实时内核

任务管理

μC/OS-II

- μC/OS-II是一个可移植的、可固化的、可扩展的、抢占式的、实时确定性的多任务内核，适用于微处理器、微控制器和DSP

面向安全性至关重要的市场的可靠性

- 为了证明软件系统的可靠性和安全性，软件认证是至关重要的。μC/OS-II目前在大量高级别的安全关键设备中实现，包括:
  - 航空电子设备DO-178B认证
  - 药品FDA上市前通知(510(k))和上市前批准(PMA)器械
  - 用于运输和核系统的SIL3/SIL4 IEC, 99%符合汽车工业软件可靠性协会(MISRA-C:1998) C编码标准

应用

- μC/OS-II广泛应用于各种行业:
  - 航空电子设备——用于火星好奇号漫游者
  - 医疗设备/器械
  - 数据通信设备
  - 白色家电(电器)
  - 移动电话，pda, MID
  - 工业控制
  - 消费电子产品
  - 汽车
  - 广泛的其他安全性至关重要的嵌入式应用

软件包

Communication Software Stacks

TCP/IP
A compact, high-performance TCP/IP stack feauring 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.

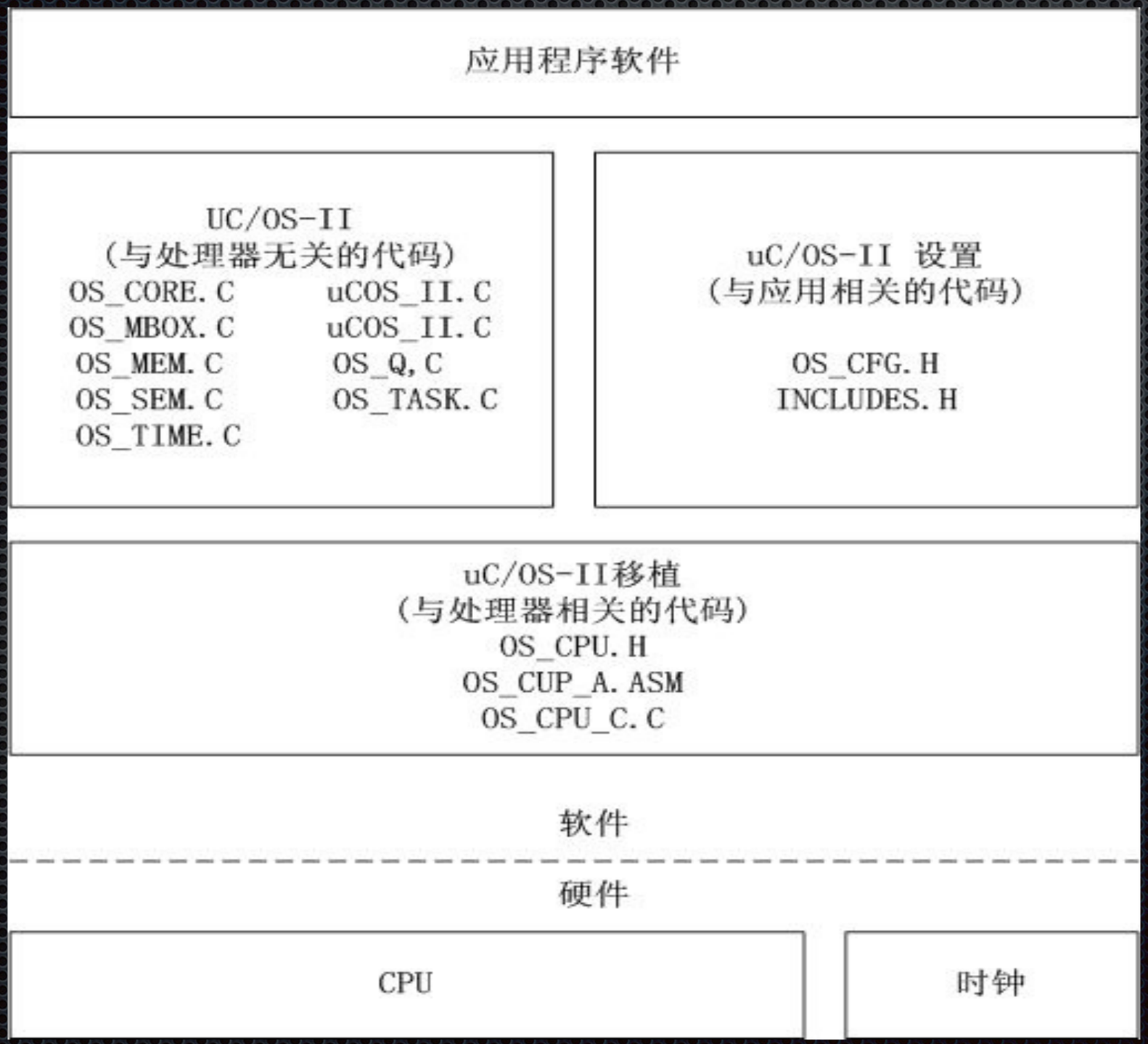
Storage & Display Software

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.

μC/OS-II的文件结构



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);
}
```

任务优先级

- μC/OS-II最多可管理64个任务
- μC/OS-II保留了四个最高优先级的任务和四个最低优先级的任务作为系统任务
- μC/OS-II实际使用的优先级只有两个：OSTaskCreate和OS_LOWEST_PRIO-1(参见OS_CFG.H)
- 留给多达56个应用程序任务
- 优先级越低，任务优先级越高
- 在μC/OS-II版本中，任务优先级号也作为任务的标识符

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

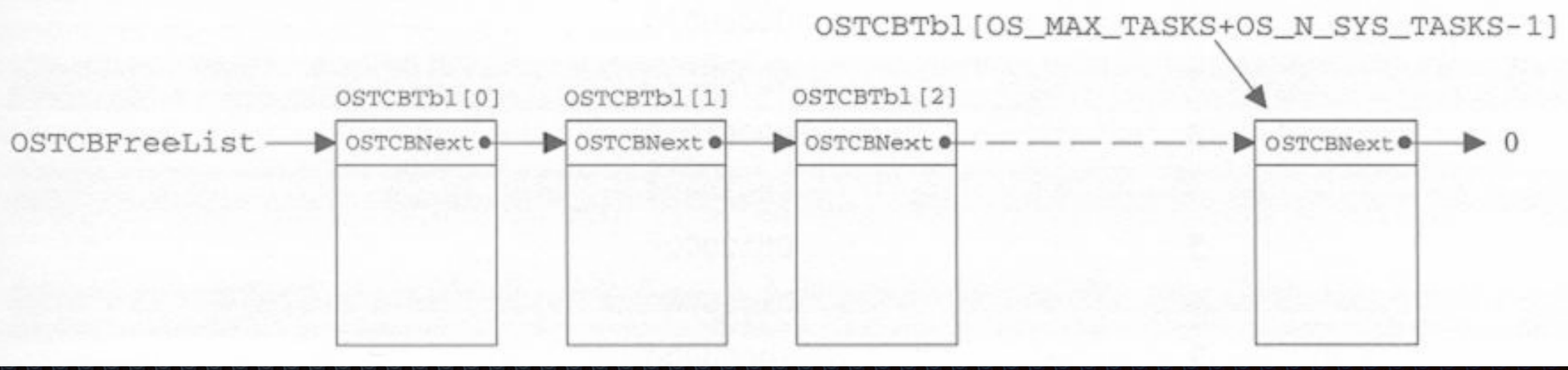
链表指针

- 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）
  - 双向的使用链表（头指针为OSTCBList）
- 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

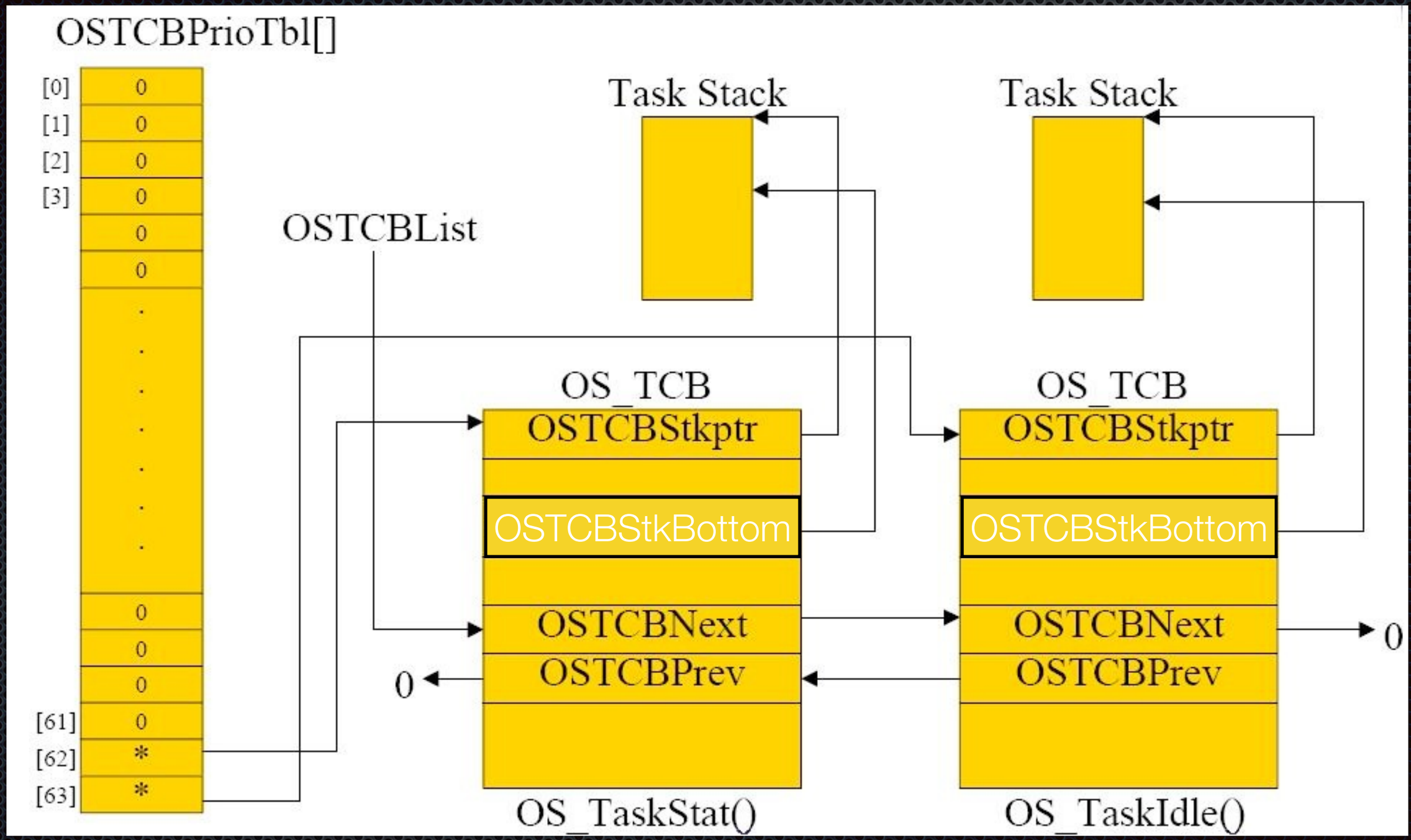
空闲TCB链表

- 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

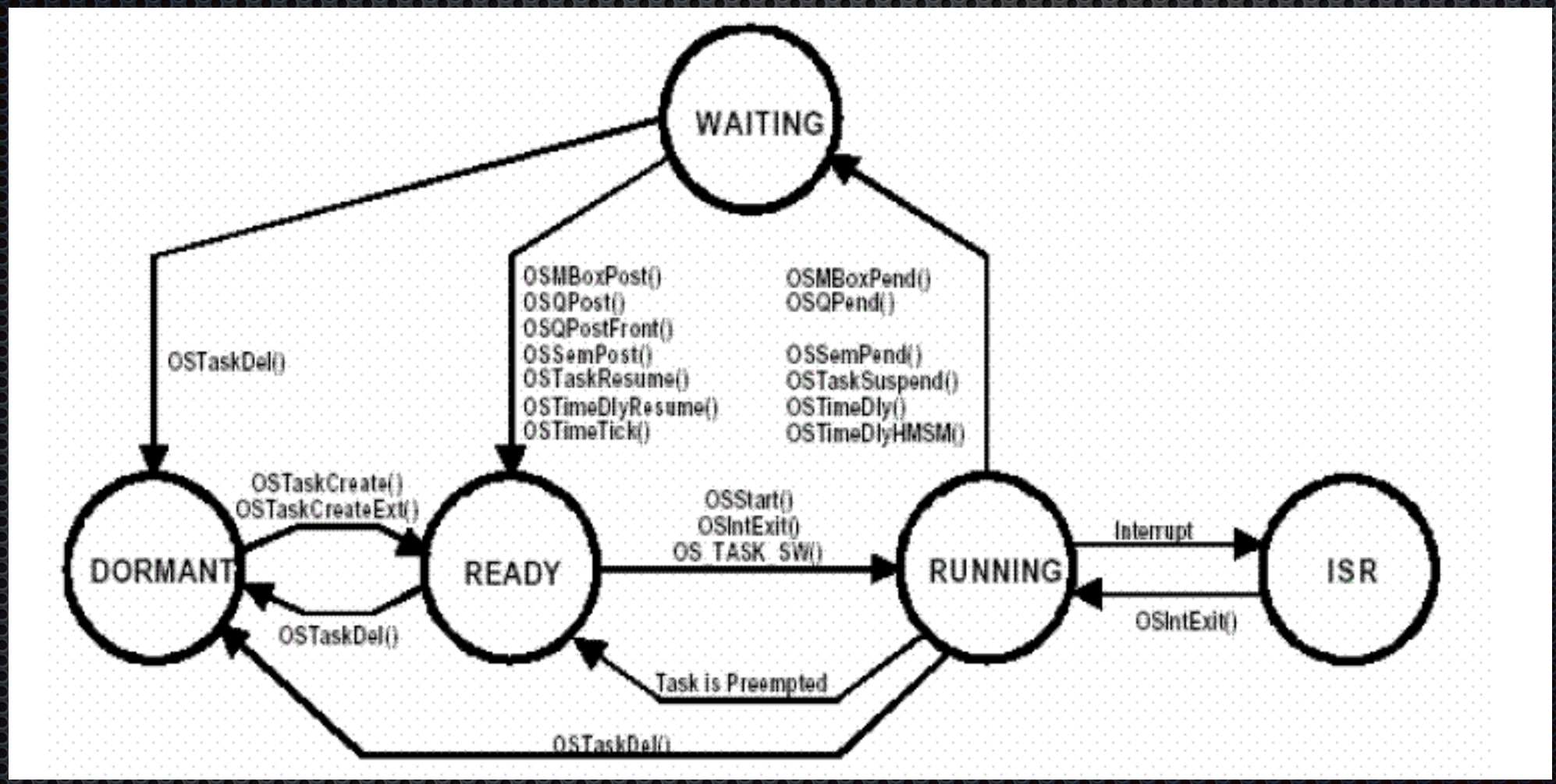
Figure 3.2 List of free OS_TCBs.



指针数组(指向相应TCB)



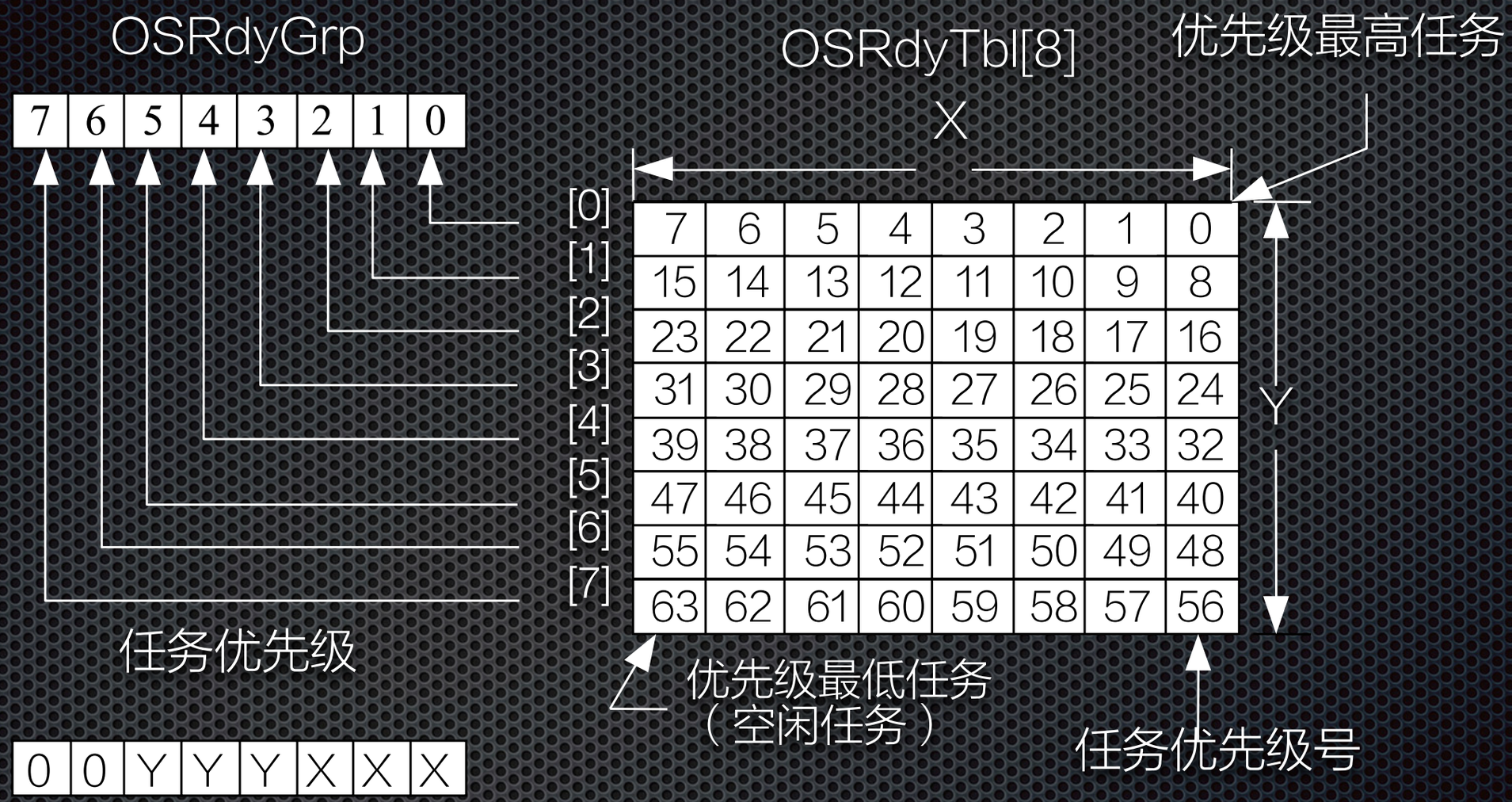
状态的转换


WAITING
OSMBoxPost()
OSQPost()
OSQPostFront()
OSSemPost()
OSTaskResume()
OSTimeDlyResume()
OSTimeTick()
OSMBoxPend()
OSQPend()
OSSemPend()
OSTaskSuspend()
OSTimeDly()
OSTimeDlyHMSM()
OSTaskDel()
OSTaskCreate()
OSTaskCreateExt()
DORMANT
READY
OSStart()
OSIntExit()
OS_TASK_SW()
RUNNING
Interrupt
ISR
OSIntExit()
OSTaskDel()
Task is Preempted
OSTaskDel()

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：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
  - OSRdyTbl[ ]：
    - 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OSRdyTbl[i]（0≤i≤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(1)

- 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 |= 0x02
  - OSRdyTbl[1] |= 0x10
- 而优先级为21的任务就绪21=10101b，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 |= 0x04
  - OSRdyTbl[2] |= 0x20

根据优先级确定就绪表(2)

- 从上面的计算可知：若OSRdyGrp及OSRdyTbl[]的第n位置1，则应该把OSRdyGrp及OSRdyTbl[]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（0000 0001）
  - OSMapTbl[1] = 2^1 = 0x02（0000 0010）
  - ……
  - OSMapTbl[7] = 2^7 = 0x80（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，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

调度器

- 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

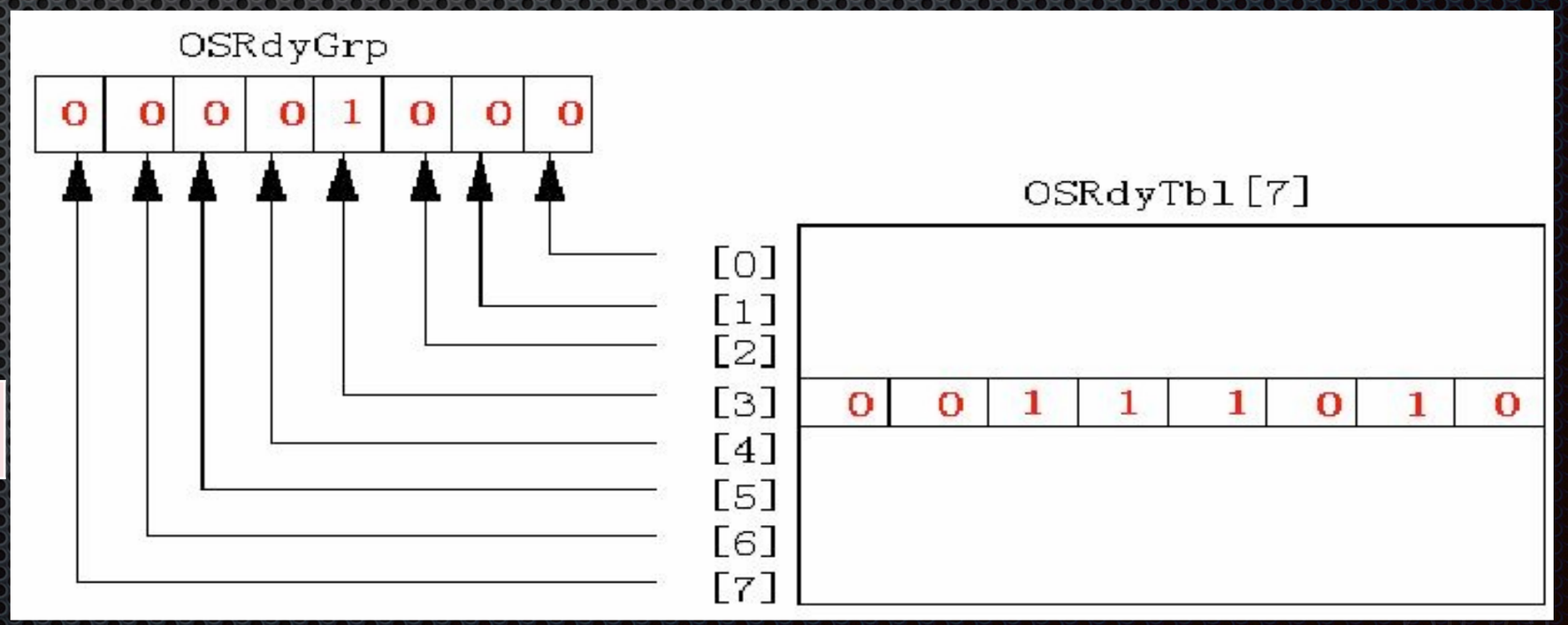
- 两个关键:
  - 将优先级数分解为高三位和低三位分别确定（64个任务版本）
  - 高优先级有着小的优先级号

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

任务优先级

0	0	0	1	1	0	0	1



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

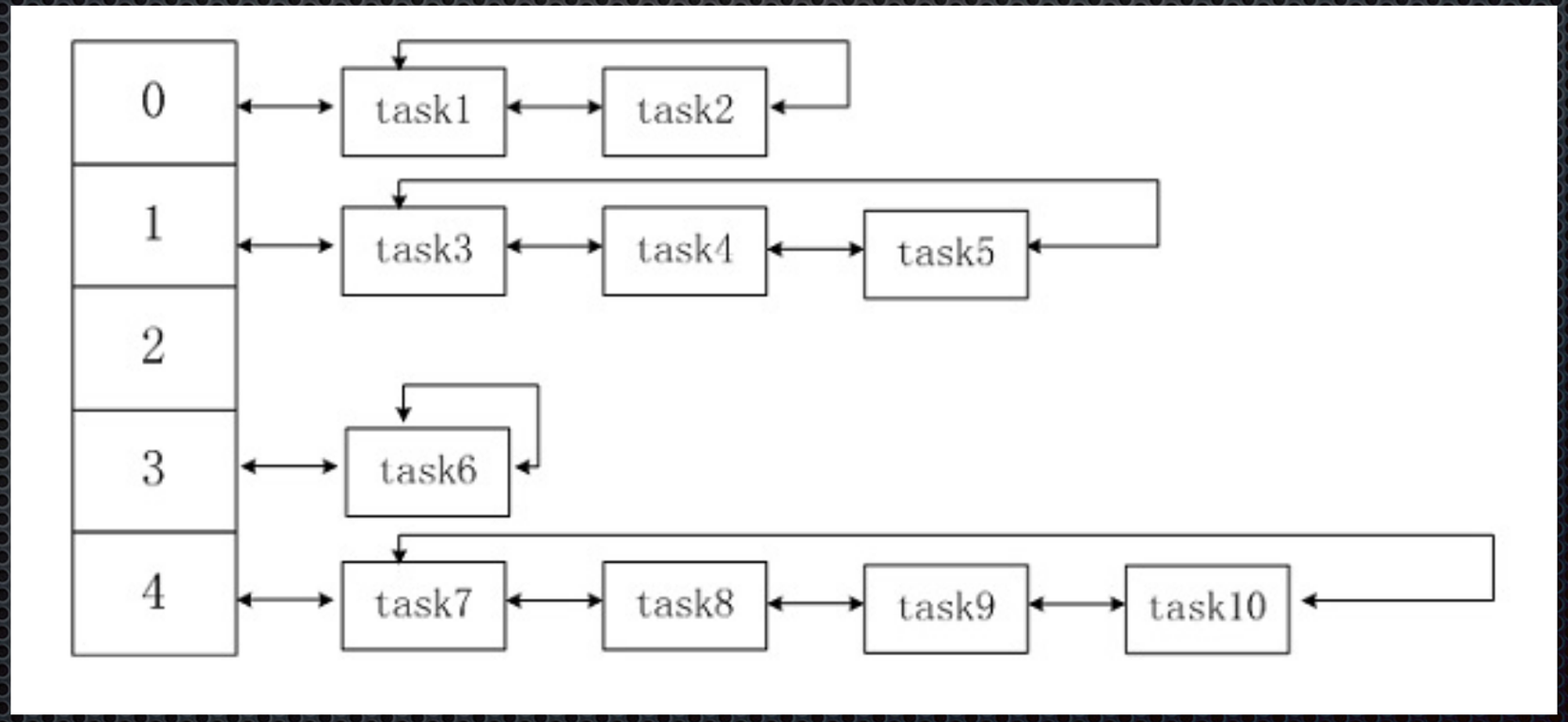
Prio=3*8+2=26

64个任务扩展到256个任务

```
static void OS_SchedNew (void)
{
#if OS_LOWEST_PRIO <= 63// μ C/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
}
```

freertos的任务优先级调度示意

- FreeRTOS 是一个实时操作系统，所奉行的调度规则：
  - 高优先级抢占低优先级任务，系统永远执行最高优先级的任务
  - 同等优先级的任务轮转调度
-



Linux增强实时能力

- 自从linux内核2.6.23以来，默认的进程调度器就被设置为完全公平调度器（CFS，complete fair scheduler），取代了之前的O(1)调度器
- 实时属性
  - 常用方法是将Linux内核与所有的Linux用户模式进程都作为底层实时操作系统的特定任务，且仅在非实时任务需要运行时才能激活。
  - 最新方法是SCHED-DEADLINE（集成在3.14（2014）版本的内核中）
    - 基于EDF

SCHED_DEADLINE

- 使用三个参数来调度任务
  - runtime：>=WCET
  - period：<=P
  - deadline：D

实时任务参数(WCET、D、P)

- 当给这个调度策略增加一个新的任务时，要进行可调度性测试，且仅当该测试成功时该任务才可被接受，并推迟到下一个执行周期
- 调度截止时限
  - 不是绝对截止时限
  - 每次任务唤醒时，调度器都会计算一个调度截止时限scheduling deadline（使用常带宽服务器 CBS 算法）
  - 然后在这些调度截止时限内使用 EDF调度任务

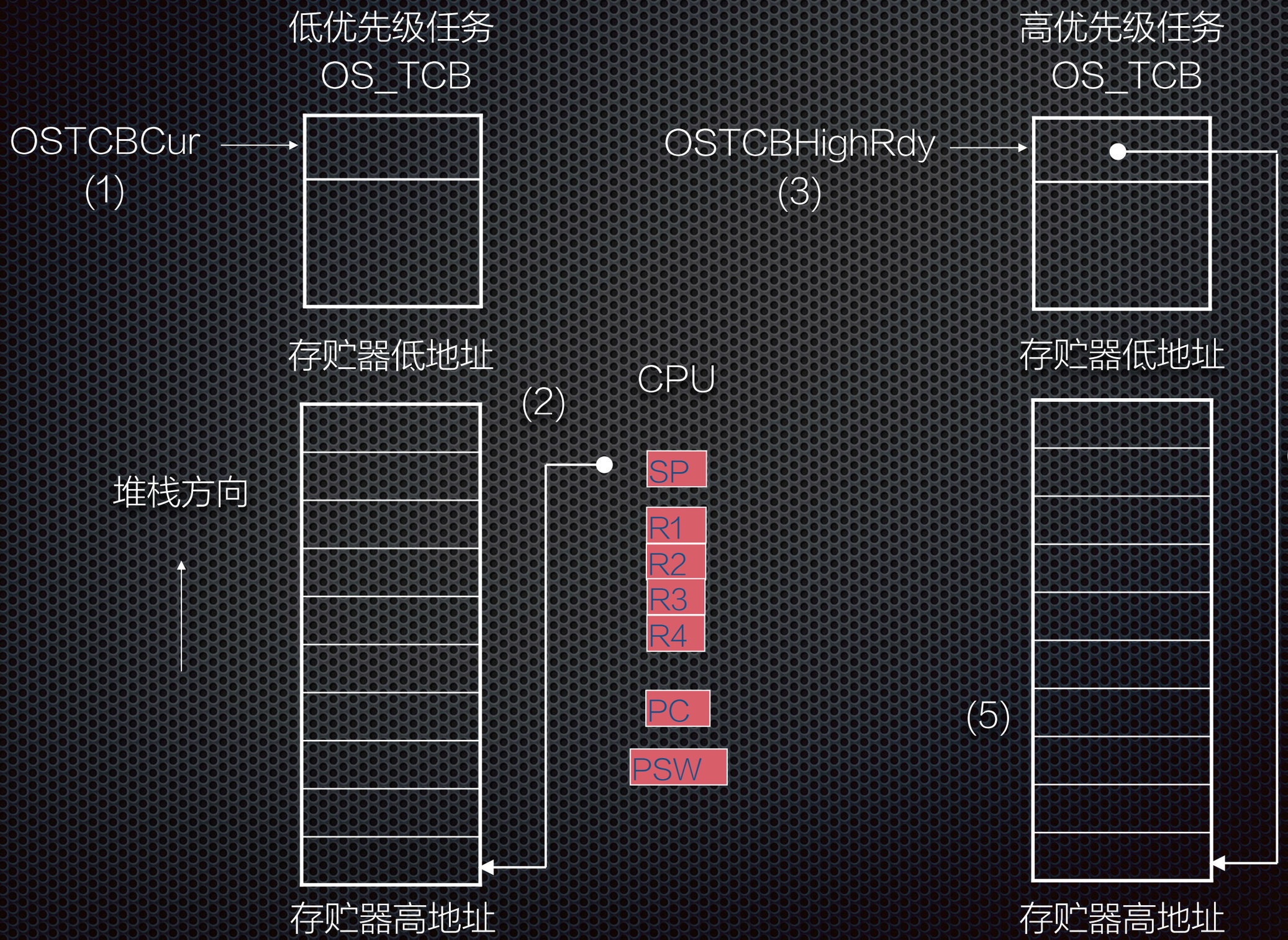
任务切换

- 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

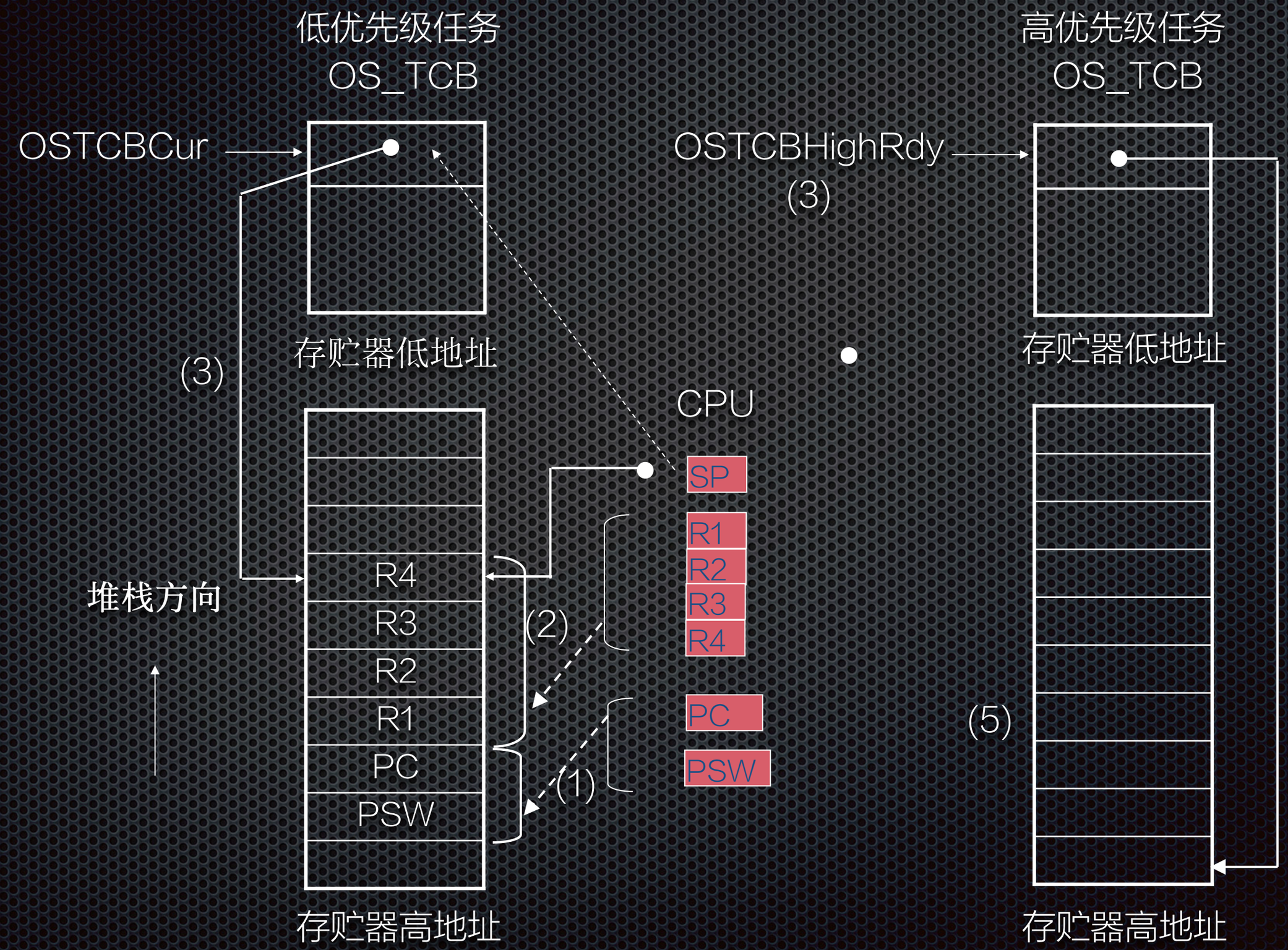
任务级的任务切换OS_TASK_SW()

- 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

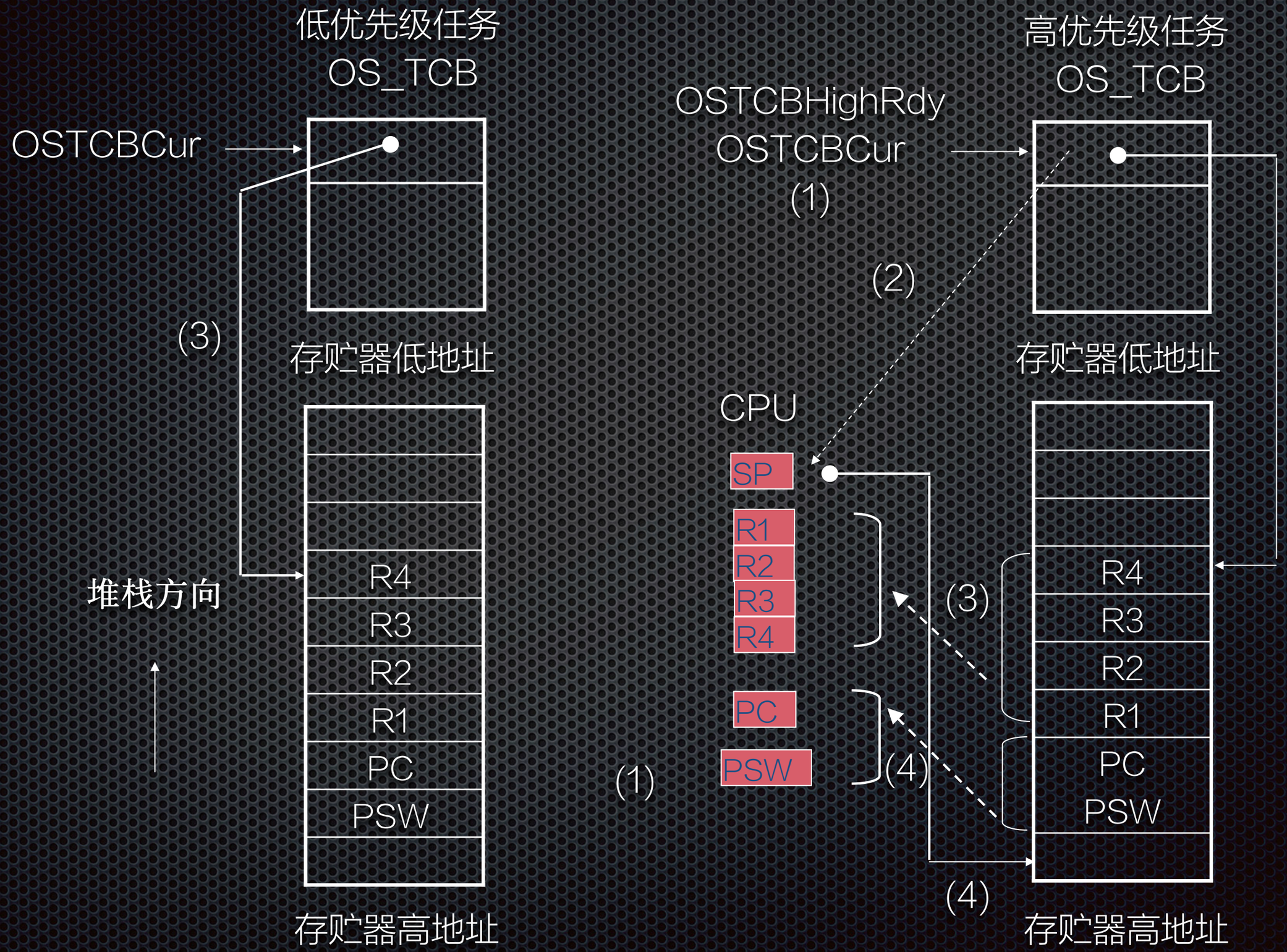
调用OS_TASK_SW()前的数据结构



保存当前CPU寄存器的值



重新装入要运行的任务


低优先级任务
OS_TCB
OSTCBCur
(3)
存贮器低地址
R4
R3
R2
R1
PC
PSW
堆栈方向
存贮器高地址
OSTCBHighRdy
OSTCBCur
(1)
高优先级任务
OS_TCB
(2)
存贮器低地址
(4)
CPU
SP
R1
R2
R3
R4
PC
PSW
(1)
(3)
(4)
R4
R3
R2
R1
PC
PSW
(4)
存贮器高地址

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;
}
```

给调度器上锁

- OSSchedlock()：给调度器上锁函数，用于禁止任务调度，保持对CPU的控制权（即使有优先级更高的任务进入了就绪态）
  - 当低优先级的任务要发消息给多任务的邮箱、消息队列、信号量时，它不希望高优先级的任务在邮箱、队列和信号量还没有得到消息之前就取得了CPU的控制权，此时，可以使用调度器上锁函数
- OSSchedUnlock()：给调度器开锁函数，当任务完成后调用此函数，调度重新得到允许

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_LOWSEST_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);
```

参考文献

- https://www.silabs.com/developers/micrium
- Deadline Task Scheduling，www.kernel.org/doc/html/latest/scheduler/sched-deadline.html