

摘要

随着云计算、物联网等新兴技术的发展，传统宏内核操作系统在灵活性和可扩展性方面的不足日益凸显。为满足多样化应用场景需求，本文基于支持组件可重用设计的 ArceOS 基层架构与 Starry-Next 框架，开展组件化宏内核操作系统任务管理组件的设计与实现研究。通过对 ArceOS 基座代码和 Starry-Next 框架的深入分析，设计任务数据结构，完善任务创建、销毁、切换等操作，并基于此补充实现 execve、futex 等多个与任务管理相关的系统调用。同时，搭建基于 Ubuntu24.04 系统、支持多种架构的开发测试环境，采用操作系统大赛测例及自主编写简单测例的方式，对任务管理组件进行功能与性能测试。测试结果表明，该组件实现了预期功能，在不同负载下展现出良好的性能表现，能够支持复杂 Linux 应用程序的运行，为定制更便捷、高效、安全的宏内核操作系统提供了有力支持。

关键词：组件化宏内核操作系统；任务管理组件；系统调用；ArceOS；Starry-Next

Abstract

With the development of emerging technologies such as cloud computing and the Internet of Things, the deficiencies of traditional monolithic kernel operating systems in terms of flexibility and scalability have become increasingly prominent. To meet the requirements of diverse application scenarios, this paper conducts research on the design and implementation of the task management component of a componentized monolithic kernel operating system based on the ArceOS underlying architecture and Starry-Next framework that support component reusable design. Through in-depth analysis of the ArceOS base code and the Starry-Next framework, the task data structure is redesigned, and operations such as task creation, destruction, and switching are improved. Based on this, multiple system calls related to task management, such as execve and getpid, are supplemented and implemented. At the same time, a development and testing environment based on the Ubuntu24.04 system and supporting multiple architectures is built. The task management component is tested for functionality and performance by using test cases from the operating system competition and independently 编写 simple test cases. The test results show that the component achieves the expected functions, demonstrates good performance under different loads, and can support the operation of complex Linux applications, providing strong support for customizing more convenient, efficient, and secure monolithic kernel operating systems.

Keywords: componentized monolithic kernel operating system; task management component; system call; ArceOS; Starry-Next

目 录

第 1 章 引 言.....	1
1.1 课题背景	1
1.2 相关研究工作	2
1.2.1 理论研究现状	2
1.2.2 已有的操作系统实现现状	3
1.3 组件化操作系统	4
1.3.1 宏内核下的组件化操作系统	4
1.3.2 任务管理模块	4
1.4 本文工作	5
第 2 章 基座代码架构分析.....	6
2.1 整体架构概述	6
2.2 基座代码的任务管理组件	7
2.2.1 axruntime	7
2.2.2 axtask	7
2.2.3 axprocess	8
2.2.4 axns.....	9
2.3 实现基础宏内核操作系统	10
2.3.1 系统启动与 axhal 初始化	10
2.3.2 axruntime 启动与初始化	12
2.3.3 应用程序运行与用户地址空间创建	13
2.3.4 系统调用处理	13
2.3.5 应用程序退出	15
第 3 章 Starry-Next 框架分析.....	16
3.1 框架总体架构	16
3.2 与基座代码的接口分析	18
3.2.1 任务调度接口	18
3.2.2 多任务管理接口	19
3.2.3 内存管理接口	20
3.2.4 文件系统接口	20
3.3 Starry-Next 中的任务管理模块	21

第 4 章 任务管理组件设计与实现	24
4.1 开发环境与工具	24
4.2 模块总体设计思路	25
4.2.1 任务数据结构设计	25
4.2.2 任务创建、销毁、切换设计	26
4.2.3 与其他组件的交互关系	27
4.3 模块完成思路	27
4.4 系统调用实现	27
第 5 章 任务管理组件测试与分析	35
5.1 测试用例	35
5.2 自己编写简单测例	36
5.3 测试结果分析	38
5.3.1 功能测试	39
5.3.2 性能测试	39
第 6 章 结 论	41
参考文献	42
附录 A 补充内容	43
致 谢	45
声 明	46

第1章 引言

1.1 课题背景

操作系统（Operating System, OS）是计算机系统的核心软件，承担着管理和协调计算机硬件与软件资源的重任，其性能和功能直接影响着整个系统的运行效率和用户体验。而以 Linux 为代表的宏内核操作系统凭借其在资源管理和系统调用方面的高效性，在服务器、个人计算机等领域被广泛应用，支撑着大量业务的稳定运行与高效管理。但随着信息技术领域的快速发展，云计算、物联网等新兴技术不断涌现，计算机设备的应用场景日益复杂，对操作系统的灵活性和可扩展性提出了更高的要求。面对种类繁多、功能各异的设备，操作系统需要灵活适配、快速响应不同设备的多样化需求，而传统的宏内核操作系统将所有功能均放在单一的内核空间中实现，服务高度集中化，虽然能避开复杂的进程间通信机制，减少用户态和内核态之间的切换开销，提高系统调用效率，却也导致了内核代码规模庞大，可维护性和可扩展性较差，难以复用到资源差异较大的异构平台，而未熟练掌握内核开发技能的用户又难以针对不同场景需求开发适配的内核，因此一个能支持用户自由搭建所需功能的组件化宏内核操作系统便显得尤为重要。

任务管理作为宏内核操作系统的核心功能，负责高效调度任务、合理分配资源等关键工作，从而保障整个系统的有序运行，因此在搭建组件化操作系统的過程中首先且必须要实现的便是任务管理组件。在传统宏内核操作系统中，任务管理模块与内核其他部分紧密耦合，这种设计虽然在一定程度上保证了任务调度的高效性，但也带来诸多弊端。当系统需求发生变化，如新增特殊任务类型或调整调度策略时，对任务管理模块的实现往往需要进行较大的改动，这不仅增加了开发和维护成本，还容易引入新的漏洞和错误。

在组件化宏内核操作系统的构建背景下，任务管理组件的实现面临新的挑战与机遇。一方面，需要将原本庞大且复杂的任务管理功能进行合理拆分与封装，使其成为具有独立功能的组件，尽量实现与内核其他组件的解耦，以提升系统的可维护性和可扩展性。另一方面也要确保组件化后的任务管理模块依然能够高效地完成任务调度、资源分配等核心工作。这就要求在设计任务管理组件时，需要充分考虑不同应用场景对任务调度策略的需求差异，提供多种任务调度策略来进行灵活切换。同时，为了提高任务管理组件的通用性和可复用性，还需遵循统一的系统调用接口规范，使其能够方便地集成到不同的组件化宏内核操作系统中，为用户快速定制满足特定需求的操作系统提供有力支持。

本课题旨在基于支持组件可重用设计的组件化操作系统基层架构 ArceOS，搭建其上层系统调用，从而设计并封装出能直接支持 Linux 应用的组件化宏内核操作系统的任务管理组件，帮助定制更便捷、高效、安全的宏内核操作系统。

1.2 相关研究工作

1.2.1 理论研究现状

在组件化操作系统的理论研究方面，众多学者均对其架构方式、设计原则以及运行机制进行了深入的探讨。对于组件化宏内核操作系统而言，其研究重点在于如何在保持宏内核架构优势的同时，实现组件化带来的灵活性与可定制性。一些早期的研究尝试将宏内核中的部分功能进行模块化封装，以组件形式实现特定服务，如文件系统、设备驱动等。然而，这些早期尝试存在诸多问题，例如组件之间的耦合度较高，导致系统的可维护性和扩展性受限；在组件的动态加载与替换方面亦缺乏有效的机制支持，难以满足运行时的动态配置需求。Fassino 等提出的 THINK 框架为组件化宏内核操作系统的设计提供了新的思路和方法^[1]，受标准化的 ODP 参考模型、ANSA 和 Jonathan 等分布式中间件研究的启发，THINK 引入并系统应用了灵活的绑定模型，允许组件以多种非预定义的方式进行绑定和组装，从而在处理异构环境时能提供更大的灵活性，利于构建更小体积的特定内核。

而随着计算机领域的飞速发展，对可用于特定场景的新型操作系统内核的需求越来越高。在面向泛在计算领域异构硬件的研究中，吉金字等提出了 BrickOS 这种积木式设计方法，用于敏捷定制领域操作系统内核^[2]。不同于传统操作系统内核的固定模式，BrickOS 围绕内核基础功能进行深度解构，将任务调度、内存管理、文件系统等基本功能划分拆解为可独立配置和自由定制的组件单元。此外，在内核架构层面，BrickOS 支持简要内核、微内核、宏内核等多种架构的动态切换，用户可以根据不同应用需求自由选择所需架构，实现内核的定制化搭建。这种积木式设计打破了传统内核的紧耦合结构，增强了面向异构平台的内核可扩展性，为组件化操作系统在复杂硬件环境下的定制应用提供了新的思路。

此外，对于组件化操作系统的性能优化和安全机制等方面也有诸多研究。在性能优化上，一些研究关注如何降低组件间通信的开销，通过优化通信机制、合理安排组件运行位置等方式，提升系统整体性能。在安全机制方面，也有研究关注如何为不同组件的内存增加隔离性，以提升整个系统的安全性，如 BrickOS 便为单地址空间的内存隔离实现了底层硬件内存保护机制的抽象^[2]。这些理论研究为组件化操作系统的实际设计与实现提供了坚实的理论基础，推动了该领域的不断发展。

1.2.2 已有的操作系统实现现状

目前，已有多个操作系统对宏内核操作系统进行了各自的创新设计与实现，其中 ByteOS、DragonOS 以及星绽操作系统均使用 Rust 语言开发并提供了完备的宏内核相关功能，兼容 Linux 应用，他们的实现为本课题的设计研究提供了宝贵的借鉴经验。

ByteOS 是一个开源的操作系统项目，将进程管理、内存管理、文件系统等核心模块紧密集成，实现了较为简洁的宏内核架构。在任务管理方面，ByteOS 使用进程控制块（PCB）作为管理进程的数据结构，记录进程状态、优先级、程序计数器等关键信息，线程管理同样使用类似的线程控制块（TCB）来管理线程的上下文信息，包括陷阱帧、信号掩码、子线程退出标志、信号队列、退出信号和线程退出码等。ByteOS 支持多种硬件平台，如 riscv64、aarch64、x86_64、loongarch64 等，并能在这些平台上运行 Linux 应用程序。

DragonOS 同样基于宏内核架构，是一个面向云计算轻量化场景的，完全自主内核的 64 位操作系统。其内核包含完备的宏内核操作系统所需功能，并实现了优秀的内存管理模块，对内核空间和用户空间的内存分配、释放、管理等进行了封装，主要由硬件抽象层、页面映射器、页面刷新器、页帧分配器、小对象分配器、MMIO 空间管理器、用户地址空间管理机制、系统调用层和 C 接口兼容层等组件组成。在任务管理方面，DragonOS 实现了完善的进程管理和多核调度，并支持内核线程和 kthread 机制，为内核线程的创建和管理提供了方便的接口，保证系统能高效处理各种任务。具体而言，其进程管理功能涵盖了进程创建、进程回收、fork、exec 等操作，同时支持进程睡眠（包括高精度睡眠）。在调度方面，DragonOS 拥有 CFS 调度器、实时调度器（FIFO、RR），不仅支持单核调度，还具备多核调度和负载均衡能力，能够根据系统的负载情况和任务的优先级合理分配 CPU 资源，提高系统的整体性能。

星绽操作系统则致力于充分发挥 Rust 潜力，以安全高效的方式实现 Linux ABI 并满足专有内核模块的业务需求。星绽在系统中引入了新颖的 framekernel 架构，将整个 OS 置于同一地址空间，但将内核划分为 OS Framework 和 OS Services 两部分。只有 OS Framework 允许使用 unsafe Rust，而 OS Services 必须使用 safe Rust。这种架构设计将内核的内存安全问题主要集中在 OS Framework 层面，从而减少了可信计算基（Trusted Computing Base，TCB），同时单一地址空间又使得内核各部分之间能够通过函数调用和共享内存等高效方式进行通信，在保证性能的同时提高了安全性。星绽操作系统的任务管理机制在任务创建、调度、同步与通信以及退出等方面都有其独特的设计和实现，在任务调度上也会根据任务的状态和资源

需求进行动态调整。在云环境中，星绽操作系统由于其高内存安全性，适用于对安全要求较高的场景，如 VM-based TEEs 和安全容器。

1.3 组件化操作系统

1.3.1 宏内核下的组件化操作系统

宏内核下的组件化操作系统在传统宏内核架构的基础上进行创新发展，将操作系统的各个核心功能封装为独立的组件，各功能不再紧耦合，而是在通过组件间的标准化接口进行交互与协作。这种架构模式既能保留宏内核的高性能优势，又能赋予系统良好的可维护性与灵活的可扩展性，从而大幅拓展宏内核操作系统的应用范围。

1.3.2 任务管理模块

任务管理模块作为组件化操作系统的关键组成部分，通过管理进程与线程、控制任务状态与生命周期以及灵活进行任务调度来实现对宏内核核心功能的支持。进程作为系统资源分配和 CPU 调度的基本单位，是程序在计算机中的一次动态执行过程，每个进程都拥有独立的地址空间及系统资源，一般通过进程控制块来实现对进程状态、优先级、资源占用等信息的精确记录。而线程作为进程内的轻量级执行单元，则通过线程控制块来实现，通过共享进程的资源实现协同工作，但也拥有独立的线程栈和程序计数器以实现独立调度，保证资源的高效利用。任务状态描述了任务在执行过程中的不同阶段，常见的任务状态包括就绪态、运行态、阻塞态和终止态，任务状态的转换构成了任务的生命周期，通过任务管理模块进行监控和控制。而任务调度策略的核心目标是平衡公平性与效率，基于此存在一系列策略对任务的状态进行管理与切换，常见策略包括时间片轮转（Round-Robin, RR）、先来先服务（First Come First Serve, FCFS）、多级反馈队列调度（Multi-Level Feedback Queue, MLFQ）等，通过灵活运用各种调度策略，操作系统能根据任务特点和资源情况对 CPU 资源进行高效分配，提升系统效率，满足不同应用场景的需求。

在宏内核架构中，任务管理模块贯穿操作系统的整个运行过程，在系统启动阶段负责任务相关数据结构与资源的初始化，在系统运行过程中则与内存管理、文件系统等模块协作，调度任务、管理资源，并在系统运行结束后回收相关资源，保障系统能够高效、稳定地运行。

1.4 本文工作

本文旨在实现高效、完备、可靠的宏内核任务管理组件，将基于基座代码 ArceOS 中的单位内核组件以及 Starry-Next 整体框架展开深入研究与实践，通过对任务管理相关数据结构的设计运用、系统调用接口的补全实现以及对整个任务管理组件功能与性能的集成测试，实现对组件化宏内核的核心支持。

此外，本课题同时为 ArceOS 及 Starry-Next 框架中的任务相关功能与接口提供了详细的说明文档，以供使用者参阅。

本文分为 6 个章节，在第一章介绍课题背景及组件化宏内核相关研究工作，在第二章将介绍本课题所使用的基座代码 ArceOS 的整体架构以及任务管理相关单元模块，在第三章则将对 Starry-Next 框架进行介绍、接口分析及任务管理模块的相关功能，在第四章将介绍任务管理组件的设计与实现，在第五、六章则将分别对组件进行测试分析以及最终总结。

第 2 章 基座代码架构分析

本章将描述基座代码 ArceOS 的整体架构，并着重分析其任务管理相关核心组件的功能与实现。通过深入分析和熟练掌握基座代码的组成，可以用其实现基础宏内核操作系统并运行简单用户程序。

2.1 整体架构概述

ArceOS 是一个使用 Rust 语言编写的实验性模块化微虚拟机操作系统，即单内核操作系统（Unikernel），它具有高度的可定制性和灵活性，旨在提供一个高效、安全且易于扩展的操作系统平台，为各种特定场景的应用提供可定制性的操作系统底层支撑。ArceOS 采用模块化设计，以比传统设计更细粒度的划分将操作系统拆分为多个独立的模块，如运行时模块、任务管理模块、设备驱动模块等，并依赖 Rust 语言的包管理机制来实现模块的组件化，为每个模块提供明确的功能和接口，模块之间通过接口进行良好交互。ArceOS 中的组件分为两类，与操作系统功能无关的称为元件，位于最底层，为其上层的组件提供可重用的数据结构、硬件处理等相关操作，而与操作系统相关的组件则称为模块，位于需要依赖特定操作系统的需求进行特定的修改，其位于元件的上层，通过调用元件层的基础功能实现对操作系统基本功能的支持。

由于 ArceOS 的底层元件具有高度的可定制性和灵活性，其可以针对不同的应用需求定制出轻量级的操作系统，提高资源利用率和性能，这也为本课题基于 ArceOS 开发组件化宏内核操作系统提供了便利的支撑，在实现过程中不再需要深入探究可复用的底层操作系统知识，而只需关注相关接口以及模块层提供的功能支持，针对宏内核的功能需求对模块层进行相关调用与修改，将其封装成完备的系统调用与宏内核操作系统进行交互，即可便利快捷地实现操作系统上层的任务管理组件，实现实任务的创建、调度等管理需求。

ArceOS 共计拥有 11 个模块，其中不可缺少的核心组件为应用运行时模块（`axruntime`）、硬件抽象层模块（`axhal`）以及动态内存分配模块（`axalloc`），分别负责内核启动初始化、硬件支持配置以及页面和字节分配。在此之外将任务管理、设备驱动、网络栈和文件系统四大模块进行抽象封装，分别形成 `axtask` 模块、`axdriver` 模块、`axnet` 模块和 `axfs` 模块。基于 `axruntime` 模块及 `axhal` 模块，结合配置平台相关参数的 `axconfig` 模块和输出内核日志信息的 `axlog` 模块，即可实现一个最简易的操作系统来运行 `helloworld` 应用程序，而在此基础上结合 `axtask` 模块和 `axalloc` 模

块，便可以实现一个支持动态内存分配和多线程的操作系统，能够运行大部分基础应用程序，对其余应用的支持扩展则可以通过组合不同的模块加入新的功能来实现。

2.2 基座代码的任务管理组件

在 ArceOS 的 11 个模块组件中，有三个组件与任务管理息息相关，它们分别是：`axruntime` 运行时模块、`axtask` 任务管理与调度模块以及 `axns` 命名空间模块。本节将重点论述这三个模块，对其功能与实现进行展开分析介绍。

2.2.1 `axruntime`

`axruntime` 是 ArceOS 的运行时库，作为整个框架的运行核心，任何使用 ArceOS 的应用程序都需要链接该库。它负责在进入应用程序的 `main` 函数之前进行一系列的初始化工作，确保系统在一个合适的状态下运行用户程序，具体而言，主要实现了如下功能：

- 初始化日志系统：配置日志级别，为后续系统运行时的信息输出提供支持。
- 初始化内存分配器：如果启用了 `alloc` 特性，会初始化全局内存分配器，为动态内存分配做准备。
- 初始化平台设备：调用 `axhal` 的 `platform_init` 函数，对平台相关的设备进行初始化。
- 初始化任务调度器：若启用了 `multitask` 特性，会初始化任务调度器，为多任务处理提供支持。
- 初始化文件系统、网络和显示模块：根据启用的特性，初始化相应的模块，如文件系统、网络和显示模块。
- 启动多核 CPU：如果启用了 `smp` 特性，会启动其他 CPU 核心。
- 初始化中断处理：若启用了 `irq` 特性，会初始化中断处理程序，设置定时器中断等。
- 初始化线程本地存储：在特定条件下，初始化线程本地存储。

2.2.2 `axtask`

`axtask` 是 ArceOS 中任务管理与调度的核心模块，负责对系统中的任务进行全生命周期的管理，由于最初版本的 ArceOS 主要是为单内核实现的，在单内核中没有进程的概念，或可将整个系统整体视为一个大的进程，因此其实现的任务管理也即线程管理，主要功能如下：

- 任务创建与销毁：提供 `spawn_task` 函数用于创建新任务，`exit` 函数用于退出当前任务。
- 任务调度：根据选择的调度算法（如 FIFO、RR、CFS）选择下一个要执行的任务。
- 任务等待与唤醒：提供 `wait_queue` 和 `run_queue` 结构用于管理任务的等待和运行，并提供 `wait` 函数用于阻塞当前任务。
- 任务优先级设置：可以通过 `set_current_priority` 函数设置当前任务的优先级。

具体而言，在任务创建方面，`axtask` 会根据用户请求或系统需求，为新任务分配必要的系统资源，如内存空间、文件描述符、CPU 时间片等，并创建相应的任务控制块（TCB）来记录任务 ID、优先级、运行状态、程序计数器等信息。在任务创建完成后，`axtask` 会将任务加入到相应的任务队列中等待调度执行。而对于任务调度部分，`axtask` 会依据预设的调度策略，如优先级调度、时间片轮转调度等，从任务队列中选择合适的任务投入运行。此外，`axtask` 还负责任务的暂停、恢复和销毁操作，在任务暂停时保存任务的当前运行状态，在任务恢复时重新激活任务并恢复其运行环境，在任务销毁时释放任务占用的所有资源，回收任务控制块，从而确保系统资源得到有效利用。

2.2.3 axprocess

为了支持对多进程、会话等的管理，ArceOS 增加了 `axprocess` 模块来对进程、进程组、会话和线程等多个层面进行管理，其主要功能如下：

- 进程创建与销毁：提供 `Process::new_init` 函数用于创建初始进程，`Process::fork` 函数用于创建子进程，`Process::exit` 函数用于终止进程，`Process::free` 函数用于释放僵尸进程。
- 进程关系管理：提供 `Process::parent` 函数用于获取父进程，`Process::children` 函数用于获取子进程列表。此外，每个进程都属于一个进程组和一个会话，可以通过 `Process::group` 函数获取进程所属的进程组，通过 `ProcessGroup::session` 函数获取进程组所属的会话。
- 进程组创建与管理：提供 `Process::create_group` 函数用于创建新的进程组，并将进程移动到该进程组中，如果进程已经是进程组的领导者，则该函数返回 `None`。
- 会话创建与管理：提供 `Process::create_session` 函数用于创建新的会话和新的进程组，并将进程移动到该会话和进程组中，如果进程已经是会话的领导者，则该函数返回 `None`。
- 线程创建与管理：提供 `Process::new_thread` 函数用于在进程中创建新的线程，

`Thread::exit` 函数用于退出线程。

具体而言，在 `axprocess` 中，进程（`Process`）是系统中资源分配和调度的基本单位，包含了进程的基本信息、线程组、子进程列表、所属进程组和会话等。而进程组（`ProcessGroup`）是一组进程的集合，每个进程组有一个唯一的 `pgid`，并且属于一个特定的会话。会话（`Session`）是一组进程组的集合，每个会话有一个唯一的 `sid`。线程（`Thread`）则是进程内的执行单元，每个线程属于一个特定的进程。

使用 `Process::new_init` 函数或 `Process::fork` 函数创建进程时，会为其分配必要的系统资源，如进程 ID、线程组、数据存储等，并创建相应的进程控制块来记录进程的相关信息，如 `pid`、`parent`、`group` 等。创建初始进程时，会为其分配一个新的会话和进程组，而子进程则会被加入到其父进程的子进程列表中，同时也会继承父进程的进程组。进程退出时首先通过 `Process::exit` 方法终止进程，将其标记为僵尸进程，并将子进程交给初始进程继承，而对于僵尸进程的释放则通过 `Process::free` 方法来将其从父进程的子进程列表中移除。对于进程组和会话的退出，当进程退出并释放后，其所属的进程组会自动清理该进程，确保进程组中不再包含已退出的进程，同样地，进程所属的会话也会自动清理该进程组，确保会话中不再包含已退出的进程组。而使用 `Process::new_thread` 函数创建新线程时会为其分配线程 ID 和关联的进程，并将线程添加到进程的线程组中。使用 `Thread::exit` 函数退出线程时若线程是线程组中的最后一个线程，则返回 `true` 表示线程组已全部退出。

2.2.4 axns

`axns` 命名空间组件在 ArceOS 中为任务提供了独立的运行环境，实现了系统资源的隔离与共享，它提供了一种机制，使得不同的线程可以共享或隔离虚拟地址空间、工作目录和文件描述符等系统资源，主要功能如下：

- 资源隔离与共享控制：支持线程本地资源和全局资源的共享。
- 资源管理与懒初始化：提供了一套机制来管理系统资源，确保资源的正确分配和释放，同时支持资源的懒初始化。
- 提高系统的安全性和可维护性：通过资源隔离与共享的机制，为多任务并发运行提供安全、稳定的环境，同时将资源的管理和使用进行分离，使得代码结构易于维护和扩展。

对于资源的隔离与共享控制，`axns` 模块为每个线程提供了自己独立的命名空间，使得不同线程在操作资源时相互隔离，从而避免了资源冲突。例如，每个线程可以有自己独立的工作目录、文件描述符表等，确保线程之间的操作不会相互干扰。此外，该模块还通过 `ResArc` 类型来管理全局共享资源，如全局配置信息、共享内存区域等，使得需要在多个线程间共享数据时不同线程可以安全地访问和修

改这些资源。

对于资源的管理与懒初始化，`axns` 模块在创建和销毁命名空间时会自动处理相关资源的分配和回收，避免资源泄漏。而使用 `ResArc` 类型的 `init_new` 方法可以实现资源的懒初始化，即在资源首次使用时才进行初始化，这有助于减少系统的启动时间和内存开销，特别是对于一些占用大量资源或初始化过程较为复杂的应用，可以提高其运行效率。

此外对于系统的安全性和可维护性，`axns` 模块通过资源隔离，减少了线程之间意外修改对方资源的风险，例如，一个线程无法直接访问另一个线程的私有文件描述符，从而防止了数据的非法访问。而资源管理与使用进的分离则有利于不同的线程独立地操作自己的命名空间，不会影响其他线程的功能。

2.3 实现基础宏内核操作系统

由于 ArceOS 的设计初衷是使用组件化的方式支持单内核操作系统，而对于借助所设计的组件完成对其他形态操作系统的支持仅停留在理论层面，因此本节将基于基座代码的详细内容与具体实现，论述如何使用基座框架实现基础宏内核操作系统并运行简单的用户程序。

从宏内核系统的启动，到用户程序的加载、执行直至最后的成功退出，主要分为如下五个步骤：

1. 系统启动与 `axhal` 初始化。
2. `axruntime` 启动与初始化。
3. 应用程序运行与用户地址空间创建。
4. 系统调用处理。
5. 应用程序退出。

2.3.1 系统启动与 `axhal` 初始化

首先 `axhal` 作为 ArceOS 的硬件抽象层，为跨平台操作提供统一的 API，确保系统能适配不同硬件架构。系统启动时，首先会涉及到 `axhal` 的初始化工作。在此之前，硬件抽象层需要先得到一系列平台常量和内核参数，这些配置信息来自于 `axconfig` 配置模块，`axconfig` 会依据不同的环境变量生成相应的配置文件，明确物理内存基地址、内核加载地址、栈大小等关键参数，这些参数是系统后续资源管理与程序加载的重要依据。其中，物理内存基地址指定了物理内存开始的地址，物理内存是硬件资源的重要组成部分，操作系统需要这个信息来正确地管理和分配内存，完成后续的内存映射、页表设置等操作。在某些平台上，物理内存可能从地

址 0x0 开始，而在其他平台上可能有不同的起始地址（这里看看不同架构给出实例），因此需要在系统启动前依据不同架构配置好相应的物理内存基地址，应用程序才能在正确的架构上成功运行。而内核是操作系统的核心部分，它需要被加载到内存中才能运行。内核加载地址指定了内核在内存中的加载位置，操作系统会根据这个地址将内核代码和数据加载到相应的内存区域，然后从该地址开始执行内核代码，不同的平台可能存在不同的内核加载地址要求，配置文件需要根据具体平台进行设置。此外，栈大小也是系统运行前需要配置的重要参数，栈是程序运行时用于存储局部变量、函数调用信息等的内存区域，栈大小决定了程序在运行过程中能够使用的栈空间大小。如果栈空间过小，可能会导致栈溢出错误；如果栈空间过大，则会浪费内存资源。因此，配置文件需要根据不同的应用场景和平台特性，合理设置栈的大小。

而 axhal 模块在获得初始化好的配置信息后，就可以开始根据程序指定的目标硬件架构和硬件运行平台生成链接脚本，以确保程序能够正确链接到相应平台。链接脚本生成后系统便进入了启动阶段，axhal 则将开始硬件初始化阶段。在 axhal 中利用 Rust 组织代码的特性实现了 cpu、irq、mem 等关键模块（mod），分别负责 CPU、中断处理、内存等硬件相关抽象与初始化工作。其中 cpu 模块定义了一系列函数来处理 CPU 相关的操作，包括 CPU 初始化、获取当前 CPU ID 等，irq 模块主要处理中断相关的操作，包括中断处理函数的注册和中断分发，mem 模块则主要处理内存区域的管理和地址转换等内存相关操作，不同模块所实现的具体功能函数如表 2.1 所示。

基于以上的关键功能，axhal 得以顺利进行硬件平台的初始化工作并为后续的应用运行提供支撑。进入启动流程后系统会根据不同的平台调用相应的硬件初始化函数，以 LoongArch 64 位 QEMU Virt 平台为例，在平台对应的 boot 文件中定义了主 CPU 和副 CPU 的最早入口点，在主 CPU 的入口点 _start 中会调用一系列函数来完成进入主函数前的设置操作，包括设置内存访问控制寄存器（CSR）、启动栈、初始化页表和 MMU、读取 CPU ID 等，最后跳转至 rust_entry 函数。rust_entry 函数会对内存、CPU 等进行相应的初始化，并最终调用主函数 rust_main 函数以进行运行程序的相关初始化操作。

简而言之，链接脚本生成后，系统会根据不同的平台和 CPU 类型调用相应的硬件初始化函数，在这个过程中会依据配置分别进行主 CPU 和副 CPU 的初始化，通常包括清除 BSS 段、设置异常向量基地址、初始化 CPU、初始化内存管理、初始化平台设备等。通过这些初始化步骤，系统可以正确地配置和管理硬件资源，为后续的程序执行提供基础。

表 2.1 axhal 模块核心功能函数列表

模块名	功能函数	描述
cpu	init_primary	初始化主 CPU，设置 CPU ID 并标记为主 CPU
	init_secondary	初始化副 CPU，设置 CPU ID 并标记为副 CPU ¹
	this_cpu_id	获取当前 CPU 的 ID
irq	this_cpu_is_bsp	判断当前 CPU 是否为主 CPU
	dispatch_irq_common	分发中断，调用相应的中断处理函数
	register_handler_common	注册中断处理函数并启用相应中断
mem	memory_regions	获取所有物理内存区域的迭代器
	virt_to_phys	将虚拟地址转换为物理地址
	phys_to_virt	将物理地址转换为虚拟地址

¹ (这里加脚注说明为何需要副 cpu)

2.3.2 axruntime 启动与初始化

在系统启动后，axruntime 开始承担起从裸机环境构建运行时环境的重任，还是以 LoongArch 64 位 QEMU Virt 平台为例，主 CPU 在经 _start 函数的一系列操作后最终跳转至 axruntime 中的 rust_main 函数，在其中它会根据 Cargo 文件定义的不同的特性进行条件初始化。首先是对日志进行初始化，借助 axlog 模块打印架构、平台、SMP 状态系统关键信息，初始化日志系统并设置日志级别。接着对不同的特性进行支持：

- 若启用了 alloc 或 alt_alloc 特性，则初始化内存分配器。
- 若启用了 paging 特性，则调用 axmm 模块中的 init_memory_management 函数来初始化内存管理。

随后调用 axhal 模块中为每个平台进行设备初始化的函数 platform_init 来对不同平台所需如中断控制器和定时器等的初始化。之后依旧根据不同特性进行不同部分的初始化：

- 当启用 multitask 多任务特性时，调用 axtask 模块的 init_scheduler 函数来初始化任务调度器。
- 当启用 fs、net 或 display 特性时，初始化设备驱动，并根据具体特性初始化文件系统、网络或显示系统。
- 若启用了 smp 多核特性时，调用 start_secondary_cpus(cpu_id) 来启动副 CPU。
- 当启用 irq 中断处理特性时，会调用 init_interrupt 来初始化中断处理。

- 在启用 tls 特性且未启用 multitask 特性时，会初始化线程本地存储。

完成上述所有初始化后，`rust_main` 函数将调用应用程序入口 `main` 函数以进行应用程序的运行。由于 ArceOS 的单地址空间特性，应用程序直接运行在核模式下，因此不需要进行上下文切换。这使得应用程序可以直接访问系统资源，避免了传统操作系统中用户态和内核态切换带来的开销，从而提高系统的执行效率。但对于实现宏内核操作系统而言，应用程序不应再与内核共享地址空间，而需要为每个用户程序进程都创建属于自己的独立地址空间，因此在下一小节将讲述如何为应用程序创建用户地址空间。

当应用程序的 `main` 函数执行完毕后，若启用了多任务特性，将会调用 `axtask` 模块的 `exit(0)` 退出函数来退出当前任务；若未启用多任务特性，则会调用 `axhal` 模块下的 `misc::terminate` 终止函数来终止整个系统的运行。

2.3.3 应用程序运行与用户地址空间创建

系统运行进入 `main` 函数就意味着到了用户地址空间创建与应用程序加载运行的步骤。在这一步，`axmm` 为实现对宏内核的支持提供了一定的支持，用户可以通过 `axmm` 模块的 `new_user_aspace` 函数创建用户地址空间，之后则需完成一个 `load_user_app` 函数来将用户应用程序二进制文件加载至地址空间指定位置，`load_user_app` 函数的示例逻辑如图 2.1 所示，主要完成的功能即读取用户程序文件，接着在地址空间映射分配内存，最后将文件数据复制到用户地址空间合适位置对应的物理地址中去：

加载好用户应用程序后需要为用户程序在用户空间中分配并初始化栈空间，之后便可通过创建一个新任务 `user_task` 并将用户地址空间和用户栈信息传递给该任务的方式来创建并启动用户进程。

2.3.4 系统调用处理

由于宏内核系统中用户态与内核态分离，在用户程序需要使用系统资源时，需要通过系统调用向内核请求服务。为此，首先需要实现系统调用处理函数，通过对系统提供的系统调用编号匹配相应的系统调用操作，分发给对应的系统调用处理函数。首先需要加入 `register_trap_handler` 宏用于注册系统调用处理函数，当发生系统调用时，会调用 `handle_syscall` 函数进行处理，图 2.2 提供了系统调用处理函数的简单逻辑，如匹配到 `SYS_EXIT` 系统调用时使用 `axtask` 模块的 `exit` 函数来退出进程，而对于未实现的调用则返回相应的错误码，确保用户程序与内核间的资源交互有序进行。

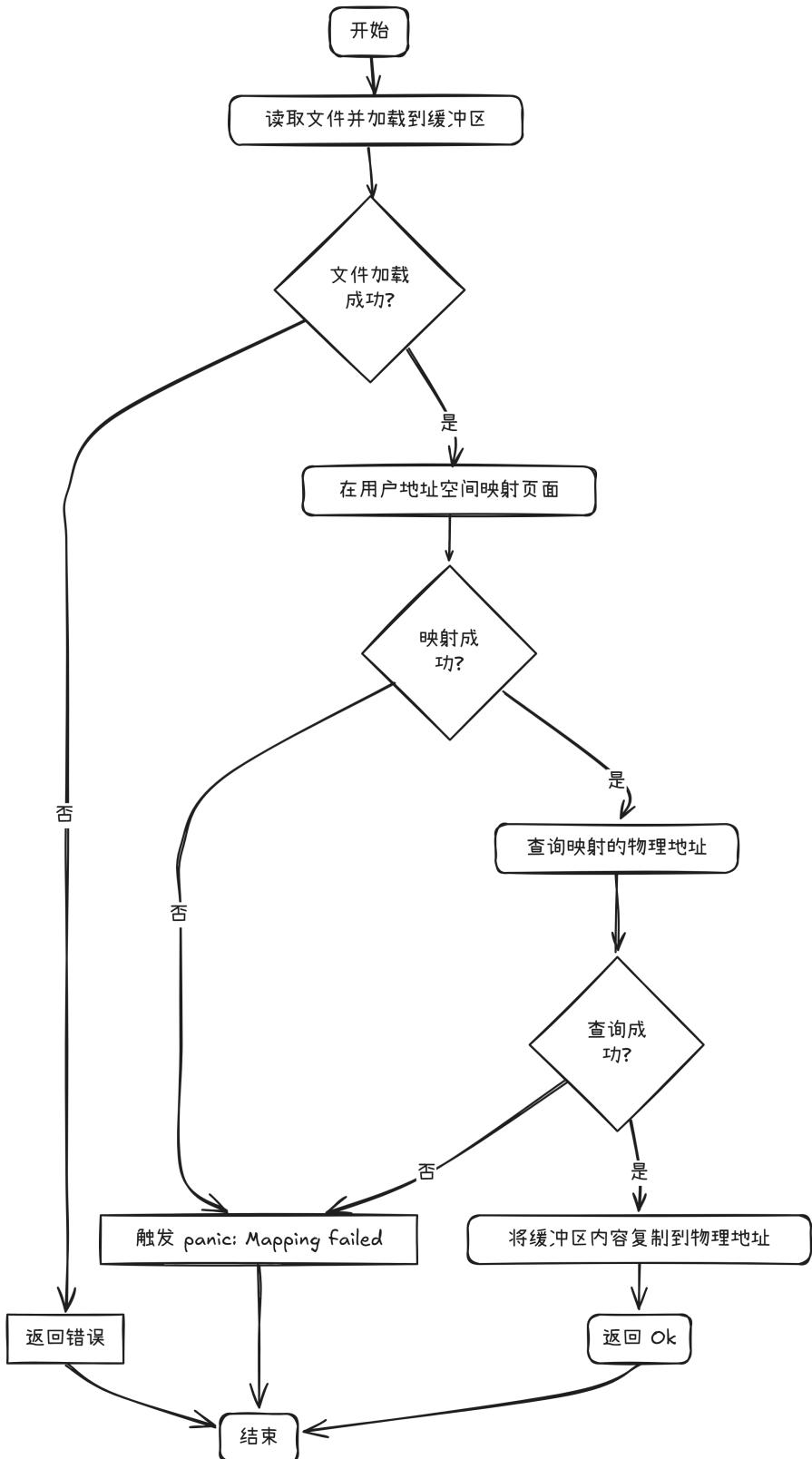


图 2.1 `load_user_app` 核心实现逻辑流程图

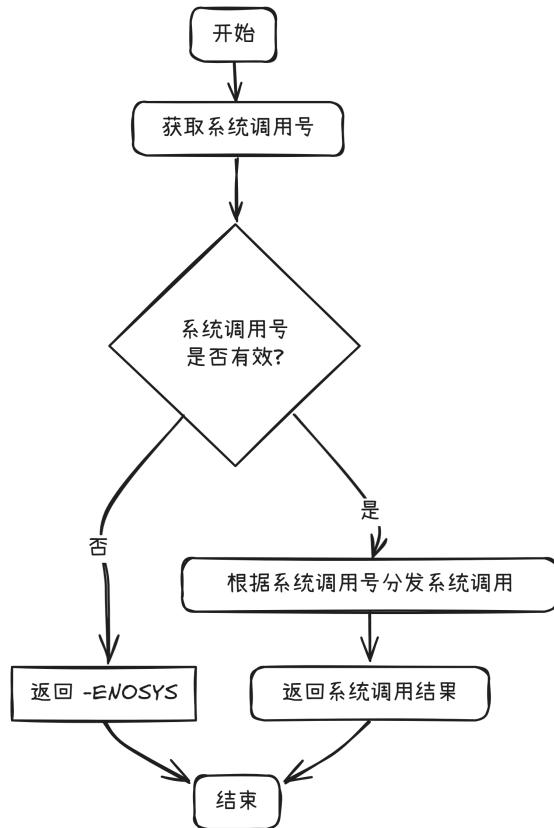


图 2.2 系统调用处理函数核心实现逻辑流程图

2.3.5 应用程序退出

当用户程序执行完毕或调用 `SYS_EXIT` 系统调用后系统将进入退出流程。在 `main` 函数中，对在第三步创建的新任务 `user_task` 通过 `user_task.join` 的方法阻塞线程，等待用户进程退出并获取退出码，至此即可完成简单用户程序在基础宏内核操作系统中的完整运行。

通过以上基于 ArceOS 基座框架的系统启动、运行时初始化、应用程序加载执行、系统调用处理以及程序退出等一系列操作，本节成功实现了基础宏内核操作系统环境下简单用户程序的运行，为进一步开发与优化宏内核操作系统的任务管理组件及其他功能模块奠定了理论与实践基础。

第3章 Starry-Next 框架分析

上一章描述了基座代码 ArceOS 的整体架构以及使用基座代码进行宏内核相关扩展的理论方式，本章将着重描述课题所使用框架 Starry-Next。作为对宏内核理论扩展方式的直接实践框架，Starry-Next 继承了 ArceOS 的组件化设计理念，以组件为基本单元构建并实现宏操作系统的各项功能。本章将对其整体架构及与基座代码的接口进行详细分析，并着重讲述 Starry-Next 中的任务管理模块。

3.1 框架总体架构

Starry-Next 整体框架采用了模块化的设计思想，如图 3.1 所示，框架主体部分大致可分为 src、core 和 api 三个模块，其中 src 模块主要为应用程序的入口进行封装并对系统调用的分发进行处理；core 模块实现了系统的核心功能，包括任务管理、内存管理、文件系统等，并向下依赖基座代码 ArceOS 中提供的底层支持，各个模块之间通过接口进行交互，保证了系统的可扩展性和可维护性。api 模块则封装了系统调用的实现，为用户空间程序提供了与内核交互的接口，在其中又依照系统调用的功能与相互的依赖关系将其划分为任务、内存、文件与网络四个组件，分别处理对应的相关系统调用实现。

具体分析每个模块，其功能罗列如下：

- 用户应用加载：负责将用户应用程序加载到用户地址空间，并设置入口点和栈指针。
- 系统调用接口：负责注册系统调用处理程序，分发处理用户程序发起的系统调用，如 sys_exit、sys_mmap、sys_brk 等。
- 内存管理：处理内存相关系统调用，调用 core 中内存相关处理函数实现用户地址空间的创建、映射、解除映射和权限管理，以及用户堆和栈的分配等。
- 任务管理：处理任务相关系统调用，调用 core 中任务相关处理函数实现用户任务的创建、调度和管理，包括任务的命名空间、时间统计和子任务管理。
- 文件系统：处理文件相关系统调用，调用 core 中文件相关处理函数实现文件的读取、写入和权限管理，以及文件系统的挂载和卸载等。
- 网络模块：处理网络相关系统调用。
- 用户程序入口点：负责读取应用程序，调用 run_user_app 函数运行用户应用程序，并记录每个用户任务的退出码。

在架构设计上，Starry-Next 框架注重组件的独立性与解耦和性，使得每个组

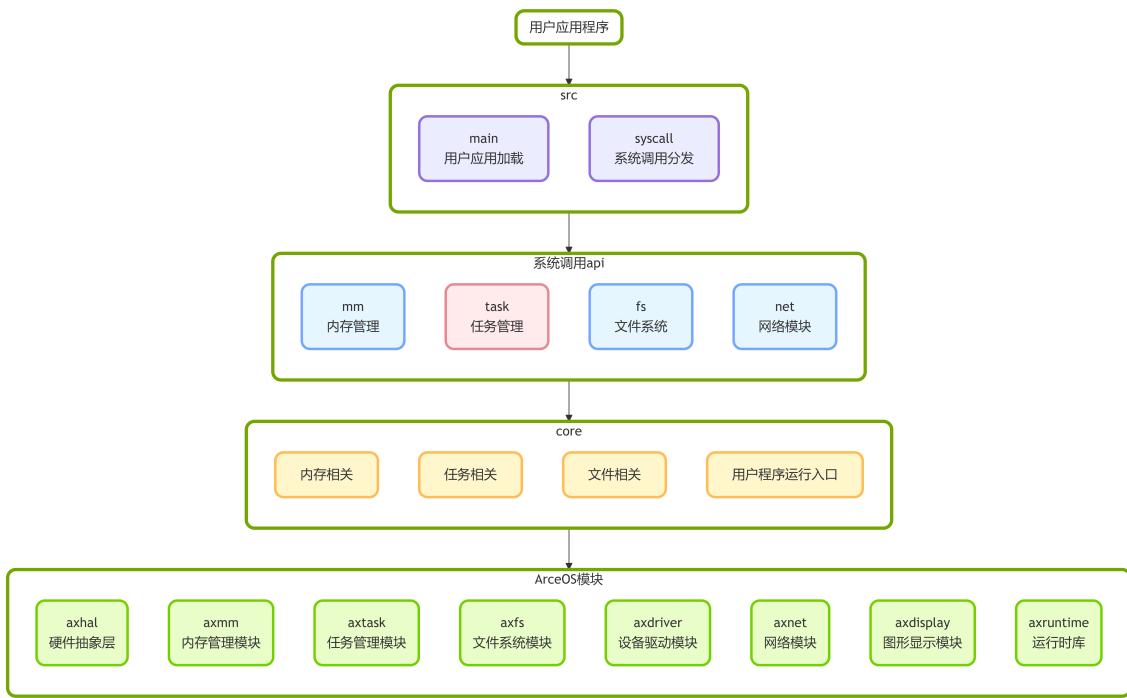


图 3.1 Starry-Next 整体框架结构图

件可以独立开发、测试和升级，而尽量减少对其他组件正常开发与运行的影响，这种设计方式大大提高了框架的灵活性与可扩展性，也使得本课题所属大方向的多名成员能以相对独立的方式对组件化宏内核操作系统进行协同式组件开发。

在实际运行上，Starry-Next 框架基本实现了第二章所讲述的基于 ArceOS 实现基础宏内核操作系统的理论方法：

在系统启动时，Starry-Next 会进行初始化操作，包括加载配置文件、初始化各个模块和组件、建立必要的连接等。

在系统初始化完成后，Starry-Next 框架进入运行阶段，等待用户的请求。当接收到用户的请求时，系统会按照以下步骤进行处理：

1. 用户接口层（src）接收请求：用户接口层接收用户的输入，并将请求传递给系统调用层。
2. 系统调用层（api）处理请求：系统调用层根据请求的类型和参数，调用相应的系统调用处理请求，在这个过程中会进入处理核心层相应的模块来实现对各个系统调用的支持。
3. 处理核心层（core）操作请求：处理核心层根据系统调用层的请求，进入底层调用 ArceOS 相关的组件完成对数据等资源的操作，并将操作结果返回给系统调用层。

4. 系统调用层返回结果：系统调用层根据处理核心层的操作结果，生成响应结果，并将其返回给用户接口层。
5. 用户接口层展示结果：用户接口层将程序运行结果展示给用户。

当应用程序全部运行完毕后，Starry-Next 会进行终止操作，释放相关资源，停止网络通信、日志记录、配置管理等基础设施服务，最终成功结束整个系统的运行。

3.2 与基座代码的接口分析

Starry-Next 在多个方面依赖于 ArceOS 的功能和库，如 axfeat、arceos_posix_api、axconfig 等。这些依赖库为 Starry-Next 提供了硬件抽象层、文件系统、任务管理等基础的功能支持。此外，Starry-Next 更是依赖 ArceOS 提供的接口来实现系统的核功能。例如，在任务管理模块中，ArceOS 的 axtask 组件提供了任务创建、调度和销毁等底层任务管理与调度的接口，axprocess 模块则提供了进程、线程、线程组、会话等高级抽象的管理接口，Starry-Next 可以直接调用这些接口来实现任务管理相关的系统调用，从而实现任务管理的完备功能。这种依赖关系使得操作系统开发者不再需要深入到宏内核的底层，去探究各种数据结构的组成、各种硬件支持的搭建逻辑等要求较高的内容，而只需要将精力集中在上层抽象出的系统调用层面，为支持宏内核上的更多功能而实现、完善各种系统调用。这种开发模式极大地降低了定制化操作系统的开发门槛，也提高了组件化宏内核的实现效率。

本节将聚焦于分析 Starry-Next 与 ArceOS 的交互基础和接口情况，探讨这些接口如何相互依赖与运行以支持框架所实现的宏内核操作系统。

3.2.1 任务调度接口

Starry-Next 通过调用 ArceOS 提供的任务创建接口来创建新的内核任务调度实体。在 ArceOS 中提供了 spawn_task 函数用于创建任务，该函数接收任务的入口函数、参数等信息，并返回一个任务的句柄。Starry-Next 在运行用户程序前会先调用该接口来生成运行单个用户程序的进程。

对于任务的调度，ArceOS 也在 axtask 组件中提供相关接口，核心接口可见表 3.1。比如，ArceOS 提供 yield_nowt 函数、sleep 函数和 exit 函数分别用于让任务主动放弃 CPU 时间，切换到另一个就绪的任务、让任务休眠指定的时长以及退出当前任务，此外 ArceOS 还提供了 set_priority 函数用于设置当前任务的优先级。Starry-Next 可以调用这些接口来对任务进行动态调整与调度，从而确保用户任务能够正确高效地执行，并合理分配系统资源。

表 3.1 任务底层调度接口函数列表

接口函数	功能	使用场景
current	获取当前正在执行的任务	直接获取已初始化的任务
spawn_task	将一个 TaskInner 类任务添加到运行队列中	将已创建的任务加入到调度系统
set_priority	设置当前任务的优先级	允许用户动态调整任务优先级
yield_now	当前任务主动放弃 CPU 时间	避免长时间占用 CPU
sleep	当前任务休眠指定时长	让任务暂停执行一段时间
exit	当前任务退出执行	让任务正常结束执行

3.2.2 多任务管理接口

在创建了基本内核调度任务后, Starry-Next 还需通过调用 ArceOS 提供的进程管理等接口来创建和管理用户进程与线程。在 ArceOS 中 axprocess 组件提供了一套完整的接口用于进程、线程、进程组及会话的生命周期管理, 这些接口构成了 Starry-Next 多任务环境的基础, 核心接口如表 3.2 所示。

表 3.2 多任务管理接口函数列表

接口函数	功能	使用场景
init_proc	获取系统初始化进程	系统启动后获取根进程
Process::new_init	创建新的进程实例	系统初始化阶段创建根进程
Process::fork	创建子进程并复制资源	进程需要派生新进程时使用
Process::new_thread	在进程内创建新线程	实现多线程执行机制
Process::group	获取进程所属进程组	用于访问进程组信息
Process::exit	终止进程并标记为僵尸进程	进程正常结束或异常退出时使用
Process::free	释放僵尸进程资源	父进程回收子进程资源时使用
Thread::process	获取线程所属进程	访问当前线程的进程上下文
Thread::exit	退出线程	线程完成任务或需要提前退出时使用
Process::create_group	创建新的进程组	实现作业控制和信号分发
ProcessGroup::session	获取进程组所属会话	用于访问会话信息
ProcessGroup::processes	获取进程组内所有进程	实现进程组级别操作
Process::create_session	创建新的会话	管理登录会话和控制终端
Session::process_groups	获取会话内所有进程组	实现会话级别操作

3.2.3 内存管理接口

在 ArceOS 的 axmm 组件中，为 Starry-Next 实现了一系列内存管理接口，核心接口可见表 3.3。

表 3.3 内存管理接口函数列表

接口函数	功能	使用场景
new_kernel_aspace	创建新的内核地址空间	为内核提供独立的地址空间
kernel_aspace	返回全局唯一内核地址空间的引用	获取内核地址空间
kernel_page_table_root	返回内核页表的根物理地址	设置和管理内核页表
copy_mappings_from	从另一地址空间复制页表映射	如用于共享内存映射
clear_mappings	清除给定地址范围内的页表映射	清除不再需要的映射
handle_page_fault	处理给定地址的页错误	在发生页错误时尝试解决
clone_or_err	在新页表中映射内存并复制用户数据	克隆一个地址空间

3.2.4 文件系统接口

在 ArceOS 的 axfs 组件中也为 Starry-Next 实现了一系列文件系统相关的接口，核心接口可见表 3.4。

表 3.4 文件系统接口函数列表

接口函数	功能	使用场景
create_dir	在指定路径创建新的空目录	动态创建目录结构
remove_dir	删除指定的空目录	清理不再需要的空目录
current_dir	返回当前工作目录的路径字符串	获取当前工作目录位置
read	将指定文件内容读取到一个字节向量中	一次性读取文件所有内容
write	将一个字节切片的内容写到指定文件中	向文件中写入数据
remove_file	从文件系统中删除指定的文件	清理不再需要的文件
metadata	查询指定路径下的元数据	获取文件或目录的属性信息
rename	将文件或目录命名为新的名称	重命名文件或目录

任务管理、内存管理、文件系统等模块之间虽基本保持独立，但也存在着相互依赖的关系。例如，在任务创建过程中需要为任务分配内存，这就依赖于内存管理接口；在任务执行过程中，如果需要更多的内存，也会再次调用内存分配接口；

当任务销毁时，依然需要调用内存回收接口释放占用的内存。同时，任务在执行过程中可能需要进行文件读写操作，如读取配置文件、保存数据等，这又依赖于文件系统接口。在 Starry-Next 所构建的宏内核中，这些接口相互协作，形成一个有机的整体。当系统接收到用户的请求时，任务管理接口负责创建和调度任务，内存管理接口为任务提供内存支持，文件系统接口为任务提供文件操作的能力，网络等接口则为任务提供额外的支持。各个接口之间通过函数调用和数据传递进行交互，确保宏内核能够高效、稳定地运行。

3.3 Starry-Next 中的任务管理模块

任务管理模块作为操作系统的核心组成部分，负责进程和线程的创建、调度、销毁以及同步通信等操作，对系统的性能和稳定性起着关键作用。本节将着重描述 tarry-Next 框架中的任务管理模块，分析其功能与运行机制。

任务管理模块的具体功能包括：

1. 任务创建：支持通过 `sys_clone` 系统调用创建新的任务，新任务可以继承父任务的地址空间和资源。
2. 任务底层调度：提供基础任务控制块和调度原语，实现任务状态管理、上下文切换和同步机制，确保系统资源的合理利用。
3. 任务退出：当任务执行完毕或出现异常时，负责销毁任务并释放其占用的资源。
4. 任务等待：支持通过 `sys_wait4` 系统调用等待子任务的结束，并获取其退出状态。
5. 进程、线程等抽象：基于底层任务构建进程、线程、进程组和会话抽象，提供资源隔离和共享机制。

为了实现上述功能，任务管理模块使用了多层次的数据结构设计来管理任务信息，在 axtask 模块提供的核心任务控制结构 `TaskInner`、axprocess 模块提供的线程 `Thread`、进程 `Process`、进程组 `ProcessGroup` 及会话 `Session` 的数据结构之上，维护了任务扩展数据结构 `TaskExt`、线程扩展数据结构 `ThreadData` 及进程扩展数据结构 `ProcessData`。

ArceOS 层的数据结构提供了各级任务的核心信息，其中 `TaskInner` 包含了内核任务的核心信息，如任务的入口函数、内核栈大小、上下文切换信息等；`Thread` 包含了线程所属 ID、所属进程及线程私有数据，同时记录了线程关联的底层任务信息，通过该字段与底层调度实体绑定；`Process` 包含了进程 ID、父进程 ID、子进程列表、所属进程组、线程列表与自己的进程数据；`ProcessGroup` 记录进程组 ID、

进程组领导者信息、进程列表及所属会话；Session 记录会话 ID、会话领导者信息、进程组列表及所属控制终端。

Starry-Next 扩展了上述数据结构的资源与内存信息，其中 TaskExt 包含了任务的扩展信息，如线程引用与时间统计等；而 ThreadData 用于存储线程扩展信息，包括线程退出时清理子线程 ID 的地址以及线程级信号处理管理器，线程 ID 及；ProcessData 则维护了进程的扩展资源，包括可执行文件路径、虚拟内存地址空间、资源命名空间、用户堆边界、子进程退出等待队列及进程级信号处理管理器。这些数据结构通过 THREAD_TABLE、PROCESS_TABLE、PROCESS_GROUP_TABLE 和 SESSION_TABLE 四个全局表进行索引和管理，形成了完整的任务层次体系。基于此，系统能够实现任务创建、资源隔离、信号处理、时间统计等核心功能，并通过系统调用接口暴露给用户空间。

任务创建可以通过调用 axtask 模块的 spawn_task 直接创建新的任务，也可以通过实现 sys_clone 系统调用来创建子任务，实现对继承父任务的地址空间和资源，具体而言，需要先解析 clone 标志和参数，如栈地址、线程局部存储等；接着创建新的 TaskInner 实例，并初始化其入口函数和内核栈；之后需要复制父任务的地址空间，并为新任务分配新的页表；再从父任务的陷阱帧中复制用户空间上下文，并根据需要修改栈地址和指令指针；最后便可基于 TaskInner 实例创建新的 TaskExt 实例，将新任务添加到父任务的子任务列表中，并启动新任务。

任务的调度可以通过 sys_sched_yield 系统调用实现，在其中调用 axtask 模块的 yield_now 即可对任务进行切换。

任务的退出可以通过 sys_exit 系统调用实现，在其中调用 axtask 模块的 exit 函数即可退出任务，对于线程、进程、进程组、会话则需要在先逐级进行判断，需要退出时调用相应的退出函数进行退出并回收对应资源，最后再进行底层内核任务的终止和资源回收。

任务的等待则可以通过 sys_wait4 系统调用实现，先解析等待标志和参数，如进程 ID、退出状态指针等以找到符合的子任务，如果子任务已经退出，则获取其退出状态，并将其从子任务列表中移除；如果子任务仍在运行，则根据等待标志决定是否阻塞当前任务。

要成功运行 Starry-Next 的任务管理模块，需要完成以下步骤：

1. 配置开发环境：根据 Cargo.toml 文件中的依赖信息，安装所需的库和工具。
2. 编译代码：使用 cargo build 命令编译项目代码。
3. 运行测试用例：通过设置 AX_TESTCASES_LIST 环境变量指定要运行的测试用例，然后使用 cargo run 命令运行项目。

通过上述步骤，即可运行一个 Starry-Next 中简易的任务管理模块来支持基础的用户程序，而想要支持更多更复杂的应用程序，如多线程并发的应用程序，则需要对目前 Starry-Next 中的任务管理模块进行进一步的扩展，修复和完善更多任务相关的系统调用。

第 4 章 任务管理组件设计与实现

本文的主要工作即对 Starry-Next 框架中的任务管理模块进行了进一步的扩展，支持更多 Linux 中的任务相关系统调用，从而扩展并完善宏内核的功能，使之能运行更复杂的应用程序。本章将对任务管理组件的进一步设计与实现进行展开描述，并给出在实际 Starry-Next 框架中实现相关系统调用的实现细节。

4.1 开发环境与工具

如第 2、3 章所述，本课题所研究的任务管理组件及最终所实现的宏内核操作系统基于 Rust 语言开发，其内存安全、零成本抽象以及灵活条件编译等特性，为操作系统内核开发提供了坚实保障和极大便利。此外，在开发过程中，本项目选用 Cargo 作为项目构建和包管理工具，它能够高效地处理依赖关系，提升组件化操作系统的开发效率。例如，在引入如 axhal、axruntime 等相关 ArceOS 模块时，只需在 Cargo.toml 文件中声明依赖，Cargo 便能自动下载并管理这些依赖项，而免去了人工管理不同模块之间依赖的成本。

在开发工具的选择上，本课题使用 Visual Studio Code（VS Code）搭配 Rust 相关插件来实现，其丰富的代码智能提示、语法检查以及调试功能，极大地便利了代码的编写与调试工作。

整个课题的开发环境基于 Ubuntu24.04 系统完成，项目支持 Riscv64、x86-64、aarch64、Loongarch64 四种架构，搭建在支持多种架构的 QEMU 模拟器上，通过 QEMU 的 pc-q35（x86_64）和 virt（riscv64/aarch64/loongarch64）平台，模拟不同硬件环境，为任务管理组件在多种架构下的测试提供了可能。同时本项目使用 Makefile 来管理不同架构的运行或测试指令，在实验环境的搭建过程中，首先需要安装 QEMU 模拟器，随后根据不同架构需求，在 make 指令中加入相应的启动参数。以 riscv64 架构为例，在启动 QEMU 时，使用类似“make AX_TESTCASE=oscomp ARCH=riscv64 EXTRA_CONFIG=../configs/riscv64.toml BLK=y NET=y SMP=4 FEATURES=fp_simd,lwext4_rs LOG=info run”的命令，这将加载编译好的 ArceOS 内核镜像，在系统中对 oscomp 测试集合进行运行，由此便能搭建起可供测试的实验环境。

4.2 模块总体设计思路

4.2.1 任务数据结构设计

在设计任务管理组件时，首先需要对任务管理相关的数据结构进行设计，其中最为关键的即任务控制块（Task Control Block, TCB）的设计。任务控制块作为描述任务的核心数据结构，包含了任务所需的全部关键信息，下面列出了本课题所设计的任务控制块（TaskExt）内容：【这里再想想怎么改】

- proc_id: usize 类型，
- parent_id: AtomicU64，
- children: Mutex<Vec<AxTaskRef>>，
- clear_child_tid: AtomicU64，
- uctx: UspaceContext，
- aspace: Arc<Mutex<AddrSpace>>，
- ns: AxNamespace，
- time: UnsafeCell<TimeStat>，
- heap_bottom: AtomicU64，
- heap_top: AtomicU64，
- futex_table: Arc<Mutex<FutexTable>>。

其中 proc_id 作为任务的唯一表示，用于记录进程 ID，在系统内的任何操作，如任务调度、资源分配、状态查询等，都需要依赖该标识来准确定位和区分不同任务。而 parent_id 字段和 children 字段则用于维护父子任务的关系，parent_id 记录了父任务的 ID，当父任务终止时，系统能根据 parent_id 对其所有子任务进行统一管理和清理；children 则采用 Mutex<Vec<AxTaskRef>> 的结构维护了一个子任务列表，使得父任务能够便捷地对子任务进行统一管理。如 wait_pid 函数在等待子任务结束时，即可通过遍历 children 列表来快速定位到目标子任务，查询其状态并进行相应操作。同时，Mutex 锁机制保证了在多任务并发访问或修改子任务列表时不会出现数据竞争问题，从而保障数据的一致性和操作的正确性。

在对任务本身进行管理之外，任务控制块还需要管理任务所处的用户空间及相关资源。其中 uctx（用户空间上下文）字段存储了任务在用户空间执行时的关键上下文信息，包括程序计数器（PC）、通用寄存器值等。这些信息对于任务的暂停与恢复至关重要，当任务因中断、调度等原因暂停执行时，uctx 记录的上下文能完整保存任务当时的执行状态，从而使得在恢复任务执行时，系统能够依据 uctx 中存储的上下文信息，将任务的执行环境精准还原到暂停前的状态，确保任务继续正确执行。而 aspace 字段则关联任务的虚拟内存地址空间，使用 Arc<Mutex<AddrSpace>>

的形式，实现对任务内存空间的安全、高效管理。在任务执行过程中，对内存的读写、分配与回收等操作都围绕 `aspace` 展开，这一字段能保证任务运行在自己独立的内存空间，避免不同任务间的内存冲突，同时实现内存资源的合理分配与利用。除此之外，`ns` 字段代表任务的资源命名空间，用于隔离文件描述符、当前目录等任务资源，确保每个任务都在自己独立的命名空间内管理资源。

为了支持宏内核中的多线程并发操作，任务控制块也需要管理线程相关信息。`TaskExt` 使用 `futex_table` 来记录同步信息，支持线程间同步与互斥操作，避免共享资源访问时的竞态条件，提升系统并发性能和稳定性。而由于 Linux 系统中的 `set_tid_address` 机制，还需要 `clear_child_tid` 字段来存储清除线程 TID 的地址，当线程退出时，若该地址不为空，内核将在此地址写入特定值，用于线程同步和状态通知。在多线程协作场景中，可以通过该字段实现线程退出时的状态传递和同步操作。

除了上述几类核心功能，`TaskExt` 还需要标记用户堆的始末地址，用于界定用户堆内存范围，从而支持内存边界检查、动态内存扩展收缩等操作，保障内存使用的正确性，这一部分记录在 `heap_bottom` 和 `heap_top` 字段中。而 `time` 字段则为系统提供任务在用户态和内核态的执行时间，便于进行系统性能分析，为任务调度策略优化提供数据支持。

4.2.2 任务创建、销毁、切换设计

完成任务管理数据结构的设计后，就可以开始对任务的创建、销毁、切换等操作进行设计。

在 ArceOS 基座代码的 `axtask` 模块中提供了 `spawn_task` 函数用于从底层的内核里创建新的任务并加入到运行队列中，而在框架的上层想要实现任务的创建，就需要对 ArceOS 提供的接口进行封装。使用 `TaskInner` 创建新的任务内部结构，先为任务设置好内核栈，接着配置其页表根地址、初始化命名空间，然后便可以调用 `spawn_task` 函数来完成任务的新建。

任务销毁时需要实现 `drop` 方法来清理任务用户空间的内存映射，释放资源。

任务的切换发生在需要改变当前运行任务时，如时间片耗尽或高优先级任务就绪时。在任务切换时，首先需要保存当前任务的上下文，包括 CPU 寄存器的值、程序计数器等信息到当前任务的 `TaskControlBlock` 中。然后从就绪任务队列中选取下一个要运行的任务，将其 `TaskControlBlock` 中的上下文信息恢复到 CPU 寄存器中，更新程序计数器，使新任务开始执行。

4.2.3 与其他组件的交互关系

在任务管理组件的设计中需要考虑与其他组件的交互与协作。首先是内存管理组件，任务的成功运行离不开内存的深度支持，在 TaskExt::new 函数创建任务时，需要通过 axmm 模块的 AddrSpace 向内存管理模块申请内存，用于存储 TaskControlBlock 及任务栈空间。任务终止时，TaskExt 的 drop 方法也需要借助内存模块的接口实现内存映射清理与空间回收。

与文件系统组件的交互则体现在任务的文件相关操作上。TaskExt::ns_init_new 函数在初始化任务资源命名空间时需要调用 FD_TABLE、CURRENT_DIR 等文件系统相关接口，来完成对文件描述符表、当前目录等信息的初始化。当任务需要执行文件读写操作时，也需要通过系统调用经任务管理组件转至文件系统组件处理。

4.3 模块完成思路

(如何一步步实现相关系统调用：1. 重要系统调用的介绍与分析；（往上移？）
2. 系统调用逐步实现安排；3. 如何基于 arceOS 基座代码进行系统调用的实现与接入（着重于新增的部分）。)

系统调用是用户程序与内核交互的重要接口，为了对宏内核实现更多任务管理相关的功能支持，需要在 Starry-Next 框架已有内容的基础上，基于任务管理相关数据结构与功能函数，补充更多的系统调用，表 4.1列出了本文所补充与实现的任务管理相关系统调用及功能概述，在此基础上，本文由易到难逐步实现了这些系统调用，最终能够在所实现的宏内核上运行复杂的 Linux 应用程序。

4.4 系统调用实现

本节将基于前面的论述，给出在实际 Starry-Next 框架中实现任务相关系统调用的实现细节，描述实现过程中遇到的问题、挑战并尝试给出解决方法。

【!! 加入传入和返回的参数信息 + 之后把描述再写详细一点】

【1】execve 系统调用

Execve 系统调用用于在当前进程中加载并执行一个新的程序，替换原有的进程映像。它会清除当前进程的地址空间内容，将新程序的代码和数据加载到内存，并从新程序的入口点开始执行，实现对新程序的运行。

在 Starry-Next 框架中的 thread 模块实现了对 sys_execve 的响应，从用户空间获取程序路径、参数列表和环境变量列表，接着调用 core 中 task 任务模块所实现的

表 4.1 任务管理相关系统调用列表

系统调用	实现功能
execve	在当前进程中加载并执行一个新的程序
getpid	获取当前进程的进程 ID
getppid	获取当前进程的父进程 ID
getuid	获取当前进程的用户 ID
set_tid_address	设置线程退出时内核写入特定值的地址
gettid	获取当前线程的线程 ID
wait4	使父进程等待子进程终止，并获取子进程的退出状态
exit	终止当前进程的执行
exit_group	终止调用进程及其所属的整个线程组
nanosleep	使当前进程暂停执行指定的时间（以纳秒为单位）
clone	创建一个新的进程或线程，与原进程共享部分资源
fork	创建一个新的子进程
futex	实现线程间的互斥、同步和条件变量等操作

exec 函数完成对 execve 系统调用功能的支持。exec 函数的核心实现逻辑如图 4.1 所示，首先获取当前任务，验证任务地址空间是否可进行替换操作，如检查地址空间是否被多个任务共享。接着调用用户空间结构 aspace 中的 unmap_user_areas 函数来清除当前任务的用户空间映射，刷新 TLB (Translation Lookaside Buffer) 以保证地址转换的正确性。然后通过 core 中内存模块的 load_user_app 函数加载新程序，该函数负责读取程序文件内容、分配内存并将程序数据写入内存。最后更新当前任务的名称、用户空间上下文等信息，并通过 uctx 结构中的 enter_uspace 方法进入新程序的执行环境，完成程序替换。

在实现过程中遇到过两个问题，其一是使用 current_task 的 set_name 方法时，若直接传入 name 变量，会访问不正确的地址导致出现缺页异常。经调试后发现由于 String::from 存在问题，导致传入的 name 处于不被支持访问的空间内，因此将 name 所引用的字符串内容新建到一个拥有所有权的字符串 program_name 中，再将其解引用后作为参数传入 set_name 函数，即可解决访问的问题。其二是在最初的版本中 aspace 被 lock 后并未手动释放，导致出现重复获得锁的错误。这是由于 exec 函数会手动改动执行流，执行完 enter_uspace 之后会进入用户态而不返回 exec 函数，从而导致在函数开头被 lock 的 aspace 无法被函数自动释放，因此需要在函数末 enter_uspace 之前加入 drop(aspace) 来将锁释放。

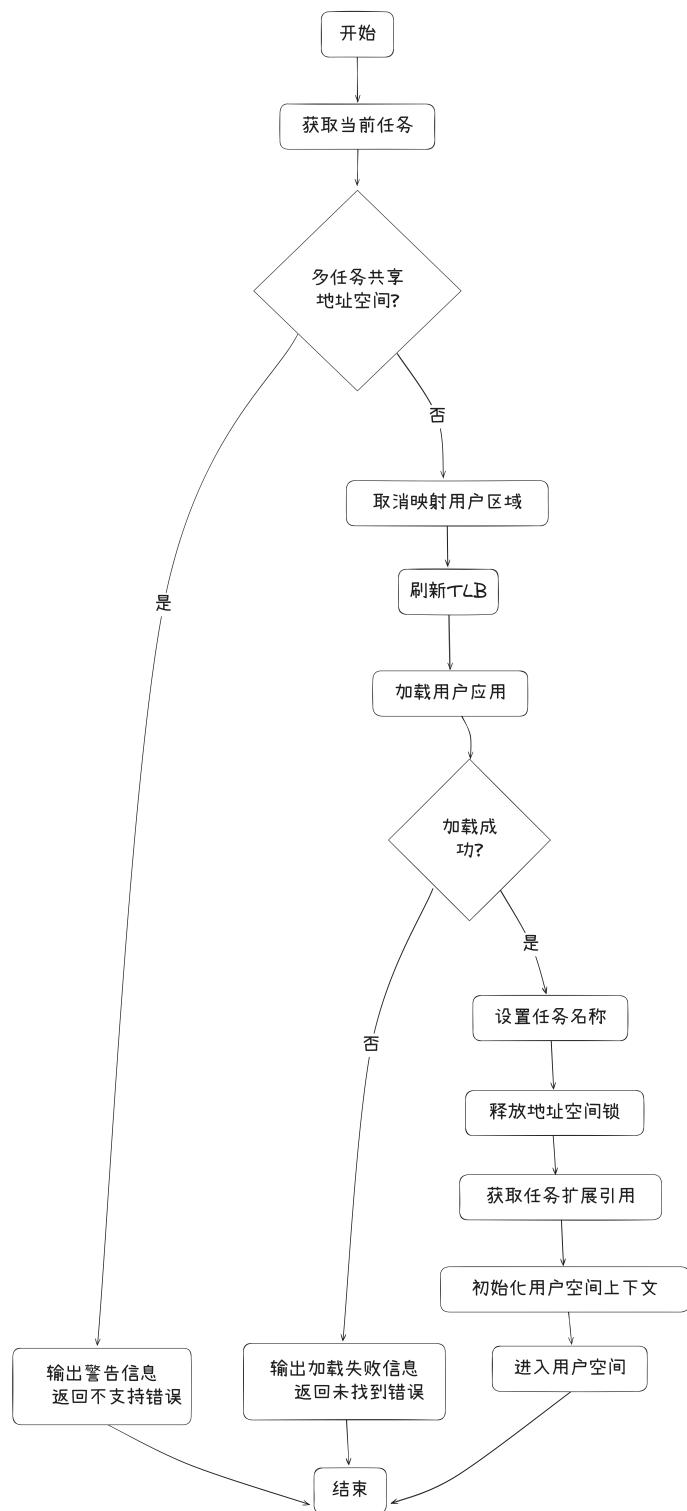


图 4.1 exec 核心实现逻辑流程图

解决了上述问题后，就完成了对本文所增加的第一个系统调用 `execve` 的支持。

【2】`getpid` 系统调用

`Getpid` 系统调用用于获取当前进程的进程 ID，在 Starry-Next 中需通过 `axtask` 模块的 `current` 函数获取当前任务控制块，从任务控制块中即可得到进程 ID 信息，即 `proc_id`。

【3】`getppid` 系统调用

`Getppid` 系统调用用于获取当前进程的父进程 ID，与 `getpid` 类似，需要先当前任务控制块，之后使用任务控制块的 `get_parent` 方法即可获取当前任务的父进程 ID。

【4】`getuid` 系统调用

`Getuid` 系统调用用于获取当前进程的用户 ID。由于本项目尚未实现对多用户的 support，因此只对该系统调用进行了伪实现，默认所有进程的用户 ID 均为 0。

【5】`set_tid_address` 系统调用

`Set_tid_address` 系统调用用于设置一个地址，当线程退出时，内核会将用于线程同步和状态通知的特定值写入该地址。该系统调用主要用于多线程程序中，方便主线程或其他线程获取线程退出的状态信息。

在 Starry-Next 中实现了 `sys_set_tid_address` 函数来对其进行支持，首先获取当前任务控制块，调用任务控制块中的 `set_clear_child_tid` 方法来将传入的地址设置为当前线程对应的 `clear_child_tid` 字段，并返回当前线程的 ID 即可。

【6】`gettid` 系统调用

`Gettid` 系统调用用于获取当前线程的线程 ID，在多线程程序中，线程 ID 用于区分不同线程，以进行线程间的同步、调度等操作。

由于 ArceOS 中仅实现单进程，因此在 `axtask` 模块中提供的任务实际即为线程，在 `TaskInner` 中记录的 `id` 即为线程 ID，在实现 `gettid` 时只需通过调用 `axtask` 模块的 `current` 方法获得当前任务的 `TaskInner`，其 `id` 字段即为所需线程 ID。

【7】`wait4` 系统调用

`Wait4` 系统调用用于使父进程等待子进程终止，并获取子进程的退出状态。父进程通过该调用可以回收子进程占用的资源，避免资源泄漏，同时了解子进程的执行结果。

在 `wait4` 系统调用中，当前任务（父任务）通过遍历 `children` 列表，查找指定 `pid` 的子任务。若子任务处于运行状态，则调用 `axtask` 的 `yield_now` 方法让出 CPU 资源，继续等待；若子任务已退出，获取其退出码，从 `children` 列表中移除该子任务，并根据需求将退出码写入指定地址 `exit_code_ptr`。

【8】exit 系统调用

Exit 系统调用用于终止当前进程的执行，释放进程占用的资源，如内存、文件描述符等，并向父进程返回退出状态码。

在实验框架提供的 TaskExt 的 drop 方法中已经包含了部分任务资源清理的逻辑，如清理内存映射。而 exit 系统调用在此基础上，进一步完善资源释放操作，如关闭打开的文件（通过文件系统组件接口）、释放锁资源等。然后设置任务的退出码，将任务状态设置为终止态，通知父任务（如在 wait_pid 函数中等待该任务的父任务），最后调用系统底层的资源回收函数完成进程的终止。

【9】exit_group 系统调用

Exit_group 系统调用用于终止调用进程及其所属的整个线程组。所有属于该线程组的线程都会被终止，系统会回收线程组占用的所有资源，包括内存、文件描述符等。

在任务管理组件中，实现 exit_group 需要遍历当前线程组的所有线程任务，对每个线程任务执行 exit 操作来清理任务相关资源、更新任务状态为终止态，并通知相关等待该任务的其他任务（如父任务）。在确保所有线程任务资源回收完毕后，线程组彻底退出。

【10】nanosleep 系统调用

Nanosleep 系统调用用于使当前进程暂停执行指定的时间（以纳秒为单位），直到指定的时间间隔过去或者被信号中断。该系统调用常用于实现精确的时间延迟或等待特定事件发生。

在本实现中，ArceOS 的 arceos_poisx_api 封装了时钟相关的接口 sys_nanosleep，只需调用即可完成对该系统调用基本的支持。其实现如图 4.2 所示，首先对输入参数进行严格校验，检查 req 指针是否为空，同时验证 req 指向的 timespec 结构体中 tv_nsec 字段的值是否在有效范围内。若参数不满足要求，函数直接返回 LinuxError 的 EINVAL 错误码，表明输入参数无效。在确认参数合法后，函数将 timespec 结构体转换为 Duration 类型，以获取精确的睡眠时长 dur，并记录当前时间 now。根据系统是否开启 multitask 特性，sys_nanosleep 采用不同的睡眠实现方式。若开启该特性，则调用 axtask 的 sleep 函数，借助任务调度机制，将当前任务挂起指定时长，使 CPU 能够调度其他任务执行，有效提高了系统的并发性能。若未开启 multitask 特性，则调用 axhal 模块的 busy_wait 函数，通过忙等待的方式实现睡眠。不过忙等待会持续占用 CPU 资源，在长时间睡眠场景下，这种方式会导致 CPU 利用率下降，影响系统整体性能。睡眠结束后，函数再次记录时间 after，并计算实际睡眠时长 actual。若睡眠时长 dur 减去实际睡眠时长 actual 存在剩余时间，且 rem 指针不

为空，函数会将剩余时间存储到 rem 指向的 timespec 结构体中。

【11】sched_yield 系统调用

Sched_yield 系统调用用于让当前正在运行的线程或进程主动放弃 CPU 的使用权，将 CPU 时间片让给其他具有相同或更高优先级的线程或进程。当调用该系统调用时，当前线程或进程会从运行状态转换为就绪状态，重新进入调度队列等待下一次被调度执行。这一操作不会导致线程或进程被阻塞，它只是暂时让出 CPU 资源，以便其他线程或进程有机会运行。

在 ArceOS 中提供了用于任务切换的接口 yield_now，而 arceos_posix_api 又将其封装为了 sys_sched_yield 以供上层应用调用，因此只需在系统调用分发的位置调用该接口，即可实现对 sched_yield 的支持。

【12】clone 系统调用

Clone 系统调用用于创建一个新的进程或线程（取决于传入的参数），新创建的进程或线程与原进程共享部分资源，如地址空间、文件描述符等，通过参数可以灵活控制资源的共享与分离情况。

在 Starry-Next 中，sys_clone 系统调用首先获取当前任务，接着调用在 core 中实现的 clone_task 函数来完成 clone 操作。clone_task 函数接收标志位、栈地址等参数，首先根据标志位解析出克隆选项，然后创建一个新的 TaskInner 实例，通过 axmm 模块提供的 clone_or_err 方法和 copy_from_kernel 方法复制原任务的地址空间，并根据参数设置新任务的用户空间上下文、初始化命名空间。最后初始化新任务的 TaskExt 实例，将新任务加入原任务的 children 列表中，完成克隆操作。

【13】fork 系统调用

Fork 系统调用用于创建一个新的子进程，子进程作为父进程的一个副本，拥有与父进程几乎相同的代码、数据和打开的文件描述符等资源。子进程与父进程从 fork 调用返回点开始各自独立执行，通过 fork 的返回值来区分父进程和子进程，父进程返回子进程的进程 ID，子进程返回 0。

由于已经实现了 clone 系统调用，其作为 fork 的高级版本，支持创建新进程或新线程，因此 fork 操作可在系统调用分发时直接转发给 clone 完成。但在实际测试操作中发现实现方式在 riscv64 和 loongarch64 下表现正常，却无法通过 x86-64 和 aarch64 的测试。经过调试分析，发现后两者在汇编层面与前两者略有不同，执行完 fork 系统调用后，wait_pid 函数在等待子进程退出后进行任务上下文切换时访问了 fs 寄存器，而 ArceOS 的基座方法并未在此前对该寄存器的设置操作，因此在实际运行时访问到空地址，引发了 page_fault。

该问题的修复需要深入到 ArceOS 层面，在硬件抽象层 axhal 内加入 tls 相关

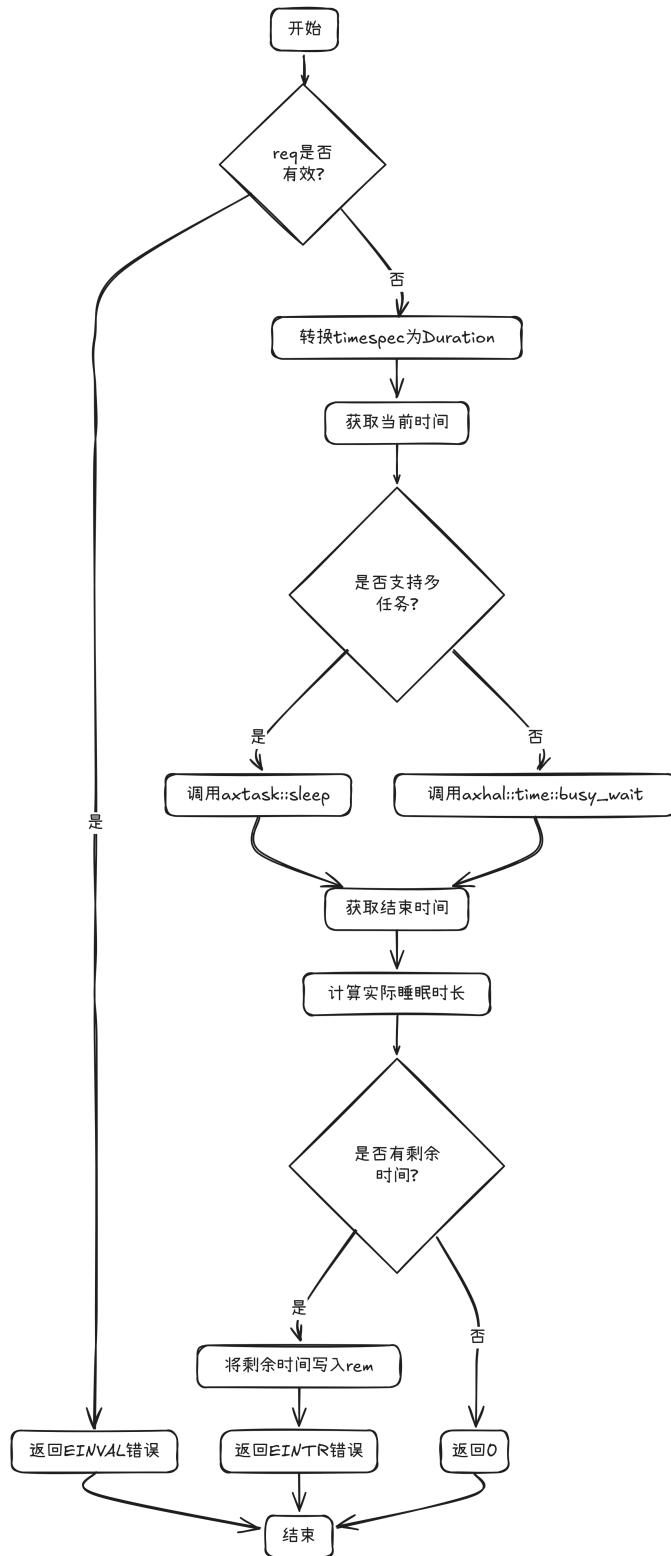


图 4.2 sys_nanosleep 核心实现逻辑流程图

处理函数。TLS 区域通常用于存储线程独有的数据，例如线程的局部变量、缓存等。所实现的 `tls` 函数用于获取当前任务的 TLS 区域的虚拟地址，它从 `axhal` 模块保存任务硬件状态的结构体 `TaskContext` 的 `fs_base` 字段中获取存储的 TLS 区域的基址值，并将其转换为 `VirtAddr` 类型返回。基址值的设置则需要 `sel_tls` 函数来完成，它接受一个 `VirtAddr` 类型的参数 `tls_area`，表示要设置的 TLS 区域的虚拟地址，然后将该地址转换为 `usize` 类型，并存储到 `TaskContext` 结构体的 `fs_base` 字段中。这两个函数与 `UspaceContext`、`TrapFrame` 等结构体以及 `enter_uspace` 等函数均存在潜在的关联，如进行任务上下文切换时，需要正确设置和使用 TLS 区域，先保存当前任务的 `fs_base`，然后将下一个任务的 `fs_base` 写入，确保线程局部存储相关的信息在任务执行和切换过程中能够正确处理。而 `tls` 和 `set_tls` 函数提供了操作 TLS 区域地址的接口，有助于在不同的任务和空间切换中维护 TLS 相关的状态。

实现上述两个函数之后，需要在 `Starry-Next` 的 `clone_task` 中使用 `new_task.ctx_mut().set_tls(axhal::arch::read_thread_pointer().into())` 来对 `fs_base` 进行设置，从而支持 x86-64 和 aarch64 中对 `fs` 寄存器的相关操作。

【14】futex 系统调用

Futex（Fast Userspace Mutex）系统调用是一种高效的用户态同步机制，用于实现线程间的互斥、同步和条件变量等操作。它结合了用户态和内核态的优势，在用户态快速尝试获取锁，减少内核态的开销；当竞争激烈时，才进入内核态进行等待和唤醒操作，提高系统的并发性能。

在 `TaskExt` 结构体中实现了 `futex_table` 字段用于管理 Futex 相关操作，其类型为 `Arc<Mutex<FutexTable>>`，其中 `FutexTable` 使用 `BTreeMap<usize, Vec<usize>>` 实现。Futex 系统调用处理函数接收操作类型（如等待、唤醒）、futex 地址等参数后，首先根据 futex 地址计算出对应的键值，在 `futex_table` 中查找相关信息。对于等待操作，若 futex 已被占用，则将当前任务加入等待队列，通过 `futex_table` 记录等待任务的 ID，并将任务状态设置为阻塞态，放入等待队列中等待唤醒；对于唤醒操作，则从 `futex_table` 中获取等待任务列表，唤醒相应的任务，将其状态改为就绪态，重新加入调度队列。在整个过程中，通过 `mutex` 机制保证对 `futex_table` 的线程安全访问，从而确保 futex 操作的正确性和一致性。

第 5 章 任务管理组件测试与分析

在实现了对操作系统宏内核下任务管理组件的支持与拓展后，本研究对其功能正确性与性能表现进行了完备的测试。本章将介绍所使用的测试用例，并分析其性能与功能的测试结果。

5.1 测试用例

本研究采用操作系统大赛测例作为核心评价指标，该测例集合提供一系列丰富且多样化的测试样例，涵盖了操作系统任务管理的多个关键维度，具有权威性与全面性。从功能层面来看，包含任务创建、销毁、切换、调度等基础功能测试，以及进程间通信、资源分配与回收等复杂功能测试。下面是部分测例及其在任务管理方面关注的重点：

- **Basic** 基本测试样例：手动编写的简单测例，主要通过系统调用访问内核达到测试效果。测试基于部分基础的 Linux syscalls，验证任务管理组件对基本系统调用的实现情况，确保能够提供正确可靠的服务。
- **Libctest** 测试样例：用于测试 C 标准库的功能和兼容性，通过执行各种静态和动态测试用例，验证内核与 C 标准库之间的兼容性和稳定性，这也与任务管理组件对不同类型任务的管理能力相关。
- **BusyBox** 测试样例：BusyBox 是一个轻量级软件包，将多种 Unix 工具合并到一个可执行文件中。该测例通过执行一系列 BusyBox 命令，判断待测内核对 BusyBox 功能的支持情况，间接反映任务管理组件在资源管理和任务调度方面的能力。
- **Lua** 测试用例：运行 Lua 脚本文件，测试 Lua 解释器的执行情况，评估任务管理组件对脚本执行任务的调度和资源分配能力。
- **UnixBench** 测试样例：评估操作系统的综合性能，涵盖 CPU 运算、内存管理、进程调度、文件系统等核心功能。每个测试点对应不同场景，可全面测试任务管理组件在各种情况下的性能表现。

本项目支持本地和线上两种测试手段，在本地配置好依赖后可以通过 `makefile` 相关指令切换不同脚本运行不同架构下的不同测试，此外操作系统大赛也提供了线上的 CI/CD 评测，只需提供开发仓库即可直接获得所有测例的测试结果。

在测试过程中，对代码的调试主要依赖两种手段，一是在代码相关位置加入 `info!` 信息，并通过修改运行指令中的 LOG 等级，使之输出相应的调试信息来辅助

判断、定位问题；二是使用 Linux 下的系统调用监控工具 strace 来输出测例的系统调用情况，捕捉调用的名称、传递的参数以及返回值等详细信息，将其与调试输出相比较，从而进一步缩小问题范围。图 5.1 展示了对 Libctest 中第一个测例 argv 进行 strace 分析的结果，其输出有效地帮助了测试过程中对 gettid 和 fork 系统调用的问题调试。在定位到问题的大致内容后，也可以通过参考 DragonOS、ByteOS 等已有实现，来进行进一步的比较，从而更准确地解决问题。

```
yyy@yydelelenovo:~/testsuits-for-oskernel/temp/mnt/musl$ sudo strace -f -e trace='read,write,readv,writev,lseek,dup' ./runtest.exe entry-static.exe argv
execve("./runtest.exe", ["./runtest.exe", "entry-static.exe", "argv"], 0x7ffd77466228 /* 15 vars */) = 0
arch_prctl(ARCH_SET_FS, 0x60a8d8)          = 0
set_tid_address(0x60aa10)                 = 287685
rt_sigprocmask(SIG_BLOCK, [CHLD], NULL, 8) = 0
rt_sigprocmask(SIG_UNBLOCK, [RT_1 RT_2], NULL, 8) = 0
rt_sigtimedwait([SIGCHLD], {sa_handler=0x40060d, sa_mask=[], sa_flags=SA_RESTART|SA_RESTORER|SA_RESTORED, sa_restorer=0x404fea}, {sa_handler=SIG_DFL, sa_mask=[], sa_flags=0}, 8) = 0
===== START entry-static.exe =====
rt_sigprocmask(SIG_BLOCK, ~[RTMIN RT_1 RT_2], [CHLD], 8) = 0
rt_sigprocmask(SIG_BLOCK, ~[], ~[KILL STOP RTMIN RT_1 RT_2], 8) = 0
fork(strace: Process 287686 attached
)                                         = 287686
[pid 287686] gettid(<unfinished ...>
[pid 287685] rt_sigprocmask(SIG_SETMASK, ~[KILL STOP RTMIN RT_1 RT_2], <unfinished ...>
[pid 287685] <... gettid resumed> = 287686
[pid 287685] <... rt_sigprocmask resumed>NULL, 8) = 0
[pid 287686] rt_sigprocmask(SIG_SETMASK, ~[KILL STOP RTMIN RT_1 RT_2], <unfinished ...>
[pid 287685] rt_sigprocmask(SIG_SETMASK, [CHLD], <unfinished ...>
[pid 287686] <... rt_sigprocmask resumed>NULL, 8) = 0
[pid 287685] <... rt_sigprocmask resumed>NULL, 8) = 0
[pid 287686] rt_sigprocmask(SIG_SETMASK, [CHLD], <unfinished ...>
[pid 287685] rt_sigtimedwait([CHLD], NULL, {tv_sec=5, tv_nsec=0}, 8 <unfinished ...>
[pid 287686] <... rt_sigprocmask resumed>NULL, 8) = 0
[pid 287686] prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
[pid 287686] prlimit64(0, RLIMIT_STACK, {rlim_cur=100*1024, rlim_max=100*1024}, NULL) = 0
[pid 287686] execve("entry-static.exe", ["entry-static.exe", "argv"], 0x7ffe46cc823b8 /* 15 vars */) = 0
[pid 287686] arch_prctl(ARCH_SET_FS, 0x68ab70) = 0
[pid 287686] set_tid_address(0x68ac90) = 287686
[pid 287686] exit_group(0) = ?
[pid 287686] +++ exited with 0 +++
<... rt_sigtimedwait resumed> = 17 (SIGCHLD)
wait4(287686, [{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0, NULL) = 287686
Pass!
===== END entry-static.exe =====
exit_group(1) = ?
+++ exited with 1 +++
```

图 5.1 argv 测试的 strace 结果图

5.2 自己编写简单测例

项目的开发时以对系统调用的支持为粒度进行层层推进的，而上述测例虽由易到难对相关系统调用进行了支持，却并没有手段对每个单一系统调用进行单独测试。并且在应用程序的运行过程中，很多系统调用也存在着相互依赖的关系，比如在 Libctest 的 argv 测例中，要测试 gettid 系统调用对线程 ID 获取是否正确，就首先需要创建线程并使用信号相关系统调用进行信号的交互，整个程序才能正确结束，而在实现 gettid 系统调用时，框架尚未支持信号相关系统调用，因此就需要自己编写一些简化的测例，来针对单一系统调用实现功能测试。

还是以测试 gettid 系统调用为例，图 5.2 展示了自己编写测例的大致思路，使用 pthread 编写了简单的线程创建操作，绕过信号相关的系统调用支持对 gettid 的测试。首先调用 pthread_create 函数创建一个新的子线程，函数执行后返回 0 则表示子线程创建成功，否则使用 assert 函数触发断言错误，终止程序。子线程创建成功后主线程调用 pthread_join 函数等待子线程正常执行完毕，之后调用 gettid 函数

数获取主线程的线程 ID 并打印，而子线程中也是简单调用 `gettid` 系统调用获取子线程 ID 并打印，成功后结束子线程。但在这个过程中发现此时尚未实现 `futex` 系统调用，而 `pthread` 操作需要 `futex` 的同步机制来实现线程结束后的合并支持，导致还是无法仅针对 `gettid` 进行测试。

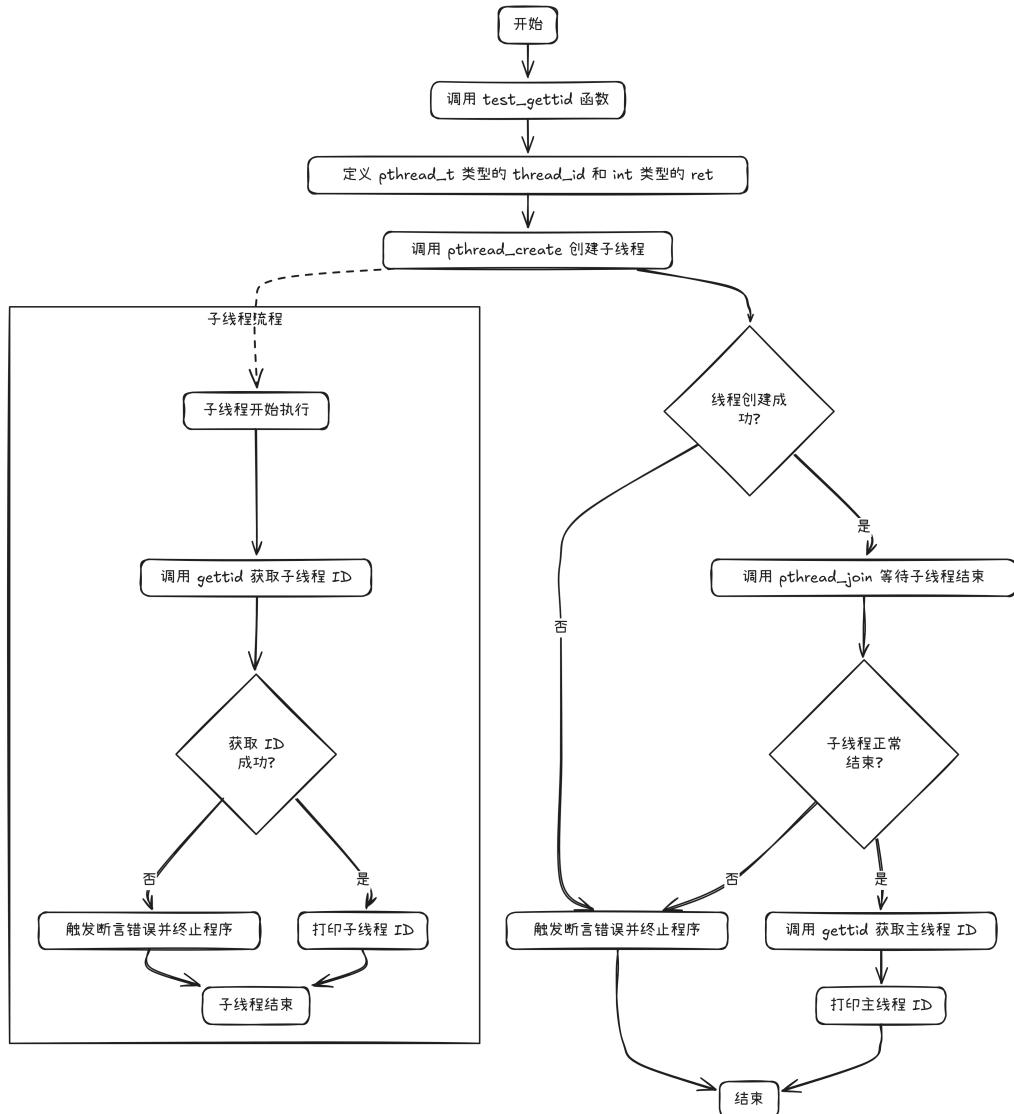


图 5.2 gettid 系统调用测例编写流程图

因此，本研究还采用了裸调用系统调用的方法来获得更简易的测试效果，基于已经实现好的 `clone` 系统调用，在测试函数中调用 `clone` 进行线程的创建，即可避开 `pthread` 相关操作，实现对线程的直接创建和线程 ID 的获取。图 5.3 展示了直接调用 `clone` 的核心操作，其中 `thread_function` 是子进程启动时执行的函数，`child_stack` 是子进程的栈空间，参数 3 则是一系列标志位 `flags`，用于指定共享行为，`func_arg` 是传递给 `thread_function` 的参数。标志位的组合决定了子进程与父进程共享哪些

资源，一些重要的标志位如下：

- **CLONE_VM**: 如果设置，子进程将与父进程共享相同的虚拟内存。
- **CLONE_FS**: 如果设置，子进程将与父进程共享文件系统信息，包括根目录和当前工作目录。
- **CLONE_FILES**: 如果设置，子进程将与父进程共享打开的文件描述符表。
- **CLONE_SIGHAND**: 如果设置，子进程将与父进程共享信号处理设置。
- **CLONE_THREAD**: 如果设置，子进程将放置在父进程的线程组中。

其中 **CLONE_VM** 是使用 `clone` 创建线程的关键，如果设置了 **CLONE_VM**，那么调用进程和子进程将运行在同一内存空间中，也就相当于实现了线程的创建。

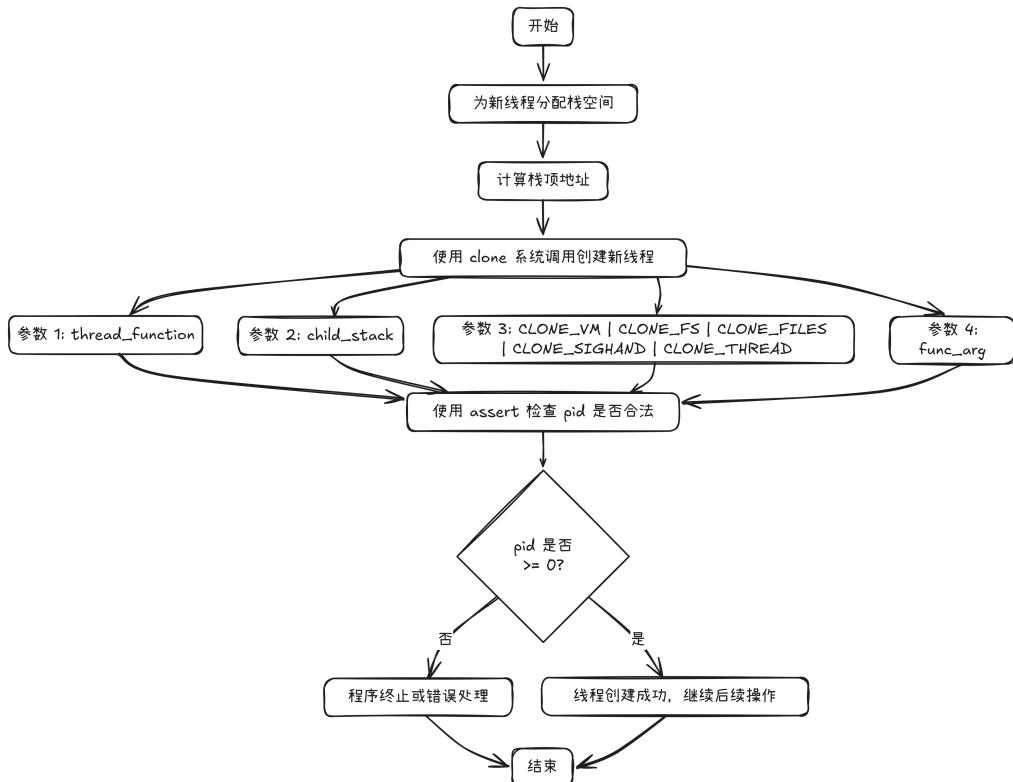


图 5.3 调用 `clone` 创建线程核心逻辑图

使用裸调用系统调用的方式，可以成功将测例的测试范围缩小到单个系统调用，从而在实现初期对其功能正确性进行验证，方便后续使用操作系统大赛的测例进行更进一步的测试。

5.3 测试结果分析

本章所描述的主要测试内容分为功能测试与性能测试，其中功能测试用于验证任务管理组件的各项基本功能是否正常，如任务的创建、调度、终止，以及资

源的分配和回收等。例如 libctest 通过一系列精心设计的测试用例，对 C 标准库的各个功能模块进行全面而细致的测试，以确保其在不同环境下的正确性和稳定性，以此判断任务管理组件是否能够正确支持 C 标准库的各项功能，从而验证内核与 C 标准库之间的兼容性和稳定性。而性能测试用于评估任务管理组件在不同负载下的性能表现，包括任务的响应时间、吞吐量、资源利用率等。例如，UnixBench 测例通过一系列性能测试指标，如算术运算速度、整数运算性能、进程间通信吞吐量等，验证 OS 的正确性和效率，全面评估任务管理组件的性能。

5.3.1 功能测试

对任务管理相关系统调用的功能测试主要依赖于 basic 和 libctest 两组测例，表 5.1 和表 5.2 分别给出了 basic 和 libctest 中的部分测例情况及其测试的系统调用，本项目已成功通过所显示的所有测例，验证了所实现系统调用的正确性。

表 5.1 basic 测例与系统调用对照表

测例名	系统调用
clone	clone
getpid	getpid
getppid	getppid
exit	exit
wait	wait4
execve	execve
waitpid	waitpid
yield	sched_yield
sleep	nanosleep

5.3.2 性能测试

表 5.2 libctest 测例与系统调用对照表

测例名	系统调用
	execve
	set_tid_address
argv	getpid
	wait4
	exit_group
	fork
daemon_failure	getpid
	getppid
pthread_cond	nanosleep
	clone
pthread_cancel_points	futex
	exit

第6章 结论

本文围绕组件化宏内核操作系统的任务管理组件展开研究，成功基于 ArceOS 基层架构和 Starry-Next 框架设计并实现了满足复杂应用需求的任务管理组件。通过重新设计任务数据结构，详细规划任务操作流程，实现了一系列关键系统调用，有效提升了宏内核操作系统对任务的管理能力，经多种测试用例验证，该组件功能正确、性能表现良好，能够稳定支持复杂 Linux 应用程序运行。

然而，本研究仍存在一定局限性。在功能方面，对于一些特殊场景下的任务管理需求，如极端实时性要求或超大规模任务并发场景，目前的组件设计还需进一步优化和扩展；在性能方面，尽管已取得较好表现，但在高并发、高负载情况下，任务调度的效率和资源利用率仍有提升空间。

未来的研究可以针对上述不足展开。一方面，通过深入研究特殊场景下的任务管理策略，进一步优化任务数据结构和调度算法，提高组件对复杂场景的适应性；另一方面，探索更高效的性能优化方法，结合新型硬件技术和算法，提升任务管理组件在高负载下的运行效率和资源利用率，推动组件化宏内核操作系统在更多领域的应用与发展。

参考文献

- [1] Fassino J, Stefani J, Lawall J L, et al. Think: A software framework for component-based operating system kernels[C/OL]//Ellis C S. Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA. USENIX, 2002: 73-86. <http://www.usenix.org/publications/library/proceedings/usenix02/fassino.html>.
- [2] 古金宇, 李浩, 夏虞斌, 等. BrickOS: 面向异构硬件资源的积木式内核[J]. 中国科学 (信息科学), 2024, 54(3): 491-513.