

摘 要

本次设计的是一个实时内核（取名叫 EMOS）。是一个可以基于优先级调度、可裁剪的、抢占式、实时多任务内核，具有高度可移植性，特别适合于微处理器和控制器。其主要包含以下几大功能特征。

任务管理。包括创建、删除、挂起、唤醒等任务管理功能。在调度策略上采用优先级调度算法；时间管理。EMOS 通过处理器的时钟确定一个系统时钟节拍，以此来标定系统当前的运行时间；内存管理。在 EMOS 中，并没有用 C 标准库的内存管理函数，而是采用每次分配或释放固定大小的内存块以保证确定的时间复杂度；高度可移植。EMOS 采用 ASIC C 编程，并将所有与 CPU 架构、编译器相关的代码集中到一个 C 文件中，以方便移植修改；功能可配置。整个内核的功能模块可通过配置剪除不需要的功能模块代码，使生成的目标代码占用空间更小，以适应嵌入式设备有限的存储空间。

关键词：操作系统，实时调度，时间管理，任务管理，内存管理，调度策略，可移植性，EMOS

1 概述

EMOS 是一个可以基于优先级调度、可裁剪的、抢占式、实时多任务内核，具有高度可移植性，特别适合于微处理器和控制器。

任务管理。EMOS 的任务管理功能包括创建任务、删除任务、挂起任务、唤醒任务。EMOS 实时内核的任务调度策略是基于优先级的调度，首先为每一个任务确定一个优先级序号，并按小号高优先级，大号低优先级的规则在就绪表中进行排列，任务调度要做的就是从就绪队列中找出优先级最高的任务，将当前正在运行的任务与之交换 CPU 使用权。

时间管理。EMOS 通过处理器的时钟确定一个系统时钟节拍，并在每个时钟节拍产生一个时钟中断，每来一个时钟节拍中断便会进入到相应的中断服务程序计数，以此来标定系统当前的运行时间，以及为每个任务计算任务超时时间值，每个任务都会有一个 timeout 计数器，每个时钟中断都会在服务程序中将当前正在运行的任务超时计数器减 1，如果 timeout 的值等于 0 时，则当前任务将被剥夺 CPU 使用权，并切换到就绪队列中优先级最高的任务。以此来解决某些高优先级一直占用 CPU 而造成的调度不公平问题。

内存管理。EMOS 的内存管理是按简单的固定块大小分配管理，对于一个实时内核，为了保证其在运行时间上的可预测性，每个动作必须有确定的时间复杂度，因此在 EMOS 中，并没有用 C 标准库的内存管理函数，而是采用每次分配或释放固定大小的内存块以保证确定的时间复杂度。

临界资源共享。在 EMOS 中通过信号量来解决临界资源的共享问题，系统会在启动前创建一个信号量并将初值设 1，以后对于需要进行互斥访问的代码段，首先必须在入口处申请信号量，此时信号量值减 1，如果信号量的值小于 0，代表当前已经有任务正在访问当前资源，便阻塞当前任务，如果信号量等于 0，则当前任务享有访问优先权。

高度可移植。为了提供最好的移植性能，EMOS 最大程度上使用 ANSI C 语言进行开发，并将所有与 CPU 架构、编译器相关的代码集中到一个 C 文件中，移植到其它平台时只需要对一个 C 文件的代码做相应修改。

功能可配置。整个内核的功能模块可通过在内核配置文件中选择使能某些需要的功能进行重新编译，剪除不需要的功能模块代码，使生成的目标代码占用空间更小，以适应嵌入式设备有限的存储空间。

2 算法与数据结构

2.1 算法的总体思想（流程）

我们实现的实时内核在任务调度策略上主要采用静态优先级调度的算法来实现的；在查找就绪队列中最高优先级任务时采用的是位示图原理，在常数级的时间复杂度就能找到需要与当前任务进行切换的最高优先级的就绪任务；在时间管理上以系统节拍为基础，运用系统时钟节拍中断对任务运行时间进行度量；在内存管理上采用简单的固定大小块分配策略。

2.2 实时内核

2.2.1 功能

完成任务调试的核心算法实现，对任务管理，内存管理，时间管理做数据结构以及算法实现上的支撑，是整个实时系统中向上服务的 API 级代码。主要包括：

创建维护 TCB 链表；

任务控制块的初始化；

任务堆栈的创建与初始化；

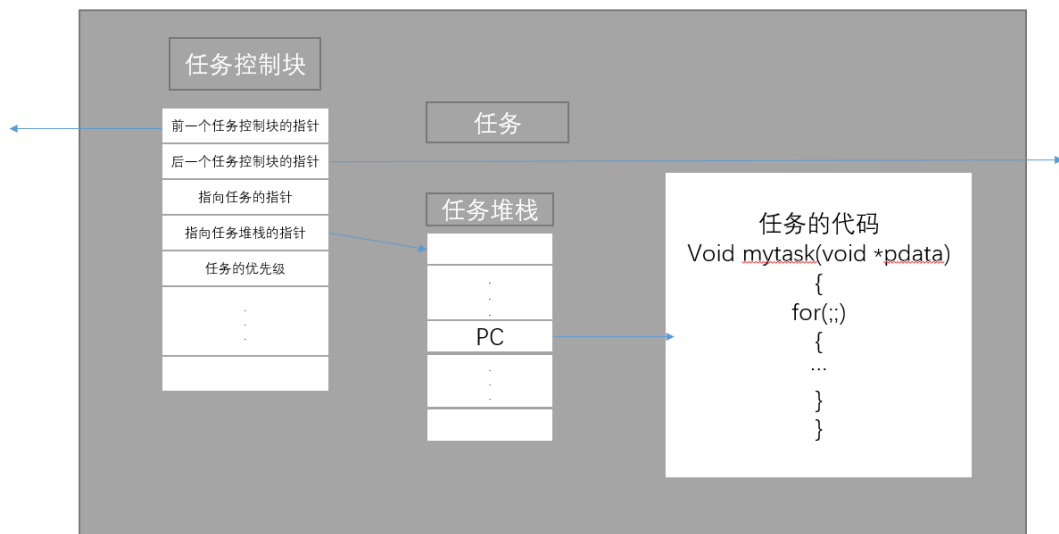
系统级任务（空闲任务与统计任务）的实现；

创建并维护就绪表；

高优先级就绪任务的查找；

2.2.2 数据结构

1) 任务控制块：



2) 标识任务是否就绪的位示图（一共可以表示 64 个任务）

任务就绪表
OSRdyGrp

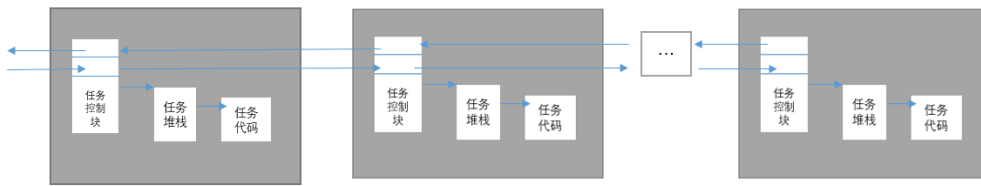
任务就绪表
OSRdyTb1[]

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0

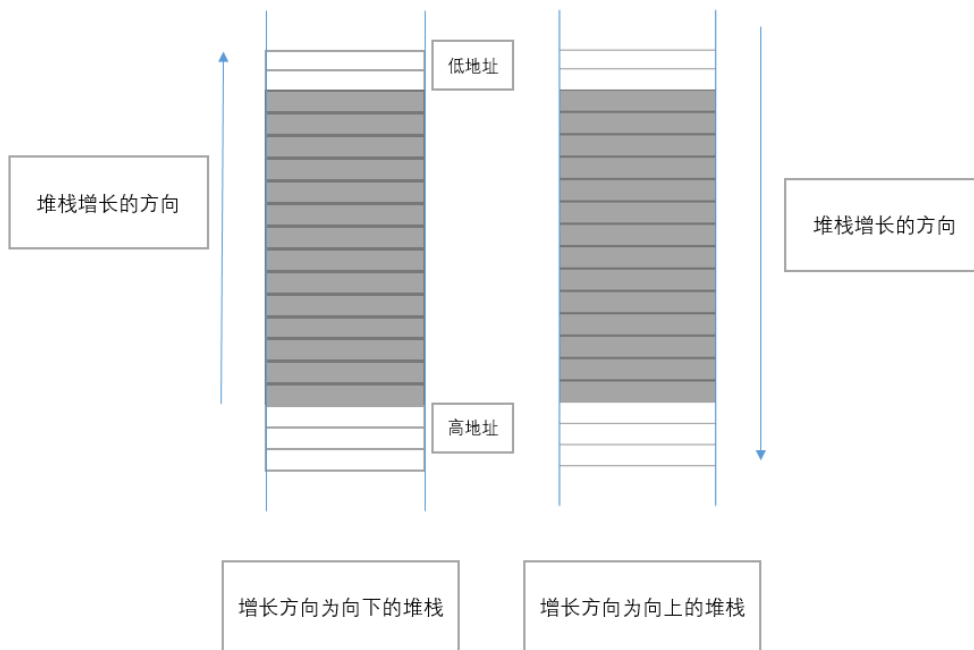
	D7	D6	D5	D4	D3	D2	D1	D0
OSRdyTb[0]	1/0	1/0	1/0	1/0	1/0	1/0	1/0	1/0
	7	6	5	4	3	2	1	0
	D7	D6	D5	D4	D3	D2	D1	D0
OSRdyTb[1]	1/0	1/0	1/0	1/0	1/0	1/0	1/0	1/0
	15	14	13	12	11	10	9	8
	D7	D6	D5	D4	D3	D2	D1	D0
OSRdyTb[2]	1/0	1/0	1/0	1/0	1/0	1/0	1/0	1/0
	23	22	21	20	19	18	17	16
	D7	D6	D5	D4	D3	D2	D1	D0
OSRdyTb[3]	1/0	1/0	1/0	1/0	1/0	1/0	1/0	1/0
	31	30	29	28	27	26	25	24

优先级别（任务的标识）

3) 任务控制块链表



4) 任务堆栈（不同 CPU 架构的栈在内存中的可能有不同的增长方向）



```
typedef struct {
    void    *pData;
    INT16U  Opt;
    void    (*Task)(void*);
    void *  Handle;
    INT32U  Id;
    INT32   Exit;
} OS_EMU_STK;
```

2.2.3 算法

任务就绪表结构：系统对于就绪表主要有三个操作：登记、注销和从就绪表的就绪任务中得知具有最高优先级任务的标识。

登记就是当某个任务处于就绪状态时，系统将该任务登记在任务就绪表

中，即在就绪表中将该任务的对应位置置 1。

在程序中，可用类似于下面的代码把优先级为 prio 的任务置为就绪状态：

```
OSRdyGrp |= ptcb->OSTCBBitY;  
OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
```

其中：ptcb->OSTCBBitY = (OS_PRI0) (1uL << ptcb->OSTCBY);

```
ptcb->OSTCBBitX = (OS_PRI0) (1uL << ptcb->OSTCBX);
```

```
ptcb->OSTCBY = (INT8U) (prio >> 3u);
```

```
ptcb->OSTCBX = (INT8U) (prio & 0x07u);
```

注销就是当某个任务需要脱离就绪状态时，系统在就绪表中将该任务的对应位置置 0。对应伪代码如下：

```
If((OSRdyTbl[pri>>3]&=-OSMapTbl[prio&0x07])==0)  
    OSRdyGrp&=-OSMapTbl[pri>>3];
```

最高优先级就绪任务的查找：系统调度器总是把 CPU 控制权交给优先级最高的就绪任务，因此调度器就必须具有从任务就绪表中查找最高优先级任务的能力。EMOS 调度器用于获取优先级最高的就绪任务的伪代码如下：

```
Y=OSUnMapTbl[OSRdyGrp];           //获得优先级位的 D5、D4、D3 位
```

```
X=OSUnMapTbl[OSRdyTbl[Y]];        //获得优先级位的 D2、D1、D0 位
```

```
Prio=(Y<<3)+Y;                     //获得就绪任务书的优先级
```

代码执行后，找的最高优先级就绪任务书的优先级别（即任务的标识）。其中 OSUnMapTbl[] 同样是 EMOS 为提高查找速度定义的一个数组，共有什么 256 个元素（16X16）：

```
INT8U  const  OSUnMapTbl[256] = {  
    0u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,  
    /* 0x00 to 0x0F */  
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,  
    /* 0x10 to 0x1F */  
    5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,  
    /* 0x20 to 0x2F */  
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,  
    /* 0x30 to 0x3F */  
    6u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,  
    /* 0x40 to 0x4F */  
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,  
    /* 0x50 to 0x5F */  
};
```

```

        5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
/* 0x60 to 0x6F */
        4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
/* 0x70 to 0x7F */
        7u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
/* 0x80 to 0x8F */
        4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
/* 0x90 to 0x9F */
        5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
/* 0xA0 to 0xAF */
        4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
/* 0xB0 to 0xBF */
        6u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
/* 0xC0 to 0xCF */
        4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
/* 0xD0 to 0xDF */
        5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,
/* 0xE0 to 0xEF */
        4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u
/* 0xF0 to 0xFF */
};

```

找到最高优先级任务与当前任务作替换后，再调用任务级调度器交换 CPU 使用权就可以完成任务切换了。

2.3 中断和时间管理

2.3.1 功能

中断：

在应用程序中经常有一些代码段必须不受任何干扰地连续运行，这样的代码段叫做临界段。因此，为了使临界段在运行时不受中断所打断，在临界段代码前必须用关中断指令使 CPU 屏蔽中断请求，而在临界段代码后必须用开中断指令解除屏蔽使得 CPU 可以响应中断请求。由于各厂商生产的 CPU 和 C 编译器的关中断和开中断的方法和指令不尽相同，为增强 EMOS 的可移植性（即在 EMOS 函数中尽可能地不出现汇编语言代码），EMOS 用两个宏来实现中断的开放和关闭，而把与系统的硬件相关的关中断和开中断的指令分别封装在这两个宏中：

OS_ENTER_CRITICAL()

OS_EXIT_CRITICAL()

在中断服务程序中调用的负责任务切换工作的函数 OSIntCtxSw() 叫做中断级任务切换函数。

时间管理:

1) 提供的延时函数

由于嵌入式系统的任务是一个无限循环, 并且 $\mu C/OS-II$ 还是一个抢占式内核, 所以为了使高优先级别的任务不至于独占 CPU, 可以给其他任务优先级别较低的任务获得 CPU 使用权的机会, $\mu C/OS-II$ 规定: 除了空闲任务之外的所有任务必须在任务中合适的位置调用系统提供的函数 OSTimeDly(), 使当前任务的运行延时 (暂停) 一段时间并进行一次任务调度, 以让出 CPU 的使用权。

其它时间管理功能有:

2) 取消任务延时函数

3) 设置系统时间函数

4) 获得系统时间函数

2.3.2 数据结构

此部分无自己的数据结构, 均是在实时内核的基础上实现的功能, 在调用中所用到的数据结构见第一部分的实时内核中。

2.3.3 算法

系统响应中断的过程为: 系统接收到中断请求后, 这时如果 CPU 处于中断允许状态 (即中断是开放的), 系统就会中止正在运行的当前任务, 而按照中断向量的指向转而去运行中断服务子程序; 当中断服务子程序的运行结束后, 系统将会根据情况返回到被中止的任务继续运行或者转向运行另一个具有更高优先级别的就绪任务。

服务子程序运行结束之后, 系统将会根据情况进行一次任务调度去运行优先级别最高的就绪任务, 而并不是一定要接续运行被中断的任务的。

系统时钟中断服务函数执行过程如下:

保存 CPU 寄存器;

调用 OSIntEnter(); //记录中断嵌套层数

if (OSIntNesting == 1;

{

OSTCBCur->OSTCBStkPtr = SP; //保存堆栈指针


```

}
调用 OSTimeTick( );           //节拍处理
清除中断;
开中断;
调用 OSIntExit( );           //中断嵌套层数减一
恢复 CPU 寄存器;
中断返回;

```

数 OSTimeTick() 的任务，就是在每个时钟节拍了解每个任务的延时状态，使其中已经到了延时时限的非挂起任务进入就绪状态。

2.4 内存管理

2.4.1 功能

应用程序在运行中为了某种特殊需要，经常需要临时获得一些内存空间，因此作为一个比较完善的操作系统必须具有动态分配内存的能力。能否合理、有效地对内存存储器进行分配和管理，是衡量一个操作系统品质的指标之一。特别地对于实时操作系统来说，还应该保证系统在动态分配内存时，它的执行时间必须是可确定的。

EMOS 改进了 ANSI C 用来动态分配和释放内存的 malloc() 和 free() 函数，使它们可以对大小固定的内存块进行操作，从而使 malloc() 和 free() 函数的执行时间成为可确定的，满足了实时操作系统的要求。EMOS 的内存管理目前主要功能是对内存进行简单管理，并向系统内核作底层支撑，当系统内核启动时，需要为任务分配加载内存。也可以用于应用程序的动态分配和释放内存。

2.4.2 数据结构

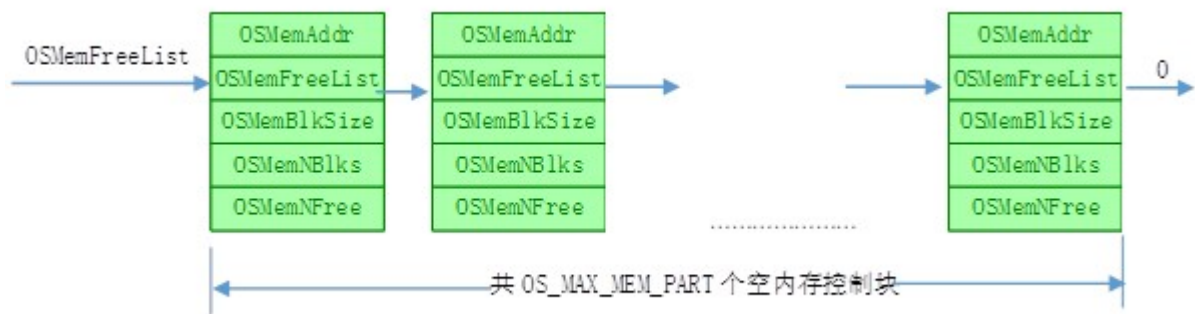
内存控制块的结构如下：

```

typedef struct {
    void    *OSMemAddr;        //内存分区的指针
    void    *OSMemFreeList;    //内存控制块链表的指针
    INT32U  OSMemBlkSize;      //内存块的长度
    INT32U  OSMemNBlks;        //分区内内存块的数目
    INT32U  OSMemNFree;        //分区内当前可分配的内存块的数目
} OS_MEM;

```

内存控制块链表：



2.4.3 算法

EMOS 对内存进行两级管理，即把一个大片连续的内存空间分成了若干个分区，每个分区又分成了若干个大小相等的内存块来进行管理(验收时只实现了用简单的分块思路，经老师提醒后，这一周又加上了分区管理，变成了两级管理)。操作系统以分区为单位来管理动态内存，而任务以内存块为单位来获得和释放动态内存。内存分区及内存块的使用情况则由表——内存控制块来记录。

应用程序如果要使用动态内存的话，则要首先在内存中划分出可以进行动态分配的区域，这个划分出来区域叫做内存分区，每个分区要包含若干个内存块。EMOS 要求同一个分区中的内存块的字节数必须相等，而且每个分区与该分区的内存块的数据类型必须相同。

为了使系统能够感知和有效地管理内存分区，EMOS 给每个内存分区定义了一个叫做内存控制块 (OS_MEM) 的数据结构。系统就用这个内存控制块来记录和跟踪每一个内存分区的状态。当应用程序调用函数 OSMemCreate() 建立了一个内存分区之后，内存控制块与内存分区和内存块之间的关系如图

在内存中划分一个内存分区与内存块的方法非常简单，只要定义一个二维数组就可以了，其中的每个一维数组就是一个内存块。例如，定义一个用来存储 INT16U 类型数据，有 10 个内存块，每个内存块长度为 10 的内存分区的代码如下：

```
INT16U IntMemBuf[10][10];
```

上面这个定义只是在内存中划分出了分区及内存块的区域，还不是一个真正的可以动态分配的内存区，只有当把内存控制块与分区关联起来之后，系统内核才能对其进行相应的管理和控制，它才能是一个真正的动态内存区。

2.5 任务管理

2.5.1 功能、

任务管理主要是对系统中的所有任务（包括已被删除但 TCB 仍在空链表中的任务）其主要功能如下：

创建任务；

删除任务；

挂起任务；

唤醒任务；

2.5.2 数据结构

此部分没有自己的数据结构，均是在实时内核的基础上实现的任务管理功能，在调用中所用到的数据结构见第一部分的实时内核中。

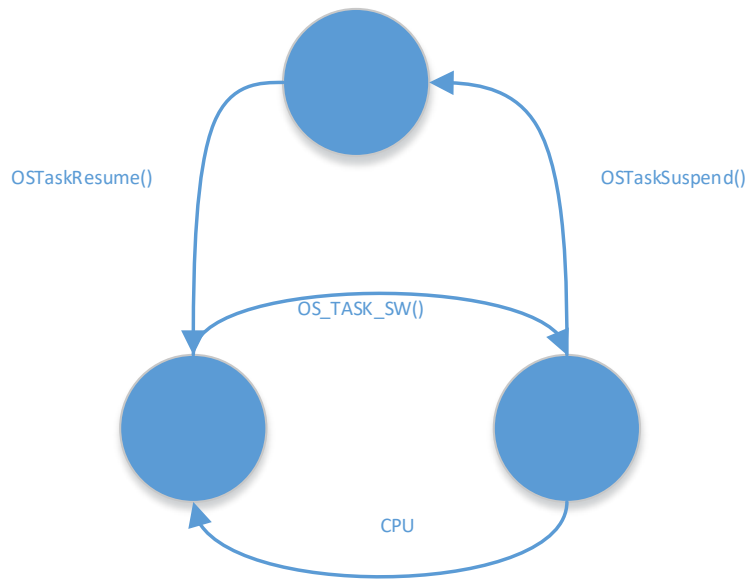
2.5.3 算法

应用程序在创建一个新任务的时候，必须把在系统启动这个任务时 CPU 各寄存器所需要的初始数据（任务指针、任务堆栈指针、程序状态字等等），事先存放在任务的堆栈中，任务堆栈的初始化就是对该任务的虚拟处理器的初始化（复位）。在应用程序中定义任务堆栈的栈区非常简单，即定义一个 OS_STK 类型的一个数组并在创建一个任务时把这个数组的地址赋给该任务就可以了。

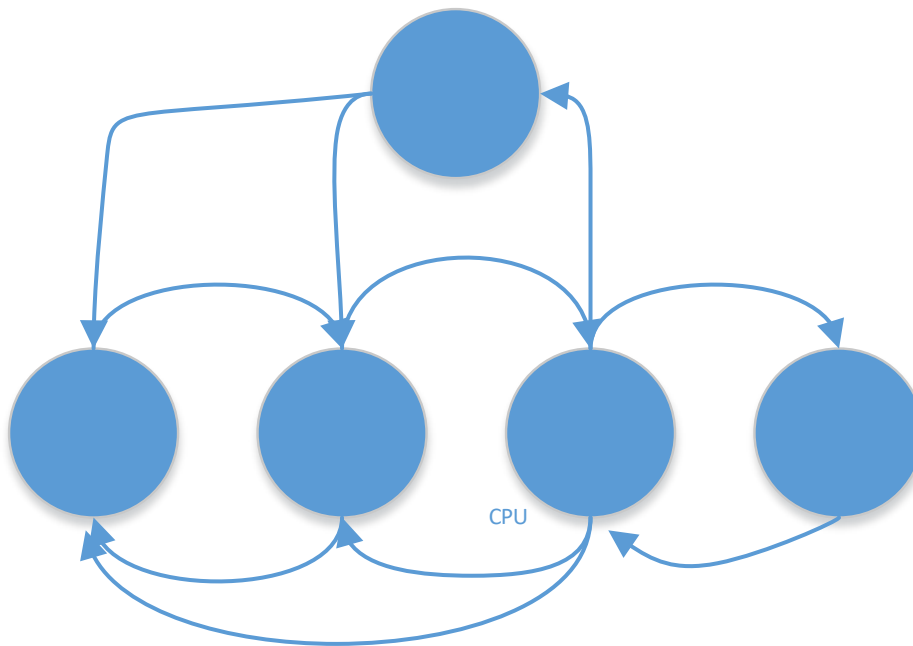
当进行系统初始化时，初始化函数会按用户提供的任务数为系统创建具有相应数量的任务控制块并把它们链接为一个链表。由于这些任务控制块还没有对应的任务，故这个链表叫做空任务块链表。即相当于是一些空白的身份证。

当应用程序调用函数 OSTaskCreate() 创建一个任务时，这个函数会调用系统函数 OSTCBInit() 来为任务控制块进行初始化。这个函数首先为被创建任务从空任务控制块链表获取一个任务控制块，然后用任务的属性对任务控制块各个成员进行赋值，最后再把这个任务控制块链入到任务控制块链表的头部。

以下是任务转换图：（包括各个任务状态发生切换时所触发的函数动作）



加上任务挂起与任务唤醒后的状态转换关系：



2.6 CPU 移植相关

2.6.1 功能

此部分主要是处理与处理器架构相关的实时内核代码，包括关于 X86 架构栈初始化，关中断，开中断的宏实现，以及利用 WINDOWS API 实现的线程间 CPU 切换功能。切换功能分为两部分，一种为任务级的切换，一种为中断级的切换。

2.6.2 数据结构

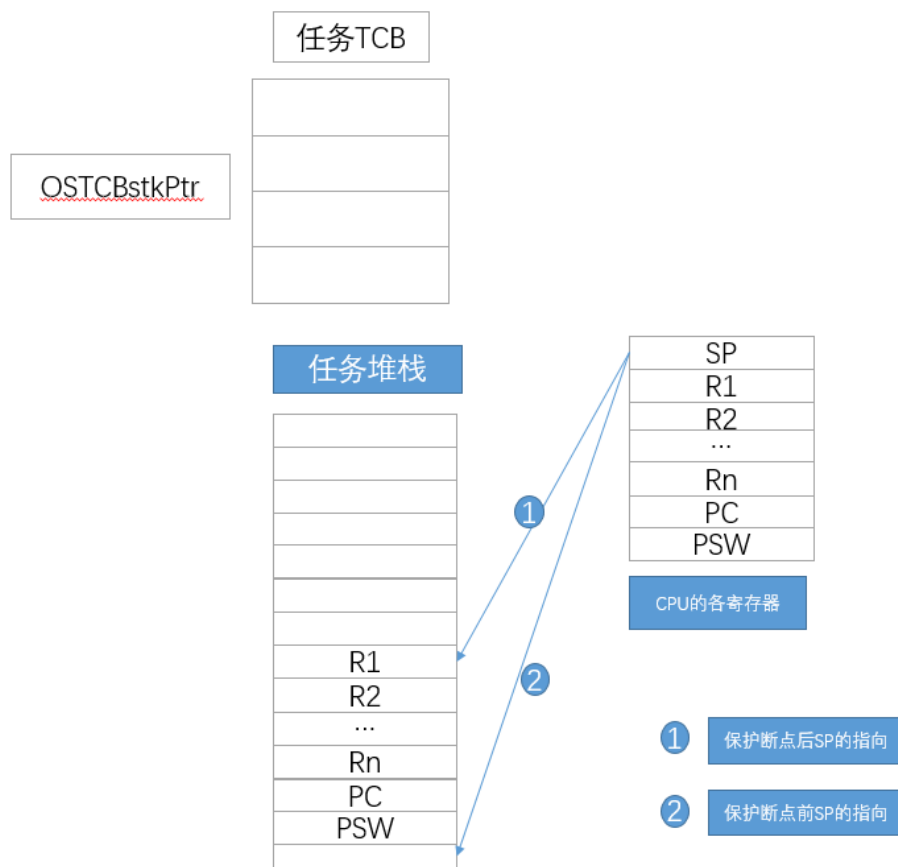
此部分也没有特别的数据结构，栈相关的数据结构均参照第一部分。

2.6.3 算法

由于系统存在着多个任务，于是系统如何来识别并管理一个任务就是一个需要解决的问题。识别一个任务的最直接的办法是为每一个任务起一个名称。

由于 EMOS 中的任务都有一个惟一的优先级别，因此 EMOS 是用任务的优先级来作为任务的标识的。所以，任务控制块还要来保存该任务的优先级别。

一个任务的任务控制块的主要作用就是保存该任务的虚拟处理器的堆栈指针寄存器 SP。具体如下图所示：



而任务的切换就是将当前的 CPU 中的 SP 指针保存到自己的 TCB 中，同时将通用寄存器值保存到 TCB 中相应的位置，这个过程就叫任务的现场保护，接下来再通过第一部分讲到的寻找就绪表中最高优先级任务的算法得到要切换所任务 TCB, 将 TCB 中的 SP 值搬运到 CPU 中，同时 TCB 中保存的通用寄存器的值也复制到 CPU 中，这个过程就叫现场恢复，整个任务的切换过程，就是完成了任务的上下文与 CPU 寄存器值的相互切换工作。

3 程序设计与实现

3.1 程序流程图

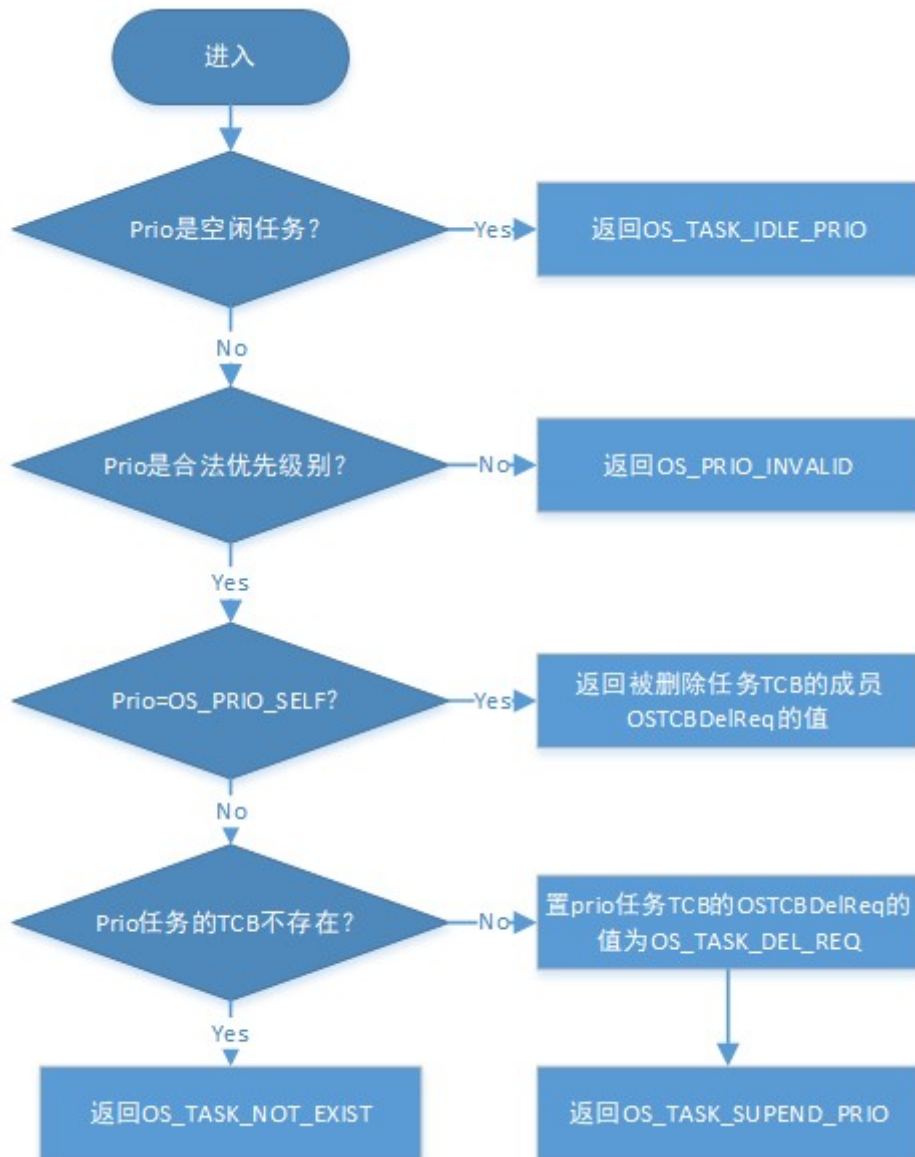
1) 实时内核启动流程:



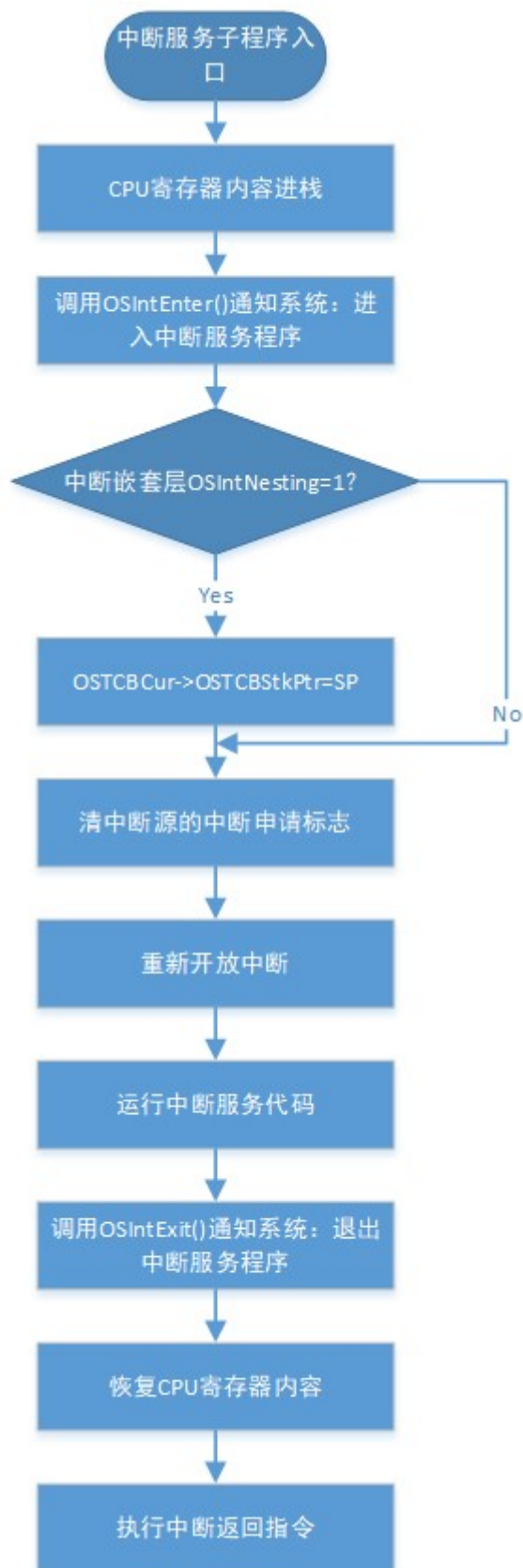
2) 创建新任务流程:



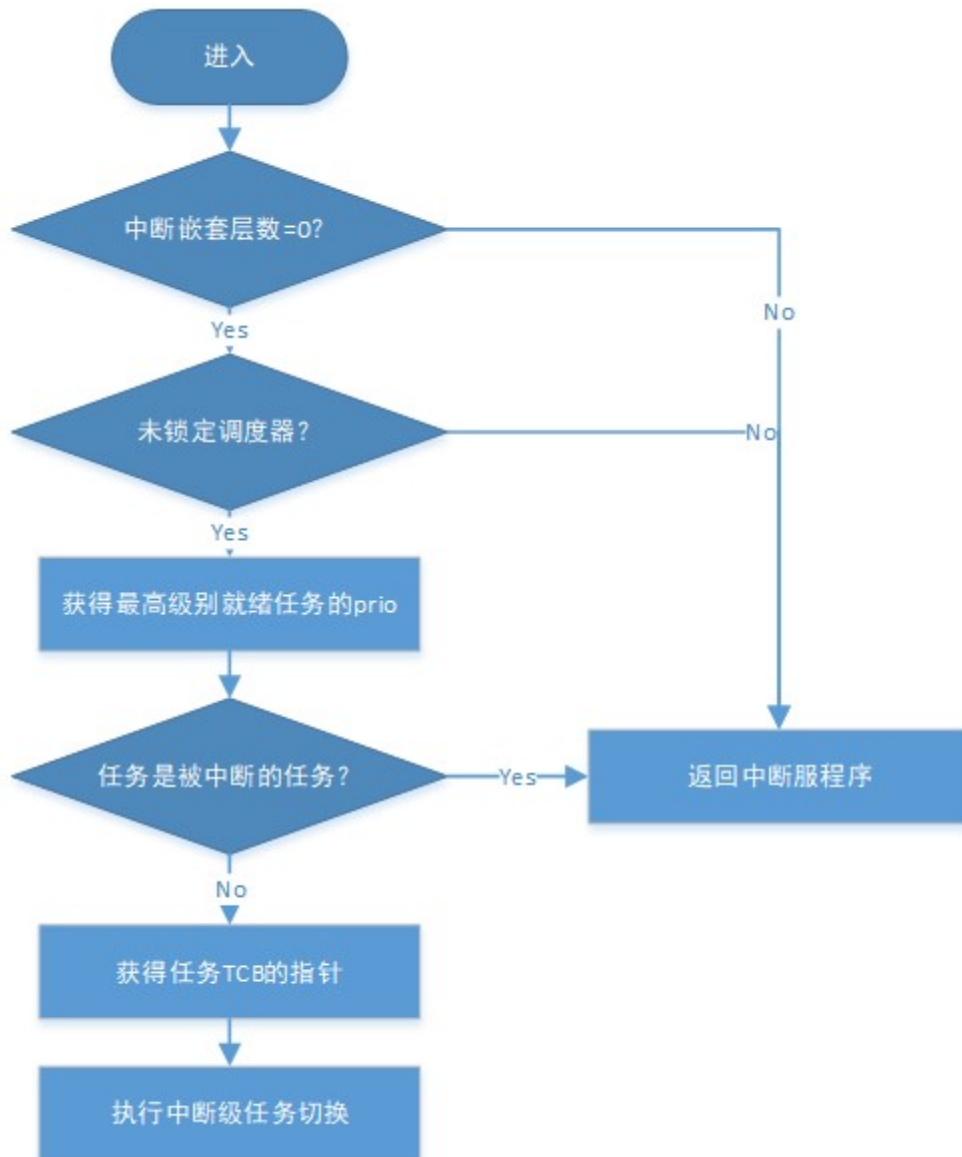
3) 请求删除任务流程:



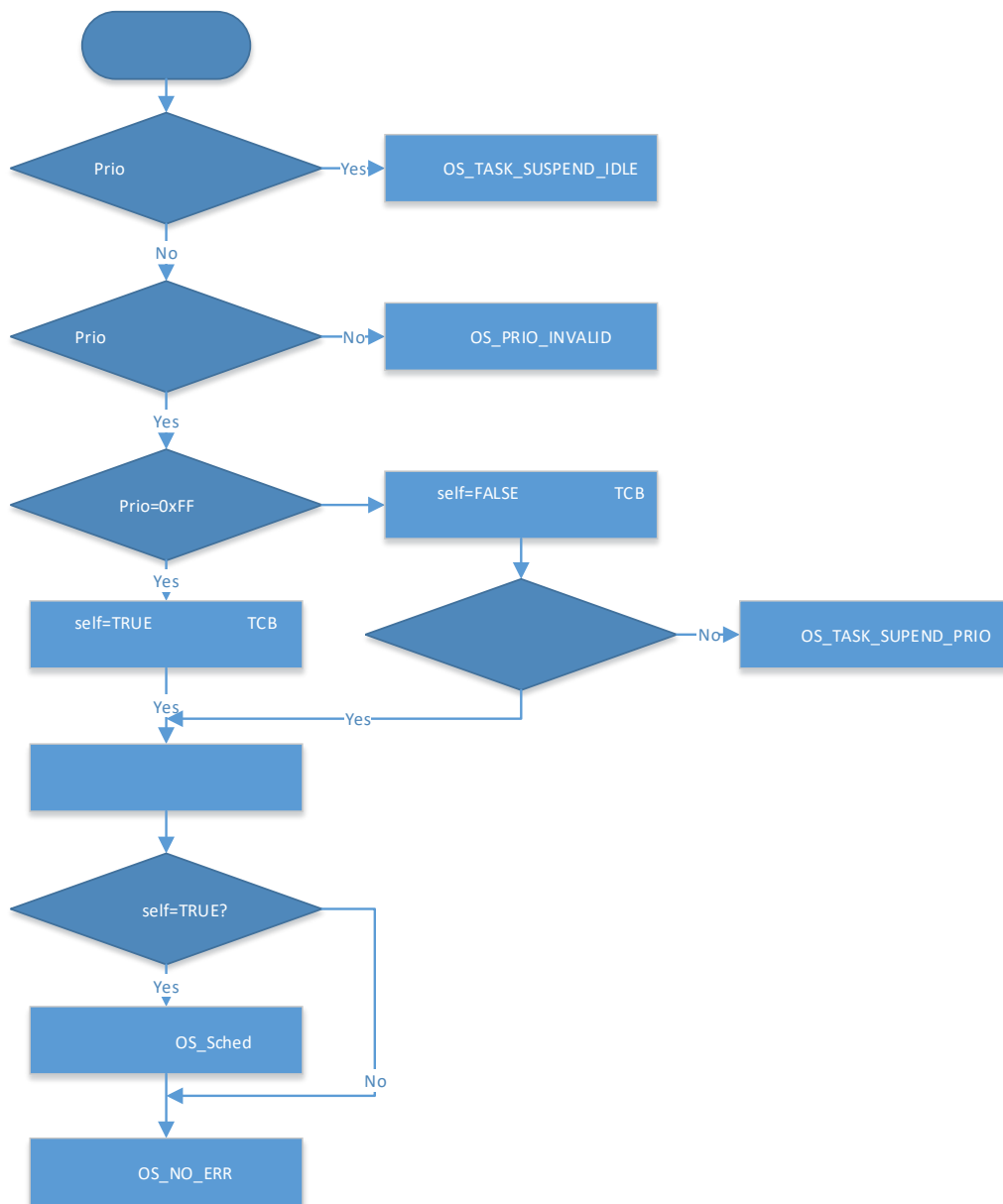
4) 中断服务子程序流程图



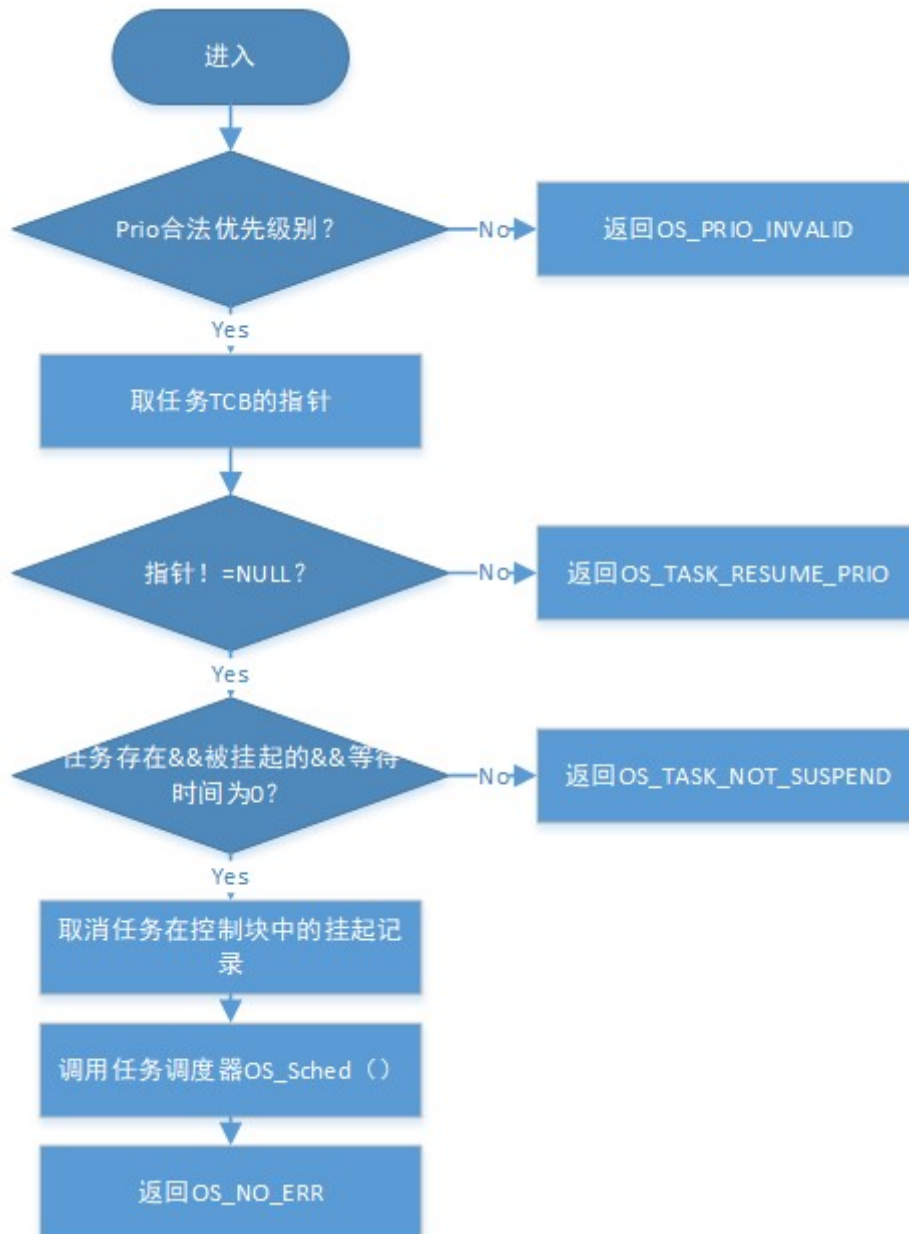
5) 退出中断服务函数 OSIntExit(). 函数流程图如下:



6) 任务挂起流程图:



7) 任务唤醒流程图:



3.2 程序说明

我们这个简单的实时操作系统内核起名为：“EMOS”

开发平台: Visual Studio 2017, Windows 10

程序的结构很清晰，一共分为七个模块：

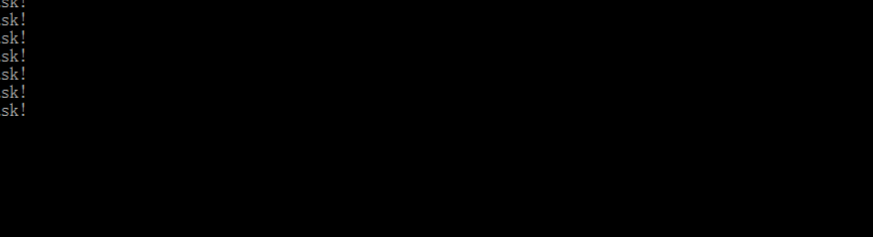
模块名	文件名	内容描述
核心代码	os_core.c	实时内核核心数据结构与算法的实现
	emos.h	包含所有内核代码需要的数据结构定义和宏定义以及函数声明
任务管理	os_task.c	创建、删除、挂起、唤醒任务等功能函数
时间管理	os_time.c	时间管理功能函数
内存管理	os_mem.c	内存管理功能函数
移植文件	os_cpu.c.c	平台移植相关代码
	os_cpu.h	平台移植相关头文件
内核配置	os_cfg_r.h	EMOS 内核功能配置头文件
主函数	main.c	内核测试相关代码
	main.h	内核测试相关头文件

3.3 实验结果

在测试程序过程中：

我们首先创建了一个主任务，优先级赋为 5，堆栈大小为 512bytes:

然后启动操作系统，调度主任务开始运行，运行周期设为 1s:



The screenshot shows a Windows command prompt window with the title bar 'C:\Windows\system32\cmd.exe'. The window contains a series of seven lines of output, each starting with 'I am MainTask!'. The text is displayed in a monospaced font on a black background. At the bottom of the window, there is a small white rectangular area containing the text '万能五笔输入法 半：'.

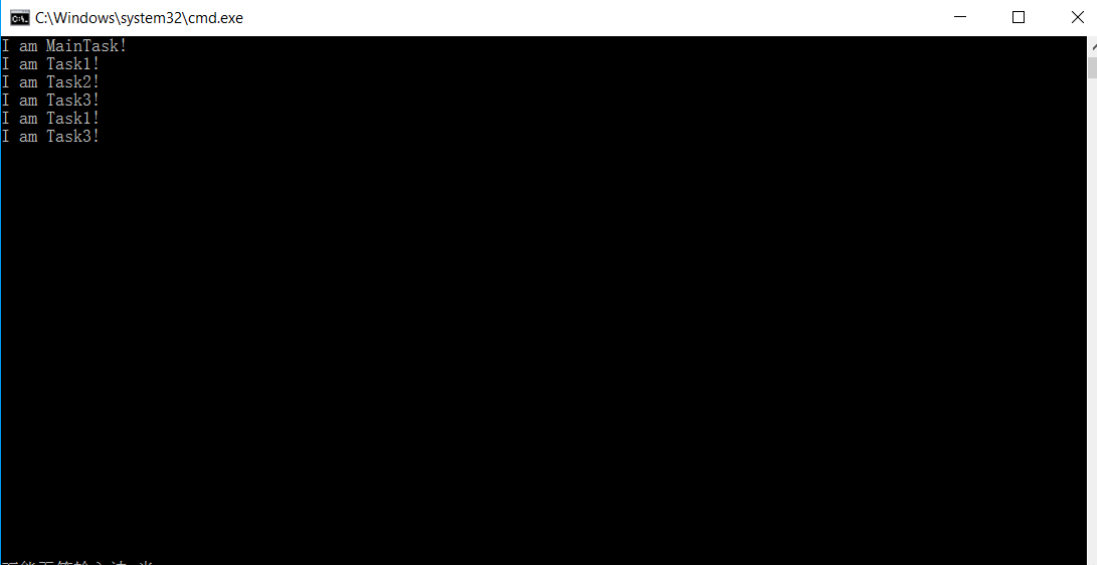
然后在主任务中创建了三个用户任务，优先级分别为 7、9、10

栈空间均为 512 字节

它们的运行周期分别为 1s、2s、1s

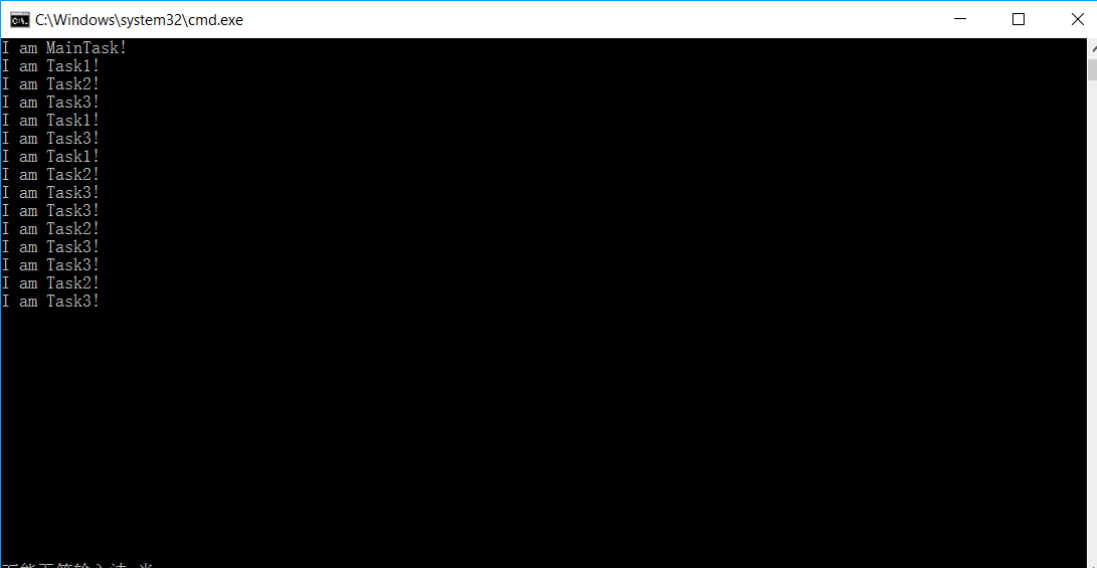
创建完用户任务后，主任务使命完成，所以删除自身。

每个任务中执行的内容分别是打印出自己的任务名称。



```
C:\Windows\system32\cmd.exe
I am MainTask!
I am Task1!
I am Task2!
I am Task3!
I am Task1!
I am Task3!
```

当任务 1 运行完第一个周期后将自身挂起：



```
C:\Windows\system32\cmd.exe
I am MainTask!
I am Task1!
I am Task2!
I am Task3!
I am Task1!
I am Task3!
I am Task1!
I am Task3!
I am Task2!
I am Task3!
I am Task3!
I am Task2!
I am Task3!
I am Task3!
I am Task3!
I am Task2!
I am Task3!
```

在 10s 后，即任务 2 运行了解到第 5 个周期时，此时再唤醒任务 1：

```
C:\Windows\system32\cmd.exe
I am MainTask!
I am Task1!
I am Task2!
I am Task3!
I am Task1!
I am Task3!
I am Task1!
I am Task2!
I am Task3!
I am Task3!
I am Task2!
I am Task3!
I am Task2!
I am Task3!
I am Task3!
I am Task2!
I am Task3!
I am Task3!
I am Task2!
I am Task3!
I am Task3!
I am Task1!
I am Task2!
I am Task3!
I am Task1!
I am Task3!
```

以上测试：包括了任务管理中的四部分功能，创建任务，删除任务，挂起任务，唤醒任务。

在任务管理的测试中也包括了对内存分配与释放的测试（创建任务会申请分配内存，删除任务会释放内存）也包括了对时间管理的测试（每个任务都定义了 deadline 和时钟周期，在运行时严格按周期计算，准确无误）。

动态任务管理接口：

在老师的建议下增加了用户在系统运行过程中动态增删任务的需求：

当系统运行时通过输入 n 个 ‘r’ 代表要让现有任务运行 n 个周期：

```
C:\Windows\system32\cmd.exe
-----EMOS kernel-----
输入a添加任务
输入d删除任务
输入n个 r 代表要当前运行n个周期
-----EMOS kernel-----
I am Task1!
I am Task2!
I am Task3!
rr
I am Task1!
I am Task2!
I am Task3!
I am Task1!
I am Task3!
I am Task1!
I am Task3!
I am Task1!
I am Task2!
I am Task3!
```

当输入 ‘a’ 时代表用户增加一个任务：

```

C:\Windows\system32\cmd.exe
-----EMOS kernel-----
输入a添加任务
输入d删除任务
输入n个'r'代表要当前运行n个周期

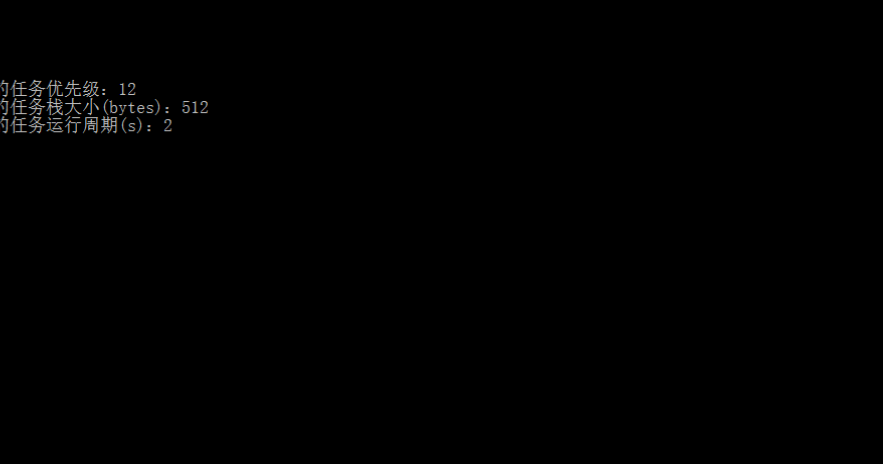
-----EMOS kernel-----
I am Task1!
I am Task2!
I am Task3!
rr
I am Task1!
I am Task2!
I am Task3!
I am Task1!
I am Task3!
I am Task1!
I am Task2!
I am Task3!
a
输入要添加的任务优先级: 11
输入要添加的任务栈大小(bytes): 512
输入要添加的任务运行周期(s): 1

1:折 2:白天 3:白色 4:拍下 5:白 6:白痴
1/6

```

A screenshot of a Windows Command Prompt window titled "C:\Windows\system32\cmd.exe". The window has standard Windows window controls (minimize, maximize, close) at the top right. The command history shows several commands related to task scheduling:

```
I am Task2!  
I am Task3!  
a  
输入要添加的任务优先级: 11  
输入要添加的任务栈大小(bytes): 512  
输入要添加的任务运行周期(s): 1  
I am Task1!  
I am Task2!  
I am Task3!  
I am Task4!  
I am Task1!  
I am Task3!  
I am Task4!  
rr  
I am Task1!  
I am Task2!  
I am Task3!  
I am Task4!  
I am Task1!  
I am Task3!  
I am Task4!  
I am Task1!  
I am Task2!  
I am Task3!  
I am Task4!
```

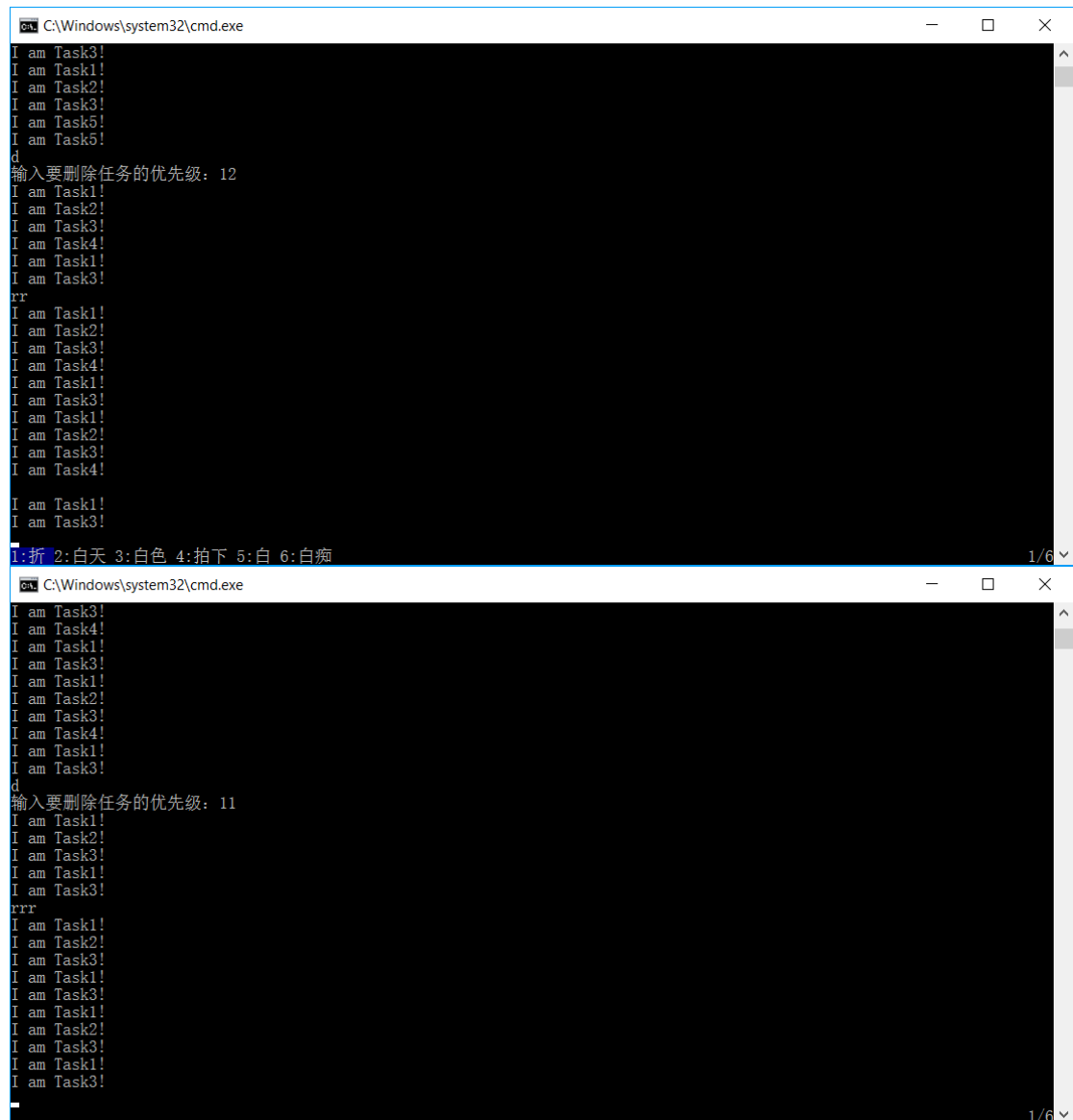

At the bottom of the window, there is a row of numbers: "1:的 2:白 3:所以 4:看看 5:看到 6:反正". In the bottom right corner, outside the command prompt window, the page number "1/6" is visible.

```

C:\Windows\system32\cmd.exe
I am Task1!
I am Task2!
I am Task3!
I am Task4!
a
输入要添加的任务优先级: 12
输入要添加的任务栈大小(bytes): 512
输入要添加的任务运行周期(s): 2
I am Task1!
I am Task2!
I am Task3!
I am Task5!
I am Task5!
I am Task1!
I am Task3!
rr
I am Task1!
I am Task2!
I am Task3!
I am Task5!
I am Task5!
I am Task1!
I am Task3!
I am Task1!
I am Task2!
I am Task3!
I am Task5!
I am Task5!
1:折 2:白天 3:白色 4:拍下 5:白 6:白痴

```

当输入 ‘d’ 时删除一个任务:



```
C:\Windows\system32\cmd.exe
I am Task3!
I am Task1!
I am Task2!
I am Task3!
I am Task5!
I am Task5!
d
输入要删除任务的优先级: 12
I am Task1!
I am Task2!
I am Task3!
I am Task4!
I am Task1!
I am Task3!
rr
I am Task1!
I am Task2!
I am Task3!
I am Task4!
I am Task1!
I am Task3!
I am Task1!
I am Task2!
I am Task3!
I am Task4!
I am Task1!
I am Task3!
1:折 2:白天 3:白色 4:拍下 5:白 6:白痴 1/6

C:\Windows\system32\cmd.exe
I am Task3!
I am Task4!
I am Task1!
I am Task3!
I am Task1!
I am Task2!
I am Task3!
I am Task4!
I am Task1!
I am Task3!
d
输入要删除任务的优先级: 11
I am Task1!
I am Task2!
I am Task3!
I am Task1!
I am Task3!
rrr
I am Task1!
I am Task2!
I am Task3!
I am Task1!
I am Task3!
I am Task1!
I am Task2!
I am Task3!
I am Task1!
I am Task3!
```

4 结论

我们采用了 C 语言编写完成了一个简单的实时操作系统内核，通过测试，这个简单的操作系统可以用于实时系统的任务管理，内存分配，以及时间管理。在移植性上也给实时内核的用户提供了最大便利。

当然，这个简单的实时内核也存在着许多需要完善的地方，比如缺少任务间的通信模块，没有对常见的 CPU 架构进行移植示例等。我还想把这个小内核继续完善，并移植到 ARM 架构上，满足自己在 ARM 上的应用程序提供实时任务调度的需求。

5 参考文献

- 1、周苏等编著.《操作系统原理实验》.北京:科学出版社.2003.
- 2、任哲等编著.《嵌入式实时操作系统 μ COS-II 原理及应用》.北京:北京航空航天大学出版社,2009.
- 3、徐虹等编著.《操作系统实验指导——基于 *Linux* 内核》.北京:清华大学出版社.2004.
- 4、陈向群等编著.《*Windows* 内核实验教程》.北京:机械工业出版社.2002