大家好！

我们组做的rust重写freeRTOS。我们的期末报告大致分为以下四个部分，首先我们先来看一下我们组取得的成果。我们的开发概况大概是这个样子的：我们的开发平台时Visual Stdio和VScode，因为在没有合适的IDE的情况下，Vscode有着对rust的语法的比较好的支持，能够简化我们的工作，此外，我们还将我们所写的东西下载到了老师提供的nordic开发板上。目前，我们采用的主要模式是c语言和rust的混合编程，而我们的最终目标依旧是利用rust完完整整的重写一遍freeRTOS，使其安全性得到有效的提升。

然后，这个是我们的一个工作的大概流程，首先，我们尝试这使用rust来重现了c中的一些比较重要的数据结构，当然也参照了freeRTOS中所使用的数据结构，主要有双向链表还有其他的一些结构体。之后，我们也掌握了在rust中调用FFI，也就是外部函数接口，掌握了用rust和c进行混合编程的技巧，这也是我们工程中的一次巨大的跃进，因为纯rust的编程对于现阶段我们的组员难度还是相对较大的。

由此，我们利用rust写了一个demo，能够实现一个嵌入式系统的基本功能，也就是进程的创建和调度，并且将其在开发板上成功运行。

当然，我们目前阶段虽然取得了比较显著的成果，但是我们依旧怀抱着我们的最终目标，用rust将freeRTOS的内核源码完全实现，当然这将是一个长期的过程，值得所有人去努力奋斗。

我们的demo实现了一下几个函数，当然这些函数里面嵌套调用的函数，我们也都大都实现了，部分采用了混合编程的策略。首先是进程的创建函数，然后

这张图大致展示了我们的demo的工作流程，由于为了方便演示，我们只采用了一个相对简单的模型，实际上我们的demo是可以进行更为复杂的调度的：

任务1和任务3都在主程序中创建，任务2由任务1创建。

任务3在创建时的优先级最高，会最先运行。其功能是删除自身，只输出一句话。

任务2由任务1创建，这两个任务的优先级交替提升。

我们组在利用rust编写程序上取得了一定的成果，我们利用rust重写了几个demo，实现了比较重要的功能，在这些demo的基础上，我们能够实现一个嵌入式操作系统的主要部分功能，从而给复写整个freeRTOS打下了非常良好的基础。以下是我们程序的一个演示：

这是我们取得的成果的演示，实际上，我们所付出的，去适应rust的编程习惯的远不止这些，我们面对的主要问题，就是rust本身的编程特性问题，正如老师上课所提到的，rust和我们通常使用的编程语言的编程风格相当不统一，我们日常在c的基础上学习java、python等语言，其实是相对轻松的，因为整个编程体系、语法结构都是相似的，但是深入学习rust这门编程语言后，我们发现rust的整个体系都是对传统编程的一个颠覆，甚至某种意义上可以说是带有一些“反人类”的特点。比如说，一些我们在c语言中见怪不怪的一些数据结构，到了rust中往往就需要我们从头适应专门适用于rust的编写方式。可以说，我们的学习的过程是一个和rust的编译器斗智斗勇的过程，我们深刻体会到，rust的学习曲线是这个样子的：

为此，我们也投入了相当多的时间去磨练自己的编程能力，同时，我们也对这个项目工程所遇到的困难进行了总结和反思：

1. 对于一些数据结构的实现。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，我们对于传统的编程的概念都被颠覆了，就拿链表来说吧，首先，在rust中，每一个变量在使用的时候我们我们通常在c语言中写一个链表，通常就是设置一个root根节点，然后以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，这样子看似稀松平常的做法也会给你一个error，在你第二次使用这个变量的时候，rust的编译器会告诉你，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。而且，rust中实现复杂的数据结构是非常麻烦的，比如说在c语言中，定义两个结构体structA，structB，然后再用指针互指，这是我们通常的做法，也是相当简便的，但是到了rust中，我们这么做就会引发编译器报错，因为在rust中，这种做法会被归结到不安全的那一类中去，而我们通常却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，这里我就不一一讲了，如果各位想体验一下rust的特色可以尝试着把程序设计二里的代码用rust去再写一遍，你就会对此有非常深刻的认识。

所有权转移的问题：报错的main函数将整个结构体作为参数传给了一个函数，这里是f3，又对这个结构体进行了访问，所以rust报告了所有权转移的错误，比如说这里“use of moved value” 就是说他使用了一个所有权已经发生转移的对象，这在rust中非常常见，这也是我们最初编程时容易碰到的问题，修改后的代码如图所示，我们不是传递整个结构体，而是传递了一个可变引用，这个时候就不会发生所有权转移，也就不会爆出刚才的错误。

1. 对于混合编程的实现

在我们的调研报告中，我们也已经说明了，考虑到rust语言的编写难度和整体巨大的工作量，我们也想到过用如rust和c来进行混合编程，在进行了资料查找后，我们也初步掌握了rust和c混合编程的方法：我们需要调用FFI，也就是外部函数接口，从而能够调用c中的函数，但是c中函数在rust中的编译很有可能无法编译通过，所以我们需要用unsafe模块，即不对函数内部做安全检测。当然，这种做法对安全性是否有所影响，这个确实有待检验，如果时间充裕，我们也会考虑进一步将这些调用的c函数用rust重写一遍，但是时间限制，我们只能先调用这些函数来实现一个功能比较全面的demo。

最后，无论是什么样子的bug带给我们的最终认知就是，对程序设计有了一个颠覆性的认知，可以说，我们日常被c等高级语言养出来的编程习惯，在rust中都要被颠覆。

1. 然后最后一个比较严重的问题，也是我们在整个工程中出现的一个非常严重的问题。在裁剪出一个demo的时候，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在同样的环境下非常容易崩溃，无法正常运行。由于我们组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，所以我们干脆做了一个极度精简版本，在这个版本里面，rust的main函数直接调用了一个C函数，这个函数没有参数且没有返回值，相当于我们实现了一个在c和rust间没有任何数据传递的demo，但是这个demo也依旧会崩溃，经过不断的调试和猜测，我们发现，这个原因主要是在目前官方给出的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，而问题就出在这个32位和64位的区别上，在程序的读取内存的时候采用了这样子的强制类型转换来获得数据：

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

也就是说，程序是32位，但是rust的编译器当成了64位进行处理，但是程序的某些地方在进行类型转换的时候却仍然是32位。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，这样子的话64位地址中的高32位地址就会被舍弃，只保留了低32位的地址，这也就导致了编译能通过但是实际跑的时候就会出现一些问题。

这里，我们也就将long修改成long long后就没有问题了。

而为什么c没问题但是rust的版本就出了问题，我们猜测可能是c和rust的编译器的默认地址的问题，c的虚拟地址可能较小导致高位全部为零，未受到影响，而rust的虚拟地址的高位则不为零，受到了较为严重的影响。

对于项目工程中仍然存在的问题的反思：

1. 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。
2. 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，所以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的三个核心函数的代码重写，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。
3. 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代码的安全性是有所保障的，但是考虑到本次实验中调用了rust和c的接口，这方面的安全性还是有待检验。实际上，我们在调用rust和c的外部接口时，也是有点心虚的，但是对于所有的编程语言，包括rust，都需要跟外部的“不安全”接口打交道，调用外部库等，在“安全”的Rust下是无法实现的; 其次，“安全”的Rust无法高效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，这也是我们目前的一个矛盾的局面，复杂的数据结构内部常常伴随一些不安全的操作，实际上，rust的标准库中也有一部分函数使用了unsafe来实现较为复杂的数据结构的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，实际上，rust标准库中有不少是这么实现的。所以rust如何去支持更加复杂的数据结构，同时保证安全性，这也是我们进一步研究下去的一个重要所在，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，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中，从而提升了系统的整体安全性，从这点来看，编写的demo的可行性较高。

这个是我们的一个对比和总结，我们将我们大作业取得的成果和我们抱有的最终期望进行了一个对比，首先是已经能运行的demo，实现的非常重要的进程调度问题，采用了c和rust混合编程的策略，虽然是比较显著的成果，但是同时必须得认识到的是，freeRTOS的整个代码工程量相当巨大，这注定了我们若要实现这个远大的目标，必须得花费大量的时间，不是一个学期中的课余时间就能简单突破的东西。

然后是我们掌握的工具的对比，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不少。 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，这也导致了初期我们花费了大量时间却依旧进展缓慢的原因，同时我们也必须意识到我们的时间问题，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，在实际编程中我们也遇到了较大的困难，比如之前说的32位和64位的问题，我们在之前的编程中从来没有遇到过，也拜此所赐，我们的工程一度陷入停滞。

不过，我们在这个漫长而艰辛的过程中，我们也确实对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，高效率的一个嵌入式内核。

C中全局变量字符串，rust中没法用

FreeRtos采用固定优先级抢占式调度，所谓”固定优先级”是指每个任务都被赋予了一个优先级，这个优先级不能被内核本身改变(只能被任务修改)。”抢占式”是指当任务进入就绪态或是优先级被改变时，如果处于运行态的任务优先级更低，则该任务总是抢占当前运行的任务

对于优先级相同的任务，Freertos将它们存储在一个循环链表中，在进入当前任务进入阻塞态或时间片结束时，执行链表中的下一个任务