山东大学

硕士学位论文

基于μC/OS--Ⅱ的嵌入式实时操作系统调度算法的改进

姓名：郝强

申请学位级别：硕士

专业：计算机应用技术

指导教师：高胜法

20111020

摘 要

在工程实践中，嵌入式系统中的系统绝大多数都需要很高的实时性，如果这样的话对于嵌入式操作系统提出了更高的实时性要求。要想使用实时操作系统内核开发嵌入式多任务操作系统，应该将系统功能进行合理分解，根据程序相对于其他程序的重要性决定其执行的优先级，构造不同的程序。进程调度算法的是否优秀以及执行效率是否优秀直接影响到这个嵌入式内核的应用范围以及对系统的实时性程度。该系统已被移植到多种处理器架构中，在某些实时性要求严格的领域中得到广泛应用，能够满足对实时性要求越来越高的需要。我们在分析μC/OS- II 的基础上，对其调度算法进行改进。本文除了对嵌入式实时系统几种典型的算法进行分析比较，研究出改进后的算法，还对当工程的复杂性增大，在μC/ OSⅡ操作系统上运行的程序数目不断增加时，程序数目过多造成的效率下降这种情况，进行分析。μC/ OS2 Ⅱ内核最多可以管理 64 个程序，其中系统保留了 8 个程序，用户最多可以有 56 个应用程序。因此，当工程的复杂性增大，在μC/ OS2 Ⅱ操作系统上运行的程序数目不断增加时，如果程序数目超过 64 个，就必须换用其他的开发平台，这样就可能造成大量的前期开发工作作废。我们考虑到这种情况，根据μC/ OS2 Ⅱ本身的程序可扩展性，在原有的优先级调度算法基础上，提出了两种可行的大量增加可管理程序的算法。一种是利用μC/ OS2 Ⅱ原有的优先级判定表格，重新定义了存放程序优先级的字节，并重新建立程序就绪表。它把 64 个程序扩充到 256 个程序，把程序放入就绪表中，给出了新的最高就绪程序的查找算法。另一种是利用原来存放优先级的字节中剩余的两位作为索引，重建就绪表，使程序优先级扩展到 256 个。我们在利用μC/ OSⅡ源码公开的基础上，对原有的内核程序优先级调度算法进行修改，使其可应用于多于 64 个程序的复杂的工程项目开发。并且通过比较得出结论，第一种算法要优于第二种算法，第一种算法在理论上更简洁清楚，并且更加易于实现。

关键词：实时操作系统；算法；嵌入式

ABSTRACT

The quality and efficiency of the gramm scheduling algorithm are critical to the applicability and extent of realtimeness of the embedded kernel. μC /OS-II is a preemptive real-time multigramming kernel that boasts the following features:

completeness, translatability, solidifiablity and modifiability. It supports up to 56 usergramms, accommodates semmaphores, mailboxes, message queues and other commonly.

used inter-process communication mechanism. It also works with a variety of microcontrollers and microprocessors, and is thoroughly coded in ANSI C. μC / OS-II's outstanding readability and translatability has made it easy to be ported to a good number of processor architectures and widely applied in areas which have increasingly demand for enhanced real-time features. Based on the μC / OS-II-derived analysis, we made several improvements on its scheduling algorithm. Therefore, when the complexity of a project heightens and the number of gramms in a μC/ OS2 II operating systems increases, one must switch to other development platforms once the number of gramms exceeds 64. This could result in voiding much of the pre-development work. Taking this situation and the gramm extendibility of μC / OS2 II into consideration, and based on the original algorithm of priority scheduling , we broached two feasible algorithms with much enhanced gramm manageability . One is to use the generic priority-determining table of μC / OS2 II , redefine the bytes storing gramm priority, and by reestablishing a gramm table. After expanding the gramms from 64 to 256, and shoveling back the gramms into the gramm table, it will formulate a new, highest searching algorithm of the highest readiness of gramm priority. Another one is to use the two bytes left in the original priority-storing byte storage as indexes, rebuilding the gramm readiness table and expanding the gramm priorities from 64 to 256. On the ground of the open source μC / OS2 II ,we made some modifications on the gramm priority scheduling algorithm of the original kernel, hence suitable to be applied on intricate development project with 64 or more gramms. We further draw our conclusions as the follow: algorithm one is superior and finer than algorithm two, for it is more clear and concise in theory, and more easily implemented. In engineering practices, there has been an elevated demand over the real-time feature of embedded system. In order to use the real-time kernel to develop embedded multi-gramm system programs, one should first devise a rational decomposition over system functions, formulate different gramms, and then determine priorities according to levels of gramm importance. The quality and efficiency of the gramm scheduling algorithm are critical to the applicability and extent of realtimeness of the embedded kernel. μC /OS-II is a preemptive real-time multigramming kernel that boasts the following features: completeness, translatability, solidifiablity and modifiability. It supports up to 56 user gramms, accommodates semmaphores, mailboxes, message queues. And other commonly used inter-process communication mechanism, works with a variety of microcontrollers and microprocessors, and is thoroughly coded in ANSI C. μC / OS-II's outstanding readability and transplantibility has made it easy to be ported to a good number of processor architectures and widely applied in areas which have increasingly demand for

enhanced real-time features. Based on the μC / OS-II-derived analysis, we made several improvements on its scheduling algorithm. Besides improving algorithm by comparing several embedded real-time systems of typical algorithm. This thesis also scrutinizes the decreased efficiency problem caused by enhanced project complexity and increased gramms running in a μC / OS II. The kernel can handle up to 64 gramms, and as 8 of them are reserved for the system. The user is actually allotted to have 56 application gramms at most. Therefore, when the complexity of a project heightens and the number of gramms in a μC/ OS2 II operating systems increases, one must switch to other development platforms. Due to the number of gramms exceed 64, which could result in voiding much of the pre-development work: Taking this situation and the gramm extendibility of μC / OS2 II into consideration, and based on the original algorithm of priority scheduling , we broached two feasible algorithms with much enhanced gramm manageability . One is to use the generic priority-determining table of μC / OS2 II, redefine the bytes storing gramm priority. By reestablishing a gramm table, expanding the gramms from 64 to 256, and shoveling back the gramms into the gramm table. It can formulate a new, highest searching algorithm of the highest readiness of gramm priority. Another one is to use the two bytes left in the original priority-storing byte storage as indexes, rebuilding the gramm readiness table and expanding the gramm priorities from 64 to 256. On the ground of the open source μC / OS2 II ,we made some modifications on the gramm priority scheduling algorithm of the original kernel, hence suitable to be applied on intricate development project with 64 or more gramms. We further draw our conclusions as the follow: algorithm one is superior and finer than algorithm two, for it is more clear and concise in theory, and more easily implemented.

KEY WORDS: Real-time operating system;Algorithm;Embedded;

CONTENTS

Abstract in Chinese..... I
Abstract in English..... II
1、Introduction..... 1
2、μC/OS- II Summary..... 2
2.1 Features..... 2
2.1.1 Portable..... 2
2.1.2 ROMable..... 2
2.1.3 Scalable..... 3
2.1.4 Multi-program..... 3
2.1.5 Preemptible..... 3
2.1.6 Stack..... 3
2.1.7 Deterministic..... 3
2.1.8 Services..... 3
2.1.9 Stability and Reliability..... 3
2.1.10 Interrupt Management..... 4
2.2 Principle..... 4
2.3 Gramm Manager..... 5
2.4 Gramm Scheduling..... 6
2.5 Gramm Create..... 8
2.6 Gramm Change..... 8
2.7 Mutual Exclusion Semmaphore..... 10
2.8 Message Mail Boxes..... 13
2.9 Prevent Priority Inversion..... 15
2.10 Prevent Deadlock..... 16
2.11 Clock Interupt..... 17
2.12 Robin..... 18
3、Scheduling Algorithm..... 20
3.1 Static Scheduling..... 20
3.1.1 Rate Monotonic Scheduling..... 20
3.1.2 Deadline Monotonic Scheduling..... 20
3.2 Dynamic Scheduling..... 21
3.2.1 Earliest Deadline First..... 21
3.2.2 Least Slack Time First..... 22
3.3 Selection of Static and Dynamic Scheduling Algorithm..... 22
4、Improvement of Typical Scheduling Algorithm..... 24
4.1 Priority Assignment Algorithm Integrated With RMS and EDF 24
4.2 Up to the Earliest Deadline First Algorithm..... 26
5、Improvement of Scheduling Algorithm Which Increase in Administrative Gramms 29
5.1 Background..... 29
5.2 First Improvement of the Scheduling Algorithm..... 29
5.3 Second Improvement of the Scheduling Algorithm..... 41

5.3.1. Take the Program into Ready Table..... 42
5.3.2. Delete the Program from Ready Table..... 43
5.3.3. The way of determine the highest PRI Ready Program.. 43
5.4 Comparison of Two Methods..... 44
References..... 47
Acknowledgement..... 48
Papers Published in the Period of Master Education..... 49

第一章 绪论

嵌入式的操作系统和传统的通用计算机系统有本质不同，应用程序完全有可能不通过操作系统而直接在处理器上执行，为了更加合理地管理多进程和利用系统的资源系统一般以更为有效的实时管理系统（RTOS）作为开发环境。

现如今的嵌入式系统中广泛采用单用户特点，因为这样会带来系统的实时性、安全性不强，编写程序比较复杂等问题，因此会使得系统频繁复位，还有可能无法达到设计目标。

这篇论文采纳了操作系统μC/OS II 多线程的实时内核,这个内核具有嵌入式软件共有的可裁剪等特点。

因为它是实时操作系统，所以除了要满足应用的功能，更重要的是还要满足应用提出的在一定的时间内必须得到处理的要求。

实时操作系统所遵循的最重要的设计原则是：采用各种算法和策略始终保证系统行为的可预测性。它的首要程序是调动一切可利用的资源完成实时控制程序，其次才着眼于提高计算机系统的工作效率。如何使程序集内各程序满足各自的时限，使系统得以正常、高效率工作的程序调度算法一直是实时系统领域内研究的焦点。程序执行的算法是好是坏，以及这个算法它的执行效率它会影响这个内核的应用范围的广度及实时性能是否优秀。为了满足对实时性要求越来越高的需要，同时避免频繁改变就绪程序的优先级，我们先分析μC/OS- II ,然后对其进行调度算法的改进。

第二章 μC/OS II 概述及其他功能实现

μC/OS II 是一个源码完全开放的实时操作系统 kernal，它的全名是 Micro-Controller Operating System Version 2。μC/OS-II 作为很多研究者所亲睐的嵌入式操作系统，它是一个多程序的完全实时管理内核，它主要能够实现程序管理功能。

μC/OS II 在管理系统中的多个程序，必须掌握知道要执行的程序的排列顺序，并且通知 cpu 哪些进程先执行，哪些进程后执行，这些由程序调度算法来完成，需要动态为停当程序的优先级排序。该系统为了满足对实时性要求越来越高的功能，又要避免出现频繁改变停止程序的优先级，对其调度算法进行改进，在分析μC/OS-II 源代码的基础上。该系统已经在很多领域得到应用，比如电子信息、医疗管理、设备控制、网络管理、自动机还有工业机器人等。

μC/OS-II 系统适用于各种微控制器和微处理器；并且是个特殊的，可移植、可固化、可裁剪的抢占式实时多程序内核;在某些实时性要求严格的领域中得到广泛应用，所有代码用 ANSI C 语言编写，程序的可读性强，具有良好的可移植性，已被移植到多种处理器架构中。该系统是一个完全抢占式的实时操作系统内核，不支持轮转法程序调度，它总是运行最高优先级的就绪程序。每个程序赋予程序有且只有一个优先级，并且可以拥有自己的独立的堆栈。μC/OS- II 提供了系统级别服务，如固定大小的内存分区、队列机制、邮箱机制、信号量机制、以及时间相关的函数等等。μC/OS-II 系统拥有这么多优点，可以完全承担系统提供的相应的程序级别管理。

2.1特点

2.1.1 可移植性（Portable）

μC/OS-Ⅱ和微处理器硬件相关的那部分是用汇编语言写的，其他的是用移植性很强的 ANSI C 写的。汇编语言写的那一部分已经很少了，使得它更加容易的移植到其他处理器上进行运行。但是也有一些客观条件作为约束，那就是这个微处理器要有相关的寄存器，例如堆栈指针寄存器等，还有条件就是由 CPU 内部 register 完成的出入栈。

2.1.2 可固化（ROMable）

μC/OS-Ⅱ非常适用于对嵌入式应用的管理和对嵌入式相关操作的管理，这也

就是说只要用户可以实现针对 C 语言的执行和安装，这个系统就能够安装并且执行到针对其他开发的产品中，并且成功是完成任务。

2.1.3 可裁剪（Scalable）

用户的应用实现了系统中的功能，这些功能如果是这些应用所需要的就可以说是成功的。可以只使用这个系统中的那些应用所需要的那些服务就可以了。

也可以这样说某种程序能够只执行并不多的几个系统级别的调用，而另一个程序则使用了几乎任何系统的功能。

2.1.4 多程序

操作系统可以实现并且有效执行 64 个程序，但是现在这一个版本的系统又给自己保留 8 个级别不允许别的程序申请。这样的话，其他程序能够最多申请那剩余的 56 个级别。

因为这个系统给每个程序的优先级的值都是不同的，这代表该系统并不能够执行时间片轮转的调度算法。

2.1.5 可以抢占

系统是抢占式的内核。这说明该系统总是运行一直准备就绪的程序中优先级最高的那个程序。

2.1.6 程序栈

以便压低应用程式对 RAM 的需求。各个程序都要有自己单独的栈，μC/OS-Ⅱ允许每个程序有不同的栈空间用于存储。

2.1.7 可确定性

系统服务的执行时间不依赖于应用程式程序的多少。全部 μC/OS-Ⅱ的方法选用和服务是不确定的。全部系统中的函数调用和服务的执行时间是可以预测的。所以说系统执行是可确定的！

2.1.8 系统服务

系统提供很多系统级别的服务。

2.1.9 稳定性和可靠性

二代系统是基于一代系统的基础进行改进和开发的，μC 是 92 年以后已经有好几百的公司就行改进和开发，并且使用到了商业。经过商业应用的检验，可以证明这个系统是非常稳定和可靠的！

2.1.10 中断管理

例如级别更高的程序如果某种事件打断，则比他级别更高的程序在全部退出后就能够被系统接收，并且可以执行了。中断可以使正在执行的程序暂时挂起。

2.2工作原理

SLOS 使我们根据μC/OS 改进的操作系统。它完全兼容μC/OS 的硬件抽象层接口规范，这样用户就可以无需改动地直接使用μC/OS 硬件抽象层，而不要再花费精力去写具体的硬件相关代码。硬件层，便是具体的硬件了。要使 SLOS 运行起来，硬件层必须提供可以进行数据运算控制的 CPU，以及可以提供周期性时钟源的定时器。系统上电后，操作系统得以运行。首先是硬件的初始化，这一部分属于硬件抽象层代码。然后是操作系统的初始化，这一部分的具体内容详见后面的各模块详细说明。初始化完毕，创建一个起始程序，在起始程序中打开时钟中断，系统真正进入多程序状态。创建其它程序后，系统开始对个程序进行调度，系统真正的运行起来。

μC/OS-II是如何实现的呢？它近似地让最高优先级的就绪程序处于运行状态。首先，按初始值设置 MCU,再进行操作系统初始值的设置，主要完成程序控制块 TCB 初始值的设置，TCB 优先级表初始化,TCB 链表初始化，事件控制块(ECB)链表初始化，空程序的创建等等；然后就可以开始创建新程序中，并可在新创建的程序中再创建其他新程序；最后调用 OSStart()函数启动多程序调度。在多程序调度开始后，启动时钟节拍源开始计时，此节拍源给系统提供周期性的时钟中断信号，实现延时和超时确认。SLOS 的设计是采用分层次设计，各层次从上到下依次为程序应用层，API 系统层，服务系统层，内核系统层，抽象硬件层，硬件层。应用程序层，即用户代码层。此处的代码由最终用户提供，用以实现用户所要求的程序。系统 API 层，是 SLOS 的对上接口，向应用程序层提供了系统的服务。应用程序层代码可以调用系统 API 的接口函数来获得操作系统提供的各种功能服务。系统服务层，SLOS 的系统服务层为应用程序提供了程序间通信等一些功能。当前的 SLOS 已提供互斥型信号量，消息邮箱。改进版本的系统中还会提供消息队列，消息邮箱，甚至 GUI,FS。但因为时间所限，目前只提供两项基础的服务，其它服务还有待开发。系统内核层，是整个 SLOS 的核心部分。主要功能是管理系统中的所有程序，进行程序调度以及各项资源的分配。硬

件抽象层，是和硬件相关的代码。为操作系统提供硬件接口。使整个系统可以在硬件平台上真正的运行起来。不同硬件平台的硬件抽象层是不同的，用户若想使系统运行于自己的硬件平台之上，则必须提供相应的硬件抽象层代码。

2.3程序管理

μC/OS-II 中，程序有 5 种状态。在任一时刻，程序的状态一定是在以下 5 种状态之一。

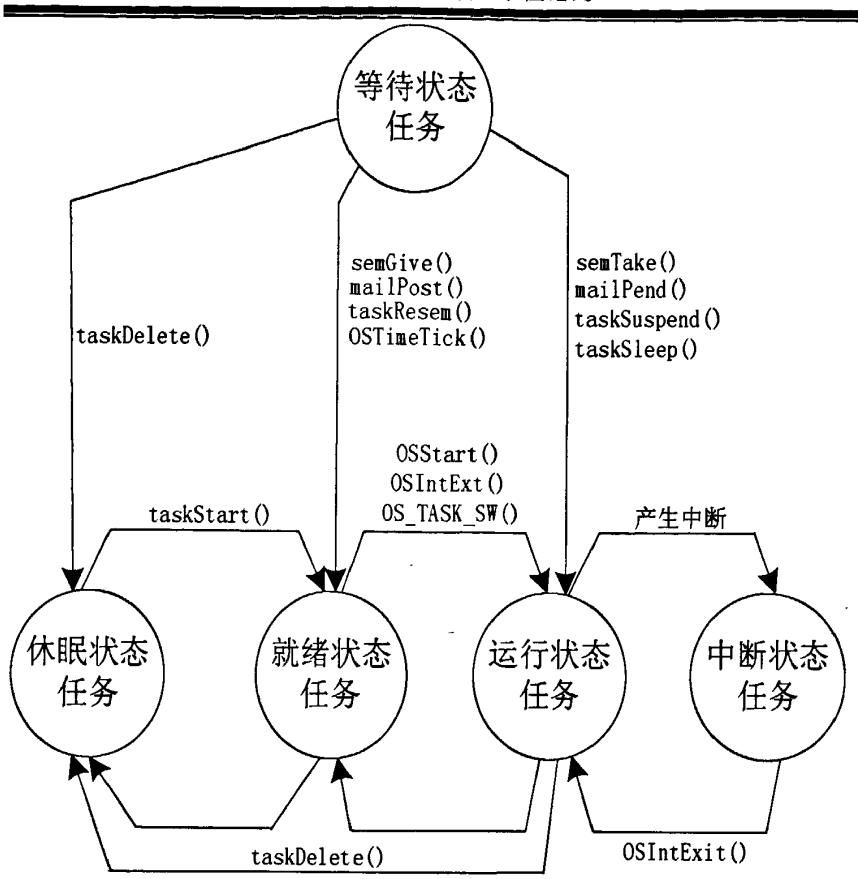
就绪态 系统运行后，调用 grammStart()，对应程序便进入到了就绪态，准备运行。就绪态的程序已经具有了除 CPU 以外的所有运行必须条件，只是在等待系统调度使程序得到 CPU 资源，进入运行态。

等待状态　导致一个程序进入到等待状态的原因有很多个，具体来说有程序自身申请延时，程序被挂起，程序等待邮箱消息，等待互斥型信号量。程序可能会因为各种原因，要等待某些事件的发生才可以再继续运行，则此程序处于等待状态。

运行态 μC/OS-II 是一个多程序操作系统，宏观上看所有程序都是并发的，都是同时在运行。但是微观上，某一时刻只有一个程序可以占有使用 CPU 资源在运行，这个程序就是处于运行态。任何一个高优先级的程序就绪，或者是当前程序统计轮转时间片已经用尽，都会使一个运行态的程序被迫放弃 CPU 资源，转入到就绪态。

中断服务态　正在运行的程序是可以被中断的，除非该程序将中断关闭了。被中断的程序于是进入了终端服务态(ISR running)。响应中断时，正在执行的程序被挂起，中断程序获得了 CPU 的使用权。中断过程中会因为中断带来的原因而使一个或者多个程序进入就绪态。因为此原因，在退出中断前，μC/OS-II 进行一次程序调度检测，以决定是否需要进行程序切换。如果中断导致更高优先级的程序就绪，则此程序会被调度执行；否则，原来被中断的程序将继续执行。

睡眠态 是指一个程序的代码已经编译到系统中，但还没有得到运行所必须的资源。或者程序已经被创建，得到了运行所必须的资源，但还没有加入到就绪位图中，即还没有参与系统中程序的调度。



2.1 5种程序状态

2.4程序调度

调度是指在有限的处理单元上对具有某些已知特征的程序集执行顺序的设计。在多程序嵌入式系统中，单纯通过提高处理器速度无法保证对重要性个不相同的程序进行合理地调度。这种程序调度只能由优化编写的系统软件来完成。在设计和实现嵌入式实时软件过程中，抢占多程序是一个普遍使用的结构

操作系统在下面的情况下进行程序调度:调用API 函数(用户主动调用)和中断(系统占用的时间片中断OSTimeTick()、用户使用的中断)。一种是当时钟中断来临时,系统把当前正在执行的程序挂起,保护现场,进行中断处理,判断有无程序延时到期,若没有别的程序进入就绪态,则恢复现场继续执行原程序。另一种调度方式是程序级的调度即调用API 函数(由用户主动调用)，是通过发软中断命令或依靠处理器在程序执行中调度。当没有任何程序进入就绪态时,就去执行空程序。新程序的投入运行是需要依据一定的调度算法对所有程序进行选择，而调

度算法及其改进就是本文主要讨论的。

```
if MIN_PRIO <= 63    //如果总的优先级数小于 64，则采用 8*8 的就绪位图
INT8U   y;
y = OSUnMapTbl[OSReadyGrp];
OS_ENTER_CRITICAL();
SLOS_Prio_highready = (INT8U)((y << 3) + OSUnMapTbl[OSReadyTbl[y]]);
else
INT8U   y;
INT16U *ptbl;
OS_ENTER_CRITICAL();
if ((OSReadyGrp & 0xFF) != 0) {
    y = OSUnMapTbl[OSReadyGrp & 0xFF];
} else {
    y = OSUnMapTbl[(OSReadyGrp >> 8) & 0xFF] + 8;
}
ptbl = &OSReadyTbl[y];
if ((*ptbl & 0xFF) != 0) {
    SLOS_Prio_highready = (INT8U)((y << 4) + OSUnMapTbl[(*ptbl & 0xFF)]);
} else {
    SLOS_Prio_highready = (INT8U)((y << 4) + OSUnMapTbl[(*ptbl >> 8) & 0xFF]
    + 8);
}
  #endif
```

从代码中可以看到，系统对 8*8 和 16*16 就绪位图采取的算法是不同的。

若系统配置中优先级不多于 64 个，即采用 8*8 就绪位图时，为了得到当前最高就绪程序在就绪位图的哪一行，系统将 OSReadyGrp 作为进行查询索引带入反查数组,。然后将位图中的此行 8 位一字节作为索引带入反查数组查询，然后经过一些换算，得到当前最高优先级就绪程序的优先级。

若系统配置为优先级总数大于64时，系统会自动采用16*16的就绪位图。此时进行反查的算法与8*8的大致相同，只是针对16位的宽度做了一些算法上的改进。

2.5创建程序

创建一个程序的大致流程为：申请 TCB 结构体，申请堆栈空间，堆栈初始化，TCB 初始化，更新系统程序总数计数。还要调用用户关于创建程序的 HOOK 函数，如果有必要。

grammCreate()负责程序的创建工作。grammStart()负责将创建的程序插入到就绪位图中。有关这两个 API 的详细信息请参见系统 API 手册。

grammStart()将创建的程序 TCB 插入到就绪链表后，并判断操作系统是否已经进入多程序状态(OSRunning == TRUE)，如果进行程序调度检测，否则直接返回。

申请堆栈空间，因为统一配置下程序堆栈个数个程序的数量是相同的。所以系统用 TCB 的程序号(GrammId)来标识相对应的堆栈。

创建人物过程中，当成功申请得到 TCB 结构体后，得到结构体中 GrammId 数据项的值(系统初始化时已赋初始值)，然后根据此 Id 取得堆栈数组中相应的堆栈元素，此堆栈空间便是当前创建人物的堆栈空间。

此处申请 TCB 结构体即从空闲 TCB 链表取出一个结构体，如果系统已经没有 TCB 结构体资源则返回错误。申请 TCB 结构体，已经将所有 TCB 结构加入到空闲 TCB 链表中， 在进行系统初始化时。

堆栈初始化，程序创建时要对其堆栈进行初始化，使其看起来就像是刚刚经历过一次程序切换，这样才可对程序进行调度切换。因为堆栈内要保存的内容依具体 CPU 硬件而定，所以此部分代码属于硬件抽象层。

2.6程序切换

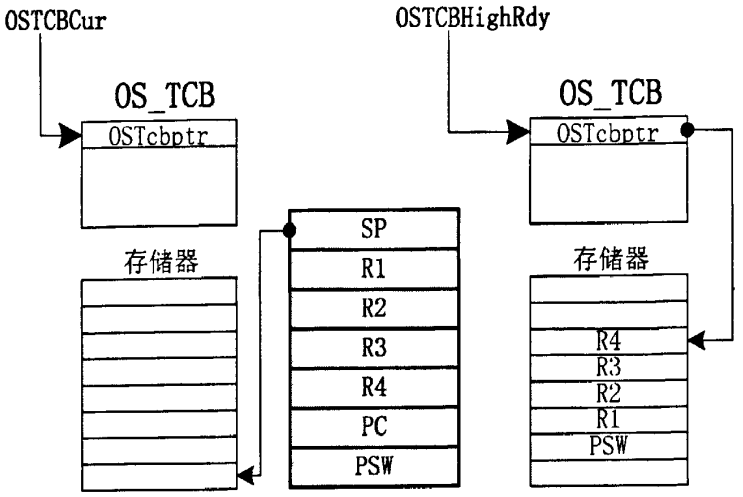
当系统调度器确定有更高优先级的程序就绪了，系统接下来会进行程序切换以保证总是最高优先级的就绪程序。

程序上下文是指程序运行的环境。例如，针对 x86 的 CPU，程序上下文可包括程序计数器、堆栈指针、通用寄存器的内容。系统中，程序的上下文保存在程序堆栈中，而程序的堆栈由 TCB 中的堆栈指针存储。这样，就保证了程序切换的顺利进行。这也是创建程序时须要针对硬件对堆栈进行初始化的原因。

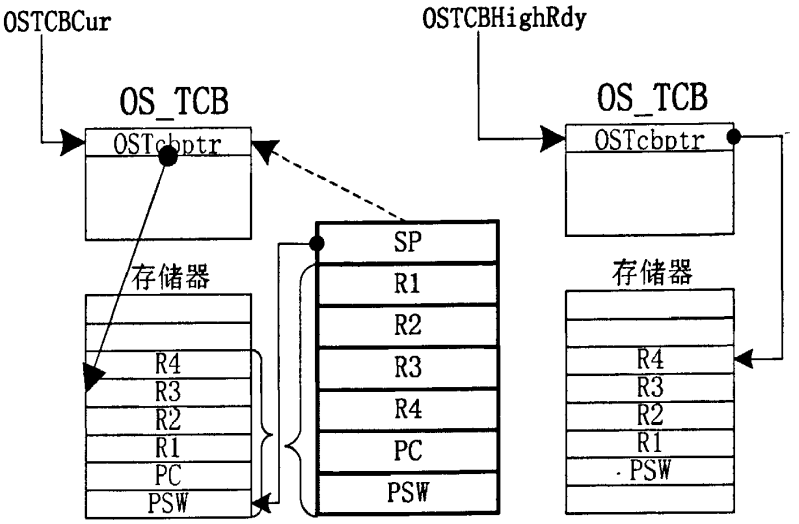
程序的切换，也就是程序上下文的切换，这是多程序操作系统并发运行的基础。

下面用几个图例来显示出程序切换的整个过程。装载将要运行的程序的堆栈

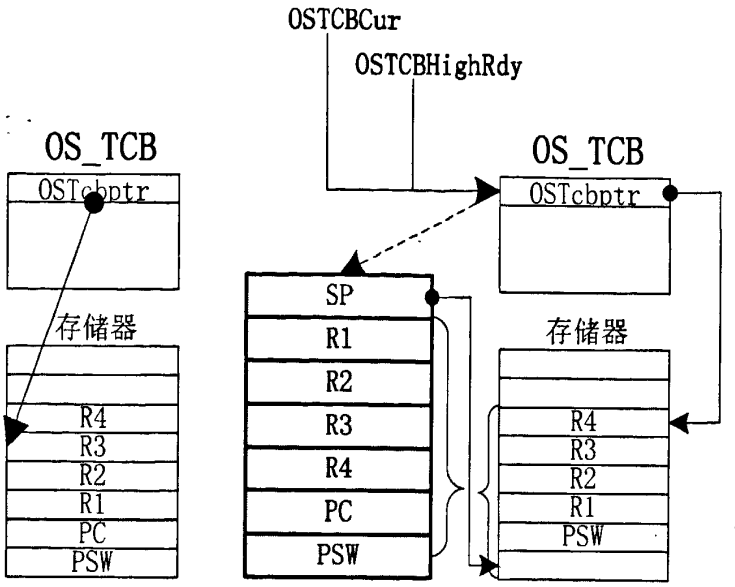
内容到 CPU 寄存器，最后跳转到切换后的程序运行。切换时，先将当前程序 CPU 各寄存器的值保存到堆栈，并更新 TCB 中堆栈指针的值一遍此程序的下一次运行。



2.2 压入堆栈



2.3 切换程序



2.4 返回执行

2.7互斥型信号量

如打印机，通信串口，一片内存空间等这一部分系统资源，要求程序必须独占式的获取享有。提供了互斥型信号量(mutex)可以实现共享资源的独占式处理。另外，互斥型信号量还提供了防优先级逆转的特性。

关于互斥型信号量的系统API有三个，它们分别是semmCreate()，semmTake()以及semmGive()。

使用互斥型信号量前必须要先用系统API semmCreate()来创建一个互斥型信号量。该API将向系统申请一个事件控制块来用作互斥型信号量恩恩数据结构体，并对此数据结构作适当的初始化操作。

当程序调用semmTake()传入要占有互斥型信号量的ID(tid)，若系统已经有其它程序占有此mutex，则当前程序会被挂起等待mutex。若还没有其它程序占有，则此程序将占有此互斥型信号量，进入临界区继续运行，实现对临界资源的访问。其它程序则必须等待此程序调用 API semmGive()释放互斥型信号量后才可以访问相应的临界资源。

已经有多个程序在等待此信号量是因为当一个程序释放了互斥型信号量。关于如何找到等待程序中优先级最高的程序，其实很简单：每一个事件控制块中都有一个类似就绪位图的位图存在。当然是让等待程序中优先级最高的那一个就绪运行，按照实时系统的要求。

```
typedef struct os_event {
    INT16U EventId;
    INT8U  OSEventType;                    //事件控制块的类型
    void    *OSEventPtr;                   //存储邮箱的邮件或者消
息队列的队列控制块
    INT16U OSEventCnt;                     //对信号量的计数
#if MIN_PRIO <= 63
    INT8U  OSEventGrp;                     //就绪标志组
    INT8U  OSEventTbl[OS_EVENT_TBL_SIZE];  //就绪标志位图
#else
    INT16U OSEventGrp;                     //就绪标志组
    INT16U OSEventTbl[OS_EVENT_TBL_SIZE];  //就绪标志位图
#endif

    OS_TCB *SLOS_tcb_by_prio[MIN_PRIO + 1];//优先级分级链表的各优先
级的表头指针

#if GRAMM_STATA_EN > 0                     //如果允许程序信息统计
    INT8U  State;
#endif

#if OS_EVENT_NAME_SIZE > 0
    INT8U    OSEventName[OS_EVENT_NAME_SIZE];//事件的名字
#endif
} OS_EVENT;
```

2.5 就绪位图

```
int Tid;

void main()
{
    ...
    Tid = semmCreate();             //创建互斥型信号量
    ...
}

void SampGramm1(void *pdata)
{
    for(;;){
        ...
        semmTake(Tid);              //申请互斥型信号量
        /*访问临界资源代码*/
        semmGive(Tid);              //释放互斥型信号量
        ...
    }
}
void SampGramm2(void *pdata)
{
    for(;;){
        ...
        semmTake(Tid);              //申请互斥型信号量
        /*访问临界资源代码*/
        semmGive(Tid);              //释放互斥型信号量
        ...
    }
```

2.6 互斥信号量实例

2.8消息邮箱

系统提供了消息邮箱用来在程序间传递消息。系统的消息邮箱发送的“消息”实际上是一个变量，它是指针类型的。一般说来，该指针变量存储一个包含了“消息”的特定数据结构。当然，如果是一个十分简单的消息的话，用户也可以选择将消息内容直接赋值于所要传递的指针。程序间通信时操作系统最基础的功能。

和互斥型信号量一样，系统提供了三个 API 来供用户使用消息邮箱。这三个 API 分别是：mboxCreate(),mboxPost(),mboxPend()。

消息邮箱的实现机制和互斥型信号量大致相同，同样是用一个事件控制块来作为邮箱的内存控制块。

若邮箱还没有邮件存在，系统会检测该事件控制块的等待位图中是否已经有程序在等待。若有，则找出等待程序中优先级最高的使其就绪，并将消息传递给次程序。如果没有程序在等待消息，则系统会将此消息暂存于事件控制块的 OSEventPtr 数据成员，等待其他程序来取走邮件消息。

当使用 mboxPost()发送一则消息，系统首先根据发送过来的 tid 找到对应于邮箱的事件控制块，然后检测邮箱事件控制块的 OSEventPtr 是否为空，若不为空值，则表明当前邮箱已经有消息存在，mboxPost()会直接返回邮箱已满的错误代码。

系统会根据传递的 tid 参数找到相应邮箱的事件控制块，当使用 mboxPend()申请接收一个消息时。继而检测其 OSEventPtr 数据成员判断是否已经有消息邮件存在有邮箱中。则直接返回消息指针，如果已经有消息存在。如果当前邮箱中并没有邮件存在，则当前接收邮件的程序会被置入邮箱事件控制块的等待位图中，其 TCB 也会被加入的系统延时链表中进行延时。

上面说到“其 TCB 也会被加入的系统延时链表中进行延时”。在系统配置中有一个 EVENT_DELAY 选项，此选项来控制等待事件控制块的程序是否要进行延时等待，具体的延时时间是多少。

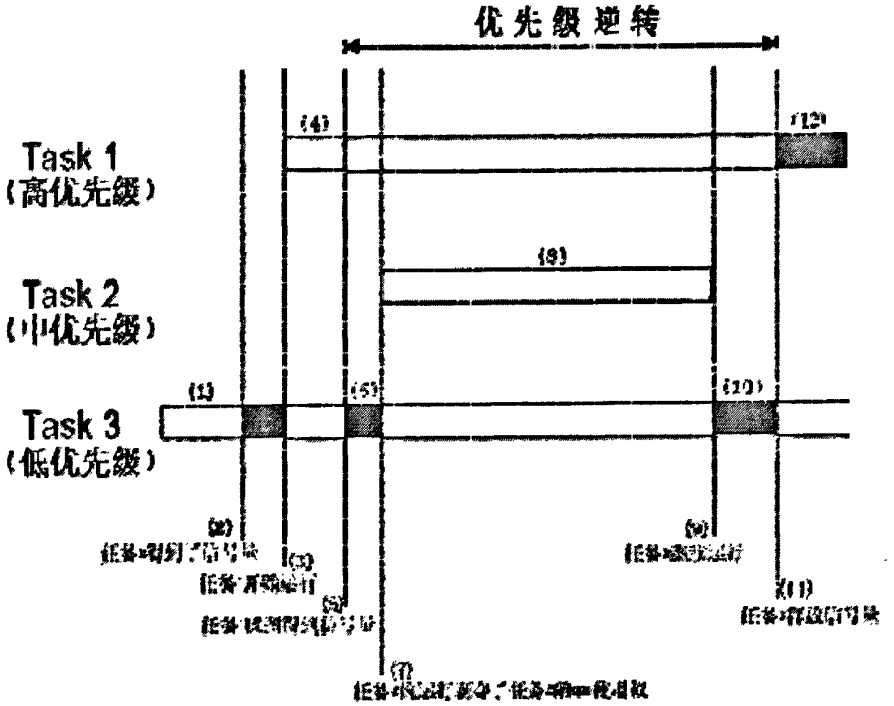
若配置了具体的延时时间，而申请接受邮件的程序并没有在此时间限定内接收到消息，则此程序会得到一个接收超时的错误消息，而不是继续在原地一直等下去。这样的设计是为了防止系统出现死锁现象。

```
int mid;
void main()
{
    ...
    mid = mboxCreate();              //创建消息邮箱
    ...
}


void SampGramm1(void *pdata)
{
    char *mesg = "This is a message.";
    for(;;){
        ......
        mboxPost(mid,mesg);
        ......
    }
}

void SampGramm2(void *pdata)
{
    Char *mesg;
    for(;;){
        ...
        mesg = (char *)mboxPend(mid);
        ...
    }
}
```

2.7 消息邮箱

2.9防优先级逆转

实时操作系统中经常会出现的问题是优先级逆转问题，在实时操作系统中。当优先级最低的程序申请得到信号量后，最高优先级的程序 1 为得到优先级不得不暂时挂起。而如果此一段时间内中等优先级的程序 2 开始运行，则会出现问题：实时操作系统首先要保证最高优先级的程序总是得到运行，而此时就绪中程序 1 的优先级高于程序 2 的优先级，而得到运行的却是优先级相对较低的程序 2。



2.8 优先级反转

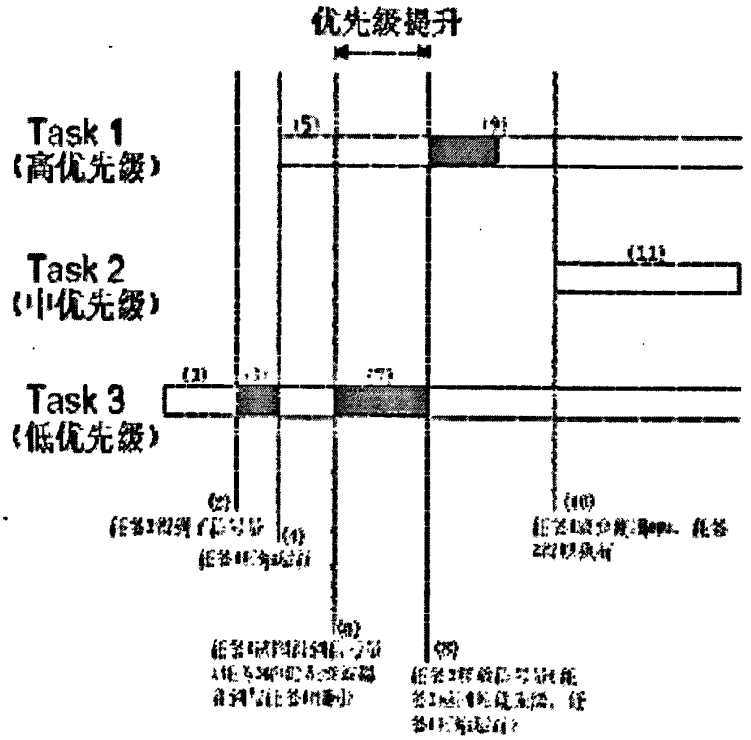
避免优先级反转的两种常见方法是：优先级继承协议(Priority Inheritance Protocol, PIP)和优先级上限协议(Priority Ceiling Protocol, PCP)，其中优先级上限协议又可以被称做优先级天花板协议。

两种方法的前提都是优先级可改变的，而且是可以动态改变的。

因为改进系统 slos 支持动态改变优先级，并且支持同优先级时间片轮转调度，所以系统采用了优先级继承的方法来防止优先级逆转的发生。首先一个进程申请得到了信号量。其后，当该进程开始申请信号量，系统发现该进程已经得到信号量且该进程的优先级低于进程 1。

此时，系统会将进程 3 的优先级提升到和进程 1 一样。当进程 3 释放信号量后，进程 1 会紧接着开始运行，这样就避免了程序 2 会先于程序 1 运行，从而避

免优先级逆转的发生。



2.9 防优先级逆转

2.10 防死锁产生

死锁也叫抱死(deadlock 或 deadly embrace)，两个或两个以上的进程都将无法推进下去.此时称系统处于死锁状态或系统产生了死锁,这些永远在互相等竺的进程称为死锁进程，在执行过程中,因争夺资源而造成的一种互相等待的现象,若无外力作用.

当某个进程提出申请资源后，因为资源占用是互斥的，永远分配不到必需的资源而无法继续运行，使得有关进程在无外力协助下，这就产生了一种特殊现象死锁。

例如，如果程序 A 已经申请了互斥型信号量，程序 B 也申请同一个互斥型型号量，程序 B 会被挂起来等待 A 释放信号量。而程序 A 这时如果要等待程序 B 的一个消息邮件才能执行，也被挂起了，那么程序 A 和程序 B 就处于相互等待的状态，谁也无法继续向下执行，这样两个线程就发生了死锁现象。

一种情形，每个线程都在等待被其他线程占用并堵塞了的资源,此时执行程序中两个或多个线程发生永久堵塞（等待）。

造成死锁应该满足以下四个先决条件，缺少任何一种就不会出现死锁：

一是互斥条件：一个资源每次只能被一个进程使用。

二是申请与保持条件：一个进程因申请资源而阻塞时，对已获得的资源保持不放。

三是不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。

四是循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

其原理是破坏上述产生死锁条件的第四个条，即在系统中，此程序进入了延时状态当一个程序因产生的申请没有被满足而被挂起后。，则此申请会返回给程序一个超时的错误报告，当超出指定时间而人物的申请人未得到满足。

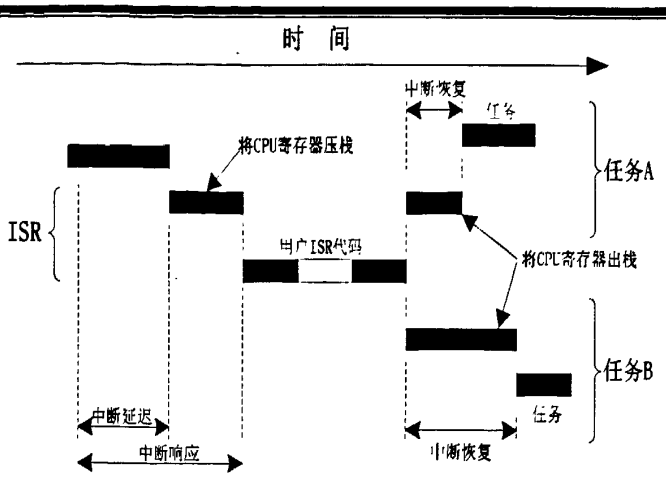
由此可以看出系统中程序并不会在一个申请上永远的等待下去，所以改进系统SLOS 中也就不会产生死锁。系统具有防止死锁发生的特性。

2.11 时钟中断

时钟中断是一个操作系统的心脏脉搏。间隔有序的时钟中断是系统持续稳定运行的前提保障。用以向程序提供延时功能。时钟节拍也是系统进行程序调度的重要依据和触发点，系统依靠时钟中断来进行系统计时。

当前正在运行的程序会转入中断态暂停运行，当中断产生时，CPU 立即去执行中断服务代码(ISR)。进行中断计数，中断处理代码应该首先调用系统的OSIntEnter()通知系统已进入中断。当中断处理代码执行完所有必要处理，还要调用 OSIntExit()通知系统中断处理完毕。因为对中断次数进行了计数，所以SLOS 是支持中断嵌套的。

OSIntExit()在中断处理的最后被调用。以检测在终端过程中是否有比中断态程序更高优先级的程序就绪，它会进行调度检测，是否要进行程序调度，当前中断态的程序是否已经用尽同级轮转的时间片。要进行程序调度，如果有更高优先级程序就绪，或者当前程序的同级轮转时间片已经用尽。



2.10 系统中断处理过程的流程图。

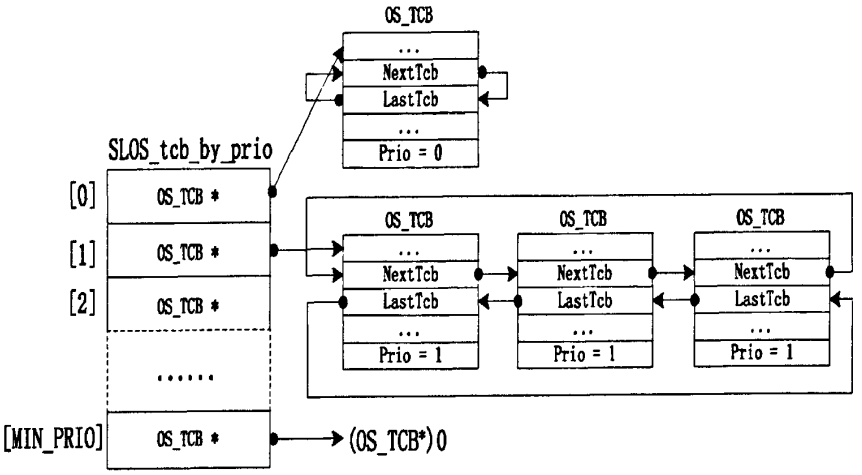
2.12 时间片轮转

时间片指程序允许运行的时间。

时间片轮转调度，指的是每个程序都被分配一个时间段，CPU 将被剥夺并分配给另一个程序，也就是说该进程或者程序如果在时间片结束时程序还在运行。

每一个优先级的双向循环链表就是为实现同优先级的时间片轮转而设计的。系统支持同优先级时间片轮转调度的算法。正如就绪分级链表所说的，SLOS 采用分级就绪链表来管理就绪的程序，如 2.11 所示。

程序控制块 TCB 里面保存了每个程序的时间片长度，若系统配置了包含同级别轮转调度功能。当每次时钟中断发生之后，该系统都会自动在相关处理函数中对于时间片计数减 1 并且检测这个计数是否已经减至为 0。时间片计数会被该系统重新定义初始值，紧接着系统会检测系统中是否有同优先级的程序存在，如果当前程序已经用尽它的时间片，有同优先级程序存在，那么循环链表中的下一个程序会被运行，而当前程序会转为就绪态，它会等待下一次运行。系统就这样完成了一次同优先级时间片轮转调度。


OS_TCB
...
NextTcb
LastTcb
...
Prio = 0
SLOS_tcb_by_prio
[0]
OS_TCB *
[1]
OS_TCB *
[2]
OS_TCB *
.....
[MIN_PRIO]
OS_TCB *
(OS_TCB*)0
OS_TCB
...
NextTcb
LastTcb
...
Prio = 1
OS_TCB
...
NextTcb
LastTcb
...
Prio = 1
OS_TCB
...
NextTcb
LastTcb
...
Prio = 1

2.11 时间片轮转

第三章 调度算法

所有操作系统的实时调度算法能够按调度算法进行分类，可以做出如下三种分类：按进程的优先级别进行判别的调度算法、按照处理器调度算法占用比例的共享式的调度算法、按照进程执行时间的长短的进程调度算法。

在每次进程调度时，调度器总是调度那个具有最高优先级的程序来执行，这是基于优先级的调度算法给每个进程分配一个优先级。基于优先级的调度算法可以分为静态调度和动态调度两种类型，根据不同的优先级分配方法。

下面将介绍几个静态调度算法和动态调度算法，对它们进行分析和总结。

3.1静态调度

静态调度算法是一种常用的有效的调度算法，在日常试验和工业应用中经常使用。该算法的调度对系统资源的占用比较小，在编程中程序员实现起来也比较简单，系统如果超载的话算法执行可预测性好，但是应为算法比较简单还是有很大的局限性。

例如：那些资源的利用率低、系统支持的优先级个数限制、以及自适应性和灵活性比较差等局限性。静态调度是在系统开始运行前进行调度的，严格的静态调度在系统运行时无法对程序进行重新调度。把程序分配到各个处理机是该静态调度的目标是，并要对每一处理机排列出所要运行程序的静态运行顺序。

3.1.1 速率单调调度(Rate Monotonic Scheduling)

Rate Monotonic Scheduling ，简称RMS。将最高优先级赋予最高执行频率的程序，以单调的顺序对剩余的程序分配优先级是速率单调调度。高优先级的程序就绪后立即抢占正在运行的程序，这是因为采用抢占式的调度方式。速率单调算法已被证明是静态最优的调度算法，在程序的周期等于其截至时间的条件下。Liu 和Layland 的理论证明了下列公式，即用RMS 调度的独立的周期性程序总能满足其截止时间的要求。

$$C1/T1 + C2/T2 + \cdots + Ci/Ti + \cdots + Cn/Tn \leqslant U(n) = n(2n - 1)$$

式中：Ci 为程序i 的最坏执行时间；Ti 为程序i的周期。RMS 算法的优点是开销小，灵活性好，可调度性测试简单。

3.1.2 截止时间单调调度(Deadline Monotonic Scheduling)

DMS 是在速率单调调度的基础上发展起来的。按截止时间来分配不同的是程

序的优先级。截止时间短的程序优先级高，截止时间长的程序优先级低。截止时间调度具有与速率单调算法相同的优点。但DMS 算法放松了对程序的周期必须等于其截止时间的限制。在程序的截止时间小于或等于其周期的情况下，DMS 已被证明是静态最优的调度算法。

3.2动态调度

动态调度可以是抢占式的或非抢占式的。在嵌入式实时系统中动态调度依赖于程序的优先级。优先级可以静态分配或者依据不同的特征参数，特征参数包括截止时间、空闲时间或关键性(即程序的重要程度或者价值) 等进行动态分配。当检查到一事件时，动态抢占式算法立即决定是运行与此事件相关的程序，或者继续执行当前程序;对于动态非抢占式算法，它仅仅知道有另一个程序可以运行，在当前程序结束后，它才在就绪程序中选择一个来运行。前者用优化的标准动态调动程序，从而减少因超过截止时间而失败的程序数，后者检查系统中每一个程序，看是否有新程序可以加入调度列表，从而满足其截止时间的要求。非抢占式内核的优点是易于分析、实现，利于保存额外的上下文切换，节省抢占式内核中因为采用互斥机制带来的开销，其缺点是可能使某些程序不能满足其截止时间的要求。几个比较著名的商用嵌入式实时操作系统，如QNX 、VxWorks 提供的都是“抢占式程序调度”。

3.2.1 最早截止时间优先算法(Earliest Deadline First)

一个动态优先级驱动的调度算法是抢占式EDF调度算法，其中分配给每个程序的优先级根据它们当前对最终期限的要求而定。当前申请的最终期限最近的程序具有最高的优先级，而申请最终期限最远的程序被分配最低优先级。不存在处理器的空闲时间，这个算法能够保证在出现某个程序的最终期限不能满足之前。该算法基于以下假设：

1) 任何程序不应该存在不可抢占的部分，并且抢占的代价可以被忽略不计；

2) 内存、I/ O 和其它资源申请可以忽略，只有处理器申请是有意义的；

3) 所有的程序都是无关的，不存在谁先谁后次序的问题约束；

4) 程序的相对最终期限与它的周期相等。

因此抢占式EDF 调度算法对于给定周期性程序集可调度性的充分必要条件为：

$$\sum_{i=1}^{n}\frac{e_i}{p_i}\leq 1$$

其中，ei ，pi 分别为程序集中程序i (1 Fi Fn) 的执行时间和周期。因此而知,抢占式EDF 调度算法最大的优势在于，只要处理器的利用率不超过100％，对于任何给定的程序集，就能够保证它的可调度性。但在系统负载较重时，系统性能急剧下降，引起大量程序错过截止期，甚至可能导致CPU 时间大量花费在调度上，在这时系统的性能还不如FIFO(First In First Out) 方法。EDF 算法在系统的负载相对较低时非常有效。

3.2.2 最小松弛时间算法(Least Slack Time First)

LSF 算法计算程序的松弛时间，其中程序截止时间与剩余执行时间之差为松弛时间。该算法在任意时刻把最高优先级分配给具有最小松弛时间的程序，以此来保证紧急程序的优先执行。然而，因为等待程序的松弛时间是严格递减的，其等待执行的缓急程度也随时间越来越紧迫，因此在系统执行过程中，等待程序随时可能会抢占当前执行的程序。LSF 算法造成程序之间的频繁切换或称为颠簸(thrashing)现象较为严重。颠簸现象增大了系统开销，并限制了LSF 算法的应用。

3.3静态和动态调度算法的选择

开销主要包括运行开销与调度开销。运行开销与队列分析和从调度队列中增加、删除程序相关。嵌入式实时系统中资源是非常有限的，所以开销要尽可能小。程序调度器在一个周期内不得不对一个程序进行两次选择，是因为每个程序在一个调度周期内至少被阻塞和唤醒一次。它必需事先进行设计，静态调度对时间触发系统的设计很合适，选择众多参数的综合考虑，一般说来需要更大范围地设计用来处理最不可能的事件。动态调度适合设计事件触发的系统，在执行期间能够动态做出决定，并且它在资源利用方面比静态调度有更大的潜力。根据上面提到的几个相关的算法比较与分析。如：RMS 算法，它根据程序的执行频率设置优先级，执行频率最高的程序不一定是最重要的，但有较小的运行开销。而像EDF 和LSF 这样的动态调度策略需要较高的调度开销来在运行时对操作进行评估。运行上下文的保存和恢复及相应的中断等额外开销大于程序正常结束，启动另一个程序的额外开销，因为抢占发生时。另外，对那些错过最终期限的操作不提供有效

控制手段在动态调度策略在调度量过大的时候。所有操作的实时安全性就降低了，当一个操作被加入调度队列以获得更高利用率时。因此对所有操作来说，当系统资源过度使用时，执行错过截止期的使用几率就增加。通过上述分析，很容易看出不论使用何种调度策略，都是各有优缺点的。

第四章 典型调度算法的改进

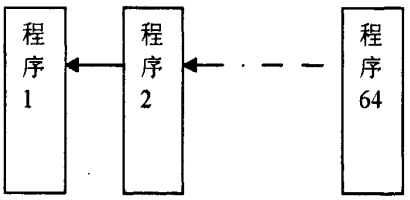
这个系统有些限制虽然简化了程序管理系统，但也给大型应用带来了很大的不便。μc/ os - II 是目前流行的一个开源免费的实时内核。μc/ os - II 2.62 可以管理 64 个程序，其中 8 个由系统保留，这样自己的应用程序最多也就拥有 56 个程序，而且赋予每个程序的优先级必须是不相同的。

4.1结合 RMS 和 EDF 的优先级分配算法

得到一个改进的优先级分配算法，通过对静态和动态调度算法的分析、比较，可以结合RMS 算法具有较小运行开销和EDF 算法具有较小调度开销的特点，即选择截止时间最短(即优先级最高) 的程序运行，在多个程序要求调度时；

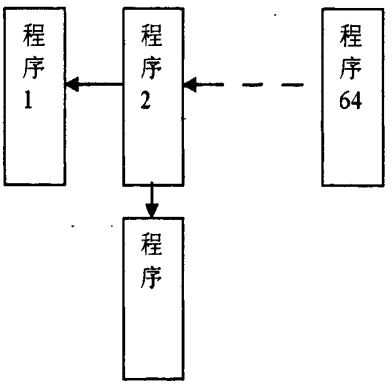
选择运行频率最高的程序运行，若程序优先级相同。为了突破原系统对程序数量的限制，并且去除对每个程序必须有不同优先级的要求。算法改进的关键之处是对优先级相同的程序之间建立散列表。在散列表中按照程序执行频率进行降序排列，这样就能保证该优先级散列表中的第一个程序就是在此优先级上执行频率最高的程序。

原μC/OS- II程序队列为：



4.1 原μC/OS- II程序队列

使用新算法后建立的程序队列：



4.2 新算法后建立的程序队列

程序create、程序delete、更改程序value、挂起程序、恢复程序resume、与程序堆栈操作相关的系统接口和其他一些接口函数等包含在程序管理系统调用接口中。

```
cxGramCreate
{
检查优先级合法性;
堆栈建立cxStackInit () ;
if (申请的优先级已经被申请)
{

看一下频率;
插入到列表中;

}
else
{
加入List列表中;
}
}
OSSched
{ if ( (OSNest | OSIniteNest) = = 0)
查找就绪状态下优先级最高的程序;
if (此优先级> 当前运行程序的优先级)
if (此优先级程序数> 1)
{ 在该优先级散列表中查找执行频率最高的程序;程序切换;
}
else
{
程序被切换出去;
```

```
}
}
```

4.2截止期最早优先算法

截止期最早优先算法,它不计算程序的运行时间是不是已经超时,系统执行这种程序就会造成系统执行的效率变低，执行时间变长,如果系统的时序要求非常严格，那么出现上述情况还会造成严重的后果。

优先级最高的进程正常运行,系统如果要保证在调度时超过截止期的程序不予调度是可达截止期最早优先算法是对截止期最早优先算法的一种改进。出现下面这种情况就要删除这个程序，如果计算的结果小于零，说明程序超时。

系统第一它是考虑系统的现在正在执行的时间、程序的调用所用时间长短进行判断，来更新程序的截止时间。

如果程序它的执行没有超时，就会与截止期优先算法中程序的处理方法相同。在本文所述可达截止期最早优先算法改进中，需在μC/OS- II 系统中增加表1所示的几个项目。

项目	功能
INT8U deadline	在 OS_TCB 增加的数据项，用来记录功能的截止期。在程序调度时，当前系统时间 OSTime 和程序被调用时间 CSTime 的差更新 Deadline.
INT8U cstime	在 OS_TCB 增加的数据项纪录程序首次调用的系统时间。
INT8U csREC[64][3]	记录每个程序优先级的调用时间、截止时间、优先级附加值（记录优先级的变化）。程序创建或销毁时要根据其截止日期分配分配合理的优先级并插入此数组，同时用数组来更新相应的值。

4.3 可达截止期优先算法改进中添加的项目

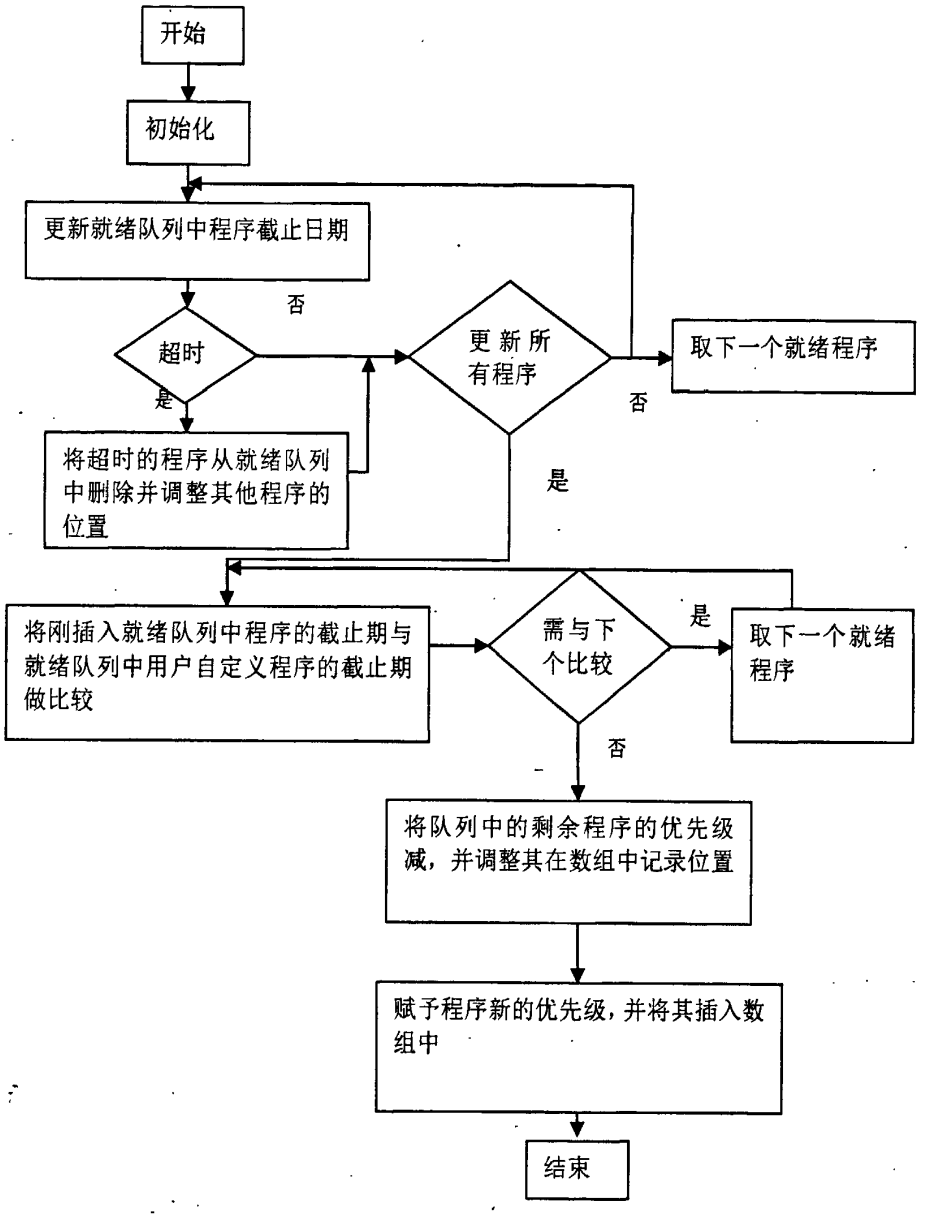
最先到达停止期优先算法要决定那些调度过程中不会超时的程序,程序的停止时间不能够超过时间,需要为μC/OS- II 系统编写OSPrioCreate函数来建立或者是删掉程序的时候处理程序优先级调整的工作。

对程序截止期是否超时的判断。该模块流程如图1 所示。

对于程序就绪队列进行变化调整的时候，这个程序第一在就绪队列中对程序的优先级进行调整并且写下程序优先级的变化状态。

在执行此函数后，就绪程序队列中的程序的优先级可能会改变，因此还需要在系统中添加prio_adjust()函数，该函数应用μC/OS- II 系统原有的OSGrammChangePrio()函数来对就绪程序更新，代码如下所示：

```
void prioadjusted(){
int i;
for(i=0;i<LOWEST_PRIO- 8;i++){
if(csRec[i][0]<0){
OSGrammChangedPrio(i,i+csRec[0]);
}
}
}
```



4.4 可达截止期最早优先算法图

为防止多个程序同时调用OSGrammPrioCreate () 函数造成混乱，这段代码应按临界资源来处里，需要在调用前关中断，调用后再开中断。

第五章 增加可管理程序式调度算法改进

5.1背景

μC/OS Ⅱkernal它可以成功的进行处理64个进程，因为它只有八位二进制数进行管理，而且其中两位进行预留。在64个进程中系统保留了8 个程序作为自己的程序管理，其他进程不得申请，其他用户若想申请优先级别，最多可以拥有56 个优先级进度。

当工程进度多，管理的进程也会随之增多，在μC/OS Ⅱ操作系统上运行的程序数目不断增加时，经常的会出现程序数目超过56个，如果出现现在这种情况，这个工程就无法使用ucos作为操作系统进行管理。

为了避免造成前期大量的开发工作都作废，我们应该想方设法把实时性非常好的系统增加其优先级别，这样就可以保证实时性，又可以增加优先级别。那么ucos的应用性就大大增加了。如果考虑到上述的这些情况，在原有的优先级调度算法基础上，根据μC/OS Ⅱ本身的程序可扩展性，提出了两种可行的大量增加可管理程序的算法：

原理：

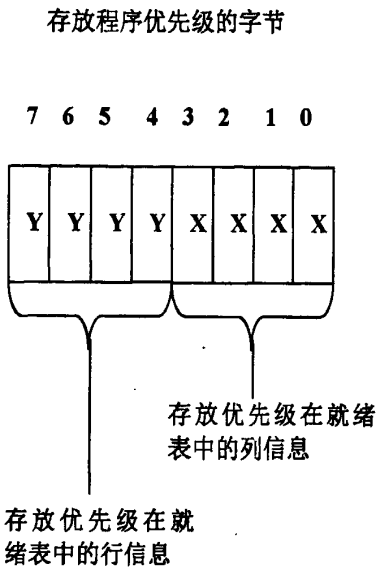
μC/OSⅡ的实时内核就是占先式内核。当系统响应时间很重要时，要使用占先式内核。能够确定最高优先级的就绪程序什么时候可以执行是它的特点，可以得到CPU 的控制权，从而使程序级响应时间得以最优化。相应地，基于优先级的抢占式调度策略是μC/OSⅡ内核采用的。

最高优先级的程序一旦就绪总能得到CPU 的控制权，系统虽然为每个程序分配一个优先级，当一个运行着的程序使一个比它优先级高的程序进入了就绪态，当前程序的CPU 使用权就被剥夺了，或者说被挂起了，那个高优先级的程序立刻得到了CPU 的控制权。

5.2增加可管理程序的第一种算法改进方法

为增加μC/ OSⅡ内核可管理程序的数目，该算法利用μC/ OS2Ⅱ原有的优先级判定表格，重新定义了存放程序优先级的字节，重新建立程序就绪表，把64个程序扩充到256 个程序，把程序放入就绪表中，给出了新的最高就绪程序的查找算法。

5.2.1 对优先级字节的改动和规定



5. 1 程序优先级字

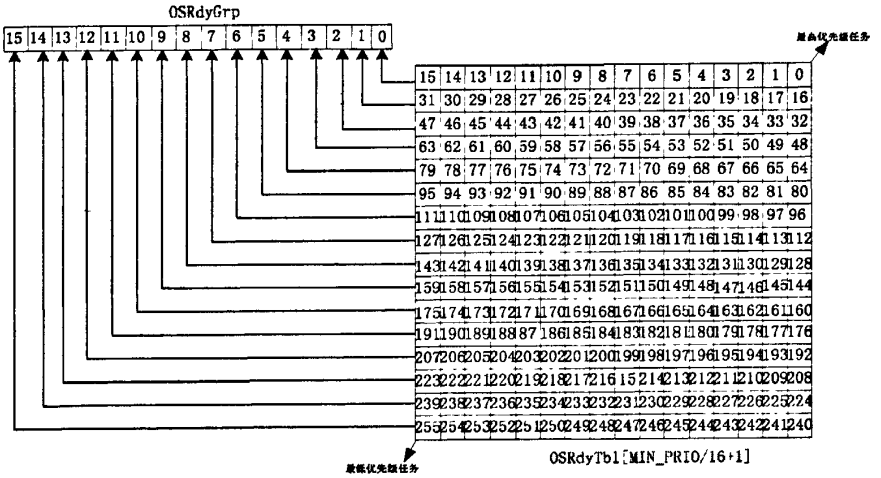
在μC/ OSⅡ中，原有的基于64 个程序调度的优先级调度算法分别用3 个比特位来定位程序优先级在就绪表(ready list) 中的行和列，即0～2 位标识该程序在总就绪表中的列信息，3～5 位标识该程序在就绪表中的行信息。因此，存放程序优先级的字节中8 个比特位只会用到6 位，而有两个比特位空闲。该算法直接扩展定位就绪程序优先级在就绪表中位置的行和列信息的比特位，使其能够区分256个不同的程序优先级。扩展后的算法规定程序优先级字节的定义如上图所示。

5.2.2 将程序放入就绪表

μC/ OS2 Ⅱ中定义的就绪表变量cxReady 和OSReadyTbl [ ]，仍然可以继续使用，减少对系统的改变，这样可增加用户的可读性，仍旧用变量OSReadyGrp 来表示优先级在就绪表中所在的行，在OSReadyGrp 中，程序按优先级分组，16 个程序为一组。OSReadyGrp 的每一位表示16 组程序中是否有进入就绪态的程序，如果存在进入就绪态的程序，则相应位置为1。使用OSReadyTbl[ ]数组(根据上面的扩展规则将该数组的大小由原来的8 位扩展为16 位) 表示优先级在就绪表中的列信息，即存放每个优先级的程序是否就绪的信息，如果某一位对应的程序

处于就绪态，则将该位的值置为1。例如，OSReadyTbl[0]对应优先级为0～15 的程序，OSReadyTbl [1]对应优先级为16～31 的程序，依次类推，OSReadyTbl[15]对应优先级为240～255 的程序。优先级为78 的程序处于就绪状态，不仅要将OSReadyTbl[4]的第14 位置1，而且要将OSReadyGrp 的第4 位置1。也就是说只要OSReadyTbl[*n*]中有一位为1，则OS2ReadyGrp 的第*n* 位就为1。变量OSReadyGrp 和OSReadyTbl[ ]之间的关系如下图 所示(下图<改进的就绪表> 中OSReadyGrp 下表格中标注的数字0～15 仅为清楚起见表示16 组程序，并非表示OSReadyGrp 中每一位的状态信息，同理，OSReadyTbl[ ]下表格中的数字0～255 也仅表示256 个程序，并非实际存放的状态信息)。

当配置系统优先级总数大于 64 时，系统自动采用 16 位宽度的位图，例如系统优先级总数为 256 时，采用所示的 16*16 就绪位图。当总数大于 64 而小于 256 时，处理方法同 8*8 位图。



5.2 16 位宽度的位图

对应于就绪位图，系统还提供一个常量数组对位图进行反向查询。OSUnMapTbl 是一个 256 字节长的字节数组，简单的说，它的功能就是对就绪位图进行了反方向的查询。将一个 8 位的字节作为索引到 OSUnMapTbl 中查询，得到的是这个字节 8 个位中最右边的值为 1 的位顺序地址。经过这样反向查询，就可以快速且时间固定的得到当前就绪位图中优先级最高的就无程序。

```
INT8U    const   OSUnMapTbl[256] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
};
```

5.3 程序就绪位图

把程序放入就绪表的程序是:

```
CXReady | = CXMap
Tbl[pr -> 4 ] ;
OSReadyTbl[prio > > 4 ] | = OSMap Tbl[prio & 0X0F] ;
```

其中,char OSMap Tbl[ ] = {01, 02, 04, 08, 10, 20, 40, 80, 100, 200, 400, 800, 1000, 2000, 4000, 8000}, 用于限制 OSReadyTbl [ ]数组的元素下标在 0 到 15 之间, prio 表示程序的优先级。

```
INT8S grammInit() //系统初始化
{
    #if GRAMM_COUNT <= 255
    INT8U i;            //初始空闲链表时使用
    #else
    INT16U i;           //初始空闲链表时使用
```

```
#endif
INT8U j;                //初始就绪位图时使用
OS_TCB     *ptcb1;          //定义循环构建链表用的两个临时 TCB 指针
OS_TCB     *ptcb2;
#if OS_EVENT_COUNT > 1
OS_EVENT   *pevent1; //定义循环构建链表用的两个临时 EVENT 指针
OS_EVENT   *pevent2;
#endif
#if MIN_PRIO<= 63     //初始化就绪位图表
 INT8U     *pReadytbl;    //定义一个存储就绪位图表的指针
#else
 INT16U    *pReadytbl;
#endif
INT16U     tid;        //错误代码，获取创建程序的返回值

#if CPU_HOOKS_EN > 0
OSInitHookBegin();
#endif

SLOS_tcb_delay_list      = (OS_TCB*)0;
SLOS_tcb_suspend_list    = (OS_TCB*)0;
OSTCBCur              = (OS_TCB*)0;
OSTCBHighReady           = (OS_TCB*)0;
OSRunning             = FALSE;
SLOS_Prio_Cur            = 0;            //当前的程序优先级初始化为0
OSIntNesting          = 0;

for(i = 0;i < MIN_PRIO + 1;i++){        //初始化分级链表的表头为0
   SLOS_tcb_by_prio[i] = (OS_TCB*)0;
}
```

```
    SLOS_tcb_free_list   = SLOS_tcb;          //初始化空闲TCB链表的头
指针
    ptcb1              = &SLOS_tcb[0];      //之所以这样做是因为一个完
整的系统至少会有两个程序
    ptcb2              = &SLOS_tcb[1];
    ptcb1->GrammId       = 0;
    ptcb1->State       = OS_STATE_FREE;
    for(i = 0;i < GRAMM_COUNT - 1;i++){       //构建空闲TCB链表
        ptcb2->GrammId    = i + 1;
        ptcb2->State  = OS_STATE_FREE;
        ptcb1->NextTcb    = ptcb2;
        ptcb1++;
        ptcb2++;
    }

    #if OS_EVENT_EN >0                        //初始化事件控制块链表,如果
需要的话
    SLOS_event_free_list = &SLOS_event[0]; //初始化空闲链表的头指针
    #if OS_EVENT_COUNT > 1                    //只有在有多余1个事件控制
块时才有必要组建链表
    pevent1    = &SLOS_event[0];
    pevent2    = &SLOS_event[1];
    pevent1->EventId = 0;
    for(i = 0;i < OS_EVENT_COUNT - 1;i++){ //构建空闲事件控制块链表
        pevent2->EventId  = i + 1;
        pevent1->OSEventPtr  = pevent2;
        pevent1++;
        pevent2++;
    }
```

```
#else                                   //如果只有一个事件控制块,那么
只需简单的处理下即可
SLOS_event_free_list->EventId = 0;
SLOS_event_free_list->OSEventPtr = (OS_EVENT *)0;
#endif
#endif

OSReadyGrp      = 0;                    //将就绪标志组以及就绪位图
全部初始化为0
pReadytbl       = &OSReadyTbl[0];
for (j = 0; j < OS_READY_TBL_SIZE; j++) {
    *pReadytbl++ = 0;
}

tid = grammCreate("free gramm",. (void *)SLOS_free_gramm, 100,
MIN_PRIO); //创建空闲程序
if(tid < 0){
    return SLOS_ERR_FAIL;
}
grammStart((unsigned char)tid);

#if GRAMM_STATA_EN > 0
tid = grammCreate("state gramm", (void *)SLOS_state_gramm, 100,
MIN_PRIO - 1);
if(tid < 0){
    return SLOS_ERR_FAIL;
}
grammStart((unsigned char)tid);
#endif
```

```
    #if CPU_HOOKS_EN > 0
    OSInitHookEnd();
    #endif

    return SLOS_ERR_NO_ERR;
}
```

5.2.3 从就绪表中删除程序

当优先级为prio 程序脱离就绪状态时，可清除OSReadyGrp 和OSReadyTbl[ ]中的对应该程序的位：

```
if ( (OSReadyTbl[prio > > 4 ] & = ~OSMap Tbl[prio & 0X0F]) = = 0)
  OSReadyGrp & = ~OSMap Tbl[prio > > 4 ] ;
```

以上代码将就绪程序表数组OSReadyTbl [ ]中相应元素的相应位清零，而对于OSReadyGrp，只有当被删除程序所在的程序组中，全组程序一个都没有进入就绪态时，才将相应位清零。也就是说,OSReadyTbl [prio> > 4]所有的位都是0时，OSReadyGrp 的相应位才清0。

```
#if OS_GRAMM_DEL_EN > 0
INT8S  grammDelete(INT16U tid)
{
    #if OS_CRITICAL_METHOD == 3          //声明关中断时保存cpu状态寄存
器的变量
    OS_CPU_SR cpu_sr;
    #endif
    OS_TCB     *tcbdelete;
    #if OS_EVENT_EN > 0
    OS_EVENT   *pevent;
    #endif
    OS_ENTER_CRITICAL();
    tcbdelete = &SLOS_tcb[tid];
    if(tcbdelete->Prio == MIN_PRIO){   //如果要删除的是空闲程序，则出
错
```

```
        return SLOS_ERR_FAIL;
    }



    #if CPU_HOOKS_EN > 0
    OSGrammDelHook(tcbdelete);
    #endif

    #if OS_EVENT_EN > 0
    if(tcbdelete->State == OS_STATE_MUTEX || tcbdelete->State ==
OS_STATE_MBOX){   //如果此TCB在事件控制块的等待链表中
        pevent = tcbdelete->Eventptr;
        pevent->OSEventTbl[tcbdelete->OSTCBY] &= ~tcbdelete->OSTCBBitX;
                //清除事件控制块的等待位图
        if (pevent->OSEventTbl[tcbdelete->OSTCBY] == 0) {
                pevent->OSEventGrp &= ~tcbdelete->OSTCBBitY;
        }
        //从延时链表中删除此TCB
        if(tcbdelete->LastTcb != (OS_TCB *)0){                    //判断
是不是链表的第一个元素
            tcbdelete->LastTcb->NextTcb = tcbdelete->NextTcb;        //
如果不是第一个
        }else{
            SLOS_tcb_delay_list = tcbdelete->NextTcb;             //如果
是第一个，要改链表头指针
        }
        if(tcbdelete->NextTcb != (OS_TCB *)0){                    //判断
是不是链表的最后一个元素
            tcbdelete->NextTcb->LastTcb = tcbdelete->LastTcb;        //
如果是，就不必设置下一个元素的LastTcb指针
```

```
        }
        //脱离事件控制块的等待链表
        if(tcbdelete->ELastTcb != (OS_TCB *)0){
            tcbdelete->ELastTcb->ENextTcb = tcbdelete->ENextTcb;
        }else{
            pevent->SLOS_tcb_by_prio[tcbdelete->Prio] =
tcbdelete->ENextTcb;
        }
        if(tcbdelete->ENextTcb != (OS_TCB *)0){
            tcbdelete->ENextTcb->ELastTcb = tcbdelete->ELastTcb;
        }
    }
    #endif


    #if GRAMM_DELAY_EN > 0
    if(tcbdelete->State == OS_STATE_DEALY){                    //如果
此TCB正在延时，则脱离延时链表
        if(tcbdelete->LastTcb != (OS_TCB *)0){                 //判断是
不是链表的第一个元素
            tcbdelete->LastTcb->NextTcb = tcbdelete->NextTcb;   //如果
不是第一个
        }else{
            SLOS_tcb_delay_list = tcbdelete->NextTcb;          //如果是
第一个，要改链表头指针
        }
        if(tcbdelete->NextTcb != (OS_TCB *)0){                 //判断是
不是链表的最后一个元素
            tcbdelete->NextTcb->LastTcb = tcbdelete->LastTcb;   //如果
是，就不必设置下一个元素的LastTcb指针
```

```
        }
    }
    #endif

    if(tcbdelete->State == OS_STATE_READY){                    //如果
此程序是在就绪状态，则脱离就绪分级链表
        slos_out_ready(tcbdelete);                             //脱离就
绪分级链表
    }

    #if GRAMM_STATA_EN > 0
    gramm_count--;                                             //减少系
统的程序计数
    #endif

    tcbdelete->State   = OS_STATE_FREE;
    tcbdelete->NextTcb = SLOS_tcb_free_list;                   //插入到
空闲tcb链表
    SLOS_tcb_free_list = tcbdelete;

    OS_EXIT_CRITICAL();

    SLOS_gramm_sched();       //进行程序调度

    return SLOS_ERR_NO_ERR;
}
#endif
```

5.2.4 判定最高优先级就绪程序的方法

μC/ OSⅡ给每一个进程一个它所需要的号码，我们称之为优先级，调度程

序负责进行筛选，它能够实现选择优先级最高的就绪程序来进行运行。

kernal运行中经常高频率的地进行程序的切换，ucos程序调度属于系统的临界资源，调度时需要独占CPU，不允许外部中断和程序切换。所以调度速度缓慢时，会影响整个系统的响应和系统的处理能力。

所以说，系统的核心问题是怎样快速查找最高优先级就绪程序。该算法在扩展了系统可管理程序数目的同时，考虑了如何在程序数目增大时，还能够快速进行最高优先级就绪程序判定的问题，并给出了一种快速的判定方法。

判定算法发出结果之前，先描述μC/ OSⅡ中原有的优先级判定表格，该表格为16 ×16 的数组，每个数组中的数是固定不变的。

当使用该表格时，数组下标变换成了八位二进制数，其中高四位对应数组的行，低四位对应数组的列(假设最右边的一位是第0 位bit0)，由此就可查表判断一组程序就绪态程序中优先级最高的那个程序所在的位置(查表后返回的值在0到7 之间)，再经过计算，就可得到最高优先级就绪程序的优先级。

具体代码如下:

```
void SLOS_gramm_sched()
{
   #if OS_CRITICAL_METHOD == 3                          //声明关中断
时保存cpu状态寄存器的变量
    OS_CPU_SR  cpu_sr;
   #endif

   OS_ENTER_CRITICAL();

   SLOS_get_heighready();                               //找到当前最
高优先级的程序

   if(OSTCBCur != SLOS_tcb_by_prio[SLOS_Prio_highready]){  //只有当
最高优先级程序不是当前程序的情况下才进行程序切换

       #if CPU_HOOKS_EN > 0
       OSGrammSwHook();
       #endif

       OSTCBHighReady = SLOS_tcb_by_prio[SLOS_Prio_highready];

       #if GRAMM_STATA_EN > 0
```

```
        gramm_switch_count++;                          //程序交换的
计数值
        #endif

        OS_GRAMM_SW();                                 //进行程序调
度
    }

    OS_EXIT_CRITICAL();
}
```

例如,如果OSReadyGrp 的值为0X0068，则利用上面的代码,首先可以判断*oy* ! = 0，则查优先级判定表格得到OSUnMap Tbl[*oy*]的值为3，假设OSReadyTbl[3]的值为0X00E4，则可判断*ox* ! = 0，再查优先级判定表格得到OSUnMap Tbl[*ox*]的值为2，最后计算得到程序的优先级prio 为50。

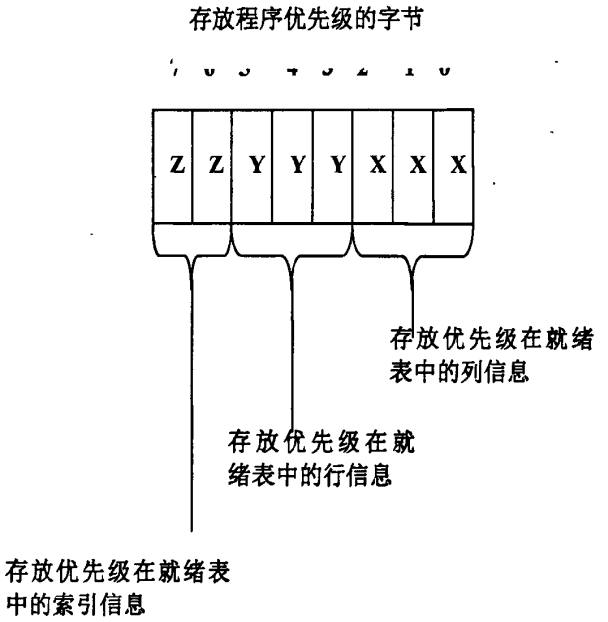
5.3增加可管理程序的第二种算法改进方法

第一版本的μC/ OS Ⅱ中,优先级调度算法使用了6 位二进制数来记录程序的优先级号码，还有两位没有用就闲置下来了，为将来的扩容留出空间。

第一种方法实现中，增加定位就绪程序优先级在就绪表中位置的行和列信息的二进制位数可以有效地增加系统的优先级别的使用。这样只需要一个表进行查询就可以了。

第二种方法是利用原来存放优先级的字节中剩余的两位作为索引，两位二进制数可以表示四种状态，利用者四种状态代表四个就绪表，每个就绪表依然是64位，这样就使程序优先级扩展到256个。

这里需要增加一个变量OSReadyXY，用于存放索引信息，另外还要使用变量OSReadyGrp [ ]存放程序优先级的行信息， OSReadyTbl0[ ]，OSReadyTbl1[ ]，OSReadyTbl2[ ]和OSReadyTbl3[ ]4 个8 位数组用于存放每个优先级的程序是否就绪的信息。这种方法的程序优先级字节的定义如下图所示。



5.4 优先级定位字

5.3.1. 将程序放入就绪表

这种方法的增加优先级是用一个字节的最高两位存放索引信息(就像进行表的选择序列)，因为只有两位二进制数，它所能表示的状态是二的二次方，也就是四种状态。

这将就绪表分为4 个部分(也就是四个表进行索引)，将程序的优先级别放入就绪表，第一要进行索引，可以通过前两位二进制数确定程序优先级在就绪表中的哪个部分,然后再通过行和列信息确定程序优先级的具体位置。图中的数字0～255 仅为清楚起见表示索引信息或程序优先级，并非实际存放的状态信息。

将就绪程序放入就绪表的具体代码可用如下方法实现:

```
OSReadyXY = prio > > 6 ;
OSReadyGrp [OSReadyXY] | = OSMap Tbl[prio > > 3 ] ;
if (OSReadyXY = = 0X00)
  OSReadyTbl0[prio > > 3 ] | = OSMap Tbl[prio & 0X07 ] ;
if (OSReadyXY = = 0X01)
  OSReadyTbl1[ (prio > > 3) & 0X07 ] | = OSMap Tbl[prio & 0X07 ] ;
if (OSReadyXY = = 0X02)
```

```
  OSReadyTbl2[ (prio > > 3) & 0X07 ] | = OSMap Tbl[prio & 0X07 ] ;
if (OSReadyXY = 0X03)
  OSReadyTbl3[ (prio > > 3) & 0X07 ] | = OSMap Tbl[prio & 0X07 ] ;
```

其中，char OSMap Tbl[ ] = {0X01 , 0X02 , 0X04 , 0X08 , 0X10 , 0X20 , 0X40 , 0X80} , prio 表示程序的优先级。

5.3.2. 从就绪表中删除程序

当程序脱离就绪态时，可以用下面的代码对存放该程序优先级的变量中的相应位清零:

```
OSReadyXY = prio > > 6 ;
if (OSReadyXY = = 0X00)
  if ( (OSReadyTbl0[prio > > 3 ] & = ~OSMap Tbl[prio & 0X07 ]) = = 0)
    OSReadyGrp [OSReadyXY] & = ~OSMap Tbl[prio > > 3 ] ;
if (OSReadyXY = = 0X01)
  if ( (OSReadyTbl1[ (prio > > 3) & 0X07 ] & = ~OSMap Tbl[prio & 0X07 ])
= = 0)
    OSReadyGrp [OSReadyXY] & = ~OSMap Tbl[prio > > 3 ] ;
if (OSReadyXY = = 0X02)
  if ( (OSReadyTbl2[ (prio > > 3) & 0X07 ] & = ~OSMap Tbl[prio & 0X07 ])
= = 0)
    OSReadyGrp [OSReadyXY] & = ~OSMap Tbl[prio > > 3 ] ;
if (OSReadyXY = = 0X03)
  if ( (OSReadyTbl3[ (prio > > 3) & 0X07 ] & = ~OSMap Tbl[prio & 0X07 ])
= = 0)
    OSReadyGrp [OSReadyXY] & = ~OSMap Tbl[prio > > 3 ] ;
```

5.3.3. 判定最高优先级就绪程序的方法

使用以上方法建立就绪表之后，我们仍然可以利用原有的优先级判定表格 OSUnMap Tbl ([256])，对最高优先级就绪程序进行查找，并且不需要象第一种方法那样增加变量。具体的查找算法如下:

```
if ( (OSReadyGrp [0 ] & 0XFF) ! = 0) {
  y = OSUnMap Tbl[OSReadyGrp [0 ] ] ;
  x = OSUnMap Tbl[OSReadyTbl0[ y ] ] ;
  prio = ( y < < 3) + x ;}
else if ( (OSReadyGrp [1 ] & 0XFF) ! = 0) {
    y = OSUnMap Tbl[OSReadyGrp [1 ] ] ;
    x = OSUnMap Tbl[OSReadyTbl1[ y ] ] ;
    prio = ( y < < 3) + x + 64 ;}
  else if ( (OSReadyGrp [2 ] & 0XFF) ! = 0) {
      y = OSUnMap Tbl[OSReadyGrp [2 ] ] ;
      x = OSUnMap Tbl[OSReadyTbl2[ y ] ] ;
      prio = ( y < < 3) + x + 128 ;}
      else{
        y = OSUnMap Tbl[OSReadyGrp [3 ] ] ;
        x = OSUnMap Tbl[OSReadyTbl3[ y ] ] ;
            prio = ( y < < 3) + x + 192 ;}
```

5.4两种方法的比较

整篇论文介绍了扩展μC/ OSⅡ内核中的优先级比较的两种方法，这两种方法都可以将系统对优先级的管理进行扩展，都可以实现防止优先级逆转，防止死锁，都可以成功实现了64级扩展到256级，并且实时性的效率没有降低。下面从几个方面讨论两种改进的调度算法的优劣。

代码数量上第一种方法的代码更少，这也说明第一种方法的代码平均执行时间少于第二种方法的代码执行时间。

程序就绪到就绪表被写入的所用时间来看，方法一直接确定位置，将就绪程序放入就绪表，方法二，必须顺序查找，然后才能确定就绪程序在就绪表中的位置，第一种方法所用时间明显少于第二种方法；考虑到更为一般的情况，有N个程序，当前运行的程序为k，下一个运行的程序为j，在共用程序堆栈时必须做的工作有：

*为每个程序定义栈顶和栈底2个堆栈指针；

*在程序切换时，将待运行程序j的堆栈内容移动到*栈顶位置，同时将其堆栈上

方的程序堆栈下移，修改被移动推栈的程序堆栈指针。

假设我们定义的程序栈空间和程序的栈指针变量为：

```
void GrammSTK[MAX_STK_LEN];/*程序堆栈空间*/
typedef struct GrammSTKPoint{
int GrammID;
int pTopSTK;
int pBottomSTK;
}GRAMM_STK_POINT;
GRAMM_STK_POINT pGrammSTK[MAX_GRAMM_NUM]; /*存放每个程序的栈顶和栈底指针*/
```

程序栈指针数组pGrammSTK的元素个数同程序个数。为了堆栈交换，需要另外一块临时存储空间，其大小可按单个程序栈最大长度定义，用于中转堆栈交换的内容。堆栈内容交换的伪C算法可写为：

```
StkEechange(int CurGrammID,int RunGrammID)
{ /*2个参数为当前运行程序号和下一运行程序号*/
void TempSTK[MAX_PER_STK_LEN]; /*注意该变量长度可小于GrammSTK*/
L=程序RunGrammTD的堆栈长度；
①将GrammSTK顶部的L字节移动到TempSTK中；
②将RunGrammID程序的堆栈内容移动到GrammSTK顶部；
③将RunGrammID堆栈上方（移动前位置）所有内容下移L个字节；
④修改RunGramm堆栈上方（移动前位置）所有程序栈顶和栈底指针（pGrammSTK变量）；
};
```

该算法的平均时间复杂度可计算如下：

O（T）=SL/2+SL/2+SL×N/2

式中，第一、二项为步骤①和步骤②时间，第三项为步骤③时间；SL表示每个任堆栈的最大长度（即MAX_PER_STK_LEN），N表示程序数。

取SL为64字节，程序数为16个，则数据项平均移动次数为576。假设每次移动指令时间为2μs，则一次程序栈移动时间长达约1ms。所以在使用该方法时，为了执行时间尽量短，编码时应仔细推敲。

从空间上说，共用程序栈比独立程序栈优越。假设独立程序栈方法中每个堆栈空间为K，程序数为N，则独立程序栈方式的堆栈总空间为N×K。在共用程序栈时，考虑各程序互补的情况，GrammSTK变量不需要定义为N×K长度，可能定义为二分之一或者更小就可以了。

另外，这种方法不需要在程序切换时修改CPU的SP指针。

从查找最高优先级就绪程序所需的时间来看，第一种方法通过变量ox 和oy 直接确定所有就绪程序中的哪一个程序优先级最高，而第二种方法必须从最高优先级开始顺序查找，直到找到第一个处于就绪状态的程序才结束查找，这种方法花费的时间显然要比第一种方法多。是否能够快速判定最高优先级就绪程序是整个调度算法的最关键问题，因此通过以上分析，可以看出第一种方法显然要大大优于第二种方法。

最后，从空间上说，共用程序栈比独立程序栈优越。假设独立程序栈方法中每个堆栈空间为K，程序数为N，则独立程序栈方式的堆栈总空间为N×K。在共用程序栈时，考虑各程序互补的情况，GrammSTK变量不需要定义为N×K长度，可能定义为二分之一或者更小就可以了。

另外，这种方法不需要在程序切换时修改CPU的SP指针。

μC/ OS2 II源码是公开的，我们就可以对其内核程序代码进行改写，特别是优先级调度算法用两种方法进行改进，让它本来处理优先级64个，改进后在实时性基本不变的情况下，成为256个。这样不但可处理多于64 个的复杂的工程项目开发，而且保证了复杂程序在运行状态下的运行效率不改变。

参考文献

[1]吴旭光,何军红.嵌入式操作系统原理与应用[M].化学工业出版社,2007.

[2]吴明晖.基于ARM的嵌入式系统开发与应用.人民邮电出版社，2004.

[3]Labrosse J J. 基于ARM的嵌入式系统开发与应用.北京：北京航空航天大学出版社，2003.

[4]Krishna C M, shin G K. Real-Time System. Columbus, OH: McGraw-Hill Companies. I C .1997.

[5]孔祥营等. 嵌入式实时操作系统VxWorks及其开发环境Tornado.中国电力出版社.

[6]JEAN J L. μC/OS- II—源代码公开的实时嵌入式操作系统.中国电力出版社.

[7]王 玲, 杨红雨, 张昭瑜. μC/ OS2 II中优先级调度算法的改进及实现. 四川大学学报（自然科学版）.

[8]马德新，徐鹏民.实时操作系统μC/OS- II 调度算法的研究.莱阳农学院

[9] Andrew N. Sloss, Dominic Symes, Chris Wright. ARM嵌入式系统开发 软件设计与优化.北京航空航天大学出版社 2005.

[10]周立功等.ARM嵌入式系统基础教程. 北京航空航天大学出版社, 2005.

[11]周立功等.ARM 嵌入式系统软件开发实例一.北京航空航天大学出版社, 2004.

[12]周立功等.ARM嵌入式系统软件开发实例二.北京航空航天大学出版社 2006

[13]杨恒主.ARM嵌入式系统设计及实践.西安电子科技大学出版社, 2005.

[14]孙秋野,孙凯,冯健.ARM嵌入式系统开发典型模块.人民邮电出版社, 2007.

[15]周立功等.ARM嵌入式系统实验教程.北京航空航天大学出版社, 2004.

[16]彭刚等. 基于ARM Cortex-M3的STM32系列嵌入式微控制器应用实践. 电子工业出版社，2011

[17]陆玲等. 嵌入式系统软件设计中的数据结构. 北京航空航天大学出版社, 2008

致 谢

这个题目的研究和学位论文的撰写是导师高胜法老师的亲切关怀和悉心指导下完成的。

老师的研究态度和严谨精神影响我对这个项目的学习态度，让我了解到一切的成果都来之不易，都是辛苦的汗水灌溉的。

课题的选择到项目的研究最后到论文的完成，高老师都始终耐心的细心的指导支持着我。

在此，我还要感谢山东大学给我提供了良好的学习机会和学习环境。正是因为在山东大学这个大家庭的帮助和支持下，我才能克服一个一个的困难和疑惑，直至本文的顺利完成。

攻读学位期间发表的学术论文目录

1. 郝强《ucosii优先级扩展的两种方法探讨》. 中国科技信息 2010.9.15. 独立作者

学位论文评阅及答辩情况表

	姓名	专业技术职务	是否博导（硕导）	所在单位	总体评价※
论文评阅人	刘方爱	教授	是	山东师范大学	B
	张华惠	教授	是	山东大学	B
	马军	教授	是	山东大学	B

		姓名	专业技术职务	是否博导（硕导）	所在单位
答辩委员会成员	主席	刘方爱	教授	是	山东师范大学
	委员	张华惠	教授	是	山东大学
		马军	教授	是	山东大学
		贾智平	教授	是	山东大学

答辩委员会对论文的总体评价※	B	答辩秘书	[illegible]	答辩日期	2011.11.26
备注					

※优秀为“A”；良好为“B”；合格为“C”；不合格为“D”。