可行性报告

目录

项目介绍

我们的项目是*基于 Rust 改造的 seL4 微内核*。

L4 是一种微内核构架的操作系统内核。seL4 是 L4 微内核系列的一种，经 Haskell 形式化验证并实现为 C。

我们计划将它改写为 Rust，进一步提升安全性。

理论依据

Rust 提供与底层语言相当的性能

在性能上，具有额外安全保证的代码会比 C++ 慢一些，但是如果以 C++ 也手工提供保证的情况下，则两者性能上是相似的。

Rust 的语法设计和 seL4 使用的 C 语系相似

Rust的语法设计，与 C语言和 C++ 相当相似，区块 (block) 使用大括号隔开，流程控制的关键字如 if, else, while 等等。在保持相似性的同时，Rust 也加进了新的关键字，如用于模式匹配 (pattern matching) 的 match (与 switch 相似) 则是使用 C/C++ 系统编程语言的人会相对陌生的概念。尽管在语法上相似，Rust 的语义 (semantic) 和 C/C++ 非常不同。

Rust 原生保证内存安全

为了提供存储器安全，它的设计不允许空指针和悬空指针。 数据只能透过固定的初始化形态来建构，而所有这些形态都要求它们的输入已经分析过了。 Rust 有一个检查指针生命期间和指针冻结的系统，可以用来预防在 C++ 中许多的类型错误，甚至是用了智能指针功能之后会发生的类型错误。

Rust 使用智能指针来进行内存管理

Rust 虽然有垃圾回收系统，但非如 Java 或 .NET 平台的全自动垃圾回收。Rust 1.0已不再使用垃圾回收器，而是全面改用基于引用计数的智能指针来管理内存。

Rust 提供更好的类型与多态

它的类型系统直接地模仿了 Haskell 语言的 type class 概念，并把它称作“traits”，可以把它看成是一种 ad hoc 多态。Rust 的作法是透过在宣告类型变量 (type variable) 的时候，在上面加上限制条件。至于 Haskell 的高端类型变量 (Higher-kinded polymorphism) 则还未支持。

类型推导也是 Rust 提供的特性之一，使用 let 语法宣告的变量可以不用宣告类型，亦不需要初始值来推断类型。但如果在稍后的程序中从未指派任何值到该变量，编译器会发出编译时 (compile time) 错误。函数可以使用泛型化参数 (generics)，但是必须绑定 Trait。没有任何方法可以使用方法或运算符，又不宣告它们的类型，每一项都必确明确定义。

Rust 的对象系统是基于三样东西之上的，即实现 (implementation)、Trait 以及结构化数据 (如 struct)。实现的角色类似提供 Class 关键字的编程语言所代表的意义，并使用 impl 关键字。继承和多态则透过 Trait 实现，它们使得方法 (method) 可以在实现中被定义。结构化数据用来定义字段。实现和 trait 都无法定义字段，并且只有 trait 可以提供继承，藉以躲避 C++ 的“钻石继承问题”(菱型缺陷)。

Rust 的开发生态友好

Rust在制作之初就非常关注它的易用性。Rust的文档齐全，编译器友好且能够给出有用的错误信息，并且有完善的包管理/构建工具。Rust还提供了用于编辑器的智能工具，包括自动补全、自动格式化、类型检查等等。

seL4 提供低耦合度的系统结构

作为微内核，它允许服务各自独立，减少系统之间的耦合度，易于实现与调试，也可增进可移植性。它可以避免单一组件失效，而造成整个系统崩溃，内核只需要重启这个组件，不致于影响其他服务器的功能，使系统稳定度增加。同时，操作系统也可以视需要，抽换或新增某些服务行程，使功能更有弹性。

seL4 通过形式化验证

seL4的实现被证明是bug-free（没有bug）的，比如不会出现缓冲区溢出，空指针异常等。还有一点就是，C代码要转换成能直接在硬件上运行的二进制代码，seL4可以确保这个转换过程不出现错误，可靠。seL4是世界上第一个（到目前也是唯一一个）从很强程度上被证明是安全的OS。

seL4 源代码行数少

seL4靠8700行C代码提供操作系统所必需的一切服务，如线程，IPC，虚拟内存，中断等，容易改写和debug。

技术依据

seL4 内核机制

Capability

Capability是用于提供访问系统中对象权限的凭证。seL4系统中所有资源的capability在启动时都被授予根进程。要对任何资源进行操作，用户都需使用libsel4中的内核API，并提供相应的capability。

CNode

CNode (capability-node) 是用于储存capability的对象，其中每个CSlot (capability-slot) 可为full或empty，分别对应是否拥有相应capability。CSlot的数量必须为2的整数次方倍。

CSpace

CSpace (capability-space) 是线程拥有的capability的集合，由一个或多个CNode组成。根进程的CSpace包含所有由seL4拥有资源的capability，在系统启动时被授予。

Untyped 内存

除了内核拥有的少量固定大小的内存，所有物理内存都在用户空间被管理。seL4内核空间外的所有物理内存资源的capability都被转移给根进程。这种capability被称为untyped内存。untyped内存是一块特定大小的连续的物理内存，可能为设备内存或RAM内存，两者在赋予类型时有区别。

Retype

untyped capability有调用seL4_Untyped_Retype，用于创建一个新的capability，来获取对untyped内存一部分的使用权，并对其赋予类型。

线程

TCB

seL4中的线程对象称为Thread Control Block。包括以下数据：

- 优先级 (0-255)
- 寄存器状态和浮点数环境
- CSpace capability
- VSpace capability (与虚拟内存有关)
- endpoint capability (用于发送错误信息)
- reply capability

调度模型

seL4使用基于优先级的循环调度器。调度器会找出优先级最大的未被阻塞的线程来执行。若有多个优先级相同的进程，将会以先到先得的顺序顺次执行。为了提供确定性的调度，可以将使用域调度。域调度是非抢占式的，线程可以与域关联，这样这些线程只有当此域是活跃时会被调度，而跨域IPC只有当域切换时才会执行。

IPC

作为微内核，seL4大量使用进程间通信。在seL4中，IPC借助称为endpoint的内核对象。endpoint对象上的调用可以收发同步的IPC消息。seL4提供了seL4_Send、seL4_NBSend、seL4_Recv、seL4_Call等系统调用在endpoint上进行进程间通信，调用这些系统调用需要相应的capability。IPC可以用于传输数据，还可以用于进程间的capability转移。由于微内核对IPC的高依赖性，seL4提供了一套优化的IPC方法。

通知

seL4借助通知对象的capability进行异步的信号收发。通知对象包含一组二元信号量，和等待通知的线程队列。通知对象可以为Waiting、Active、Idle三种状态。seL4提供了seL4_Signal、seL4_Wait、seL4_Poll系统调用以进行异步的通信。

Rust 与 C 交互

必要性

很多大型的软件项目都是使用多种程序设计语言写成的，比如操作系统往往是用汇编和C写成的，而web项目往往有前端和后端之分并使用了不同的语言。将不同的语言混合在一起使用，并不是什么坏事；恰恰相反，不同的编程语言往往有着不同的特性，“术业有专攻”，混合在一起可能能达到更好的效果。

seL4的内核源码也是用C和汇编写成的。(https://github.com/seL4/seL4) 如果我们需要用rust改写这个内核，有两种可能的方案，一是保留一部分C的内容的同时改写另一部分；二是将所有C代码用rust改写。第二种方案在难度上比较大，并不一定能实现，而且很多底层操作（比如boot和内存管理）使用C比unsafe Rust要更方便和自然。而如果在保留一部分C的内容的前提下用Rust改写，就会需要Rust与C的交互。

需要Rust与C交互还有另外一个原因，就是操作系统需要一定的应用程序接口(API)。L4是一个通用性的操作系统内核，在改写它的时候我们必须要保留好内核和其上的应用程序的接口。这种接口不仅仅是包括系统调用，还包括以库的形式提供的API。C语言是远比Rust用得广泛的多的系统编程语言，因此操作系统不可避免地与C应用程序打交道。如果我们能够以C语言可以直接调用的形式给出API，就可以把很多建立在L4上的应用程序移植到改写的代码上来。

可行性

Rust与C交互的可行性有两方面，一是语言层面上的可行性，二是工具层面上的可行性。

语言层面上的可行性，也就是在代码中进行交互的语法细节，在后面会详细给出，这里就不再赘述了。

工具层面上的可行性，就是说借助已有的构建工具可以把Rust代码和C代码放在一起进行编译、链接变成可执行的程序。由于一些历史原因，C语言不存在统一的管理和构建工具，在Unix系统上非常常见的构建工具就是Make。seL4使用了一种跨平台的构建工具CMake，它可以根据所用平台的不同，生成对应的makefile或其他文件。而Rust语言存在一个统一的包管理工具cargo，它同时也被用作Rust程序的构建。

可以看到，C和Rust在构建的工具上存在巨大的差异。所幸的是，Make和CMake足够灵活，我们还是可以比较容易地实现Rust和C程序的构建。如果使用Make的话，因为makefile像shell一样可以随意调用系统中的命令，所以可以直接用cc和rustc进行编译（比如`$(CC)` `$(RUSTC)`），再进行链接等操作。如果使用CMake的话，也有一些办法把cargo集成到CMake中去。https://stackoverflow.com/questions/31162438/how-can-i-build-rust-code-with-a-c-qt-cmake-project 提供了把cargo作为一个外部项目衔接到CMake中去的方案；https://github.com/AndrewGaspar/cmake-cargo 是一个可以把cargo中的包(叫做crate)集成入已有的CMake项目的工具。借助这些办法，在seL4已有代码和CMakeList的基础上添加和改写Rust程序就变得可行了。

FFI 与 ABI

FFI是外语言函数接口(Foreign Function Interface)的简称。就像它的名字一样，FFI用于在一个程序语言的函数中调用另一个语言的函数。这一功能保障了不同程序语言之间能够自由地交互，从而可以把不同语言的代码像积木一样搭建起大型程序。

在操作系统的层面上，有时还需要ABI。ABI是应用二进制接口(Application Binary Interface)的简称。ABI是从底层细节的层面给出应用程序和操作系统、应用程序之间等等进行交互的具体规范。统一的ABI是实现FFI的基础，保证不同语言程序编译出的二进制机器码在数据类型、调用约定、系统调用方法等层面上保持一致，就保证了不同语言的程序进行互相调用的可行性。但由于ABI较为底层，所以往往特定的操作系统/体系结构上有特定的ABI。

由于C作为系统编程语言的地位，很多ABI是以C ABI的形式给出的：即为C语言提供机器码层面上的规范，而让其他语言遵守这个相同的规范。Rust也是这样，当需要调用C函数或为C函数所调用时，通过特定的语法启用C ABI，保证编译出来的函数可以为C语言或其他语言所调用。

语法

调用函数

在Rust官方教程the book中简短地提及了调用C函数的FFI语法(https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html#calling-an-unsafe-function-or-method)。

粗略地说，Rust语言提供了`extern`关键字，用来标识相关的函数能够用于同其他语言进行交互。当我们需要把Rust函数拿给C程序使用的时候，可以在函数定义前面加上`extern "C"`，并使用`#[no_mangle]`标识来保证编译器不会修改函数的名字。示例的代码如下：

```
fn main() {
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
}
}
```

而如果要把C程序拿给Rust使用，会稍微麻烦一点。我们需要用`extern "C"`标识出一个块，在块内给出C函数的声明；由于C语言对代码的限制很少，不像Rust为了保证程序的安全性而做出了很多限制，直接调用C函数可能会扰乱Rust的这些限制。因此，Rust规定，调用C函数时必须在unsafe块中进行。示例的代码如下：

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

数据类型

Rust和C在数据类型上存在一定的区别。然而，很多数据类型是可以对应起来的。

- 基本数据类型：`std::os::raw`库中给很多C中的基本数据类型做了对应，包括整数、浮点数和字符类型。不过很多基本的数据类型从名字上就可以对应起来，如（通常而言）32位有符号整数类型`i32`直接对应`int`或者`int32_t1`，无符号的`u32`直接对应`unsigned int`或者`uint32_t`。
- 字符串：Rust中的字符是Unicode字符，不能直接对应到C中的字符。Rust中的字符串结构与C风格的字符串结构更是大相径庭。为此，`std::ffi`库中提供了两种不同的字符串类型`CStr` `CString`，前者用于从C到Rust转换，后者用于从Rust到C转换，具体细节可以参考https://doc.rust-lang.org/stable/std/ffi/index.html 。
- 数组与指针：C语言的指针和Rust中的裸指针(raw pointer)是直接对应的，比如`*mut c_uint`对应`uint32_t *`。由于裸指针不受所有权等规范的限制，所以也需要在`unsafe`块中进行操作。C语言在函数中传递数组其实就是传递指针，而要转换成Rust中的数组切片可以使用`slice::from_raw_parts`函数转换。`void*`指针可以利用`std::ffi::c_void`对应`void`来实现。
- 结构体和枚举体：C语言和Rust都有结构体。从ABI层面去看，结构体其实只是将若干类型的数据按照一定的顺序存在一起，因而约定了结构体的内存布局就能把C和Rust的结构体对应起来。为了让Rust中的结构体的内存布局遵守C ABI，需要使用`#[repr(C)]`进行标识，之后即可直接互相传递。枚举体也是同样使用这个标识。
- 复杂数据类型：对于比较复杂的数据类型，比如包装了高级数据结构的结构体，可以在不暴露内部

结构的前提下使用指针进行传递。具体而言，我们在Rust中定义一个具体的结构体`xxx`，并给出几个传递指针进而操作这个结构体的函数，并在C语言中声明`struct xxx;`而不给出其内部结构(这似乎叫做不透明数据类型)，通过给那些操作函数传递指针就能达到传递复杂数据类型的效果。

unsafe

可以看到，由于C语言自身的性质使然，它对于很多“危险”的操作，包括可能越界的数组访问、对任意指针进行解引用、在各种类型之间进行转换等等，都没有任何的限制，而是在C语言标准中规定了某些情况属于“未定义行为(Undefined Behavior)”。但Rust在设计之初就希望对很多会带来不可预知后果的行为进行编译期的限制，这就带来了C和Rust的不同。为了实现Rust与C的交互，需要将很多“危险”的行为放在unsafe块中，标识出“这是不安全的”。

那么，是不是与C交互的行为就是不安全的、需要尽量避免的呢？并不是这样的。在官方的文档中(https://doc.rust-lang.org/stable/nomicon/safe-unsafe-meaning.html)指出，Rust其实是分为了safe和unsafe两部分，unsafe代表的意义是由程序员自己而不是编译器来保证程序的安全。在很多情况下，unsafe是难以避免的，因为编译器的能力是有限的。有一个典型的例子：编写将可变切片`&mut [T]`切分为两块的函数`split_as_mut`，这在safe Rust中无法实现，因为可变的引用只能转移而不能存在两个。这体现出了Rust编译器的能力有限：因为切分后的两片是不相交的，各自成为一个可变引用并没有什么问题。为了让编译器信任我们能保证程序的安全，我们引入unsafe块来实现这个函数：

```
fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32])
{
    let len = slice.len();
    let ptr = slice.as_mut_ptr();
    assert!(mid <= len);
    unsafe {
        (slice::from_raw_parts_mut(ptr, mid),
         slice::from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

类似地，在标准库中存在着一些利用了unsafe实现的函数（比如，为了能够高效地使用所有权关系模糊地链表数据结构，需要引入unsafe块），这并不意味着它们就是不安全的。Rust允许我们在函数内部使用unsafe块，而把这个函数作为safe函数来调用。

在与底层打交道的操作系统里，不安全的行为比比皆是。除了和操作系统中的C代码交互以外，内存的分配和管理等操作本身也是明显“不安全”的。在用Rust改写曹祖系统内核的过程中，我们会不可避免地和unsafe打交道。（比如纯利用Rust编写的操作系统Redox，它的代码实现里就出现了很多的unsafe）