFS 的核心组件 axfs_crates。具体来说，就是在描述 VFS 节点属性的 VfsNodeAttr 结构体中增加 atime 和 mtime 字段。同时，定义 VFS 节点操作的 VfsNodeOps Trait 也要增加获取和设置这些时间戳的方法。在实际开发中，我们一开始尝试修改了一个非官方的 VFS 分支仓库，但很快遇到了依赖库（比如 axerrno）版本不兼容的问题。为了保证系统的稳定和依赖的一致，后续我们把开发工作转移到了 ArceOS 官方维护的 axfs_crates 仓库的分支上。

VFS 底层能力增强后，还需要把这些新功能逐步暴露给上层模块。这包括对 ext4 文件系统实现的 lwext4_rust 模块、axfs 中的文件操作抽象层，以及 starry-api

的文件描述符模块 fd 进行适配。例如，在 fd 模块中，对类文件对象进行抽象的 FileLike Trait 也增加了操作时间戳的新方法。

在适配具体文件系统实现（比如基于内存的 ramfs）时，我们遇到了一些和 Rust 所有权、可变性相关的问题。因为 VFS 节点通常是通过不可变引用（VfsNodeRef）访问的，直接修改 ramfs::FileNode 里的时间戳字段就变得比较困难。为了解决这个问题，我们参考了 ramfs 内部管理文件内容时用到的 RwLock（读写锁）机制，把 atime 和 mtime 封装到一个新的 Metadata 结构体里，并用 RwLock 保护起来。这样，即使只有 FileNode 的不可变引用，也能安全地修改它内部的时间戳数据。

当 VFS 层具备了时间戳管理能力后，utimensat 系统调用的上层逻辑就可以实现了。这个函数主要负责解析用户态传递的参数，包括目录文件描述符 dir_fd、目标文件路径 path，以及一个包含两个 timespec 结构体的数组，分别指定新的访问时间和修改时间。timespec 结构体中的纳秒字段 tv_nsec 有特殊含义：UTIME_NOW 表示把对应时间戳设置为当前系统时间，UTIME_OMIT 表示不修改该时间戳。对于 UTIME_NOW，内核会通过 axhal::time::monotonic_time_nanos() 获取当前的高精度时间。系统调用的主要逻辑会根据 dir_fd 的值判断操作对象：如果 dir_fd 是有效的文件描述符，就直接操作已打开的文件；否则就根据 path 打开目标文件再进行操作。无论哪种情况，最后都会调用从 VFS 层逐层向上暴露的接口来更新文件的 atime 和 mtime。

这样完成 utimensat 的实现后，我们不仅能通过相关的功能测试，也为 fstat 等依赖文件时间戳的系统调用提供了准确的数据支持。在过程中，笔者还注意到，在涉及修改外部依赖库（比如 axfs_crates）的多仓库协作开发中，主项目（比如 starry 及其.arceos 子模块）要及时执行 cargo update 命令来同步最新的依赖变更。否则，在持续集成（CI）环境下，由于依赖版本不一致，很容易导致编译失败，这也是开发过程中需要特别注意的问题。

5.4 测试环境与测试用例

本项目的测试工作主要基于操作系统大赛官方发布的系列测例[①]，这些测例旨在从功能、兼容性及稳定性等多个维度对操作系统内核进行全面考察，涵盖了从基础系统调用到复杂应用程序的多个层面，共分为 12 组，包括 basic、busybox、lua、libctest、iozone 等，并且均支持在不同架构（包括 Riscv64、LoongArch64、Aarch64 和 X86_64）及不同 C 运行时库（包括 glibc 和 musl）上运行。

每种测试题目通常配有详尽的说明文档，核心内容包括题目介绍、编译流程、

① https://github.com/oscomp/testsuits-for-oskernel/tree/pre-2025

样例输出以及评分标准。题目介绍部分主要说明测试的基本内容和考查重点。编译流程则详细列出了如何将源代码构建为符合竞赛要求的可执行文件，通常需要针对不同的硬件架构（如 RISC-V、LoongArch）及 C 运行时库（如 glibc、musl）分别编译，并通过官方 Docker 镜像环境保证编译环境的一致性。样例输出部分会给出具有代表性的标准输出，并对其含义进行解释，便于参赛者理解测试过程及预期结果。评分标准则明确了如何依据测试程序的输出信息进行打分，例如通过比对特定输出字符串，或依据性能参数（如带宽、耗时）利用特定公式计算得分。自动化评分脚本（如 judge_xxxxx.py）会据此完成评测。

以 basic 基础测例为例，该组测试主要利用一系列基本的 Linux 系统调用来验证系统调用实现的正确性。例如，brk 系统调用用于测试内存分配，chdir 和 getcwd 用于目录操作。测试过程中，一个包装脚本（如 basic_testcode.sh）会依次运行各个子测试，并收集其输出。评分时则依据每个子测试的执行结果及关键环节（如堆指针、系统调用返回值、当前目录等）是否符合预期来判定得分。

再如 libctest 测试样例，主要聚焦于操作系统对 C 标准库（libc）各项功能的支持和兼容性。C 标准库涵盖了输入输出、内存分配、字符串处理等基础功能。libctest 通过运行一系列预编译的测试用例（包括静态和动态链接版本），并检查其输出是否为“Pass!”来判断内核对 C 标准库的支持情况。

借助这些标准化的测试用例和清晰明确的评估方法，我们能够对 Starry 系统的整体功能进行系统性、客观的验证与评估。

5.5 测试结果和分析

截止到本文撰写时，Starry 已经通过了全部的 basic 测例、大部分 libctest 测例、部分 busybox 测例和全部的 lua 测例，如表5.1所示。

表 5.1 Starry 测例通过情况表

测例组	通过情况（通过数/总数）
basic	32/32
libctest	176/217
busybox	38/54
lua	9/9
其它	尚未测试

其中，libctest 测例中与文件系统相关性较强的测例通过情况如表5.2所示。

表 5.2　libctest 文件系统相关测例通过情况表

测例名称	涉及的文件相关系统调用	通过情况
utime	utimensat, open, unlink, brk, mmap, ioctl, fstat, close, munmap, exit_group, wait4	通过
ungetc	open, unlink, brk, mmap, ioctl, close, munmap, exit_group, wait4	通过
getpwnam_r_errno	open, fcntl, brk, mmap, munmap, close, socket, connect, exit_group, wait4	通过
setvbuf_unget	exit_group, wait4	通过
fgetwc_buffering	pipe, dup2, close, exit_group, wait4	通过
statvfs	statvfs, statfs	未通过
dirname	exit_group, wait4	通过
rlimit_open_files	open, exit_group, wait4	未通过
basename	exit_group, wait4	通过
fwscanf	open, unlink, brk, mmap, ioctl, close, munmap, exit_group, wait4	通过
fdopen	open, brk, mmap, close, munmap, unlink, exit_group, wait4	通过
mkdtemp_failure	exit_group, wait4	通过
fgets_eof	brk, mmap, exit_group, wait4	通过
daemon_failure	pipe, close, wait4, getpid, chdir, open, dup2, getppid, exit_group	通过
ftello_unflushed_append	open, close, brk, mmap, fcntl, ioctl, munmap, unlink, exit_group, wait4	通过
iconv_open	exit_group, wait4	通过
fscanf	pipe, brk, mmap, close, munmap, open, unlink, ioctl, exit_group, wait4	通过
mkstemp_failure	exit_group, wait4	通过
fnmatch	exit_group, wait4	通过
getpwnam_r_crash	open, fcntl, brk, mmap, munmap, close, socket, connect, exit_group, wait4	通过
stat	stat, open, unlink, brk, mmap, ioctl, fstat, geteuid, getegid, close, munmap, exit_group, wait4	通过
memstream	brk, mmap, munmap, exit_group, wait4	通过

续表 5.2　libctest 文件系统相关测例通过情况表

测例名称	涉及的文件相关系统调用	通过情况
fflush_exit	open, wait4, close, ioctl, exit_group, pread64, unlink	通过
rewind_clear_error	close, exit_group, wait4	通过
lseek_large	open, unlink, brk, mmap, ioctl, exit_group, wait4	通过
pthread_cancel_points	mmap, mprotect, clone, futex, tkill, rt_sigreturn, exit, munmap, open, fcntl, close, unlink, exit_group, wait4	通过
iconv_roundtrips	exit_group, wait4	通过
syscall_sign_extend	open, exit_group, wait4	通过

可以看到，在 libctest 测例中，与文件系统相关性较强的测例通过情况较好，只有 statvfs 和 rlimit_open_files 测例未通过。statvfs 测例未通过是因为 statfs 系统调用尚未实现；而 rlimit_open_files 测例未通过是因为目前尚未实现对文件描述符数量的限制，即始终为默认的每个进程 1024 个。未来，我们将针对此类尚未完善的功能继续进行开发和优化。

第 6 章　总　结

本文围绕组件化操作系统 ArceOS 的宏内核扩展 Starry，系统性地设计与实现了一个高可靠性且功能基本完备的文件系统管理组件，相关代码已开源在 Github 仓库[①]。本文的主要工作体现在：深入剖析了 ArceOS 及 Starry 的系统架构与既有文件系统基础，为目标组件的设计奠定了坚实的理论与工程依据；为 Starry 实现了统一的文件描述符抽象与管理机制，保证了系统的 POSIX 兼容性与未来的可扩展性；全面实现并优化了文件操作、目录管理、挂载机制等相关的核心系统调用，完善了 Starry 的功能；最后，通过引入标准化的综合测试套件，对所实现的文件系统组件进行了严格的功能性、兼容性与稳定性验证，结果表明其已具备良好的工程价值。基于当前实现的文件系统管理组件，Starry 已经能支持较为复杂的 Linux 应用运行，具有一定的实用性。

尽管本文已成功实现了 Starry 宏内核下文件系统管理组件的核心功能，并通过测试基本验证了其可用性，但该文件系统管理组件仍存在许多不足之处，如尚有部分系统调用未实现或实现不完全，文件系统的操作性能有提升空间，所支持的文件系统类型有限等等。

总之，随着硬件技术和应用需求的不断演进，文件系统作为操作系统内核的关键基础设施，其设计与实现仍将面临新的挑战和机遇。本文的工作为 Starry 宏内核扩展实现提供了坚实的基础，未来将继续围绕高性能、高可靠性、强兼容性和易扩展性等目标，持续推进相关研究与工程创新，助力新一代操作系统的发展与应用。

① https://github.com/Ressed/starry-next

参考文献

[1] Isaac O A, Okokpujie K, Akinwumi H, et al. An Overview of Microkernel Based Operating Systems[J/OL]. IOP Conference Series: Materials Science and Engineering, 2021, 1107(1): 012052. DOI: 10.1088/1757-899X/1107/1/012052.

[2] Hildebrand D. An Architectural Overview of QNX.[C]//USENIX Workshop on Microkernels and Other Kernel Architectures. Citeseer, 1992: 113-126.

[3] 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. DOI: 10.1145/1629575.1629596.

[4] Malallah H, Zeebaree S, Zebari R R, et al. A comprehensive study of kernel (issues and concepts) in different operating systems[J]. Asian Journal of Research in Computer Science, 2021, 8(3): 16-31.

[5] Engler D R, Kaashoek M F, O'Toole J. Exokernel: An operating system architecture for application-level resource management[J/OL]. SIGOPS Oper. Syst. Rev., 1995, 29(5): 251-266. DOI: 10.1145/224057.224076.

[6] 舒红梅, 谭良. 库操作系统的研究及其进展[J]. 计算机科学, 2018, 45(11): 37-44.

[7] 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. DOI: 10.1145/1254810.1254817.

[8] Raza A, Sohal P, Cadden J, et al. Unikernels: The Next Stage of Linux's Dominance[C/OL]// HotOS '19: Proceedings of the Workshop on Hot Topics in Operating Systems. New York, NY, USA: Association for Computing Machinery, 2019: 7-13. DOI: 10.1145/3317550.3321445.

[9] Rosenblum M, Ousterhout J K. The lfs storage manager[C]//Proceedings of the 1990 Summer Usenix. 1990.

[10] Zhang Y, Rajimwale A, Arpaci-Dusseau A C, et al. End-to-end Data Integrity for File Systems: A ZFS Case Study.[C]//FAST. 2010: 29-42.

[11] Rodeh O, Bacik J, Mason C. Btrfs: The linux b-tree filesystem[J]. ACM Transactions on Storage (TOS), 2013, 9(3): 1-32.

[12] Pawlowski B, Noveck D, Robinson D, et al. The nfs version 4 protocol[C]//In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000. Citeseer, 2000.

[13] Kala Karun A, Chitharanjan K. A review on hadoop — HDFS infrastructure extensions[C/OL]// 2013 IEEE Conference on Information & Communication Technologies. 2013: 132-137. DOI: 10.1109/CICT.2013.6558077.