



The Real-Time Kernel

μ C/OS-II



μ C/OS-II

- Getting Started with μ C/OS-II
- Task Management
- Timer Management
- Inter-task Communication
- Memory Management



What is µC/OS-II?

- µC/OS-II is a portable, ROMable, scalable, preemptive, real-time deterministic multitasking kernel for microprocessors, microcontrollers and DSPs.

What does it do?

- Offering unprecedented ease-of-use, µC/OS-II is delivered with complete 100% ANSI C source code and in-depth documentation. µC/OS-II runs on the largest number of processor architectures, with ports available for download from the Micrium Web site.
- µC/OS-II manages up to 250 application tasks. µC/OS-II includes: semaphores; event flags; mutual-exclusion semaphores that eliminate unbounded priority inversions; message mailboxes and queues; task, time and timer management; and fixed sized memory block management.
- µC/OS-II's footprint can be scaled (between 5 Kbytes to 24 Kbytes) to only contain the features required for a specific application. The execution time for most services provided by µC/OS-II is both constant and deterministic; execution times do not depend on the number of tasks running in the application.

Reliability for safety-critical markets

- Software certification is vital in order to demonstrate the reliability and safety of software systems. µC/OS-II is currently implemented in a wide array of high level of safety-critical devices, including:
 - Those certified for Avionics DO-178B
 - Medical FDA pre-market notification (510(k)) and pre-market approval (PMA) devices
 - SIL3/SIL4 IEC for transportation and nuclear systems, 99% compliant with the Motor Industry Software Reliability Association (MISRA-C:1998) C Coding Standards



Applications

- µC/OS-II is used in a wide variety of industries:
 - Avionics — used in the Mars Curiosity Rover!
 - Medical Equipment/Devices
 - Data Communications Equipment
 - White Goods (Appliances)
 - Mobile Phones, PDAs, MIDs
 - Industrial Controls
 - Consumer Electronics
 - Automotive
 - A wide range of other safety critical embedded applications

Others

Communication Software Stacks



TCP/IP

A compact, high-performance TCP/IP stack featuring IPv4 & IPv6.



USB Device

USB device stack designed for embedded systems.



USB Host

Real-time USB host software stack designed for embedded systems.



CAN Bus

A communications stack for industrial applications.



Modbus

A communications stack for industrial applications.



Bluetooth

ClarinoxBlue is a protocol stack for embedded Bluetooth applications.

Storage & Display Software



File System

Compact, reliable, high-performance file system.



Graphical UI

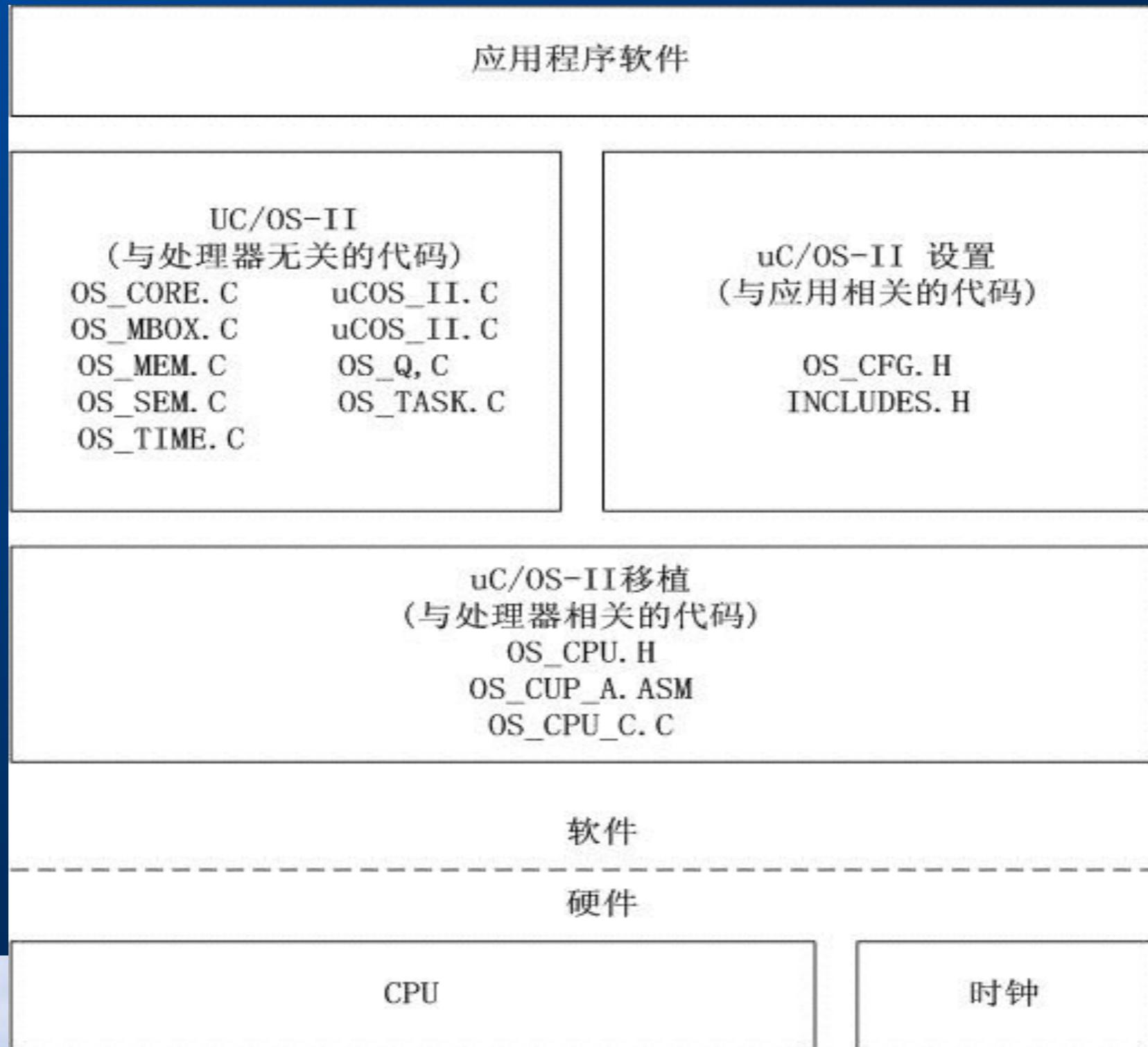
Universal graphical software for embedded applications.



Building Blocks

Round out the capabilities of your embedded design.

μ C/OS-II的文件结构



μ C/OS-II

- Getting Started with μ C/OS-II
- Task Management
- Timer Management
- Inter-task Communication
- Memory Management



任务主函数

```
void YourTask (void *pdata) {  
    for (;;) {  
        /* USER CODE */  
        Call one of uC/OS-II's services:  
        OSFlagPend();  
        OSMboxPend();  
        OSMutexPend();  
        OSQPend();  
        OSSemPend();  
        OSTaskSuspend(OS_PRIO_SELF);  
        OSTimeDly();  
        OSTimeDlyHMSM();  
        /* USER CODE */  
    }  
}  
or,  
void YourTask (void *pdata) {  
    /* USER CODE */  
    OSTaskDel(OS_PRIO_SELF);  
}
```

task priority

- μC/OS-II can manage up to 64 tasks
- although μC/OS-II reserves the four highest priority tasks and the four lowest priority tasks for its own use. However, at this time, only two priority levels are actually used by μC/OS-II: OSTaskCreate and OS_LOWEST_PRIO-1 (see OS_CFG.H). This leaves you with up to 56 application tasks.
- The lower the value of the priority, the higher the priority of the task.
- In the current version of μC/OS-II, the task priority number also serves as the task identifier.

空闲任务和统计任务

- 内核总是创建一个空闲任务OSTaskIdle();
 - 总是设置为最低优先级，OS_LOWEST_PRIOR；
 - 当所有其他任务都未在执行时，空闲任务开始执行；
 - 应用程序不能删除该任务；
 - 空闲任务的工作就是把32位计数器OSIdleCtr加1，该计数器被统计任务所使用；
- 统计任务OSTaskStat(), 提供运行时间统计。每秒钟运行一次，计算当前的CPU利用率。其优先级是OS_LOWEST_PRIOR-1，可选。

任务控制块TCB

- 任务控制块 OS_TCB是描述一个任务的核心数据结构，存放了它的各种管理信息，包括任务堆栈指针，任务的状态、优先级，任务链表指针等；
- 一旦任务建立了，任务控制块OS_TCB将被赋值。

任务控制块TCB

```
typedef struct os_tcb
{
    栈指针;
    INT16U OSTCBId;      /*任务的ID*/
    链表指针;
    OS_EVENT *OSTCBEventPtr; /*事件指针*/
    void *OSTCBMsg;        /*消息指针*/
    INT8U OSTCBStat;       /*任务的状态*/
    INT8U OSTCBPrio;       /*任务的优先级*/
    其他.....
} OS_TCB;
```

栈指针

- OSTCBStkPtr：指向当前任务栈顶的指针，每个任务可以有自己的栈，栈的容量可以是任意的；
- OSTCBStkBottom：指向任务栈底的指针；
- OSTCBStkSize：栈的容量，用可容纳的指针数目而不是字节数（Byte）来表示。

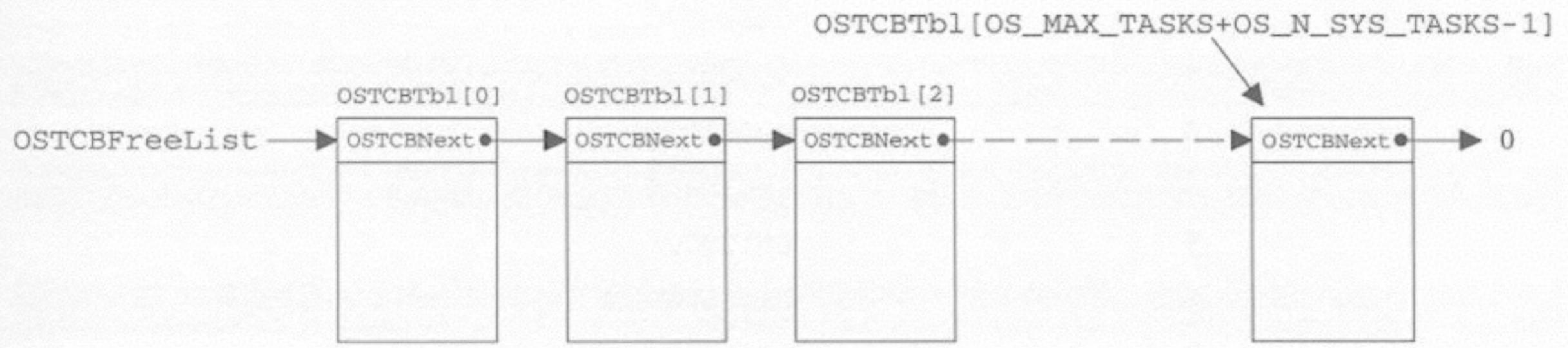
链表指针

- 所有的任务控制块分属于两条不同的链表，单向的空闲链表（头指针为OSTCBFreeList）和双向的使用链表（头指针为OSTCBLList）；
- OSTCBNext、OSTCBPrev：用于将任务控制块插入到空闲链表或使用链表中。每个任务的任务控制块在任务创建的时候被链接到使用链表中，在任务删除的时候从链表中被删除。双向连接的链表使得任一成员都能快速插入或删除。

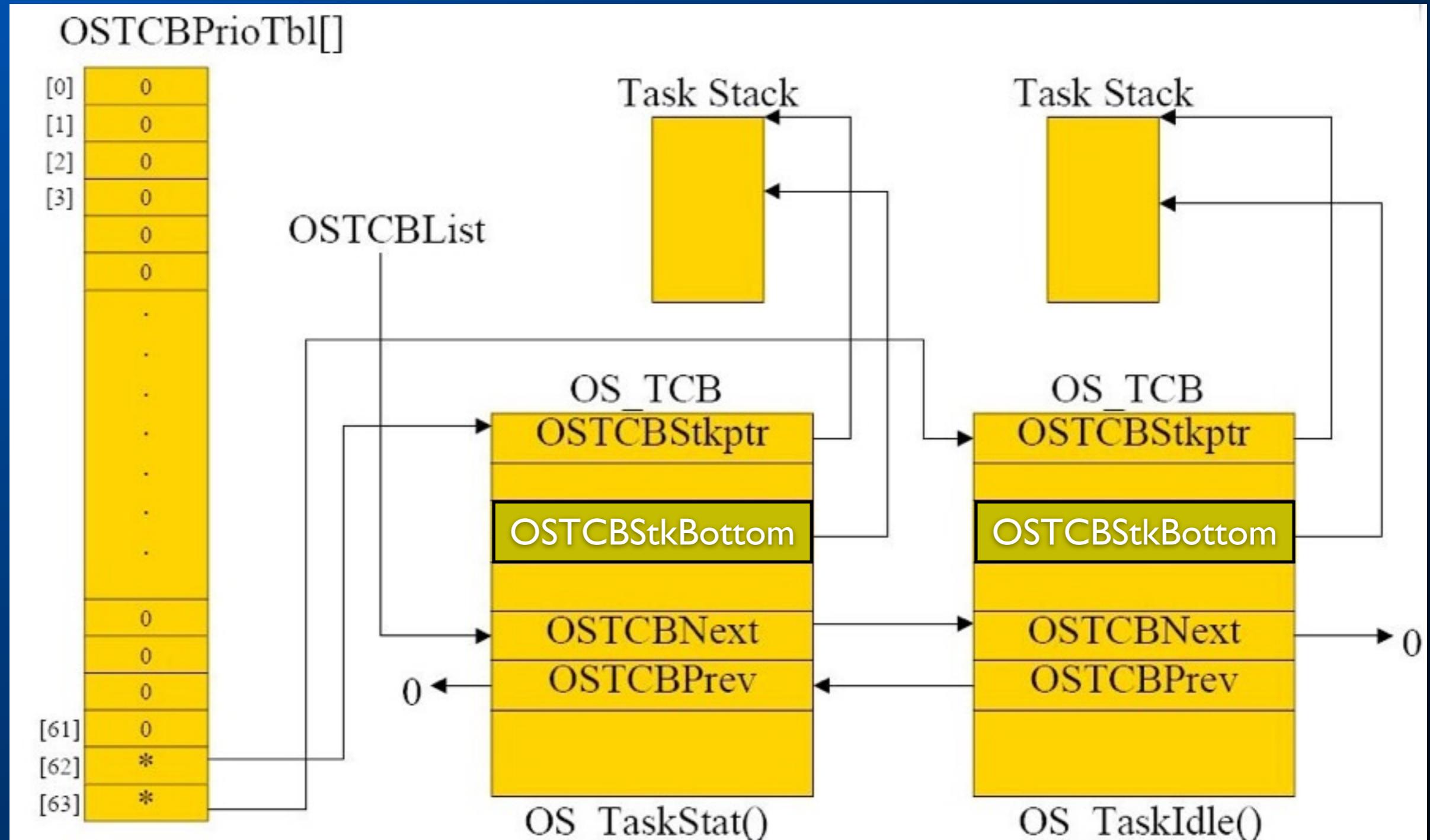
空闲TCB链表

- 所有的任务控制块都被放置在任务控制块列表数组OSTCBtbl[]中，系统初始化时，所有TCB被链接成空闲的单向链表，头指针为OSTCBFreeList。当创建一个任务后，就把OSTCBFreeList所指向的TCB赋给了该任务，并将它加入到使用链表中，然后把OSTCBFreeList指向空闲链表中的下一个结点。

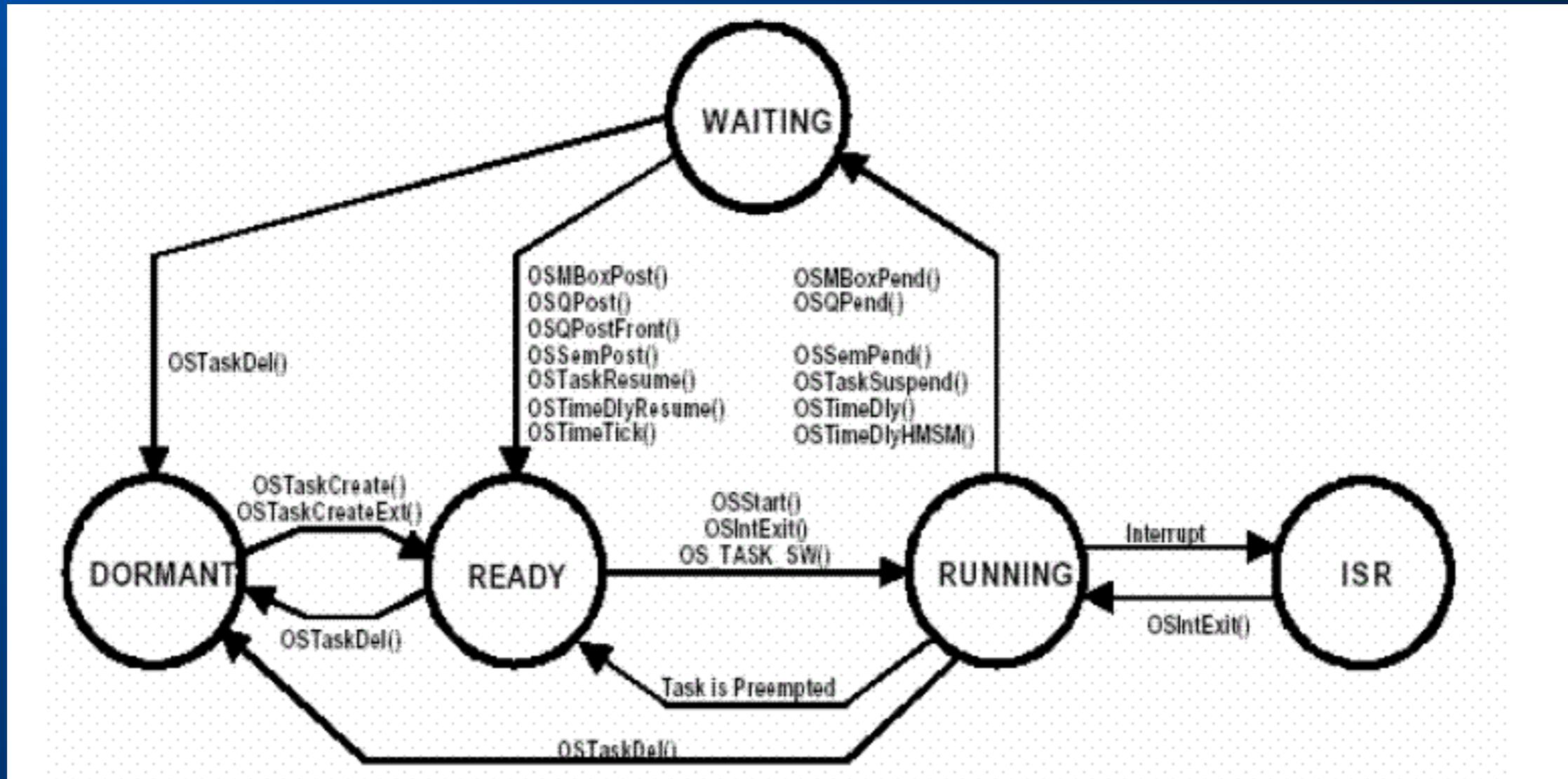
Figure 3.2 List of free OS_TCBs.



指针数组(指向相应TCB)



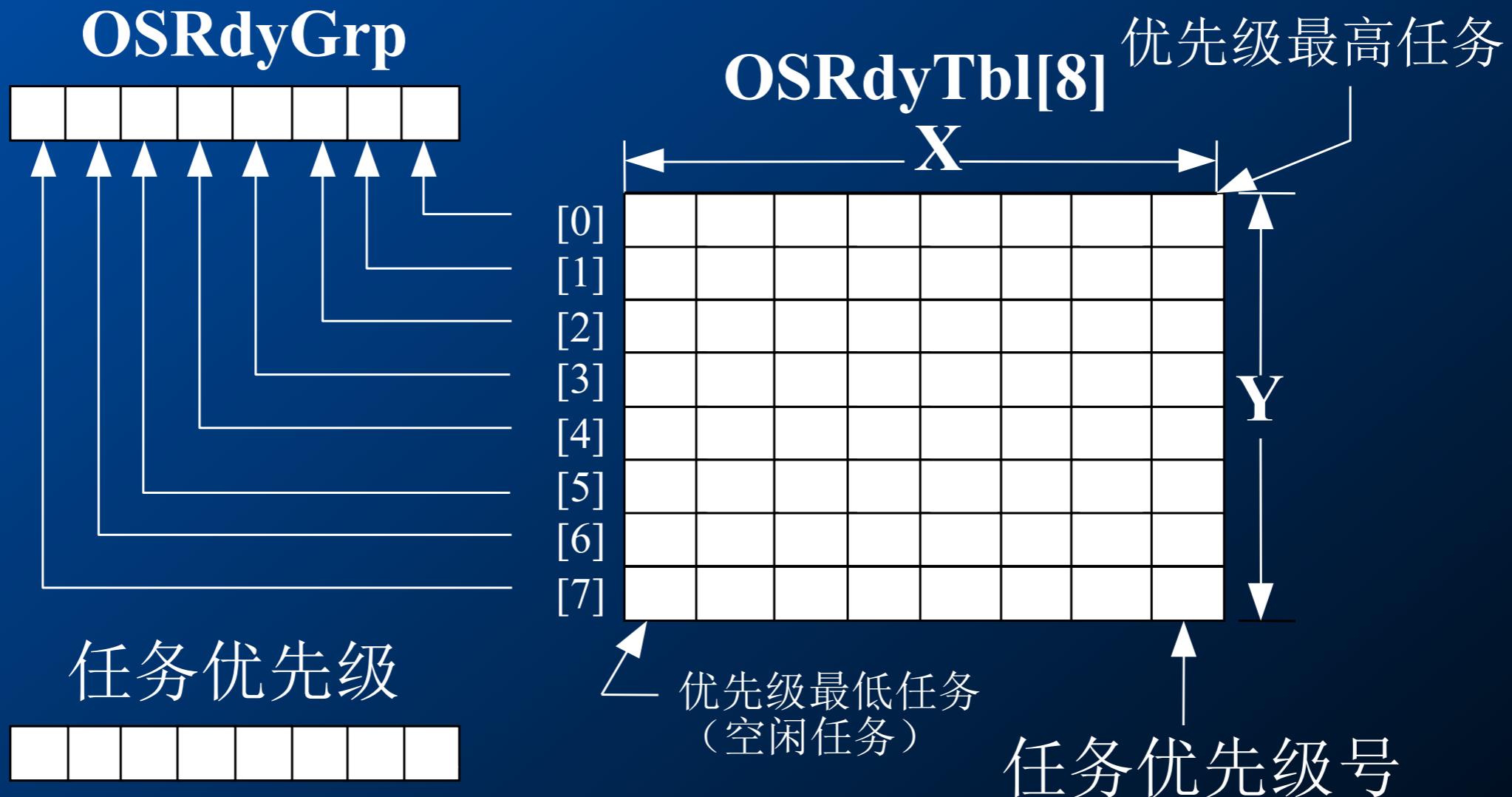
状态的转换



任务就绪表

- 每个任务的就绪态标志放入在就绪表中，就绪表中有两个变量OSRdyGrp和OSRdyTbl[]。
- 在OSRdyGrp中，任务按优先级分组，8个任务为一组。OSRdyGrp中的每一位表示8组任务中每一组中是否有进入就绪态的任务。任务进入就绪态时，就绪表OSRdyTbl[]中的相应元素的相应位也置位。

任务就绪表



对于整数 $\text{OSRdyTbl}[i]$ ($0 \leq i \leq 7$)，若它的某一位为1，则 OSRdyGrp 的第 i 位为1。
任务的优先级由 X 和 Y 确定

根据优先级确定就绪表(I)

- 假设优先级为12的任务进入就绪状态， $12=1100b$,则OSRdyTbl[1]的第4位置1，且OSRdyGrp的第1位置1，相应的数学表达式为：
 - $OSRdyGrp \quad | = 0x02;$
 - $OSRdyTbl[1] \quad | = 0x10;$
- 而优先级为21的任务就绪 $21=10\ 101b$ ， 则OSRdyTbl[2]的第5位置1，且OSRdyGrp的第2位置1,相应的数学表达式为：
 - $OSRdyGrp \quad | = 0x04;$
 - $OSRdyTbl[2] \quad | = 0x20;$

根据优先级确定就绪表(2)

- 从上面的计算可知: 若OSRdyGrp及OSRdyTbl[0]的第n位置1, 则应该把OSRdyGrp及OSRdyTbl[0]的值与 2^n 相或。uC/OS中, 把 2^n 的n=0-7的8个值先计算好存在数组OSMapTbl[7]中,也就是:
 - $OSMapTbl[0] = 2^0 = 0x01 \text{ (0000 0001)}$
 - $OSMapTbl[1] = 2^1 = 0x02 \text{ (0000 0010)}$
 -
 - $OSMapTbl[7] = 2^7 = 0x80 \text{ (1000 0000)}$

使任务进入就绪态

- 如果prio是任务的优先级，即任务的标识号，则将任务放入就绪表，使任务进入就绪态的方法是：
 - OSRdyGrp | = OSMapTbl[prio>>3];
 - OSRdyTbl[prio>>3] |= OSMapTbl[prio&0x07];
- 假设优先级为12——1100b
 - OSRdyGrp |= OSMapTbl[12>>3](0x02);
 - OSRdyTbl[1] |= 0x10;

使任务脱离就绪态

- 将任务就绪表OSRdyTbl[prio>>3]相应元素的相应位清零，而且当OSRdyTbl[prio>>3]中的所有位都为零时，即该任务所在组的所有任务中没有一个进入就绪态时，OSRdyGrp的相应位才为零。

```
if((OSRdyTbl[prio>>3] &= ~OSMapTbl[prio&0x07]) == 0)  
    OSRdyGrp &= ~OSMapTbl[prio>>3];
```

任务的调度

- μC/OS-II是可抢占实时多任务内核，它总是运行就绪任务中优先级最高的那一个。
- μC/OS-II中不支持时间片轮转法，每个任务的优先级要求不一样且是唯一的，所以任务调度的工作就是：查找准备就绪的最高优先级的任务并进行上下文切换。
- μC/OS-II任务调度所花的时间为常数，与应用程序中建立的任务数无关。

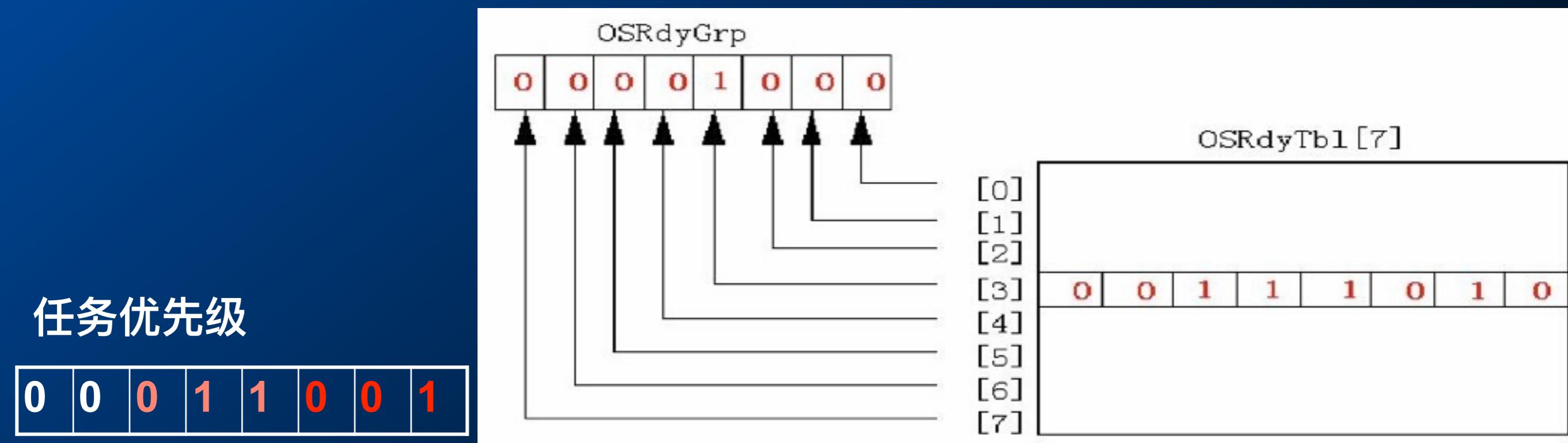
- 确定哪个任务的优先级最高，应该选择哪个任务去运行，这部分的工作是由调度器（Scheduler）来完成的。
 - 任务级的调度是由函数OSSched()完成的；
 - 中断级的调度是由另一个函数OSIntExt()完成的。

根据就绪表确定最高优先级

- 两个关键：
 - 将优先级数分解为高三位和低三位分别确定；
 - 高优先级有着小的优先级号；

根据就绪表确定最高优先级

- 通过OSRdyGrp值确定高3位，假设OSRdyGrp = 0x08=0x00001000，第3位为1，优先级的高3位为011；
- 通过OSRdyTbl[3]的值来确定低3位，假设OSRdyTbl[3] = 0x3a，第1位为1，优先级的低3位为001， $3*8+2-1=25$



任务调度器

```
void OSSched (void)
{
    INT8U y;
    OS_ENTER_CRITICAL();
    if ((OSLockNesting | OSIntNesting) == 0) {
        y = OSUnMapTbl[OSRdyGrp];
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
        if (OSPrioHighRdy != OSPrioCur) {
            OSTCBHighRdy=OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;
            OS_TASK_SW();
        }
    }
    OS_EXIT_CRITICAL();
}
```

检查是否中断调用和允许任务调用

找到优先级最高的任务

该任务是否正在运行

源代码中使用了查表法

- 查表法具有确定的时间，增加了系统的可预测性，uC/OS中所有的系统调用时间都是确定的
 - $Y = OSUnMapTbl[OSRdyGrp];$
 - $X = OSUnMapTbl[OSRdyTbl[Y]];$
 - $Prio = (Y << 3) + X;$

参见OS_CORE.C

优先级判定表OSUnMapTbl[256]

```
INT8U const OSUnMapTbl[] = {  
    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  
};
```

举例：

如**OSRdyGrp**的值为**01101000B**, 即**0X68**,
则查得**OSUnMapTbl[OSRdyGrp]**的值是
3, 它相应于**OSRdyGrp**中的第**3**位置**1**;

如**OSRdyTbl[3]**的值是**11100100B**, 即
0XE4, 则查**OSUnMapTbl[OSRdyTbl[3]]**
的值是**2**, 则进入就绪态的最高任务优先
级

$$\text{Prio} = 3 * 8 + 2 = 26$$

64 --> 256

```
static void OS_SchedNew (void)
{
#if OS_LOWEST_PRIO <= 63 //μC/OS-II v2.7之前方式
    INT8U y;
    y = OSUnMapTbl[OSRdyGrp];
    OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
#else
    INT8U y;
    INT16U *ptbl;
    //OSRdyGrp为16位
    if ((OSRdyGrp & 0xFF) != 0) {
        y = OSUnMapTbl[OSRdyGrp & 0xFF];
    } else {
        y = OSUnMapTbl[(OSRdyGrp >> 8) & 0xFF] + 8; //矩形组号y>=8
    }
    ptbl = &OSRdyTbl[y]; //取出x方向的16bit数据
    if ((*ptbl & 0xFF) != 0) {
        OSPrioHighRdy = (INT8U)((y << 4) + OSUnMapTbl[*ptbl & 0xFF]); /*16
    } else {
        OSPrioHighRdy = (INT8U)((y << 4) + OSUnMapTbl[*ptbl >> 8] & 0xFF] + 8);
    }
#endif
}
```

任务切换

- 将被挂起任务的寄存器内容入栈；
- 将较高优先级任务的寄存器内容出栈，恢复到硬件寄存器中。

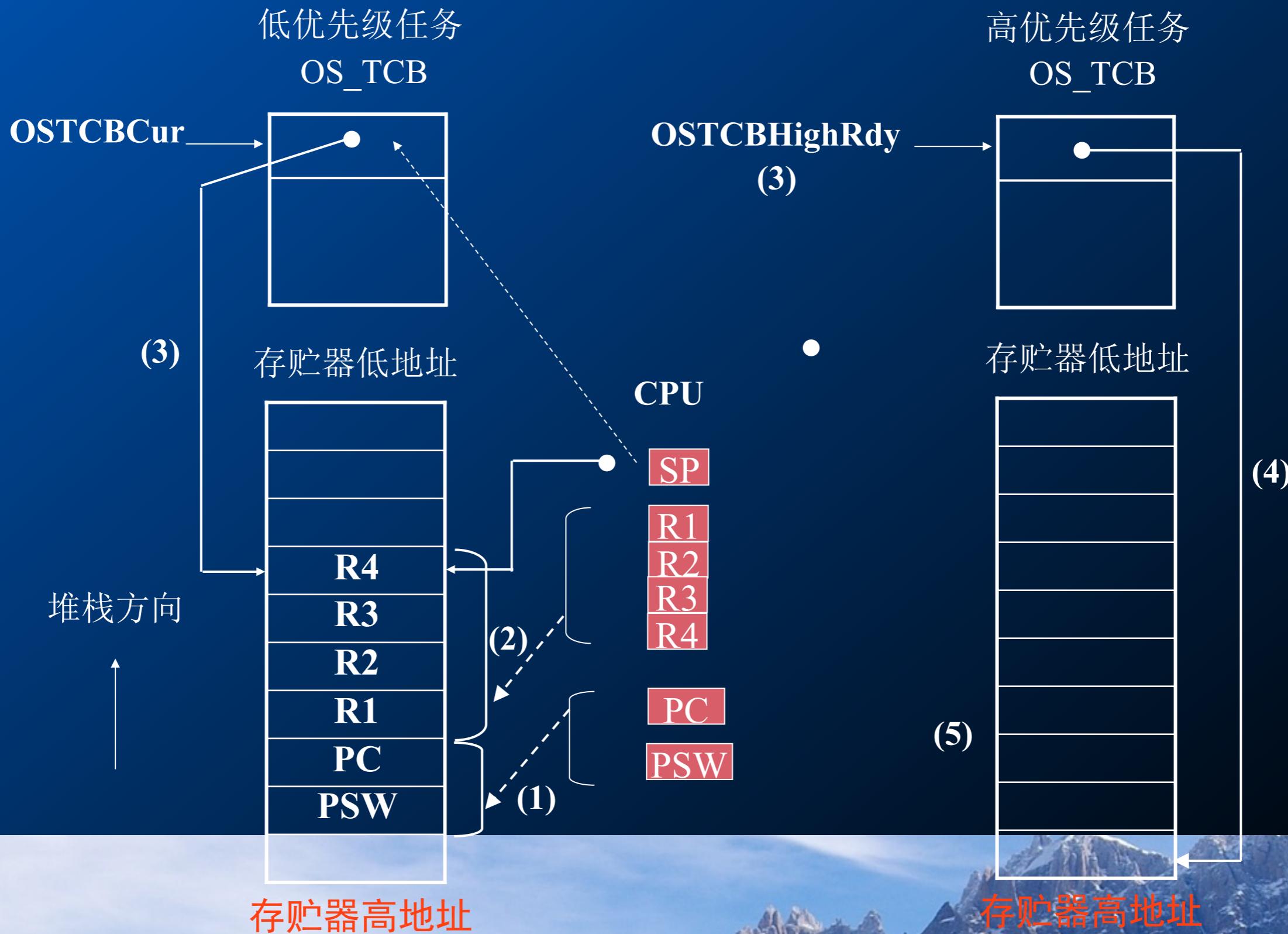
任务级的任务切换OS_TASK_SW()

- 保护当前任务的现场
- 恢复新任务的现场
- 执行中断返回指令
- 开始执行新的任务

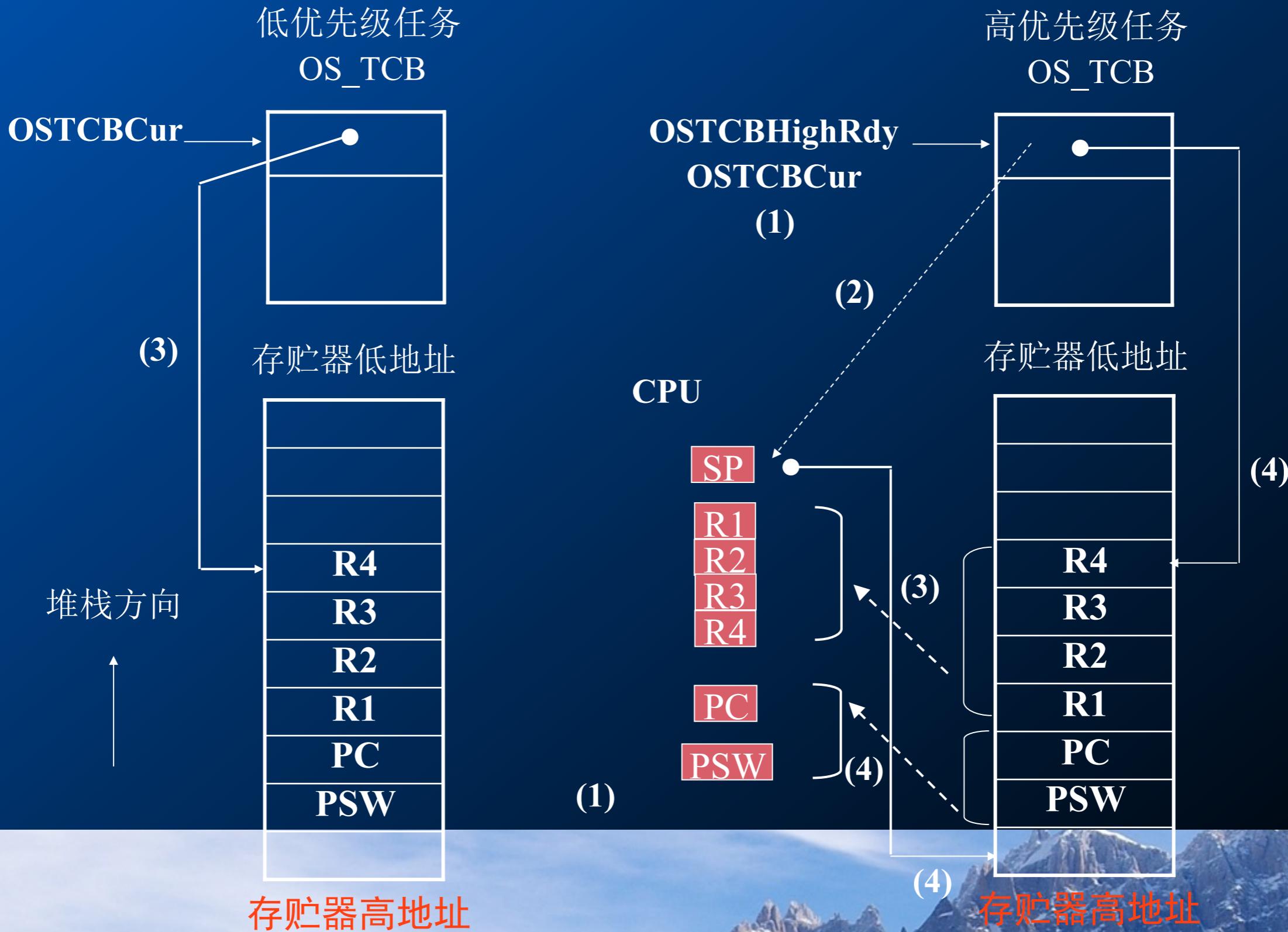
调用OS_TASK_SW()前的数据结构



保存当前CPU寄存器的值



重新装入要运行的任务



任务切换OS_TASK_SW()的代码

```
Void OSCTxSw(void)
```

```
{
```

将R1,R2,R3及R4推入当前堆栈；

OSTCBCur→OSTCBStkPtr = SP;

OSTCBCur = OSTCBHighRdy;

SP = OSTCBHighRdy → OSTCBSTKPtr;

将R4,R3,R2及R1从新堆栈中弹出；

执行中断返回指令；

```
}
```

任务管理的系统服务

- 创建任务
- 删除任务
- 修改任务的优先级
- 挂起和恢复任务
- 获得一个任务的有关信息

创建任务

- 创建任务的函数
 - OSTaskCreate();
 - OSTaskCreateExt();
- OSTaskCreateExt()是OSTaskCreate()的扩展版本，提供了一些附加的功能；
- 任务可以在多任务调度开始(即调用OSStart())之前创建，也可以在其它任务的执行过程中被创建。但在OSStart()被调用之前，用户必须创建至少一个任务；
- 不能在中断服务程序(ISR)中创建新任务。

OSTaskCreate()

```
INT8U OSTaskCreate (
    void (*task)(void *pd), //任务代码指针
    void *pdata, //任务参数指针
    OS_STK *ptos, //任务栈的栈顶指针
    INT8U prio //任务的优先级
);
```

- 返回值
 - OS_NO_ERR: 函数调用成功;
 - OS_PRIO_EXIT: 任务优先级已经存在;
 - OS_PRIO_INVALID: 任务优先级无效。

OSTaskCreate()的实现过程

- 任务优先级检查
 - 该优先级是否在0到OS_LOWEST_PRIO之间?
 - 该优先级是否空闲?
- 调用OSTaskStkInit(), 创建任务的栈帧;
- 调用OSTCBInit(), 从空闲的OS_TCB池 (即OSTCBFreeList链表) 中获得一个TCB并初始化其内容, 然后把它加入到OSTCBList链表的开头, 并把它设定为就绪状态;
- 任务个数OSTaskCtr加1;
- 调用用户自定义的函数OSTaskCreateHook();
- 判断是否需要调度 (调用者是正在执行的任务)

OSTaskCreateExt()

- INT8U OSTaskCreateExt(

前四个参数与OSTaskCreate相同，

INT16U id, //任务的ID

OS_STK *pbos, //指向任务栈底的指针

INT32U stk_size, //栈能容纳的成员数目

void *pext, //指向用户附加数据域的指针

INT16U opt //一些选项信息

);

- 返回值：与OSTaskCreate()相同。

任务的栈空间

- 每个任务都有自己的栈空间（Stack），栈必须声明为 OS_STK类型，并且由连续的内存空间组成；
- 栈空间的分配方法
 - 静态分配：在编译的时候分配，例如：

```
static OS_STK MyTaskStack[stack_size];
```

```
OS_STK MyTaskStack[stack_size];
```
 - 动态分配：在任务运行的时候使用malloc()函数来动态申请内存空间；

动态分配

```
OS_STK *pstk;
```

```
pstk = (OS_STK *)malloc(stack_size);
```

```
/* 确认malloc()能得到足够的内存空间 */
```

```
if (pstk != (OS_STK *)0)
```

```
{
```

```
Create the task;
```

```
}
```

内存碎片问题

在动态分配中，可能存在内存碎片问题。特别是当用户反复地建立和删除任务时，内存堆中可能会出现大量的碎片，导致没有足够大的一块连续内存区域可用作任务栈，这时malloc()便无法成功地为任务分配栈空间。



栈的增长方向

- 栈的增长方向的设置
 - 从低地址到高地址：在OS_CPU.H中，将常量 OS_STK_GROWTH设定为 0；
 - 从高地址到低地址：在OS_CPU.H中，将常量 OS_STK_GROWTH设定为 1；
 - OS_STK TaskStack[TASK_STACK_SIZE];
 - OSTaskCreate(task, pdata,
&TaskStack[TASK_STACK_SIZE-1],
prio);

删除任务

- OSTaskDel(): 删除一个任务，其TCB会从所有可能的系统数据结构中移除。任务将返回并处于休眠状态（任务的代码还在）。
 - 如果任务正处于就绪状态，把它从就绪表中移出，这样以后就不会再被调度执行了；
 - 如果任务正处于邮箱、消息队列或信号量的等待队列中，也把它移出；
 - 将任务的OS_TCB从OSTCBLList链表当中移动到OSTCBFreeList。

任务删除

- 任务也可以自我删除（并非真的删除，只是内核不再知道该任务）

```
void MyTask (void *pdata)  
{  
    ..... /* 用户代码 */  
    OSTaskDel(OS_PRIO_SELF);  
}
```

- OSTaskChangePrio(): 在程序运行期间，用户可以通过调用本函数来改变某个任务的优先级。

```
INT8U OSTaskChangePrio(INT8U oldprio,  
                         INT8U newprio)
```

- OSTaskQuery(): 获得一个任务的有关信息
 - 获得的是对应任务的OS_TCB中内容的拷贝。

挂起和恢复任务

- OSTaskSuspend(): 挂起一个任务
 - 如果任务处于就绪态，把它从就绪表中移出；
 - 在任务的TCB中设置OS_STAT_SUSPEND标志，表明该任务正在被挂起。
- OSTaskResume(): 恢复一个任务
 - 恢复被OSTaskSuspend()挂起的任务；
 - 清除TCB中OSTCBStat字段的OS_STAT_SUSPEND位

实时操作系统μC/OS-II

- μC/OS-II概述
- 任务管理
- 中断和时间管理
- 任务之间的通信与同步
- 存储管理

中断处理

- 中断：由于某种事件的发生而导致程序流程的改变。产生中断的事件称为中断源。
- CPU响应中断的条件：
 - 至少有一个中断源向CPU发出中断信号；
 - 系统允许中断，且对此中断信号未予屏蔽。

中断服务程序ISR

- 中断一旦被识别，CPU会保存部分（或全部）运行上下文（context，即寄存器的值），然后跳转到专门的子程序去处理此次事件，称为中断服务子程序(ISR)。
- μC/OS-II 中，中断服务子程序要用汇编语言来编写，然而，如果用户使用的C语言编译器支持在线汇编语言的话，用户可以直接将中断服务子程序代码放在C语言的程序文件中。

用户ISR的框架

1. 保存全部CPU寄存器的值;
2. 调用OSIntEnter(), 或直接把全局变量OSIntNesting (中断嵌套层次) 加1;
3. 执行用户代码做中断服务;
4. 调用OSIntExit();
5. 恢复所有CPU寄存器;
6. 执行中断返回指令。

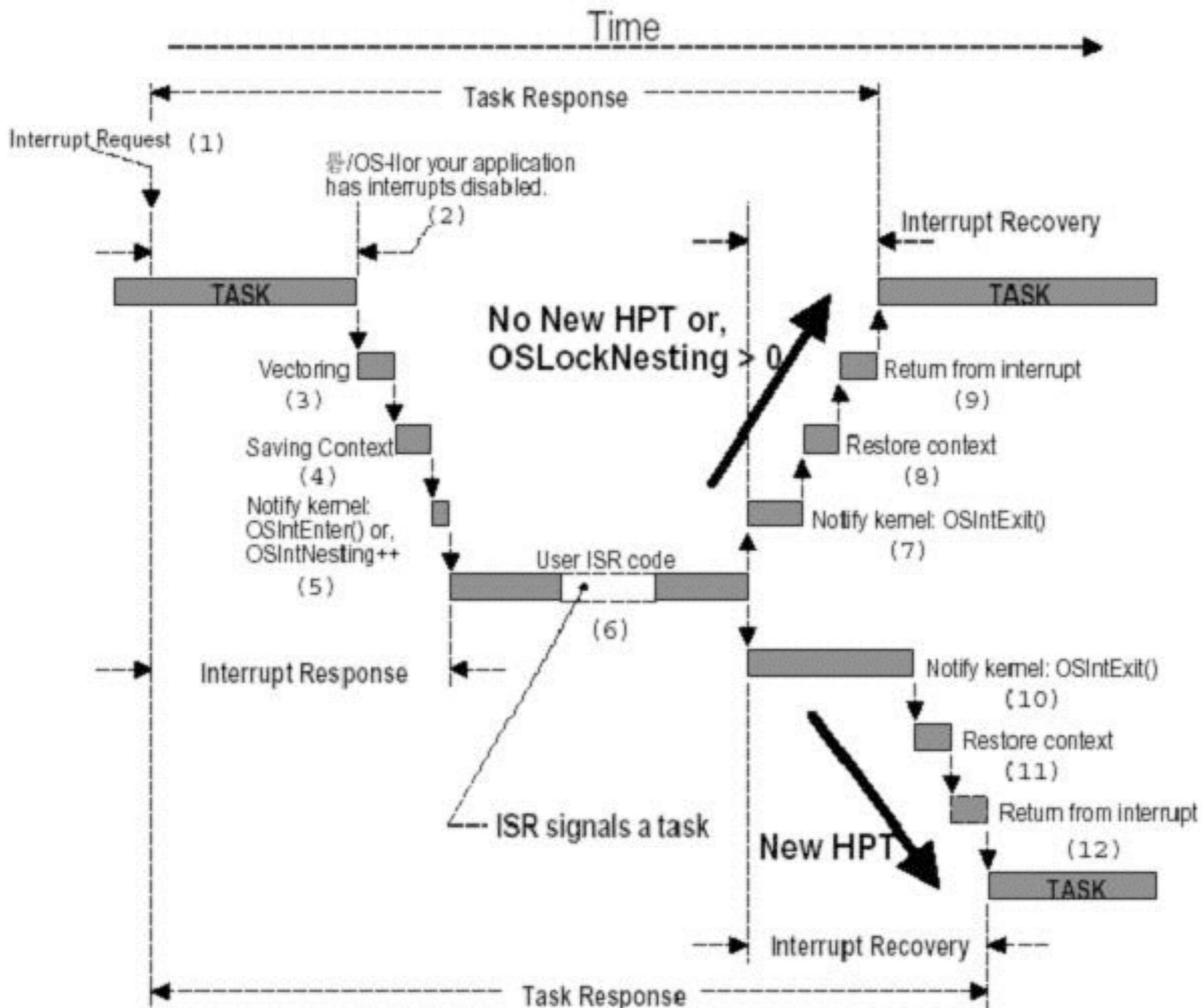


Figure 3-5, Servicing an interrupt

OSIntEnter()

```
/* 在调用本函数之前必须先将中断关闭 */
```

```
void OSIntEnter (void)
```

```
{
```

```
    if (OSRunning == TRUE) {
```

```
        if (OSIntNesting < 255) {
```

```
            OSIntNesting++;
```

```
}
```

```
}
```

```
}
```

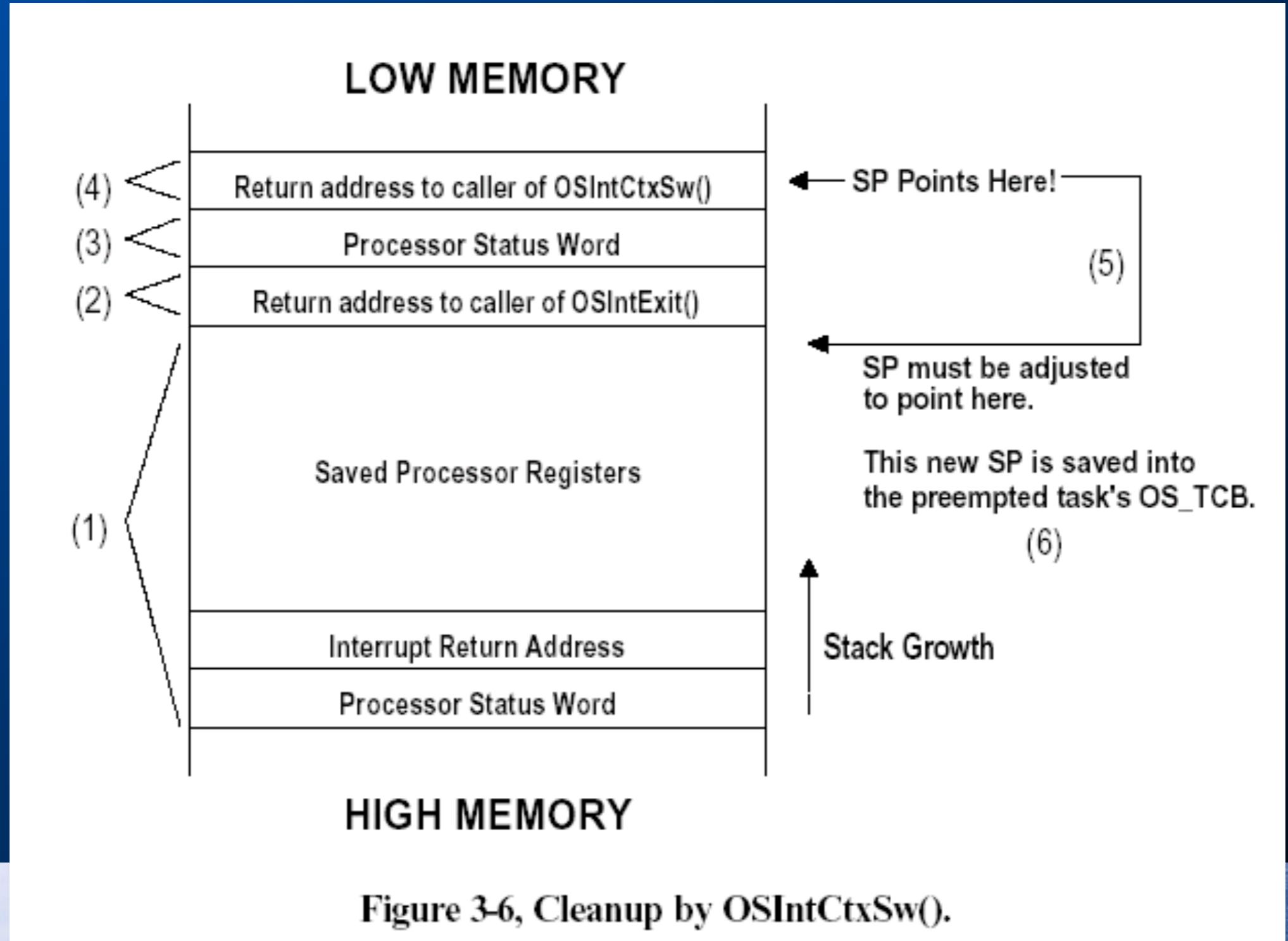
OSIntExit()

```
void OSIntExit (void)
{
    OS_ENTER_CRITICAL(); //关中断
    if ((--OSIntNesting | OSLockNesting) == 0) //判断嵌套是否为零
    { //把高优先级任务装入
        OSIntExitY = OSUnMapTbl[OSRdyGrp];
        OSPrioHighRdy=(INT8U)((OSIntExitY<< 3) +
            OSUnMapTbl[OSRdyTbl[OSIntExitY]]);
        if (OSPrioHighRdy != OSPrioCur) {
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;
            OSIntCtxSw();
        }
    }
    OS_EXIT_CRITICAL(); //开中断返回
}
```

OSIntCtxSw()

- 在任务切换时，为什么使用OSIntCtxSw()而不是调度函数中的OS_TASK_SW()？
- 原因如下：
 - 一半的任务切换工作，即CPU寄存器入栈，已经在前面做完了；
 - 需要保证所有被挂起任务的栈结构是一样的。

调用中断切换函数OSIntCtxSw() 后的堆栈情况



时钟节拍

- 时钟节拍是一种特殊的中断；
- μC/OS需要用户提供周期性信号源，用于实现时间延时和确认超时。节拍率应在10到100Hz之间，时钟节拍率越高，系统的额外负荷就越重；
- 时钟节拍的实际频率取决于用户应用程序的精度。时钟节拍源可以是专门的硬件定时器，或是来自50/60Hz交流电源的信号。

时钟节拍ISR

```
void OSTickISR(void)
```

```
{
```

- (1)保存处理器寄存器的值;
- (2)调用OSIntEnter()或将OSIntNesting加1;
- (3)调用OSTimeTick(); /*检查每个任务的时间延时*/
- (4)调用OSIntExit();
- (5)恢复处理器寄存器的值;
- (6)执行中断返回指令;

```
}
```

时钟节拍函数 OSTimetick()

时间管理

- 与时间管理相关的系统服务：
 - OSTimeDLY()
 - OSTimeDLYHMSM()
 - OSTimeDlyResume()
 - OSTimeGet()
 - OSTimeSet()

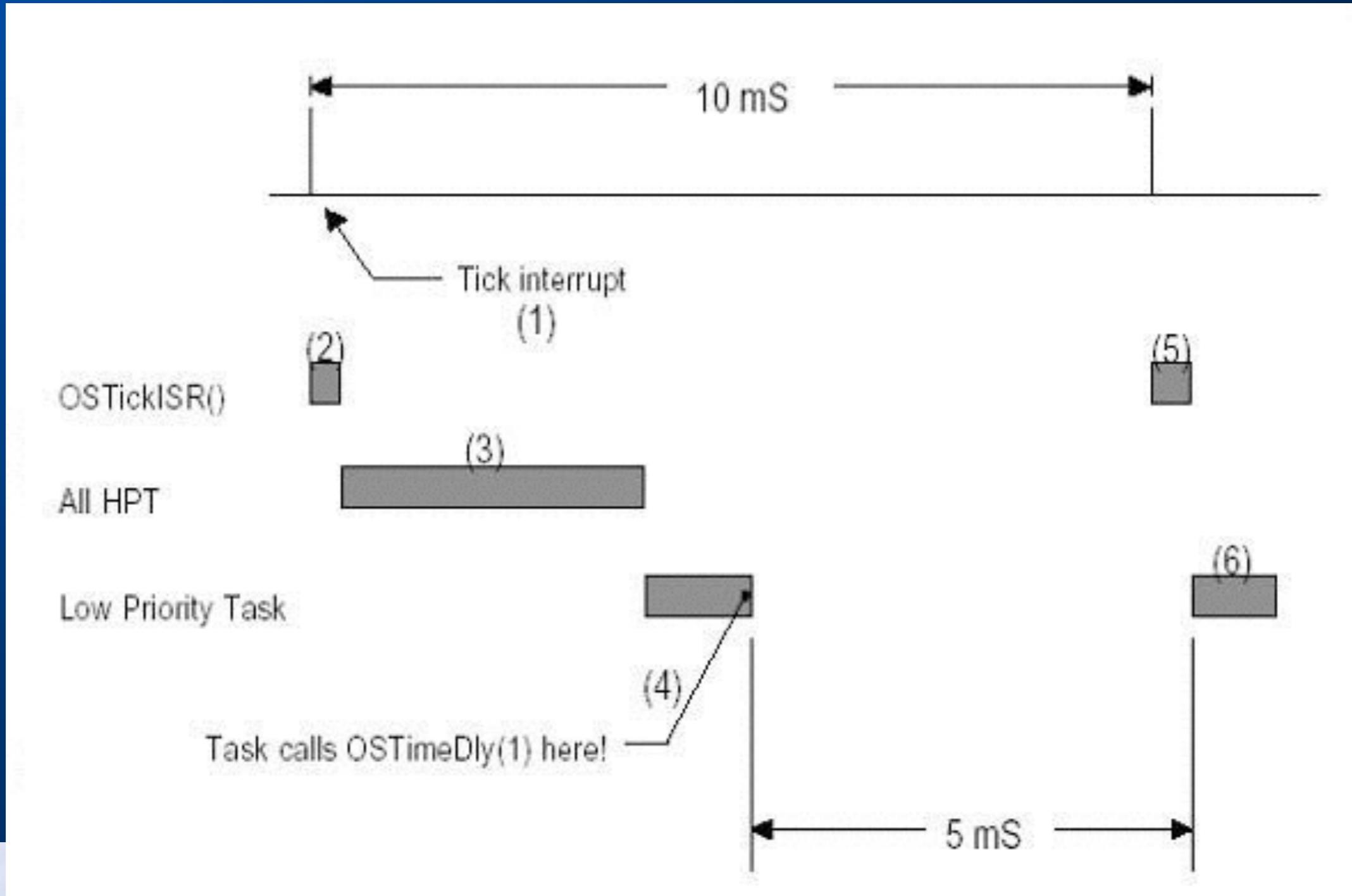
OSTimeDLY()

- OSTimeDLY(): 任务延时函数，申请该服务的任务可以延时一段时间；
- 调用OSTimeDLY后，任务进入等待状态；
- 使用方法
 - void OSTimeDly (INT16U ticks);
 - ticks表示需要延时的时间长度，用时钟节拍的个数来表示。

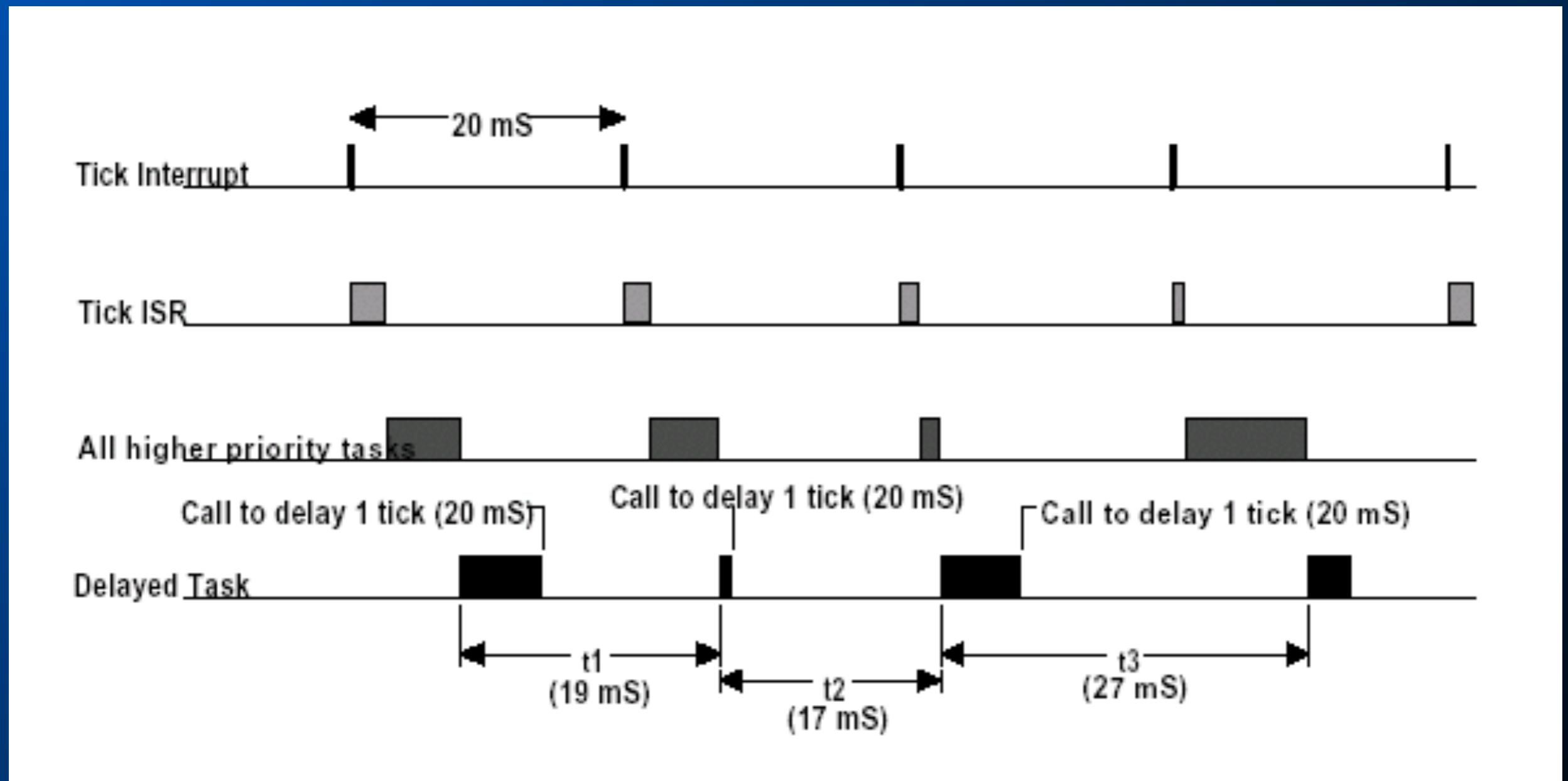
OSTimeDLY()

```
void OSTimeDly (INT16U ticks)
{
    if (ticks > 0)
    {
        OS_ENTER_CRITICAL();
        if ((OSRdyTbl[OSTCBCur->OSTCBY] &=
            ~OSTCBCur->OSTCBBitX) == 0)
        {
            OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
        }
        OSTCBCur->OSTCBDly = ticks;
        OS_EXIT_CRITICAL();
        OSSched();
    }
}
```

OSTimeDLY(1)的问题

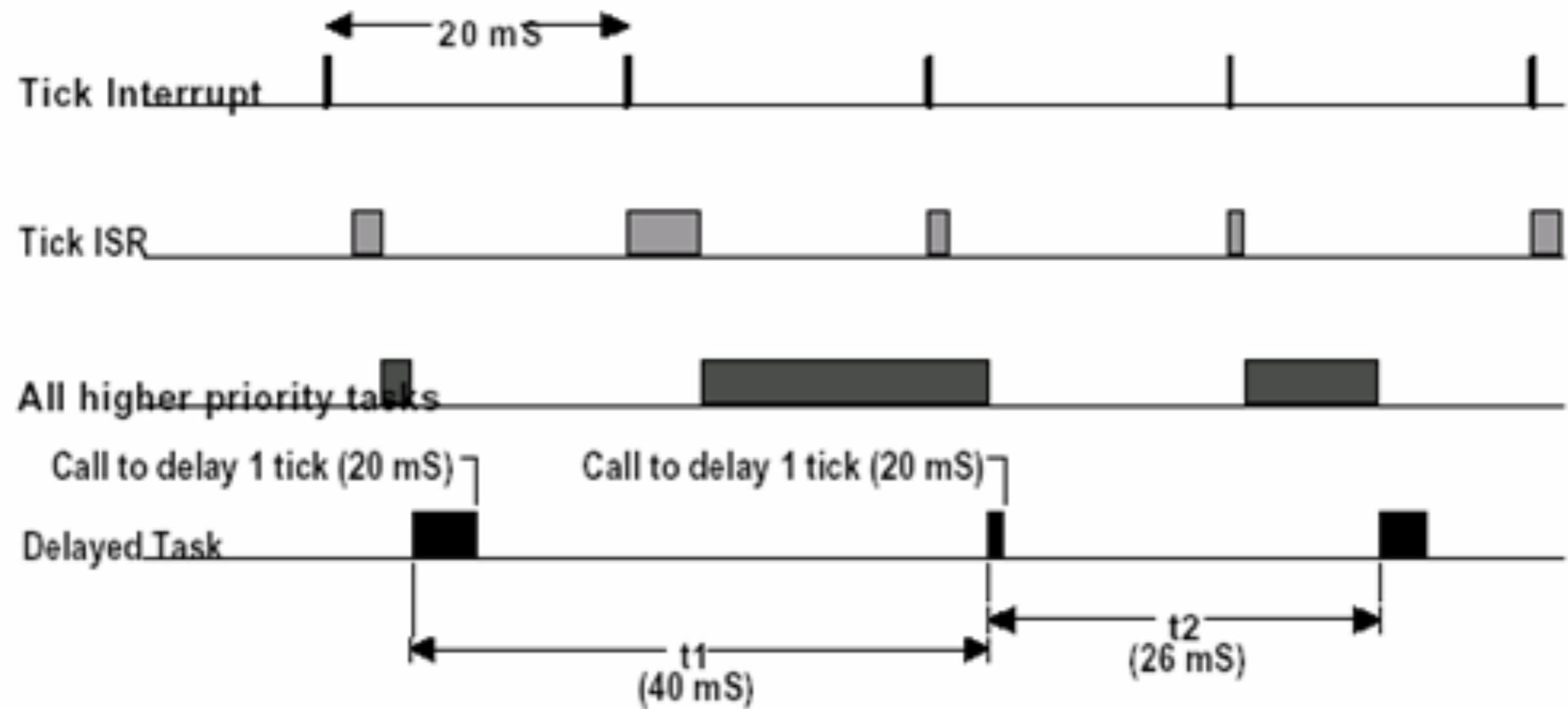


OSTimeDLY的问题(I)



将任务延迟一个时钟节拍(第一种情况)

OSTimeDLY的问题(2)



将任务延迟一个时钟节拍(第二种情况)

OSTimeDlyHMSM()

- OSTimeDlyHMSM(): OSTimeDly()的另一个版本，即按时分秒延时函数；
- 使用方法

```
INT8U OSTimeDlyHMSM(  
    INT8U hours, // 小时  
    INT8U minutes, // 分钟  
    INT8U seconds, // 秒  
    INT16U milli // 毫秒  
);
```

OSTimeDlyResume()

- OSTimeDlyResume(): 让处在延时期的任务提前结束延时，进入就绪状态；
- 使用方法
 - INT8U OSTimeDlyResume (INT8U prio);
 - prio表示需要提前结束延时的任务的优先级/任务ID。

系统时间

- 每隔一个时钟节拍，发生一个时钟中断，将一个32位的计数器OSTime加1；
- 该计数器在用户调用OSStart()初始化多任务和4,294,967,295个节拍执行完一遍的时候从0开始计数。若时钟节拍的频率等于100Hz，该计数器每隔497天就重新开始计数；
- OSTimeGet(): 获得该计数器的当前值；
 - INT32U OSTimeGet (void);
- OSTimeSet(): 设置该计数器的值。
 - void OSTimeSet (INT32U ticks);

何时启动系统定时器

- 如果在OSStart之前启动定时器，则系统可能无法正确执行完OSStartHighRdy
- OSStart函数直接调用OSStartHighRdy去执行最高优先级的任务，OSStart不返回
- 系统定时器应该在系统的最高优先级任务中启动
- 使用OSRunning变量来控制操作系统的运行

时钟节拍的启动

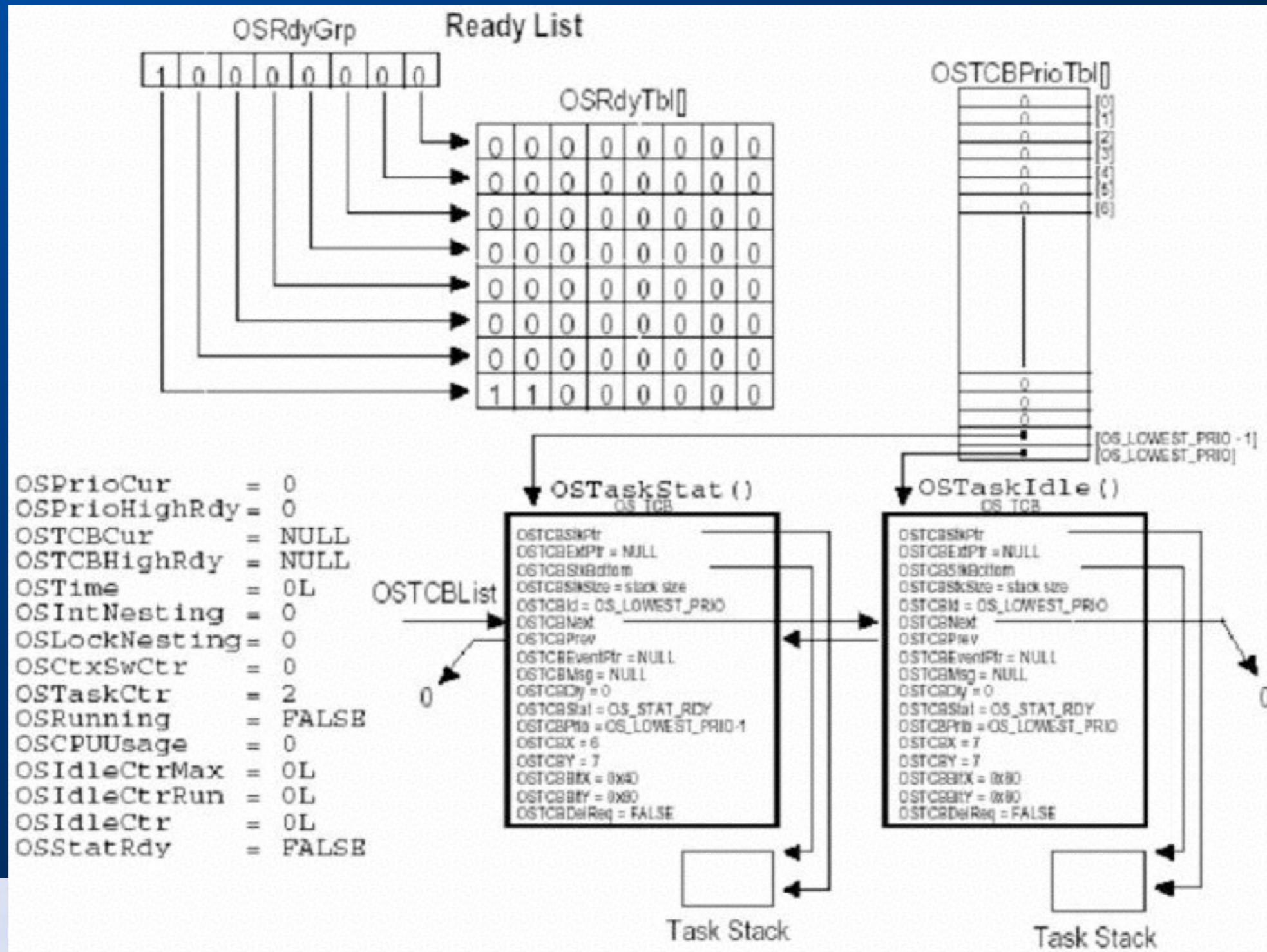
- 用户必须在多任务系统启动以后再开启时钟节拍器，也就是在调用OSStart()之后
- 在调用OSStart()之后做的第一件事是初始化定时器中断

```
void main(void)
{
    ...
    OSInit(); /* 初始化uC/OS-II*/
    /* 应用程序初始化代码... */
    /* 调用OSTaskCreate()创建至少一个任务*/
    允许时钟节拍中断; /* 错误！可能crash!*/
    OSStart(); /* 开始多任务调度 */
}
```

系统的初始化与启动

- 在调用μC/OS-II的任何其它服务之前，用户必须首先调用系统初始化函数OSInit()来初始化μC/OS的所有变量和数据结构；
- OSInit()建立空闲任务OSTaskIdle()，该任务总是处于就绪状态，其优先级一般被设成最低，即OS_LOWEST_PRIO；如果需要，OSInit()还建立统计任务OSTaskStat()，并让其进入就绪状态；
- OSInit()还初始化了4个空数据结构缓冲区：空闲TCB链表OSTCBFreeList、空闲事件链表OSEventFreeList、空闲队列链表OSQFreeList和空闲存储链表OSMemFreeList。

系统初始化后的状态



μ C/OS-II的启动

- 多任务的启动是用户通过调用OSStart()实现的。然而，启动 μ C/OS-II之前，用户至少要建立一个应用任务。

```
void main (void)
{
    OSInit(); /* 初始化uC/OS-II */
    ...
    // 通过调用OSTaskCreate()或OSTaskCreateExt()
    // 创建至少一个任务;
    ...
    OSStart(); /*开始多任务调度! 永不返回*/
}
```

OSSStart()

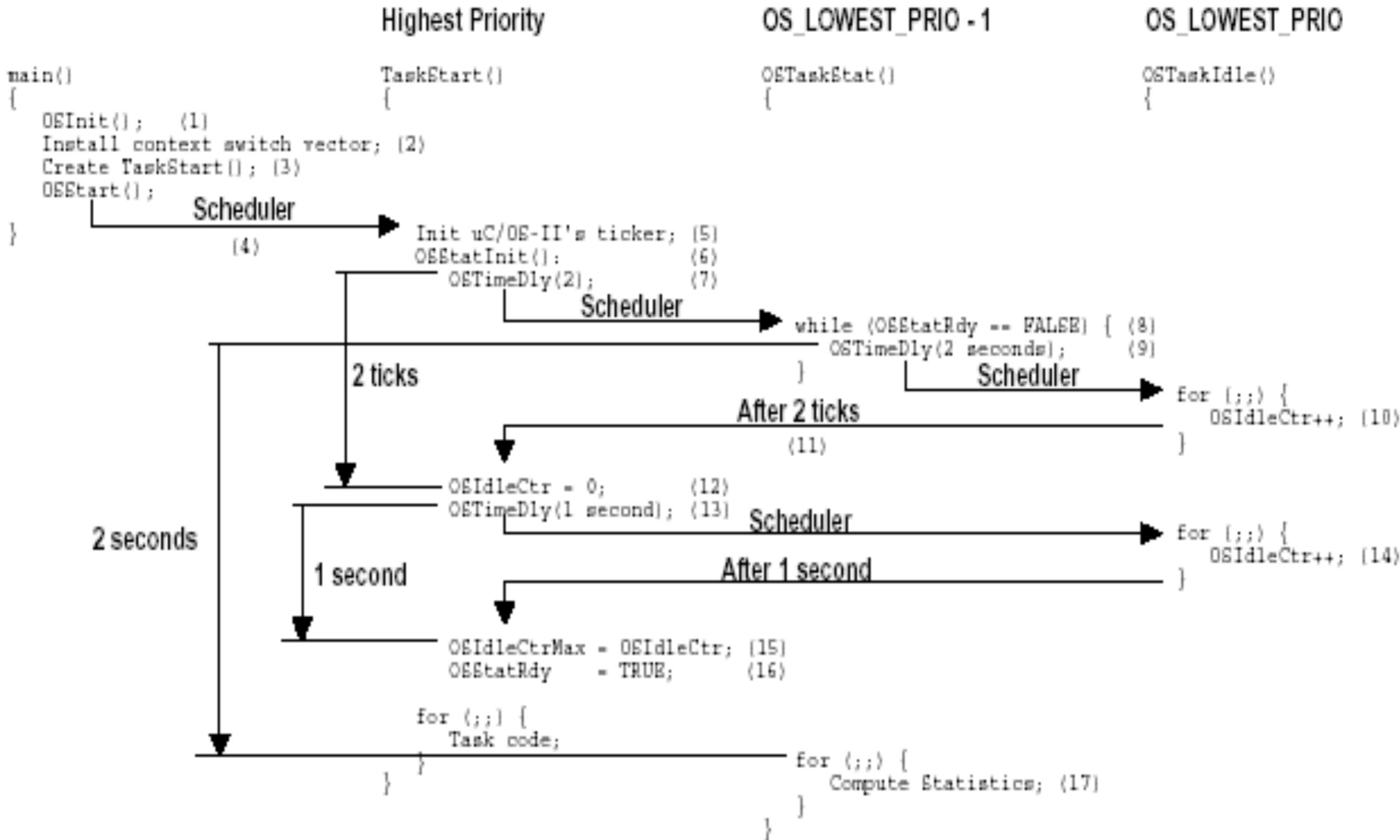
```
void OSSStart (void)
{
    INT8U Y;
    INT8U X;
    if (OSRunning == FALSE) {
        y = OSUnMapTbl[OSRdyGrp];
        x = OSUnMapTbl[OSRdyTbl[y]];
        OSPrioHighRdy = (INT8U)((Y<<3) + X);
        OSPrioCur = OSPrioHighRdy;
        OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
        OSTCBCur = OSTCBHighRdy;
        OSSStartHighRdy();
    }
}
```

统计任务初始化函数

OSStatInit (void)

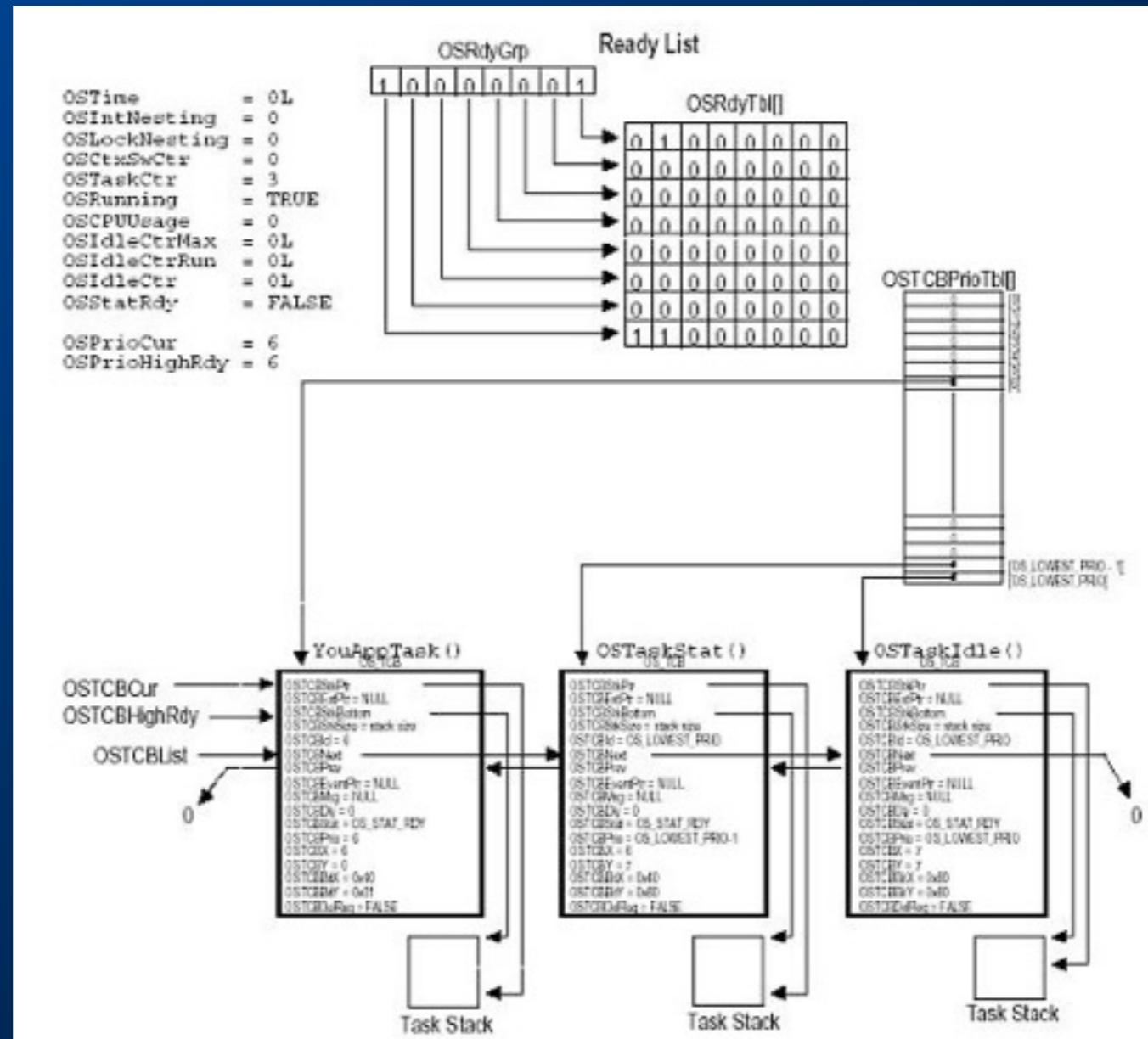
```
void OSStatInit (void)
{
    OSTimeDly(2);
    OS_ENTER_CRITICAL();
    OSIdleCtr = 0L;
    OS_EXIT_CRITICAL();
    OSTimeDly(OS_TICKS_PER_SEC);
    OS_ENTER_CRITICAL();
    OSIdleCtrMax = OSIdleCtr;
    OSStatRdy = TRUE;
    OS_EXIT_CRITICAL();
}
```

统计任务初始化



系统启动后的状态

假设用户创建的任务优先级为6



实时操作系统μC/OS-II

- μC/OS-II概述
- 任务管理
- 中断和时间管理
- 任务之间的通信与同步
- 存储管理

任务间通信与同步

- 任务间通信的管理：事件控制块ECB
- 同步与互斥
 - 临界区 (Critical Sections)
 - 信号量 (Semaphores)
- 任务间通信
 - 邮箱 (Message Mailboxes)
 - 消息队列 (Message Queues)

事件控制块ECB

- 所有的通信信号都被看成是事件(event), μC/OS-II通过事件控制块(ECB)来管理每一个具体事件。

ECB数据结构

```
typedef struct {  
    void *OSEventPtr; /*指向消息或消息队列的指针*/  
    INT8U OSEventTbl[OS_EVENT_TBL_SIZE];//等待任务列表  
    INT16U OSEventCnt; /*计数器（当事件是信号量时） */  
    INT8U OSEventType; /*事件类型：信号量、邮箱等*/  
    INT8U OSEventGrp; /*等待任务组*/  
} OS_EVENT;
```

与TCB类似的结构，使用两个链表，空闲链表与使用链表

事件控制块ECB数据结构



任务和ISR之间的通信方式

- 一个任务或ISR可以通过事件控制块ECB（信号量、邮箱或消息队列）向另外的任务发信号；
- 一个任务还可以等待另一个任务或中断服务子程序给它发送信号。对于处于等待状态的任务，还可以给它指定一个最长等待时间；
- 多个任务可以同时等待同一个事件的发生。当该事件发生后，在所有等待该事件的任务中，优先级最高的任务得到了该事件并进入就绪状态，准备执行。

等待任务列表

- 每个正在等待某个事件的任务被加入到该事件的ECB的等待任务列表中，该列表包含两个变量OSEventGrp和OSEventTbl[]。
- 在OSEventGrp中，任务按优先级分组，8个任务为一组，共8组，分别对应OSEventGrp 当中的8位。当某组中有任务处于等待该事件的状态时，对应的位就被置位。同时，OSEventTbl[]中的相应位也被置位。

OSEventGrp

7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---

任务的优先级

0	0	Y	Y	Y	X	X	X
---	---	---	---	---	---	---	---

OSEventTbl[]中相应位的位置

**OSEventGrp 中相应位的位置及
OSEventTbl[]中的数组下标**

OSEventTbl [OS_LOWEST_PRIO / 8+1]

最高优先级任务

[0]	7	6	5	4	3	2	1	0
[1]	15	14	13	12	11	10	9	8
[2]	23	22	21	20	19	18	17	16
[3]	31	30	29	28	27	26	25	24
[4]	39	38	37	36	35	34	33	32
[5]	47	46	45	44	43	42	41	40
[6]	55	54	53	52	51	50	49	48
[7]	63	62	61	60	59	58	57	56

正在等待该事件的任务的优先级

最低优先级任务(即空闲任务, 不可能处于等待状态)

使任务进入/脱离等待状态

- 将一个任务插入到事件的等待任务列表中

```
pevent->OSEventGrp          |= OSMapTbl[prio >> 3];  
pevent->OSEventTbl[prio >> 3] |= OSMapTbl[prio & 0x07];
```

- 从等待任务列表中删除一个任务

```
if ((pevent->OSEventTbl[prio >> 3] &= ~OSMapTbl[prio & 0x07]) == 0) {  
    pevent->OSEventGrp &= ~OSMapTbl[prio >> 3];  
}
```

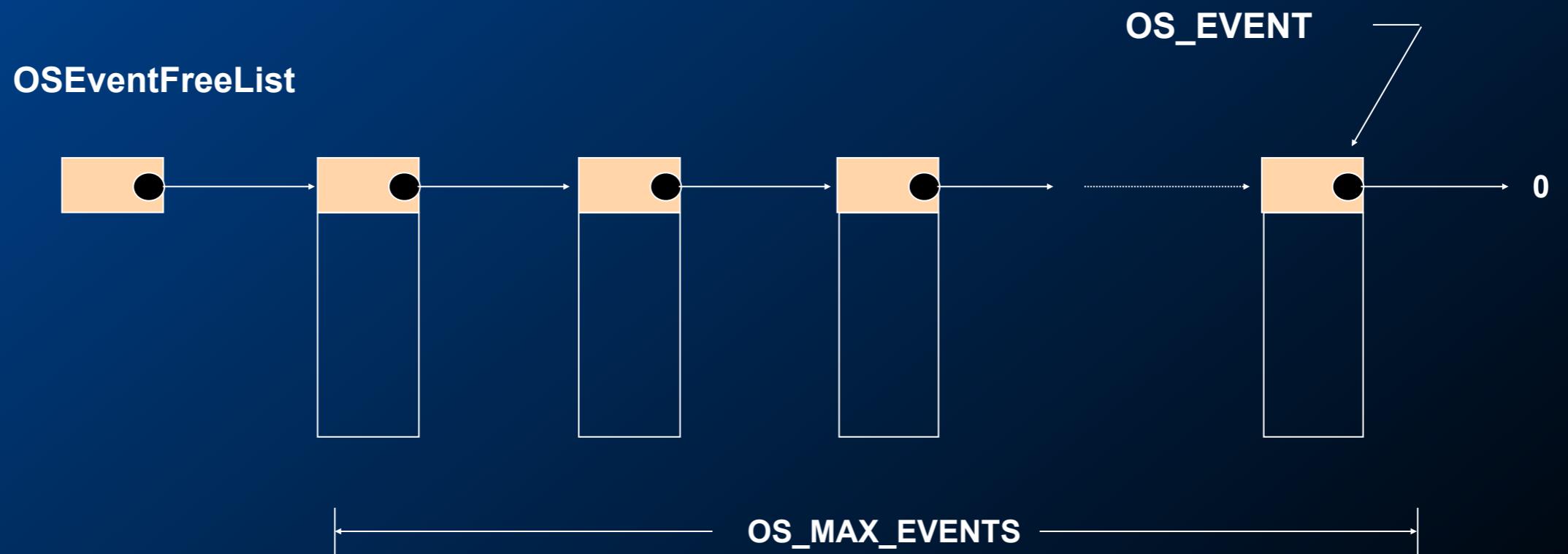
在等待事件的任务列表中查找优先级最高的任务

- 在等待任务列表中查找最高优先级的任务

```
y      = OSUnMapTbl[pevent->OSEventGrp];  
  
x      = OSUnMapTbl[pevent->OSEventTbl[y]];  
  
prio = (y << 3) + x;
```

空闲ECB的管理

- ECB的总数由用户所需要的信号量、邮箱和消息队列的总数决定，由OS_CFG.H中的#define OS_MAX_EVENTS定义。
- 在调用OSInit()初始化系统时，所有的ECB被链接成一个单向链表——空闲事件控制块链表；
- 每当建立一个信号量、邮箱或消息队列时，就从该链表中取出一个空闲事件控制块，并对它进行初始化。



ECB的基本操作

- OSEventWaitListInit()
 - 初始化一个事件控制块。当创建一个信号量、邮箱或消息队列时，相应的创建函数会调用本函数对ECB的内容进行初始化，将OSEventGrp和OSEventTbl[]数组清零；
 - OSEventWaitListInit (OS_EVENT *pevent);
 - pevent: 指向需要初始化的事件控制块的指针。
- OSEventTaskRdy()
 - 使一个任务进入就绪态。当一个事件发生时，需要将其等待任务列表中的最高优先级任务置为就绪态；
 - OSEventTaskRdy (OS_EVENT *pevent,
void *msg, INT8U msk);
 - msg: 指向消息的指针；msk: 用于设置TCB的状态。

ECB的基本操作（续）

- OSEventTaskWait()
 - 使一个任务进入等待状态。当某个任务要等待一个事件的发生时，需要调用本函数将该任务从就绪任务表中删除，并放到相应事件的等待任务表中；
 - OSEventTaskWait (OS_EVENT *pevent);

同步与互斥

- 为了实现资源共享，一个操作系统必须提供临界区操作的功能；
- μC/OS采用关闭/打开中断的方式来处理临界区代码，从而避免竞争条件，实现任务间的互斥；
- μC/OS定义两个宏(macros)来开关中断，即：
OS_ENTER_CRITICAL()和OS_EXIT_CRITICAL();
- 这两个宏的定义取决于所用的微处理器，每种微处理器都有自己的OS_CPU.H文件。

任务1

```
...  
OS_ENTER_CRITICAL();  
任务1的临界区代码;  
OS_EXIT_CRITICAL();  
...
```

任务2

```
...  
OS_ENTER_CRITICAL();  
任务2的临界区代码;  
OS_EXIT_CRITICAL();  
...
```

临界资源

μ C/OS-II中开关中断的方法

- 当处理临界段代码时，需要关中断，处理完毕后，再开中断；
- 关中断时间是实时内核最重要的指标之一；
- 在实际应用中，关中断的时间很大程度上取决于微处理器的结构和编译器生成的代码质量；

μ C/OS-II中采用了3种开关中断的方法

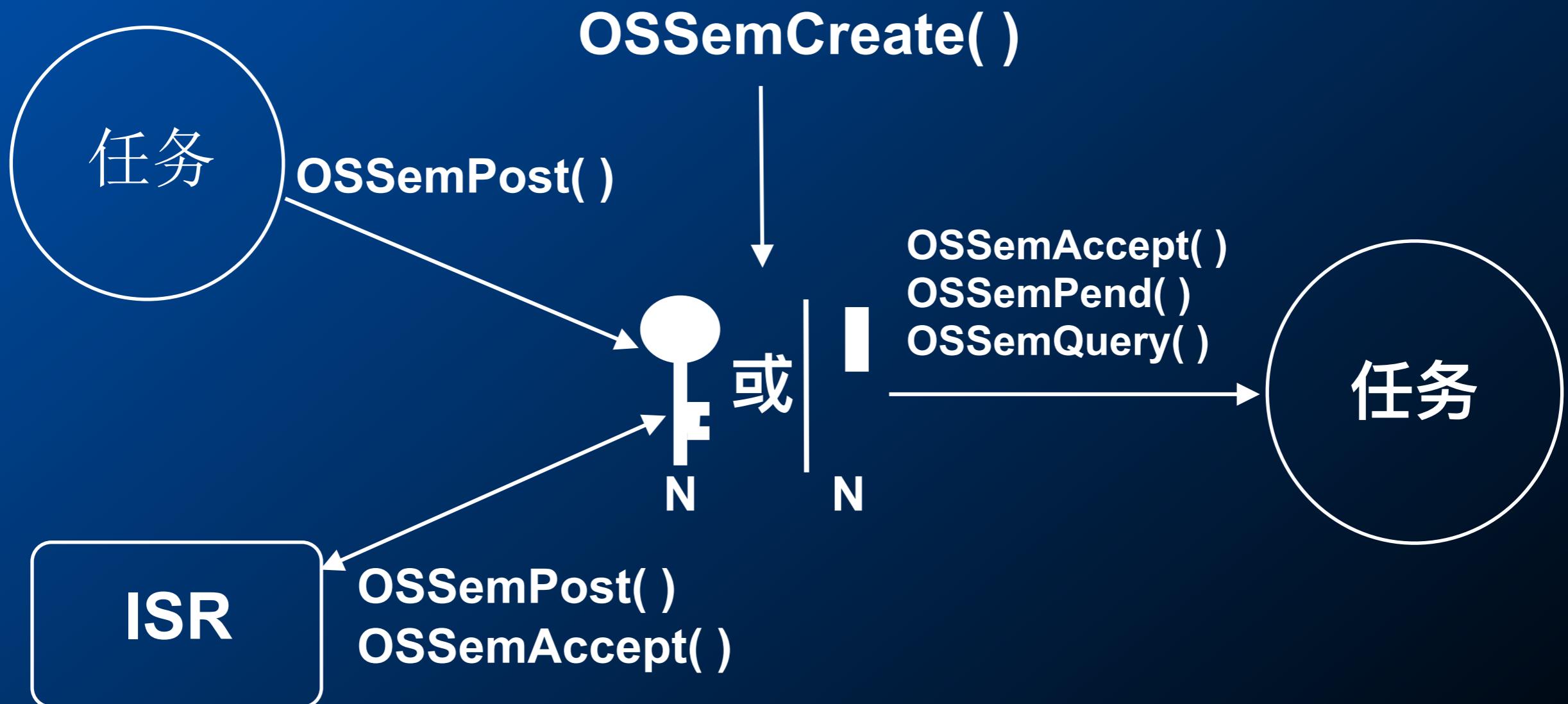
- OS_CRITICAL_METHOD==1
 - 用处理器指令关中断，执行OS_ENTER_CRITICAL(), 开中断执行OS_EXIT_CRITICAL();
- OS_CRITICAL_METHOD==2
 - 实现OS_ENTER_CRITICAL()时，先在堆栈中保存中断的开/关状态，然后再关中断；实现OS_EXIT_CRITICAL()时，从堆栈中弹出原来中断的开/关状态；
- OS_CRITICAL_METHOD==3
 - 把当前处理器的状态字保存在局部变量中（如OS_CPU_SR），关中断时保存，开中断时恢复

信号量

- 信号量在多任务系统中的功能
 - 实现对共享资源的互斥访问（包括单个共享资源或多个相同的资源）；
 - 实现任务之间的行为同步；
- 必须在OS_CFG.H中将OS_SEM_EN开关常量置为1，这样μC/OS才能支持信号量。

- uC/OS中信号量由两部分组成：信号量的计数值（16位无符号整数）和等待该信号量的任务所组成的等待任务表；
- 信号量系统服务
 - OSSemCreate()
 - OSSemPend(), OSSemPost()
 - OSSemAccept(), OSSemQuery()

任务、ISR和信号量的关系



创建一个信号量

- OSSemCreate()
 - 创建一个信号量，并对信号量的初始计数值赋值，该初始值为0到65,535之间的一个数；
 - OS_EVENT *OSSemCreate(INT16U cnt);
 - cnt：信号量的初始值。
- 执行步骤
 - 从空闲事件控制块链表中得到一个ECB；
 - 初始化ECB，包括设置信号量的初始值、把等待任务列表清零、设置ECB的事件类型等；
 - 返回一个指向该事件控制块的指针。

等待一个信号量

- OSSemPend()
 - 等待一个信号量，即操作系统中的P操作，将信号量的值减1；
 - OSSemPend (OS_EVENT *pevent,
INT16U timeout, INT8U *err);
- 执行步骤
 - 如果信号量的计数值大于0，将它减1并返回；
 - 如果信号量的值等于0，则调用本函数的任务将被阻塞起来，等待另一个任务把它唤醒；
 - 调用OSSched()函数，调度下一个最高优先级的任务运行。

发送一个信号量

- OSSemPost()
 - 发送一个信号量，即操作系统中的V操作，将信号量的值加1；
 - OSSemPost (OS_EVENT *pevent);
- 执行步骤
 - 检查是否有任务在等待该信号量，如果没有，将信号量的计数值加1并返回；
 - 如果有，将优先级最高的任务从等待任务列表中删除，并使它进入就绪状态；
 - 调用OSSched(), 判断是否需要进行任务切换。

无等待地请求一个信号量

- OSSemAccept()
 - 当一个任务请求一个信号量时，如果该信号量暂时无效，则让该任务简单地返回，而不是进入等待状态；
 - INT16U OSSemAccept(OS_EVENT *pevent);
- 执行步骤
 - 如果该信号量的计数值大于0，则将它减1，然后将信号量的原有值返回；
 - 如果该信号量的值等于0，直接返回该值(0)。

查询一个信号量的当前状态

- OSSemQuery()
 - 查询一个信号量的当前状态；
 - INT8U OSSemQuery(OS_EVENT *pevent,
OS_SEM_DATA *pdata);
 - 将指向信号量对应事件控制块的指针pevent所指向的ECB的内容拷贝到指向用于记录信号量信息的数据结构OS_SEM_DATA数据结构的指针pdata所指向的缓冲区当中。

任务间通信

- 低级通信
 - 只能传递状态和整数值等控制信息，传送的信息量小；
 - 例如：信号量
- 高级通信
 - 能够传送任意数量的数据；
 - 例如：共享内存、邮箱、消息队列

共享内存

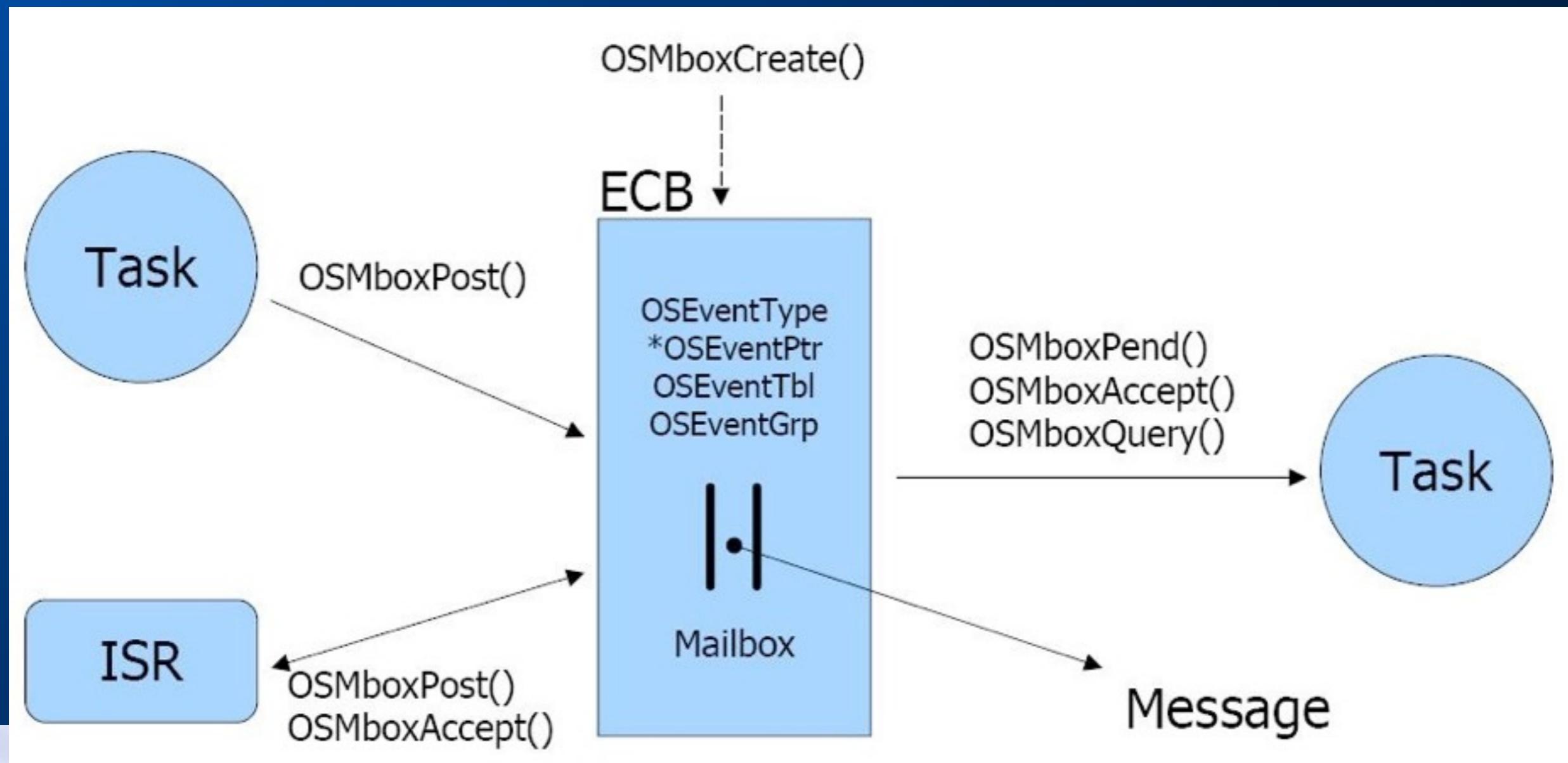
- 在μC/OS-II中如何实现共享内存?
 - 内存地址空间只有一个，为所有的任务所共享！
 - 为了避免竞争状态，需要使用信号量来实现互斥访问。

消息邮箱

- 邮箱（MailBox）：一个任务或ISR可以通过邮箱向另一个任务发送一个指针型的变量，该指针指向一个包含了特定“消息”（message）的数据结构；
- 必须在OS_CFG.H中将OS_MBOX_EN开关常量置为1，这样μC/OS才能支持邮箱。

- 一个邮箱可能处于两种状态：
 - 满的状态： 邮箱包含一个非空指针型变量；
 - 空的状态： 邮箱的内容为空指针NULL；
- 邮箱的系统服务
 - OSMboxCreate()
 - OSMboxPost()
 - OSMboxPend()
 - OSMboxAccept()
 - OSMboxQuery()

任务、ISR和消息邮箱的关系



邮箱的系统服务 (I)

- OSMboxCreate(): 创建一个邮箱
 - 在创建邮箱时，须分配一个ECB，并使用其中的字段OSEventPtr指针来存放消息的地址；
 - OS_EVENT *OSMboxCreate(void *msg);
 - msg: 指针的初始值，一般情形下为NULL。
- OSMboxPend(): 等待一个邮箱中的消息
 - 若邮箱为满，将其内容（某消息的地址）返回；若邮箱为空，当前任务将被阻塞，直到邮箱中有了消息或等待超时；
 - OSMboxPend (OS_EVENT *pevent,
INT16U timeout, INT8U *err);

邮箱的系统服务 (2)

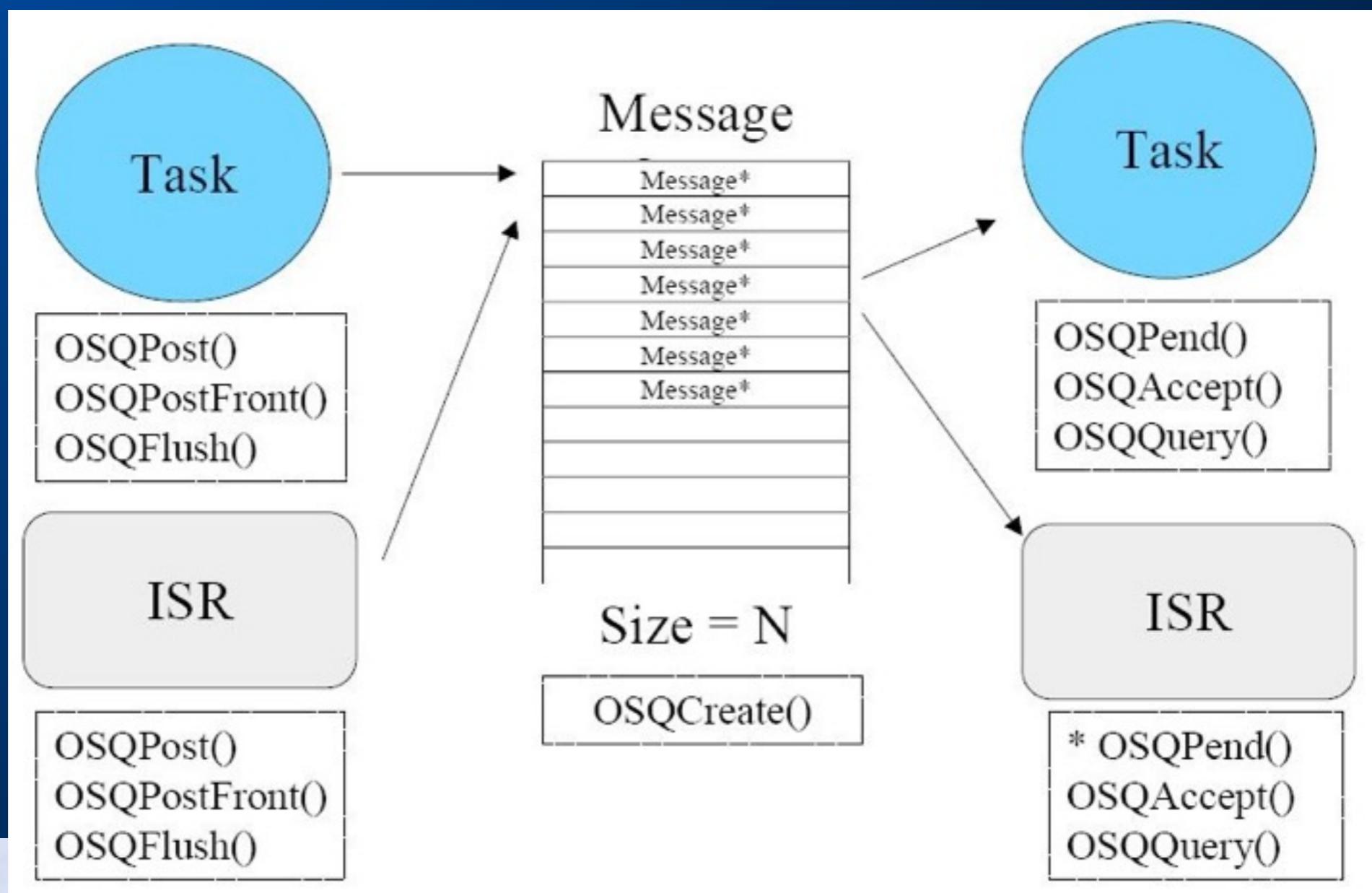
- OSMboxPost(): 发送一个消息到邮箱中
 - 如果有任务在等待该消息，将其中的最高优先级任务从等待列表中删除，变为就绪状态；
 - OSMboxPost(OS_EVENT *pevent, void *msg);
- OSMboxAccept(): 无等待地请求邮箱消息
 - 若邮箱为满，返回它的当前内容；若邮箱为空，返回空指针；
 - OSMboxAccept (OS_EVENT *pevent);
- OSMboxQuery(): 查询一个邮箱的状态
 - OSMboxQuery (OS_EVENT *pevent, OS_MBOX_DATA *pdata);

消息队列

- 消息队列（Message Queue）：消息队列可以使一个任务或ISR向另一个任务发送多个以指针方式定义的变量；
- 为了使μC/OS能够支持消息队列，必须在OS_CFG.H中将OS_Q_EN开关常量置为1，并且通过常量OS_MAX_QS来决定系统支持的最多消息队列数。

- 一个消息队列可以容纳多个不同的消息，因此可把它看作是由多个邮箱组成的数组，只是它们共用一个等待任务列表：
- 消息队列的系统服务
 - OSQCreate()
 - OSQPend()、OSQAccept()
 - OSQPost()、OSQPostFront()
 - OSQFlush()
 - OSQQuery()

消息队列的体系结构



回忆一下ECB数据结构

- 在实现消息队列时，哪些字段可以用？

ECB数据结构

```
typedef struct {

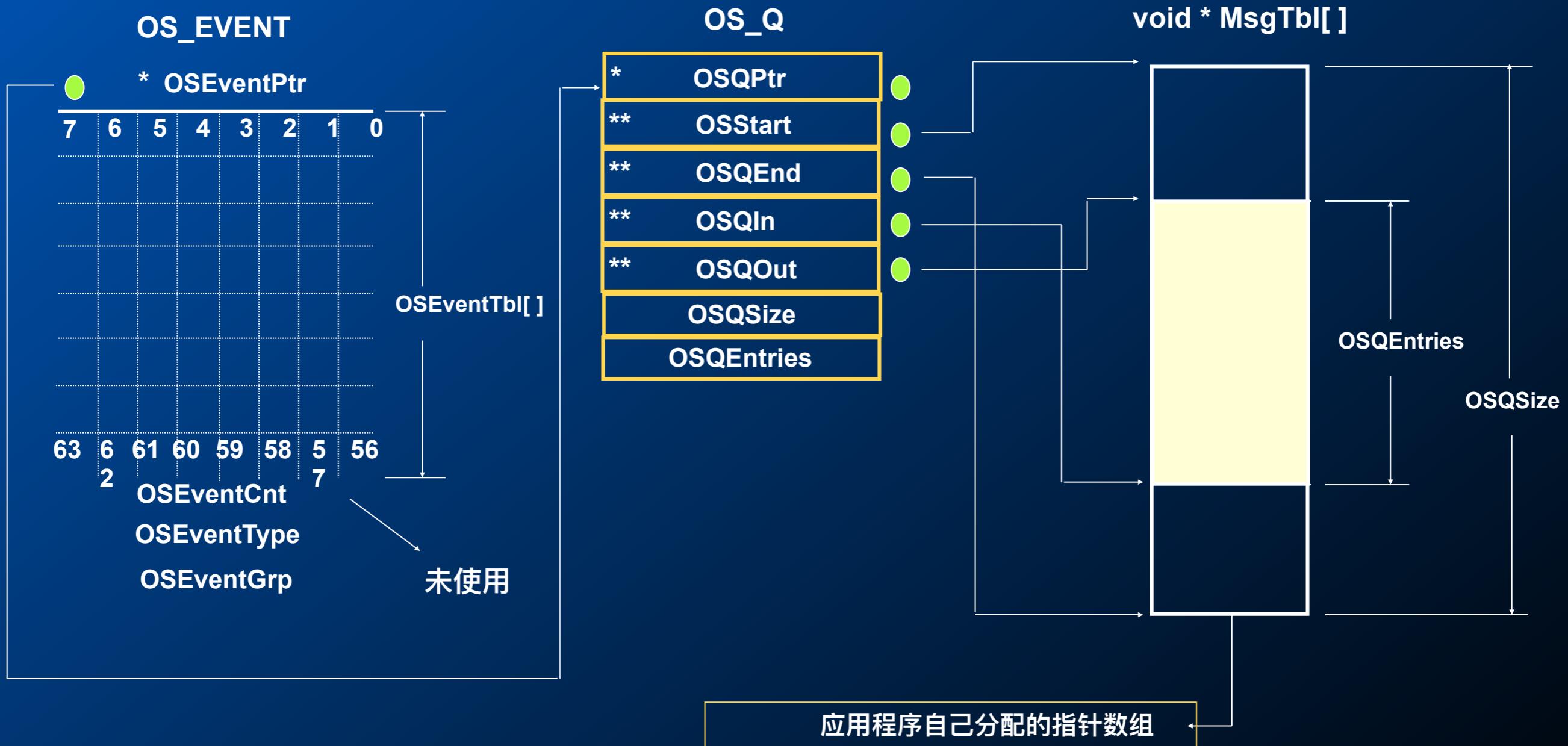
    void *OSEventPtr; /*指向消息或消息队列的指针*/
    INT8U OSEventTbl[OS_EVENT_TBL_SIZE]; //等待任务列表
    INT16U OSEventCnt; /*计数器（当事件是信号量时）*/
    INT8U OSEventType; /*事件类型：信号量、邮箱等*/
    INT8U OSEventGrp; /*等待任务组*/
} OS_EVENT;
```

队列控制块

- 队列控制块数据结构

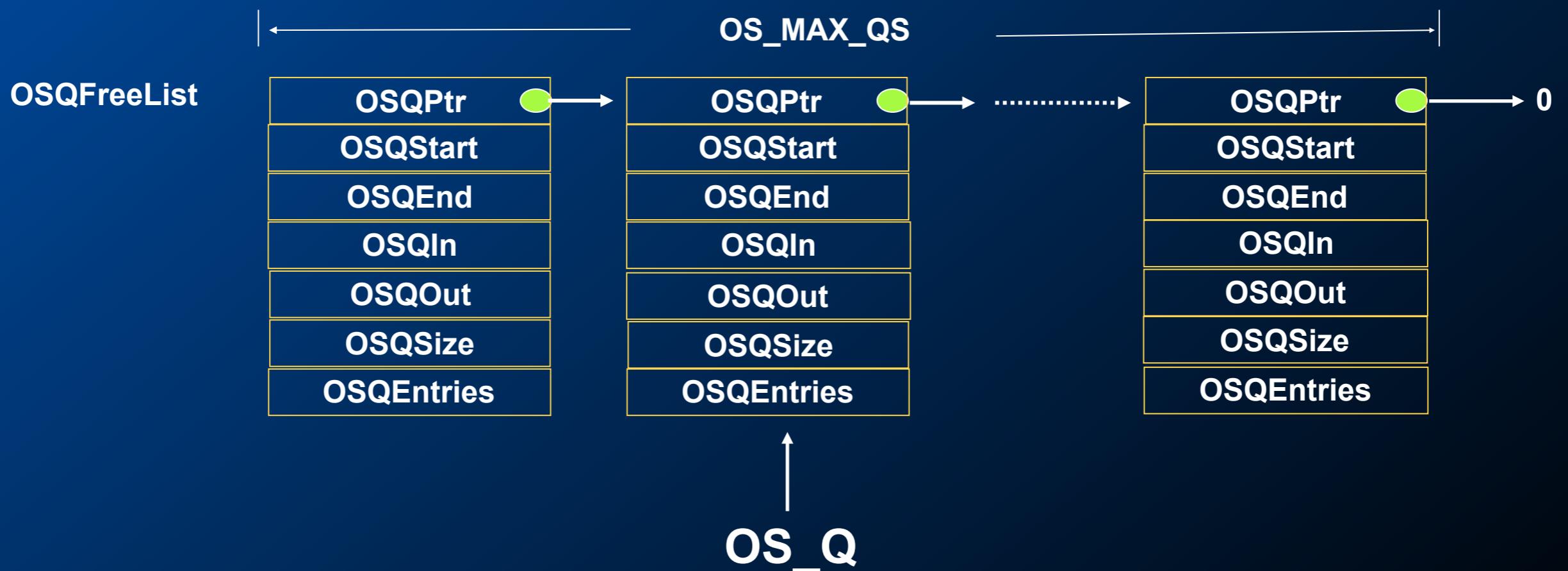
```
typedef struct os_q {  
    struct os_q *OSQPtr;//空闲队列控制块指针  
    void **OSQStart; //指向消息队列的起始地址  
    void **OSQEnd; //指向消息队列的结束地址  
    void **OSQIn; //指向消息队列中下一个插入消息的位置  
    void **OSQOut;//指向消息队列中下一个取出消息的位置  
    INT16U OSQSize; //消息队列中总的单元数  
    INT16U OSQEntries; //消息队列中当前的消息数量  
} OS_EVENT;
```

消息队列的数据结构

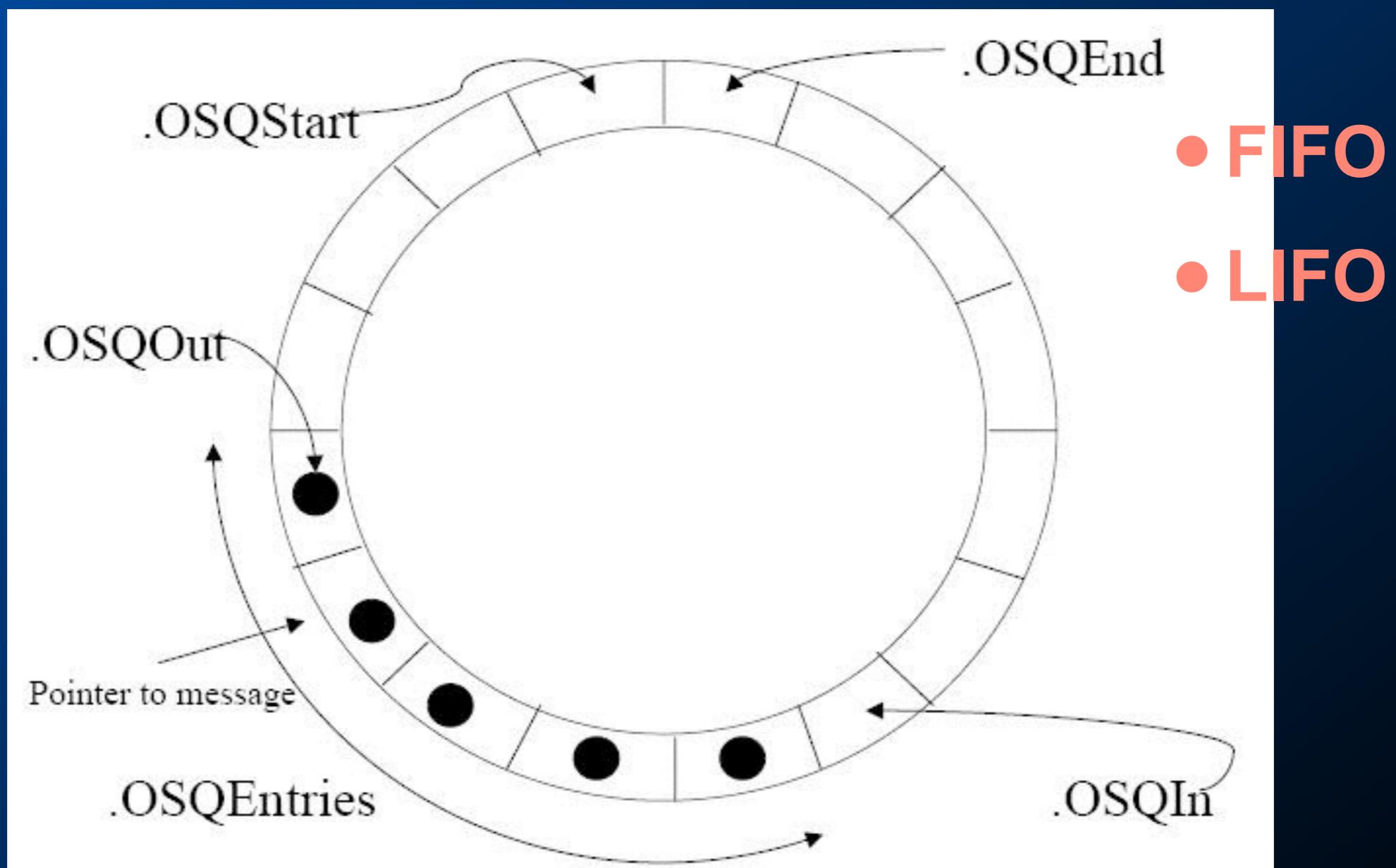


空闲队列控制块的管理

- 每一个消息队列都要用到一个队列控制块。在μC/OS中，队列控制块的总数由OS_CFG.H中的常量OS_MAX_QS定义。
- 在系统初始化时，所有的队列控制块被链接成一个单向链表——空闲队列控制块链表OSQFreeList。



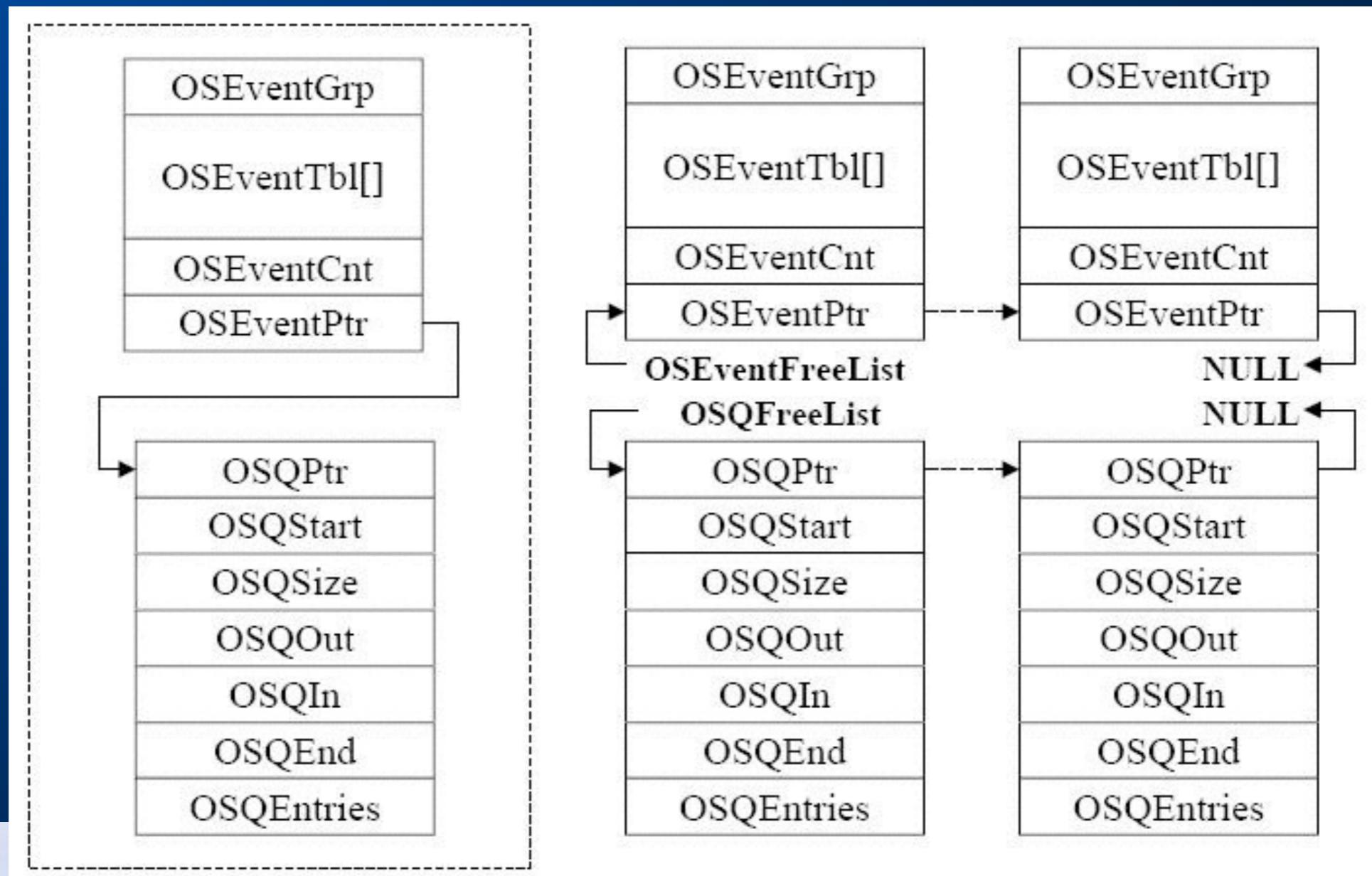
消息缓冲区



创建一个消息队列

- OSQCreate()
 - OS_EVENT *OSQCreate (void **start,
INT16U size);
 - start: 指针数组，用来存放各个消息的地址
 - size: 数组的大小（即消息队列的元素个数）
- 执行步骤
 - 从空闲事件控制块链表中取得一个ECB；
 - 从空闲队列控制块列表中取出一个队列控制块，并对其进行初始化；
 - 初始化ECB的内容（事件类型、等待任务列表），并将OSEventPtr指针指向队列控制块。

队列控制块与事件控制块



请求消息队列中的消息

- OSQPend(): 等待一个消息队列中的消息
 - `void *OSQPend (OS_EVENT *pevent,
INT16U timeout, INT8U *err);`
 - 如果消息队列中有至少一条消息，返回消息的地址；
 - 如果没有消息，相应任务进入等待状态。
- OSQAccept(): 无等待地请求消息队列中的消息
 - `void *OSQAccept(OS_EVENT *pevent);`
 - 如果消息队列中有消息，返回消息的地址；
 - 如果消息队列中没有消息，返回NULL。

向消息队列发送一个消息

- OSQPost(): 以FIFO方式向消息队列发送一个消息
 - INT8U OSQPost (OS_EVENT *pevent,
void *msg);
 - 如果有任务在等待该消息队列，唤醒其中优先级最高的任务，并重新调度；
 - 如果没有任务在等待该消息队列，而且此时消息队列未满，则以FIFO方式插入这个消息。
- OSQPostFront(): 以LIFO方式向消息队列发送一个消息
 - INT8U OSQPostFront(OS_EVENT *pevent, void *msg);

清空操作与查询操作

- OSQFlush(): 清空一个消息队列
 - INT8U OSQFlush (OS_EVENT *pevent);
 - 删除一个消息队列中的所有消息；
- OSQuery(): 查询一个消息队列的状态
 - INT8U OSQuery (OS_EVENT *pevent, OS_Q_DATA *pdata);

实时操作系统μC/OS-II

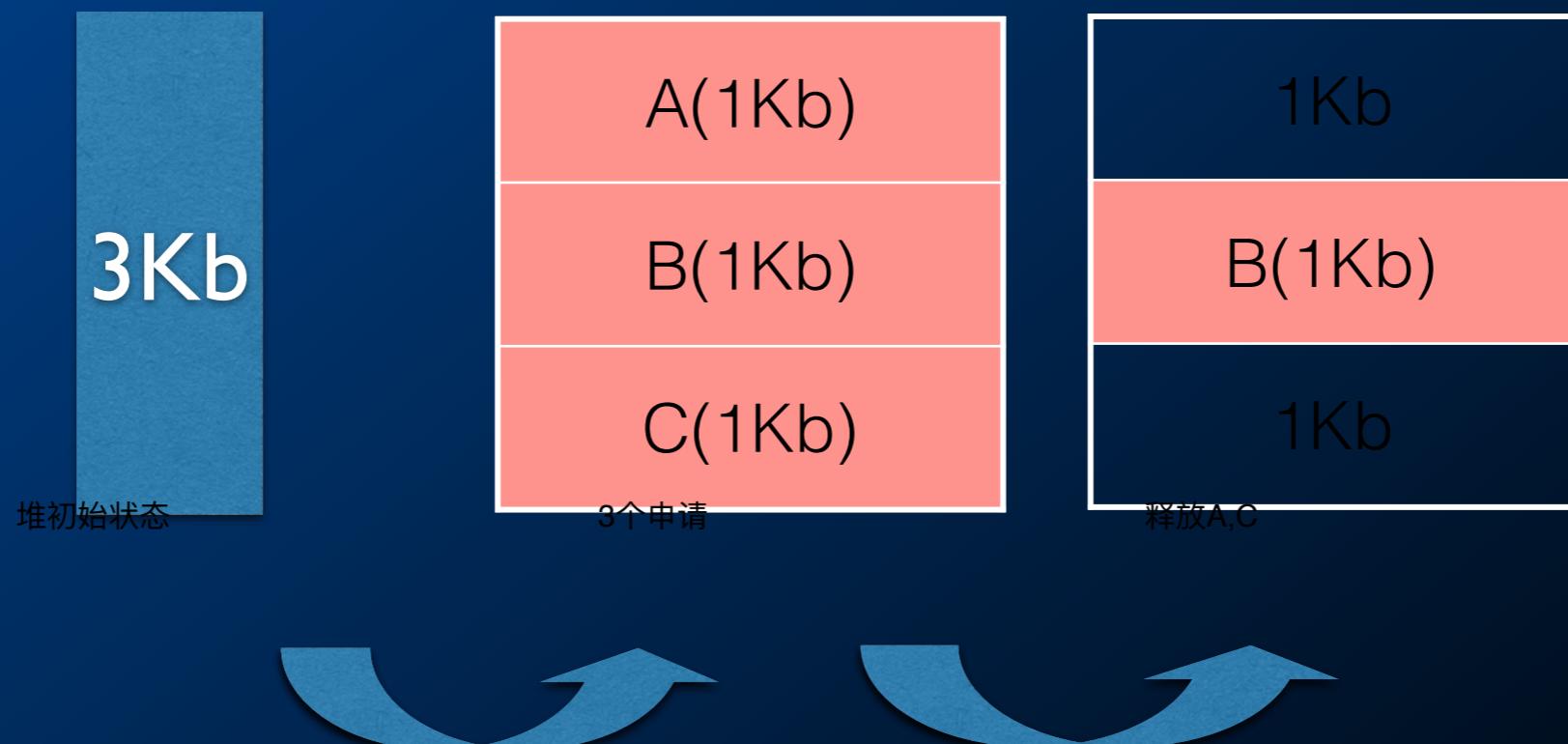
- μC/OS-II概述
- 任务管理
- 中断和时间管理
- 任务之间的通信与同步
- 存储管理

概述

- μC/OS中是实模式存储管理
 - 不划分内核空间和用户空间，整个系统只有一个地址空间，即物理内存空间，应用程序和内核程序都能直接对所有的内存单元进行访问；
 - 系统中的“任务”，实际上都是线程——只有运行上下文和栈是独享的，其他资源都是共享的。
- 内存布局
 - 代码段(text)、数据段(data)、bss段、堆空间、栈空间；
 - 内存管理，管的是谁？

malloc/free?

- 在ANSI C中可以用malloc()和free()两个函数动态地分配内存和释放内存。在嵌入式实时操作系统中，容易产生碎片。
- 由于内存管理算法的原因，malloc()和free()函数执行时间是不确定的。 μ C/OS-II 对malloc()和free()函数进行了改进，使得它们可以分配和释放固定大小的内存块。这样一来，malloc()和free()函数的执行时间也是固定的了



μ C/OS中的存储管理

- μ C/OS采用的是固定分区的存储管理方法
 - μ C/OS把连续的大块内存按分区来管理，每个分区包含有整数个大小相同的块；
 - 在一个系统中可以有多个内存分区，这样，用户的应用程序就可以从不同的内存分区中得到不同大小的内存块。但是，特定的内存块在释放时必须重新放回它以前所属于的内存分区；
 - 采用这样的内存管理算法，上面的内存碎片问题就得到了解决。

内存分区示意图



内存控制块

- 为了便于管理，在μC/OS中使用内存控制块MCB（Memory Control Block）来跟踪每一个内存分区，系统中的每个内存分区都有它自己的 MCB。

```
typedef struct {  
    void *OSMemAddr; /*分区起始地址*/  
    void *OSMemFreeList; //下一个空闲内存块  
    INT32U OSMemBlkSize; /*内存块的大小*/  
    INT32U OSMemNBlks; /*内存块数量*/  
    INT32U OSMemNFree; /*空闲内存块数量*/  
} OS_MEM;
```

内存管理初始化

- 如果要在μC/OS-II中使用内存管理，需要在OS_CFG.H文件中将开关量OS_MEM_EN设置为1。这样μC/OS-II 在系统初始化OSInit()时就会调用OSMemInit()，对内存管理器进行初始化，建立空闲的内存控制块链表。

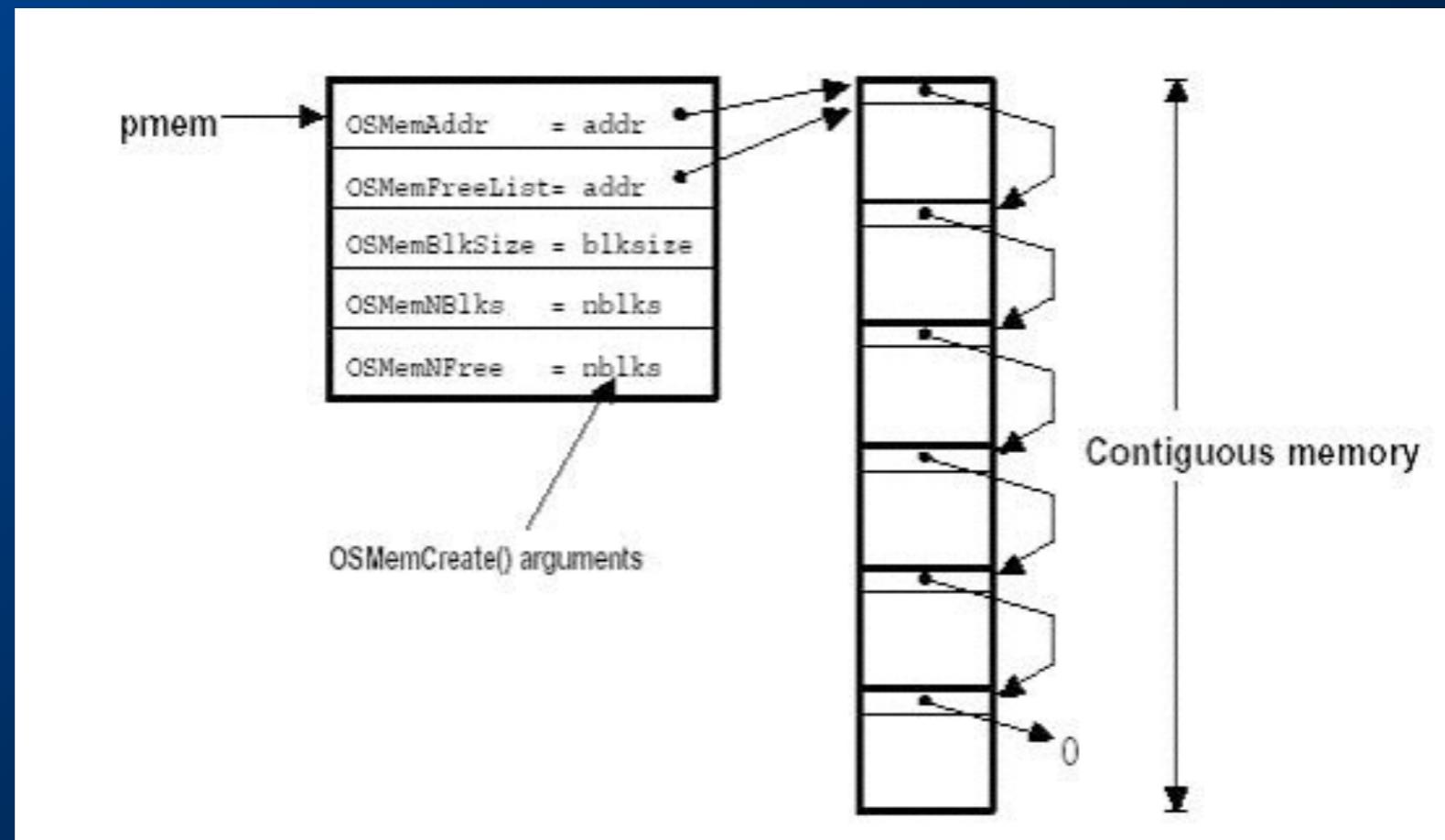


创建一个内存分区

- OSMemCreate()
 - OS_MEMORY *OSMemCreate (
void *addr, // 内存分区的起始地址
INT32U nblks, // 分区内的内存块数
INT32U blksize, // 每个内存块的字节数
INT8U *err); // 指向错误码的指针
- 例子
 - OS_MEMORY *CommTxBuf;
 - INT8U CommTxPart[100][32];
 - CommTxBuf = OSMemCreate(CommTxPart,
100, 32, &err);

- OSMemCreate()

- 从系统的空闲内存控制块中取得一个MCB；
- 将这个内存分区中的所有内存块链接成一个单向链表；
- 在对应的MCB中填写相应的信息。



分配一个内存块

- `void *OSMemGet(OS_MEM *pmem, INT8U *err);`
- 功能：从已经建立的内存分区中申请一个内存块。该函数的唯一参数是指向特定内存分区的指针。如果没有空闲的内存块可用，返回NULL指针。
- 应用程序必须知道内存块的大小，并且在使用时不能超过该容量。

释放一个内存块

- INT8U OSMemPut(OS_MEMORY *pmem, void *pblk);
- 功能：将一个内存块释放并放回到相应的内存分区中。
- 注意：用户应用程序必须确认将内存块放回到了正确的内存分区中，因为OSMemPut()并不知道一个内存块是属于哪个内存分区的。

等待一个内存块

- 如果没有空闲的内存块，OSMemGet()立即返回NULL。能否在没有空闲内存块的时候让任务进入等待状态？
- μC/OS-II本身在内存管理上并不支持这项功能，如果需要的话，可以通过为特定内存分区增加信号量的方法，来实现此功能。
- 基本思路：当应用程序需要申请内存块时，首先要得到一个相应的信号量，然后才能调用OSMemGet()函数。

```
OS_EVENT *SemaphorePtr;
OS_MEM  *PartitionPtr;
INT8U Partition[100][32];
OS_STK TaskStk[1000];
void main(void)
{
    INT8U err;
    OSInit();
    ...
    SemaphorePtr = OSSemCreate(100);
    PartitionPtr = OSMemCreate(Partition, 100, 32, &err);
    OSTaskCreate(Task, (void *)0, &TaskStk[999], &err);
    OSSStart();
}
```

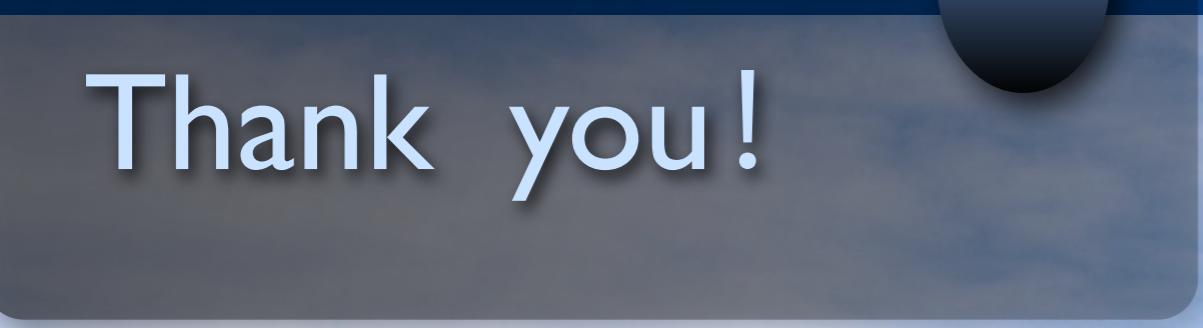
```
void Task (void *pdata)
{
    INT8U err;
    INT8U *pblock;

    for (;;) {
        OSSemPend(SemaphorePtr, 0, &err);

        pblock = OSMemGet(PartitionPtr, &err);
        /* 使用内存块 */

        ...
        OSMemPut(PartitionPtr, pblock);

        OSSemPost(SemaphorePtr);
    }
}
```



Thank you!

