基于 Rust 的 RROS 实时与非实时代码分离与优化项目申请书

1.项目介绍

1.1 项目概述

本项目旨在利用Rust语言的类型系统在RROS操作系统中实现实时代码与非实时代码的分离，以提高系统的实时性能、安全性和可维护性。

1.2 技术背景

1.2.1 现有问题分析

实时任务与非实时任务的混合问题:

- 在现有的操作系统（如基于Linux内核并打了实时补丁的系统）中，实时任务和非实时任务的编程规则缺乏清晰的区分界限
- 实时任务中可能包含非实时任务的操作，导致混合调用问题
- 这种混合可能严重影响实时性保障，特别是在性能和响应时间方面
- 虽然系统能够检测到任务的实时性差，但缺乏明确的编程警告或防止混合调用的机制

1.2.2 Rust语言优势

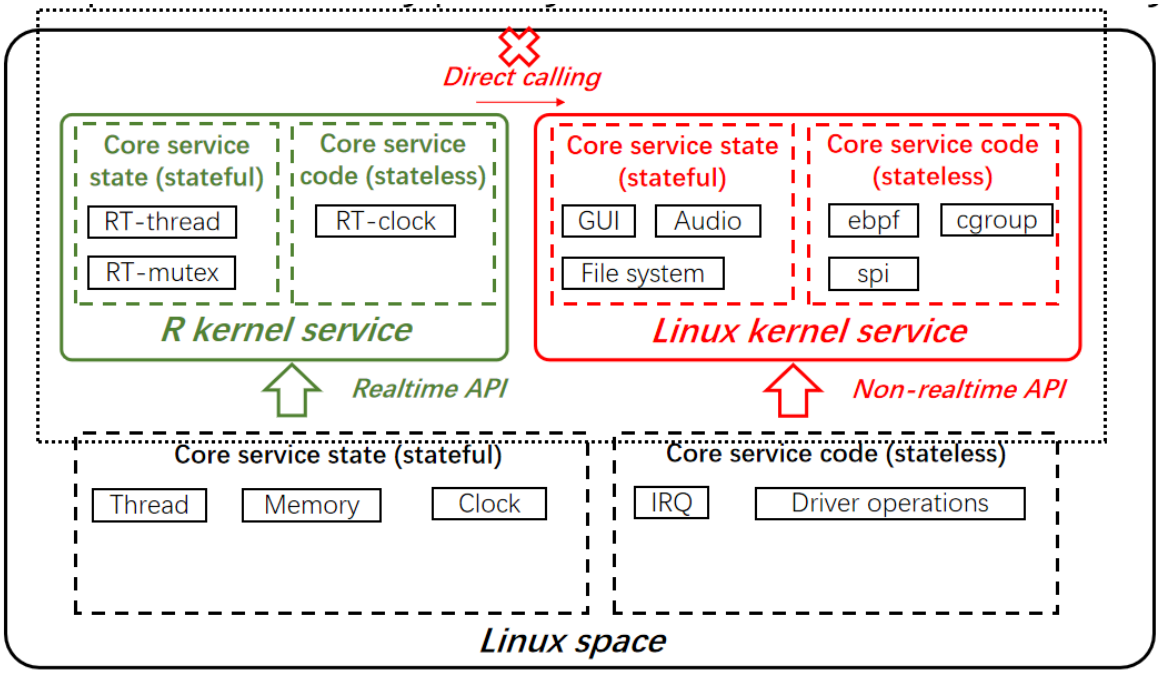
Rust语言的类型系统具备以下关键特性，使其特别适合解决实时系统的安全性和性能问题：

- **内存安全**: 通过编译时静态检查提供内存安全保障
- **并发安全**: 防止数据竞争和并发相关的错误
- **高效抽象**: 零开销抽象，不牺牲性能
- **静态分析**: 强大的编译时检查能力

1.3 解决方案

1.3.1 基于Rust类型系统的可靠性保障机制

本项目提出的核心解决方案是利用Rust类型系统在编译时确保实时与非实时代码的严格分离，从而保障系统的可靠性和实时性。我们的方案基于"Technique 1: Rust type system assures reliability"的核心理念，通过静态类型检查来防止可能导致实时性违规的代码混合调用。



如上图所示，我们的架构设计将RROS系统明确划分为两个独立的执行域：实时内核服务域（R kernel service，绿色区域）和Linux内核服务域（红色区域）。这种分离不仅体现在运行时的执行隔离上，更重要的是通过Rust类型系统在编译时就能够检测和阻止跨域的直接调用。图中的红色叉号（❌ Direct calling）明确表示了我们要解决的核心问题：禁止实时代码直接调用非实时服务，以及非实时代码对实时资源的不当访问。

1.3.2 编译时实时性冲突检测机制

我们的解决方案的核心创新在于能够在编译时检测潜在的实时性冲突。通过扩展Rust编译器的类型检查器，我们实现了对实时性违规行为的静态分析和警告生成。

For example, this code is invalid:

```
let guard = spinlock.lock();
sleep();
```

because acquiring a spinlock disables preemption, but `sleep` expects preemption to be enabled.

With `Spinlock::lock` being annotated with `#[klint::preempt_count(adjust = 1)]` and `sleep` being annotated with `#[klint::preempt_count(expect = 0)]`, the lint will generate the following error:

```
error: this call expects the preemption count to be 0
 --> example.rs:2:1
  |
2 | sleep();
  | ^^^^^^^
  |
  = note: but the possible preemption count at this point is 1..
```

上图展示了一个典型的实时性冲突检测案例。在这个例子中，代码尝试在获取spinlock之后调用sleep()函数：

```
let guard = spinlock.lock();
sleep();
```

这种代码模式在实时系统中是极其危险的，因为获取spinlock会禁用抢占（disables preemption），而sleep()函数却期望抢占能够被启用。我们的类型系统通过为SpinLock::lock标注`#[klint::preempt_count(adjust = 1)]`属性，为sleep函数标注`#[klint::preempt_count(expect = 0)]`属性，使得编译器能够自动检测这种冲突并生成明确的错误消息：

```
error: this call expects the preemption count to be 0
--> example.rs:2:1
2 | sleep();
  | ^^^^^^^
= note: but the possible preemption count at this point is 1..
```

这种编译时检查机制确保了程序员无法意外地编写出违反实时性要求的代码，从根本上提高了系统的可靠性。

1.3.3 实时性期望值管理系统

为了实现更细粒度的实时性控制，我们设计了一套完整的期望值管理系统，通过预定义的常量来表达不同操作的实时性要求。

```
const NO_ASSUMPTION: (i32, ExpectationRange) = (0, ExpectationRange::top());
const MIGHT_SLEEP: (i32, ExpectationRange) = (0, ExpectationRange::single_value(0));
const SPIN_LOCK: (i32, ExpectationRange) = (1, ExpectationRange::top());
const SPIN_UNLOCK: (i32, ExpectationRange) = (-1, ExpectationRange { lo: 1, hi: None });

const USE_SPINLOCK: (i32, ExpectationRange) = (0, ExpectationRange::top());
```

```
Some(match symbol {
    // Interfacing between libcore and panic runtime
    "rust_begin_unwind" => NO_ASSUMPTION,
    // Basic string operations depended by libcore.
    "memcmp" | "strlen" | "memchr" => NO_ASSUMPTION,

    // Compiler-builtins
    "__eqsf2" | "__gesf2" | "__lesf2" | "__nesf2" | "__unordsf2" | "__unorddf2"
    | "__ashrti3" | "__muloti4" | "__multi3" | "__ashlti3" | "__lshrti3"
    | "__udivmodti4" | "__udivti3" | "__umodti3" | "__aeabi_fcmpeq" | "__aeabi_fcmpun"
    | "__aeabi_dcmpun" | "__aeabi_uldivmod" => NO_ASSUMPTION,

    // Memory allocations glues depended by liballoc.
    // Allocation functions may sleep.
    "__rust_alloc"
    | "__rust_alloc_zeroed"
    | "__rust_realloc"
    | "__rg_alloc"
    | "__rg_alloc_zeroed"
    | "__rg_realloc" => MIGHT_SLEEP,
```

如上图所示，我们的实现包含了多个层次的期望值定义：

基础期望值常量：

- `NO_ASSUMPTION`: (i32, ExpectationRange) = (0, ExpectationRange::top()) - 表示无特殊假设
- `MIGHT_SLEEP`: (i32, ExpectationRange) = (0, ExpectationRange::single_value(0)) - 表示可能休眠的操作
- `SPIN_LOCK`: (i32, ExpectationRange) = (1, ExpectationRange::top()) - 表示自旋锁操作
- `SPIN_UNLOCK`: (i32, ExpectationRange) = (-1, ExpectationRange { lo: 1, hi: None }) - 表示自旋锁释放
- `USE_SPINLOCK`: (i32, ExpectationRange) = (0, ExpectationRange::top()) - 表示使用自旋锁

符号映射机制：
我们实现了一个复杂的符号映射系统，能够将特定的函数调用映射到相应的期望值。例如：

- 基础字符串操作：memcmp, strlen, memchr等映射到NO_ASSUMPTION
- 编译器内建函数：各种数学运算函数映射到相应的期望值
- 内存分配函数：_rust_alloc系列函数映射到特定的休眠期望值

这套系统使得我们能够为每个函数调用分配精确的实时性期望值，编译器可以通过这些期望值来验证整个调用链的实时性一致性。

2.时间安排

时间范围	项目阶段	主要任务内容
7.1-7.31	**项目准备阶段**	• 深入研究RROS操作系统架构和现有实时机制 • 阅读Rust类型系统相关文档和源码 • 理解实时与非实时代码分离的技术挑战 • 制定详细的技术实现方案
8.1-8.31	**项目开发阶段**	• 设计并实现基于Rust类型系统的代码分离机制 • 开发编译时静态检查功能模块 • 完成核心代码实现和RROS系统集成 • 进行初步的功能测试验证
9.1-9.30	**项目测试阶段**	• 执行全面的功能验证基准测试 • 性能测试与原有系统对比分析 • 系统优化和问题修复 • 完成项目文档和技术总结报告