# 项目背景

计算机发展至今，随着摩尔定律，硬件发展已经达到了一些瓶颈。从而诞生了一些问题：软件的演进速度大大低于硬件的演进。软件在语言级别上无法真正利用多核计算带来的性能提升。而Rust语言是针对多核体系提出的语言，并且吸收一些其他动态语言的重要特性，比如不需要管理内存，比如不会出现Null指针等等。

同时Rust语言系统设计于保证内存安全，它在安全代码里不允许空指针，悬垂指针和数据竞争。数值只能用一系列固定形式来初始化，要求所有输入已经被初始化。在其它语言中复制函数指针或者有效或者为空，比如在链表和二叉树等数据结构中，Rust核心库提供Option类型，用来测试指针是否有值。Rust同时引入添加语法来管理生命周期，而且编译器通过租借检查器来说明相关理由。这些都保证了Rust语言极高的安全性。

在嵌入式领域中，嵌入式实时操作系统正得到越来越广泛的应用。采用嵌入式实时操作系统(RTOS)可以更合理、更有效地利用CPU的资源。FreeRTOS则是一个轻量级的操作系统，可基本满足较小系统的需要。而现今嵌入式系统不断得到更加广泛的使用，人们对于RTOS的可靠性和安全性也有了更高的要求。

# 立项依据

## C语言安全性问题与Rust的可靠性

众所周知，众多操作系统内核都是由C语言编写而成的，但是由于设计原因，C语言有灵活高效的指针操作，但是这些使得它的安全性不能保证，主要体现在：

- 空指针引用（**NULL Dereference**）
- 释放内存后再使用（**Use After Free**）
- 返回悬空指针（**Dangling Pointers**）
- 超出访问权限（**Out Of Bounds Access**）

声名狼藉的程序分段错误（Segmentation Fault）是C语言的常见问题，而通常NULL dereferences是第一大诱因。如果开发者忘记了检查所返回的指针是否正确性，就可能会导致空指针引用。Rust处理这类指针错误的方式非常极端，在“安全”代码中粗暴简单地禁用所有裸指针。此外在“安全”代码中，Rust还取消了空值。

像C++一样，Rust也使用资源获取即初始化（Resource Acquisition Is Initialization）的方式，这意味着每个变量在超出范围后都一定会被释放，因此在“安全的”Rust代码中，永远不必担心释放内存的事情。但Rust不满足于此，它更进一步，直接禁止用户访问被释放的内存。这一点通过Ownership规则实现，在Rust中，变量有一个所有权（Ownership）属性，owner有权随意调用所属的数据，也可以在有限的lifetime内借出数据（即Borrowing）。此外，数据只能有一个owner，这样一来，通过RAII规则，owner的范围指定了何时释放数据。最后，ownership还可以被“转移”，当开发者将ownership分配给另一个不同的变量时，ownership就会转移。

C语言老手都知道，向stack-bound变量返回指针很糟糕， 返回的指针会指向未定义内存。虽然这类错误多见于新手，一旦习惯堆栈规则和调用惯例，就很难出现这类错误了。事实证明，Rust的lifetime check不仅适用于本地定义变量，也适用于返回值。与C语言不同，在返回reference时，Rust的编译器会确保相关内容可有效调用，也就是说，编译器会核实返回的reference有效。即Rust的reference总是指向有效内存。

另一个常见问题就是在访问时，访问了没有权限的内存，多半情况就是所访问的数组，其索引超出范围。这种情况也出现在读写操作中，访问超限内存会导致可执行文件出现严重的漏洞，这些漏洞可能会给黑客操作你的代码大开方便之门。著名的就是Heartbleed bug问题。

# Rust并发模型

对于一个操作系统，我们有相当关注这些问题：异步、无锁、并发。

Rust语言项目初始是为了解决两个棘手问题：

1. 如何进行安全的系统编程？
2. 如何实现无痛苦的并发编程？

最初，这些问题似乎是正交的不相关，但是让我们惊讶的是，最终解决方案被证明是相同的：同样使Rust安全的工具也帮助你正面解决并发。

内存的安全错误和并发错误往往归结为代码访问数据引起的问题，这是不应该的。Rust秘密武器是ownership，系统程序员需要服从的访问控制纪律，Rust编译器也会为你静态地检查。

对于内存安全，意味着你在一个没有垃圾回收机制下编程，不用害怕segfault，因为Rust会抓住这些错误。

对于并发，这意味着你可以选择各种各样的并发范式（消息传递、共享状态、无锁、纯函数式），而Rust会帮助你避免常见的陷阱。

下面是Rust的并发风格：

- channel只传送属于其的消息，你能从一个线程发送指针到另外一个线程，而不用担心这两个线程因为同时访问这个指针产生竞争争夺，Rust的channel通道是线程隔离的。
- lock知道其保护数据，当一个锁被一个线程hold住，Rust确保数据只能被这个线程访问，状态从来不会意外地被分享，"锁住数据，而不是代码" 是**Rust**特点
- 每个数据类型都能知晓其是否可以在多线程之间安全传输或访问，Rust增强这种安全用途；也就没有数据访问争夺，即使对于无锁的数据结构，线程安全不只是文档上写写，而是其实在的法律规则。
- 你能在线程之间分享stack frames, Rust会确保这个frame在其他线程还在使用它时一直活跃，在**Rust**中即使最大胆的共享也会确保安全。

所有这些好处都是得益于Rust的所有权模型，和事实上锁、通道channel和无锁数据结构等之类的库包，这意味着Rust的并发目标是开放的，新的库包对新的范式编程更有力，也能捕获更多bug，这些都只要使用Rust的所有权特性来增加拓展API。

Rust的并发编程模型保证了：编译器阻止了所有的数据竞争

> A data race is any unsynchronized, concurrent access to data involving a write.

上面这句话中的同步包括底层的原子指令。它本质上说明了你不能在线程间意外地共享状态，状态的所有(可变)访问都需要以某种同步方式进行。

数据竞争只是一种(非常重要)竞争(race condition)，但是通过阻止它，Rust经常帮助你有效地阻止其他的，更加微妙的竞争。举个例子，把不同位置的数据更新弄成原子操作是很重要的：其他线程要么能看到所有更新，要么一个更新也看不到。在Rust中，同时拥有相关位置的`&mut`引用，将保证对他们的更新是原子的。因为不可能会有其他的线程能同时访问。许多语言通过垃圾回收来保障内存安全。但是垃圾回收并没有在阻止数据竞争方面给你提供任何帮助。

Rust则用ownership和borrowing来实现它的两个关键的价值观：

- 没有垃圾回收的内存安全
- 没有数据竞争的并发

# Rust的高效性

作为新生的编程语言，我们关注问题之一就是他的性能。虽然不能说Rust再所有环境和所有情况下都是高效的，但是绝大部分情况下，Rust都具有更大的优势。在Rust VS C++这里对于这两种语言的性能有一个小小的评测。此外，在girhub的lang-vs项目里有更为全面的比较，结果如下：

| Language | CPU time | Slower than | Language version | Source code | | | |
|---|---|---|---|---|---|---|---|
| User | System | Total | C++ | previous | | | |
| C++ *(optimized with -O2)* | 0.899 | 0.053 | 0.951 | – | – | g++ 6.1.1 | link |
| Rust | 0.898 | 0.129 | 1.026 | 7% | 7% | 1.12.0 | link |
| Java 8 *(non-std lib)* | 1.090 | 0.006 | 1.096 | 15% | 6% | 1.8.0_102 | link |
| Python 2.7 + PyPy | 1.376 | 0.120 | 1.496 | 57% | 36% | PyPy 5.4.1 | link |
| C# .NET Core Linux | 1.583 | 0.112 | 1.695 | 78% | 13% | 1.0.0-preview2 | link |
| Javascript (nodejs) | 1.371 | 0.466 | 1.837 | 93% | 8% | 4.3.1 | link |
| Go | 2.622 | 0.083 | 2.705 | 184% | 47% | 1.7.1 | link |
| C++ *(not optimized)* | 2.921 | 0.054 | 2.975 | 212% | 9% | g++ 6.1.1 | link |
| PHP 7.0 | 6.447 | 0.178 | 6.624 | 596% | 122% | 7.0.11 | link |
| Java 8 (see notes) | 12.064 | 0.080 | 12.144 | 1176% | 83% | 1.8.0_102 | link |
| Ruby | 12.742 | 0.230 | 12.972 | 1263% | 6% | 2.3.1 | link |
| Python 3.5 | 17.950 | 0.126 | 18.077 | 1800% | 39% | 3.5.2 | link |
| Perl | 25.054 | 0.014 | 25.068 | 2535% | 38% | 5.24.1 | link |
| Python 2.7 | 25.219 | 0.114 | 25.333 | 2562% | 1% | 2.7.12 | link |

# RTOS现今的需求

## 资源管理需求

使RTOS脱颖而出的是其管理资源(包括时间和存储器)的能力。时序问题与中断响应时间有关，但资源管理时序问题也会出现。虽然中断解决了一系列时序问题，但各应用仍必须利用资源。

考虑存储器分配情况。许多实时应用不采用动态存储器分配，以确保存储器分配和回收时所产生的不同不会变成一个问题。需要动态存储器分配的应用常把存储器划分为实时和非实时。后者处理动态存储器分配。典型情况下，在使用前，实时部分必须被分配有足够的存储器。

在实时嵌入式应用中采用C和C++是因为存储器和其它资源的用法是显式的。实时任务需要避免采用C和C++。特别是，当存储器分配和回收更容易隐藏时采用C++是很困难的。

像Java和C#这样的语言带来的挑战更大，它们与生俱来地采用动态存储器分配。程序员可控制存储器分配和回收。在某些情况下，编程环境可以强化存储器分配和回收。Java实时规范(RTSJ)定义了创建不需要垃圾回收的Java应用的方法。RTSJ是在Java框架内这样做的，从而使程序员在不被存储器分配限制的条件下享有Java的好处。

Sun和DDC-I都实现了RTSJ。DDC-I的实现支持x86和PowerPC平台。Aonix有一个称为PERC的类似平台。这些平台以实时、同时的垃圾回收为特征，从而使在不受存储器分配限制的情况下，在Java内编写实时应用成为可能。

但因系统必须允许线程为垃圾回收器进行转换，所以实时要求并非那么紧迫。另一方面，垃圾回收器将耗费时序资源，所以，只有实时任务方可保证满足一定的期限要求。快是好事，但及时才是RTOS的信条。考察实时平台时，考虑之一是存储器分配对系统的整体影响。许多系统可工作在从不改变的静态分配环境，但更多的动态系统可从实时垃圾回收中获益。研究表明，垃圾回收的效益与确定的存储器分配是可比的。

围绕诸如Java和C#等虚拟机类型平台的另一个问题是对just-in-time(JIT)编译器的使用限制。基于这些系统的实时系统必须采用类似C和C++等所用的提前（ahead-of time，AOT）编译器。

设计师会因其更高的生产力、更低的出错率以及安全性等特点选用Java 或C#。所以，对制定一个称为 JSR-302的用于对安全有至高要求应用的Java规范就不足为奇了。

## 安全性需求

RTOS受到其运行的硬件平台的限制。可对缺少存储器保护的硬件加以保护，但安全级别会受到限制。但存储器和虚拟机可以更高水平的安全性支持引导。诸如SE **Linux**、Green Hills Integrity和 LynuxWorks LynxSecure Embedded Hypervisor以及 LynxOS-SE RTOS内的安全策略可比典型RTOS提供可靠得多的保护。但成本也高，所以开发者需对此进行权衡。实时系统开发者不得不应对策略实现和边界问题。取决于信息的来所去处，安全支持会花很长时间。正是为此引入了分区系统，所以，可在边界采取安全措施且把应用的非实时部分放在这部分空间内。

# 参考


http://lambda-the-ultimate.org/node/4009

https://www.freertos.org/about-RTOS.html

https://tonyarcieri.com/would-rust-have-prevented-heartbleed-another-look