uC/OS-III 的任务调度算法研究

1. 就绪列表

1.1 概述

准备好运行的任务被放到就绪列表中 ,如图 1.1。就绪列表是一个数组(OSRdyList[]) ，它一共有 OS_CFG_PRIO_MAX 条记录，记录的数据类型为 OS_RDY_LIST(见 OS.H)。就绪列表中的每条记录都包含了三个变量.Entries、.TailPtr、.HeadPtr。

.Entries 中该优先级的就绪任务数。当该优先级中没有任务就绪时， .Entries 就会被设置为 0。

.TailPtr 和.HeadPtr 用于该优先级就绪任务的建立双向列表。.HeadPtr 指向列表的头部，.TailPtr 指向列表的尾部。

表中的记录跟任务的优先级有关。例如，如果一个任务的优先级是 5，那么当它就绪时会被放入 OSRdyList[5] 中。

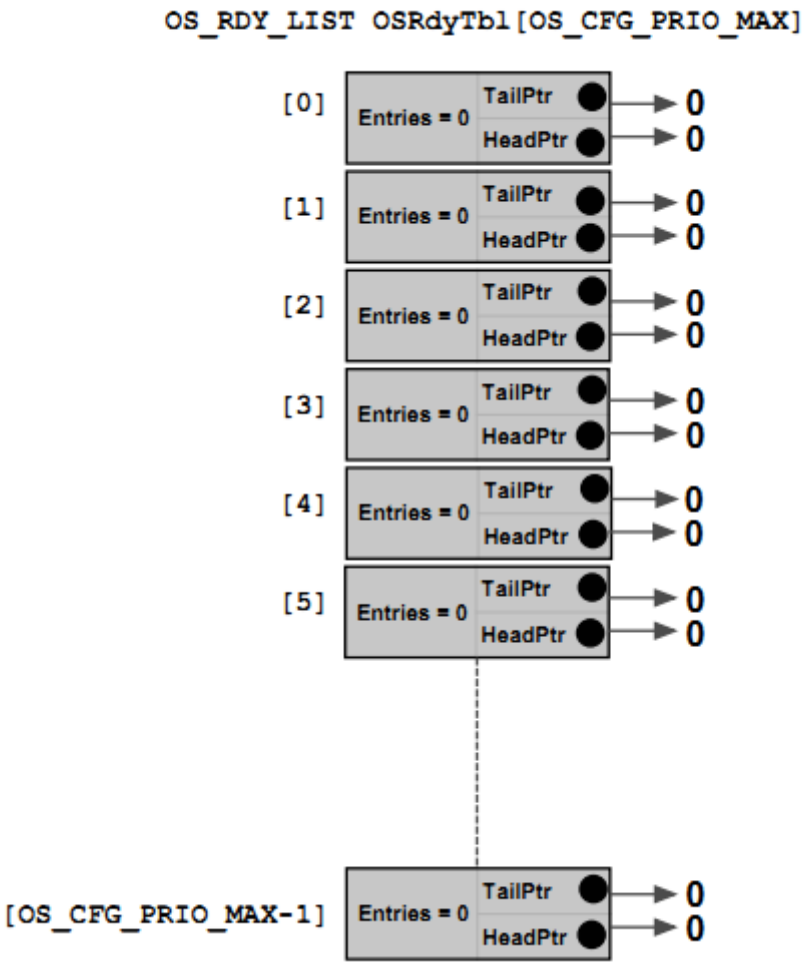

图 1.1：任务就绪表

1.2 调用 OSInit()后的就绪列表

有多少种优先级，就绪列表中就由多少条记录。每个记录中都有 3 个变量。Entries 为该记录中的任务数。.PrevPtr 和 .NextPtr 用于指向具有相同优先级到 TCB 组成的双向列表。对于空闲任务，这两个值为 NULL 。此时任务就绪表如图 1.2 所示。

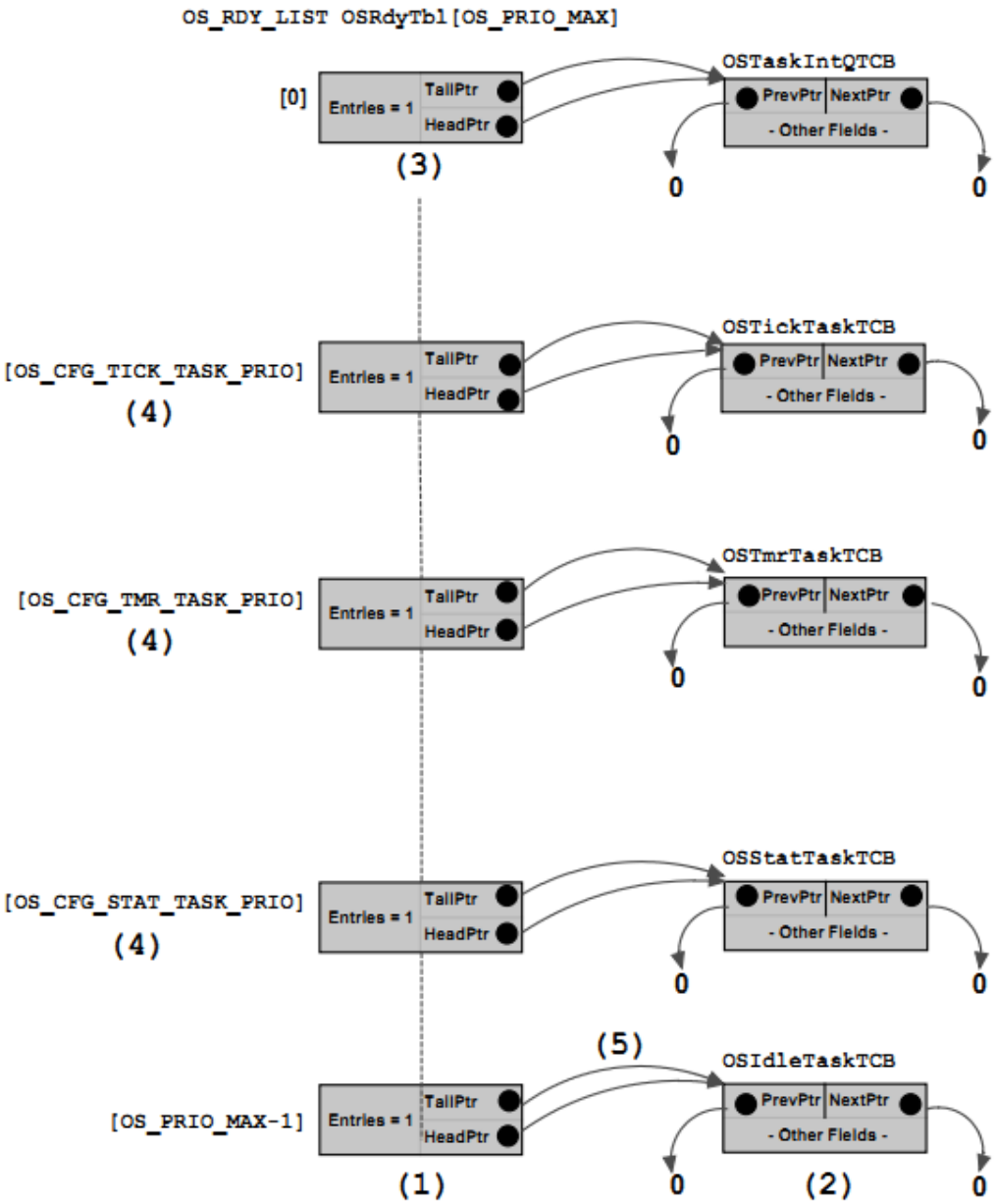

图 1.2：任务初始化后列表

1.3 添加任务到就绪列表

uC/OS-III 提供很多服务可以把任务添加到就绪列表中。最明显的服务是 OSTaskCreate()，它通常创建准备运行的任务并将任务放入就绪列表中。如图 1.3 所示，就绪列表中该优先级中已经有两个任务了。OSTaskCreate() 就会将这个任务插入到列表的末部。

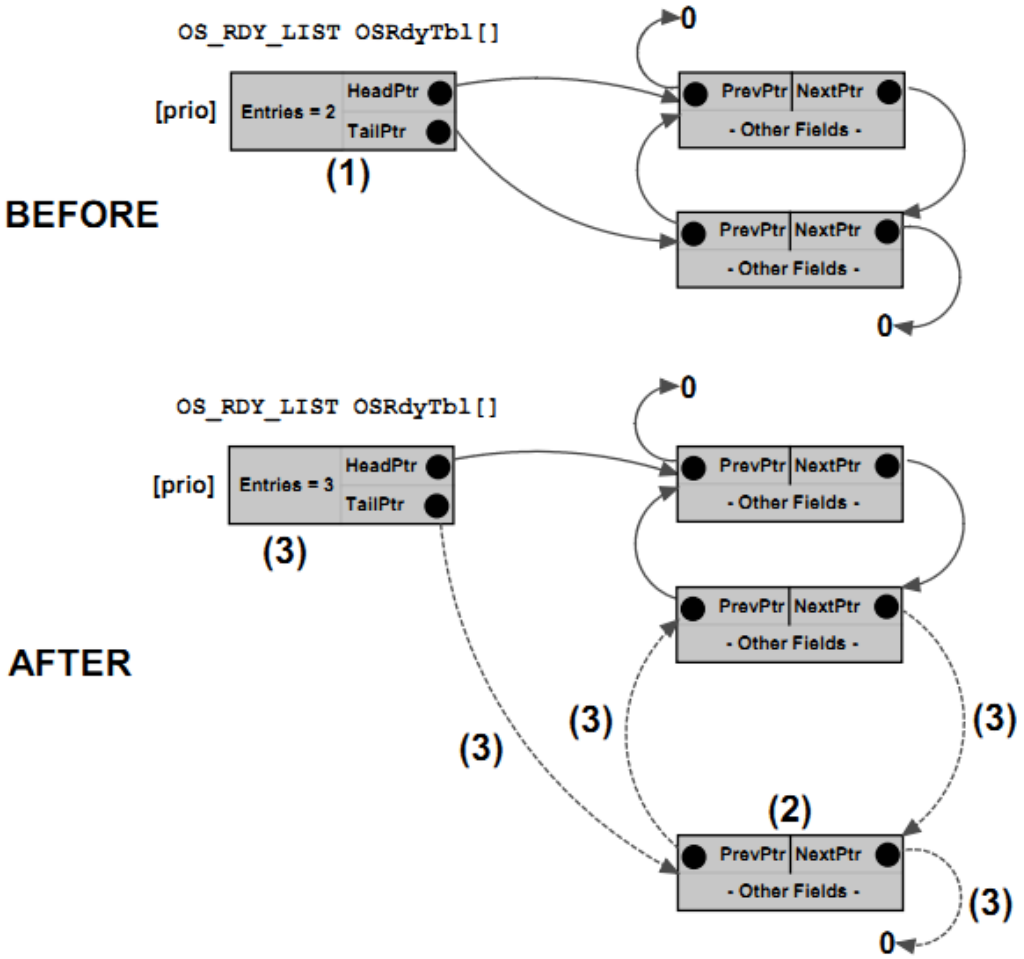

图 1.3：添加任务

2. 优先级

2.1 uC/OS-III 简介

uC/OS-III 是一个可扩展的，可固化的，抢占式的实时内核，它管理的任务个数不受限制。它是第三代内核，提供了现代实时内核所期望的所有功能包括资源管理、同步、内部任务交流等。uC/OS-III 也提供了很多特性是在其他实时内核中所没有的。比如能在运行时测量运行性能，直接得发送信号或消息给任务，任务能同时等待多个信号量和消息队列。

以下列出 uC/OS-III 相对于 uC/OS-II 来说最大个改变：

- 时间片轮转调度

  uC/OS-III 允许多个任务拥有相同的优先级。当多个相同优先级的任务就绪时，并且这个优先级是目前最高的。uC/OS-III 会分配用户定义的时间片给每个任务去运行。每个任务可以定义不同的时间片。当任务用不完时间片时可以让出 CPU 给另一个任务。

- 任务数无限制

  uC/OS-III 对任务数量无限制。实际上，任务的数量限制于处理器能提供的内存大小。每一个任务需要有自己的堆栈空间，uC/OS-III 在运行时监控任务堆栈的生长。

  uC/OS-III 对任务的大小无限制

- 优先级数无限制

uC/OS-III 对优先级的数量无限制。然而，配置 uC/OS-III 的优先级在 32 到 256 之间已经满足大多数的应用了。

2.2 uC/OS-III 的优先级调度

图 2-1 到 2-3 显示了优先级的位映像组。它的宽度取决于 CPU_DATA 的数据类型（见 CPU.H），它可以是 8 位 、16 位、32 位，根据处理器相应地设定。

uC/OS-III 支持多达 OS_CFG_PRIO_MAX 种不同的优先级（见 OS_CFG.H）。在 uC/OS-III 中，数值越小优先级越高。因此优先级 0 是优先级最高的。优先级 OS_CFG_PRIO_MAX-1 的优先级最低。uC/OS-III 将最低优先级唯一地分配给空闲任务，其它任务不允许被设置为这个优先级 。当任务准备好运行了，根据任务的优先级，位映像表中相应位就会被设置为 1。如果处理器支持位清零指令 CLZ，这个指令会加快位映像表的设置过程。

OS_PRIO.C 中包含了位映像表的设置、清除、查找的相关代码。这些函数都是 uC/OS-III 的内部函数，可以用汇编语言优化。

函数	功能
OS_PrioGetHighest()	查找最高优先级
OS_PrioInsert()	设置位映像表中相应的位
OS_PrioInsert()	清除位映像表中相应的位

为了确定就绪列表中优先级最高的任务，位映像表会被扫描，通过 OS_PrioGetHighest()函数找到优先级最高的任务。代码如下所示：

```
OS_PRIO  OS_PrioGetHighest (void)
{
    CPU_DATA  *p_tbl;
    OS_PRIO    prio;


    prio  = (OS_PRIO)0;
    p_tbl = &OSPrioTbl[0];
    while (*p_tbl == (CPU_DATA)0) {          /* Search the bitmap table for the highest priority   */
        prio += DEF_INT_CPU_NBR_BITS;        /* Compute the step of each CPU_DATA entry      */
        p_tbl++;
    }
    prio += (OS_PRIO)CPU_CntLeadZeros(*p_tbl);  /* Find the position of the first bit set at the entry  */
    return (prio);
}
```

1) OS_PrioGetHighest() 函数扫描 OSPrioTbl[]表直到找到非 0 的记录。这个循环最终会停止，因为总是有非 0 记录（空闲任务的存)。
2) 当这个表中全是 0 记录时，就会从下一个表中查找。优先级"prio"会被增加（如果

表长 32 位，就将 prio 加上 32)。

3) 当找到第一个非 0 位时，计算该位之前 0 位的个数，返回该优先级值。*如果 CPU 提供清零指令，可以通过这个指令优化代码。如果 CPU 没有这个指令，那么这个指令就只能用 C 语言模拟了。*

4) 函数 CPU_CntLeadZeros()统计了 CPU_DATA 记录中 0 的个数 （从左边开始，以位计)。例如，假定位映像表长 32 位，0xF0001234 返回的非 0 位前 0 位数的个数是 0。0x00F01234 返回的是 8。

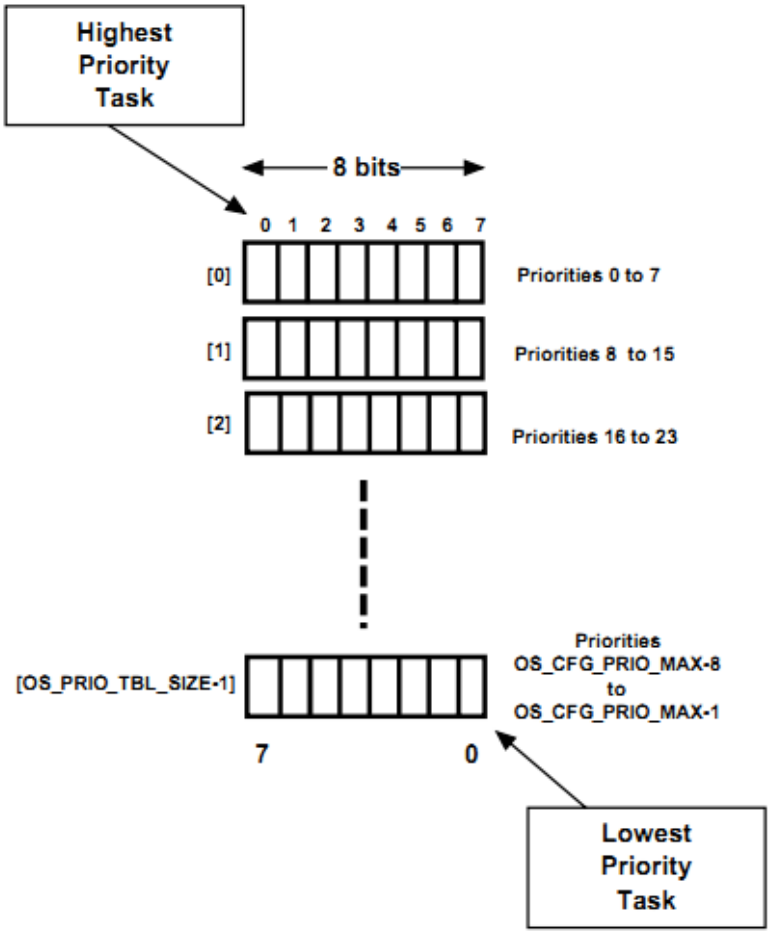

图 2.1：CPU_DATA 被声明为 CPU_INT08U

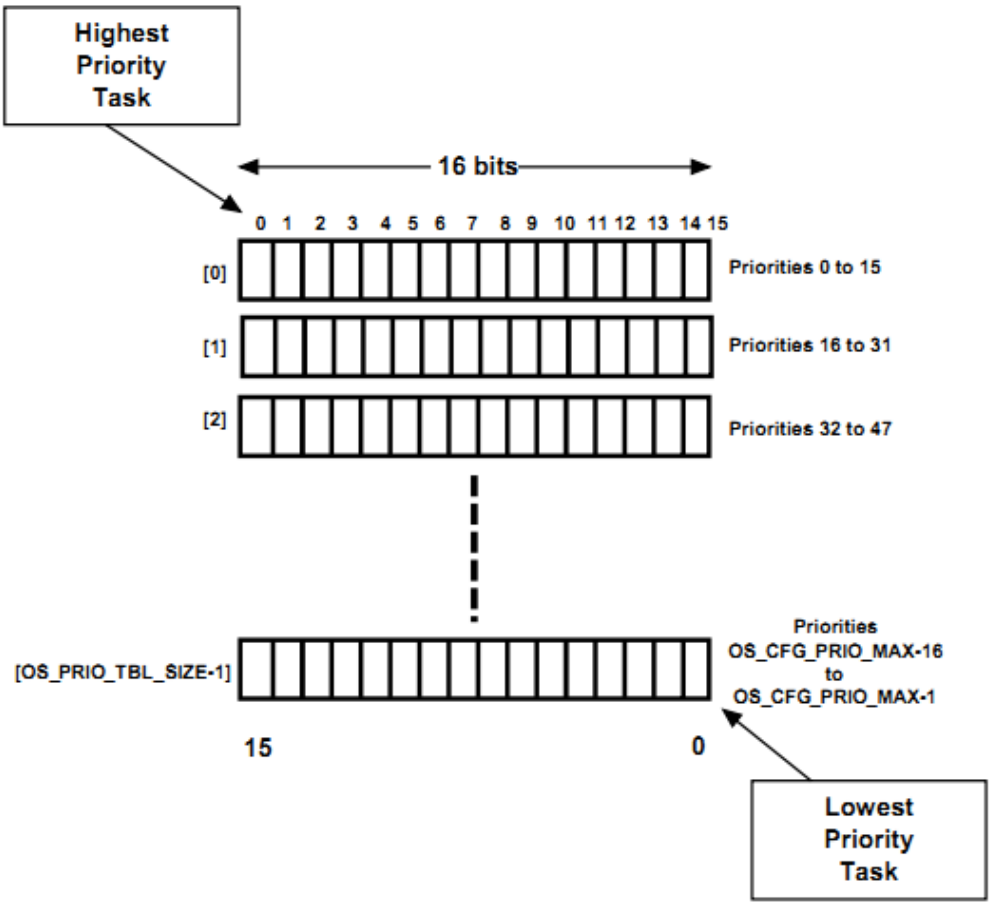

图 2.2：CPU_DATA 被声明为 CPU_INT16U

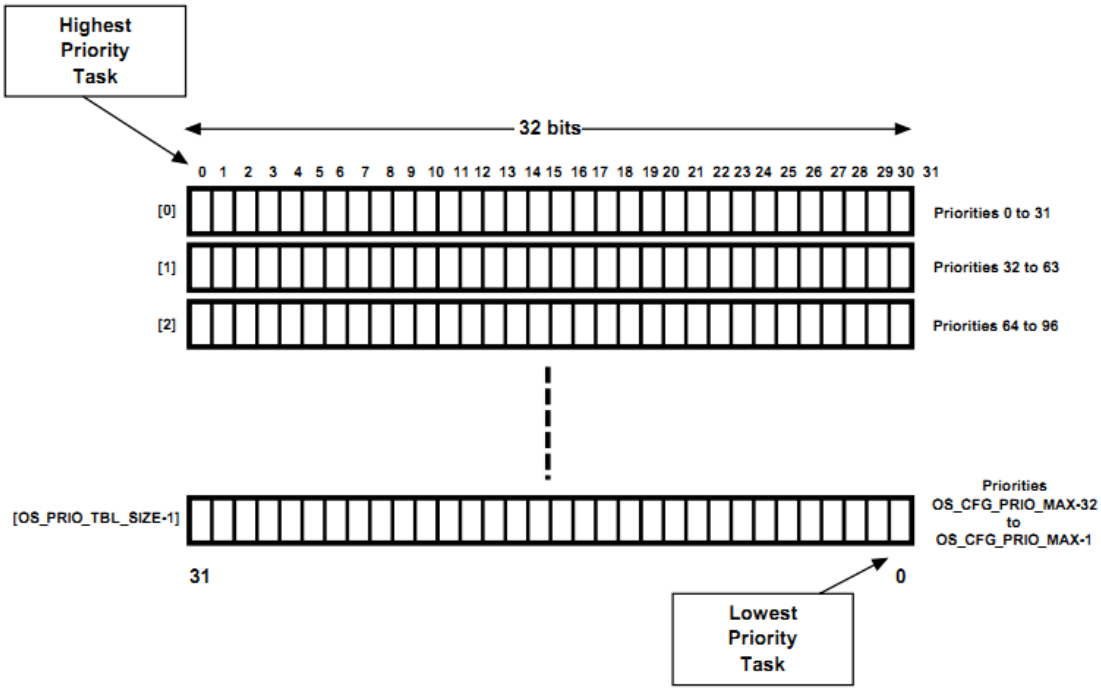

图 2.3：CPU_DATA 被声明为 CPU_INT32U

2.3 CPU_CntLeadZeros 和 CPU_CntLeadZerosTbl[256]

CPU_CntLeadZeros 中提供了针对不同 CPU_DATA 的支持，我们以 CPU_CFG_DATA_SIZE == CPU_WORD_SIZE_08 为例，说明 CPU_CntLeadZeros 的工作原理。

```
#if   (CPU_CFG_DATA_SIZE == CPU_WORD_SIZE_08)
    ix              = (CPU_INT08U)(val >>  0u);
    nbr_lead_zeros = (CPU_DATA  )(CPU_CntLeadZerosTbl[ix] +  0u);
```

ix 用来区分把 val 分成没 8 位一段，CPU_CFG_DATA_SIZE == CPU_WORD_SIZE_08 时，不要划分，故：

ix = (CPU_INT08U)(val >> 0u);

此时 ，当前行中最高优先级除 8 的余数为：

nbr_lead_zeros = (CPU_DATA)(CPU_CntLeadZerosTbl[ix] + 0u);

假如 ix 此时为：

ix = 0010 0000

则 CPU_CntLeadZerosTbl[ix]=2u，即：当前系统任务的最高优先级可能为 2，10，18 等等。

由 CPU_CntLeadZerosTbl 不难看出，CPU_CntLeadZerosTbl[ix]的值只跟 ix 的最高位有关，而与最高位后面的值无关，这样就取到了系统的当前行的最高优先级。

CPU_CntLeadZerosTbl[256]的定义如下：

```
static  const  CPU_INT08U  CPU_CntLeadZerosTbl[256] = {                                 /* Data vals : */
/*   0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F    */
    8u,  7u,  6u,  6u,  5u,  5u,  5u,  5u,  4u,  4u,  4u,  4u,  4u,  4u,  4u,  4u,  /*   0x00 to 0x0F   */
    3u,  3u,  3u,  3u,  3u,  3u,  3u,  3u,  3u,  3u,  3u,  3u,  3u,  3u,  3u,  3u,  /*   0x10 to 0x1F   */
    2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  /*   0x20 to 0x2F   */
    2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  2u,  /*   0x30 to 0x3F   */
    1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  /*   0x40 to 0x4F   */
    1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  /*   0x50 to 0x5F   */
    1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  /*   0x60 to 0x6F   */
    1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  1u,  /*   0x70 to 0x7F   */
    0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  /*   0x80 to 0x8F   */
    0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  /*   0x90 to 0x9F   */
    0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  /*   0xA0 to 0xAF   */
    0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  /*   0xB0 to 0xBF   */
    0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  /*   0xC0 to 0xCF   */
    0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  /*   0xD0 to 0xDF   */
    0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  /*   0xE0 to 0xEF   */
    0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u,  0u   /*   0xF0 to 0xFF   */
};
```

注：因为计算时 CPU_CntLeadZerosTbl[]的坐标不会取 0000 0000，所以，8u 仅凑数用，无实际意义。

3. 调度

调度器，决定了任务的运行顺序。uC/OS-III 是一个可抢占的 ， 基于优先级的内核。根据其重要性每个任务都被分配了一个优先级。uC/OS-III 支持多个任务拥有相同的优先级。

“可抢占的 ” 意味当事件发生时，如果事件让高优先级任务被就绪，uC/OS-III 马上将 CPU 的控制权交给高优先级任务 。因此，当一个任务提交信号量、发送消息给一个高优先级的任务（若该任务被就绪了），当前的任务就会被停止，更高优先级的任务获得 CPU 的控制权。类似的 ,当 ISR 提交信号量或发送消息给一更高优先级的任务(若该任务被就绪了), 那么中断返回的时候不会返回到原任务，而是高优先级任务。

3.1 抢占式调度

uC/OS- III通过两种方法处理中断提交的事件 :直接提交或延迟提交。从调度的角度看，这两种方法产生的结果是一样的 ;最高优先级的就绪任务会占用 CPU ,如图 3.1 ,3.2 所示：

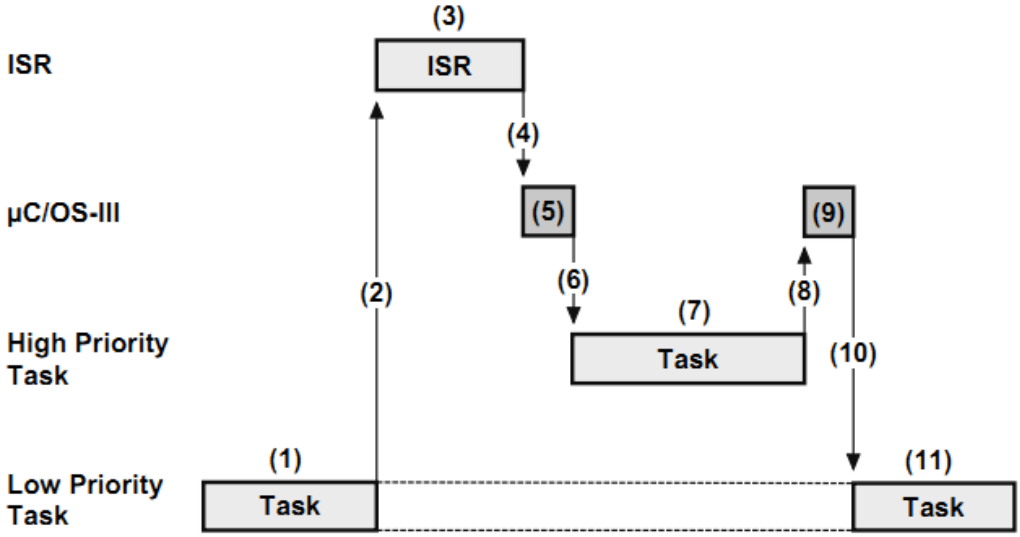

图 3.1：直接提交

1) 一个低优先级任务正在执行，这时中断发生了。
2) 如果中断使能，指令指针 IP 会跳转到中断服务程序。
3) 中断服务程序处理相关的程序，发送信号量或消息给一个高优先级任务。高优先级任务被就绪。
4) 中断服务程序完成操作后，它将 CPU 的控制权还给 uC/OS-III。
5) uC/OS-III 执行相应的操作。
6) 因为就绪队列中有一个更重要的任务，uC/OS-III 将不会返回中断前那个低优先级的任务，而是执行高优先级任务。
7) 开始执行高优先级任务。
8) 高优先级任务处理完成后，将 CPU 的控制权交给 uC/OS-III。
9) uC/OS-III 执行相应的操作。
10) uC/OS-III 转向执行原先那个低优先级任务。
11) 低优先级任务从被中断处继续执行。

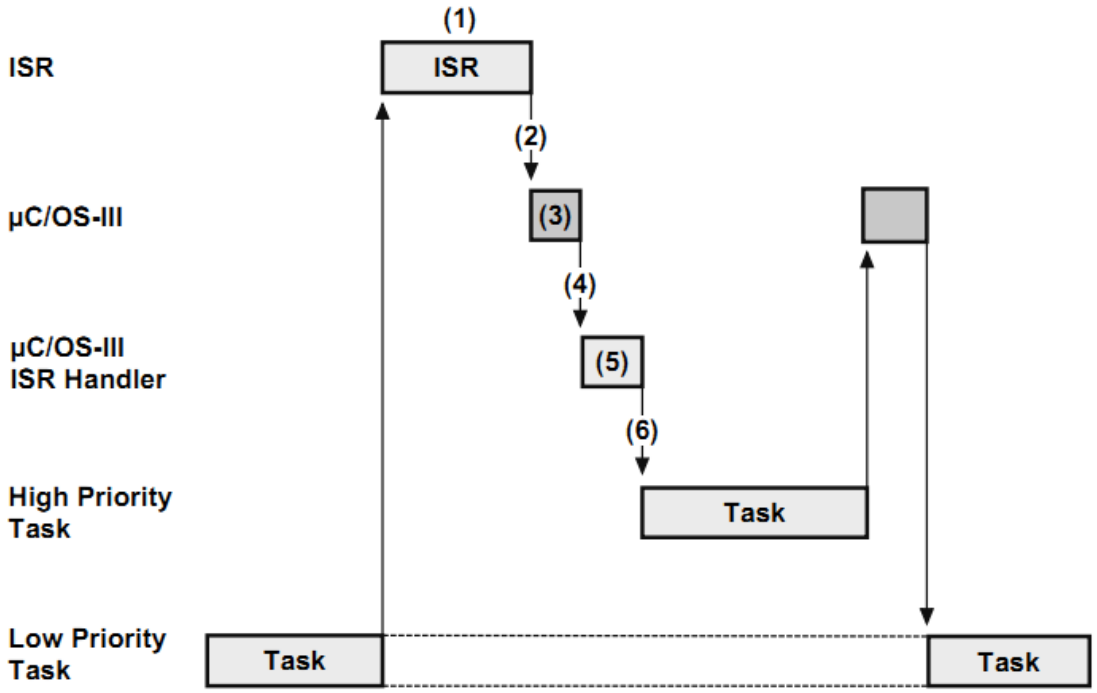

图 3.2：延时提交

1) 中断服务程序中，不是直接发送信号量或消息给任务。而是先将信号量或消息放人外部消息队列，并就绪中断处理任务。
2) 当 ISR 处理完它的工作时，就把 CPU 的使用权交给 uC/OS-III 。
3) uC/OS-III 处理一些操作。
4) 因为中断处理任务被就绪，uC/OS-III 将 CPU 的使用权交给它。
5) uC/OS-III 处理一些操作。
6) 中断处理任务将外部消息队列中的消息移除并重新提交给相应的任务。

这样，就将这个过程消息提交从中断级变成了任务级。使用这种方法的目的是为了减小关中断时间（详见第 9 章）。当消息队列为空时，uC/OS-III 将中断处理任务从就绪队列中移除，然后执行就绪队列中的最高优先级任务。

3.2 循环轮转调度

当多个任务有相同的优先级时，uC/OS-III 允许每个任务运行规定的时间片。当任务没有用完分配给它的时间片时，它可以自愿地放弃 CPU。uC/OS-III 允许任务在运行时开启或者关闭循环轮转调度。

图 3.3 是相同优先级下任务运行的时序图。如图，有三个优先级都为 X 的任务。为了说明，时间片的长度为 4。

1) 任务 3 正在被运行。在这段时间内，时基中断发生，但任务 3 还没有到期。
2) 任务 3 主动放弃剩下的时间片。
3) uC/OS-III 恢复任务 1，因为它是队列中任务 3 的下一个任务。
4) 任务 1 被执行直到分配给它的时间片到期。
5) 任务 3 正在被运行。在这段时间内，时基中断发生，但任务 3 还没有到期。
6) 任务 3 主动放弃剩下的时间片。

7) 有趣的事情发生了，uC/OS-III 会重新设置任务 1 的时间片长度为 4 个时基。但剩余时间片的计数是每个时基中断递减，即在第 4 个时基中断发生时时间片到期。而任务 1 是在时基即将发生时接手的，所以事实上它的时间片会少一个时基。

8) 任务 1 执行，uC/OS-III 允许用户在任务运行时修改时间片的长度

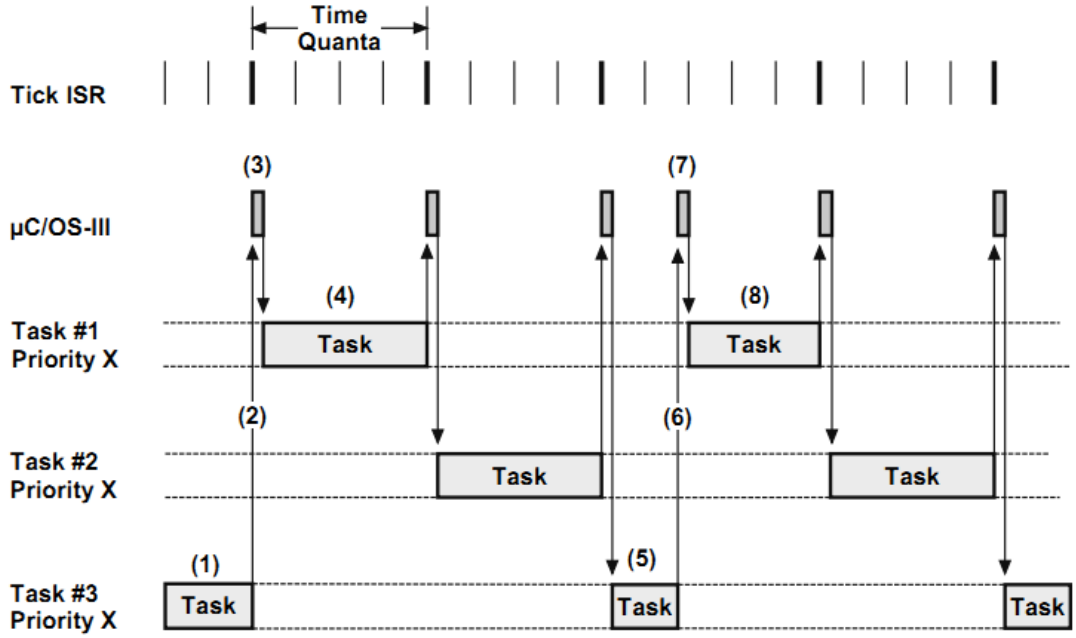

图 3.3：时间片轮转调度

uC/OS-III 允许用户为每个任务设置不同的时间片。不同的任务可以有不同的时间片。当任务被创建时，其时间片长度被设置。也可以在运行时调用 OSTaskTimeQuantaSet()修改。

3.3 调度的实现

通过这两个函数完成调度功能：OSSched()和 OSIntExit()。OSSched()在任务级被调用，OSIntExit()在中断级被调用。图 3.4 展示了调度所需用的两个数据结构：位映像表和就绪列表

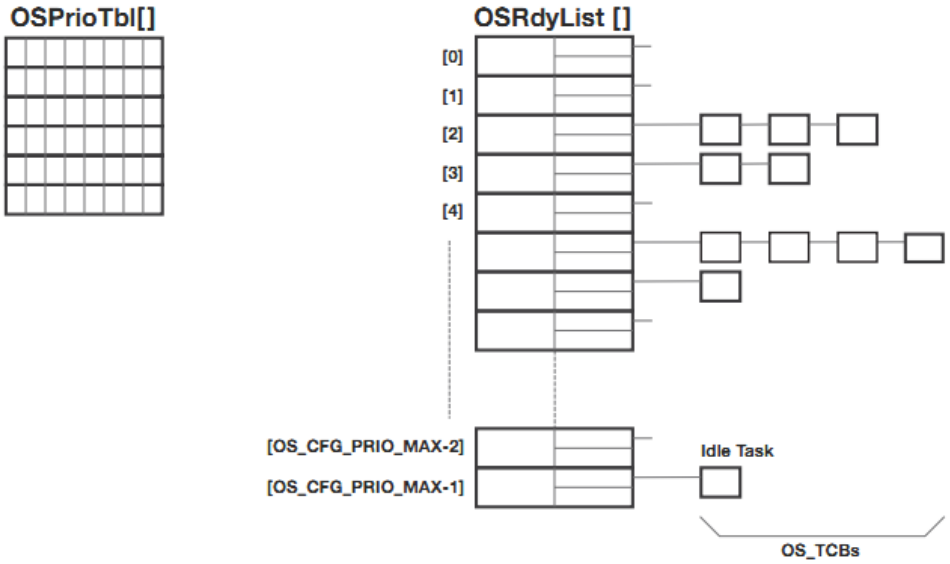

图 3.4：位映像表和就绪列表

3.3.1 OSSched()

以下是任务调度的伪代码：

```
void OSSched(void)
{
    Disable interrupts;
    if(OSIntNestingCtr > 0){                                    (1)
        return;
    }
    if(OSSchedLockNestintCtr > 0){                              (2)
        return;
    }
    Get highest priority ready;                                 (3)
    Get pointer to OS_TCB of next highest priority tast;        (4)
    if(OSTCBNHighRdyPtr != OSTCBCurPtr){                        (5)
        Perform tast level context switch;
    }
    Enable interrupts;
}
```

1) 因为要进入调度程序，关中断。OSSched()首先确定自己不是被中断服务程序调用，因为它是任务级调度程序。中断级调度需调用 OSIntExit()。
2) 第二步是确保调度器没有被锁。如果调度器被锁，就不能运行调度器，函数马上返回。
3) OSSched() 通过扫描位映像表 OSPrioTbl[] 找到就绪列表中优先级最高的位。
4) 确定好优先级后，索引到 OSRdyList[]并提取该记录中的首个 TCB（OSRdyList[highestpriority].HeadPtr）。到现在为止，已经知道了哪个任务将要切换为运行状态。特别的，OSTCBCurPtr 指向的是当前任务的 TCB，OSTCBHighRdyPtr 指向的是将要被切换的 TCB。
5) 当就绪表中的最高优先级任务不为当前正在运行的任务时（OSTCBCurPtr!=OSTCBHighRdyPtr），进行任务级的上下文切换。然后在开中断。

注意的是，调度时和上下文切换是运行在关中断的状态下的。这是必要的因为这些操作是原子性的。

3.3.2 OSIntExit()

以下是 ISR 级的调度器代码 OSIntExit() 如下所示。需要注意它假定当调用了 OSIntExit() 之后中断被关闭。

1) OSIntExit() 开始时确保中断嵌套级不为 0。（如果为 0 的话，那么接下来的 OSIntNestingCtr--就会发生溢出了）
2) OSInrExit()将 OSIntNestingCtr 递减，若 OSIntNestingCtr 不为 0 时则返回。确保该函数是在第一级 ISR 中执行。当还有中断程序未被处理时就不能运行调度器。
3) OSIntExit() 检查调度器是否被锁。如果被锁，OSIntExit()不会运行调度器并返回到中

断前的任务。

4) 当这是第一级中断且调度器没有被锁时。查找优先级最高的任务。
5) 从 OSRdyList[]中获得最高优先级的 TCB。
6) 如果最高优先级就绪任务不是当前正在运行的任务。uC/OS-III 就执行中断级的上下文切换。中断级切换的上文已经在中断开始时被保存了，它可以直接切换到任务。

```
void OSIntExit(void)
{
    if(OSIntNestingCtr == 0){                                   (1)
        return;
    }
    OSIntNestingCtr--;
    if(OSIntNestingCtr > 0){                                    (2)
        return;
    }
    if(OSSchedLockNestingCtr > 0){                              (3)
        return;
    }
    Get highest priority ready;                                 (4)
    Get pointer to OS_TCB of next highest priority tast;        (5)
    if(OSTCBNHighRdyPtr != OSTCBCurPtr){                        (6)
        Perform ISR level context switch;
    }
}
```

3.3.3 OS_SchedRoundRobin()