

# Rust kernel 结题报告

---

小组成员：陆万航，王浩宇，邱浩宸，雷婷，段逸凡

## 1. 课题概况

在物联网技术日益火爆的今天，嵌入式系统的安全性逐渐成为了人们所关注的焦点。在一般的嵌入式操作系统编程中，C语言凭借着自身跨平台性、兼容性好、容易嵌入汇编代码、运行速度快、使用者众多等优点，常常成为了编程者首选的语言。但安全性是C语言的一个致命缺陷，C编写的操作系统难免会产生很多问题，轻则系统崩溃，重则信息泄露，文件丢失。所以如何改善操作系统的安全性，成为了人们关注的重点。

### 1.1 调研工作

#### 1.1.1 嵌入式系统

嵌入式系统对实时性与安全性有着严苛的要求。

- 实时性：嵌入式系统是一个激励 - 运行 - 回应的电子系统。一方面，嵌入式应用系统有十分可观的激励 - 回应时间 $T_a$ ，导致系统实时能力的降低；另一方面，由于嵌入对象体系的多样性，复杂性，不同的对象体系会提出不同的回应时间 $T_a$ 要求。因此，在嵌入式应用系统的具体设计中，必须考虑系统中每一个任务运行时，能否满足 $T_s \leq T_a$ 的要求，这就是嵌入式系统的实时性问题。
- 安全性：社会发展日新月异，物联网离百姓生活越来越近，目前很多运行在局域网甚至Internet上的产品如雨后春笋般涌向市场，如智能家居、智能手机等。这些产品在方便用户的同时也出现一些安全问题，系统置于网络上相当于暴露给所有人，故对嵌入式产品安全性研究刻不容缓。嵌入式产品由于尺寸、成本的约束注定不可能从硬件部分提供更多的安全措施，故提升系统安全性应重点考虑内核方面的安全。

#### 1.1.2 Rust

经过了学期初的调研，我们了解到近年来一门新兴的语言——Rust。Rust 是一门年轻的语言，从创立之初，就瞄准了C和C++，从这两门语言中借鉴了许多优点，并规避了这两门语言的许多问题。在底层的内核开发方面，Rust比C语言更加注重安全性、高效性和并发性，成为了传统的C语言的一位有力的挑战者，有人也说，Rust是未来最有可能替代C和C++的一门语言。

Rust有许多优良特性值得我们关注。

- 安全性：Rust有极好的安全性，比如通过RAII机制实现的内存安全，绿色线程实现的并发安全，所有权实现的数据安全。
- 高效性：其次，Rust语言在运行时做的工作极少，把大量的工作留到编译时完成，这使得Rust语言具有高效性。
- 并发性：Rust 的内存安全功能也适用于并发环境，并发的 Rust 程序也会是内存安全的，并且没有数据竞争。Rust 的类型系统也能胜任对并发情形的处理，并且在编译时能提供强大而有效的方式去推论并发代码。

## 1.2 可行性分析

### 1.2.1 Rust对嵌入式操作系统开发的支持

- Rust通过asm!宏可以实现内联汇编，将变量与实际的寄存器关联起来。

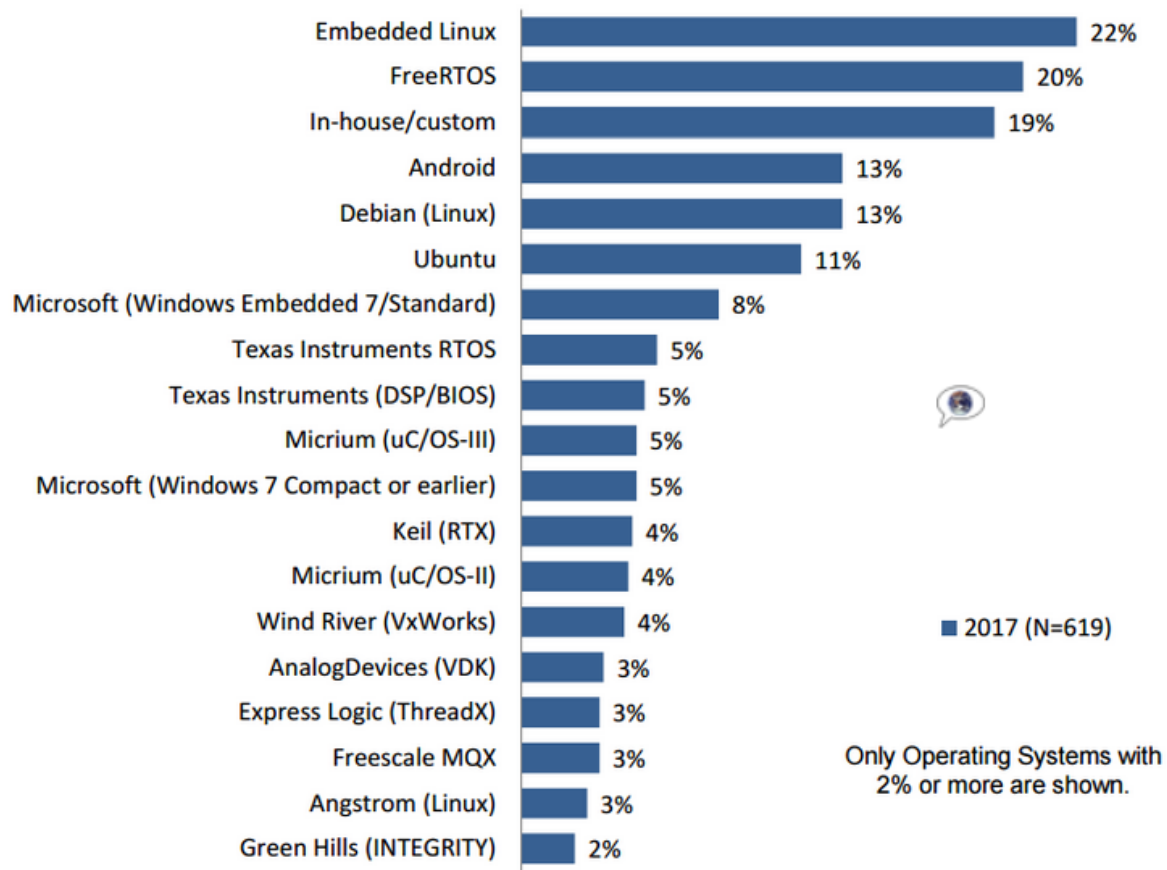
- Rust可以通过snappy库进行对C语言代码的调用，这样一来我们就可以利用C语言丰富的库函数进行相关工作，实现Rust和C的混合编程。
- 早在2016年的Rust 1.6版本，Rust就实现了libcore的稳定，这使得使用稳定的Rust进行OS和嵌入式开发成为可能。

1.2.2 改写操作系统的选择

在近一个月的调研和反复的讨论后，对比了现存的众多RTOS，并分析了其理论可行性和实际可操作性后，我们最终决定选择freeRTOS作为我们的参考源码。

freeRTOS有如下优点：

1. 免费且开源，在其官网上有所有版本的源代码下载，有助于我们增强对RTOS的理解，也便于我们对其进行裁剪，优化。
2. 被大量的嵌入式开发者认可，并且使用人数保持了逐年增长的趋势，根据统计数据显示，freeRTOS在全球的嵌入式系统开发语言排行榜中，在过去的几年内一直稳居前五，有若干年甚至达到了榜首，每年都约有20%的嵌入式开发者基于其进行开发。下图为2017年的统计数据：



2017年RTOS使用榜

图一：2017年开发者使用rust的比例

3. 文档齐全，作为一个开源项目，在其官网上可以找到所需的任何文档，这对于深入理解嵌入式操作系统有着巨大的帮助。
4. 相比于一些其他的大型嵌入式系统而言，freeRTOS作为轻量级OS较为简单，但是在嵌入式操作系统所遇到的绝大多数应用场景下，freeRTOS都可以满足需求。

### 1.2.3 技术路径选择

我们起初设想了三种可以选择的路径：

1. 将一切现有内核源码推倒重来，根据Rust自身的高安全性的特点来编写内核源码。
2. 根据已有的嵌入式系统的C语言源码，针对C语言的缺点来用Rust改写代码实现原来的功能，在已有的基础上重构内核。
3. 通过了解已有的嵌入式操作系统内核必要的模块，并利用Rust来重写这些函数和模块。

在经过综合考虑复杂度，难度，时间分配等各种方面因素后，我们选择了第二条路径。虽然方案二仍然存在着一些问题和难点，但是其相对较小的工作量、C和Rust之间易于实现的接口、带来的对于操作系统更为深刻的理解，使我们仍然倾向于方案二，即根据已有的嵌入式操作系统的C语言源码，通过针对C语言缺点和漏洞来编写代码，在已有的基础上重构内核。

### 1.2.4 具体实现

根据与导师的交流，我们利用了导师提供的Nordic nRF52840进行开发工作。

## 2. 工作总结

### 2.1 工作日程

- 第1周、第2周：选定了题目，决定使用Rust重写Freertos内核，并将其在开发板上运行。
- 第3周、第4周：查询资料，编写调研报告初稿。
- 第5周：对调研报告的格式和内容加以修改，形成第二稿。
- 第6周、第7周：查询资料，编写可行性报告。
- 第8周、第9周：准备中期汇报材料，并进行中期汇报。
- 第10周——第13周：熟悉Rust语法，掌握混合编程的技巧，了解Freertos的工作原理和具体实现，为正式开始工作做准备。
- 第14周：裁剪Freertos内核，生成一个能够在Windows平台上运行的Demo（用C语言编写），这是我们下一步改写参照的目标。同时，完成了用Rust语言编写主程序（调用C语言的API函数）的工作，但是程序执行时遭遇问题，报读写权限异常错误。
- 第15周：通过反复调试，找到了bug的原因，并加以解决。分配了7个API函数的改写工作，以及在尝试开发板上运行程序的工作。
- 第16周：完成了API函数的改写工作，并加以整合，形成了完整的Demo（用Rust和C语言混合编程）。开始结题汇报的准备工作。
- 第17周、第18周：完成了项目结题汇报的准备工作，包括演示文稿的制作、讲稿的编写以及汇报的排练，并进行了结题汇报。

### 2.2 团队工作

在初期的调研工作中，王浩宇和雷婷负责Rust方向的调研，邱浩宸和段逸凡负责Freertos方向的调研，陆万航负责Nordic开发板和相关工作方向的调研，最后集体开会整合成完整的报告。

在中期汇报的准备中，邱浩宸负责PPT的制作，陆万航负责上台进行汇报，其他组员对此提出了许多有益的意见和建议。

在项目的具体实现中，王浩宇负责C语言Demo的裁剪，以及Rust语言Demo的主函数的编写。对于7个API函数，王浩宇、雷婷、陆万航、邱浩宸分别负责1~2个函数的改写工作，最后由王浩宇负责整合，形成最终的Demo。段逸凡负责将写好的Demo移植到Nordic开发板上。

在结题汇报的准备中，陆万航负责PPT的制作以及上台汇报，其他组员也对PPT和讲稿做出了一定修改，对

汇报工作提出了许多有益的意见建议。在Q&A环节中，各个组员都回答了一些同学提出的问题。

在结题报告的写作中，段逸凡负责项目概况的编写，王浩宇负责团队工作和取得成果的编写，陆万航负责难点分析、不足与改进和未来规划的编写，雷婷、邱浩宸为报告的写作整理了素材，并提出了宝贵的建议。

### 3. 取得成果

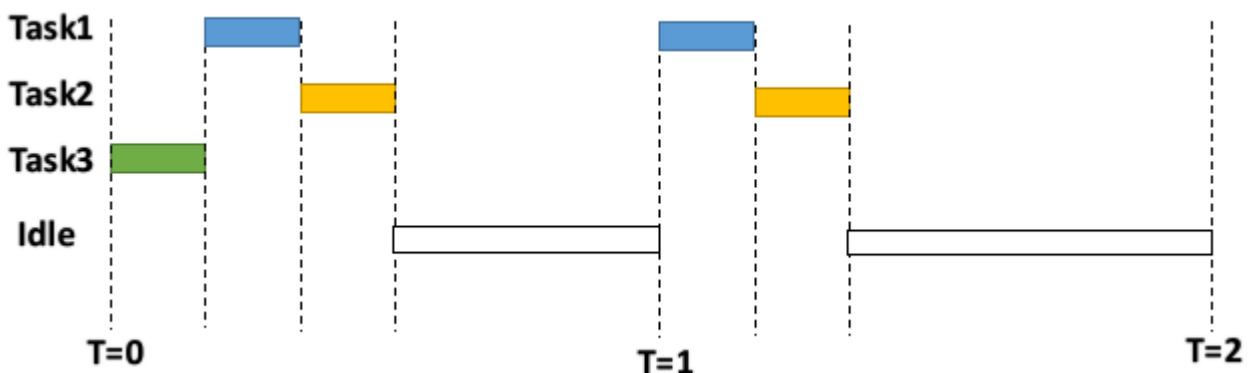
在所有组员的共同努力下，我们完成了使用Rust语言改写Freertos内核的工作，改写了7个API函数：

xTaskCreate(): 任务创建函数  
vTaskStartScheduler(): 进程调度启动函数  
vTaskPrioritySet(): 任务优先级设置函数  
uxTaskPriorityGet(): 任务优先级获取函数  
vTaskDelay(): 任务延迟函数（相对延时）  
vTaskDelayUntil(): 任务延迟函数（绝对延时）  
vTaskDelete(): 任务删除函数

同时，我们编写了一个Demo。在Demo中，我们创建了3个任务。其中，任务1和任务3都是在主函数中创建的，任务2是由任务1创建的。通过这三个任务，我们使用了前述的七个API。

任务3在创建时的优先级最高，会最先运行。但它的功能是删除自身，所以只输出一句话，之后就不再出现。任务2由任务1创建，这两个任务的优先级交替提升，因此都有执行的机会，不会饿死。

任务的执行流程如下：



图二：demo进程调度图解

主函数创建了两个任务——Task1和Task3，并启动了任务调度函数。代码如下：

```
fn main() {
    let name=CString::new("Task1").unwrap();
    xTaskGenericCreate(vTask1,name.as_ptr(),1000,ptr::null_mut(),2,ptr::null_mut(),ptr::null_mut(),ptr::null());
    xTaskGenericCreate(vTask3,CString::new("Task3").unwrap().as_ptr(),1000,ptr::null_mut(),3,ptr::null_mut(),ptr::null_mut(),ptr::null());
    vTaskStartScheduler();
    loop {

    }
}
```

Task1创建了任务Task2（只执行一次），并在输出语句后将Task2的优先级提高，使其抢占系统资源，即将控制权移交给Task2。代码如下：

```
extern fn vTask1(_pvParameters: *mut c_void){
    let uxPriority:c_ulong;
    let mut xTask2Handle: xTaskHandle=ptr::null_mut();
    /* 本任务将会比任务2更先运行，因为本任务创建在更高的优先级上。任务1和任务2都不会
    阻塞，所以两者要
    么处于就绪态，要么处于运行态。
    查询本任务当前运行的优先级 - 传递一个NULL值表示说“返回我自己的优先级”。 */
    uxPriority = uxTaskPriorityGet( ptr::null_mut() );
    xTaskGenericCreate(vTask2,CString::new("Task2").unwrap().as_ptr(),1000,ptr
::null_mut(),1,&mut xTask2Handle,ptr::null_mut(),ptr::null());
    /*任务2的创建*/
    loop{
        /* Print out the name of this task. */
        print!( "Task1 is running\r\n" );
        /* 把任务2的优先级设置到高于任务1的优先级，会使得任务2立即得到执行(因为
        任务2现在是所有任务
        中具有最高优先级的任务)。注意调用vTaskPrioritySet()时用到的任务2的句
        柄。程序清单24将展示
        如何得到这个句柄。 */
        print!( "About to raise the Task2 priority\r\n" );
        vTaskPrioritySet( xTask2Handle, uxPriority + 1 );
        /* 本任务只会在其优先级高于任务2时才会得到执行。因此，当此任务运行到这里
        时，任务2必然已经执
        行过了，并且将其自身的优先级设置回比任务1更低的优先级。 */
        vTaskDelay(50);
    }
}
```

Task2和Task1类似，不过它在输出后降低自己的优先级，归还控制权。

```
extern fn vTask2(_pvParameters: *mut c_void){
    let uxPriority:c_ulong;
    let mut xLastWakeTime:c_ulong;
    /* 本任务将会比任务2更先运行，因为本任务创建在更高的优先级上。任务1和任务2都不会
    阻塞，所以两者要
    么处于就绪态，要么处于运行态。
    查询本任务当前运行的优先级 - 传递一个NULL值表示说“返回我自己的优先级”。 */
    uxPriority = uxTaskPriorityGet( ptr::null_mut() );
    xLastWakeTime = xTaskGetTickCount();//获取当前时间
    loop{
        /* 当任务运行到这里，任务1必然已经运行过了，并将本身务的优先级设置到高于
        任务1本身。 */
        print!( "Task2 is running\r\n" );
        /* 将自己的优先级设置回原来的值。传递NULL句柄值意味“改变我自己的优先
        级”。把优先级设置到低
        于任务1使得任务1立即得到执行 - 任务1抢占本任务。 */
        print!( "About to lower the Task2 priority\r\n" );
```

```

        vTaskPrioritySet( ptr::null_mut(), uxPriority - 2);
        vTaskDelayUntil(&mut xLastWakeTime, 50);
    }
}

```


Task3只运行一次，会删除自身。

```

extern fn vTask3(_pvParameters: *mut c_void){
    loop{
        print!("Task3 is running.\r\n");
        print!("I'm going to delete myself.\r\n");
        vTaskDelete(ptr::null_mut());
        print!("This sentence will never be printed.\r\n");
    }
}

```

实际运行这个Demo，运行结果如下：

 E:\我的程序\操作系统\X-rocker\demo\_rust\target\debug\demo\_rust.exe

```

Task3 is running.
I'm going to delete myself.
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority

```

图三：demo的运行结果

此外，我们还把改写后的Freertos内核移植到了Nordic开发板上。在开发板上运行的Demo只有两个进程，一个是控制LED灯亮的进程，一个是空闲进程，两个进程交替执行，具体表现为LED灯闪烁。

## 4. 难点分析

在实验过程中，rust本身的编程特性和我们的日常编程习惯之间产生冲突，这也造成了我们在不同的工作阶段遇到了不同的问题：

在工作的初期阶段，我们在一些数据结构的实现上遇到了问题。Rust对于变量的绑定是严格的一一对应的关系，即变量和数据是完全的一一绑定的关系，这也导致了在实现一些比较简单的数据结构的时候，我们对于编程的传统观念都被颠覆了。

首先，以链表为例，我们通常在c语言中写一个链表，就是先设置一个根节点，然后依次用头插法或者尾插



法来增加节点，在这个过程中需要反复用到根节点。但是在rust中，当你第二次使用这个变量的时候，rust的编译器会报错——这个根节点已经被访问过一次，内容已经被修改，所以重新访问可能会出错。

其次，rust中实现复杂的数据结构是非常麻烦的。在c语言中，定义两个结构体structA和structB，然后再通过指针互指，是相当寻常的事。但是到了rust中，我们这么做就会引发编译器报错，因为相互引用的对象会导致所有权的混乱。但在没有所有权概念的语言中，我们通常认为这种做法是合理而高效的。

此外，我们还遇到了全局变量中无法使用vector、空指针需要进行特殊定义等诸多问题。相比于几乎没有限制的C语言，Rust语言对初学者而言是令人痛苦的。

在项目的调研和规划阶段，我们就考虑到了rust语言的编写难度和整体巨大的工作量，因而决定采用rust和c混合编程，这是我们中期规划中的一个重点和难点所在。在进行了资料查找后，我们初步掌握了rust和c混合编程的方法：我们调用了FFI（外部函数接口），从而调用C中的函数，但Rust编译器无法保证C函数的安全性，所以我们需要将外部函数移入unsafe块来注明这一点，以通过编译检测。由于没有编译器的帮助，我们必须自行保证外部函数的安全性，在这里，我们默认这些函数是安全的。如果时间允许，我们会考虑将这些外部函数用rust重写。

最后，在整合收尾阶段，工程出现了非常严重的问题。在最终将用rust编写的函数整合成一个demo前，为了对比运行结果，我们首先裁剪出了一个c的demo，并且能够在windows平台下正常运行。但是用rust改写的功能完全一致的demo在同样的环境下运行会立刻崩溃。

由于小组成员在之前的编程中都从未见过这一类问题，所以整个工程在该阶段停滞了较长的时间。起初我们认为是rust调用FFI的时候引发的错误，所以做了一个极度精简版本。在这个版本里面，rust的main函数直接调用了C函数，这个函数既没有参数也没有返回值，相当于在c和rust间没有任何数据传递，但是这个程序也依旧会崩溃。经过不断的调试和猜测，我们发现，目前官方给出的rust的demo是基于32位系统的，但是我们自己编写的c和rust的demo都是64位的，而问题就出在程序位数上。在程序的读取内存的时候采用了如下的强制类型转换来获得数据：

```
pxThreadState = ( xThreadState *) ( ( unsigned long ) pvOldCurrentTCB );
```

原本的程序是32位的，却被rust的编译器当成了64位程序，但是程序的某些地方在进行类型转换的时候却仍然是以32位来处理。64位的数据被强制转换为了32位的无符号长整形，再强制转换为了其他类型，导致64位地址中的高32位地址被舍弃，只保留了低32位的地址。这也是程序能通过编译，但会在运行时崩溃的原因。因此，我们将上图中的long修改成long long，也就是把程序中的强制类型转换变成64位后，问题也就得到了解决。

至于为什么实现同样的功能，两种语言的程序有着截然不同的表现，我们猜测可能是c和rust的编译器的默认地址的问题。在编译时，c的虚拟地址较小导致高位全部为零，于是32位和64位的互相转换未受到影响；而rust的虚拟地址的高位则不为零，于是在转换时地址发生了改变，引发了错误。

在代码的编写过程中遇到的种种困难，使我们对程序设计有了一个颠覆性的认知，也使我们对我们的编程习惯有了一个新的认识和架构，对嵌入式系统和系统级编程都有了更深层次的理解和认知。

## 5. 不足与改进

本次大作业取得的成果是比较显著的，但是项目工程中仍然存在的一定的问题，以下是我们对现阶段的仍然存在的不足的总结和反思：

调用了大量c函数的接口，仍未能实现全rust的freeRTOS的复现。我们目前的工作实际上也是一个相对初步的过程，我们虽然实现了运行一个进程最重要的几个函数，但是对这些函数中调用的其他函数，我们选择了调用了c函数接口来实现。当然，对于这些c函数的改写也是相对简单的，因为其中一些关键的数据结构我们都已经基本实现了，所以对于进一步的编程来说，目前的工作已经打下了相对坚实的基础。

时间、精力，投入上的不足。由于rust确实是一门需要抛弃以往的编程习惯，一些在c、java等编程语言中忽略的细节在rust中都可能是致命的错误。而且由于相关资料的缺乏，所以我们对于rust编程的进展是比较缓慢的。尽管整个团队都投入了相当多的时间，最后仍然没有实现对于FreeRTOS的三个核心文件的代码重写。当

然，我们也计划将这个�项目整体开源，让更多的有志之士来参与进这个�项目，从而集众人之力来完成这个艰巨的任务。

安全性在实验中没有得到有效的检验。考虑到rust本身的高安全性和高效性，demo代码的安全性有所保障。但demo的代码实现中调用了rust的FFI接口，使用了unsafe块，这些接口的安全性有待检验。

其次，“安全”的Rust难以高效地表示复杂的数据结构，特别是数据结构内部有各种指针互相引用的时候，这也是实验中目前面临的一个矛盾的局面。复杂的数据结构内部常常伴随一些不安全的操作。实际上，在使用某些复杂的数据结构时，rust会采用一些不安全的方式，rust标准库中有不少也调用了unsafe模块来实现某些数据结构。所以rust如何去支持更加复杂的数据结构，同时保证安全性，这也是进一步扩大代码规模、研究下去的一个重要问题。这个看似矛盾的问题如何得到有效解决，也是值得开发者思考的问题。

当然，所有的程序都不能做到绝对的安全，所有的程序都存在着一定的风险性。在和助教讨论之后，我们得出了以下结论：Rust最大的特点就是将原来分散在各处的不安全模块集中到了unsafe的模块中，这也和管程的做法类似，将不安全的东西集中到一处，方便管理和调度，从而提升了系统的整体安全性。从这点来看，编写的demo的安全性和c相比，是得到了显著的提升的。

## 6. 未来进一步规划

### 6.1 对比与总结

我们的大作业取得了相当丰硕的成果，但和我们抱有的最终目标进行对比，仍然存在着一一定的距离。

首先，目前大作业编写的demo，采用c和rust混合编程的策略，实现了进程调度。虽然是一个比较显著的成果，但同时我们必须认识到，freeRTOS的整个代码工程量相当巨大，这注定了这是一个非常庞大的工程，也是一个相当艰巨的任务，需要我们更加深入的理解rust，去编写一个合适的程序。

其次，需要掌握的工具上也有不同。c语言是一种相对简单易懂的高级语言，由于有程序设计课程的基础，程序的编写也就相对轻松。但是rust的优势和难度都是显而易见的，而且rust对于复杂的数据结构的支持是相当不友好的，其学习曲线成指数函数型，这也导致了项目初期花费了大量时间却依旧进展缓慢的原因。

最后，我们也必须意识到时间上的问题，在课业压力较大的情况下，我们抽出时间来学习rust也是一个相当艰难的过程，由于时间紧，任务重，实际编程中也遇到了较大的困难，所以距离最终目标仍然有着距离。

尽管仍然存在着一一定程度上的不足，但是在漫长而艰辛的项目实现过程中，我们对FreeRTOS有了更深刻的理解，对于rust的编程特性也有了不少的掌握和认知，随着不断深入的钻研和理解，我们对于rust的编程学习应当也会逐步突破瓶颈，对系统层面的编程有一个更加深刻的理解，能够真正做出一个高安全性，高效率的嵌入式内核。

### 6.2 项目未来的走向和规划

由上文的对比总结可知，我们距离最终目标仍然存在着一相当的距离，但是本次大作业已经实现了核心部分，成功运行了demo，并进行了下载。但从目前项目小组成员的课业压力和日程安排来看，仅凭小组内部的力量是很难实现整个FreeRTOS的改写的。正如上文提到的，Rust重写FreeRTOS是一个相当漫长而艰巨的任务，因此我们需要利用开源社区的力量，将整个项目推广出去，集众人之力而成大事，从而最终实现用rust重写整个FreeRTOS的目标。