适用于微型控制器的操作系统 Amazon FreeRTOS

邓皓巍 方宇辰 王梓涵 朱河勤

Amazon FreeRTOS

邓皓巍 PB16020582 王梓涵 PB16001707
朱河勤 PB16030899 方宇辰 PB16110579

目录

小组成员

邓皓巍 计算机学院大二学生，熟练使用 google 学术进行搜索，擅长 c、Python 等语言，对数据结构，计算机系统概论均较为熟练

朱河勤 计算机学院大二学生，擅长 c 语言，Python 的编程，编程能力较强

王梓涵 少年班学院大二学生，擅长 c 语言编程，对 FreeRTOS 的内存调度有较多研究

方宇辰 计算机学院大二学生，熟练掌握 c，c++等语言，有 c#的使用经历，阅读 paper 能力较强。

项目背景

总述：

21 世纪一项将会极大程度改变人们生活的事物便是物联网。何为物联网？**物联网**（英语：Internet of Things，缩写 IoT）是互联网、传统电信网等资讯承载体，让所有能行使独立功能的普通物体实现互联互通的网络。[1]

作为互联网行业的独角兽公司之一的亚马逊（Amazon）自然不会错过这个充满商机的领域。亚马逊早从 2013 年就开始布局物联网行业。2017 年 11 月 29 日亚马逊做出了一个重要的决策：雇佣了 FreeRTOS 的发起人 Richard Barry 并开源了基于 FreeRTOS 内核并专注于微控制器的操作系统 Amazon FreeRTOS。

Amazon 一开始对 Amazon FreeRTOS 的设想是主要专攻于感测器节点，后来将目标拓宽到了各种微控制器。微控制器作为小型物联网设备重要的一环，由于物理硬件上的设备限制以及需要面对的任务的复杂程度的日渐提升，对操作系统各方面都提出了更多的需求。

FreeRTOS 简要介绍：

成果

- **低功耗**

  对于嵌入式系统来说,能量消耗是一个很大的议题. 到目前为止,大多是在硬件上减少消耗. 然而有很多可能减少能量消耗的机制:

  1. 只在软件上
  2. 软件,硬件结合,
  3. 只在硬件上

  从不同层面上,可以在应用层面,操作系统层面,芯片层面,等等方面来减少功耗.
  FreeRTOS 采用在空闲时运行 idle 程序,可以简单的理解为

  ```
  while( ! event_occurs())
  {
  };
  ```

  在空闲时几乎不占用内存资源,cpu 也占用得很少,由此来减少能量的消耗.

- **实时性**

  对于嵌入式操作系统,还有实时性的要求,要能及时反馈真实世界的事件. 而且这些事件有时效性, 一旦错过,可能对人,对机器都造成灾难性的伤害,带来巨大的损失.
  目前主要有如下方法来实现嵌入式系统的实时性:

  1. 事件触发方法
     这个方法需要在系统上运行一个能执行任务模块的触发器.然而, [3] 中显示这种方法难以取得明显的效果, 因为函数模块必须实时性地执行,而且会遇到一些限制(比如 资源共享)
  2. 面向服务方法
     通过连接程序的执行单元与实时性的资源限制, 系统将一个服务请求视为有效当且仅当这个服务请求的资源是可以获取的时候, 进一步地, 程序执行单元在进行服务请求时通过某个特定的算法进行反馈. 最后,当资源不再被需要时,就释放此资源
  3. 面向对象方法
     OOP,面向对象编程的好处是可以很大程度上实行代码的重用,增强程序的安全性与稳定性. 然而对于这种嵌入式系统的实现与构建,还需要满足一些要求, 比如: I/O configuration, direct access to I/O signals.. 如果在解决一个系统的性能与降低复杂性上面有所提高,那么这种方法值得尝试.

- 不同 RTOS 之间的比较
  通过比较 µITRON, µTKernel, µC-OS/II, EmbOS, FreeRTOS, Salvo, TinyOS,,SharcOS, XMK OS, Echidna, eCOS, Erika, Hartik, KeilOS, ,and PortOS.. RTOS 的特征与 api 数量了解 freertos 的优点
  共有的特征:
  1. 通过优先–级实现任务调度
  2. 大部分都是用 C 语言实现的
  3. 只有少部分在 IDE 方面有系统级别的支持:
     :
  :

µC-OS/II, EmbOS	IAR compiler;
KeilOS	Keil compiler
µITRON , µTkernel	Renesas HEW compiler.

api 方面

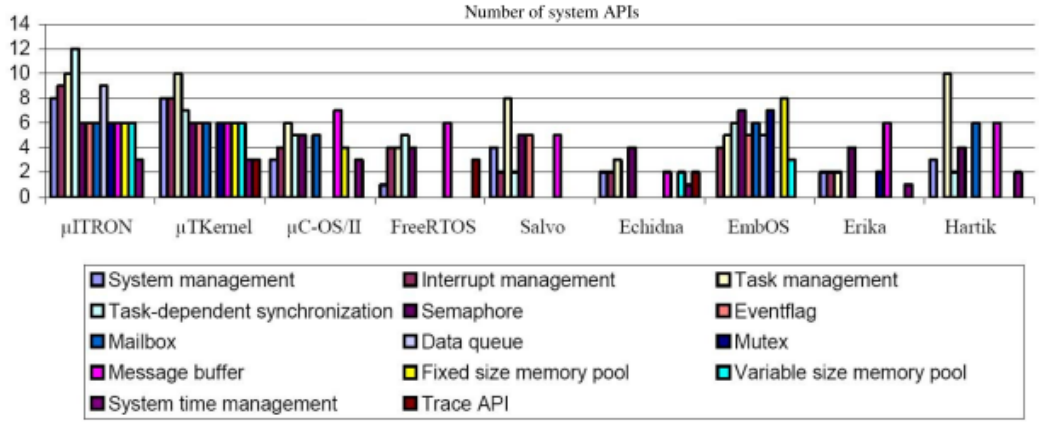

Figure 1: Number of system APIs for various RTOSes

可以发现,freeRTOS 的各种 api 都很少(只略多于 Echidna), api 上,实现起来简单,内核小,而且专注于某个特定的功能, 对于小型的,嵌入式系统, 在功耗和内存上都有很大的优势..

为什么 Amazon 要支持与推展 FreeRTOS?

第一个原因是正式完成其在 AWS IOT 上的技术布局。

对于布局 FreeRTOS，Amazon 可能在 2014 年就将其纳入了自己的物联网规划。Amazon 的云端事业单位 AWS 在该年推出 Lambda，Lambda 是一种供他方呼叫（call）运用的云端函式（或称函数），透过此函式可自动增减执行云端程序所需的资源，包含运算力、储存空间等。在最理想状态下，程序员可以完全不考虑、不理会程序所需要耗用的实体资讯资源，尽情专注于程序逻辑、业务需求的撰写上，此理念即是所谓的 Serverless（无伺服器）。

接着 2015 年 10 月 AWS 推出 AWS IoT Core，这是一套在 Amazon 机房建立的物联网云端平台，可上传、分析前端感测资讯，支援前端物联网运作。2016 年 11 月 AWS 的年

度技术盛会 re: Invent 上，AWS 宣示将不仅仅发展云端（Cloud）技术，也将重视与发展端缘（Edge）技术，因而推出 Greengrass Core，一款在前端的端缘机器内执行的软体。

Greengrass Core 支援前述的 Lambda，可以跟云端上的 Lambda 连结呼应运作。因此，Greengrass Core 等于是把云端上的 Lambda 技术带到前端，进行更大范畴的扩展。而要执行 Greengrass Core，至少要 128MB RAM 才行，AWS 官方也言明可用 RPi 3（第三代树莓派）。有趣的是，Android Things 官方支援的 4 片开发板中，RPi 3 也在其中。

如果说 AWS IoT Core 是 AWS 的 IoT Cloud 技术代表，那 Greengrass Core 即是 AWS 的 IoT Edge 技术代表，但至此还缺乏一样东西，那就是最前端的 IoT Sensor Node（感测器节点）。

有关更前端的技术布局，其实 AWS 也早有初步尝试，即 2014 年 4 月推出的 Amazon Dash。这是一个装置在家庭各处的无线按钮，安置在洗衣粉、卫生纸附近，一旦洗衣粉快用尽、卫生纸快用尽，按下按钮便可完成网路叫货而后到货，此有助加速 Amazon 的电子商务业务。

Amazon 把 Amazon Dash 开放成 AWS IoT Button，可让其他人发挥运用。但 AWS IoT Button 毕竟只是个按钮，尚缺乏一个完整的感测软体平台，因而 Amazon 才推出 Amazon FreeRTOS，将这些年一直在不断进步的 FreeRTOS 作为基石，在其上开发适合自己的技术，补足了最前端的一块技术。至此的 AWS IoT 技术算已近乎完整布局。

那么为什么是 FreeRTOS 呢？

当我们想到物联网设备时，可能会想到连接的摄像头或传感器。大多数这些设备具有非常基本的 CPU 并运行 Linux 或类似的操作系统。不过，并不是每个物联网设备都具有真正的 CPU。相反，它们由基本的微控制器驱动。这些设备（比如烟雾探测器等）通常很旧，不能直接连接到云端。因此 Amazon 倾向于使用 FreeRTOS 操作系统作为基础内核，在其上进行改造，使得到的操作系统可以让微控制器更方便的与 Amazon 自己的 Greengrass Core 传递信息

Amazon FreeRTOS 的优势简述：

Amazon FreeRTOS (a:FreeRTOS)是一种适用于微型控制器的作业系统，可让小型、低功率的边缘装置易于进行程式设计、部署、保护、连接及管理。Amazon FreeRTOS 是以适用于微型控制器的常用开放原始码作业系统 FreeRTOS 核心为基础，并使用软体程式库加以扩展，以便轻松安全地将小型、低功率的装置连线至 AWS IoT Core 等 AWS 云端服务或执行 AWS Greengrass 的强大边缘装置。

微型控制器(MCU) 是包含简易处理器的单一晶片，可在设备、感应器、健身追踪器、工业自动化和汽车等众多装置中找到。上述的大多数小型装置都能透过连线至云端或连线至其他本机装置获得不少好处。例如，智慧型电度表必须连线至云端以回报使用量；建筑物安全系统则需要进行本机通讯，需要使用徽章将门解锁。微型控制器的运算能力和记忆体容量有限，因此通常只能执行简单的功能型任务。微型控制器常用的作业系统都没有内建连线至本机网路或云端的功能，使得 IoT 应用程式难以使用。Amazon FreeRTOS 可协助解决这个问题，它提供核心作业系统(用于执行边缘装置) 及软体程式库，让设备可以安全简便地连线至云端(或其他边缘装置)，方便开发者从这些装置收集 IoT 应用程式资料并采取行动。

Amazon FreeRTOS 以 FreeRTOS 内核为基础。FreeRTOS 内核具有 6-15kb 的内存空间，非常适合内存和计算能力有限的、基于微控制器的设备。它包括任务优先级、代码模块化和电源管理等功能，可以帮助确保应用程序满足其处理截止日期要求、应用程序代码更简单、应用程序功耗实现优化。FreeRTOS 内核的最新版本（v10）包括字符串缓冲区、消息缓冲区、更新的 API 参考和新的设备支持。

同时 Amazon 为 FreeRTOS 增添了一套外围库使其嵌入的微控制器可以更加方便地与云端进行联系并传输数据。同时将授权模式修改为 MIT 增强了该系统的安全性。

立项依据

概述：

IoT 行业还未能完全渗透进人们的日常生活。为了进一步推进 IoT 行业的发展，Amazon 在 2017 年 12 月推出了自己的专注于微控制器的嵌入式操作系统 Amazon FreeRTOS。Amazon FreeRTOS 刚刚推出不久，在许多场景下还不能达到 Amazon 设立的预期。而在我们身边中的生活用品中嵌入可高效传递信息的感测器可以有效地提升人们的生活质量。我们的工作就是针对将要嵌入这些感测器中的微控制器的操作系统 Amazon FreeRTOS 进行优化，使其在内存管理，任务调度以及功耗方面更加贴合感测器节点的工作。

Amazon FreeRTOS 的问题：

如前文所述，Amazon FreeRTOS 主要用于物联网设备中的微控制器部分。随着物联网行业的发展，设备所需要面临的任务也越来越复杂，同时硬件可提供的资源也越来越充足，无需担心 RTOS 会拖累微控制器的性能。因此有必要在其中的微控制器中嵌入 RTOS 系统已更加高效的利用 CPU 来解决问题，而不是满足于状态机。

我们来看 Amazon 决定采用的 Amazon FreeRTOS。

对于嵌入式设备，功耗是一个需要仔细考虑的问题。在最新版本的 FreeRTOS 内核 FreeRTOS 10.0 中，内存调度算法教常使用的算法较为复杂，并不完全适合使用在微控制器上，因为微控制器一般需要长时间运转并且其上可搭载的动力源有限。由于动力源有限，IoT 的发展便对嵌入微控制器的操作系统的功耗提出了要求。而操作系统的功耗与内存调度算法以及任务调度算法的复杂性是息息相关的。

虽然如上文所说 Amazon 已完整布局，但 Amazon 的前端技术在端缘处要配置至少 128MB RAM 的系统，而各感测器节点的 Amazon FreeRTOS 需 64KB RAM，与 Amazon 声称的 16KB 尚有距离。Amazon 目前给出的四块开发板均高于 64KB。若把通讯协定卸除转移，可以进一步降低需要的内存空间。

Amazon FreeRTOS 開發板的 RAM 容量

開發板	RAM
Amazon 官方原標榜	16-64KB
Microchip Curiosity PIC32MZ EF 開發板	512KB
NXP LPC54018 IoT 模組	360KB
STM32L4 Discovery Kit IoT 節點	128KB
SimpleLink Wi-Fi CC3220SF 無線微型控制器 LaunchPad 開發套件	256KB

为了有效地降低所需要的内存，FreeRTOS 所使用的内存调度方法以及任务调度过程也需要修改可调度内存大幅减小的同时，任务调度过程为了保证微控制器运行的效率也需要搭配地做出优化。常规 RTOS 的优先级嵌套算使得任务执行顺序、执行时序不易确定，任务嵌套对所需的最大堆栈 RAM 大小估计也变得困难，这对于安全性和缩小 RAM 的大小是不利的。因此，为了适应对于安全有严格要求的微控制器我们需要对此进行优化。

与此同时，Amazon 希望旗下的微控制器能够有效地将收集到的数据传输到自己的 Greengrass Core，这需要对 FreeRTOS 进行一定程度的修改来适应这种对连接性需求大的场景。并且若要在商业上得到稳定的应用，安全性也是操作系统必须考虑的因素。

由以上功耗、内存、连接性、安全性等方面可见，若想达到 Amazon 对物联网行业的设想，Amazon FreeRTOS 还存在需要优化的地方，这也将是我们这个项目的重点。

重要性分析

智慧化 / 物联网产品应用的发展进程，建构出物物相连的全新世界，从各式智慧终端产品、穿戴式装置、智慧家居、工厂应用中，皆可发现各种智慧化应用。然由于装置在走向更轻薄短小、续航力更长、支援度更广的趋势下，微控制器(MCU)所提供的功能愈来愈复杂，扮演的角色愈来愈重要。如果能将感测器节点收集到的数据高效的传递并且进行有效的计算，可以更加高效的分析用户的行为以提供更好的服务。

随着 IoT 行业的发展，人们发现物联网技术的扩散将带来新的挑战。特别是对于最终用户和提供商来说，物联网的规模部署会非常麻烦。我们如何使用低成本高效微控制器以及可以轻松连接到云的安全操作系统创建这些 IoT 解决方案？

与此同时，微控制器往往需要长时间的运转，并且有时候还身负收集传递数据的重任。由于任务需求，往往需要一款硬实时操作系统来控制。

Amazon 开发出自己的专注于微控制器的操作系统正是为了使设备更加能适应与云端的联系。Amazon 扩展了 FreeRTOS 内核，可简化基于微控制器的边缘设备的开发，安全性，部署和维护，可以更加方便的部署基于边缘计算的物联网设备。同时 Amazon FreeRTOS 的重点目标是给予微控制器更加高效的与云端进行联系的能力，这与物联网的发展方向是一致的。

前瞻性分析

Amazon 为了推行 FreeRTOS，除了雇用 FreeRTOS 创办人，将 FreeRTOS 授权变更成 MIT（新授权方式更利于 FreeRTOS 软体技术的商业化运用）外，还推出两个配套，一是 Amazon FreeRTOS Qualification Program，简称 Amazon QFP，即大众以 Amazon FreeRTOS 开发出应用系统时，可交由 Amazon 测试验证。

另一是 AWS 推出 AWS IoT Device Management，可以透过云端上单一主控台，以无线通讯方式更新每个前端装置内的 Amazon FreeRTOS 作业系统韧体，以无线方式更新韧体也称为 OTA（Over- The-Air）。如此，Amazon 强化推展前期的验证工作，也强化 IoT 实际营运布建后的维护管理。由此可知，Amazon 加持后的 FreeRTOS，也对 Maker 圈有利，以 FreeRTOS 为基础进行各种开发，将可更快商业化发展程序。

相关工作

1. 内存管理

传统 RTOS 动态存储分配算法(Dynamic Storage Accelerator)概述

DSA 算法的目标是为应用程序分配内存块，对于 RTOS 而言，应用程序通常是在系统的整个生命周期中运行的，因此对于 DSA 算法要求通常有一下几点：

（1）时间有界性

执行内存分配和释放的最坏执行时间是预先已知的，是独立于应用程序的数据，这是必满足的最主要的要求。

（2）快速响应时间

除了具有一个有界的响应时间外，使用的 DSA 算法的响应时间应该很短。有界的 DSA 算法，如果响应时间是普通算法的 10 倍，是不适用的。

（3）满足内存需要

系统运行内存不足时，非实时应用程序能够接收一个空指针或被操作系统终止。显然，任何时候都能满足内存需要是不切实际的。但 DSA 算法必须把内存碎片和内存浪费降到最少以降低耗尽内存池的可能性。

不同的算法在寻找最佳空闲块时的策略有所不同。DSA 算法可以分为以下类别：[1]

（1）顺序拟合算法

在顺序拟合算法中，空闲内存块由单向或者双向链表管理。查询空闲内存块的时间复杂度为 O（n），当内存块数目较大时，不能保证查询内存块的实时性，所以不宜在内存较大的 RTOS 中使用。

（2）索引查找算法

索引查找算法使用排序二叉树等非常复杂的数据结构来管理空闲内存，具有复杂的实现过程，并且因采用的数据结构的不同而具有不同的性能。

（3）分类搜索算法

分类搜索算法把空闲内存划分为范围不同的多个区间，每个区间上的内存块由另一个数组链表管理，该数组链表保存着查询空闲内存块的头指针。需要说明的是，同一区间内的空闲内存块，不一定物理相邻。分类搜索算法虽然复杂，但查找空闲内存块的时间复杂度为 O（1），能有效满足实时性，适合在内存较大但对效率要求较高的 RTOS 使用。

（4）位图搜索算法

位图搜索算法使用位图管理空闲内存块，搜索空闲内存所需信息都被存储在一小块内存中，可以实时响应系统需求，是 RTOS 中普遍采用的算法。

FreeRTOS 已有内存管理策略

在 FreeRTOS 中，内存管理使用 pvPortMalloc 与 vPortFree 函数，函数原型如下：
void*pvPortMalloc(size_txWantedSize);
voidvPortFree(void*pv);
pvPortMalloc 函数类似于 c 语言中的 malloc 函数, 形参是调用者请求的内存大小, 类型 size_t 是 unsignedint 的重定义。返回值为一个 void 型指针，指向申请到的起始地址。
vPortFree 函数类似于 c 语言中的 free 函数，释放掉由 pv 指向的一段空间。

在 FreeRTOSv10.0.1 中有 5 中内存管理策略, 分别被命名为 heap_1, heap_2, heap_3, heap_4, heap_5。

heap_1
将所有内存看作一个数组。类型为 unsignedchar，也就是一个字节，数组大小为内存堆总大小。
第一次调用 pvPortMalloc 时，heap_1 首先确定全部堆空间的起始地址，用 pucAlignedHeap 保存。
如果调用者申请大小为 xWantedSize 的空间，heap_1 在判断是否有足够的空间分配之后，直接返回下一个空闲字节的地址，同时把下一个空闲字节地址加上本次分配的空间大小。
由于 heap_1 策略没有记录每次分配的起始地址及大小, vPortFree 函数只能把传入的指针清零，并没有实际释放空间。

heap_2
heap_2 使用一个链表 FreeList 记录内存中所有的空闲空间。
当调用者申请大小为 xWantedSize 的空间时，heap_2 检索整个链表，寻找一个空闲空间大于等于 xWantedSize 的空间。如果该空间大于 xWantedSize，heap_2 会将该空间分为两块，并将未使用的部分放入 FreeList。FreeList 中的地址空间按从小到大排列，保证每次调用时所需的时间最短。
heap_2 的 vPortFree 函数把传入指针重新加入 FreeList，完成空间释放。
heap_2 可以释放空间，但不能把已经释放的连续空间重新整合成一个大空间，因此大量的申请、释放随机大小的空间会导致大量的碎片。因此 heap_2 适合动态申请、删除相等空间的应用。heap_2 申请空间的时间不确定，但仍然比系统提供的 malloc 与 free 效率高。

heap_3
heap_3 简单的调用了 c 语言的 malloc 与 free 函数，对于嵌入式操作系统来说有可能无法使用，也有可能占用宝贵的代码空间。heap_3 会在调用 malloc 和 free 之前挂起调度器，保证线程安全。由于 heap_3 的内存堆由链接器控制，不做重点分析。

heap_4

heap_4 是 heap_2 的改进版本，在 heap_2 的基础上把释放出的连续空间重新整合成一个大的空间并加入 FreeList。
heap_4 相比于 heap_2 在随机大小的申请释放过程中产生的碎片更少，但由于仍然需要遍历链表申请时间仍然不确定。

heap_5
heap_5 是 heap_4 的改进版本，同样可以重新整合空间。在此基础上，heap_5 可以给与调用者多块非连续的空间。heap_5 的 FreeList 不再按空间大小排序，而是按照未分配内存的地址从低到高排序。分配时根据需求大小从低地址到高地址从 FreeList 中依次取出空间并用指针连接起来返回。
heap_5 的 vPortFree 函数的实现和 heap_4 的类似，同样需要把要释放的空间和它在物理上相连的未分配空间整合后放入 FreeList。
由于 heap_5 需要花费额外空间来连接不相邻空间，对于某些小内存设备来说是一种浪费，并且加大了内存使用者的负担。在时间上 heap_5 虽然能做到比 heap_4 更快，仍然到不了 O(1)量级。
综合上述 5 种已有的内存调度策略，对于时间确定（即时间复杂度为 O(1)）的算法(heap_1)，无法释放空间，而其余可以释放空间的算法申请空间时都需要遍历链表，无法做到 O(1)时间。在最新的 FreeRTOSv10.0.1 中默认采用 heap_5。

一种新的针对 RTOS 的 DSA 算法——TLSF
TLSF[2]是一种二级分隔拟合算法，使用链表与位图相结合的方式对内存进行管理。采用空链表数组，并将数组设计为二级数组：第一级数组按照 2 的幂将空内存块分级，分级的数目称为第一级。第二级数组将第一级的结果进一步线性划分。每一个链表数组都有一个相关联的位图来记录内存块的使用情况及空内存块的大小和位置。
TLSF 算法通过单链表管理从物理内存获取的不连续的内存区域。当用户申请指定大小内存时，TLSF 算法首先在当前已有的内存区域中查找是否有满足要求的内存块。与传统算法不同的是，TLSF 算法可以通过公式 1 直接计算出指定大小内存在单链表中的位置。如果链表

$$\texttt{mapping}\,(size) \rightarrow (f, s)$$
$$\texttt{mapping}\,(size) = \begin{cases} f := \lfloor log_2\,(size) \rfloor \\ s := \left(size - 2^f\right) \frac{2^{SLI}}{2^f} \end{cases}$$

中有满足要求的内存则返回给用户，没有则从物理内存新申请一块固定大小的内存区域，并从其中分配出指定大小内存块给用户，同时将该内存区域中剩下的空闲块添加到二级索引中。与此同时，TLSF 算法尝试合并新分配的内存区域到已有内存区域中，形成一个大的内存区域，从而提高内存使用效率。
针对 TLSF，已有在多种条件下的改进，比如针对媒体服务系统中的视频编解码转换及流化处理[3]，μCOS- II[4]等的改进。对于 FreeRTOS，仅有简单的移植和对比实验。实验对比对象是 FreeRTOS 已有的 heap_1 至 heap_4，结果发现 TLSF 算法对于碎片的降低、存储分配、释放时间均有 10%左右的改进[5]。

2. 任务调度

FreeRTOS 任务相关的代码大约占总代码的一半左右，这些代码都在为一件事情而努力，即找到优先级最高的就绪任务，并使之获得 CPU 运行权。任务切换是这一过程

的直接实施者，为了更快的找到优先级最高的就绪任务，任务切换的代码通常都是精心设计的，甚至会用到汇编指令或者与硬件相关的特性，比如 Cortex-M3 的 CLZ 指令。因此任务切换的大部分代码是由硬件移植层提供的，不同的平台，实现发方法也可能不同

概略来说，FreeRTOS 操作系统支持三种调度方式：抢占式调度，时间片调度和合作式调度。 实际应用主要是抢占式调度和时间片调度，合作式调度主要用在资源有限的设备上面，现在已经很少使用了。

抢占式调度器简单来说就是每个任务都被分配了不同的优先级，抢占式调度器会获得就绪列表中优先级最高的任务，并运行这个任务。Freertos 的一个特点就是支持时间片调度，只要资源允许，同一优先级支持任意多个任务。
Freertos 使用的时间片调度算法就是 Round-robin 调度算法。这种调度算法可以用于抢占式或者合作式的多任务中。实现 Round-robin 调度算法需要给同优先级的任务分配一个专门的列表，用于记录当前就绪的任务，并为每个任务分配一个时间片（也就是需要运行的时间长度，时间片用完了就进行任务切换）。另外，时间片调度适合用于不要求任务实时响应的情况。
总而言之，FreeRTOS 中处于运行状态的任务永远是当前能够运行的最高优先级任务。是一种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而这个优先级不能被内核本身修改（只能被任务修改）。抢占是指当任务进入就绪态或者优先级改变时，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任务。

任务调度包含三个部分：保存当前任务现场；选择下一个执行的任务；恢复任务现场。对于选择下一个执行的任务这一步骤，freertos 提供了一种优化，可以加快选择的速度，不过对优先级总数和硬件平台有一定的要求（考虑到 freertos 同一优先级支持任意多任务，优先级数量的限制不构成重要问题）。

调度过程中一个重要的问题就是中断的处理，对中断的处理方案是与硬件高度相关的。
比如说 freertos 设计了一种新的开关中断实现机制。关闭中断时仅关闭受 FreeRTOS 管理的中断，不受 FreeRTOS 管理的中断不关闭，这些不受管理的中断都是高优先级的中断，用户可以在这些中断里面加入需要实时响应的程序。freertos 提供了“可管理的中断范围”。以某个中断优先级为阈值，优先级低于（或等于）此分界线的中断将全部屏蔽，较高的中断则不受影响。而 freertos 不会让高优先级的中断参与任何 api 的调用。所以这些高级中断是否触发都不会影响 freertos 的正常运行。

以在 cortex-M 上的实现为例，它带有 NVIC，即嵌套向量中断控制器，他有一个 BASEPRI 寄存器。屏蔽中断所使用的阈值就存储在其中，屏蔽时将需要屏蔽中断的最高优先级值填入寄存器便能达到效果。

又比如 nvic 支持优先级分组，即分为抢占优先级和子优先级(或亚优先级)，但是 freertos 不允许存在子优先级，假如设置中出现了子优先级，操作系统会将它去掉。

注释：

1.刘云浩编. 物联网导论. 北京：科学出版社. 2010-12：4. ISBN 9787030292537 （中文（简体））.

参考文献:

[1] Herman Roebbers, Altran Netherlands, B.V.,Eindhoven, “Achieving Ultra Low Power in Embedded Systems” embeddedworld2017

[2] Vicent Rutagangibwa, Babu Krishnamurthy “A Survey on The Implementation of Real Time Systems for Industrial Automation Applications” IJIRST –International Journal for Innovative Research in Science & Technology| Volume 1 | Issue 7 | December 2014

[3] D. Fennibay, A. Yurdaku, and A. Sen, “A heterogeneous simulation and modeling framework for automation systems,” in IEEE Transactions on Computer-Aided Design of Integrated circuits and systems, vol. 31, no. 11, November 2012, pp. 1642 – 1655.

[4] Su-Lim TAN, Tran Nguyen Bao Anh “Real-time operating system (RTOS) for small (16-bit) microcontroller” The 13th IEEE International Symposium on Consumer Electronics (ISCE2009)

Using the FreeRTOS Real Time Kernel

[1]王秀虎,张昕伟.基于μCOS-Ⅱ的TLSF动态内存分配算法的应用与仿真[J].微型机与应用,2013,32(5):4-7.

[2] Masmano M, Ripoll I, Crespo A, et al. TLSF: a new dynamic memory allocator for real-time systems[C]// Real-Time Systems, 2004. Ecrts 2004. Proceedings. Euromicro Conference on. IEEE, 2004:79-86.

[3] 陈君，樊皓，吴京洪. 基于TLSF算法改进的动态内存管理算法研究[J]. 网络新媒体技术，2016, 5(3):55-60.

[4] 王秀虎，张昕伟. 基于μCOS-Ⅱ的TLSF动态内存分配算法的应用与仿真[J]. 微型机与应用，2013, 32(5):4-7.

[5] 刘林，朱青，何昭晖. FreeRTOS内存管理方案的分析与改进[J]. 计算机工程与应用, 2016, 52(13):76-80.