综 合 论 文 训 练

操作系统宏内核内存管理组件接口的设计与实现

系　　别：计算机科学与技术系

专　　业：计算机科学与技术

姓　　名：陈 羿 华

指导教师：陈　　渝　　副教授

二〇二五年六月

关于论文使用授权的说明

本人完全了解清华大学有关保留、使用综合论文训练论文的规定，即：学校有权保留论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

作者签名：　　导师签名：

日　　期：　　日　　期：

摘　要

内核是作为操作系统的核心组件，负责管理计算机系统的处理器、内存、文件系统等资源，其设计和实现对于操作系统的性能和稳定性至关重要。而模块化或者说组件化的内核设计则可以提高内核的可维护性和扩展性，同时也可以减少内核的复杂性和风险。

内存管理模块是模块化内核设计中最基本的模块之一，其主要任务是管理计算机系统的内存资源，提供内存分配、释放和映射等功能。本文基于 ArceOS 基座和已有架构，对 starry-next 的内存管理模块进行详细地分析和设计，包括 ArceOS 内存管理相关的组件和模块、starry-next 内存管理模块的结构和内存管理机制，以及 mmap、munmap、mprotect、brk 等系统调用的实现与测试，验证了当前内存管理模块设计的正确性和稳定性，一定程度上明确了内存管理模块的功能与边界。

同时，本文与该课题下其他模块共同开发宏内核的过程，也体现了内核的模块化设计在减少相互依赖、降低系统复杂度等方面的重要作用，为进一步的内核设计和优化提供了重要的参考和借鉴。

关键词：操作系统；内核；内存管理；模块化；系统调用

Abstract

The kernel serves as the core component of an operating system, tasked with managing the processor, memory, file system, and other critical resources of a computer system. Its design and implementation are pivotal to the overall performance and stability of the operating system. Adopting a modular or component-based approach to kernel design can significantly enhance its maintainability and extensibility, while simultaneously mitigating complexity and associated risks.

The memory management module constitutes one of the fundamental components within a modular kernel architecture. It is primarily responsible for overseeing the memory resources of a computer system, offering essential functionalities such as memory allocation, deallocation, and mapping. This paper presents an in-depth analysis and design of the memory management module for starry-next, based on the ArceOS framework and existing architecture. This encompasses an examination of the components and modules related to memory management in ArceOS, an exploration of the structure and mechanisms of the memory management module in starry-next, and the implementation and testing of key system calls, including mmap, munmap, mprotect, and brk. These efforts have validated the correctness and stability of the current memory management module design, thereby contributing to a better understanding of its functions and boundaries.

Furthermore, the collaborative development of the macrokernel within this project, alongside other modules, underscores the crucial role of modular kernel design in reducing interdependencies and diminishing system complexity. This experience provides valuable insights and serves as a significant reference for future kernel design and optimization endeavors.

Keywords: operating system; kernel; memory management; modular; system call

目 录

插图清单

附表清单

符号和缩略语说明

OS	操作系统
FIFO	先进先出
LRU	最近最少使用
MMU	内存管理单元
AArch64	64 位 ARM 架构
RISCV64	64 位 RISC-V 架构
LoongArch64	64 位龙芯架构
x86_64	64 位 x86 架构
ELF	可执行和可链接格式
TLB	转换后备缓冲器
DMA	直接内存访问
QEMU	可以模拟各种硬件设备的虚拟化模拟器

第 1 章　引　言

1.1　课题背景

1.1.1　操作系统宏内核

操作系统是计算机硬件和用户之间的接口。它负责管理和调度计算机的硬件资源（如 CPU、内存、输入输出设备等），同时为用户提供一个易于使用的操作界面，使得用户可以通过简单的命令或图形界面来操作计算机。操作系统内核是计算机系统的核心组件，它负责管理计算机硬件资源（如处理器、内存、输入输出设备等）和提供用户程序运行的环境。随着计算机技术的不断发展，操作系统内核也在不断演化和改进，以适应不同的使用场景和需求。而根据操作系统内核的设计模式，又可以将操作系统内核划分为不同的类型，如微内核、宏内核等。

宏内核是一种操作系统内核的设计模式，它将操作系统的所有功能（如进程管理、内存管理、文件系统、设备驱动等）集成在一个单一的地址空间中。整个内核运行在内核态，具有较高的执行效率。宏内核的优点是可以提供更好的性能和更灵活的功能，但是也存在一些缺点，如内核的复杂性和可拓展行问题——当需要添加或修改功能时，可能会涉及到多个模块的修改；内核的稳定性问题——当出现故障时，可能会导致整个系统崩溃。

1.1.2　操作系统的组件化

为了增强操作系统的可拓展性和稳定性，操作系统通常采用组件化设计。组件化设计是一种将操作系统的功能划分为多个模块的设计模式，每个组件负责特定的功能，组件之间通过明确定义的接口进行通信。这样做的好处是可以提高系统的可拓展性和稳定性，因为每个组件都可以独立开发、测试和维护。

1.1.3　操作系统的内存管理

内存管理是操作系统的核心功能之一，它负责管理和分配计算机系统的内存资源。内存管理的主要目标是确保系统能够高效地利用有限的物理内存，同时为用户提供透明的、虚拟的内存空间。内存管理模块的主要功能包括：

- 内存分配：当用户程序请求内存时，内存管理模块需要从可用的内存池中分配一块合适的内存空间给程序。分配策略可以是多种多样的，如首次适应、最佳适应、最坏适应等。
- 内存回收：当用户程序释放内存时，内存管理模块需要将这块内存回收到可

用内存池中。回收时需要考虑内存碎片问题，如合并相邻的空闲内存块等。

- 内存保护：内存管理模块需要确保每个进程只能访问自己被分配的内存区域，防止一个进程非法访问其他进程的内存空间。还可以设置不同的访问权限，例如只读、可读写等。
- 虚拟内存管理：虚拟内存是一种计算机系统内存管理技术，它使得应用程序认为自身拥有一块连续的、足够大的内存空间，实际上这些内存可能被分隔成多个物理内存碎片，并且部分数据可能暂时存储在外部磁盘上。当程序运行时，需要的数据会从磁盘交换到物理内存中。虚拟内存通常以页面为单位进行管理。内存管理模块负责将虚拟内存地址映射到物理内存地址，并且当物理内存不足，需要从磁盘上读取页面时，内存管理模块需要决定将哪个页面从物理内存中置换到磁盘上。常见的页面置换算法包括先进先出（FIFO）、最近最少使用（LRU）等。
- 内存映射：内存映射是一种将磁盘上的文件映射到内存中的技术，使得应用程序可以像访问内存一样访问文件，这种方式可以提高文件访问的效率，因为它减少了文件 I/O 操作的开销。内存管理模块负责将磁盘上的文件映射到内存中，并提供相应的接口供应用程序访问。
- 共享内存：共享内存是一种允许多个进程共享同一块内存区域的技术，这种技术可以提高进程间的通信效率，因为它避免了数据的复制。内存管理模块负责管理共享内存区域，并提供相应的接口供应用程序访问。

1.2 相关研究工作

除了 Linux 内核，还有许多其他较为完善的操作系统内核，如 DragonOS、星绽、ByteOS 等。这些操作系统内核都提供了自己的内存管理模块，并且在内存管理的设计和实现上有所不同。

1.2.1 DragonOS

DragonOS 是一个面向云计算轻量化场景的，完全自主内核的，提供 Linux 二进制兼容性的 64 位操作系统，它具有优秀完善的架构设计，支持虚拟化，在设备模型、调度子系统等方面具有一定优势。DragonOS 的内存管理模块主要由以下类型的组件组成：

- 硬件抽象层（MemoryManagementArch）- 提供对具体处理器架构的抽象，使得内存管理模块可以在不同的处理器架构上运行
- 页面映射器（PageMapper）- 提供对虚拟地址和物理地址的映射，以及页表

的创建、填写、销毁、权限管理等操作。分为两种类型：内核页表映射器（KernelMapper）和用户页表映射器（位于具体的用户地址空间结构中）

- 页面刷新器（PageFlusher）- 提供对页表的刷新操作（整表刷新、单页刷新、跨核心刷新）
- 页帧分配器（FrameAllocator）- 提供对页帧的分配、释放、管理等操作。具体来说，包括 BumpAllocator、BuddyAllocator
- 小对象分配器 - 提供对小内存对象的分配、释放、管理等操作。指的是内核里面的 SlabAllocator （SlabAllocator 的实现目前还没有完成）
- MMIO 空间管理器 - 提供对 MMIO 地址空间的分配、管理操作。（目前这个模块待进一步重构）
- 用户地址空间管理机制 - 提供对用户地址空间的管理。
- VMA 机制 - 提供对用户地址空间的管理，包括 VMA 的创建、销毁、权限管理等操作
- 用户映射管理 - 与 VMA 机制共同作用，管理用户地址空间的映射
- 系统调用层 - 提供对用户空间的内存管理系统调用，包括 mmap、munmap、mprotect、mremap 等
- C 接口兼容层 - 提供对原有的 C 代码的接口，使得 C 代码能够正常运行。

DragonOS 使用 AddressSpace 结构体来管理用户地址空间，这个结构体包含了诸多成员，如 mappings（VMA 列表）、mmap_min（最小映射地址）、brk（堆的当前顶部地址）等，这些成员用于管理用户地址空间的不同区域。其中 UserMappings(mappings) 代表了用户虚拟内存空间，并且提供了查找、插入、删除用户虚拟内存区域 VMA 的接口，VMA 中包含了虚拟内存区域的起始地址、结束地址、权限等信息，一个 VMA 可能包含多个虚拟页，每个虚拟页都对应一个物理页帧。虚拟地址和物理地址的映射由页表管理器 PageMapper 来管理，PageMapper 包含页表类型等属性，指定了物理页帧分配器的类型，提供管理页表、虚拟地址与物理地址相互映射的接口，并连接到物理内存管理器，提供物理页帧的分配和回收等功能。

DragonOS 的物理内存由物理页管理器 (PageManager) 管理，它是一个以物理地址和物理页对象为键值对的哈希表，使用页帧分配器进行物理页的分配，使用页面回收线程和页面回收器来回收空闲的物理页。

当用户程序需要对内存进行操作，例如分配内存时，它会调用相应的 mmap 系统调用，系统调用 AddressSpace 结构体的接口在用户地址空间中创建一个新的虚拟内存区域，并通过页表管理器分配物理页，将其映射到物理内存中；当用户程序需要释放内存时，它会调用 munmap 系统调用，munmap 系统调用会将用户地址空

间中的虚拟内存区域与物理内存中的物理页断开映射关系，释放物理页，并将虚拟内存区域从用户地址空间中删除，在需要写回的情况下，会唤醒页面回收线程，将物理页标记为可回收状态，由页面回收器进行回收。

1.2.2 星绽

星绽 (asterinas) 是一个用 Rust 编写的安全、快速且通用的操作系统内核，且与 Linux 兼容。在物理内存管理方面，星绽定义了多种内存区域类型，如 BadMemory、Unknown、NonVolatileSleep、Reserved、Kernel、Module、Framebuffer、Reclaimable 和 Usable，以帮助内核识别不同用途的内存区域。使用 MemoryRegion 结构体表示一个内存区域，包含基地址、长度和类型信息；使用 MemoryRegionArray 无堆集合管理多个内存区域，它提供了 push 方法用于添加区域，into_non_overlapping 方法用于将区域排序并合并为非重叠的区域集合。

在虚拟内存管理方面，星绽用 Vmo 结构体来表示虚拟内存实例，提供了读写内存页的接口。进程的虚拟内存空间则由 ProcessVm 结构体表示，它包含一个根 Vmar（虚拟内存地址区域），而 Vmar 则关联着用户模式任务的虚拟内存空间管理器 VmSpace 和内存映射实例 VmMapping 的集合。VmSpace 结包含页表等信息，并提供了内存映射的相关接口。当为进程分配虚拟内存时，ProcessVm 会借助 Vmar 来创建和管理内存映射，而这些映射最终会反映在关联的 VmSpace 中。

在更上层的系统调用接口方面，星绽提供了与 Linux 兼容的系统调用接口，如 mmap、munmap、mprotect、mremap 等，这些接口可以被用户程序调用，用于管理用户模式任务的虚拟内存空间。

1.2.3 ByteOS

ByteOS 是一个基于 Rust 语言开发的组件化操作系统宏内核，由进程管理、内存管理、文件系统、网络协议栈等多个模块组成。ByteOS 的内存管理模块主要包括以下几个部分：

- 堆内存空间定义：定义了一个静态的堆内存数组 HEAP，其大小由 HEAP_SIZE 常量指定。
- 全局堆内存分配器：使用 buddy_system_allocator::LockedHeap 实现了一个全局的堆内存分配器 HEAP_ALLOCATOR。
- 物理页帧管理：FrameTracker 用于表示一个已经被分配的页帧，并且利用 Drop 机制保证页帧能够顺利被回收。当 FrameTracker 实例被销毁时，会自动调用 drop 方法，将对应的页帧标记为未使用状态。FrameRegionMap 是页帧分布图，用于保存页帧分配器中的空闲内存，并且利用 BitArray 记录页帧使用情

况。每个 FrameRegionMap 表示一段连续的物理内存区域。FrameAllocator 是一个总的页帧分配器，包含多个 FrameRegionMap，负责管理整个系统的物理页帧分配与释放。

- 内存映射：使用 MapTrack 结构体表示一个内存映射，包含虚拟地址和对应的物理页帧 FrameTracker。
- 页帧分配函数：在 UserTask 结构体包含进程页表、进程控制块、虚拟内存空间等信息，并提供了 frame_alloc 和 map_frames 方法用于页帧分配和映射。
- 进程内存空间管理：MemArea 结构体表示一个内存区域，包含内存类型、页帧映射、文件关联等信息，多个 MemArea 组成一个 MemSet。进程控制块中记录了进程使用的内存区域，并通过页表机制实现虚拟内存到物理内存的映射。
- 系统调用接口：提供了 mmap、munmap、mprotect、mremap 等系统调用接口，用于进程内存管理。

当用户程序当用户进程申请一块内存时，ByteOS 会根据不同的系统调用（如 sys_mmap 或 sbrk），进而调用用户进程（UserTask）的 frame_alloc 接口，通过物理页帧分配器申请物理页帧，创建新的虚拟内存区域，并将其映射到物理内存中，最后将虚拟内存区域加入到进程的内存空间中进行管理。

1.3 课题内容和意义

1.3.1 课题内容

本课题的内容是基于 ArceOS 基座和 starry-next 宏内核的已有结构，实现内存管理模块及其接口的设计与实现，包括实现对 ArceOS 和 starry-next 的内存管理模块的设计与分析，以及内存管理直接或间接相关的系统调用的实现，并对内存管理模块及其接口进行测试。

1.3.2 课题意义

通过研究并设计实现宏内核内存管理组件与系统调用接口，可以明确内存管理系统的职责和边界，帮助内存管理系统作为一个独立的模块进行设计和实现。实现系统调用也有助于发现和解决当前内存管理组件可能存在的问题。

不同的应用场景对内存管理有着不同的要求。模块化的内存管理系统能有效降低功能拓展的难度，可以根据需求进行相对独立的配置和开发。而统一的系统调用接口的实现也有助于内存管理组件根据具体的应用场景，提供相应的内存管理策略和优化措施，满足多样化的应用需求。

1.4 论文结构

本文分为 6 个章节。其中，第一章节主要介绍课题背景、相关研究和与课题内容，第二章节将介绍基座 ArceOS 的组件与其内存管理，第三章节将介绍 starry-next 宏内核的架构和内存管理模块的设计，第四章节将介绍 starry-next 内存管理模块接口的设计和实现，第五章节将介绍接口的测试。第六章节是对工作的总结与展望。

第 2 章　ArceOS 内核

2.1　ArceOS 概述

ArceOS 是一个基于 Rust 语言开发的组件化操作系统，它是目标宏内核 starry-next 的基底，为宏内核提供了底层组件和相应接口的支持。ArceOS 由以下模块组成：

- axruntime：负责从裸机环境启动并进行初始化工作，在进入应用程序的 main 函数之前完成一系列准备操作，如日志初始化、内存分配器初始化、平台设备初始化等。
- axhal：硬件抽象层，负责指定平台的启动和初始化过程，通过针对不同的硬件平台和架构实现特定的代码，将底层硬件的差异封装起来，为上层提供统一的接口。支持多种架构，如 x86_64、riscv64、aarch64、loongarch64。
- axconfig：存储平台特定的常量和参数，如物理内存基地址、内核加载地址、栈大小等。
- axlog：提供多级格式化日志记录功能。
- axalloc：全局内存分配器，实现了一套内存分配和管理算法，负责为系统和应用程序分配内存。在需要内存时，会根据请求的大小和内存状态，从空闲内存池中分配合适的内存块，并记录内存的使用情况。
- axdisplay：图形模块，用于处理图形相关的操作。
- axfs：文件系统模块，实现了文件系统的基本操作，如文件的创建、读取、写入、删除等。会维护文件系统的元数据（如文件的目录结构、文件属性等），并将文件数据存储在存储设备上。
- axnet：网络模块，封装了底层网络栈的功能，提供了类似 POSIX 的网络 API。在接收到网络请求时，会将请求传递给底层网络栈进行处理，并将处理结果返回给上层应用程序。
- axdriver：设备驱动模块，负责检测和初始化各种设备驱动，并将检测到的设备封装到 AllDevices 结构体中供上层子系统使用。
- axtask：任务管理模块，负责维护一个任务队列，提供任务创建、调度、睡眠、终止等接口。调度算法会根据任务的优先级、状态等因素，决定哪个任务应该在何时运行。在任务调度时，会保存当前任务的上下文信息，加载下一个任务的上下文信息，实现任务的切换。
- axsync：同步原语模块，通过硬件提供的原子操作或软件算法实现同步原语。

例如，互斥锁可以使用原子操作来实现对共享资源的互斥访问，确保同一时间只有一个任务可以访问共享资源。

- axdma：DMA 模块，用于管理直接内存访问（DMA）操作，允许某些硬件子系统在不经过 CPU 控制的情况下直接访问系统内存。
- axmm：虚拟内存管理模块，其主要功能在于对虚拟内存进行高效的分配、映射以及管理，确保系统和应用程序能够合理、安全地使用内存资源。
- axns：命名空间模块，用于控制线程间系统资源共享。它通过命名空间来管理系统资源，包括虚拟地址空间、工作目录和文件描述符，用于在不同场景下访问系统资源。

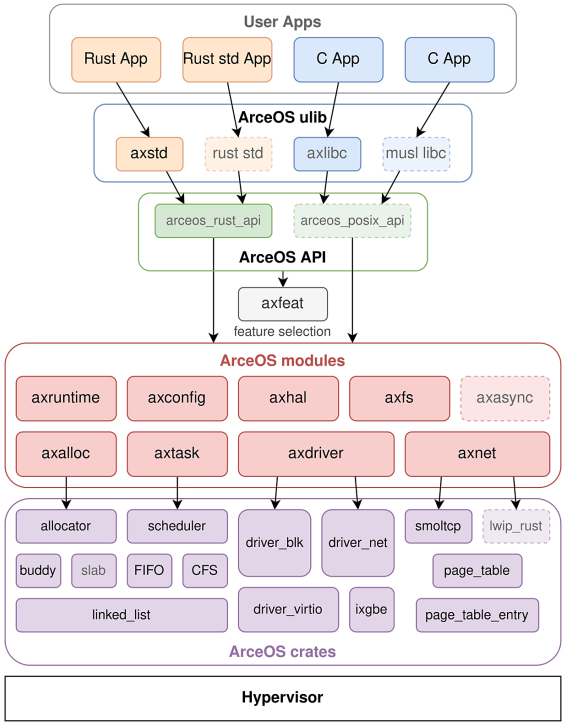


图 2.1　ArceOS 内核结构图

每个模块包含多个组件，不同模块组合组合为应用程序提供底层支持。其中应用运行时模块（axruntime）、硬件抽象层模块（axhal）以及动态内存分配模块（axalloc）是不可缺少的核心组件，其他模块可以根据具体需求进行选择和使用。

在 ArceOS 的启动过程中，axhal 模块会将对应架构的 _start 函数链接到".text.boot" 段，作为 ArceOS 运行的第一段代码，完成一些基础的硬件相关初始化操作，例如设置栈指针、初始化页表和 MMU（内存管理单元）等，并跳转到 Rust 主函数 rust_entry。rust_entry 函数位于 axruntime 模块中，它会依据不同的 Cargo 特性进行有针对性的初始化工作，若启用日志功能，会初始化日志系统；启

用内存分配特性时，会查找物理内存区域并初始化全局内存分配器；接着会进行平台设备初始化，确保硬件正常工作。对于多任务、文件系统、网络、图形显示、对称多处理以及中断等特性，也会分别完成调度器、设备驱动、从 CPU、中断处理程序等的初始化。最后，它会调用应用的 main 函数，开启应用程序的执行，为操作系统和应用程序的运行构建起完整的基础环境。

当应用程序的 main 函数执行完毕后，若启用了 multitask （多任务）特性，将会调用 axtask::exit(0) 退出函数来退出当前任务；若未启用该特性，则会调用 axhal 模块下的 misc::terminate 函数来结束整个系统的运行。

2.2 ArceOS 内存管理组件及接口

在 ArceOS 的模块中，axhal、axalloc、axmm 和 axdma 模块是 ArceOS 内存管理的核心模块，它们分别负责对不同硬件平台的硬件封装、物理内存分配、虚拟内存管理和直接内存访问，组成了如图 2.2 所示的内存管理系统。

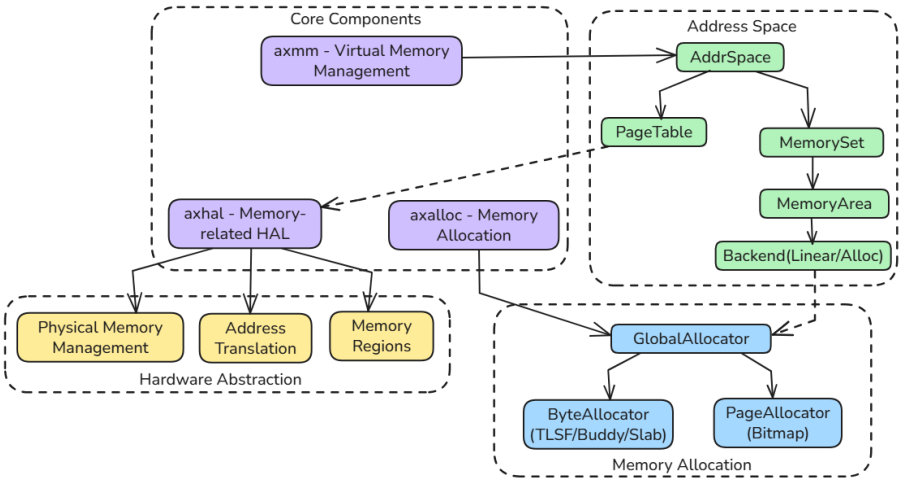


图 2.2 ArceOS 内存管理系统

2.2.1 axhal

axhal 组件提供了一层针对不同硬件平台的硬件封装，它为指定的操作平台进行引导和初始化过程，并提供对硬件的操作。在内存方面，axhal 同样进行了一系列的封装操作来支持内存管理。例如：

- 页表管理封装：axhal 模块将不同架构的页表统一封装为 PageTable 类型，并提供了一致的操作接口，例如 alloc_frame（通过页表分配物理页帧）、dealloc_frame（通过页表释放物理页帧）、phys_to_virt（将物理地址转换为虚拟地址）、set_kernel_page_table_root（设置内核页表根地址）和 kernel_page_table_root（获取内核页表根地址）等。

- 内存区域封装：axhal 模块中定义了 MemRegion 结构体，用于表示和架构无关物理内存区域。MemRegion 结构体包含起始物理地址、区域大小、区域标志和区域名称等属性。通过该结构体，可以方便地管理和操作物理内存区域。
- 获取内存区域的函数封装：axhal 提供 kernel_image_regions、default_mmio_regions、default_free_regions 接口用于获取内核映像区域、默认 MMIO 区域和默认空闲区域的内存区域列表，并将其封装为 MemRegion 类型。默认的 MMIO 区域和默认空闲区域的位置由 axconfig 模块定义，在不同的硬件平台上可能会有所不同。

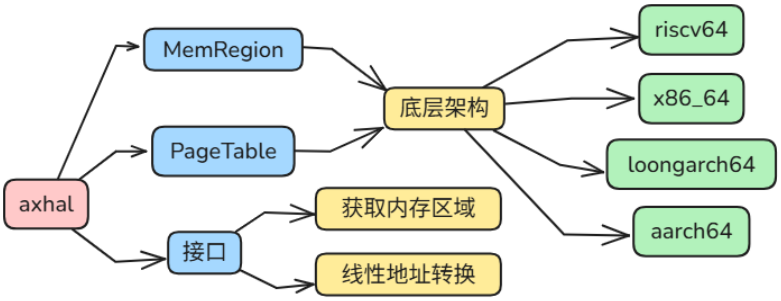


图 2.3　axhal 内存管理相关结构图

在 Rust 的 page_table_multiarch 包的支持下，不同架构的地址和页表有不同的结构和布局，例如 x86_64 架构采用 4 级页表，支持 48 位虚拟地址和 56 位物理地址；AArch64 架构采用 4 级页表，支持 48 位虚拟地址和 48 位物理地址；LoongArch64 架构采用 4 级页表，支持 48 位虚拟地址和 48 位物理地址[1]；RISC-V64 架构则使用 Sv39 结构，采用 3 级页表，支持 39 位虚拟地址和 56 位物理地址。这里以 Sv39 结构为例，介绍其页表的结构和布局，以及虚拟地址和物理地址的转换。

如图2.4所示，在 Sv39 标准下，虚拟地址由 39 位组成，其中前 27 位为页目录索引，每 9 位对应一级页表，最后 12 位为页内偏移。页表则为三级结构，每个节点都是一个 4K 的物理页，包含 512 个页表项。当虚拟地址转换为物理地址时，首先根据 satp 寄存器找到根页表（二级页表）的物理地址，然后根据 VPN2 找到对应的页表项，根据其中物理页号（PPN）可以找到一级页表的物理地址。接着按照同样的方法根据 VPN1 获取零级页表和 VPN0 对应的页表项，最后根据页表项中的物理页号找到对应的物理页，再加上页内偏移得到最终的物理地址[2]。

2.2.2　axalloc

如图2.5，axalloc 模块中定义了全局内存分配器 GlobalAllocator，由字节分配器和页面分配器组成。其中，字节分配器用于小内存块的分配和释放，在 ArceOS 中我们提供了三种不同的字节分配器：

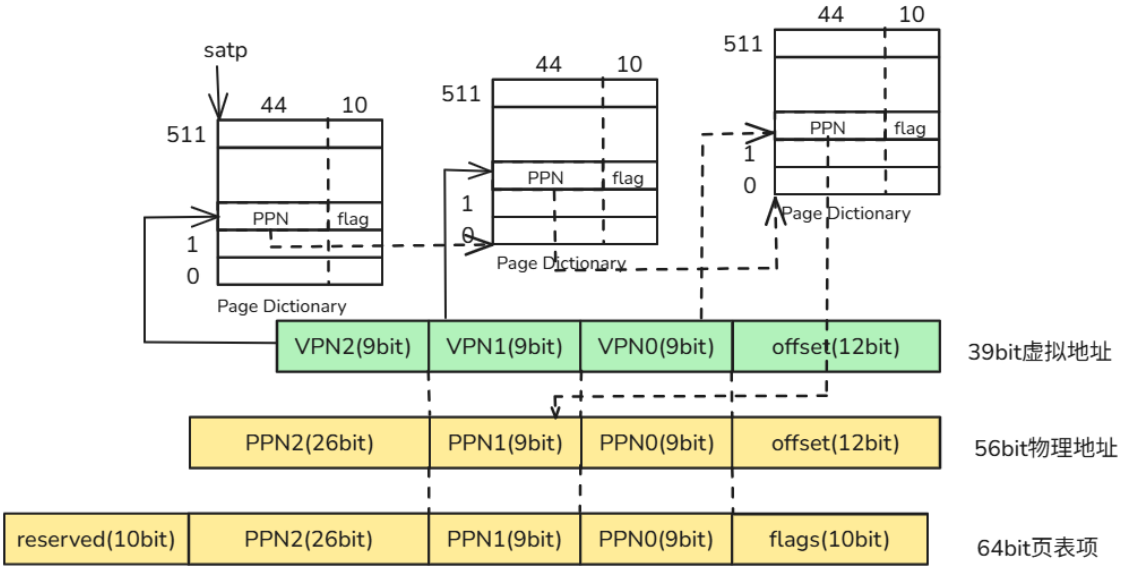


图 2.4　Sv39 标准下虚实地址结构、页表结构和寻址过程

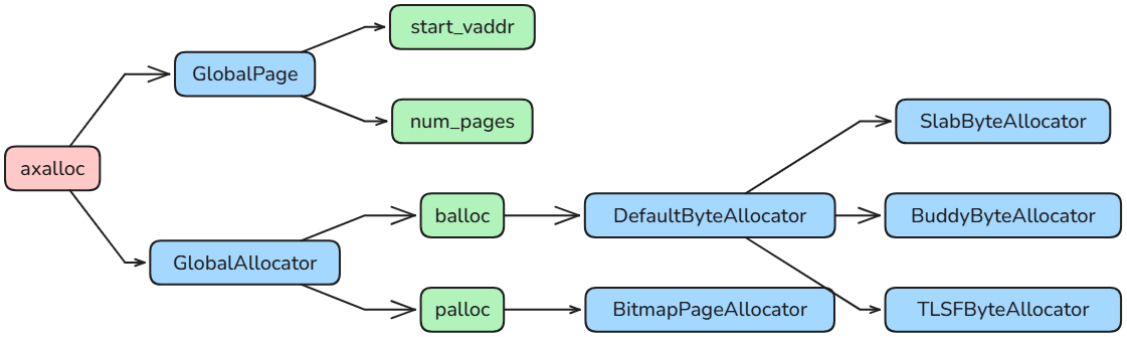


图 2.5　axalloc 结构图

- SlabByteAllocator——基于 Slab 分配算法的字节分配器，其核心思想是将内存按照对象的大小进行分类，对于每一类对象，预先分配好一组连续的内存块，这些内存块组成一个“slab”。每个 slab 包含了多个相同大小的对象槽（Object Slots），当需要分配某个特定大小的对象时，就从对应的 slab 中获取一个空闲的对象槽；当对象释放时，对应的对象槽又可以被重新利用。同时，为了更好地管理这些 slab，会有 slab 缓存（Slab Cache）机制，用于跟踪空闲和已使用的 slab 等状态。slab 算法对于频繁分配和释放相同类型、相同大小对象的场景表现卓越，同类型对象集中存放使得在数据访问时对 CPU 缓存的利用更为充分，并且可以有效地减少内存碎片。但固定大小的 slab 可能会导致内存浪费和对于多样化、大小不一的内存分配需求适应性较差的问题[3]。
- BuddyByteAllocator——基于 Buddy 算法的字节分配器，以一种二叉树的思想来管理内存，它将整个内存空间看作是一个完整的、大小为 2 的幂次方的

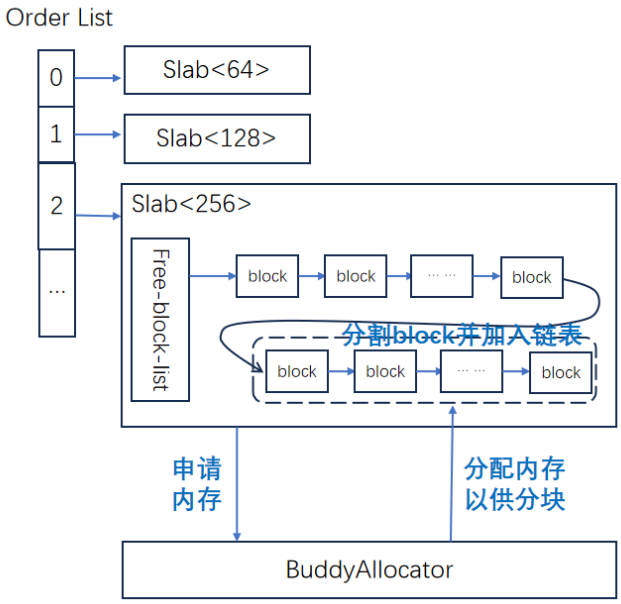


图 2.6　Slab 算法示意图

内存块，当需要分配内存时，如果有符合要求大小（同样是 2 的幂次方）的空闲内存块，就直接分配；若没有，则将更大的空闲内存块不断地二等分（即找到它的“buddy”，也就是相邻且同样大小的另一半内存块），直到得到合适大小的内存块进行分配。在内存释放时，会检查释放的内存块与其 buddy 是否都空闲，如果是，则将它们合并成一个更大的空闲内存块，如此不断向上合并，直到不能合并为止。Buddy 算法通过不断地合并空闲的“伙伴”内存块，能让内存空间保持相对规整，避免出现大量零散的小空闲块，内存释放时的合并操作判断和执行合并的过程也相对简单高效，但内存分配粒度受限于 2 的幂次方，可能会导致内存浪费和内部碎片问题。[4]

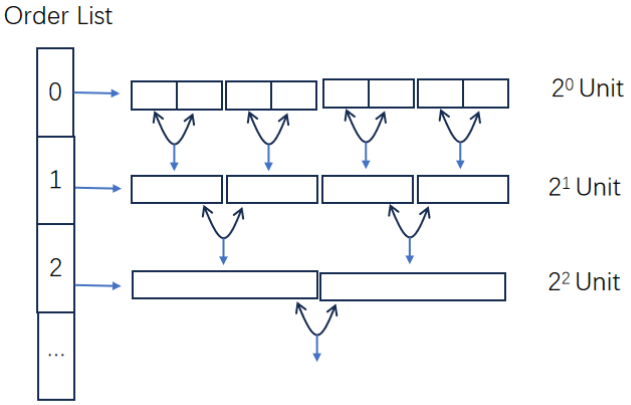


图 2.7　Buddy 算法示意图

- TlsfByteAllocator——基于 TLSF（Two-Level Segregate Fit）算法的字节分配器，TLSF 采用了两级的分离适配结构。第一级是将整个内存空间按照不同大小范围划分成多个区间，每个区间对应着不同大小的内存块类别，形成一个区间链表。第二级则是针对每个区间，再使用一个位图（Bitmap）和一个空闲块链表来管理该区间内具体的空闲内存块。在进行内存分配时，先根据要分配的内存大小确定所属的区间，然后在位图中查找是否有合适的空闲块，若有则从对应的空闲块链表中取出进行分配；内存释放时，将释放的内存块重新插入到对应的空闲块链表，并相应更新位图信息。TLSF 算法具有高效的分配速度和高内存利用率，可以灵活应对各种不同大小内存块的分配情况，但是维护两级结构中的各种管理信息使得内存开销相对较大[5]。

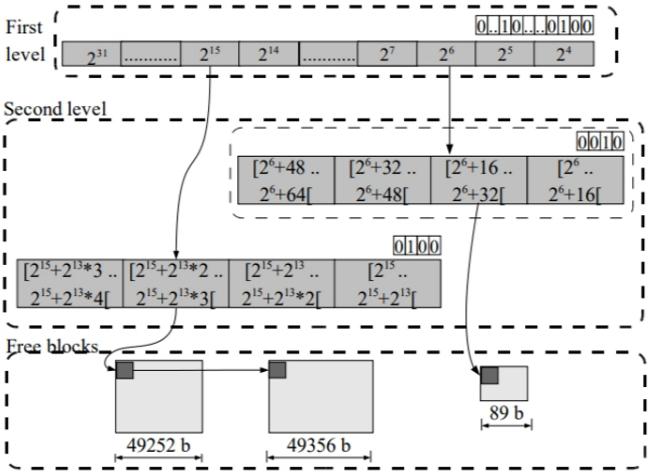


图 2.8　TLSF 算法示意图

页分配器负责大型分配和扩展字节分配器，使用了标准库中的 BitmapPageAllocator，它是一个基于位图（bitmap）的页粒度内存分配器。它通过使用位图中的每一位来表示一个内存页的分配状态，从而实现高效的内存管理。该分配器支持不同大小的内存分配（从 256MB 到 1TB），并且可以处理不同页大小（PAGE_SIZE）的情况，要求页大小必须是 2 的幂次方。它提供了初始化内存区域、分配和释放内存页等基本功能，同时支持对齐分配和指定地址分配等特性。

全局内存分配器通过接口函数提供了对物理内存的操作功能，包括内存的分配、释放、查询等。其主要功能如下：

- 内存分配器类型选择：通过 Cargo 特性，axalloc 支持用户可以根据需求在 Buddy 分配器、Slab 分配器、TLSF 分配器中选择一种作为全局内存分配器的字节分配器部分。
- 全局内存分配器初始化：axalloc 定义了一个静态全局变量

GLOBAL_ALLOCATOR，并实现了 core::alloc::GlobalAlloc 特性，将其注册为标准库的默认分配器。在初始化时，需要调用 global_init 函数，该函数会初始化页分配器和字节分配器。

- 内存分配与释放：包括字节分配和释放、页面的分配和释放。
- 内存状态查询：axalloc 提供了一些函数用于查询内存分配器的状态，如当前内存使用量、空闲内存量等。

另外，axalloc 模块提供了用于管理全局内存页面的类型 GlobalPage，其属性包括起始地址和包含的页数。并在全局内存分配器的支持下，提供分配单页和连续页面、填充页面等接口。同时，通过实现 Drop 特性，确保在 GlobalPage 被销毁时，自动释放其占用的内存。这种机制确保了内存资源的高效利用，避免了内存泄漏。

2.2.3 axmm

使用 axalloc 模块提供的内存管理功能，我们已经可以构建起一个简单的内存管理系统，支持内核和应用程序都处于同一地址空间，并且相互可见。但是，在多任务或多进程的环境下，内核和应用程序之间的内存隔离是非常重要的。为了实现这一点，ArceOS 引入了 axmm 模块来实现虚拟内存管理，其结构如图2.9所示。

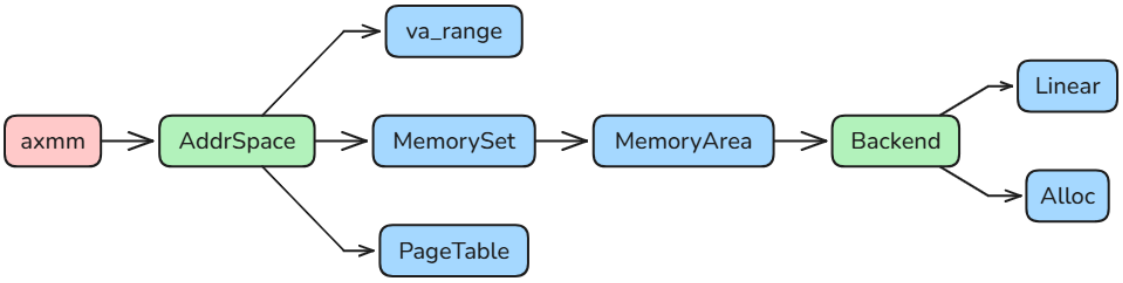


图 2.9 axmm 结构图

axmm 模块的核心是 AddrSpace 结构体，它表示一个虚拟内存地址空间，包含虚拟地址范围、内存区域集合以及页表。通过该结构体提供的接口，可以对虚拟内存地址空间进行管理，包括创建、映射、解除映射、读写数据等操作。AddrSpace 的具体接口如附录中图B.1所示。

实现 AddrSpace 结构体及其接口对于构建一个高效、灵活且安全的虚拟内存管理系统具有极其重要的意义。一方面，通过为每个进程分配独立的虚拟地址空间，AddrSpace 不仅实现了内核与用户空间的隔离，还确保了进程之间的相互独立，从而防止了进程间的非法访问和潜在的破坏行为。

另一方面，AddrSpace 提供的内存保护机制进一步增强了系统的安全性。它通

过页表和访问权限控制，确保每个进程只能访问自己被授权的内存区域，同时在检测到非法访问时能够及时触发异常并进行处理，从而避免系统崩溃。这种机制不仅保护了内核的稳定性，还为用户程序提供了可靠的运行环境。

除 AddrSpace 结构体及其接口外，axmm 模块还定义 Backend 核心抽象，用于处理不同类型的内存映射实现细节。它作为 MemoryArea（内存区域）的后端，将虚拟地址与物理存储关联起来，将映射、解除映射、处理页面异常等操作传递到具体的后端实现，进而完成对页表的操作。axmm 模块支持两种类型的内存映射后端：

- Linear Backend：用于与已知物理地址的直接映射，例如内存映射的 I/O 区域或内核映像部分，虚拟地址和物理地址具有固定的偏移关系；
- Allocation Backend：按需分配物理页，支持延迟分配，常用于用户空间内存管理。

内存区域对应的映射类型在内存区域对象创建时，即调用 AddrSpace 的 map_linear 或 map_alloc 方法申请空间并进行映射时指定。

2.2.4 axdma

axdma 模块实现了 DMA 内存管理，提供了直接内存访问的接口。DMA（Direct Memory Access，即直接内存访问）是一种允许特定硬件子系统绕过 CPU 直接访问系统内存的硬件特性，通常用于高速数据传输，例如从硬盘读取数据到内存，或者从内存将数据发送到网络接口。

在传统的数据传输中，CPU 需要逐字节地从外设读取数据，然后将其写入内存，或者从内存读取数据后写入外设。这种操作不仅效率低下，还会占用大量的 CPU 时间。而 DMA 允许外设直接访问内存，无需 CPU 参与，从而提高了数据传输的速度和效率，并释放了 CPU 的计算资源，使其可以处理其他任务。

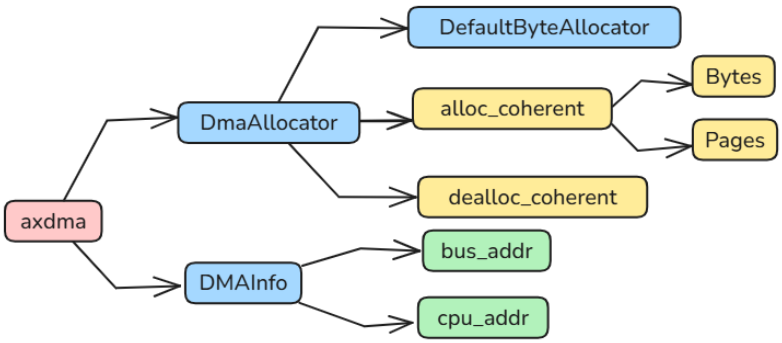


图 2.10　axdma 结构图

如图2.10，ArceOS 的 DMA 模块通过实现 DMA 分配器（DmaAllocator），向

设备驱动模块（axdriver）提供了 DMA 内存分配和释放的接口。它会根据设备的需求，分配合适大小的内存块，并在 DMA 操作完成后释放这些内存块。DMA 分配器还会处理 DMA 操作中的地址转换问题，将虚拟地址转换为总线地址，以便于 CPU 访问。DMAInfo 结构体则记录了用于 DMA 操作的内存区域相关的信息，包括 CPU 访问该区域的虚拟地址和该区域在总线上的物理地址。同时，DMA 分配器分配的内存需要标记为 UNCACHED，以避免缓存一致性问题，这需要虚拟内存管理模块（axmm）提供的接口来完成。

DMA 模块的具体接口如表2.1所示：

表 2.1　DmaAllocator 结构体接口

接口名	功能
alloc_coherent	分配用于 DMA 操作的内存区域
alloc_coherent_bytes	分配小内存块（<4KB）
alloc_coherent_pages	分配大内存块（≥4KB）
update_flags	修改指定内存区域的映射标志
dealloc_coherent	释放之前分配的 DMA 内存区域
virt_to_bus	将虚拟地址转换为总线地址，用于设备访问
layout_pages	计算给定布局所需的页数

2.3　ArceOS 内存管理机制

在本章第一节中我们提到了 ArceOS 的启动和初始化过程，而在这里我们将对其中与内存分页相关的部分进行详细分析：架构的初始化函数 _start 中进行了启动页表和内存管理单元的初始化，为 QEMU 虚拟机的物理内存等关键区域构建映射，设置根页表地址并刷新 TLB，完成分页的早期启动阶段。这一阶段是 ArceOS 默认开启的，构建的映射为恒等映射，虚拟地址和物理地址相同，主要用于内核的启动和初始化。

```
phys_virt_offset = const PHYS_VIRT_OFFSET,
boot_stack_size = const TASK_STACK_SIZE,
boot_stack = sym BOOT_STACK,
init_boot_page_table = sym init_boot_page_table,
init_mmu = sym init_mmu,
```

清单 2.1　ArceOS 中 _start 启动函数内存初始化相关代码

如图 2.11所示，分页的早期启动阶段完成后，如果开启了 paging 特性，在 rust_main 函数中会调用 axmm 模块的 init_memory_management 函数初始化内存管

理，即创建内核的虚拟地址空间，为内核的虚拟地址空间构建线性映射，调用 axhal 的接口设置内核的虚拟地址空间的页表，并将其设置为当前的页表根，完成分页的重建映射阶段。该阶段将内核和用户空间的虚拟地址空间隔离开来，让内存管理模块能够管理更大的地址空间，为 DMA 和多任务等功能提供支持。

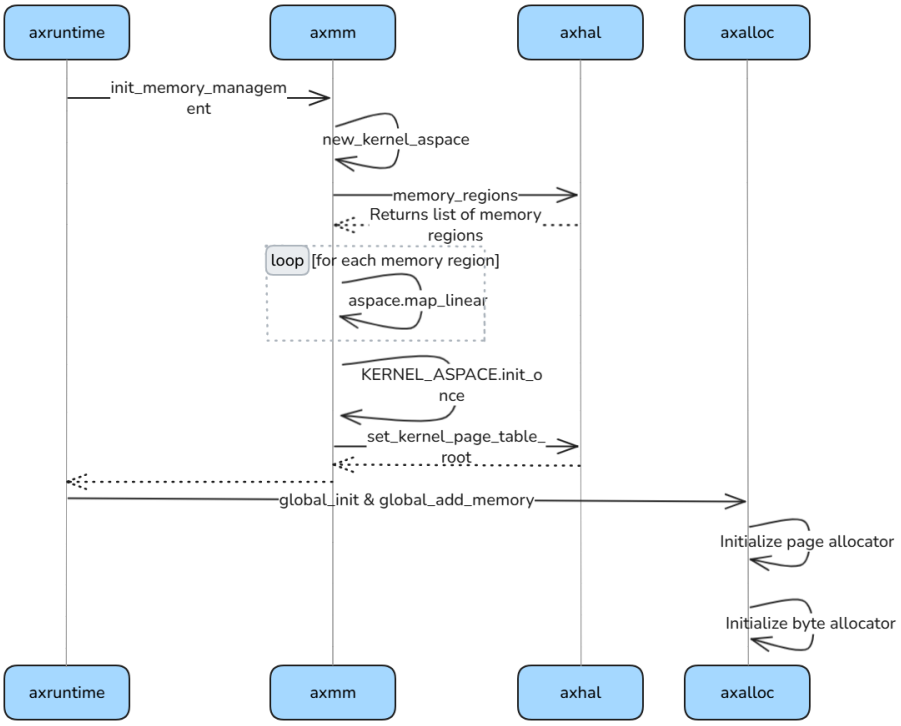


图 2.11　分页的重建映射阶段

开启 paging 特性且上层架构或者应用需要对内存空间进行操作时，会调用 AddrSpace 的相应接口，如 find_free_area（寻找空闲内存区域）、map_linear（建立线性映射）、map_alloc（建立分配映射）等方法，这些方法会根据具体的内存操作需求，对页表和虚拟地址空间进行操作，并调用全局内存分配器进行物理内存的分配和释放。全局分配器在分配时会先尝试从字节分配器分配，如果字节分配器内存不足，则计算需要多少内存，从页面分配器分配页面后将页面添加到字节分配器重新进行分配，此方法允许高效处理小型和大型分配，同时保持合理的内存使用率。

同时，ArceOS 还在 AddrSpace 结构体中提供了页面异常的处理接口，例如当访问的虚拟地址对应的物理页不存在时，会触发缺页异常，此时会调用 AddrSpace 的 handle_page_fault 函数处理缺页异常，该函数会对后端为 Alloc 类型的地址，调用 axalloc 中的全局内存分配器进行物理页的分配，并通知 axhal 中的页表更新映射关系。对于后端为 Linear 类型或者需要填充的地址，则会处理失败，返回 false，因为这两类的地址在创建时就应完成物理页的分配和映射，缺页异常的处理不适

用。AddrSpace 的 handle_page_fault 函数的流程图如附录中图 B.2所示。

借助缺页异常的处理，ArceOS 还实现了 Lazy Map 机制，即当创建不需要立即填充的虚拟内存区域时，不会立即分配物理页，而是在第一次访问该虚拟地址时才触发缺页异常，进行分配和映射。

第 3 章　starry-next 宏内核

3.1　starry-next 宏内核整体架构

starry-next (StarryOS) 是构建在 ArceOS 基础之上的操作系统内核。它实现了一个兼容 Linux 的系统调用接口，同时支持多种硬件架构，包括 x86_64、riscv64、aarch64 和 loongarch64。

代码结构上，starry-next 主要由以下模块组成：

- core：基于 ArceOS 的底层支持实现 StarryOS 的核心内核功能，包括内存管理（mm）、任务管理（task）和时间管理（time）等。其中内存管理模块负责分配、释放和映射，如 copy_from_kernel 函数用于将内核空间的映射复制到用户地址空间，load_user_app 函数用于加载用户应用程序到内存中。任务管理模块处理任务和线程的创建、调度和管理，例如 new_user_task 函数用于创建新的用户任务。时间管理模块管理系统时间和定时器。
- api: 现面向用户的 API 和系统调用接口，涵盖文件系统操作、进程管理、线程管理、内存管理等多个方面。
- src：主要负责启动主函数、加载应用程序并对系统调用的分发进行处理。
- 其他：包括用于适配不同架构的配置文件和工具链，以及用于构建和运行 StarryOS 的构建工具和脚本。

功能上，starry-next 也可以划分为较为独立的模块：

- 内存管理模块：负责管理物理内存和虚拟内存，包括内存分配、映射和释放等操作。
- 进程管理模块：负责管理进程和线程，包括进程创建、调度和管理等操作。
- 文件系统模块：负责管理文件系统，包括文件的创建、读取、写入和删除等操作。
- 网络模块：负责管理网络操作，包括网络协议栈、网络设备驱动等操作。
- 系统信息模块：负责提供系统信息和状态，包括 CPU、内存、网络等信息。
- 信号模块：负责处理信号和中断，包括信号的发送、接收和处理等操作。

每个功能模块都需要向应用程序提供相应的系统调用接口，为应用程序提供内核服务。同时，功能模块还需要尽可能复用 ArceOS 的底层组件和接口，以减少代码的重复和维护成本。功能模块之间通过明确定义的接口进行通信，确保系统的稳定性和可拓展性。

通过结构和功能上的划分与解耦，Starry-next 宏内核的每个模块都可以独立开

发、测试和维护，增强了系统的可拓展性和稳定性，也使得本课题大方向上的多名开发者可以在同一代码库上进行协作开发。

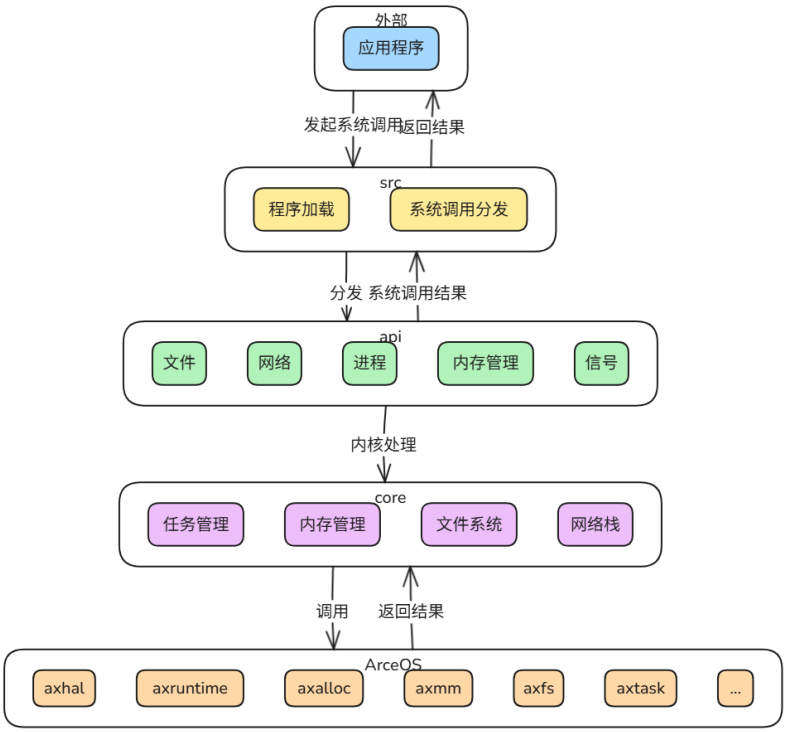


图 3.1　starry-next 宏内核架构

3.2　starry-next 初始化和运行过程

starry-next 的初始化从 Bootloader 将控制权交给内核入口点开始，之后 ArceOS 的 axhal 模块和 axruntime 模块会完成硬件的初始化，并根据选择的特性进行内存管理器、任务管理器、文件系统、网络等模块的初始化。接着，axruntime 模块以 src 中 main 函数创建一个初始化进程，进而为环境变量指定的测试用例创建用户应用程序，并将用户程序加载到内存中。

在运行过程中，starry-next 会逐个运行应用程序，并根据系统资源的需求，动态地分配和管理内存。当应用程序发起系统调用时，先由 src 层将参数根据系系统调用号分发到 api 层的相应处理函数，api 层的函数会检查参数的类型、取值范围等是否符合系统调用的规范。验证参数无误后，api 层函数会根据系统调用的具体功能，调用 core 层的相关功能模块及其底层的 ArceOS 组件来完成实际的操作。当 core 层完成操作并返回结果后，api 层会对返回结果进行处理。如果操作成功，api 层会将结果按照系统调用的约定进行封装，以便返回给应用程序。如果操作失败，api 层会根据错误类型设置合适的错误码。最后，api 层的函数会将处理结果返回给 src 层，src 层会将结果返回给应用程序。

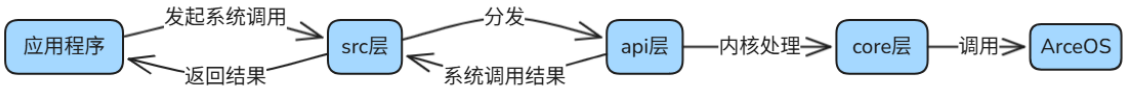


图 3.2　系统调用流程

3.3　加载程序到内存空间

在上一节的初始化过程中提到，要运行应用程序，首先需要将其加载到内存空间中。本节将详细介绍 starry-next 是如何加载应用程序到内存空间并执行的。

加载应用程序到内存空间中，首先要创建一个新的用户地址空间，并将内核地址空间的映射复制到用户地址空间，同时映射信号处理的跳板代码。这一步在 run_user_app 函数中完成。接下来，调用 load_user_app 函数加载将用户应用程序加载到用户地址空间。该函数使用 axfs::api::read 函数从文件系统中读取用户应用程序的 ELF 文件数据并进行解析，如果 ELF 文件指定了动态链接器（解释器），则递归调用 load_user_app 加载。其中，解释器是一种能够将源代码逐行翻译成机器代码并执行的程序，负责加载和解析可执行文件所依赖的共享库。然后使用 map_elf 函数将 ELF 文件中的段映射到用户地址空间中，这一步使用了 map_alloc 进行的物理页的分配和映射。最后，分配用户栈和堆空间，并将参数、环境变量和辅助向量（auxv）写入用户栈，返回应用程序的入口地址和用户栈顶地址。

完成程序的加载后，内核会根据用户地址空间创建任务实例，设置任务上下文等相关信息，并将该任务与新的进程和线程关联起来。最后调用 axtask::spawn_task 接口将任务添加到任务调度器中，等待调度器调度执行。run_user_app 和 load_user_app 函数的流程如图B.3和B.4所示。

3.4　starry-next 内存管理模块

3.4.1　模块结构

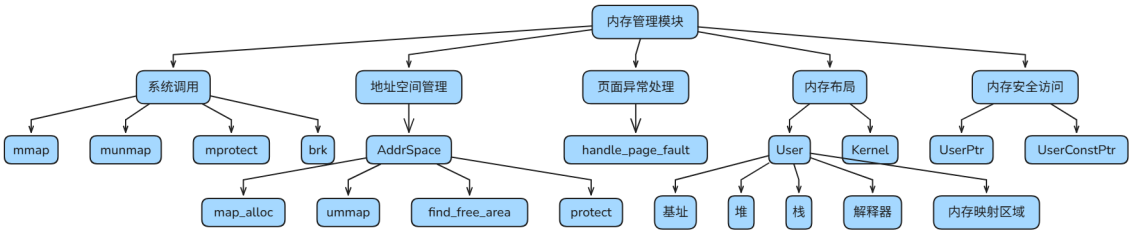


图 3.3　starry-next 内存管理模块结构

starry-next 的内存管理模块主要由以下部分组成：

- 系统调用：提供了一系列系统调用接口，用于管理内存，如 mmap、munmap、mprotect、brk 等。
- 地址空间管理：使用 AddrSpace 结构体及其接口管理用户程序的地址空间，包括内存映射、权限控制等。
- 页面异常处理：当用户程序访问内存空间时触发的缺页异常会调用相应的处理函数。
- 内存布局：定义了内存布局，包括内核空间和用户空间的划分。
- 安全内存访问：通过指针封装提供对用户空间内存的安全访问。

3.4.2 内存管理机制

starry-next 的内存管理基本上复用了 ArceOS 的内存管理机制: 使用 ArceOS 提供的 AddrSpace 结构体管理用户程序的内存空间，当用户程序需要对内存进行操作时，会调用相应的系统调用，如 mmap、munmap、mprotect、mremap 等，这些系统调用会调用 AddrSpace 结构体的接口在用户地址空间中创建和管理内存映射，同时也会调用全局内存分配器进行物理页的分配和映射。

但作为宏内核，starry-next 需要区分内核态和用户态，因此还对内存管理进行了进一步的优化和扩展，例如为用户态和内核态分别设置了不同的地址空间，并且用户态进程不能直接访问内核态内存空间，而是需要发出系统调用，通过 trap 机制进入内核态，再进行对应处理，通过这种方式来实现用户态和内核态的隔离；在一些架构中将内核部分的映射复制到用户地址空间，使用户态进程可以直接访问内核的某些数据结构或代码片段，可以减少内核态和用户态之间的切换开销，提高系统的性能。

3.4.3 页面异常处理

页面异常的处理方面，starry-next 在 ArceOS 的基础上增加了判断：如果是用户态程序访问内存空间或者是内核态程序访问用户内存空间时触发的缺页异常，才会调用用户程序的 handle_page_fault 函数，也就是 ArceOS 的 AddrSpace 结构体提供的页面错误处理函数处理缺页异常，该函数会根据虚拟地址的映射关系，调用全局内存分配器进行物理页的分配和映射。如果 AddrSpace 结构体不能成功处理该异常，则会打印错误信息并终止当前任务，并发送 SIGSEGV 信号。

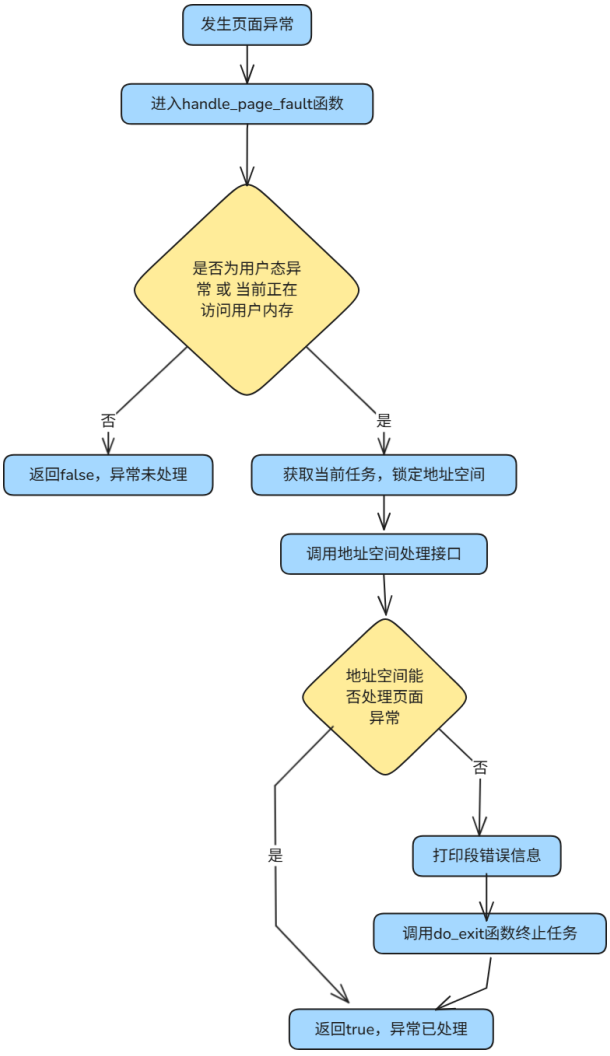


图 3.4　starry-next 页异常处理

3.4.4　内存布局

不启动 ArceOS 的 paging 特性时，应用和内核在同一空间中运行，系统直接使用物理地址。物理空间的结构如图3.5所示。在 Cargo 项目启动时，axhal 组件的 build.rs 构建文件会根据根据 linker.lds.S 文件中的链接脚本，将物理空间划分为多个部分——内核部分从指定的基地址开始，首先放置可执行代码段（.text），然后是只读数据段（.rodata）和初始化函数数组（.init_array）。接着是可读写数据段（.data），线程局部存储相关段（.tdata 和.tbss），以及每个 CPU 核心的私有数据段（.percpu）。最后是未初始化数据段（.bss），其中包含了启动栈空间，设备内存则位于 0 地址和内核空间之间。整个布局严格遵循 4K 对齐，并在最后定义了一些与中断处理、系统调用等相关的自定义段。

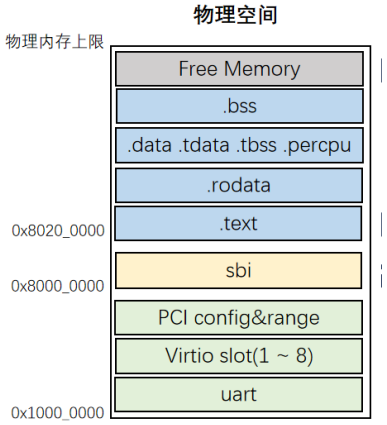


图 3.5　物理内存布局

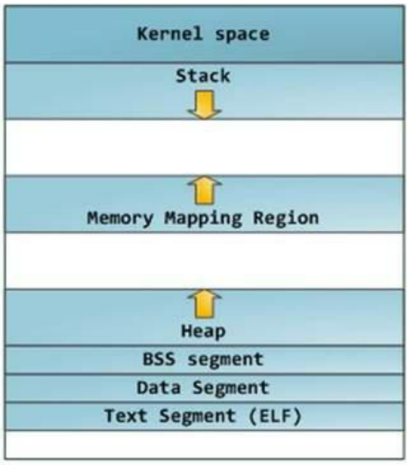


图 3.6　进程虚拟空间布局

starry-next 启动了 paging 特性，为每个应用程序分配一个独立的的虚拟地址空间，即 AddrSpace 结构体，用于管理用户程序的内存空间。如图3.6所示，进程地址空间分为用户空间和内核空间两部分，用户空间位于低地址，包括了代码段、数据段、堆、栈、内存映射区域。其中，代码段和数据段是用户程序的只读和读写数据，堆是用户程序的动态内存分配区域，栈是用户程序的函数调用和局部变量存储区域。内核空间位于高地址，包括了内核代码段、内核数据段、内核堆、内核栈、页表。不同架构下各部分的具体起始地址和大小会有所不同，具体设置可参照表3.1。

表 3.1　各架构虚拟空间设置

	x86_64	RISC-V 64	AArch64	LoongArch64
User space base	0x1000	0x1000	0x1000	0x1000
User space size	128 TB	4 GB	128 TB	4 GB
User interpreter base	64 MB	64 MB	64 MB	64 MB
User heap base	1 GB	1 GB	1 GB	1 GB
User heap initial size	64 KB	64 KB	64 KB	64 KB
User stack top	0x7fff00000000	0x400000000	0x7fff00000000	0x400000000
User stack size	64 KB	64 KB	64 KB	64 KB
Kernel stack size	256 KB	256 KB	256 KB	256 KB

3.4.5　安全内存访问

为了保证系统的安全性和稳定性，starry-next 对用户空间内存的访问进行了严格的控制和保护，通过指针封装提供对用户空间内存的安全访问。具体来说，starry-next 定义了 UserPtr 和 UserConstPtr 两个安全指针类型，分别表示指向用户空间内

存的可变和不可变指针。在访问用户空间内存时，要检查地址是否对齐、当前任务是否有权限访问该地址、地址是否在用户空间范围内并于已经分配映射。如果这些条件都满足，则可以安全地访问用户空间内存，否则会抛出异常并终止当前任务。

UserPtr 和 UserConstPtr 提供了一系列接口，用于对用户空间内存进行访问和修改。例如：负责基本信息的查询的 address（获取指针所指向的虚拟地址）和 is_null（判断指针是否为空）和获取切片和引用的 get_as_mut 和 get_as_mut_slice 等。

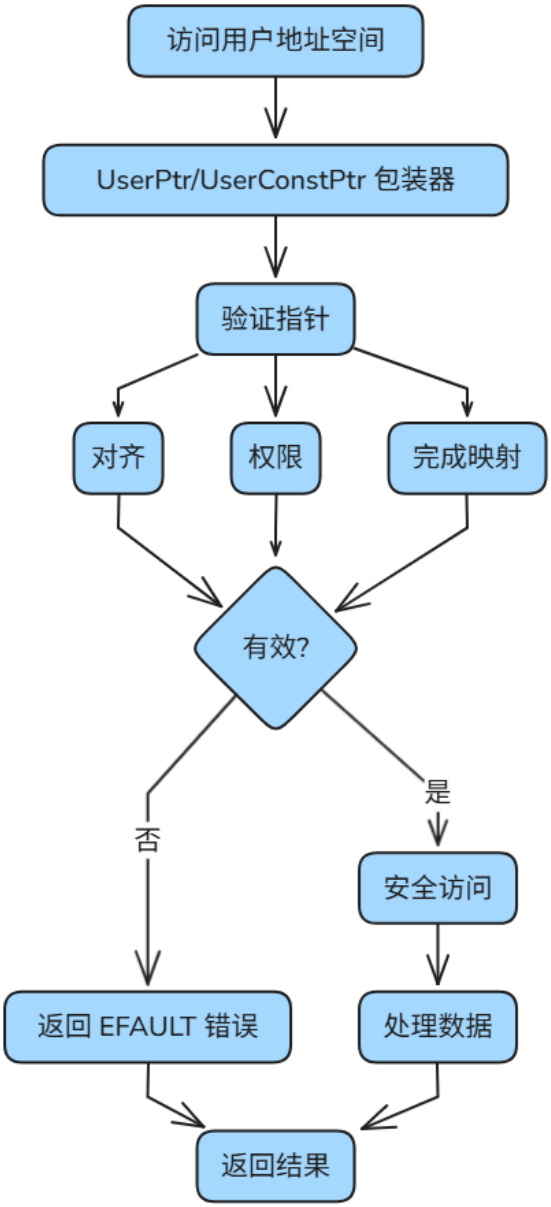


图 3.7　安全内存访问流程

第 4 章　内存管理子系统接口的设计与实现

4.1　内存管理子系统概述

前两章中我们分别介绍了 ArceOS 基座中的内存管理组件和 starry-next 的内存管理模块，而本小节则将二者串联起来，从 crate（即 Rust 模块，本节简称为模块）层面介绍内存管理子系统的结构。

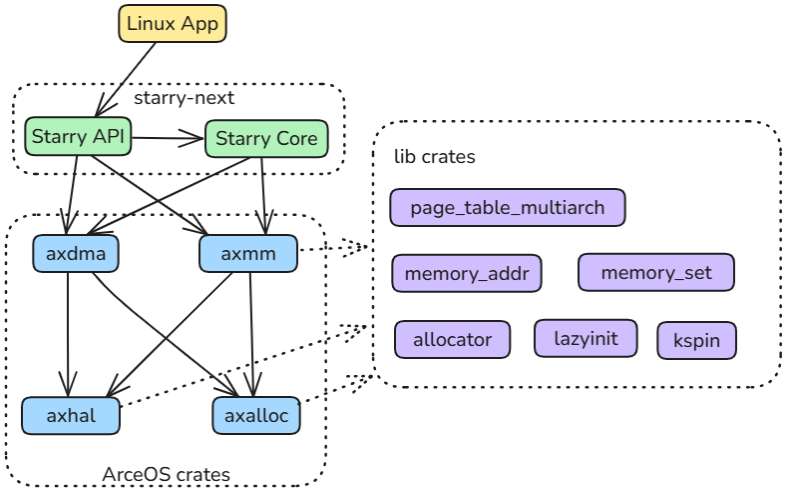


图 4.1　内存管理子系统结构图

如图4.1所示，starry-next 中的 API 模块为应该提供系统调用支持，并根据需要调用 Core 模块中的方法。而在这两个模块的功能又依赖于 ArceOS 中的模块，例如 axmm 模块提供了内存管理的基本功能，axdma 模块提供了 DMA 相关的功能。ArceOS 中的模块间也存在相互依赖关系，例如 axmm 和 axdma 依赖 axalloc 提供的全局内存分配器，并且需要 axhal 提供不同架构的页表抽象相关功能和 axconfig 提供相关配置信息。在 ArceOS 的下层，是 Rust 和第三方库的支持，例如 page_table_multiarch 提供了页表抽象的支持，memory_addr 和 memory_set 提供内存地址和内存集合的抽象，lazyinit 和 kspin 提供了懒初始化和自旋锁的支持等。

内核启动后，先通过 axhal 和 axruntime 进行初始化，包括页表初始化、全局内存分配器的初始化和分配内核空间等。然后启动 starry-next 的 main 函数，加载用户程序、分配用户空间和通过任务管理子系统进行任务调度。在用户程序运行过程中，如果需要进行内存分配或释放等操作，则通过 mmap、munmap 等系统调用进入 starry-next 的 API 模块，进而进入 ArceOS 的 axmm 等模块进行内存管理操作。任务结束后，任务管理子系统会调用 axmm 等模块进行内存等资源的释放和

回收。

4.2 内存管理子系统直接相关的系统调用

4.2.1 mmap 系统调用

mmap 用于将文件或设备内存映射到进程的地址空间。它在 Linux 和类 Unix 系统中广泛使用，主要用于高效地读写文件、实现内存共享以及分配匿名内存等。

mmap 系统调用的原型如下：

```
void *mmap(void addr[.length], size_t length, int prot, int
   flags, int fd, off_t offset);
```

清单 4.1 mmap 系统调用函数原型

可以看到，mmap 系统调用共有 6 个参数，其中 addr 和 length 用于指定内存映射的起始地址和长度，prot 用于指定内存映射的保护属性，flags 用于指定内存映射的标志，fd 用于指定要映射的文件描述符，offset 用于指定文件偏移量。当 addr 为 NULL 时，表示由内核自动选择内存映射的起始地址。在 Linux 中，内核会选择一个附近的页面边界，并尝试创建映射。如果该地址已经存在另一个映射，内核会选择一个新的地址。flags 参数决定了对映射区域的更新是否对映射同一区域的其他进程可见，以及更新是否会被写入底层文件，例如，MAP_SHARED 标志表示对映射区域的更新对所有共享该映射的进程可见，而 MAP_PRIVATE 标志则表示对映射区域的更新对其他进程不可见。

mmap 系统调用的返回值是一个指向内存映射区域的指针，该指针指向映射区域的起始地址。如果调用失败，返回值为 MAP_FAILED（通常为 -1）。

内存映射主要分为两种类型：匿名映射是指不与任何文件关联的内存映射，通常用于分配动态内存，类似于 malloc，但提供了更多的灵活性和控制能力，例如多个进程可以共享同一个内存映射区域；文件映射是指将文件的内容映射到内存中，这样就可以像访问内存一样访问文件的内容，通常用于实现文件的高效读写。

如图4.2所示，在 starry-next 中，mmap 系统调用的实现主要包括以下几个步骤：首先解析参数并检查其有效性；然后进行地址对齐处理，根据是否指定映射起始地址选择将映射长度向上对齐到 4KB （页面大小）或将映射的起始和结束地址分别对齐到页面边界；接着确定映射的真正地址：如果 flag 中设置了 MAP_FIXED 标志，则使用对齐后的 addr 地址作为映射的起始地址，并且如果 addr 和 length 指定的内存区域与任何现有的映射重叠，那么现有映射的重叠部分将被丢弃，否则，内核会在指定地址附近或者整个用户地址空间中寻找符合条件的空闲地址，并调用 AddrSpace 结构体的 map_alloc 方法创建映射；最后返回起始地址；如果参数中

fd 不为 -1 且 MAP_ANONYMOUS 标志未设置，即该映射是文件映射，则在返回前需要从文件中读取数据填充到内存中。

map_alloc 是 ArceOS 提供的一个方法，用于创建内存映射。会先验证要映射的区域，然后创建一个新的内存区域，并将其映射到页表中。在构建映射的过程中，会根据映射的类型（线性映射或分配映射），选择不同的后端进行处理。同时还需要为映射区域分配物理页，并在页表中记录。

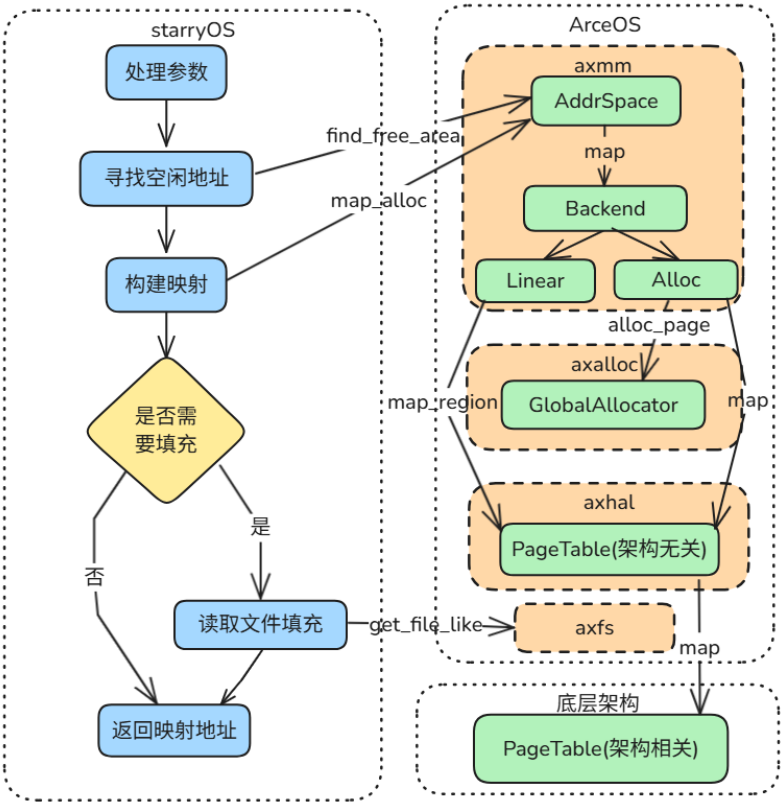


图 4.2　mmap 系统调用流程

需要注意的是，由于 Lazy Map 机制的存在，不需要填充的内存区域不会立即被分配物理页，而是在第一次访问该内存区域并触发页面异常时才会分配物理页。该机制可能会导致内核在访问内存区域时出现 NotMapped 错误，因此需要在代码中调用 handle_page_fault 函数，为内存区域分配物理页，具体可以参考 AddrSpace 结构体的 clone_or_err 方法。

```
let new_addr = match new_aspace.pt.query(vaddr) {
    Ok((paddr, _, _)) => paddr,
    // If the page is not mapped, try map it.
    Err(PagingError::NotMapped) => {
        if !backend.handle_page_fault(vaddr, area.flags(), &mut
            new\_aspace.pt) {
            return Err(AxError::NoMemory);
```

```
            }
            match new_aspace.pt.query(vaddr) {
                Ok((paddr, _, _)) => paddr,
                Err(_) => return Err(AxError::BadAddress),
            }
        }
        Err(_) => return Err(AxError::BadAddress),
};
```

清单 4.2　Lazy Map 机制下访问内存区域的处理

除成功返回映射的起始地址外，starry-next 中 mmap 还有其他的返回情况：MAP_FIXED 标志设置但地址为空，或者需要从文件读取数据填充映射区域但偏移量 offset 小于 0 或者超出文件大小，函数会返回 EINVAL 错误；若系统无法找到足够的连续空闲内存区域来完成映射操作，函数会返回 ENOMEM 错误；当需要从文件读取数据填充映射区域时，若文件描述符 fd 对应的文件对象无法转换为 arceos_posix_api::File 类型，函数会返回 EBADF 错误。

4.2.2　munmap 系统调用

munmap 用于解除进程地址空间中的内存映射。munmap 系统调用的原型如下：

```
int munmap(void addr[.length], size_t length);
```

清单 4.3　munmap 系统调用函数原型

munmap 系统调用共有 2 个参数，其中 addr 用于指定要解除映射的内存区域的起始地址，length 用于指定要解除映射的内存区域的长度。

munmap 系统调用的返回值是一个整数，通常为 0 表示成功，-1 表示失败。

如图4.3，starry-next 中，munmap 系统调用的实现主要包括以下几个步骤：首先解析参数并检查其有效性；然后调用 AddrSpace 结构体的 unmap 方法，该方法会将指定的内存区域从页表中解除映射；刷新 TLB；最后返回 0 表示成功。

ArceOS 对 munmap 系统调用的支持方式与 mmap 系统调用类似，都是通过 AddrSpace 结构体向上提供接口，通过映射后端调用全局分配器和页表进行内存解除映射并释放物理页。

另外，和 Linux 的标准实现相似，当需要取消映射的区域和当前存在的内存区域有重叠部分时，会将重叠部分的映射关系从页表中删除，并调整内存区域的起始地址和结束地址，如图4.4所示，共存在三种情况：

1. 完全覆盖：要解除映射的内存区域完全覆盖了当前存在的内存区域，此时需要将当前存在的内存区域从页表中删除，并将其从内存区域集合中移除。如图中第一项。

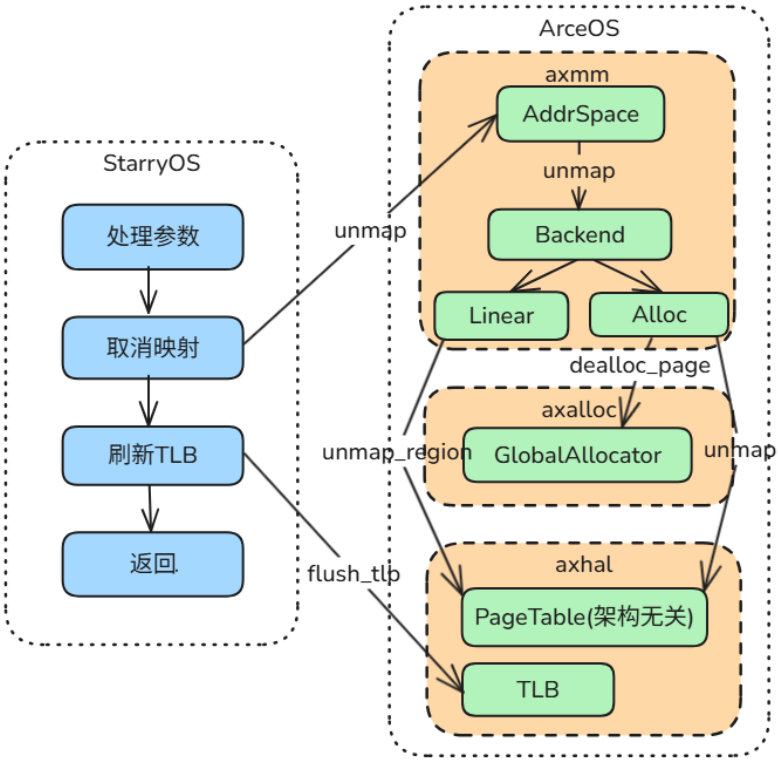


图 4.3　munmap 系统调用流程

2. 部分覆盖：要解除映射的内存区域位于当前存在的内存区域的中间部分，此时需要将当前存在的内存区域分为两部分，分别是要解除映射的内存区域的前半部分和后半部分，并加入到内存区域集合中，如图中第二项。
3. 前后覆盖：要解除映射的内存区域包括当前存在的内存区域的左边界或右边界，需要调整当前存在的内存区域的起始地址或结束地址，如图中第三和第四项。

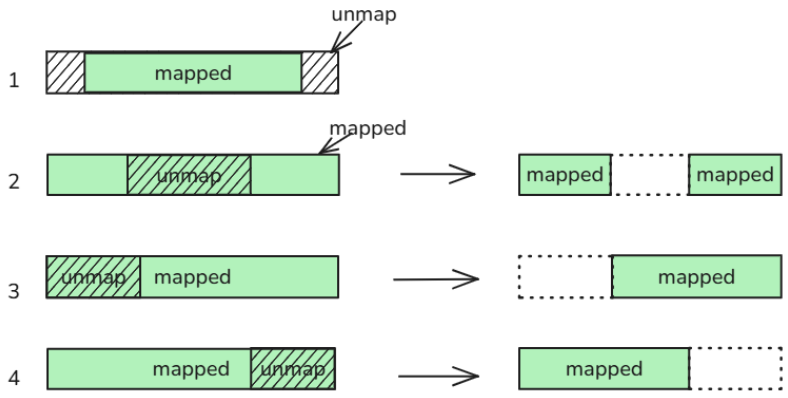


图 4.4　munmap 处理交叉区域

4.2.3 brk 系统调用

brk 用于改变进程的堆顶位置。其原型如下：

```
int brk(void addr);
```

清单 4.4 brk 系统调用函数原型

可以看到，brk 系统调用只有一个参数，即 addr，用于指定数据段的结束位置。其返回值是一个整数，通常为 0 表示成功，-1 表示失败。

starry-next 中，brk 系统调用通过设置任务信息结构体的 heap_top 字段来改变堆顶地址。具体来说，brk 系统调用会将 heap_top 字段设置为 addr，然后返回新的堆顶，但是如果 addr 小于当前 heap_top，或者 addr 到堆底的距离大于进程的最大数据大小限制，那么 brk 系统调用会放弃设置新的堆顶并返回原来的堆顶地址。

需要注意的是，brk 系统调用只会改变进程的堆顶位置，并不会分配实际的物理内存。因此，在使用 brk 系统调用时，需要确保进程已经分配了足够的物理内存来支持新的堆顶位置。

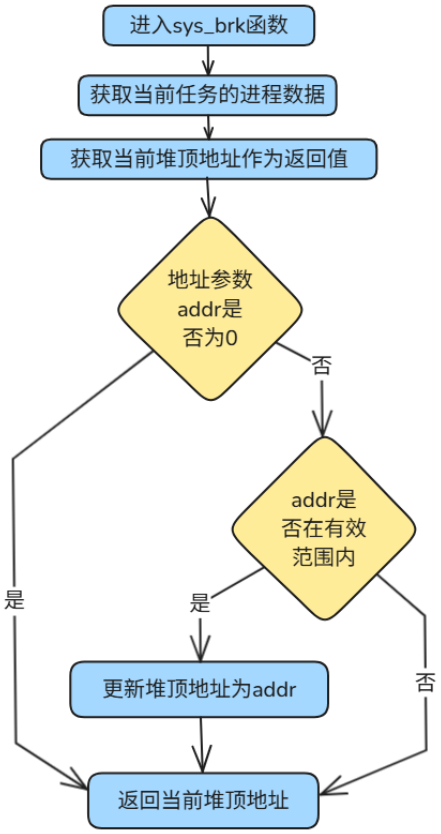


图 4.5 brk 系统调用流程

4.2.4 mprotect 系统调用

mprotect 用于修改指定内存区域的保护属性。mprotect 系统调用的原型如下：

```
int mprotect(void addr[.len], size_t len, int prot);
```

清单 4.5 mprotect 系统调用函数原型

mprotect 系统调用接受三个参数：addr（要修改权限的内存区域的起始地址，必须是页面对齐的）、len（要修改权限的内存区域的长度）以及 prot（指定新的访问权限）。prot 参数可以是 PROT_NONE（无法访问）、PROT_READ（可读取）、PROT_WRITE（可写入）、PROT_EXEC（可执行）、PROT_GROWSDOWN（向下增长）、PROT_GROWSUP（向上增长）的按位或组合。如果进程尝试以违反保护的方式访问内存，内核将生成一个 SIGSEGV 信号。

mprotect 系统调用的返回值是一个整数，通常为 0 表示成功，-1 表示失败。可能的错误包括：EFAULT——addr 指向的地址超出了进程的地址空间、EINVAL——len 为 0 或 prot 包含无效的标志、EACCES——进程没有足够的权限来修改内存区域的保护属性、ENOMEM——内存不足等。

如图4.6所示，在进行地址长度页对齐等参数处理后，starry-next 使用了 AddrSpace 结构体的 protect 方法，进而通过 axhal 组件的页表接口来修改内存区域的保护属性。指定范围和当前存在的内存区域的关系和处理方式与 munmap 系统调用基本一致。但暂未实现 PROT_GROWSDOWN 和 PROT_GROWSUP 标志的处理。

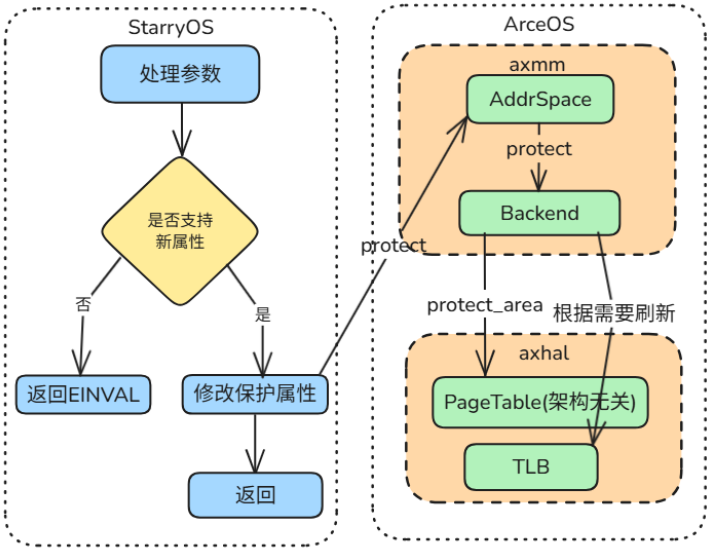


图 4.6 mprotect 系统调用流程

4.3 内存管理子系统间接相关的系统调用

4.3.1 sysinfo 系统调用

sysinfo 系统调用用于获取系统的信息，如内存使用情况、CPU 信息、文件系统信息等。sysinfo 系统调用的原型如下：

```
int sysinfo(struct sysinfo *info);
```

清单 4.6 sysinfo 系统调用函数原型

sysinfo 系统调用只有一个参数，即 info，用于存储系统信息的结构体指针。该结构体包含多个字段，例如 uptime（系统运行时间）、loads（1、5 和 15 分钟内的平均负载）、totalram 和 freeram（总内存和空闲内存）、sharedram 和 bufferram（共享内存和缓冲区内存）等。

sysinfo 系统调用与包括内存管理在内的组件相关，例如通过 ArceOS 的 sys_conf 接口获取 axconfig 组件中的核数等配置信息、axfs 组件的文件限制信息以及全局内存分配器的总页数和空闲页数；通过 sys_clock_gettime 接口获取 axhal 组件的系统运行时间等。

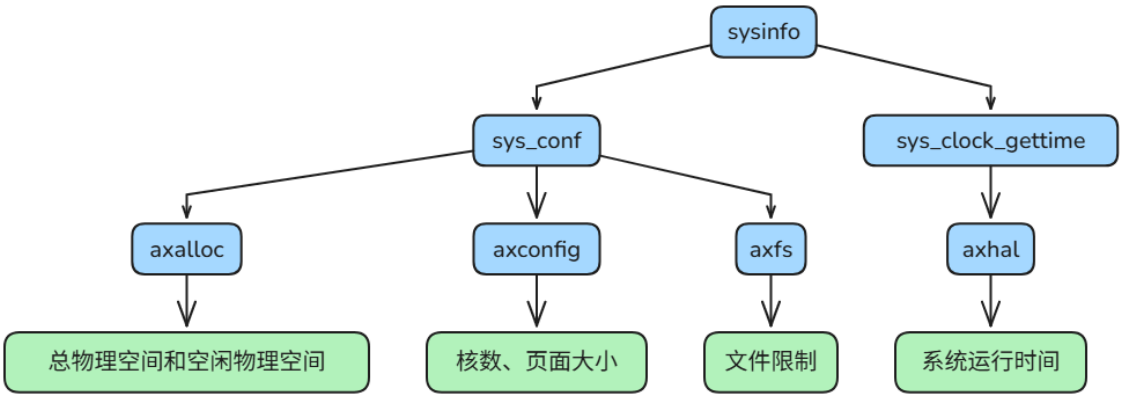


图 4.7 sysinfo 系统调用流程

4.3.2 arch_prctl 系统调用

arch_prctl 系统调用用于设置或获取进程的特定架构相关的参数。其系统调用的原型如下：

```
int syscall(SYS_arch_prctl, int op, unsigned long addr);
int syscall(SYS_arch_prctl, int op, unsigned long *addr);
```

清单 4.7 arch_prctl 系统调用函数原型

arch_prctl 系统调用有两个参数，code 用于指定要设置或获取的参数，addr 用于指定参数的值，设置参数时会读取 addr 指向的内容，读取参数时将结果写入其

指向的地址。常见的操作包括设置或获取进程的指针追踪（Pointer Tracing）状态、设置或获取 GS 段寄存器的值等。

arch_prctl 系统调用的返回值是一个整数，通常为 0 表示成功，-1 表示失败。

当前 starry-next 中，arch_prctl 系统调用支持四个参数的处理：GetFs、SetFs、GetGs 和 SetGs。其中 GetFs 和 SetFs 操作用于获取和设置进程的 FS 段寄存器的值，通过读取和设置 TrapFrame（陷阱帧）中的 TLS（Thread Local Storage，线程本地存储）寄存器实现；GetGs 和 SetGs 操作用于读取和设置进程的内核 GS 段寄存器的值，通过读取和设置 x86 架构下 msr 寄存器实现。

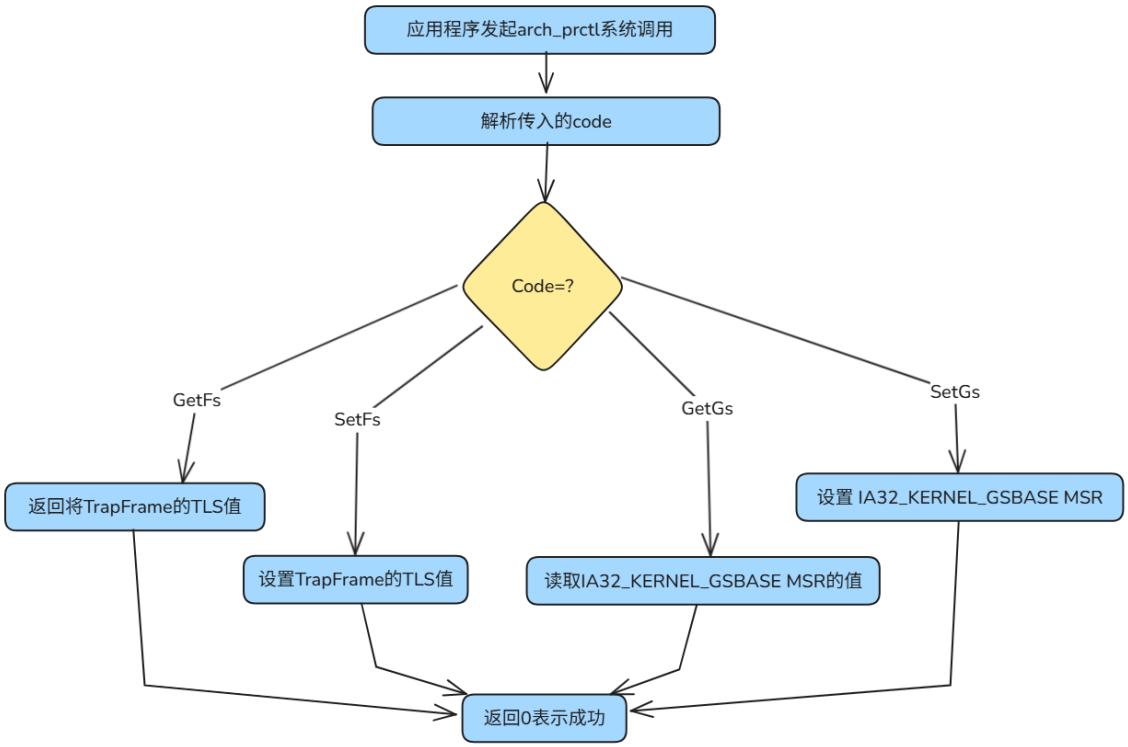


图 4.8　arch_prctl 系统调用流程

4.3.3　prlimit64 系统调用

prlimit64 系统调用用于设置或获取进程的资源限制，涉及操作系统的多个模块，例如栈大小和进程地址空间资源与内存管理模块有关，进程最大打开文件数与文件系统有关。prlimit64 系统调用的原型如下：

```
#include <sys/resource.h>
int prlimit64(pid_t pid, int resource, const struct rlimit64 *
   new_limit, struct rlimit64 *old_limit);
```

清单 4.8　prlimit64 系统调用函数原型

prlimit64 接受四个参数：pid——目标进程的进程 ID，可以是当前进程或另一

个进程）、resource——要查询或设置的资源类型，例如 RLIMIT_CORE（核心转储文件大小）、RLIMIT_CPU（CPU 时间）、RLIMIT_DATA（数据段大小）等）、new_limit——指向 rlimit64 结构体的指针，用于指定新的资源限制，如果为 NULL 则表示仅获取当前限制，以及 old_limit——指向 rlimit64 结构体的指针，用于存储当前的资源限制，如果为 NULL 则不返回旧值。rlimit64 结构体包含 rlim_cur（当前资源限制，又称软限制）和 rlim_max（最大资源限制，又称硬限制）两个字段，软限制是内核对相应资源强制执行的值，硬限制则作为软限制的上限。一个非特权进程只能将其软限制设置为从 0 到硬限制范围内的值，并且（不可逆地）降低其硬限制。

目前 starry-next 支持设置的资源限制包括 RLIMIT_STACK（栈大小）和 RLIMIT_NOFILE（最大打开文件数），二者分别通过修改进程信息结构体的 stack_size 和 max_files 字段来实现，其他类型的资源不会作处理。同时限制了只有当前进程可以设置自己的资源限制，不能设置其他进程的资源限制，当传入的 pid 不为当前进程时，prlimit64 系统调用会返回 EINVAL 错误。

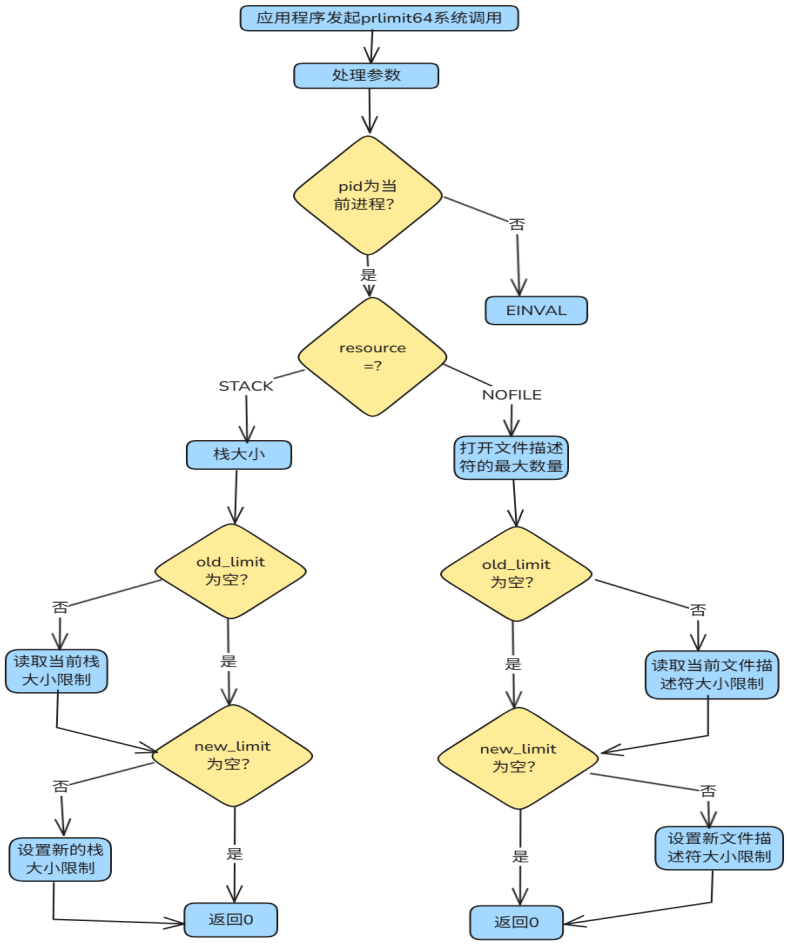


图 4.9　prlimit64 系统调用流程

第 5 章　内存管理组件开发与测试

5.1　开发过程

本节主要介绍内存管理组件的开发过程，包括开发中遇到的困难以及相应的解决方案，并按时间顺序分为前期准备、环境配置、基础实现和进阶开发四个部分。

5.1.1　前期准备

项目的第一到第六周是前期准备阶段，主要包括对 Rust 语言的学习、对操作系统内核的理解和对实验环境的准备。

Rust 语言是一种具有内存安全和并发性特点的系统级编程语言，因而适合用于开发操作系统内核和系统调用接口。ArceOS 和 starry-next 都是基于 Rust 语言开发的，因此在开发内存管理组件时，需要对 Rust 语言有一定的了解和掌握。在本课题前期，我们参考了《Rust 语言升级》等相关资料，并结合操作系统训练营基础阶段的练习题目，深入学习了 Rust 语言的基本语法与特性。其中，所有权、借用、生命周期等核心概念尤为重要，它们在支持操作系统内核的并发性和安全性方面发挥着关键作用。

在操作系统内核的学习方面，我们以 rCore-Tutorial 为例，学习如何实现一个简单的操作系统内核。rCore-Tutorial-v3 是一个用 Rust 语言实现的以教学为目的的基本功能完备的操作系统，根据学习的顺序可分为九个章节：实验环境配置、应用程序与基本执行环境、批处理系统、多道程序与分时多任务、地址空间、进程及进程管理、文件系统与 I/O 重定向、进程间通信、并发，每个章节都有对应的代码分支和练习。跟随 rCore-Tutorial 的学习，能够对操作系统的基本概念和实现有一个清晰的了解，为后续开发内存管理组件提供了基础。

同时，操作系统训练营的项目基础阶段详细介绍了 Unikernel 的搭建流程，包括内存管理、任务调度、文件系统等组件的原理和实现方式，以及如何基于 Unikernel 搭建宏内核，提供用户空间管理和 Linux 应用支持并进行相关的拓展。这部分内容有助于加深对项目 starry-next 和基座 ArceOS 的理解。

5.1.2　环境配置

第七和第八周主要进行环境配置工作。为了验证 starry-next 内存管理模块及接口的设计与实现在 riscv64、x86_64、loongarch64 和 aarch64 这四个架构下的正确性

与性能，我们使用了 qemu 模拟器来模拟这四个架构的运行环境，并且 qemu 模拟器的版本不应低于 8.2.0。qemu 是一个开源的虚拟机模拟器，支持多种硬件架构和操作系统，能够在主机上创建虚拟机并运行不同架构的操作系统。同时，为了支持高版本的 qemu 模拟器，测例的本地运行都在 Ubuntu-24.04 下进行，并且需要在本地安装 rustling 工具链等 rust 相关工具，具体的操作可以参考项目的 README.md 文件。

另外，为了支持自动化测试，也为了探索合理高效宏内核开发模式，本课题不同方向的开发人员一同尝试了多种工具。例如在项目初期使用 Gitlab 管理代码，并使用“全国大学生计算机系统能力大赛”平台进行测试，但由于平台的测试运行速度较慢且有时间限制，因此转而使用 Github 进行代码管理，利用 GitHub Actions 进行自动化测试，定义了多个工作流配置文件，涵盖了不同架构和测试场景的自动化构建与验证。

5.1.3 基础实现

第九周到第十二周基础实现阶段的工作内容是完善宏内核基础功能，使其能够运行简单的 c 语言程序，具体到内存管理方面，即实现 mmap、munmap、brk 系统调用，完善内存管理组件的功能，并通过 basic 阶段的测例。这一阶段中，测例反映出的问题主要分为三个层面：

- 测例层面：测例本身存在问题，例如在 basic 阶段的 times 测例中，用于接受返回值的变量类型为 int 而非 man 手册中描述的 clock_t，导致溢出，测例无法通过。
- starry-next 层面：宏内核的相关实现不完善，例如初始 mmap 系统调用接口的设计未完全参照 Linux 标准，构建新映射时没有取消旧映射。
- ArceOS 层面：ArceOS 相关组件或接口存在欠缺，例如在读取内存区域时范围错误等。

基础实现阶段的难点一是需要在开发中熟悉 Rust 语言和已有框架，二是要准确定位问题所在，因此需要掌握灵活的调试方法，例如根据 starry-next 各个级别的日志信息进行调试，或者是使用 gdb 等工具。另外需要注意的是，内存管理模块存在的问题可能会影响到其他模块的正常运行，例如在 clone 系统调用的实现中存在读取内存空间操作，当启用 Lazy 映射时，会读取到未映射的内存区域，导致测例无法通过。因此，在开发中需要认识到测例集合的局限性，并和其他模块的实现进行充分的沟通。

```
[  1.740581 0:5 starry:task:96] Enter user space:entry=0x404d7f,ustack=0x1,kstack=VA:0xffff8000008ee010
[  1.744583 0:5 starry:mm:172] Page fault at VA:0x0,access_flags:READ USER
[  1.766803 0:5 starry:mm:186] Task(5,"Task(4,\"userboot\")"):segmentation fault at VA:0x0,exit!
[  1.782333 0:4 starry:task:333] wait pid _5_ with code _-1_
```

图 5.1　Lazy 映射下的缺页异常

5.1.4　进阶开发

十三到十六周为进阶开发阶段，工作内容是完善内存管理组件的功能，使其能够支持更多的系统调用，如 mprotect、arch_pctl、sysinfo、prlimit64 等，并通过 libc-test 阶段的测例，同时，对于测例未覆盖的系统调用，也需要编写相应的测例。

在通过 libc-test 测例的过程中需要注意，不同架构下运行测例所需的系统调用可能不同，例如在 x86_64 架构下才会使用 fork 系统调用，其他三种架构下 fork 系统调用会被自动转换为 clone 系统调用。同时，不同架构下对系统调用的实现细节也有不同的要求，例如在 clone 系统调用中，四个架构的参数传递顺序不完全相同，其中 x86_64 和 loongarch64 中第五个参数为子进程 id，第六个为 TLS，但在其他架构中二者的顺序相反。因此，在进行调试或者编写测例时，需要根据具体的架构和系统调用要求进行调整。

```
#[cfg(any(target_arch = "x86_64", target_arch = "loongarch64"))
   ] child_tid: usize,
tls: usize,
#[cfg(not(any(target_arch = "x86_64", target_arch = "
   loongarch64")))] child_tid: usize,
```

清单 5.1　不同架构下 clone 的部分参数传递顺序

本阶段测例涉及的系统调用较多，对各系统调用的实现完成度也有所不同。在开发的过程中，会存在部分其他子系统的系统调用还未实现导致测例无法通过的情况，但这些系统调用的具体功能并不需要实现，此时可以尝试通过令该系统调用直接返回成功的方式来绕过该问题。但是对于 clone、mmap 等重要的系统调用，需要结合 man 手册和 DragonOS 等开源项目进行详细的实现。

在这一阶段中，比较常用的调试方法是观察 starry-next 各个级别的日志信息，并和配合 qemu 模拟器和 strace 指令得到的系统调用信息进行对比，以定位问题所在。

```
[  1.719986 0:4 starry:syscall_imp:57] Syscall rt_sigprocmask
[  1.721138 0:4 starry:syscall_imp:signal:13] sys_rt_sigprocmask:not implemented
[  1.722763 0:4 starry:syscall_imp:177] Syscall rt_sigprocmask return 0
```

图 5.2　返回 0 绕过部分系统调用

5.2 内存管理组件的测试

5.2.1 测试用例分析

测试用例分为两部分：开源测试用例和自定义测试用例。其中开源测试用例主要来自 libc-test 库。libc-test 是一个针对 C 标准库（libc）功能及兼容性的检测工具。C 标准库作为 C 语言开发的基石，涵盖输入输出、内存管理、字符串操作等基础功能。libc-test 借助一系列精心设计的测试用例，全面且细致地检验 C 标准库各功能模块，保障其在不同环境下的正确性和稳定性。在具体测试中，将编译完成的 libc-test 可执行文件作为输入，运行其中的静态与动态测试用例。通过执行这些测试用例并检查输出结果，可判断待测内核是否能正确支持 C 标准库的各项功能，进而评估内核与 C 标准库的兼容性与稳定性。

尽管 libc-test 库提供了丰富的测试用例，可以涵盖大多数系统调用，例如 mmap、munmap、mprotect 等，但由于 libc-test 针对的是 C 标准库，而不是操作系统内核和系统调用的实现，同时一部分系统调用无法使用 libc-test 进行测试，例如 sysinfo、prlimit64 等系统调用。因此，我们还编写了一些自定义测试用例来验证 starry-next 内存管理模块的功能以及系统调用接口的正确性。这些用例将通过 C 标准库中 sys 子目录直接使用相关的系统调用，并验证其返回值，例如：

```
#include <sys/syscall.h>
syscall(SYS_write, fd, buf, len);
```

清单 5.2　直接使用系统调用进行测试的示例

在测例的编写和运行过程中，strace 指令可以帮助我们跟踪系统调用的执行情况，在 Ubuntu-24.04 下通过 strace 指令可以得到 x86_64 架构下的系统调用信息，使用 qemu 模拟器配合 strace 指令也可以得到其他架构下的系统调用信息。这些不仅可以帮助验证测例的正确性，还可以作为对照组可以帮助我们分析 starry-next 内存管理模块的实现是否符合 Linux 的标准实现，以及系统调用的实现的正确性，从而进行进一步的调试和优化。

5.2.2 测试用例实现

【1】brk、mmap 和 munmap 系统调用：

在使用 malloc 等函数申请内存空间时，通常会先使用 brk 系统调用扩充堆空间，然后再使用 mmap 系统调用进行内存映射，释放空间时则会使用 munmap 系统调用。此处以 libc-test 中的 malloc_0 为例，该测例使用 malloc 函数申请了三个大小为 0 的空间，分配相应指针，并检测三个指针的地址是否相同，若不同则测例通过，然后使用 free 进行释放指针指向的内存空间，该测例可以检测 mmap 的匿

名映射和 munmap 的释放是否正确。

另一方面，我们通过自定义测试用例来验证 mmap 系统调用文件映射的正确性，例如：

```
fd = open("test_mmap.txt", O_RDWR | O_CREATE);
array = mmap(NULL, kst.st_size, PROT_WRITE | PROT_READ,
   MAP_FILE | MAP_SHARED, fd, 0);
printf("mmap content: %s\n", array);
```

清单 5.3　自定义 mmap 测例中调用 mmap 进行文件映射

该测试用例打开了一个内容为“Hello, mmap successfully!”的文件，并使用 mmap 系统调用将该文件映射到内存中，然后读取文件内容，如果能正确输出文件内容，则说明 mmap 系统调用的文件映射功能正常。

【2】mprotect 系统调用：

mprotect 系统调用用于修改指定内存区域的访问权限，在 libc-test 中常用于多线程及线程间通信的相关测例。例如 pthread_cond 测例测试了线程间通信的条件变量功能，需要使用 mprotect 系统调用将内存区域的访问权限修改为可读可写，以保证线程间通信的正确性。

```
TEST(r, pthread_mutex_init(&mtx, 0), "");
TEST(r, pthread_cond_init(&cond, 0), "");
TEST(r, pthread_mutex_lock(&mtx), "");
TEST(r, pthread_create(&td, 0, start_signal, (void *[]){ &cond,
    &mtx }), "");
TEST(r, pthread_cond_wait(&cond, &mtx), "");
TEST(r, pthread_join(td, &res), "");
TEST(r, pthread_mutex_unlock(&mtx), "");
TEST(r, pthread_mutex_destroy(&mtx), "");
TEST(r, pthread_cond_destroy(&cond), "");
```

清单 5.4　pthread_cond 测例通过条件变量实现线程间通信

【3】sysinfo 和 prlimit64 系统调用：

sysinfo 和 prlimit64 系统调用的自定义测试用例主要是通过获取系统信息和进程限制来验证其实现是否正确，编写测例时需要注意二者的参数中都有特定的类型，例如 sysinfo 系统调用的参数类型为 sysinfo 结构体，prlimit64 系统调用的参数类型为 rlimit 结构体。

```
#include <sys/syscall.h>
syscall(SYS_prlimit64, getpid(), RLIMIT_STACK, NULL, &old_limit
   );
```

清单 5.5　prlimit64 测例部分代码

```
#include <sys/sysinfo.h>
struct sysinfo info;
sysinfo(&info);
```

清单 5.6　sysinfo 测例部分代码

而 prlimit64 的最大文件数量限制则可以通过 libc-test 的 rlimit_open_files 测试用例来验证，该测例会在设置资源限制后重复创建新的文件描述符（打开文件）至失败，并取当前最大文件描述符和资源限制值进行比较，相等则说明 prlimit64 系统调用能正常限制文件数量。

```
#include <sys/resource.h>
struct rlimit rl;
if (setrlimit(RLIMIT_NOFILE, &rl))
        t_error("setrlimit(%d, %ld) failed: %s\n", r, lim,
           strerror(errno));
if (getrlimit(RLIMIT_NOFILE, &rl))
        t_error("getrlimit(%d) failed: %s\n", r, strerror(errno
           ));
if (rl.rlim_max != lim || rl.rlim_cur != lim)
        t_error("getrlimit %d says cur=%ld,max=%ld after
           setting the limit to %ld\n", r, rl.rlim_cur, rl.
           rlim_max, lim);
while((fd=dup(1)) != -1)
        if (fd > maxfd) maxfd = fd;
if (errno != EMFILE)
        t_error("dup(1) failed with %s, wanted EMFILE\n",
           strerror(errno));
if (maxfd+1 != lim)
        t_error("more fds are open than rlimit allows: fd=%d,
           limit=%d\n", maxfd, lim);
```

清单 5.7　rlimit_open_files 测例设置资源限制并进行检验

【4】性能测试

当前并未对 starry-next 内存管理模块的性能进行较完备的测试，但重复使用系统调用和内存读写的方式，也可以在一定程度上评估 starry-next 内存管理模块的性能。因此设计测例，分别进行 1000 次 mmap、mummap、mprotect 和读写操作，观察各架构单核和四核下的运行时间，并与运行在 qemu 模拟器下的 Linux 标准实现进行对比。

```
for (int i = 0; i < NUM_ALLOCS; i++) {
    ptrs[i] = mmap(NULL, ALLOC_SIZE, PROT_READ | PROT_WRITE,
       MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
}
```

清单 5.8　性能测试测例的部分代码

5.2.3 测试结果与分析

【1】brk、mmap 和 munmap 系统调用：

清单5.9中展示了 malloc_0 测试的结果，可以看到程序先调用 brk 系统调用拓展了堆空间，调用了一次 mmap 系统调用为拓展的空间建立映射，length 为十六进制的 0x1000，说明分配了一个页（4096 字节）的内存空间，这是分配的最小单位。但因为 malloc 申请空间的大小为 0，所以在第二和第三次 malloc 时空间足够，没有触发新的 mmap 系统调用，之后 free 释放时也只调用了一次 munmap 系统调用。最终测例通过，匿名映射和映射解除功能正常。

```
========== START entry-static.exe malloc_0 ==========
[  7.927777 1:11 starry::syscall:13] Syscall brk
[  7.930924 1:11 starry::syscall:211] Syscall brk return
   1073741824
...
[  7.946120 1:11 starry::syscall:13] Syscall mmap
[  7.950708 1:11 starry_api::imp::mm::mmap:85] sys_mmap: addr:
   40000000, length: 1000, prot: MmapProt(0x0), flags:
   MmapFlags(PRIVATE | FIXED | ANONYMOUS), fd: -1, offset: 0
[  7.972843 1:11 starry::syscall:211] Syscall mmap return
   1073741824
...
[  8.022895 1:11 starry::syscall:13] Syscall munmap
[  8.028983 1:11 starry::syscall:211] Syscall munmap return 0
...
Pass!
========== END entry-static.exe malloc_0 ==========
```

清单 5.9　运行 malloc_0 测例的部分输出（info 级信息）

自定义 mmap 测试用例的结果如清单5.10所示，可以看到程序成功读取到了文件中的内容，说明文件映射功能正常。

```
========== START test_mmap ==========
file len: 27
mmap content:   Hello, mmap successfully!
========== END test_mmap ==========
```

清单 5.10　自定义 mmap 测例运行结果

【2】mprotect 系统调用：

清单5.11中展示了 pthread_cond 测例的结果，可以看到程序先调用 mmap 系统调用分配了一段内存空间，由于 Lazy 映射的特性，这段内存空间并没有被映射到物理内存中，并且保护属性为空。调用 mprotect 系统调用后，这段空间被分配了对应的物理地址，并且保护属性为可读可写，说明 mprotect 系统调用的功能正常。

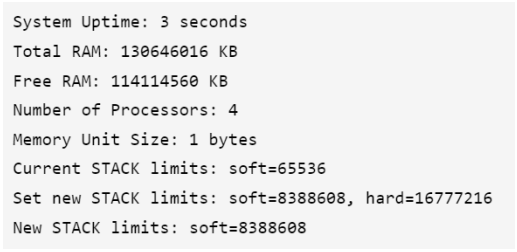
```
[  6.315702 3:11 starry::syscall:13] Syscall mmap
```

```
[  6.319449 3:11 starry_api::imp::mm::mmap:86] sys_mmap: addr:
   0, length: 23000, prot: MmapProt(0x0), flags: MmapFlags(
   PRIVATE | ANONYMOUS), fd: -1, offset: 0
[  6.326679 3:11 starry::syscall:211] Syscall mmap return
   667648
[  6.330263 3:11 starry::syscall:13] Syscall mprotect
[  6.333987 3:11 starry_api::imp::mm::mmap:169] sys_mprotect:
   addr: a5000, length: 21000, prot: MmapProt(READ | WRITE)
[  6.340052 3:11 starry_api::imp::mm::mmap:178] page not mapped
[  6.352141 3:11 starry_api::imp::mm::mmap:185] after
   sys_mprotect: paddr: PA:0x8140d000, flags: READ | WRITE |
   USER
[  6.357734 3:11 starry::syscall:211] Syscall mprotect return 0
```

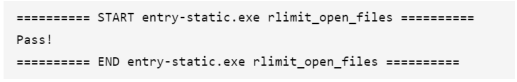
清单 5.11　运行 pthread_cond 测例的部分输出（info 级信息）

【3】sysinfo 和 prlimit64 系统调用：

从图5.3(a)中可以看到，sysinfo 系统调用返回了系统信息，例如当前系统共有 4 各处理器，系统运行时间为 3 秒，总内存为 130646016 KB，剩余内存为 114114560 KB。prlimit64 系统调用则返回了进程用户栈的限制信息，并且在设置用户栈大小时，prlimit64 系统调用正确使用了新的软限制值，这说明 sysinfo 和 prlimit64 系统调用设置用户栈的功能正常。同时，如图5.3(b)所示，rlimit_open_files 测例也是通过的，这说明 prlimit64 系统调用的文件数量限制功能正常。



(a) sysinfo 和 prlimit64 测试结果



(b) rlimit_open_files 测例结果

图 5.3　sysinfo 和 prlimit64 测试结果

【4】性能测试：

各架构下，单核和四核的运行时间如表5.1和表5.2所示。可以观察到，各架构的运行时间相差较大，这可能是由不同架构使用的页表结构、虚实地址结构、寻址方式等内存管理机制不同导致的，同时，不同架构使用的交叉编译工具链对于程序编译、链接、优化方式的影响也可能导致运行时间的差异。另外，四核下运行时间大于或等于单核的运行时间，可能是测例代码未充分使用多核的特性，加上多核使得内核空间管理开销增加所致。

另一方面，如图5.4所示，以 x86_64 架构下的结构为例，starry-next 在内存读

表 5.1　各架构单核运行 1000 次操作的运行时间（单位：ms）

架构	x86_64	LoongArch64	RISC-V 64	AArch64
mmap	59.49	96.01	166.32	358.3
munmap	244.17	1107.39	586.18	735.41
mprotect	54.06	552.82	498.47	387.09
写	82.17	576.72	487.73	314.1
读	13.93	19.55	18.38	17.59

表 5.2　各架构四核运行 1000 次操作的运行时间（单位：ms）

架构	x86_64	LoongArch64	RISC-V 64	AArch64
mmap	57.05	106.56	106.56	387.46
munmap	259.87	1110.76	1110.76	804.91
mprotect	54.06	552.82	498.47	387.09
写	81.73	592.46	592.46	443.21
读	11.15	20.16	20.16	18.7

写方面的性能和 Linux 相近，但是系统调用方面的性能低于 Linux，这可能是由于二者在页表结构、寻址方式等方面相似，但是 Linux 提供了更多页面映射和管理的优化，例如提供了更灵活的页面置换算法等。

```
Testing memory allocation performance...
Memory mmap time: 25.52 ms
Testing memory free performance...
Memory free time: 47.71 ms
Testing memory access performance...
Memory write time: 86.24 ms
Memory read time: 13.15 ms
Testing memory protect performance...
Memory protect time: 11.83 ms
```

(a) Linux

```
Testing memory allocation performance...
Memory mmap time: 60.90 ms
Testing memory free performance...
Memory free time: 259.63 ms
Testing memory access performance...
Memory write time: 84.50 ms
Memory read time: 11.78 ms
Testing memory protect performance...
Memory protect time: 48.65 ms
```

(b) starry-next

图 5.4　重复 1000 次各操作运行时间对比

第 6 章　结论与展望

6.1　结论

到目前为止，我们已经基本完成了 starry-next 内存管理模块及其接口的设计与实现，并且对其进行了测试。我们的实现与 Linux 内核的内存管理模块接口基本一致，并且在功能测试中表现良好，性能方面则需要进一步测试和优化。具体的代码已开源至 GitHub。

在开发过程中，由于操作系统的各模块间存在一定的依赖关系，我们使用了多种协同开发 OS 的方法，包括使用 GitHub 的 issue 和 pull request 进行协作，根据模块和系统调用分配任务，以通过某个测例为目的进行调试等。我们不仅参与了宏内核接口的设计、实现与测试，还一定程度上参与了基座微内核 ArceOS 的修改。另外，我们还将开发过程中的问题和解决方法记录在了实验报告中，以便后续的开发工作中可以参考。希望这些工作能为后续组件化操作系统的开发提供一些参考和帮助。

6.2　展望

在未来的工作中，我们计划进一步完善 starry-next 内存管理模块及其接口，包括完善当前仅实现部分功能的系统调用接口、优化内存管理模块的性能等方面。另外，当前 starry-next 内存管理模块还未支持各类页面置换算法，我们计划在未来的工作中增加对不同页面置换算法的支持，以提高内存管理模块的性能和效率。同时，增加对内存管理模块的安全性和稳定性的支持也会纳入未来的计划，以确保内存管理模块在实际应用中能够稳定运行。

参考文献

[1] 陈灵. 微内核架构操作系统内存管理系统的研究与实现[D]. 电子科技大学, 2024.

[2] 焦旭峰. 基于 RISC-V 的 IOMMU 设计[D]. 湖南大学, 2023.

[3] Guo J, Chen X, Tang Y, et al. Slab: Efficient transformers with simplified linear attention and progressive re-parameterized batch normalization[A/OL]. 2024. arXiv: 2405.11582. https://arxiv.org/abs/2405.11582.

[4] 崔昕宇. 基于 μCOS-III 嵌入式实时系统内存管理的设计与实现[D]. 沈阳工业大学, 2020.

[5] Masmano M, Ripoll I, Crespo A, et al. Tlsf: a new dynamic memory allocator for real-time systems[C/OL]//Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004. 2004: 79-88. DOI: 10.1109/EMRTS.2004.1311009.

附录 A　外文资料的书面翻译

Ariel OS：适用于网络传感器和多核微控制器的嵌入式 Rust 操作系统

摘要

大量原本用 C/C++ 实现的低级系统软件组件，目前正在被用 Rust 语言重新编写。Rust 是一种相对更安全、更可靠的编程语言。然而，到目前为止，还没有用 Rust 编写的嵌入式操作系统支持微控制器上的多核抢占式调度。因此，本文填补了这一空白，提出了一个新的操作系统：Ariel OS。我们描述了它的设计，提供了其实现的源代码，并在主流的 32 位微控制器架构上进行了微基准测试，包括 ARM Cortex-M、RISC-V 和 Espressif Xtensa。我们展示了我们的调度器如何在利用多核优势的同时，仅在单核硬件上产生极小的额外开销。正因如此，Ariel OS 为研究和行业从业者针对小型联网设备提供了一个便捷的嵌入式软件平台。

关键词：嵌入式软件、Rust、微控制器、多核、操作系统、实时操作系统（RTOS）、物联网（IoT）

A.1　引言

我们对网络物理系统（cyberphysical systems）和分布式计算系统的依赖程度日益加深。这些系统所涵盖的硬件不仅包括微处理器领域的机器，还涵盖了更多资源受限的设备，例如基于微控制器单元（MCU）的传感器。根据 RFC7228[1] 的描述，这些设备实现了超低功耗和超低价格，但与微处理器相比，它们的内存容量要小得多（仅在千字节范围内），处理能力也弱得多（CPU 时钟速度在兆赫兹范围内）。

众多的多核微控制器——为了在设备保持可用以进行传感、驱动或通过网络进行数据推送/拉取的同时，执行计算密集型任务（例如实时音频处理或边缘机器学习，），必须高效利用多核。这些任务需要在设备板上执行。因此，许多厂商的旗舰产品如今都基于多核 32 位架构。例如，Espressif ESP32-S3 微控制器基于双核 Xtensa LX7；RP2350 微控制器基于双核 RISC-V Hazard3 和双核 ARM Cortex-M33，被用于流行的 Raspberry Pi Pico 2 开发板。早期型号 RP2040 基于双核 ARM Cortex-M0+，已经售出数百万个单位。其他厂商（如 Nordic、NXP、ST 等）也推出了 32 位多核微控制器。

微控制器的嵌入式操作系统——随着在微控制器（MCU）和传感器上运行的软件日益复杂，操作系统（OS）的使用变得愈发普遍。目前，最知名的操作系统大多采用 C 语言编写，例如 RIOT、Zephyr 和 FreeRTOS[2]。近年来，随着 Rust 语言的兴起，一种新型的操作系统和嵌入式软件平台应运而生，它们以 Rust 语言为核心进行开发。

这一领域的先驱工作由 Tock OS[3] 和众多 裸机嵌入式 Rust 编程项目引领，这些工作极大地提升了嵌入式 Rust 的性能和可靠性[4]。适用于微控制器的嵌入式 Rust 平台示例包括 Hubris[5]、Drone OS[6]、RTIC[7]，以及支持异步编程的 Embassy[8]。然而，迄今为止，这些平台中尚无一个支持在主流 32 位微控制器上进行多核调度。

为了填补这一空白：

- 我们设计了 Ariel OS，这是一个创新的嵌入式 Rust 操作系统，它结合了以下特性：
  (i) 针对单核与多核微控制器优化的调度器；
  (ii) 异步 Rust；
- 我们实现了 Ariel OS，并在多种主流 32 位微控制器架构上进行了性能测试，包括 ARM Cortex-M、Espressif Xtensa 和 RISC-V；
- 们对操作系统进行了全面概述，它不仅包含调度器，还集成了多种库和跨硬件的 API；
- 我们将 Ariel OS 的完整代码开源发布。

A.2 微控制器上的多核调度的相关研究

在对现有软件和相关文献进行调研后，我们发现微控制器上的多核调度的显著示例包括 ThreadX[9] 和 FreeRTOS[10]，以及 Zephyr[11] 和 NuttX[12]。通常，处于就绪状态且等待执行的线程会被列入一个称为 运行队列（runqueue）的有序数据结构中。我们观察到在多核架构上，运行队列主要有两种方法。第一种方法是 全局调度，它使用一个单一的全局运行队列，线程从该队列中被分配到可用的核上。相比之下，分区调度方法为每个处理器使用单独的运行队列，线程被静态分配到这些队列中。此外，我们还观察到，支持多核调度的微控制器操作系统主要使用全局调度。我们还发现，在调度器内部进行同步时，通常会使用全局临界区。

我们还发现，这一领域的操作系统在将线程分配给核心时采用了不同的方法。一方面，ThreadX 采用了一种 核心重分配方法，即通过一个重分配例程将优先级最高的 n 个线程映射到 N 个核心上。调度器中断处理程序在每次被调用时，会读

取为每个核心分配的线程信息。在运行队列发生每次变化后，ThreadX 中的重分配例程——*rebalance*——会被执行。

另一方面，FreeRTOS、Zephyr 和 NuttX 采用了*动态线程选择方法*的变体。在这种方法中，调度器中断处理程序在被调用时直接从运行队列中选择某个核心的下一个线程。随后，该线程将从运行队列中移除，以防止其被多个核心选中。相反，当一个正在运行的线程被抢占时，它将重新加入运行队列。

A.3 Ariel OS 及其调度器的目标

用例与假设——我们的研究聚焦于典型的微控制器，这些设备通常具备 N 个核心和 n 个线程，并且仅配备共享缓存（如果有的话）。此外，N 和 n 的数值相对较小：通常 $N < 3$ 且 $n < 15$。值得注意的是，这与 Linux、L4 或高性能计算（HPC）等场景存在显著差异，后者的 N 和 n 值通常更大，并且涉及更复杂的缓存和公平性机制。

在本文中，我们的目标是支持多核，特别是对称多处理（SMP）场景，这在微控制器上是最常见的。在这种情况下，多个核心在架构上是相同的，并且共享整个地址空间我们致力于实现以下目标：

- **持续工作**——调度器必须在 N 个可用核心上执行优先级最高的 n 个线程；
- **实时性**——保留基于用户设置的线程优先级的调度器的实时特性；
- **可移植性**——确保在不同架构和平台之间的可移植性；
- **透明性**——从单核到多核的切换必须是透明的，同时保持调度器的性能，并且不会给用户带来额外的复杂性。

操作系统——除了调度功能外，Ariel OS 还致力于成为异构 32 位微控制器上分布式计算和网络应用的一站式解决方案。更多关于这一愿景的细节将在第 A.8 节中介绍。

A.4 单核调度器的基础与优化

Ariel OS 以类似于 RIOT[13] 的单核调度器为基础构建。具体而言，它采用无时钟的实时调度机制，结合抢占式优先级调度策略。线程的最大数量在编译时确定，所有调度器数据结构均采用静态分配。上下文切换采用惰性执行方式，即仅当运行队列的头部自上次调度器调用以来发生变化时，才会触发上下文切换。此外，调度器还进行了优化，以避免不必要的中断。具体而言，Ariel OS 在设置调度器中断之前，会检查是否有优先级更高的线程就绪。

A.5 Ariel OS 多核调度器设计

在下文中，不失一般性，我们假设 $N = 2$ 个核心（核心 0 和核心 1）。然而，需要注意的是，扩展到 $N > 2$ 是微不足道的，前提是 N 和 n 保持较小，符合我们的目标（回顾 section A.3）。

A.5.1 启动

Ariel OS 操作系统运行时初始化在核心 0 上执行，而另一个核心则处于深度休眠或停滞模式。在核心 0 上完成通用初始化后，核心 1 将设置使用共享中断向量表（IVT）、一个独立的主栈指针（SP），以及一个入口函数，该函数最终将在该核心上启动线程，类似于核心 0。这一过程如图 A.1 所示。线程启动是通过启用最低优先级的调度中断并调用调度器来实现的，调度器随后会选择下一个线程进入核心。

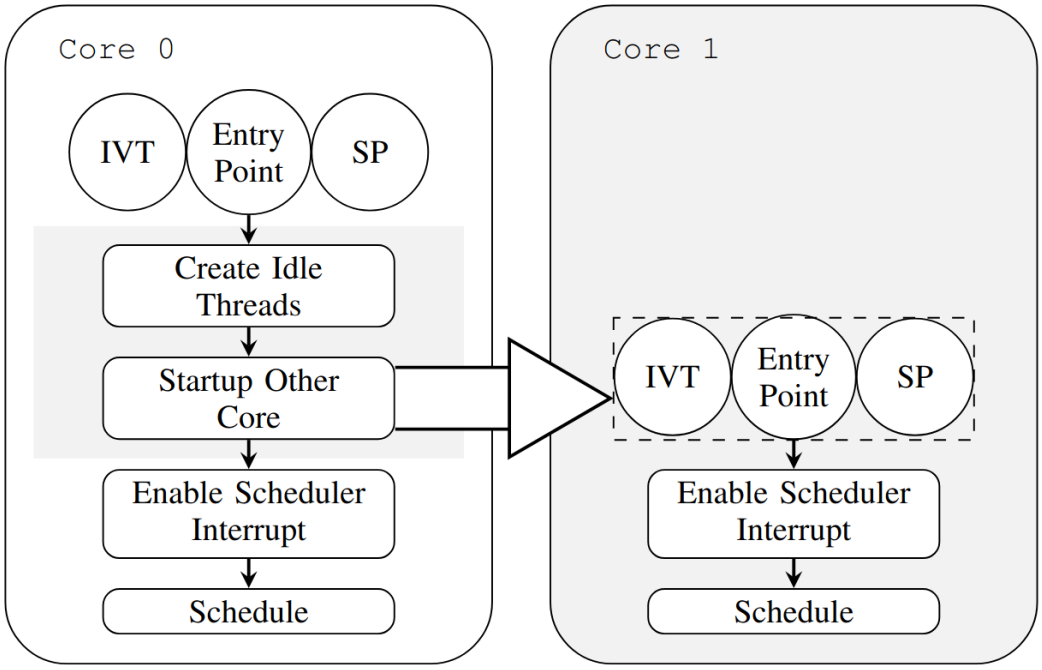


图 A.1　双核系统上的线程启动

A.5.2 全局调度方案

Ariel OS 使用全局调度方案

全局调度支持线程迁移，允许线程在执行过程中在不同核心之间动态移动。我们选择全局调度，是因为它能够有效减少上下文切换的频率，降低优先级反转的风险，并且更高效地利用整体计算资源[14-15]。此外，全局调度还避免了分区调度中常见的线程分配难题，即如何为每个核心找到最优的线程分配方案。全局调度

的主要缺点是其在可扩展性方面的局限性，尤其是在全局运行队列可能成为上下文切换瓶颈的情况下[16]。然而，在微控制器场景中，由于处理器核心数量通常较少，这一问题的影响几乎可以忽略不计。

A.5.3 线程分配到核

在单核配置中，线程选择过程仅需读取运行队列的头部。然而，在多核配置下，这种简单的选择方式会导致同一个线程在多个核上被执行。为了避免这种情况，我们探索了多种线程分配策略。我们最初设计的调度器能够灵活支持 *核重分配*或 *动态线程选择*（参见 section A.2）。此后，我们不仅实现了这两种方法，还在 section A.7 中详细报告了它们在不同硬件平台上的性能表现。

A.5.4 核亲和性掩码

核亲和性是一种特性，允许将线程固定到特定的核上运行。这为用户提供了对线程运行位置和方式的精细控制。

核亲和性通过位掩码实现，其中设置的位表示线程可以在该索引对应的核上运行。调度器在选择线程应运行的核时会使用这些信息

A.5.5 动态优先级与优先级继承

在 Ariel OS 中，所有活动线程的优先级都可以在运行时动态调整。如果运行中的线程优先级被降低，或者等待中的线程优先级被提高，这可能会触发上下文切换。此外，动态优先级还支持互斥锁的优先级继承。当一个高优先级线程因等待一个当前由低优先级线程持有的互斥锁而阻塞时，持有互斥锁的线程将继承等待线程的较高优先级。这可以防止持有者被其他低优先级线程抢占，从而有助于避免由间接阻塞引起的优先级反转问题。

A.5.6 调度器中的同步

Ariel OS 通过全局临界区在内核中实现互斥，这实际上形成了一种 *大锁设计*。因此，在内核中不存在并发问题，鉴于 Ariel OS 中此类操作的持续时间较短，这种设计是可以接受的。它确保了操作的正确性，防止了数据竞争和死锁，并且简化了涉及多个数据结构的操作，例如运行队列和线程控制块（TCBs），因为无需单独锁定。在单核系统中，屏蔽所有中断就足以创建一个临界区。在多核系统中，需要额外的全局自旋锁来确保其他核心上的线程也无法进入另一个临界区。

A.5.7 调度线程与异步 Rust 任务结合

Ariel OS 使用基于 *Embassy*[8] 的异步代码进行系统初始化和硬件抽象层（HAL）的实现。系统中始终存在一个用于执行异步任务的系统执行器. 使用抢占式调度器以及由此产生的线程是可选的。如果未使用调度器，系统执行器将在中断上下文中运行。如果使用了调度器，系统执行器则会在一个线程中执行。额外的执行器可以在其他线程中启动。当某个执行器上的所有任务都处于挂起状态时，拥有该执行器的线程将被挂起。线程可以阻塞在异步函数上，并等待来自执行器的异步资源。因此，Ariel OS 在调度器、异步 Rust、基于 future 的并发以及异步 I/O 之间架起了桥梁。

A.6 Ariel OS 调度器实现

截至本文撰写之时，Ariel OS 已在三种主流的硬件平台/架构上实现了多核调度功能：RP2040 和 RP2350（分别为双核 ARM Cortex-M0+/ 双核 ARM Cortex-M33）以及 ESP32-S3（双核 Xtensa LX7）。目前，Ariel OS 对 RISC-V 的多核调度支持仍在开发中。

RP2040、RP2350 和 ESP32-S3 在 Rust 生态中得到了良好的支持：前者通过 Embedded-Rust 工作组和 `rp-rs` 项目获得支持，后者则通过 Espressif 的 `esp-hal` 项目直接获得支持。Ariel OS 在其实现中充分利用了这些支持。

A.6.1 硬件抽象

我们工作的主要目标之一是实现清晰的硬件抽象以及减少特定平台的代码。当生态中已经存在对某个芯片的支持时，增加对该芯片的支持应当是一项轻而易举的任务。

在 Ariel OS 中，硬件抽象发生在两个层面：

CPU 架构抽象——在这一层，实现了针对特定 CPU 架构（例如 Cortex-M）的调度器逻辑。它涵盖了设置线程栈、配置触发调度器的异常以及实际的上下文切换逻辑等所有与架构相关的代码。

芯片级抽象——在这一层，实现了特定平台的对称多处理（SMP）逻辑。

对于多核调度，需要两种机制：

(i) 用于启动其他核心（或多个核心）；

(ii) 用于在特定核心上调用调度器。

Ariel OS 利用了 Rust 的类型系统，通过将上述抽象定义为 *特性*（traits）[17] 来实现。在调度器中为另一个平台添加支持，仅需实现两个特性——每个层面各一

个。因此，在 Ariel OS 中，RP2040 的芯片级对称多处理（SMP）实现仅有 70 行代码，而 ESP32-S3 的实现则为 66 行代码。

A.6.2 特定平台的调度逻辑

在 RP2040/RP2350 上—— RP2040/RP2350 芯片在硬件层面实现了两个 FIFO 队列，用于连接两个物理核心。这些队列在 Ariel OS 的启动过程和调度器调用中发挥关键作用。在启动阶段，核心 1 保持睡眠模式，直到通过 FIFO 队列接收到向量表、栈指针和入口函数，这一过程遵循一个固定的协议。启动完成后，FIFO 队列用于在另一个核心上调用调度器。接收到的消息将触发一个中断，中断处理程序会设置本地调度器异常，从而启动调度过程。

在 ESP32-S3 上——该芯片未实现任何处理器间通信通道。相反，它通过触发两个不同的软件中断来分别在每个核心上调用调度器。在 ESP32-S3 上启动第二个核的启动是通过将入口函数的地址写入核心 1 的启动地址，然后重置并解除该核心的停机状态来实现的。入口函数会设置该核的向量表地址和栈指针，随后运行我们的线程启动逻辑。

A.6.3 中断处理

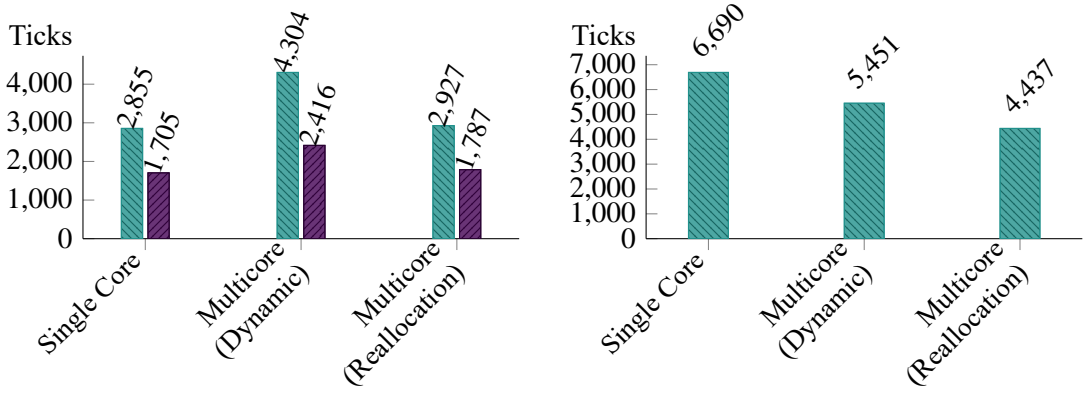
在 RP2040/RP2350 上，每个核心都配备了各自的 ARM 嵌套向量中断控制器（NVIC）。在 ESP32-S3 上，每个核心都有其可配置的中断矩阵。在两种芯片上，都支持嵌套中断，使得低优先级的中断可以被高优先级的中断抢占。外部中断被路由到两个核心，但仅在一个核心上启用以进行处理，具体如下所述。

在启动期间，外设初始化要么在核心 0 上以中断模式进行，且在启动线程之前完成，要么由一个高优先级线程完成。这种初始化会在其执行的核心上配置并启用所需的中断，而该核心随后将负责处理相关的中断。

尽管如此，两个核心共享同一个中断向量表，因此也可以手动在各个核心上屏蔽和解除屏蔽中断，以配置中断的处理位置。

A.6.4 调度器中的同步

调度器通过一个单一结构实现，该结构包含运行队列、线程控制块（TCBs）以及其他调度数据，并实现了所有调度逻辑。该结构被一个包装类型保护，确保所有对调度器的访问都在临界区中执行，并且无法在临界区之外获得对调度器的引用。



(a) RP2040/RP2350 (blue/purple)　　(b) ESP32-S3 (dual Xtensa LX7)

图 A.2　两种多核调度器设计的上下文切换性能与单核配置的对比

A.7　使用 Ariel OS 多核的微基准测试

接下来，我们在主流的硬件平台上评估了 Ariel OS 的性能。我们的基准测试代码已发布在[18]。

我们使用的**双核 MCU** 包括：

(i) Espressif ESP32-S3，配备双核 Xtensa LX7，主频 240 MHz；

(ii) RP2040/RP2350，配备双核 Cortex-M0+/M33，主频分别为 133 MHz 和 150 MHz。

我们用于对比的**单核开发板**包括：

(i) Nordic nRF52840，配备 Cortex-M4，主频 64 MHz；

(ii) Espressif ESP32-C3，配备 RISC-V RV32IMC，主频 160 MHz。

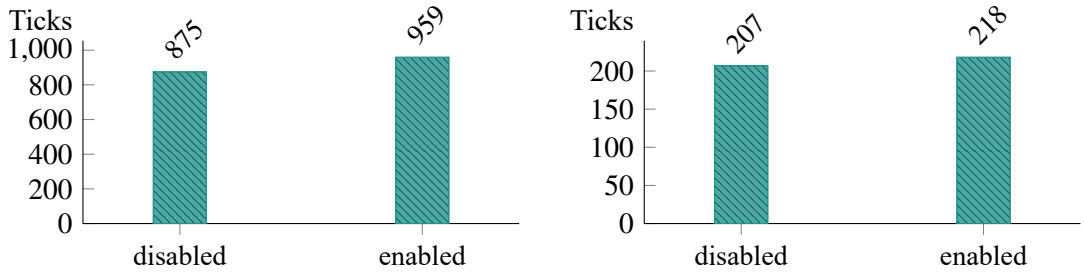
在 Xtensa 和 Cortex-M 架构上，性能是通过时钟周期（ticks）来衡量的。在基于 RISC-V 的 ESP32-C3 上，我们使用系统定时器进行测量，该定时器运行频率为 16 MHz。我们所展示的测量结果是 1000 次运行的平均值。

A.7.1　比较多核调度器的差异

Figure A.2 展示了一个微基准测试，其中四条线程被分为两组，交替唤醒对方并挂起自己的执行。我们比较了不同调度器变体的性能：(i) 在 section A.4 中描述的 *单核变体*，(ii) 使用 *核重分配*的多核变体，以及 (iii) 在 subsection A.5.3 中提到的使用 *动态线程选择*的多核变体。

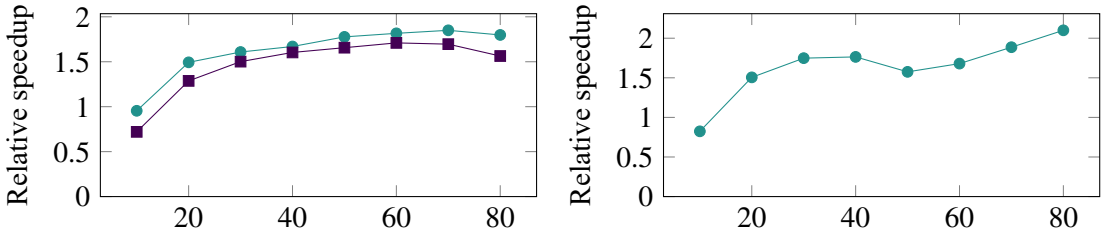
结果表明，在我们测试的所有多核微控制器上，动态分配优于重新分配。相对复杂的重新分配例程在每次线程状态变化后执行，会带来一定的代价。

该基准测试主要由调度器操作构成。因此，RP2040 上的调度器互斥机制导致其主要以顺序方式执行。相比之下，在 ESP32-S3 上，硬件层面的并行化机制有所



(a) nRF52840 (Cortex-M4)　　(b) ESP32-C3 (RISC-V)

图 A.3　在单核硬件上测量的多核调度特性开销



(a) RP2040/RP2350 (blue/purple)　　(b) ESP32-S3 (dual Xtensa LX7)

图 A.4　$N \times N$ 矩阵的乘积, $N \in \{10, 20, ..., 80\}$

不同，较少强制执行顺序操作。此外，ESP32-S3 上完成一次基准测试迭代所需的时钟周期远多于 RP2040。因此，在此基准测试中，ESP32-S3 能够从并行化中获得更多优势。接下来，我们将重点关注多核调度中的动态线程选择结构。

A.7.2　多核调度的开销

接下来，我们测量多核调度功能的开销，并将其与我们在section A.4中开始时的单核调度器进行比较。为此，我们专注于线程抢占，因为与单核相比，多核功能增加了一个额外的步骤：将被抢占的线程重新插入运行队列。Figure A.3报告了在不同单核开发板上进行的微基准测试结果，其中低优先级线程设置了一个标志以触发高优先级线程的运行，从而导致上下文切换。我们观察到，启用多核功能时的开销仍然很小，在 nRF52840 上约为 9.6

A.7.3　计算密集型任务的加速

接下来，我们分别在启用和未启用多核功能的情况下, 进行了 $N \times N$ 矩阵乘法的基准测试。在单核配置中，两次计算是顺序执行的。对于多核配置，任务被分配到两个线程中，并且这两个线程是并行调度的。从图 A.4中我们可以观察到，当 N 较小时，IPC（进程间通信）开销主导了并行化的收益。而当 N 增大时，我们观察到性能提升逐渐趋于 2 倍。我们还注意到性能提升并非严格线性增长。这有

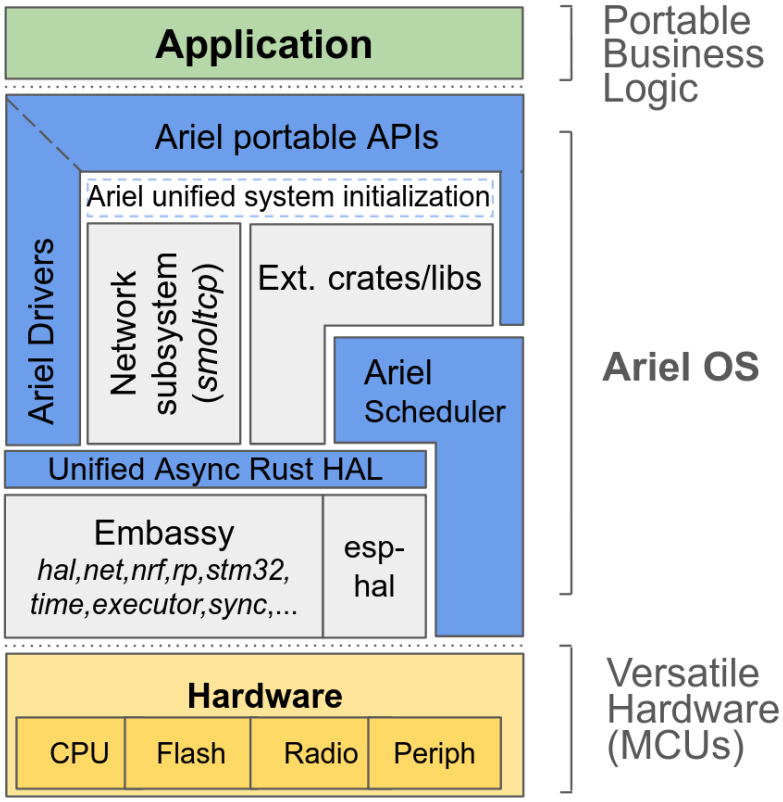


图 A.5　Ariel OS 体系结构图

待进一步研究——它可能是由于 MCU（微控制器）特定的存储子系统特性所导致的。例如，在 RP2040 上，两个核会竞争共享的内存总线[19]。

总结 — 通过采用全局调度方案和优先级机制，我们实现了持续工作和实时性特性。我们的硬件抽象提供了可移植性，并且我们通过实验测量展示了透明性。因此，我们已经实现了我们在第 A.3节中设定的目标。

A.8　Ariel 操作系统概述

Ariel OS 是完全开源的[20]。对于基础硬件抽象和异步 Rust 编程，Ariel OS 基于 Embassy 构建[8]。图A.5展示了 Ariel OS 组件（蓝色部分）是如何利用嵌入式 Rust 生态系统（灰色部分）的。具体来说，Ariel OS 结合了以下元素：

灵活的网络栈配置 — Ariel OS 集成了一个网络栈（*embassy-net/smoltcp*[21]），并结合了我们提供的额外模块，支持多种网络配置。这些配置选项包括 IPv4 和 IPv6、HTTP/TCP 和 CoAP/UDP，支持无线和有线链路层，并通过开放标准（包括 COSE、OSCORE、EDHOC、TLS[22]）以及精选的密码学后端库进行安全保护。

抽象化的系统初始化 — 在微控制器（MCU）上，初始化系统的代码对开发人员来说可能极具挑战性。因此，Ariel OS 对初始化过程进行了抽象化，例如设置：

(i) 网络栈，
(ii) 密码学材料和身份，
(iii) 随机数生成器，
(iv) USB 外设，

等等。配置在构建系统层面进行处理，并提供了方便的默认设置。这样可以最小化样板代码，同时为应用逻辑提供了高级构建模块。

统一的外设 API — Ariel OS 设计了外设初始化和设置（针对 GPIO、I2C、SPI 访问传感器/执行器），使其在不同微控制器（MCU）系列中保持一致。因此，一次编写的应用程序代码可以编译到所有 Ariel OS 支持的设备上。这比嵌入式 Rust 的现有状态有了显著改进（例如 *embedded-hal* 或 *embedded-hal-async* 等 crate），这些 crate 省略了初始化，从而导致了初始化 API 的混乱——以及非常有限的应用程序代码可移植性。

元构建系统 — Ariel OS 工具链充分利用了嵌入式 Rust 的 *crates* 生态系统和 Rust 构建系统 *Cargo*。为了应对极其多样模块化的目标配置的限制，我们将 Cargo 封装在 *laze*[23] 中，这是一个元构建系统，用于处理各种开发板上庞大的软件配置矩阵。

A.9 结论

在本文中，我们介绍了 Ariel OS，这是第一个支持单核和多核抢占式调度以及异步 Rust 的微控制器嵌入式 Rust 操作系统。我们通过实验评估了在所有支持的多核平台上，独特的多核调度器如何可以作为便捷的默认选项。然而，当并行化并非必要时，应用程序可以选择退出多核调度。因此，Ariel OS 丰富了现有的开源工具集，适用于涉及传感器/执行器或使用 32 位微控制器（如 ARM Cortex-M、RISC-V 或 ESP32）的小型网络设备的安全高效分布式计算应用。

参考文献

[1] Bormann C, Ersue M, Keränen A. Request for comments: No. 7228 Terminology for Constrained-Node Networks[M/OL]. RFC Editor, 2014. https://www.rfc-editor.org/info/rfc7228. DOI: 10.17487/RFC7228.

[2] Hahm O, et al. Operating systems for low-end devices in the Internet of Things: a survey[J]. IEEE Internet of Things Journal, 2015.

[3] Levy A, et al. Ownership is theft: Experiences building an embedded OS in Rust[C]//ACM PLOS. 2015.

[4] The Rust Community. The Embedded Rust Book[EB/OL]. 2024. https://docs.rust-embedded.org/book/.

[5] Hubris OS[EB/OL]. 2025. https://github.com/oxidecomputer/hubris.

[6] Drone OS[EB/OL]. 2025. https://github.com/drone-os.

[7] RTIC[EB/OL]. 2025. https://github.com/rtic-rs/rtic.

[8] Embassy Framework[EB/OL]. 2025. https://github.com/embassy-rs/embassy.

[9] Eclipse ThreadX[EB/OL]. 2025. https://github.com/eclipse-threadx.

[10] FreeRTOS[EB/OL]. 2025. https://www.freertos.org/.

[11] Zephyr Project[EB/OL]. 2025. https://zephyrproject.org/.

[12] Apache NuttX[EB/OL]. 2025. https://nuttx.apache.org/.

[13] Baccelli E, et al. RIOT: An open source operating system for low-end embedded devices in the IoT[J]. IEEE Internet of Things Journal, 2018.

[14] Davis R I, Burns A. A survey of hard real-time scheduling for multiprocessor systems[J]. ACM Comput. Surv., 2011.

[15] Brandenburg B B. Scheduling and Locking in Multiprocessor Real-Time Operating Systems [C]//PhD Diss., Uni. North Carolina. 2011.

[16] Zapata O, Álvarez P M. EDF and RM Multiprocessor Scheduling Algorithms : Survey and Performance Evaluation[C]//2005.

[17] Klabnik S, Nichols C. The rust programming language[M]. 2022.

[18] Ariel OS Benchmarks[EB/OL]. 2025. https://github.com/elenaf9/ariel-os-benchmarks.

[19] Warden P. Understanding the Raspberry Pi Pico’s Memory Layout[Z]. 2024.

[20] Ariel OS[EB/OL]. 2025. https://github.com/ariel-os/ariel-os.

[21] SmolTCP Net. Stack[EB/OL]. 2025. https://github.com/smoltcp-rs/smoltcp.

[22] Tschofenig H, Baccelli E. Cyberphysical Security for the Masses: A Survey of the Internet Protocol Suite for Internet of Things Security[J]. IEEE Security & Privacy, 2019.

[23] Laze Meta-Build System[EB/OL]. 2025. https://github.com/kaspar030/laze.

附录 B　补充内容

B.1　插图

addrspace.md　　2025-05-09

接口名称	功能
`base`	返回地址空间的基址
`end`	返回地址空间的结束地址
`size`	返回地址空间的大小
`page_table`	返回内部页表的引用
`page_table_root`	返回页表的根物理地址
`contains_range`	检查地址范围是否在地址空间内
`new_empty`	创建一个新的空地址空间
`copy_mappings_from`	从另一个地址空间复制页表映射，通常用于将内核空间映射的一部分复制到用户空间
`clear_mappings`	清除给定地址范围内的页表映射，通常与 `copy_mappings_from` 配合使用
`validate_region`	验证地址范围是否在地址空间内且地址是否对齐
`find_free_area`	查找一个可以容纳给定大小的空闲区域，从给定的提示地址开始搜索，区域应在给定的限制范围内
`map_linear`	添加一个新的线性映射
`map_alloc`	添加一个新的分配映射
`populate_area`	用物理帧填充区域，若区域包含未映射区域则返回错误
`unmap`	移除指定虚拟地址范围内的映射
`unmap_user_areas`	移除地址空间中的所有用户区域映射
`process_area_data`	使用给定的函数处理该区域的数据
`process_area_data_with_page_table`	辅助函数，用于处理数据的底层逻辑
`read`	从地址空间读取数据
`write`	向地址空间写入数据
`protect`	更新指定虚拟地址范围内的映射
`clear`	移除地址空间中的所有映射
`check_region_access`	检查对指定内存区域的访问是否有效
`handle_page_fault`	处理给定地址处的页错误，若成功处理则返回 `true`
`clone_or_err`	通过在新的页表中重新映射所有 `MemoryArea` 并复制用户空间中的数据来克隆一个 `AddrSpace`
`fmt`	实现 `fmt::Debug` 特性，用于调试输出地址空间的信息
`drop`	在 `AddrSpace` 实例被销毁时，移除所有映射

1 / 1

图 B.1　AddrSpace 结构体接口

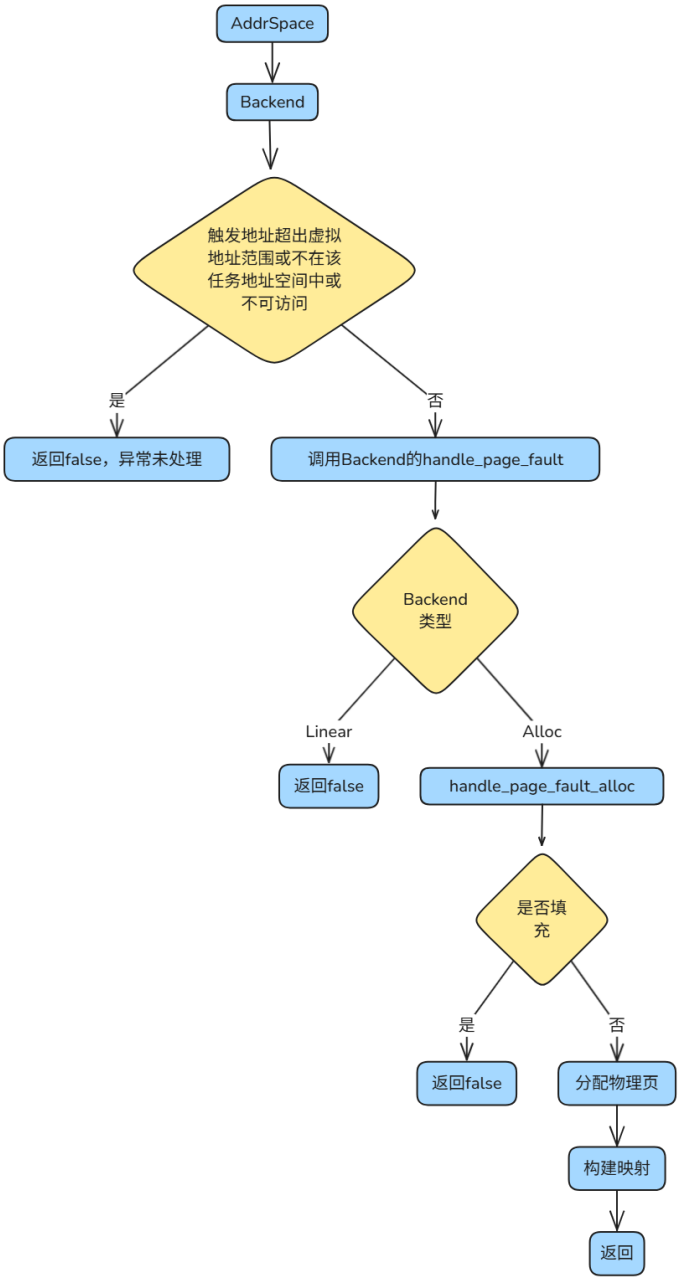


图 B.2　handle_page_fault 函数流程图

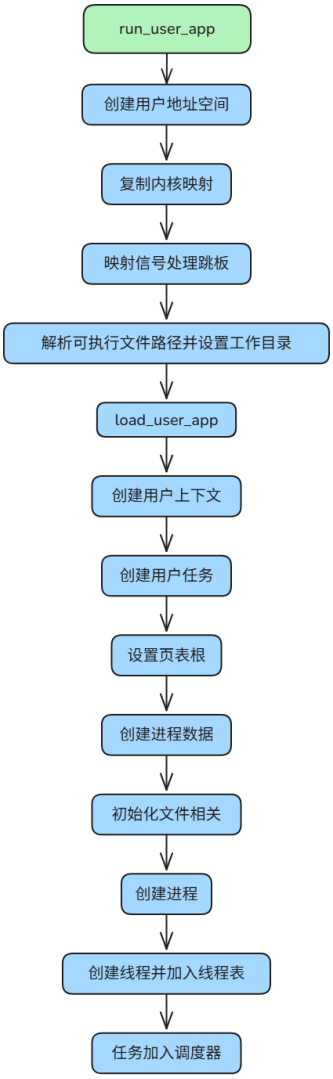


图 B.3　run_user_app 函数流程图

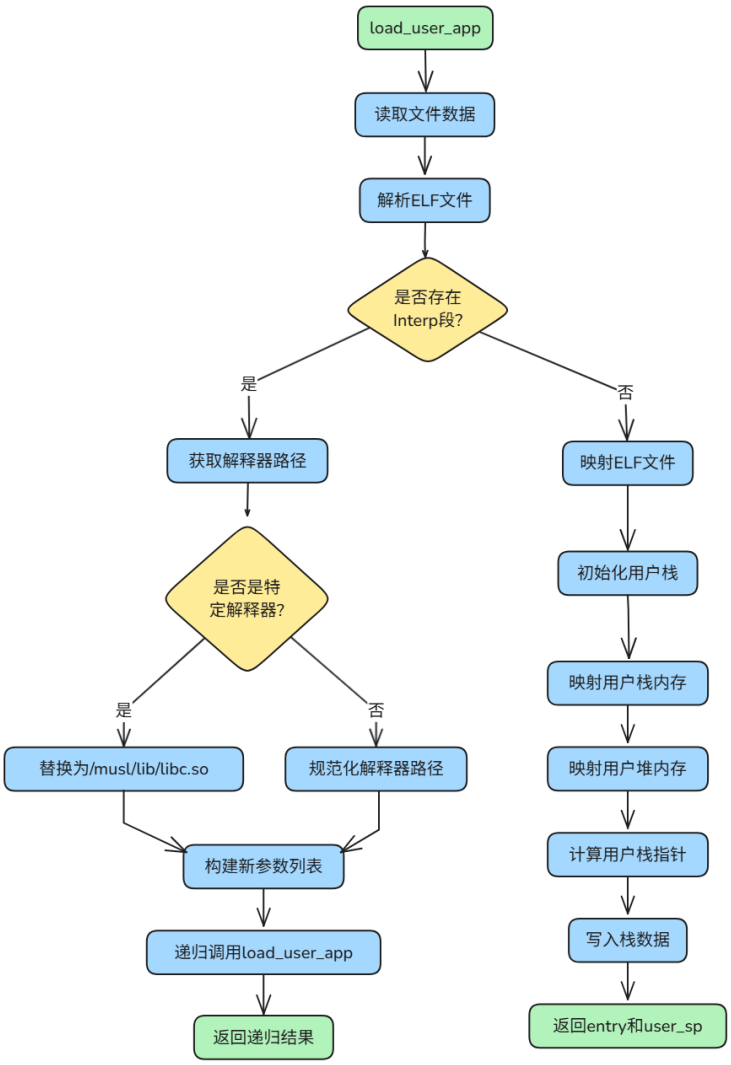


图 B.4　load_user_app 函数流程图

致　谢

衷心感谢导师陈渝副教授在项目完成过程中对我的的鞭策和指导。在研究过程中，陈老师提供的思路和指导总能让我有所突破。

感谢参与 ArceOS 和 starry-next 项目的所有同学，他们的帮助和支持使我的工作能够顺利进行。

感谢我的家人和朋友们，他们的鼓励和陪伴让我能够以积极的态度面对挑战和困难。

感谢四年来遇到的每一位老师与同学，你们的陪伴、教导和帮助，让我的校园时光美好又难忘。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____