x-realism 调研报告

目录

小组成员

黄瑞轩 刘良宇 叶升宇 许坤钊

项目概述

内核是操作系统的核心。它是硬件和计算机进程之间的**主要接口**。内核将这两者连接起来，以便尽可能有效地调度资源。本项目旨在基于现有的轮子，吸纳多个平台的优点，实现一个我们自己的操作系统内核。

我们的操作系统内核计划实现以下特性：

- 微内核
- x86 架构
- MMU 进行轻量级隔离
- 多任务并发的支持
- **高性能**
- 安全（漏洞少）

在上述特性中，高性能是我们的一大关注点，因为从当前的微内核架构来看，性能问题（主要涉及进程间通信，上下文切换的高开销）一直备受关注，我们期望就此部分进行优化，提高效率。此外，针对 MMU 的安全问题，我们期望进行合理的设计，设计上追求简单但相对灵活，尽量减少漏洞，保证良好的隔离。

我们建构思路是从对应用由简到繁的支持角度出发，满足应用的阶段性需求。根据特性（需求）逐步添加或增强操作系统功能，最终形成一个相对完善的操作系统内核。我们期望通过此项目来加强对操作系统相关理论概念的理解，同时掌握操作系统设计的能力。

项目背景

操作系统

目前流行的三大操作系统 Windows，Mac OS（现也称 OS X 或 Darwin），Linux 都面临着不足，下面简述它们的主要缺点：

Windows

- 系统稳定性差。Windows 的各个程序不是相互独立的，软件的崩溃容易导致系统瘫痪。
- 软件管理安装机制差，软件和软件之间不隔离配置，而是共用一个庞大的全局注册表，各个软件有自己设计的安装和卸载机制，从而很难说删除“干净”某个软件。
- 安全性差，即使存在自带的 Windows Defender，电脑仍然容易遭受病毒的攻击，因此常常需要不断地进行系统更新以获取最新的安全补丁。

Mac OS

- 软件兼容性和生态差。
- 硬件升级和定制化困难，因为 Mac 电脑的 CPU 和内存等与其他硬件和操作系统的耦合性很高，更换硬件可能导致系统拒绝启动。

Linux

- 驱动问题。Linux 无法做到系统与驱动分离，驱动没有稳定的接口，内核变动时驱动就得跟着变动，导致驱动的开发困难，很多设备缺乏好用的驱动。
- 发行版过多，缺乏统一的社区，不同 Linux 发行版的软件生态不同。
- Linux 内核是宏内核，可移植性较差，同时很多系统服务进程运行在内核态，服务的故障会影响整个系统（具体可以参看 ysy 的调研报告）。

宏内核

概述

宏内核（实际上更好的说法应该是单片内核）是一种操作系统架构，其中整个操作系统都在内核空间中工作。单体模型与其他操作系统架构（例如微内核架构）的不同之处在于它单独定义了计算机硬件上的高级虚拟接口。一组原语或系统调用实现所有操作系统服务，例如进程管理、并发和内存管理。设备驱动程序可以作为**模块**添加到内核中。

基本架构

优缺点

优点

- 拥有宏内核的主要优点之一是它通过系统调用提供 CPU 调度、内存管理、文件管理和其他操作系统功能。
- 它是一个完全在单个地址空间中运行的单个大型进程。
- 它是一个单一的静态二进制文件。一些基于单片内核的操作系统的示例包括 Unix、Linux、Open VMS、XTS-400、z/TPF。

缺点

- 如果任何服务出现故障，都会导致整个系统出现故障。
- 如果用户必须添加任何新服务。用户需要修改整个操作系统。

函数调用(对比微内核 IPC)

宏内核通常使用函数调用，以在运行相同的操作系统上下文中的子系统之间转移控制：

1. 将参数放在处理器寄存器中（由编译器完成）。
2. 调用子程序。
3. 子程序访问寄存器来解释请求（由编译器完成）。
4. 子程序将结果返回到另一个寄存器中。

从上面的描述中已经很明显地可以看出，宏内核可以依赖比微内核级别低得多的处理器组件，同时被现有的编程语言很好地支持，用于实现操作系统。

微内核必须操作消息队列，这些消息队列是更高级别的结构，并且与寄存器不同，不能由处理器直接修改和处理。

微内核

简述

微内核是可以提供实现操作系统(OS)所需机制的近乎最少数量的软件。这些机制包括低级**地址空间**管理、**线程**管理和**进程间通信**(IPC)。

如果硬件提供多个环或CPU 模式，则微内核可能是唯一在最高特权级别执行的软件，通常称为超级用户或内核模式。传统的操作系统功能，例如**设备驱动程序**、**协议栈**和**文件系统**，通常从微内核本身中移除，而是在用户空间中运行。

就源代码大小而言，微内核通常比单体内核小。例如，MINIX 3微内核只有大约 12,000 行代码。

基本架构

基本组件和最小化

由于微内核必须允许在其上构建任意操作系统服务，它必须提供一些核心功能。至少，这包括：

- 一些处理**地址空间**的机制，是管理内存保护所必需的。
- 一些用于管理 CPU 分配的执行抽象，通常是**线程**或**调度程序激活**
- **进程间通信**，需要调用在它们自己的地址空间中运行的服务器

> A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality.

其他一切都可以在用户模式程序中完成，尽管在某些处理器架构上实现为用户程序的设备驱动程序可能需要特殊权限才能访问 I/O 硬件。

与最小原则相关，对微内核设计同样重要的是**机制和策略的分离**，它使得在最小内核之上构建任意系统成为可能。内核中内置的任何策略都不能在用户级别被覆盖，因此限制了微内核的**通用性。**可以通过更换服务器（或让应用程序在提供类似服务的竞争服务器之间进行选择）来更改在用户级服务器中实施的策略。

为了提高效率，大多数微内核都包含调度程序和管理定时器，这**违反**了最小原则和策略机制分离的原则。

启动（引导）基于微内核的系统需要不属于内核的**设备驱动程序。**通常这意味着它们在引导映像中与内核一起打包，并且内核支持定义驱动程序如何定位和启动的引导协议；这是L4 微内核的传统引导程序。一些微内核通过在内核中放置一些关键驱动程序来简化这一点（违反了最小化原则），LynxOS和最初的Minix就是例子。有些甚至包括**文件系统**在内核中以简化引导。基于微内核的系统可以通过兼容多重引导的引导加载程序进行引导。此类系统通常加载静态链接的服务器以进行初始引导或安装操作系统映像以继续引导。

微内核的一个关键组件是良好的IPC系统和虚拟内存管理器设计，它允许以安全的方式在用户模式服务器中实现页面错误处理和交换。由于所有服务都由用户模式程序执行，因此程序之间的有效通信方式是必不可少的，远比单片内核更重要。IPC 系统的设计决定了微内核的成败。为了有效，IPC 系统不仅必须具有低开销，而且还必须与 CPU 调度很好地交互。

微内核与宏内核对比

Basis for Comparison	Microkernel	Monolithic Kernel
Size	Microkernel is smaller in size	It is larger than microkernel
Execution	Slow Execution	Fast Execution
Extendible	It is easily extendible	It is hard to extend
Security	If a service crashes, it does effects on working on the microkernel	If a service crashes, the whole system crashes in monolithic kernel.
Code	To write a microkernel more code is required	To write a monolithic kernel less code is required
Example	QNX, Symbian, L4Linux etc.	Linux,BSDs(FreeBSD,OpenBSD,NetBSD)etc.

MINIX

Minix，是一个采用微内核的迷你版本的类 Unix 操作系统，由塔能鲍姆教授为了教学之用而创作。它也影响了 Linux 内核的开发。

全套 Minix 除了启动的部分以汇编语言编写以外，其他大部分都是 C 语言编写。分为：内核、存储器管理及文件系统三部分。

文件系统与存储器管理作为模块，不是在操作系统核心中运作，而是在用户空间运作。至 Minix 3 时，IO 设备也被移到用户空间运作。此外，由于 Minix 主要用于教学，因此代码简洁，可读性较强。

从MINIX 3开始，开发的主要目标从教育转移到创建高度可靠和自我修复的微内核操作系统

Mach

Mach 由 CMU 开发，也是第一代微内核架构的操作系统，主要是出于研究目的被开发出来，特别是在分布式与并行计算上。

Mach 按以下几个概念作为其基础：

- “任务” 即拥有一组系统资源的对象，允许“线程”在其中执行。
- “线程” 是执行的基本单位，拥有一个任务的上下文，并且共享任务中的资源。
- “port” 是任务间通讯的一组受保护的消息队列；任务可以对任何port发送或接收数据。
- “消息” 是某些有类型的数据对象的集合，它们只可以发送至port - 而非某特定任务或线程。

但作为第一代微内核系统， Mach 的性能并不好：这一方面是因为 IPC 的开销，另外一方面也是因为频繁的上下文切换

这最终导致一些在 Mach 基础上继续改进的操作系统实现上将很影响时间的文件系统模块和驱动模块移动到了内核内部，一定程度上违背了微内核的设计哲学

L4

> L4 是一种微内核构架的操作系统内核，最初由约亨·李德克（Jochen Liedtke）设计，前身为 L3 微内核。在最开始，L4 只是一个由约亨·李德克设计并实现的单一的产品，用于 Intel i386 上的一个高度优化内核。L4微内核系统由于其出色的性能和很小的体积而开始被计算机工业所认知。随后，L4 的系统在多个方面上有了高速的发展，值得提及的是一个更加独立于硬件平台的版本，被称为 Pistachio，之后又被移植到了许多不同的硬件构架上。现在已经形成一个微内核家族，包括 Pistachio，L4/MIPS，与 Fiasco。

后序发展上，L4 主要用于类 Unix、可移植操作系统接口(POSIX) 兼容类型。

L4 秉承极简，高效和安全的设计理念：

- 相对于 Mach, Mach 的 ipc 运行缓慢的一个很重要原因是 ipc 代码段过大，会发生较多 L1 cache miss，很影响时间。这启发了微内核的一个设计逻辑： **微内核本身必须充分小**
  - 为此，L4 以及它的前身 L3 的很多代码采用汇编语言编写
- L4 微内核仅提供以下基本概念的抽象
  - 地址空间 (抽象出页表，并提供内存保护)
  - 线程和调度 (抽象出代码执行过程，并提供时间上的保护)
  - IPC 通信 (用于跨越不同（进程）的抽象边界，进行受控的通信)

IPC

相比较 Mach 采取异步的利用内核缓冲区进行进程间通信的策略，L4 使用同步 IPC，这意味着一个集合通信模型，当发送者和接收者都准备好时交换消息。如果两者都在同一个内核上运行，这意味着其中一个将阻塞，直到另一个调用 IPC 操作。

在 L4 中，IPC 通过 “端点对象”。端点可以被认为是一个邮箱，发送者和接收者通过该邮箱通过握手交换消息。任何拥有 Send 能力的人都可以通 Endpoint 发送消息，任何拥有 Receive 上限的人都可以接收消息。这意味着每个端点可以有任意数量的发送者和接收者。特别是，无论有多少线程尝试从 Endpoint 接收，特定消息仅传递给一个接收者（队列中的第一个接收者）。

seL4

项目简介

seL4 是 L4 微内核家族的一员，着重强化了 L4 内核的安全性

seL4 的形式验证使其有别于任何其他操作系统。简而言之，它在系统中运行的应用程序之间提供了最高的 *隔离* 保证，这意味着可以控制系统某个部分的妥协并防止损害系统的其他可能更关键的部分。

具体来说，seL4 的实现在形式上通过不同层次的接口的抽象以及每一层次的状态机形式验证被证明是正确的，并且如果配置正确，它的操作也已被证明在最坏情况下执行时间具有安全上限。它是世界上第一个具有这种证明的操作系统，并且仍然是唯一一个经过验证的具有基于细粒度能力的安全性和高性能的操作系统。它还为 混合临界实时系统 提供最先进的支持。

特性

地址空间

内核在引导到第一个用户进程（称为“根任务”）后，通过将相应的上限存放在根任务的 Cspace 中，将所有空闲内存（称为“未类型化”）交给用户空间来控制系统资源。然后根任务可以实施其资源管理策略，例如通过将系统划分为安全域并将每个域交给一个不相交的无类型内存子集。

用户空间可直接访问的唯一对象是“框架对象”：这些对象可以被映射到页表，之后用户空间可以写入由这些框架对象表示的物理内存。

简而言之，seL4 将内核资源的管理导出到用户级别，并使它们受到与用户资源相同的基于能力的访问控制。

线程通信

通信可以通过 IPC 或共享内存进行。IPC 通常应用于短消息，不长于几百字节的消息大小，这是依赖实现定义和体系结构的限制，但通常消息应该保持在几十个字节。对于较长的消息，应使用共享缓冲区。

共享缓冲区访问可以通过通知机制同步。

L4 相关思想

1. **规范**：也即前文的形式上(功能上)的正确性证明
2. **最小化**：这是微内核的核心思想。seL4 在内核中没有设备驱动程序（除了中断控制器和计时器），内存管理采用极端方法，甚至内核内存也由用户级代码管理。
3. **性能**：这是 seL4 的核心驱动力
    - 关注代码的关键路径（实际上类似 Amdahl 定律，某些操作比其他操作更频繁地使用，并且可以通过将成本从经常使用的“热”操作转移到不经常使用的操作来最大化整体性能。）
    - 不要为不使用的东西买单。(有些功能即使不使用它也是有代价的：进行额外检查的性能成本，异常处理的复杂性成本。）
4. **安全**：安全性实际上也是一个核心原则，内核从根本上是为提供尽可能强的隔离而设计的，同时也要注意不能让安全性称为降低性能的借口。

RT-Thread Smart

而 RT-Thread Smart 定位于成为一个面向实时应用场合的高性能混合微内核操作系统，以期填补传统 RTOS 和大型操作系统之间的留白。

RT-Thread Smart 上，使用了共享内存的方式，把交换的数据内存块，分别投到不同的进程地址空间上，从而不需要做额外的数据拷贝。

共享内存的方式进一步降低了 IPC 通信的成本。

由于采用共享内存， RT-Thread Smart 主要面向带 MMU（Memory Management Unit，内存管理单元）的中高端处理器，为众多领域提供更具竞争力的操作系统基础软件平台，具备快速启动、POSIX 接口全兼容等特性。

共享内存通信

当数据在保护域之间复制时，它会从源位置复制到内核内存区域的临时存储中，然后再复制到目标位置。对于大量数据，这些复制操作占据了大部分通信时间。为了使通信更快，应避免复制操作。可考虑通过使源位置在目标保护域中可见（在内核内存区域内）消除了一次复制操作。因此，消息可以从源位置直接复制到目标位置。但是，临时映射的成本会影响整体通信成本。此外，通信的源需要将消息的参数复制到消息缓冲区中，而目标需要将参数从消息缓冲区中复制出来。

避免这些复制操作的一种机制是共享内存通信。无需将数据复制到消息缓冲区，然后将消息缓冲区从发送方复制到接收方，最后再将数据从消息缓冲区中复制出来，消息缓冲区可以位于发送方和接收方共享的内存区域中。需要注意的是，必须同步对此共享消息缓冲区的访问以避免数据损坏

此外，当使用共享内存通信时，我们在保护域中有一系列应用程序处理相同的数据。由于数据处理是顺序的，因此一次只能有一个应用程序对数据具有写访问权。通常建议将可写的共享内存区域映射到仅一个应用程序。当下一个应用程序应该处理数据时，必须转移写入共享内存区域的权限。

Fuchsia

Fuchsia 是谷歌开发的基于能力的开源操作系统。在 Google 的基于 Linux 的操作系统（如 Chrome OS 和 Android）之后，Fuchsia 基于名为 Zircon 的独特内核。它于 2016 年 8 月作为自托管 git 存储库公开亮相，没有任何官方公司公告。经过多年的发展，它的正式产品发布在第一代 Google Nest Hub 上，取代了原来的 Cast OS。

Fuchsia 基于一种新的消息传递内核，以矿物锆石命名为 Zircon。它的代码库源自嵌入式设备的 Little Kernel (LK)，旨在为各种设备提供低资源使用。 LK 由 Travis Geiselbrecht 开发，他也是 Haiku 使用的 NewOS 内核的共同作者。那是 BeOS 的免费软件重新实现，它是由 Erich Ringewald 在 1990 年代设计的, 1988 年 Apple 的 Pink 的原始首席设计师。

Fuchsia 作为一个实时操作系统，内核是 Zircon ，采用的也是微内核架构。事实上，很多像 Fuchsia 和 FreeRTOS 这样的实时操作系统采用的都是微内核架构，微内核的结构更为小巧，也更容易对每个模块分别维护更新，这也使得它更能适应高度定制化的嵌入式设备的内核开发

Rust for OS

用于实现操作系统的语言

最初的 Unix 系统是完全用汇编语言写出来的，之后 B 语言和 NB (New B) 语言都被使用过。由于这些语言中只能处理计算机字节，没有类型并且不支持浮点运算，Dennis Ritchie 发明了 C 语言，C 语言从那以后就成为了开发操作系统最流行的编程语言。如今主流操作系统内核的少数部分也用 C++ 实现。

但是编写操作系统内核并不是只能用汇编跟 C，C++，一门语言能否用于编写操作系统，取决于其二进制代码是否能够在裸机上执行（也即不依赖标准库），因为标准库要依赖操作系统为其提供系统调用。早期的 Mac OS 曾经使用 Pascal 编写。

> In principle, any language with the ability to interact with lower-level code can be used to write operating systems.

Rust 与 C/C++ 的对比

C/C++ 诞生在硬件极为昂贵的时代，所以追求性能，其过于灵活，最大的问题就是安全性问题，很容易出现漏洞，包括但不限于以下

- **释放后使用/双重释放错误**：由于需要手动管理内存，导致需要在`free`时小心翼翼
- **悬空指针**：可能导致空指针引用等不安全行为
- **缓冲区溢出错误**：这是造成大量漏洞和导致被攻击的原因
- **数据竞争**：数据的不安全并发修改
- **未初始化的变量**：会引发一系列未定义行为

在编写、调试程序时通常需要花费大量的时间来解决内存或数据竞争问题，而人肉 code review 大大降低了效率，也给后续的维护造成了极大的挑战，而下文会提及 Rust 是如何实现安全的。

Rust 与带 GC 的编程语言对比

随着硬件成本的降低，Java 等语言用性能(GC)来换安全性，但是 GC 的劣势也很明显。

- **代价昂贵**：无论是何种类型的 GC ，其维护代价都不低。
- **内存开销**：运行时需要动态回收，降低性能
- **非确定性**：不知道何时会暂停进行回收，取决于所用内存
- **难以优化**：无法自行优化缓存，因为 GC 不知道程序将如何使用内存，其优化方式未必最优。

> In our production environments, we have seen unexplainable large
> STW pauses (> 5 seconds) in our mission-critical Java applications.

Rust 优势

Rust 是一门强调**安全**、**并发**、**高效**的系统编程语言。无 GC，实现内存安全机制、无数据竞争的并发机制、无运行时开销的抽象机制，它声称解决了传统 C 语言和 C++ 语言几十年来饱受诟病的内存安全问题，同时还保持了很高的运行效率、很深的底层控制、很广的应用范围，在系统编程领域具有强劲的竞争力和广阔的应用前景。

高效性

Rust 无 GC，无 VM，无解释器，具有极小的运行时开销，能充分高效利用CPU和内存等系统资源。

> It is an explicit goal of Rust to be as fast as C++ for most things. Given that Rust is built on top of LLVM, any performance improvements in it also help Rust become faster.

以下为几门语言的性能对比

Language	User	System	Total	Slower than (C++)	Language version	Source code
C++ *(optimized with -O2)*	0.899	0.053	0.951	–	g++ 6.1.1	link
Rust	0.898	0.129	1.026	7%	1.12.0	link
Java 8 *(non-std lib)*	1.090	0.006	1.096	15%	1.8.0_102	link
Go	2.622	0.083	2.705	184%	1.7.1	link
C++ *(not optimized)*	2.921	0.054	2.975	212%	g++ 6.1.1	link
Python 3.5	17.950	0.126	18.077	1800%	3.5.2	link
Python 2.7	25.219	0.114	25.333	2562%	2.7.12	link

安全性

Rust 设计上是内存安全的，这也是一大亮点和相较 C/C++的优势。

> Rust's rich type system and ownership model guarantee memory-safety and thread-safety — and enable you to eliminate many classes of bugs at compile-time.

它不允许**空指针**、**悬空指针**或**数据竞争**。其丰富的**类型系统**和**所有权模型**保证了内存安全和线程安全，使得能够在编译时消除许多类别的错误。也就是说，一段能跑起来的代码大概率是安全的。具体特性如下

- **内存管理**：采用 **RAII**(resource acquisition is initialization) 模式；有引用(`&`)，此类指针不涉及运行时引用计数。安全性在编译时进行验证，防止未定义行为。
- **所有权模型**：**变量**与存放在内存某块的**值**绑定
  - 每个值只能绑定到一个变量，此时该变量拥有值的**所有权**；
  - 变量离开作用域，它负责回收位置和销毁值。
- **智能指针**：通过智能指针 `Box<T>` 来控制存放在**堆**内存中的类型为 `T` 的值；Rust 的智能指针功能丰富，许多开箱即用
- **生命周期**：通过生命周期注释，保证生命周期一致，杜绝**悬空指针**。
- **借用规则**：可以**借用**变量控制的值得到**不可变引用**(`&T`)和**可变引用**(`&mut T`)
  - 在一个引用存在的全程，被它借用的值不能销毁
  - 一个值同时只能有一个可变引用，但可以有多个不可变引用
- **移动语义**：内置的静态分析器不允许移动后使用，由 borrow checker 进行检查。
- **类型安全**：对一些基本类型的行为进行了限制，较少甚至消除**语义不明**确行为。
- **并发编程模型**：简单来说就是编译器阻止了一切可能的**数据竞争**。
- **错误处理**：使用 `Option<T>` 解决空指针问题；针对可恢复和不可恢复错误有不同处理。

生产力

Rust 有内容详尽的**文档**以及开放、友好、高效的**开源社区**。并且有一流的开发**工具链**。

- 集成的包管理工具 cargo 。
- 编译器能提供有效的错误提示和修正信息，减少了 debug 的时间。
- 自动格式化程序 clippy 规定了代码格式，减少了团队磨合统一标准的时间。
- 支持单元测试，不用引入测试框架。

内存管理单元（MMU）

概述

内存管理单元(MMU)，有时称为分页内存管理单元(PMMU)，是一种**计算机硬件单元**，所有内存引用都通过自身传递，主要执行**虚拟内存地址到物理地址的转换**。

现代 MMU 通常将虚拟地址空间（处理器使用的地址范围）划分为页面，每个页面的大小为2的幂，通常为几千字节，但它们可能更大。地址的低位（页内的偏移量）保持不变。高地址位是虚拟页号。

内核进行内存管理的必要性

为了限制应用访问内存空间的范围并给操作系统提供内存管理的灵活性，计算机硬件需要引入**内存保护/映射/地址转换硬件机制**，如 RISC-V 的基址-边界翻译和保护机制、x86 的分段机制、RISC-V/x86/ARM 都有的分页机制。如果在地址转换过程中，无法找到物理地址或访问权限有误，则处理器产生非法访问内存的异常错误。

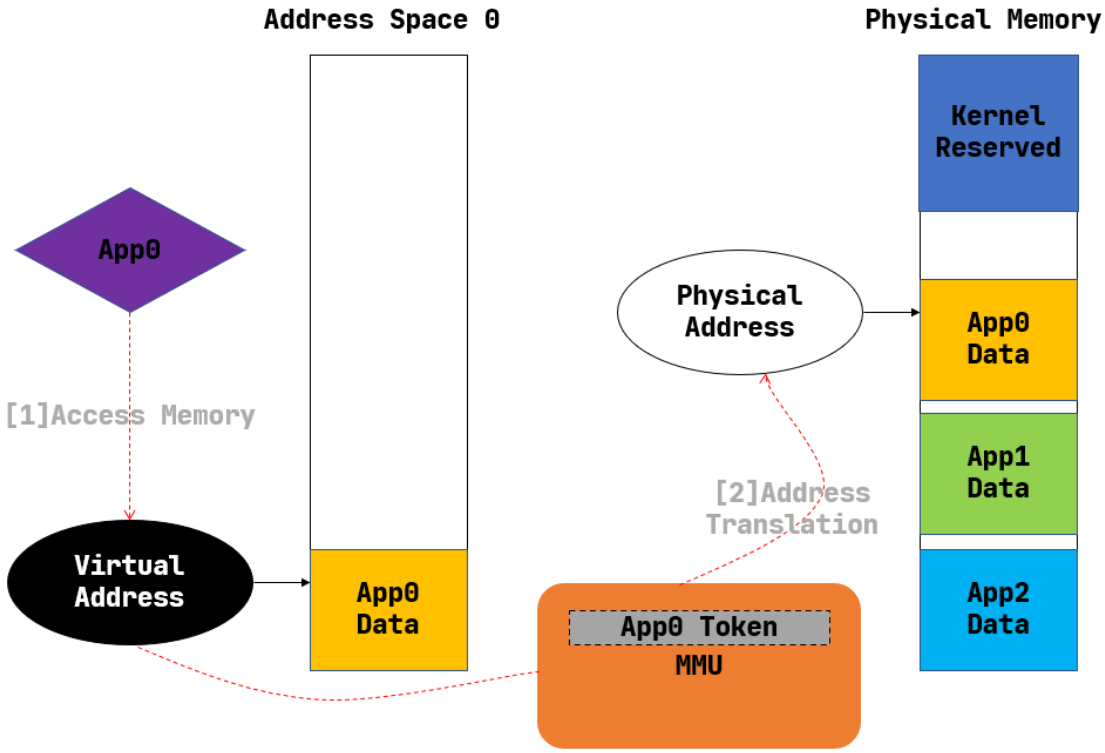
操作系统与真实内存之间的隔离

为了发挥上述硬件机制的能力，操作系统也需要适配，以便更好地管理物理内存和虚拟内存，并给应用程序提供统一的虚拟内存访问接口。CPU 访问数据和指令的内存地址是虚地址，通过硬件机制（比如 MMU +页表查询）进行地址转换，找到对应的物理地址。**地址空间（Address Space）** 抽象由此产生。在内核中建立虚实地址空间的映射机制，给应用程序提供一个基于地址空间的安全虚拟内存环境，就能让应用程序简单灵活地使用内存。

> - 从应用开发的角度看，需要应用程序决定自己会被加载到哪个物理地址运行，需要直接访问真实的物理内存。这就要求应用开发者对于硬件的特性和使用方法有更多了解，产生额外的学习成本，也会为应用的开发和调试带来不便。
> - 从内核的角度来看，将直接访问物理内存的权力下放到应用会使得它难以对应用程序的访存行为进行有效管理，已有的特权级机制亦无法阻止很多来自应用程序的恶意行为。

应用能够直接看到并访问的内存就只有操作系统提供的地址空间，且**它的任何一次访存使用的地址都是虚拟地址**。应用不再具有直接访问物理内存的能力。应用所处的执行环境在安全方面被进一步强化，形成了用户态特权级和地址空间的二维安全措施。

这样，每个应用独占一个地址空间，里面只含有自己的各个段，它可以随意规划属于它自己的各个段的分布而无需考虑和其他应用冲突；同时鉴于应用只能通过虚拟地址读写它自己的地址空间，它完全无法窃取或者破坏其他应用的数据（不在其地址空间内）。这是地址空间抽象和具体硬件机制对应用程序执行的**安全性**和**稳定性**的一种保障。



MMU与操作系统之间的配合

在 MMU 的帮助下，应用对自己虚拟地址空间的读写被实际转化为对于物理内存的访问。MMU 可能会**将来自不同两个应用地址空间的相同虚拟地址转换成不同的物理地址**。要做到这一点，就需要硬件提供一些寄存器，软件可以对它进行设置来控制 MMU 按照哪个应用的地址映射关系进行地址转换。

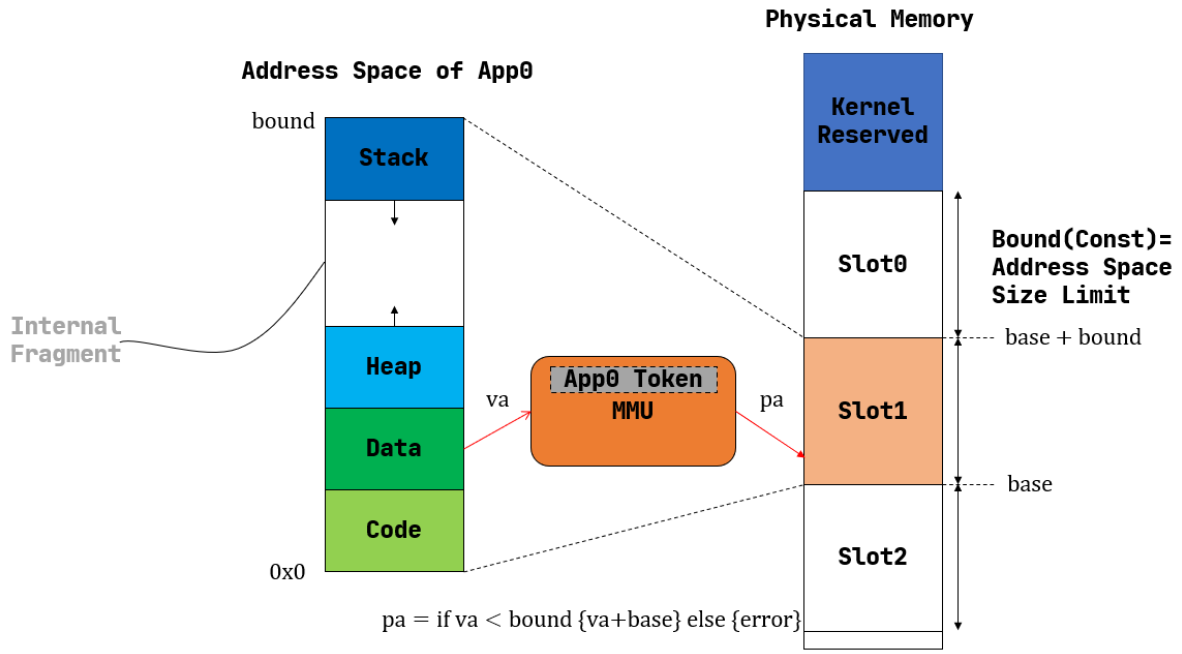
将应用的代码/数据放到物理内存并进行管理，建立好应用的地址映射关系，在任务切换时控制 MMU 选用应用的地址映射关系，是作为软件部分的内核需要完成的重要工作。

> 内核对于 CPU 资源的抽象——**时分复用**，它为应用制造了一种每个应用独占整个 CPU 的幻象，而隐藏了多个应用分时共享 CPU 的实质。地址空间也是如此，应用只需、也只能看到它独占整个地址空间的幻象，而藏在背后的实质仍然是**多个应用共享物理内存，它们的数据分别存放在内存的不同位置**。

现有的地址空间实现策略

地址空间只是一层抽象接口，它有很多种具体的实现策略。对于不同的实现策略来说，操作系统内核如何规划应用数据放在物理内存的位置，而 MMU 又如何进行地址转换也都是不同的。

插槽式内存管理



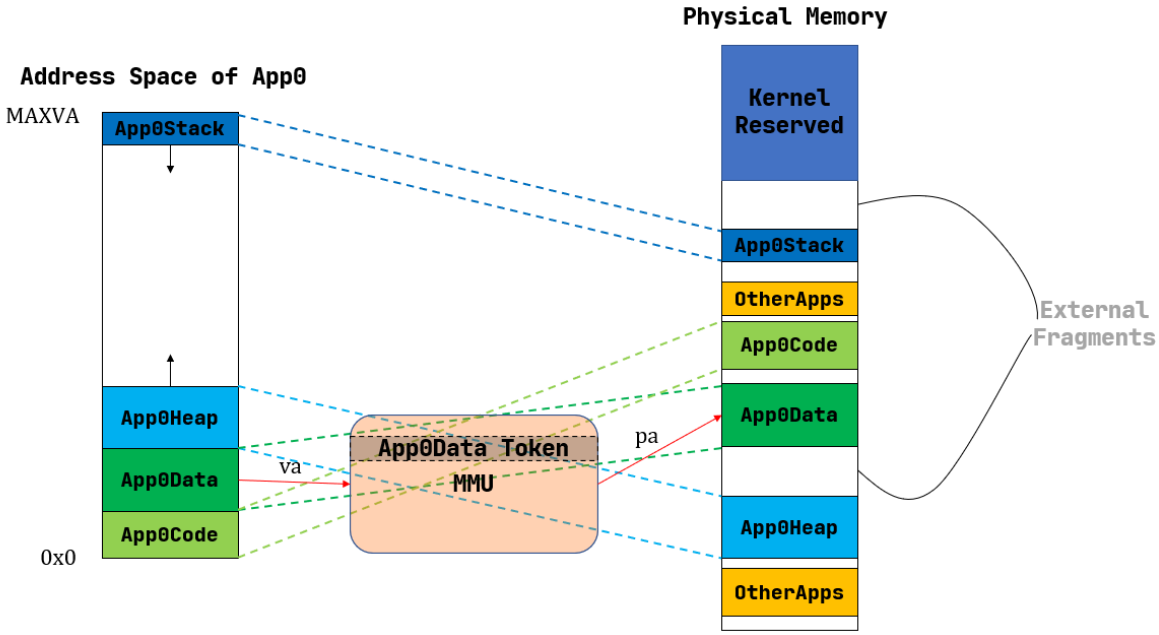
每个应用的地址空间大小限制为一个固定的常数 `bound` ，也即每个应用的可用虚拟地址区间均为 $[0, \text{bound})$ 。随后，就可以以这个大小为单位，将物理内存**除了内核预留空间之外的部分**划分为若干个大小相同的 **插槽** (Slot)，每个应用的所有数据都被内核放置在其中一个插槽中，对应于物理内存上的一段连续物理地址区间，假设其起始物理地址为 base ，则由于二者大小相同，这个区间实际为 $[\text{base}, \text{base}+\text{bound})$ 。因此地址转换很容易完成，只需检查一下虚拟地址不超过地址空间的大小限制（借助特权级机制通过异常来进行处理），然后做一个线性映射，将虚拟地址加上 base 就得到了数据实际所在的物理地址。

好处：实现极其简单，MMU 只需要 base, bound 两个寄存器，在地址转换进行比较或加法运算即可；内核只需要在任务切换时完成切换 base 寄存器。在对一个应用的内存管理方面，只需考虑一组插槽的占用状态，可以用一个位图来表示，随着应用的新增和退出对应置位或清空。

不足：可能浪费的内存资源过多，即固定参数让实现简单，但是不够灵活。

> 注意到应用地址空间预留了一部分，它是用来让栈得以向低地址增长，同时允许堆往高地址增长。每个应用的情况都不同，内核只能按照在它能力范围之内的消耗内存最多的应用的情况来统一指定地址空间的大小，而其他**内存需求较低的应用根本无法充分利用内核给他们分配的这部分空间**。但这部分空间又是一个完整的插槽的一部分，不能再交给其他应用使用。这种在已分配/使用的地址空间内部无法被充分利用的空间就是 **内碎片** (Internal Fragment)，它限制了系统同时共存的应用数目。如果应用的需求足够多样化，那么内核无论如何设置应用地址空间的大小限制也不能得到满意的结果。

分段式内存管理



分段式内存管理是以更细的粒度，也就是应用地址空间中的一个逻辑段作为单位来安排应用的数据在物理内存中的布局。

对于每个段来说，从它在某个应用地址空间中的虚拟地址到它被实际存放在内存中的物理地址中间都要经过一个**不同的线性映射**，于是 MMU 需要用一对不同的 base/bound 进行区分。这里由于每个段的大小都是不同的，也不再能仅仅使用一个 bound 进行简化。当任务切换的时候，这些对寄存器也需要被切换。

> 这里忽略一些不必要的细节。比如应用在以虚拟地址为索引访问地址空间的时候，它如何知道该地址属于哪个段，从而硬件可以使用正确的一对 base/bound 寄存器进行合法性检查和完成实际的地址转换。
>
> 这里只关注分段管理是否解决了内碎片带来的内存浪费问题。

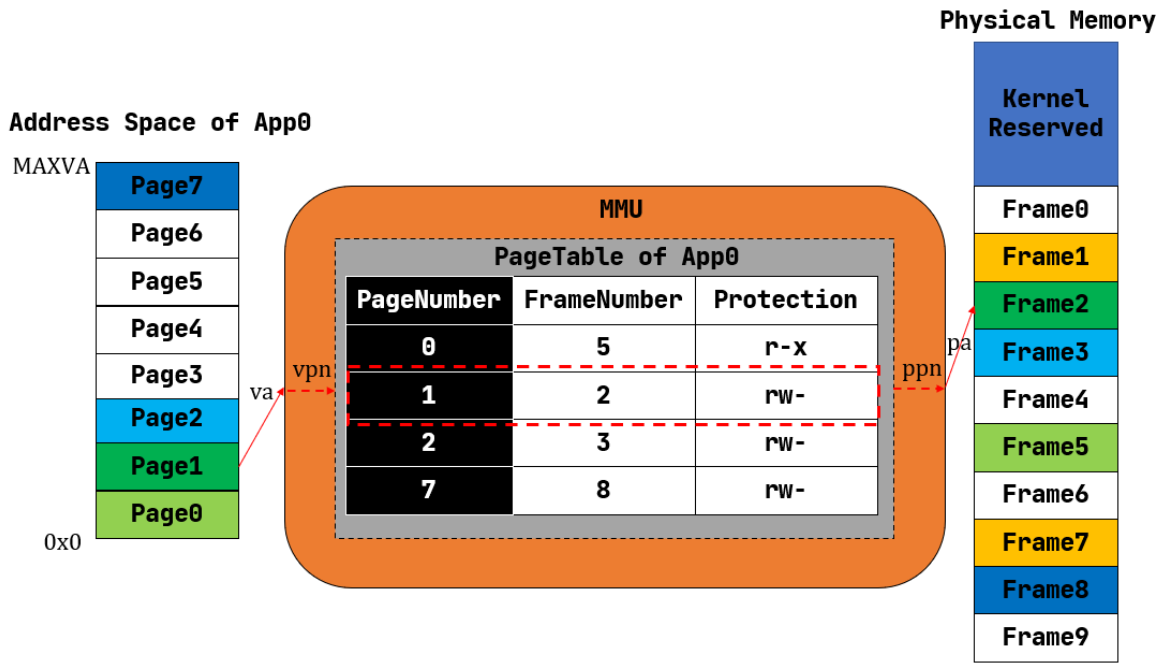
好处：注意到每个段都只会在内存中占据一块与它实际所用到的大小相等的空间。也就是说这是一种按需分配，而不再是内核在开始时就给每个应用分配一大块很可能用不完的内存。因此，不再有内碎片了。

> 堆的情况可能比较特殊，它的大小可能会在运行时增长，但是那需要应用**通过系统调用向内核请求**。

不足：尽管内碎片被消除了，但内存浪费问题并没有完全解决。这是因为每个段的大小都是不同的，内核就需要使用**更加通用、也更加复杂的连续内存分配算法**来进行内存管理，而不能像之前的插槽那样以一个比特为单位。连续内存分配算法就是每次需要分配一块连续内存来存放一个段的数据。随着一段时间的分配和回收，物理内存还剩下一些相互不连续的较小的可用连续块，其中有一些只是两个已分配内存块之间的很小的间隙，它们自己可能由于空间较小，已经无法被用于分配，这就是 **外碎片** (External Fragment) 。

如果这时再想分配一个比较大的块，就需要将这些不连续的外碎片“拼起来”，形成一个大的连续块，然而这涉及到极大的内存读写开销。

分页式内存管理



段的大小不一是外碎片产生的根本原因。若要结合前面两者的优点的话，就需要**内核始终以一个同样大小的单位来在物理内存上放置应用地址空间中的数据**，这样内核就可以使用简单的插槽式内存管理，使得内存分配算法比较简单且不会产生外碎片；同时，这个单位的大小要足够小，从而其内部没有被用到的**内碎片的大小也足够小**，尽可能提高内存利用率。

如图所示，内核以页为单位进行物理内存管理。每个应用的地址空间可以被分成若干个（虚拟）**页面** (Page)，而可用的物理内存也同样可以被分成若干个（物理） **页帧** (Frame)，虚拟页面和物理页帧的大小相同。每个虚拟页面中的数据实际上都存储在某个物理页帧上。相比分段内存管理，分页内存管理的粒度更小且大小固定，应用地址空间中的每个逻辑段都由多个虚拟页面组成。而且每个虚拟页面在地址转换的过程中都使用与运行的应用绑定的不同的线性映射。

为了方便实现虚拟页面到物理页帧的地址转换，我们给每个虚拟页面和物理页帧一个编号，分别称为 **虚拟页号** (VPN, Virtual Page Number) 和 **物理页号** (PPN, Physical Page Number) 。每个应用都有一个表示地址映射关系的 **页表** (Page Table)，里面记录了**该应用地址空间中的每个虚拟页面映射到物理内存中的哪个物理页帧**，即数据实际被内核放在哪里。我们可以用页号来代表二者，因此如果将页表看成一个键值对，其键的类型为虚拟页号，值的类型则为物理页号。当 MMU 进行地址转换的时候，虚拟地址会分为两部分（虚拟页号，页内偏移），MMU首先找到虚拟地址所在虚拟页面的页号，然后查当前应用的页表，根据虚拟页号找到物理页号；最后按照虚拟地址的页内偏移，给物理页号对应的物理页帧的起始地址加上一个偏移量，这就得到了实际访问的物理地址。

在页表中，还针对虚拟页号设置了一组保护位，它限制了**应用对转换得到的物理地址对应的内存的使用方式**，最典型的如 `rwx` 。一旦违反了这种限制则会触发异常，并被内核捕获到。通过适当的设置，可以检查一些应用在运行时的明显错误：比如应用修改只读的代码段，或者从数据段取指令来执行。

好处：分页内存管理既简单又灵活，它逐渐成为了主流的内存管理机制，RISC-V 架构也使用了这种机制。

不足：页内碎片；动态地址变换、方案实施需耗用额外的系统资源；存储扩充问题没有解决——作业大小受到限制，可用块数小于作业需求时需等待。

立项依据

如前所述，当前已经有不少优秀的微内核和 MMU 设计值得我们学习，如划分地址空间进行映射来安全地管理内核内存，针对短消息用 IPC 通信，针对长消息通过共享缓冲区进行“通知”同步，通过细粒度的划分或分页式的方法来提高内存的利用率等。

但是值得关注的是，微内核的 IPC 普遍存在效率低下问题，这是制约其性能提升的关键问题；MMU 方面，无论是插槽式内存管理、分段式内存管理还是分页式内存管理都有各自的问题，或是浪费内存资源，或是造成“外碎片”，或是作业大小受限，没有一个相对完善的设计。

我们计划对微内核的 IPC 做出改进，一个思路是改进 seL4 的快速路径思想，快速路径是内核的附加前端，它可以快速执行一些常见操作的简单案例，这样能加速 IPC 的一些步骤执行，从而达到高 IPC 性能。启用或禁用快速路径不会对内核行为产生任何影响，但对性能可以起到优化。

同时我们试图提出一个新的 MMU 设计，其中一些关键特性如下

- 实现轻量级隔离
- 拥有字节级权限
- 内存使用率高
- 基于可以快速遍历的数据结构
- 快速快照/恢复
- 处理完整地址空间的能力
- 能够在字节级别跟踪未初始化的内存

前瞻性/重要性分析

基于微内核设计操作系统的优化

当前微内核设计操作系统存在的问题

在大多数主流处理器上，在基于微内核的系统中获取服务本质上比单片系统更昂贵。在单体系统中，服务是通过单个系统调用获得的，需要两次*模式切换*（处理器环或CPU模式的改变）。在基于微内核的系统中，服务是通过向服务器发送一个 IPC 消息，并在另一个 IPC 消息中从服务器获得结果来获得的。这需要 **上下文切换** 对于宏内核而言，这些通常被实现为内核进程中的过程，即为函数调用，开销较小。此外，将实际数据传递到服务器并返回可能会产生额外的复制开销，而在单片系统中，内核可以直接访问客户端缓冲区中的数据。

为了在一个微内核的两个组件之间进行通信微内核必须执行以下步骤：

1. 原始组件上下文中的发起线程必须格式化并将请求（或多个请求）放入消息队列中。
2. 原始线程必须以某种方式通知消息已到达的目标组件。要么使用中断（或某种其他形式的信号），要么必须使用目标组件轮询其消息队列。
3. 原始线程可能需要执行上下文切换，如果没有足够的处理器持续运行所有线程。
4. 目标组件现在必须访问消息队列并解释消息，然后执行请求的操作。

可能必须重复以上 4 个步骤才能返回请求的结果给组件。

所以与宏内核相比，IPC 成本高，上下文切换开销大，微内核最大的瓶颈就是性能问题。

x-Realism 的改进

- 一方面，我们尽可能精简内核的规模，减少上下文切换的开销
- 另一方面，我们希望通过共享内存通信的方式降低 IPC 的成本，原先，当数据在保护域之间复制时，它会从源位置复制到内核内存区域的临时存储中，然后再复制到目标位置。对于大量数据，这些复制操作占据了大部分通信时间。可考虑通过使源位置在目标保护域中可见（在内核内存区域内）消除了一次复制操作。但是，临时映射的成本会影响整体通信成本。此外，通信的源需要将消息的参数复制到消息缓冲区中，而目标需要将参数从消息缓冲区中复制出来。避免这些复制操作的一种机制是共享内存通信，消息缓冲区可以位于发送方和接收方共享的内存区域中。这样就减少了大量信息拷贝的开销

Rust 对操作系统的优化

当前用 Rust 实现操作系统的难点

跟 C 语言之间的互相调用（ABI）

在操作系统底层，可能需要 C 和 Rust 语言之间的互相调用，接口目前已经实现好了，可以实现在编译 C 程序时把 Rust 静态链接库链接到其中，也可以在编译 Rust 时链接 C 语言代码，示例参见下面的 GitHub 仓库：RustSamples. 但有时 Rust 与 C 的交互也并不那么直接，这主要是因为很多 C 库有它们自己对对象生命周期的管理方式。

内存模型

由于 Rust 特有的变量生命周期及借用等机制，所有变量在内存中都是可移动的，这对于引用 Linux 的数据结构来说是会有问题的。例如在 safe Rust 中，诸如双向链表这样的数据结构无法被实现，为了解决该问题就需要引入 unsafe 的 Rust 代码，这样的话 Rust 的安全优势也将荡然无存。而且内存模型的问题还远不止数据的申请、释放与内存布局，在多核时代内存模型还增加了操作系统在保证执行顺序与并发灵活性之间复杂的取舍策略。

panic、alloc到底如何实现

在 Linux 当中一旦内核态的代码执行中出现不可恢复的错误，一般是通过 panic 操作来记录相关信息及调用栈，但由于 Rust 的内存申请与释放机制，其编译器通常会隐藏内存分配的操作，这就很可能使 `panic!` 的调用出现问题。而且在某些驱动程序中，内存分配失败不应该直接使内核产生 panic，因此 Rust 在申请内存失败后如果直接调用 `panic!`，可能也是错误的。

而且在 Linux 标准接口中的内存分配 alloc API 也需要为 Rust For Linux 项目做好准备。标准库的所有堆分配对象类型（例如 Box 和 Vec）都使用假定不会失败的分配机制。在用户空间中，这种假设并非完全不切实际，因为页面交换、写入时复制和按需分页会产生可用内存充足的错觉。可用物理内存实际用完的故障点可能只有在用于分配它的系统调用很久之后才能发现。然而，在内核空间中，几乎没有内存可以安全地换出到持久存储，并且链中没有更靠后的系统可以处理分配失败的问题。因此，在内核中使用标准容器类型是危险的。建议的解决方案是明确分配失败的可能性。更改各种容器类型以应对故障，这需要对 Rust 代码的编写方式进行重大更改，但简单地扩展当前实现以提供额外的构造函数可能是可行的，如果分配失败，它将返回错误。从目前来看，实现 alloc 这些标准接口，很可能会大量引入很多 unsafe 的 Rust 代码，这将使 Rust 的价值大大降低。

稳定内联汇编

其实即使是 C 语言也无法单独完成开发一整套操作系统的任务，汇编语言在很多情况下是操作系统所必须的，因为有一些关键操作必须直接调用 CPU 底层的指令才能执行，目前 Rust 在开启 `#!(feature(asm))` 的情况下可以支持内联汇编，但是目前 Rust 对于内联汇编语言的支持并不成熟，更不算稳定，还需要进一步完善。

架构支持

目前，唯一成熟 Rust 编译器只有 rustc 这一个，它是通过 LLVM 来生成指令码。Linux 内核支持许多种体系架构，其中一些架构并没有现成的 LLVM 后端（backend）。其他一些架构存在 LLVM 后端，但 rustc 还尚未不支持该后端。

x-Realism 的改进

针对以上的缺点逐一列出解决方案：

和 C 的互相调用

由于 C 代码是通过 `extern` 引入的外部代码块，Rust 编译器不会对其进行检查，我们需要保证声明的正确性，同时对原生的 C API 进行封装，以保证内存安全。例如，在使用 Vector 与 C 模块对接时，需要把 Vector 转换为内存中的指针，这时我们可以用一个模块封装处理对接的函数，只暴露安全、高级的接口，隐藏非安全的内部细节。

内存模型

内存模型的问题其实是一个取舍问题，小组成员将尽量避免需要使用双向链表这样无可避免要引入 unsafe Rust 代码的数据结构，但如果实在需要，只能选择牺牲一定的安全性。这里牺牲的安全性也只是在编译层面 Rust 编译器提供的安全性，只要合理编写代码还是可以避免一定的安全问题。

panic、alloc的实现、内联汇编和架构支持

这三者都偏向于 Rust 语言本身的问题。其中 panic 和 alloc 尽管目前在语言层面没有支持，但其引发问题的条件也较为苛刻——没有更多的存储空间可以分配，我们可以假定用户并不需要分配这样多的存储空间。架构支持上则是 Linux 内核开发的问题，与本项目关系不大。

内存管理设计优化

现有内存管理安全性问题

Row Hammer 漏洞

2014年，卡内基梅隆大学宣布发现一种存在于动态随机储存器DRAM，也就是现代数字设备使用的内存芯片上的漏洞 Row Hammer。Rowhammer 攻击可以悄悄地破坏 MMU 强制隔离，因为它们根本不需要访问受害者行，并且它们不依赖于隔离机制中的任何设计或实现缺陷。
Rowhammer 攻击已被证明可以打破所有流行的隔离形式（如进程内隔离、进程间隔离、内核-用户隔离、虚拟机间隔离和特权用户-访客隔离等）。利用这种漏洞，攻击者可以通过反复加载

内存的特定行来实现篡改权限等恶意攻击，无论是运行何种操作程序，PC、手机等几乎所有搭载DRAM内存的X86构架CPU设备都会都会存在安全隐患，这种漏洞被称为RowHammer。随后，内存行业中引入了TRR等手段，认为此漏洞已经被修复。

2020年5月，长期跟踪行业学术与技术发展的联想集团内存研发团队发现，学术界在讨论一种新的威胁，高危的内存漏洞RowHammer没有被完全解决，现有的安全手段可能无法完全阻止利用这个漏洞进行攻击，RowHammer可能死灰复燃：一种新的攻击方法有机会绕过TRR、ECC等内存保护机制进行更有危害的攻击。联想集团内存团队花费半年时间，利用开源工具进行了大量测试，确认在多个供应商提供的元器件中依然存在可利用的RowHammer漏洞。

CATTmew 漏洞

为了减轻Row Hammer攻击，由 CATT 引入的物理域隔离通过将物理内存划分为多个分区并保持每个分区仅由一个域占用来物理分离每个域。 CATT 将物理内核隔离作为第一个通用且实用的纯软件防御来保护内核免受攻击，因为内核是最吸引人的目标之一。

> 观察到，Row Hammer 攻击本质上需要攻击者控制的内存与特权内存（例如，页表）在物理上相邻，CATT 概念旨在物理分离不同域的内存。
>
> 具体而言，它将物理内存划分为多个分区，并进一步确保分区由至少一个未使用的DRAM行分隔，并且每个分区仅由单个域拥有。例如，用户空间中的堆会从用户分区分配，页表会从内核分区分配。这样做可以将一个域引起的位翻转限制在自己的分区中，从而防止 Row Hammer 攻击影响其他域。

2018年，有人提出一种名为CATTmew的漏洞，声称能够打败纯软件的物理内核隔离。

漏洞利用可以在**不耗尽**页面缓存或系统内存的情况下工作，或者依赖于虚拟到物理地址映射。该漏洞利用的是，现代操作系统具有的双重内核缓冲区（例如，视频缓冲区和 SCSI 通用缓冲区）同时由内核和用户域拥有。这种缓冲区的存在使物理内核隔离失效，并使基于Row Hammer 的攻击再次成为可能。现有的 Row Hammer 攻击实现了 root/内核权限提升，耗尽了页面缓存甚至整个系统内存。他们提出了一种新技术，称为记忆伏击（memory ambush）。它能使可Hammer的双重拥有的内核缓冲区与目标对象（例如，页表）物理上相邻，并且只占用少量内存。

此漏洞本质是**物理内核隔离的内存所有权问题**，即最初为内核分配了一块内核内存，但后来映射到用户空间，允许用户进程访问内核内存，从而避免额外的数据从用户复制到内核，反之亦然。这种内存所有权的变化使物理内核隔离失效，使内核仍然可以 Hammer。 对于 CATT 概念本身，如果其在实践中的部署没有仔细考虑现代操作系统中的性能优化，那么物理域隔离也是不安全的，因此可能存在类似的内存所有权问题。

> 现代 CPU 采用多级缓存来有效减少内存访问时间。如果数据存在于 CPU 缓存中，则访问它将由缓存完成，并且永远不会到达物理内存。因此，必须刷新 CPU 缓存才能 Row Hammer。

x-Realism 的改进

动态分页

结合 Hammer 和 CATTmew 的原理，可以采取**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。操作系统在收到页表超出范围的异常后，再将新页表加载到内存中。

页表分离

将内核态使用的页表与用户态使用的页表隔离，避免用户态拥有页表所有权。

相关工作

微内核操作系统

Redox

Homepage,GitLab Repo

> **Redox** is a Unix-like Operating System written in **Rust**, aiming to bring the innovations of Rust to a modern microkernel and full set of applications.

文档非常详细，有一本书 Redox book 详细介绍了设计的思路，可以作为参考。

其主要特性如下：

- 类 Unix 操作系统，包括常见的 Unix 命令
- **微内核设计**
- 包括可选的 GUI 程序 - Orbital
- 支持 Rust 标准库
- 用 Rust 编写的 *libc*
- 驱动运行在用户空间

微内核设计的 Rust 内核较少，该项目当前仍在积极维护并且有详细的文档，可参考性**很高**。

Tock

Homepage, GitHub Repo

是一个**嵌入式操作系统**，主要特性如下：

- 低功耗
- **微内核设计**
- 占用资源少
- 扩展性强
- **并发**
- **隔离性好**

由于主要面对嵌入式领域，特性明显是针对嵌入式设备的特点，但依然可以参考其并发和隔离的部分，可参考性**较高**（文档同样不够详细）。

Rust 操作系统

rCore

GitHub Repo

THU 的操作系统项目，基于上面说的 BlogOS，相对更贴近大作业项目的水平。特点：

- 支持多种架构（包括 **x86**，但是没有 ARM）
- 模块化内存管理和进程管理
- 支持多核调度

中文项目，文档详细，可参考性**高。**

Rust for Linux

GitHub Repo

一个 GitHub 组织，得到了 Google 的大力支持，致力于提供 Rust 对 Linux 内核的各种支持.

> 2021年7月4日，Linux内核团队发布添加Rust支持的“v1”补丁；
>
> 2021年12月6日，Linux内核团队发布支持Rust 的“v2”补丁。

在 2021 年底，Rust 已经成为 Linux 开发的官方语言。据报道，到 2022 年，开发者有望看到 Linux 内核正式支持 Rust。截止 2022 年 4 月 3 日该仓库已经发布了 "v5" 补丁。

由于我们的目标并不一定要实现一个类 Unix 操作系统，更不可能实现一个完整功能的 Linux 系统，可参考性**一般。**

Red Leaf

GitHub Repo

> RedLeaf is a research operating system developed from scratch in Rust to explore the impact of language safety on operating system organization.

文档不详，近期（3个月内）没有维护，可参考性**一般。**

其他可参考的工作

Theseus

Homepage, GitHub Repo

同样有详细的文档，主要特性如下：

- 内核架构：既不是宏内核也不是微内核，被称作 Safe-language SAS/SPL OS

  官方对该架构的解释是

> Theseus is a safe-language OS that runs all components within a Single Address Space (SAS) and Single Privilege Level (SPL).

与其他内核对比的优势可以参考 Theseus Book，简单概括一下就是该操作系统的结构设计不在任何方面依赖于底层硬件的结构。所有内容（包括应用程序，系统服务和核心内核组件）都存在并在单个地址空间和单个权限级别中运行（在"内核空间"中）。Theseus 的结构纯粹是软件定义的，并基于 *cell*（该项目模块的叫法）的模块化概念。因此，通信和共享内存访问是**高效**的，因为编译器确保了隔离和保护。

- **高性能**
- **虚拟内存映射到物理内存**
- 支持 **multitask**，多个任务可以并发执行，并共享 CPU 和内存资源。

在其文档中专门介绍了内存管理和任务管理两个模块，可参考性**高**。

BlogOS

Homepage, GitHub Repo

一个博客系列，每篇博客有教程和完整的代码，对应 GitHub Repo 里的一个个 branch。主要特性如下：

- **中断**（同时介绍了硬件中断）
- 段页式内存管理
- **multitask**

每个 branch 有直接能跑的轮子，可参考性**高**

Kerla

Homepage, GitHub Repo

仓库介绍:

> Kerla is a monolithic operating system kernel written from scratch in Rust which aims to be compatible with the Linux ABI, that is, it runs Linux binaries without any modifications.

目标是在 ABI 层面上兼容 Linux，主要特性如下：

- 进程管理，包括上下文切换，`fork`，`execve`，`wait` 等
- 常见的系统调用
- 伪文件系统
- tty 和伪终端
- **x86 架构**

一个有意思的点在于我们可以直接通过 ssh 进入 Kerla，无需部署就可以看看别人做出来的内核是什么样的。

文档不够详细，可参考性**较高。**

参考文献

操作系统对比和未来展望

Windows 有哪些令人无法忍受的缺点?

Advantages and disadvantages of Windows operating system

macOS Review: The Pros and Cons

Linux系统有哪些缺点?

Ownership is Theft: Experiences Building an Embedded OS in Rust

Linux Plumbers: 想在内核里引入Rust，还需要做很多决定

世界上最著名的操作系统是用什么语言编写的?

写操作系统只能用汇编和 C 语言吗?

Rust no-std 工程实践

用 Rust 开发 Linux，可行吗?

OS Development in Rust

Rust OS comparison

新增3.2万行代码，Linux内核有望在2022年正式支持Rust

如何用 Rust 编写一个 Linux 内核模块

开源项目：使用 Rust 写一个兼容 Linux 的内核

Rust is now an official language for Linux development

CATTmew: Defeating Software-only Physical Kernel Isolation

Memory management unit From Wikipedia

Operating System - Memory Management

rCore-Tutorial-Book 第三版

分页存储管理：分区式存储管理最大的缺点是什么？

Microkernel Wikipedia

seL4 白皮书

Extreme High Performance Computing or Why Microkernels Suck

Mach Wikipedia

Minix Wikipedia

RT-Thread Smart 微内核操作系统

Communication in Microkernel-Based Operating Systems

Fuchsia 操作系统