中期报告预想

本项目为使用 Rust 编程语言组织一个**高性能**、**高并发**和注重**安全**的操作系统微内核 (及拓展部件)，基于 x86 架构 。

高性能

尽可能精简内核的规模，减少上下文切换的开销；希望通过共享内存通信的方式降低 IPC 的成本。

高并发

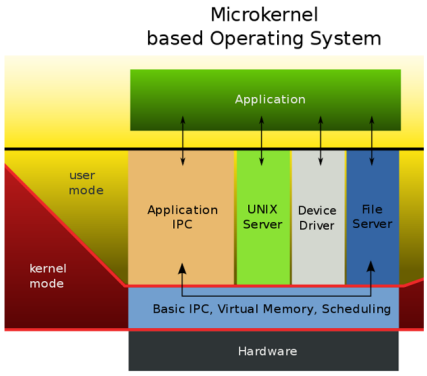
利用 Rust 语言本身提供对并发的良好支持，提升内核并发性能。

注重安全

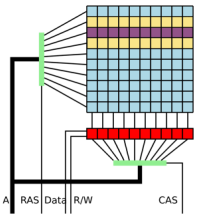
实现安全的动态内存分配和分页式内存管理，建立虚拟内存和物理内存的页映射关系，实现轻量级隔离。

- **微内核架构的不足之处：**

  IPC 成本高，上下文切换开销大。



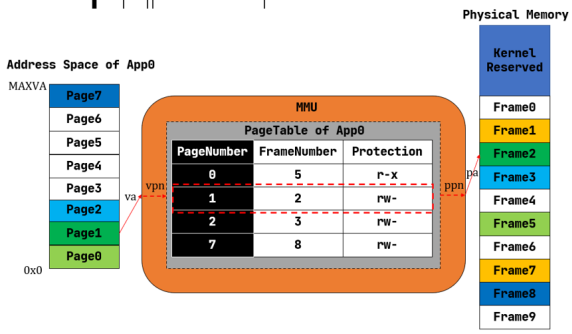
- **x-realism 所作的改进：**
  - 尽可能精简内核的规模，减少上下文切换的开销。
  - 通过共享内存通信的方式降低 IPC 的成本。
  - 基于 seL4 快速路径思想实现高效 IPC。



Row Hammer

- **分页式内存管理的不足之处：**

  当前实现中，页表处于内存分配策略的核心位置，容易产生安全性问题。



- **x-realism 所作的改进：**
  - 采取动态分页策略。即对于页表，按照内存使用情况分配页表使用权限。
  - 将内核态使用的页表与用户态使用的页表隔离，分散风险。

目前问题

- 微内核设计的 Basic IPC 和 Application IPC、File Server 的定义不清楚。
- 性能问题如何解决。

Rustpi：一个可靠的微内核操作系统

摘要：

- 在我们的系统中，微内核服务器之间的隔离是通过Rust语言实现的，而不是昂贵的硬件机制

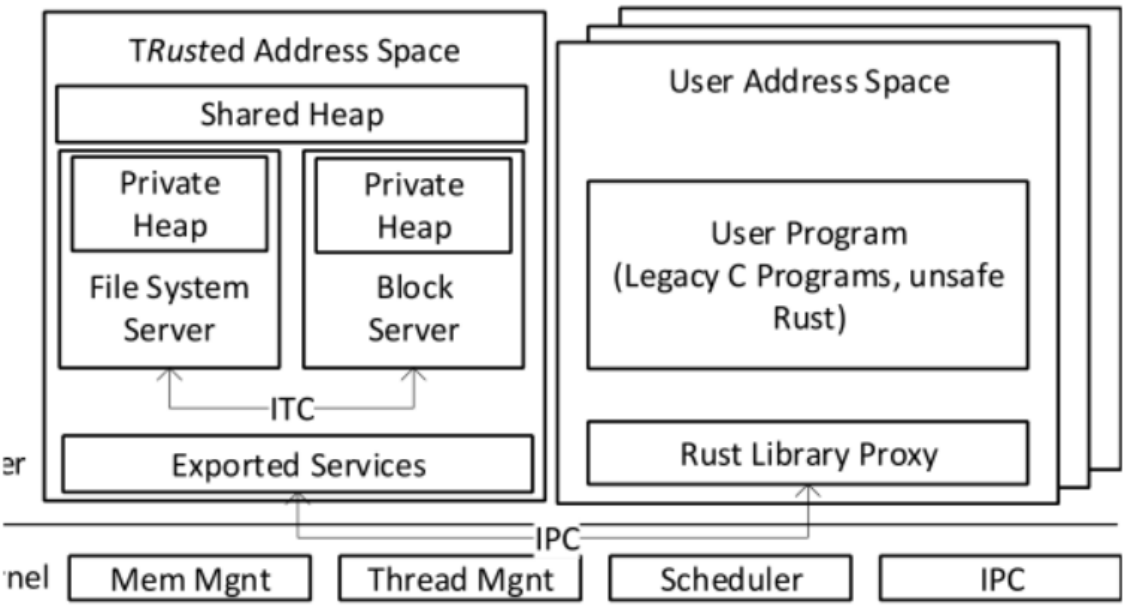
- Rust语言的功能（如控制流完整性和展开）可以实现硬件瞬时故障检测和错误恢复，而不会导致资源泄漏

引入：

- 虽然许多用Rust编写的操作系统出现了，但它们在系统可靠性方面仍存在不足。Redox是Rust中的微核操作系统，但它缺乏故障恢复机制。
- Theseus[4]是一个优秀的单内核操作系统，具有故障恢复功能。然而，作为单内核操作系统，它无法将用户程序从内核模块中分离出来，这使得恶意用户任务危及整个系统。
- 在本文中，我们提出了一种用Rust编写的微内核操作系统Rustpi，它利用Rust的语言特性来提高整个微内核操作系统的可靠性。

设计：

- Rustpi的体系结构由三部分组成：内核、用户空间服务器和用户空间程序。Rustpi的内核提供了基本的原语，如内存管理、地址空间切换和线程调度。与其他微内核设计不同，Rustpi的用户空间服务器运行在相同的地址空间（称为可信地址空间，TAS）。当服务器间连接时，如图1所示。实现了Rustpi工控机的总体设计，无需地址空间切换或内存复制。内核和用户空间服务器都是用Rust编写的。其他用户空间程序使用TAS中服务器导出的服务。还包括一个提供POSIX文件操作API的C库。



- Rust是一种利用所有权模型的现代编程语言。Rust编译器可以防止典型的内存损坏故障和并发错误。
- 在我们的设计中，服务器或用户进程的地址空间是隔离的。因此，当发生内存访问故障时，微内核可以确保故障被限制在进程本身的地址空间内。
- Rustpi为内核实现了一个独立的内存堆，以增强总体内存安全性，并防止由于堆内存分配而导致的死锁错误。传统上，Rust程序使用全局内存分配器来处理各种堆内存分配。但是，如果程序的一个模块损坏堆内存，则其他模块的数据也可能损坏。
- 因此，Rustpi利用了nightly Rust std库中的`allocator-api`特性，可以区分来自不同模块的分配请求。因此，关键内核模块不再使用同一堆，非关键模块的故障不会影响关键系统模块。

结论：

- Rustpi利用Rust的语言特性提供了一组设计来构建可靠的微核操作系统。它使用一个隔离堆来防止跨关键模块的内存损坏。Rustpi还利用 CFI技术 检测瞬时硬件故障，并通过恢复过程进行处理。此外，Rustpi还采用了Rust的拖放功能，以保持系统调用接口和服务器服务接口的可靠性。
- 一般来说，Rustpi证明了Rust的特性有助于构建可靠的微内核操作系统，其设计也适用于其他微内核操作系统，甚至Linux内核。

NileOS：一种用于大数据处理的分布式非对称核微内核

摘要：

- 大数据应用程序在计算资源方面有着苛刻的期望。因此，通用操作系统并不适合。
- 本文针对大数据应用的需求，提出了一种新的专用分布式微内核。新的微内核采用了基于内核的非对称多处理（AMP）方法。它优化了中断管理和输入/输出，以适应Map-Reduce模型。提出的微内核设计基于以太网

处理器间中断（IPIoE）帧和裸机操作系统标记语言（BOSML）。提出了一种透明的部署机制，将微内核服务的开发人员与底层分发基础设施完全隔离开来，并从部署的角度将应用程序实现解耦。

读到这里有个想法：我们的操作系统没有一个特定的服务对象。

- 设计目标2：尽量减少对数据处理任务的中断，或尽可能消除这些中断。由于我们的微核目标是优化大数据处理，因此它旨在最大限度地减少对执行长时间大数据处理的任务的中断。消除所有中断可能无法实现任务之间的同步。然而，作为一般设计目标，应该消除不必要的中断。这将通过对每个核心采用单个地址空间的方法来减少内存开销，这也将增强缓存和TLB效应。在分时操作系统中，抢占式调度允许用户程序以最小的响应时间共享计算机资源。相反，除非认为有必要，否则将批处理作业分配给不间断的CPU将大大受益【6】。计时器为长批量处理带来了多个挑战。进程运行时，会定期触发中断，因此会延迟CPU并带来开销。使用“始终切换”设备来实现调度器的定期计时器所带来的浪费。
- 设计决策3：禁用工作内核上的计时器中断，除非需要，并将计时器中断路由到管理内核。每个节点至少有一个核心需要启用计时器。为了支持设计目标1和2，我们的设计旨在强调减少整个系统的中断，尤其是在工作内核上。中断的一个主要来源是定期定时器中断，它被路由到通用操作系统中的所有内核。这种计时器中断对于调度很重要，但对于工作内核来说，它们会造成不必要的中断。因此，将在工作内核上禁用计时器中断，并且仅通过配置本地APIC计时器启用计时器，将其路由到一些管理内核。