

嵌入式系统

An Introduction to Embedded System

第六课 开源嵌入式RTOS内核分析

浙江大学计算机学院人工智能研究所

课程大纲

 μ C/OSII操作系统简介

 操作系统内核移植简介

课程大纲

 μ C/OSII操作系统简介

 操作系统内核移植简介

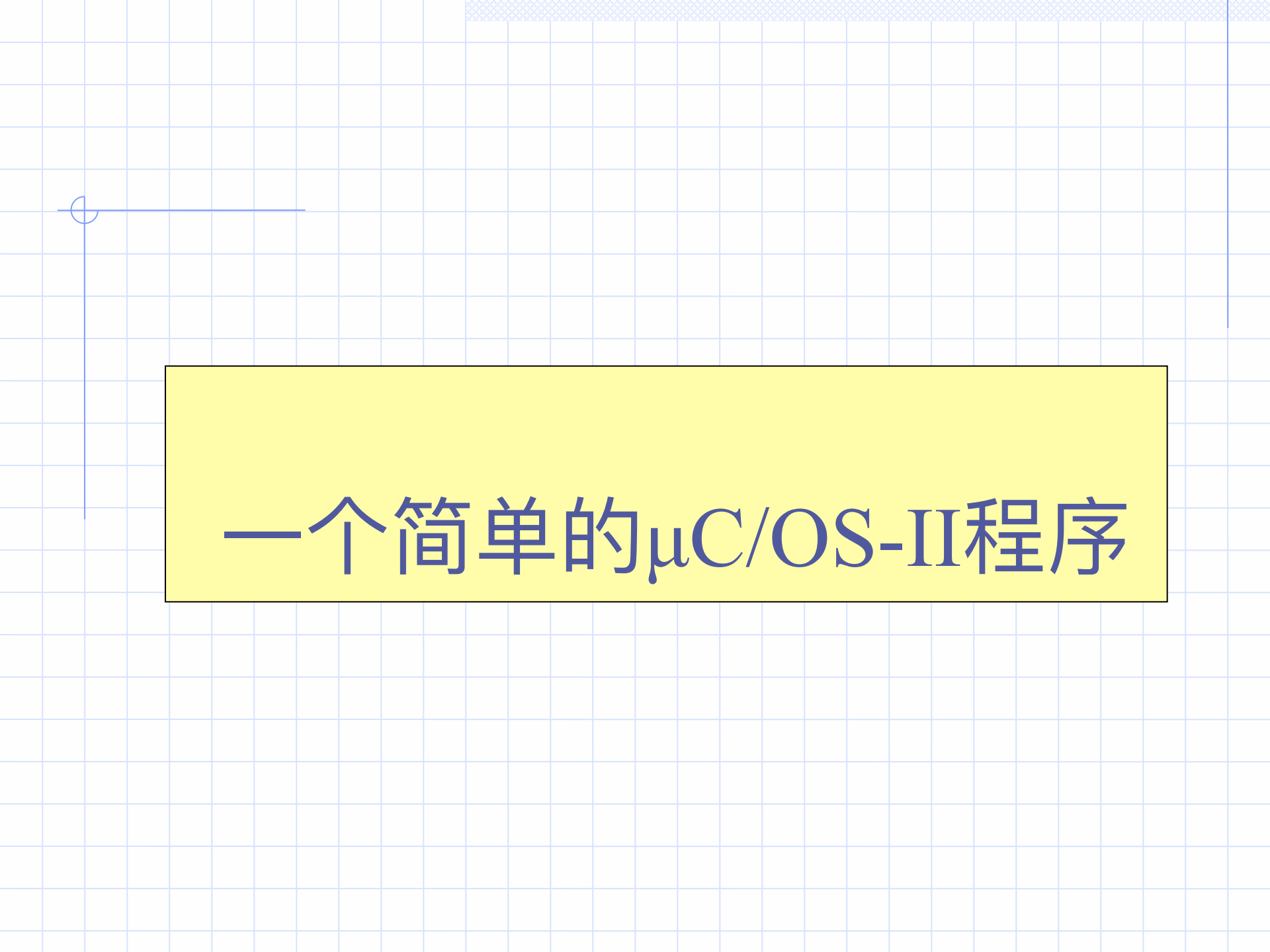
μC/OS简介 (1/2)

- ◆ μC/OS全称：micro Control OS，意为“微控制器操作系统”
- ◆ μC/OS由嵌入式领域研究人员Jean J.Labrosse于1992年在《嵌入式系统编程》杂志的5月、6月刊上刊登的文章连载，并把μC/OS的源码发布在该杂志的BBS上，被广泛下载和应用。
- ◆ Jean J.Labrosse于1999年建立了Micrium公司，提供高质量的嵌入式软件和解决方案，出售μC/OS-II及其他软件的商用许可证。



μC/OS简介 (2/2)

- p μC/OS是一种基于优先级抢占式、可移植、可裁剪的多任务实时操作系统，其特点为短小、精悍。
- p μC/OS绝大部分源码是用**ANSI C**写的，与硬件相关的那部分汇编代码被压缩至最低限度，使得系统移植性强。
- p μC/OS经裁剪最小可达**2KB**，最小数据**RAM**需求**10KB**。
- p μC/OS可以在**8位~64位**，超过**40种**不同架构的微处理器上运行，在世界范围内得到广泛应用，包括：手机、路由器、集线器、不间断电源、飞行器、医疗设备及工业控制。



一个简单的 μ C/OS-II程序

一个简单的应用程序

- 初始化、启动

一个简单的应用程序

● 初始化、启动

```
OS_STK userAppTaskStk1[1000];
OS_STK userAppTaskStk2[1000];

extern void userApp1(void *);
extern void userApp2(void *);

void main()
{
    ...                //硬件的配置

    OSInit();           //初始化操作系统

    //创建两个优先级分别为5、6的任务

    OSTaskCreate(userApp1, (void *) 0, &userAppTaskStk1[1000-1],5);

    OSTaskCreate(userApp2, (void *) 0, &userAppTaskStk2[1000-1],6);

    OSStart();          //启动操作系统
}
```


一个简单的应用程序

- 两个任务

操作系统启动后任务1先运行，输出“in userApp1”，任务1延时，任务2运行，输出“in userApp2”，1秒后任务1继续运行，输出“in userApp1”...

一个简单的应用程序

- 两个任务

操作系统启动后任务1先运行，输出“in userApp1”，任务1延时，任务2运行，输出“in userApp2”，1秒后任务1继续运行，输出“in userApp1”...

```
void userApp2(void * args)
{
    while(1)
    {
        uart_string("in userApp2");
        OSTimeDly(100);
    }
}

void userApp1(void * args)
{
    while(1)
    {
        uart_string("in userApp1");
        OSTimeDly(100);
    }
}
```

μC/OSII代码树结构

与处理器无关的文件

```
source
|-- OS_CORE.C
|-- OS_DBG_R.C
|-- OS_FLAG.C
|-- OS_MBOX.C
|-- OS_MEM.C
|-- OS_METEX.C
|-- OS_Q.C
|-- OS_SEM.C
|-- OS_TASK.C
|-- OS_TIME.C
|-- uCOS_II.C
`-- uCOS_II.H
```

与应用相关的文件

```
编译器\处理器
|-- OS_CFG_R.H
`-- INCLUDES.H
```

与处理器有关的文件

```
处理器\编译器
|-- OS_CPU.H
|-- OS_CPU_A.ASM
`-- OS_CPU_C.C
```

μC/OSII核心代码（C语言）统计

序号	模块名称		开发语言	代码行数
1	任务管理和调度		ANSI C, x86 asm	983
2	任务 间同 步与 通讯	邮箱	ANSI C	366
		消息队列	ANSI C	519
		互斥信号量	ANSI C	440
		计数信号量	ANSI C	349
		事件标志	ANSI C	700
		公共部分	ANSI C	189
3	内存管理		ANSI C	257
4	时钟和中断		ANSI C	931
总计			4,734	

μC/OSII系统功能

序号	模块名称	功能	支持情况
1	任务管理和调度	任务调度方式	优先级抢占 (不支持时间片轮转)
		内核抢占	不
		优先级数量	64
		任务数量	64 (其中, 8个系统任务)
2	同步和通信	同步	信号量、事件标志组
		通信	消息队列、邮箱
		优先级反转	不支持
		有限等待	支持
		递归申请	不支持
3	内存管理	MMU	不支持
		内核数据结构	静态分配
		用户内存使用	固定大小的内存分配
4	中断管理	中断服务程序	不支持ISR自动插桩

μC/OSII内核结构—临界区保护

p 为了处理临界区，代码需要关中断，处理完毕后再开中断。使μC/OSII能够避免同时有其它任务或中断服务进入临界段代码。

p μC/OSII定义两个宏(macros)来关中断和开中断：

✓ OS_ENTER_CRITICAL()

✓ OS_EXIT_CRITICAL()。

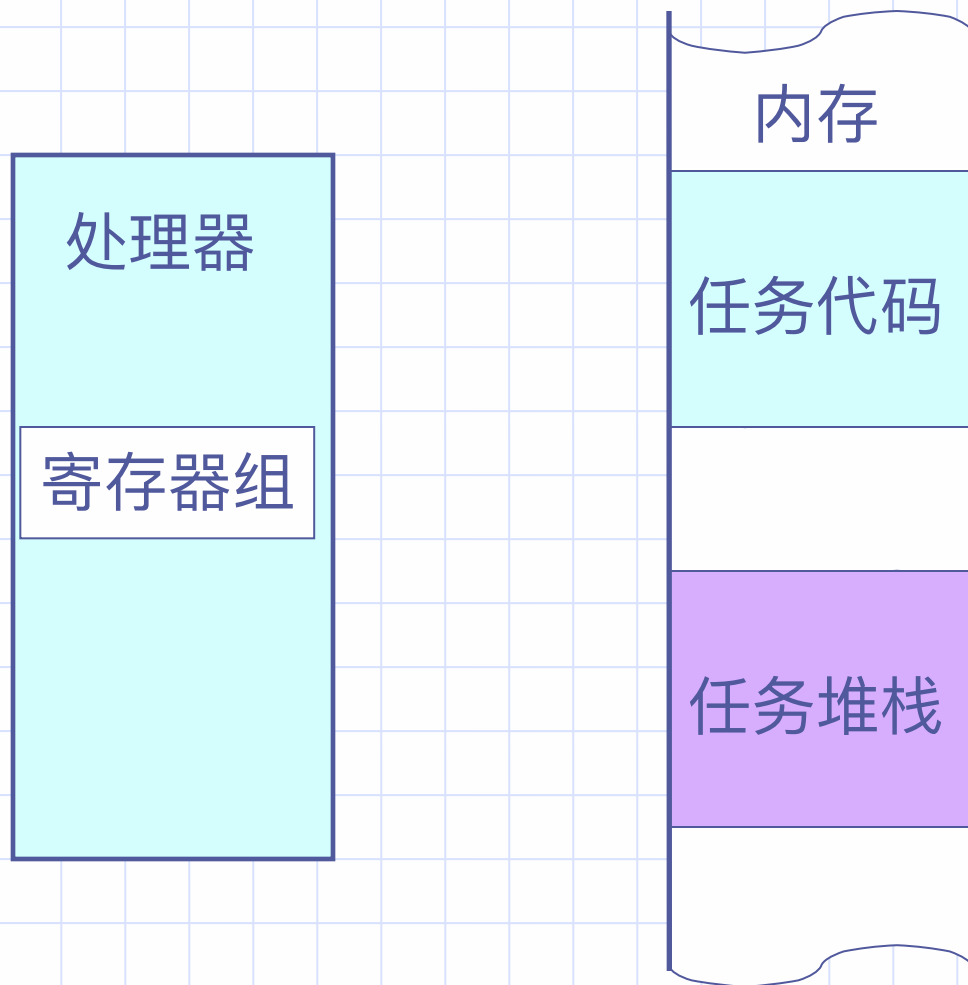
```
#define OS_ENTER_CRITICAL() asm CLI  
#define OS_EXIT_CRITICAL()  asm STI
```

OS_CPU.H
(x86)

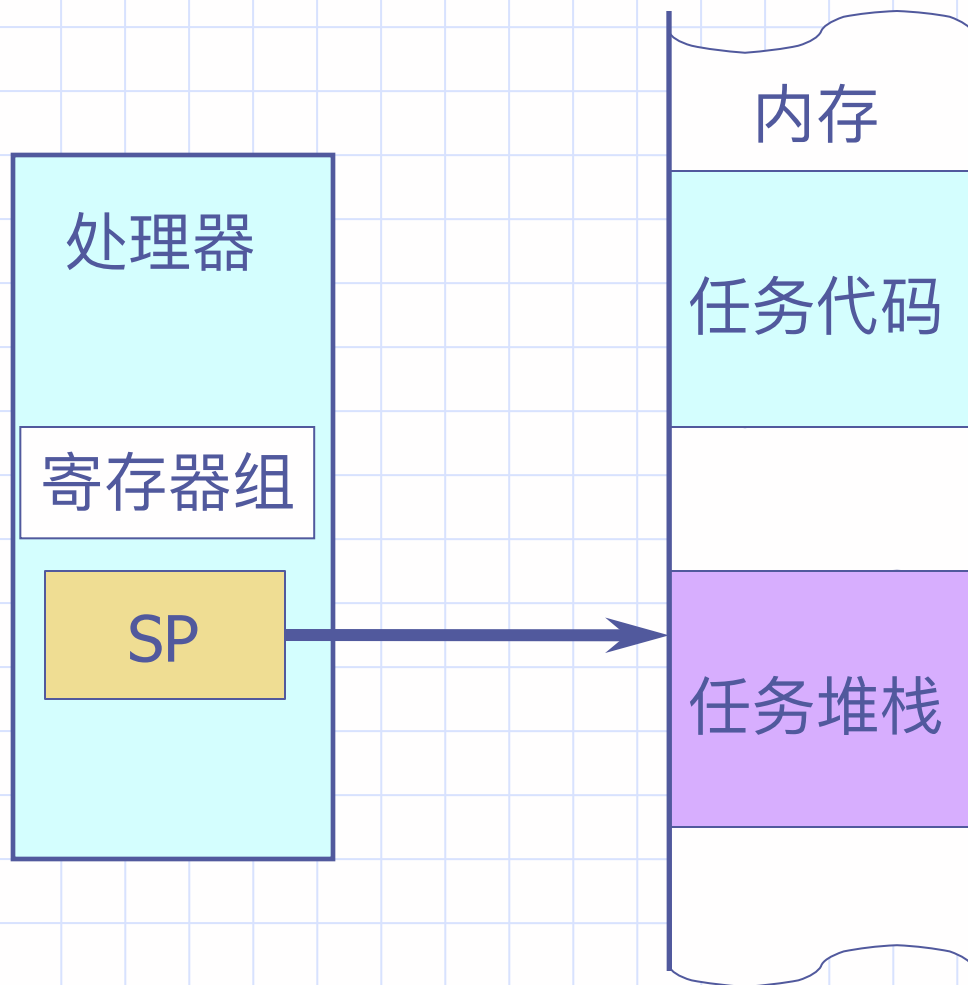


任务的基本概念

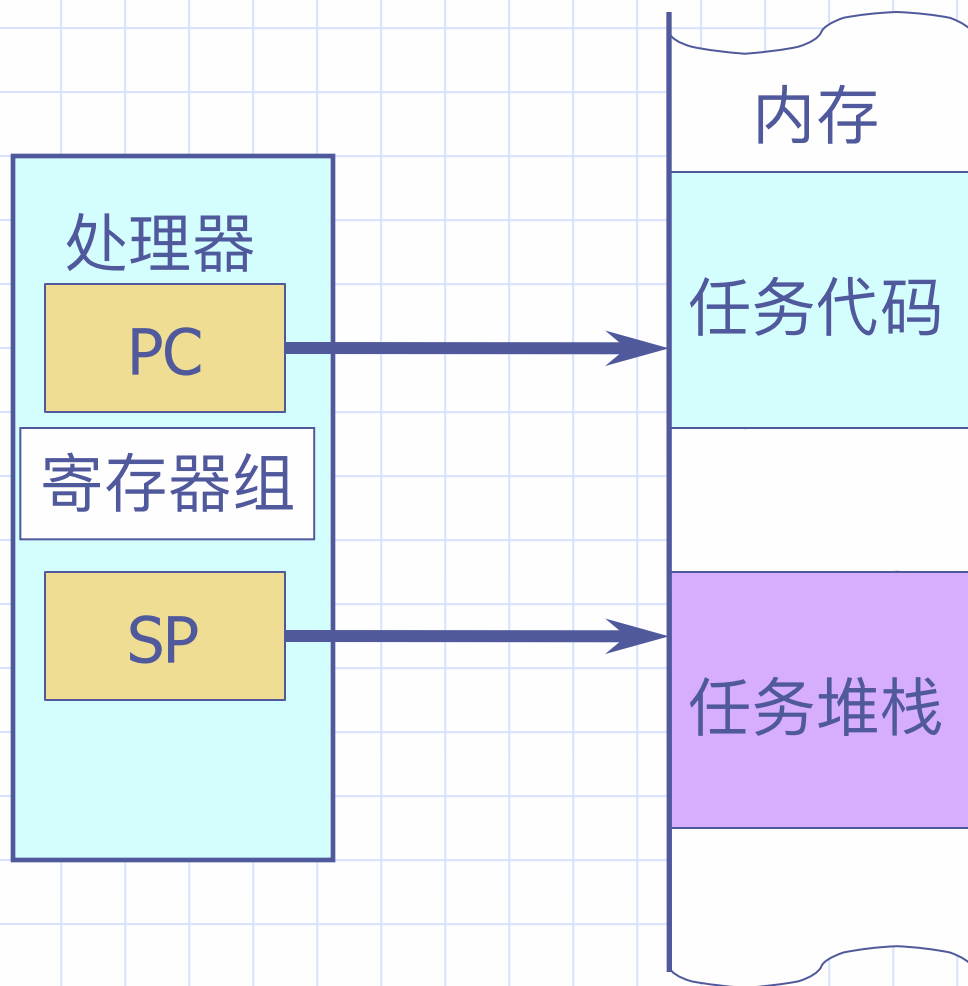
任务运行时与处理器之间的关系



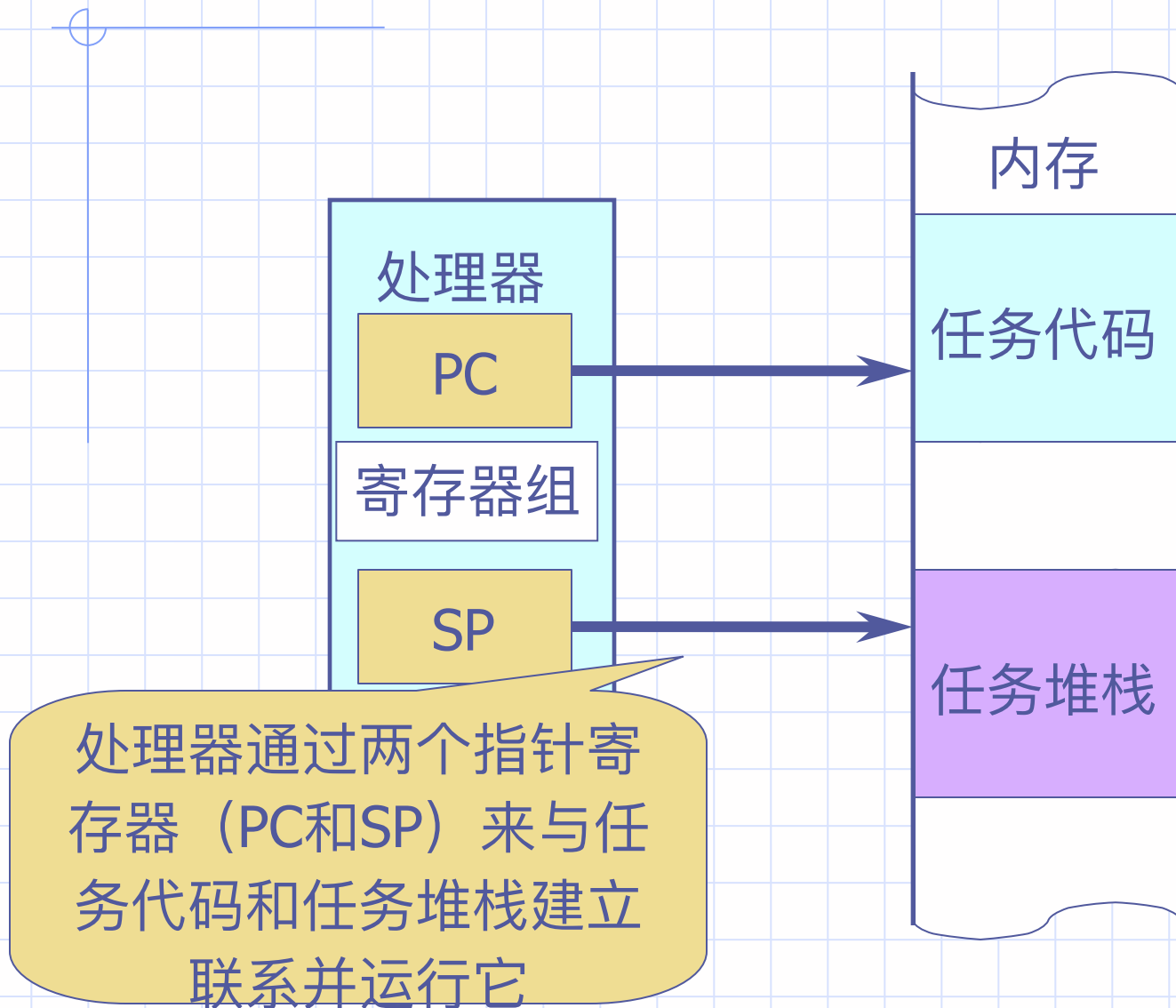
任务运行时与处理器之间的关系



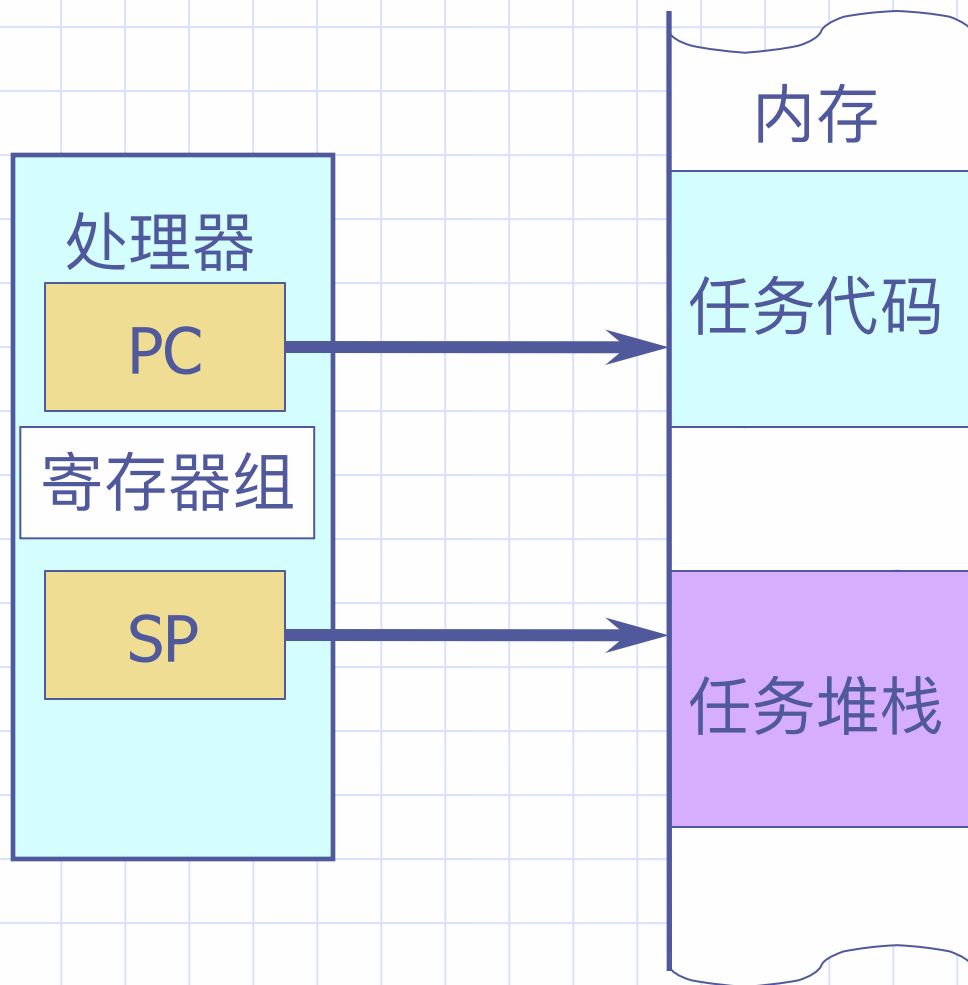
任务运行时与处理器之间的关系



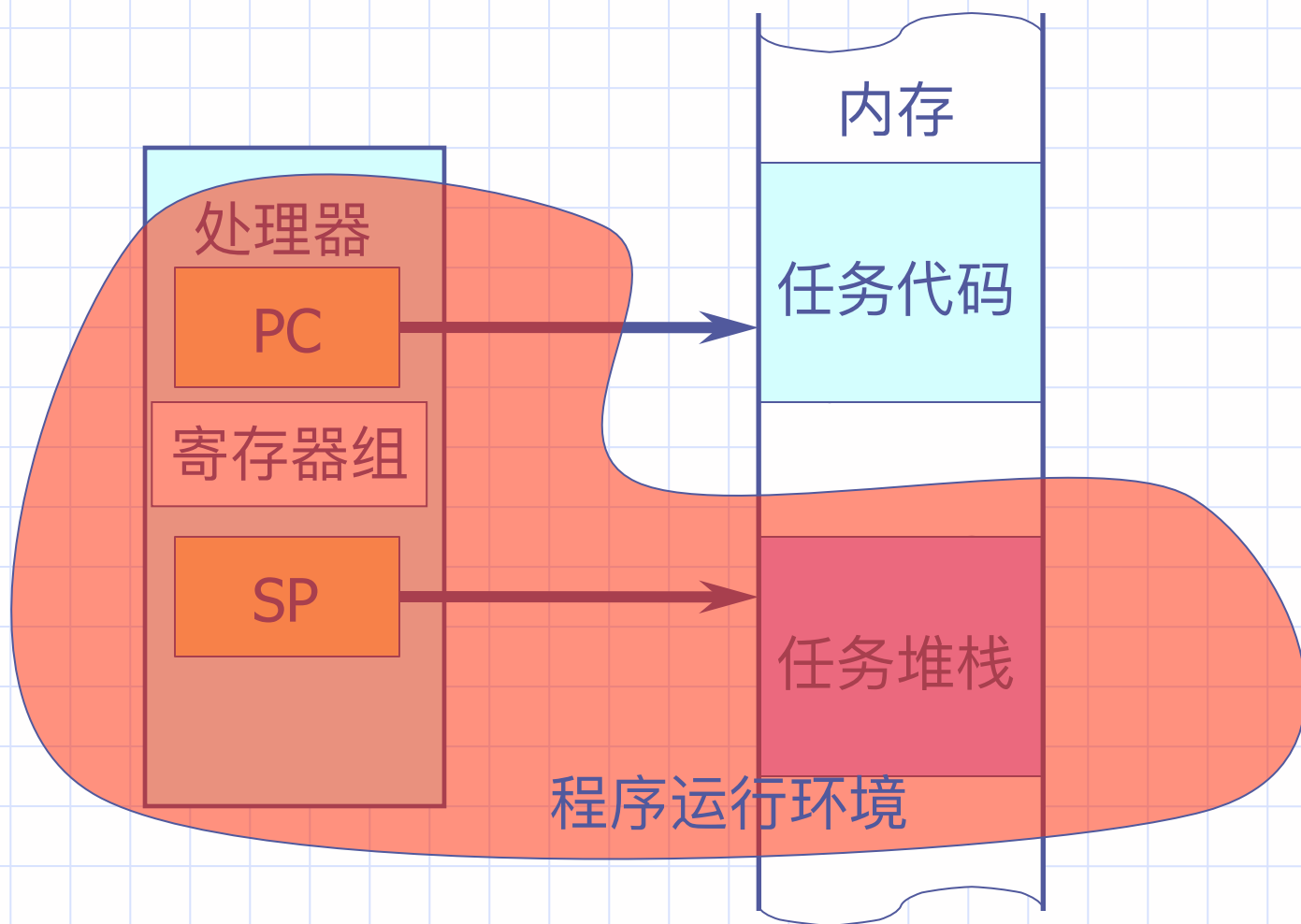
任务运行时与处理器之间的关系



任务运行时与处理器之间的关系

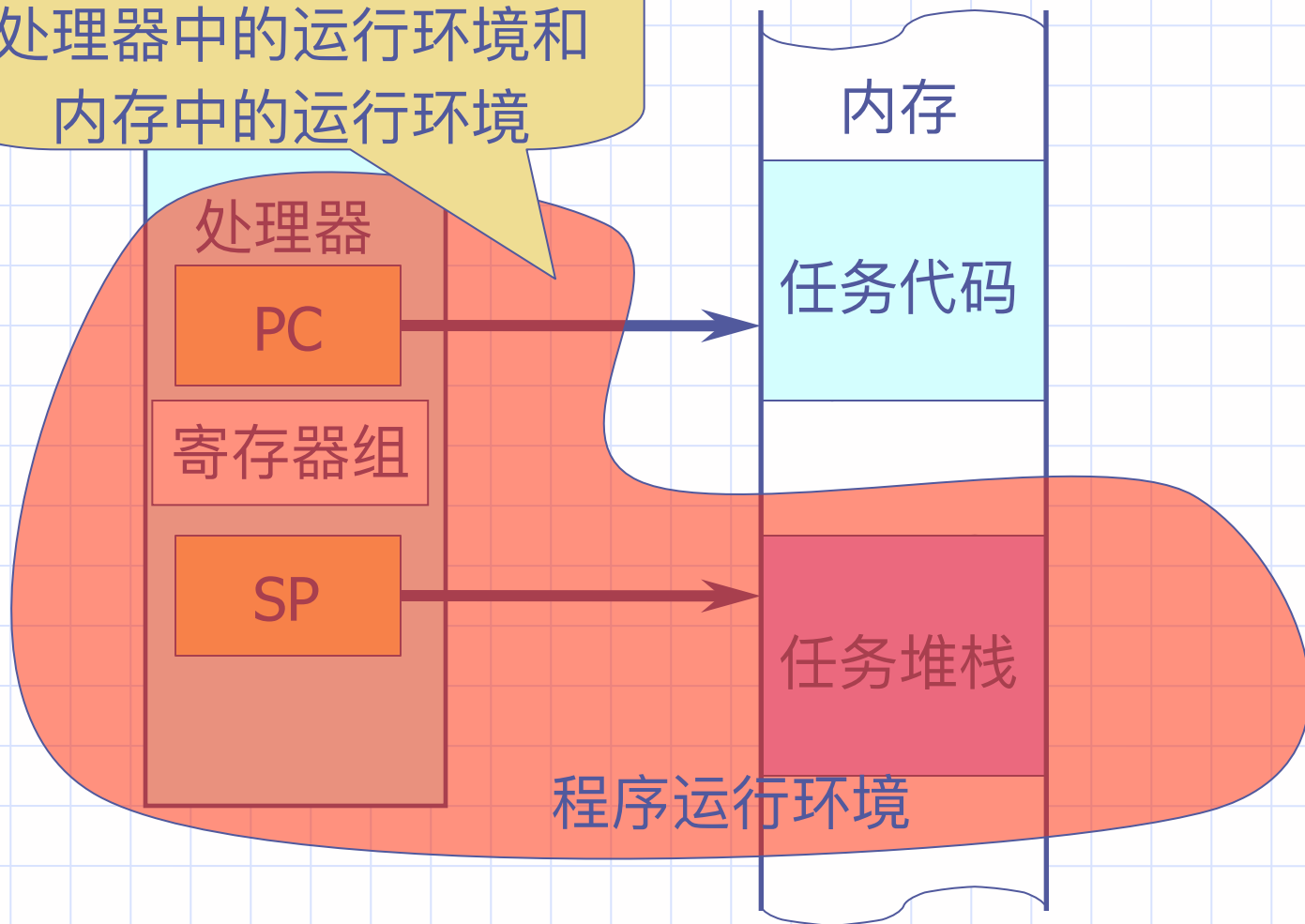


任务运行时与处理器之间的关系

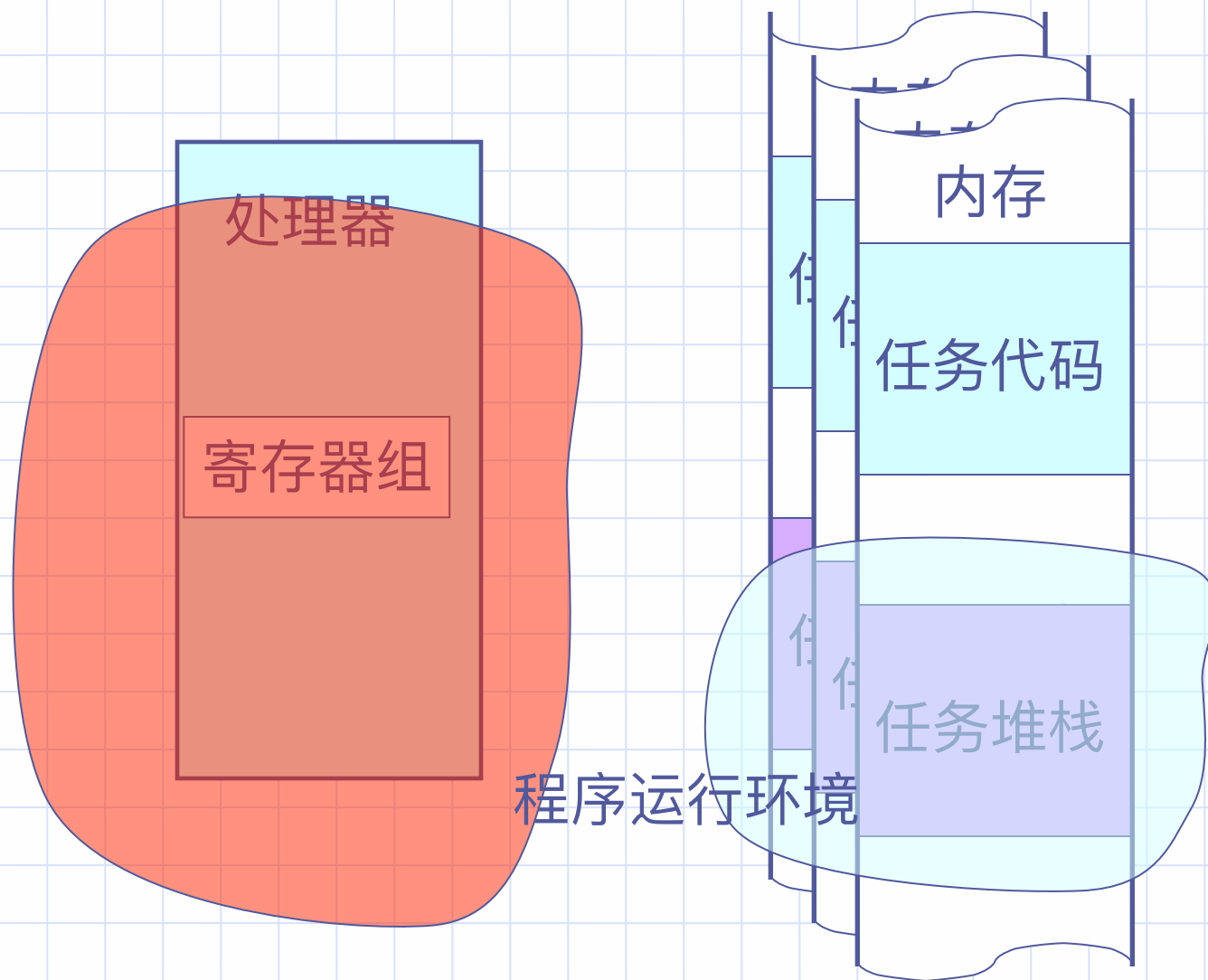


任务运行时与处理器之间的关系

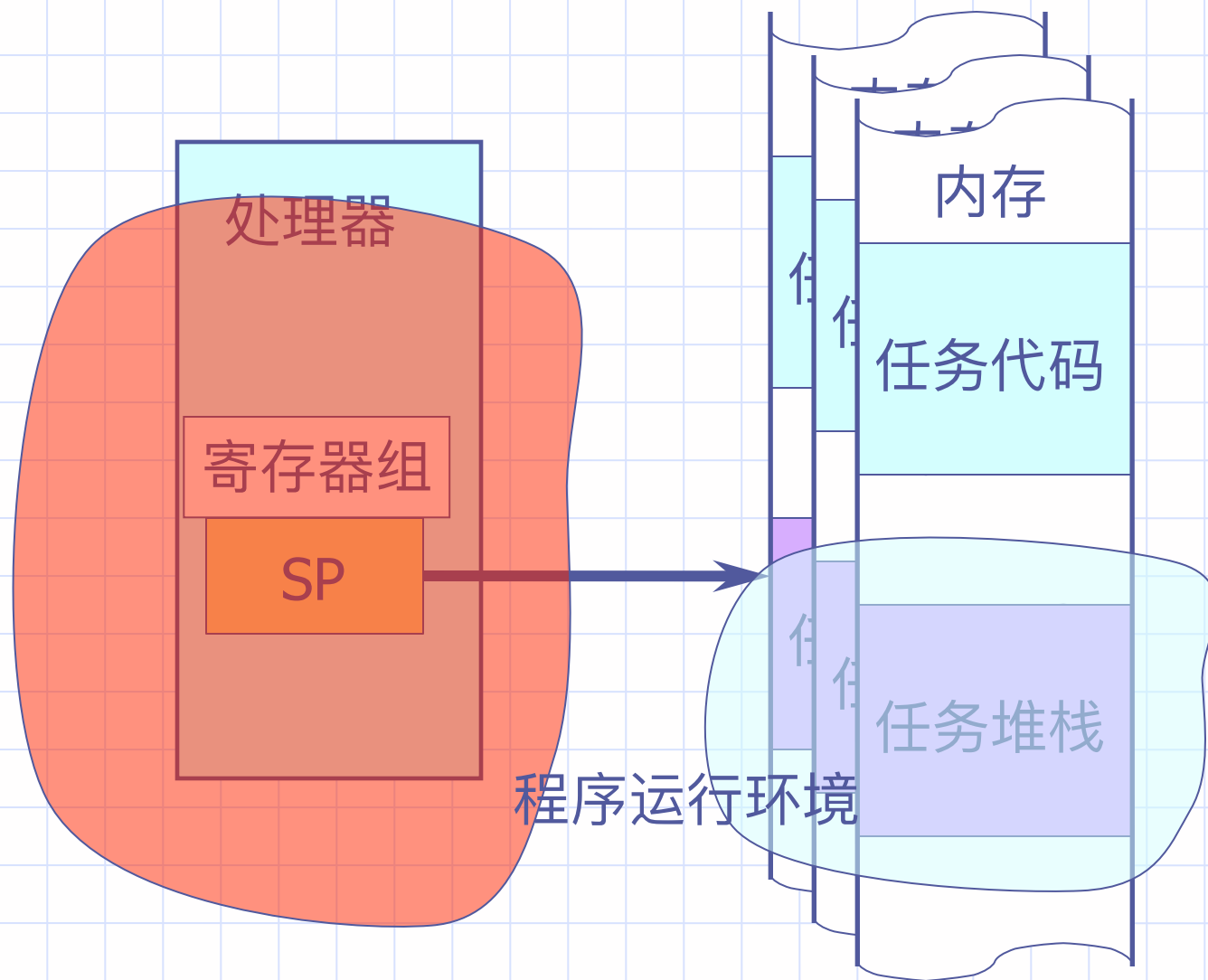
运行环境包括两部分：
处理器中的运行环境和
内存中的运行环境



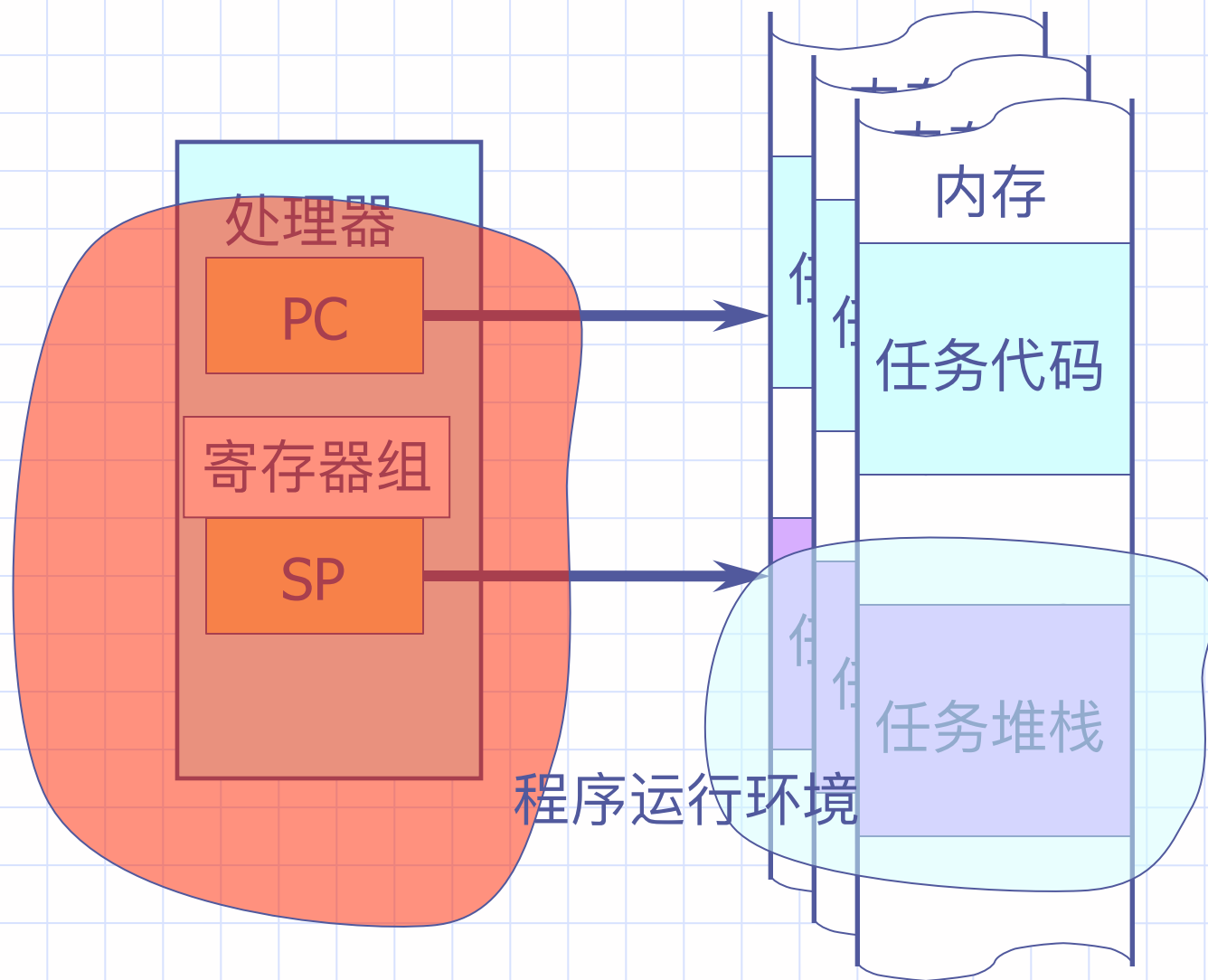
多任务时的问题



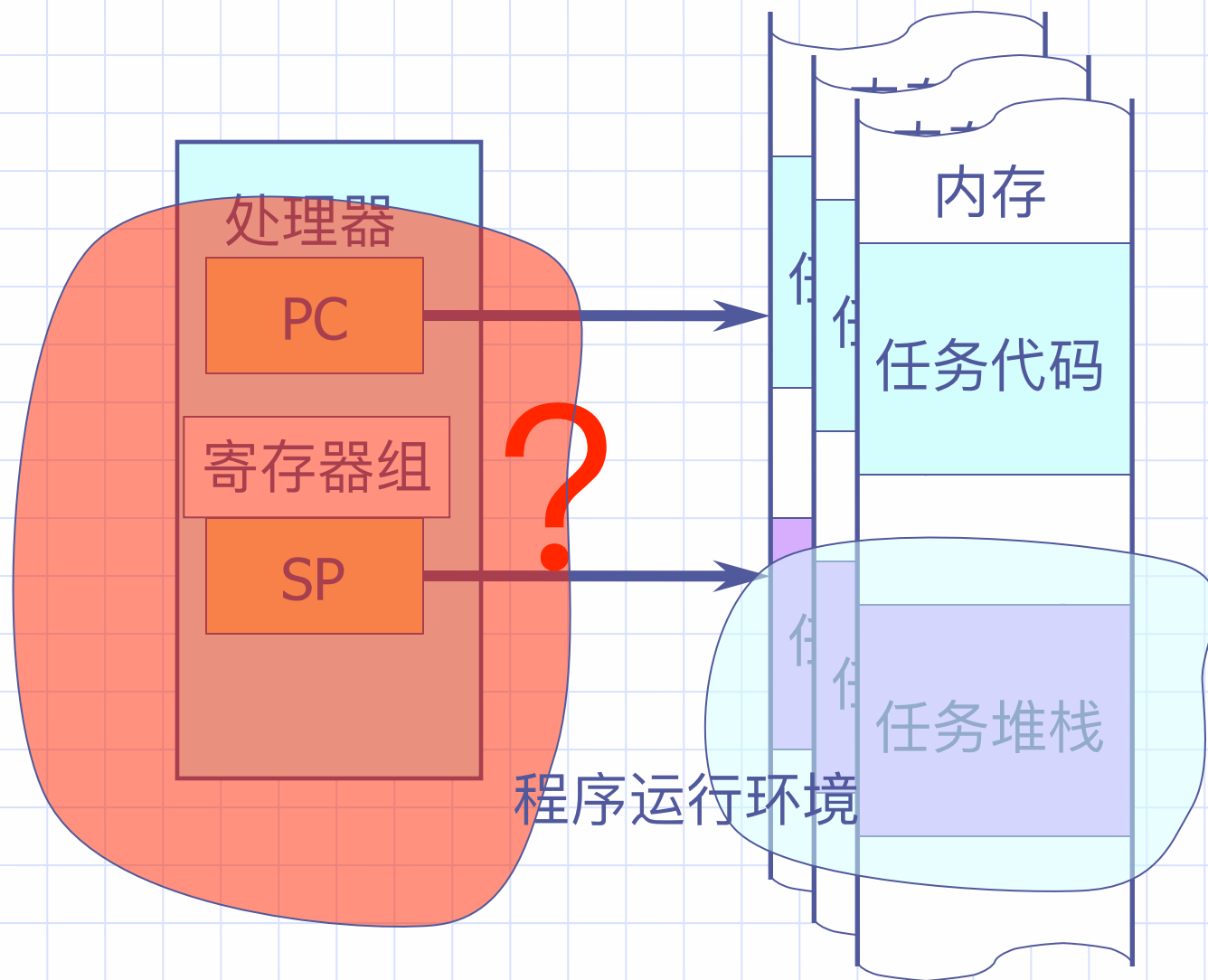
多任务时的的问题



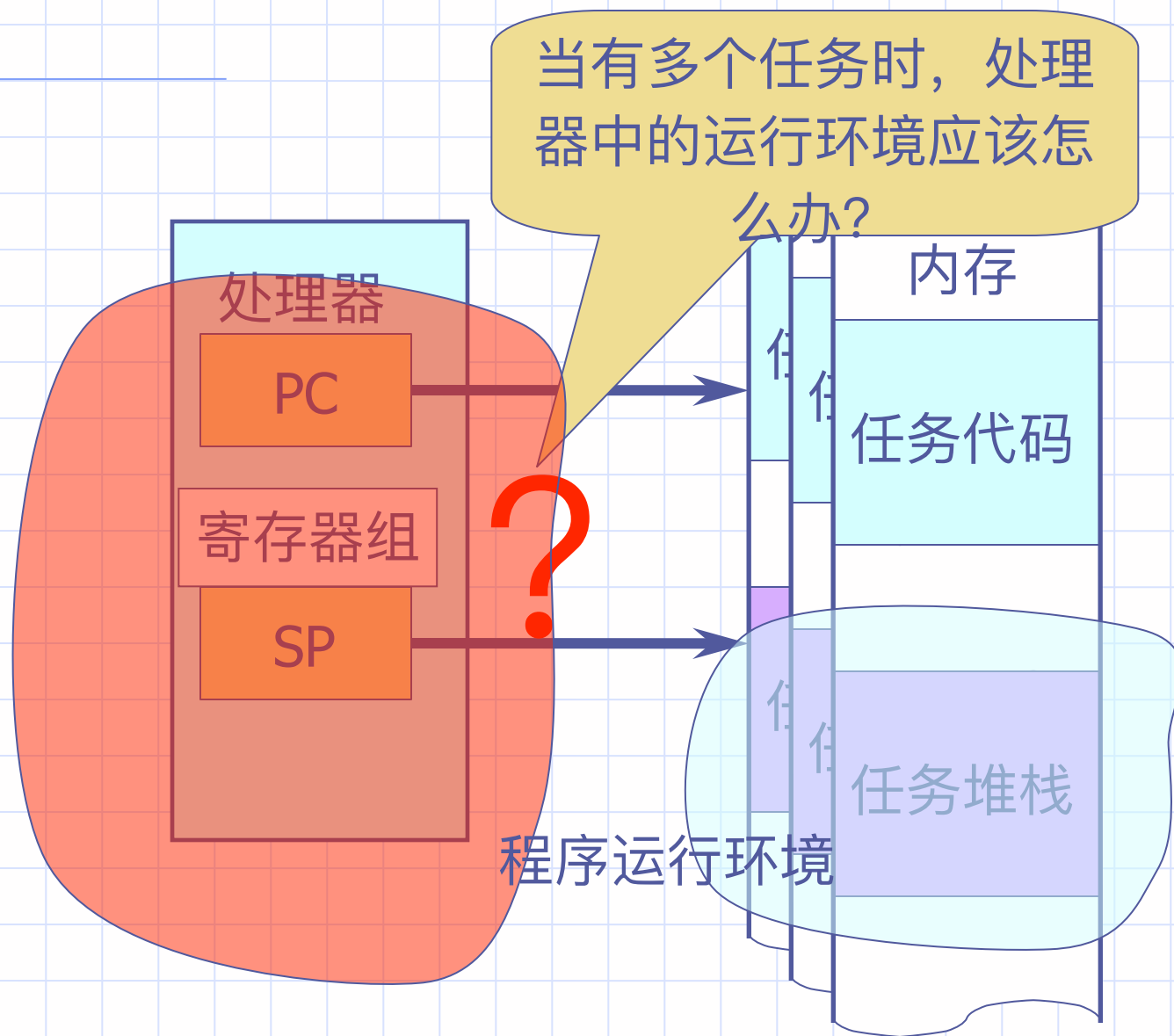
多任务时的的问题



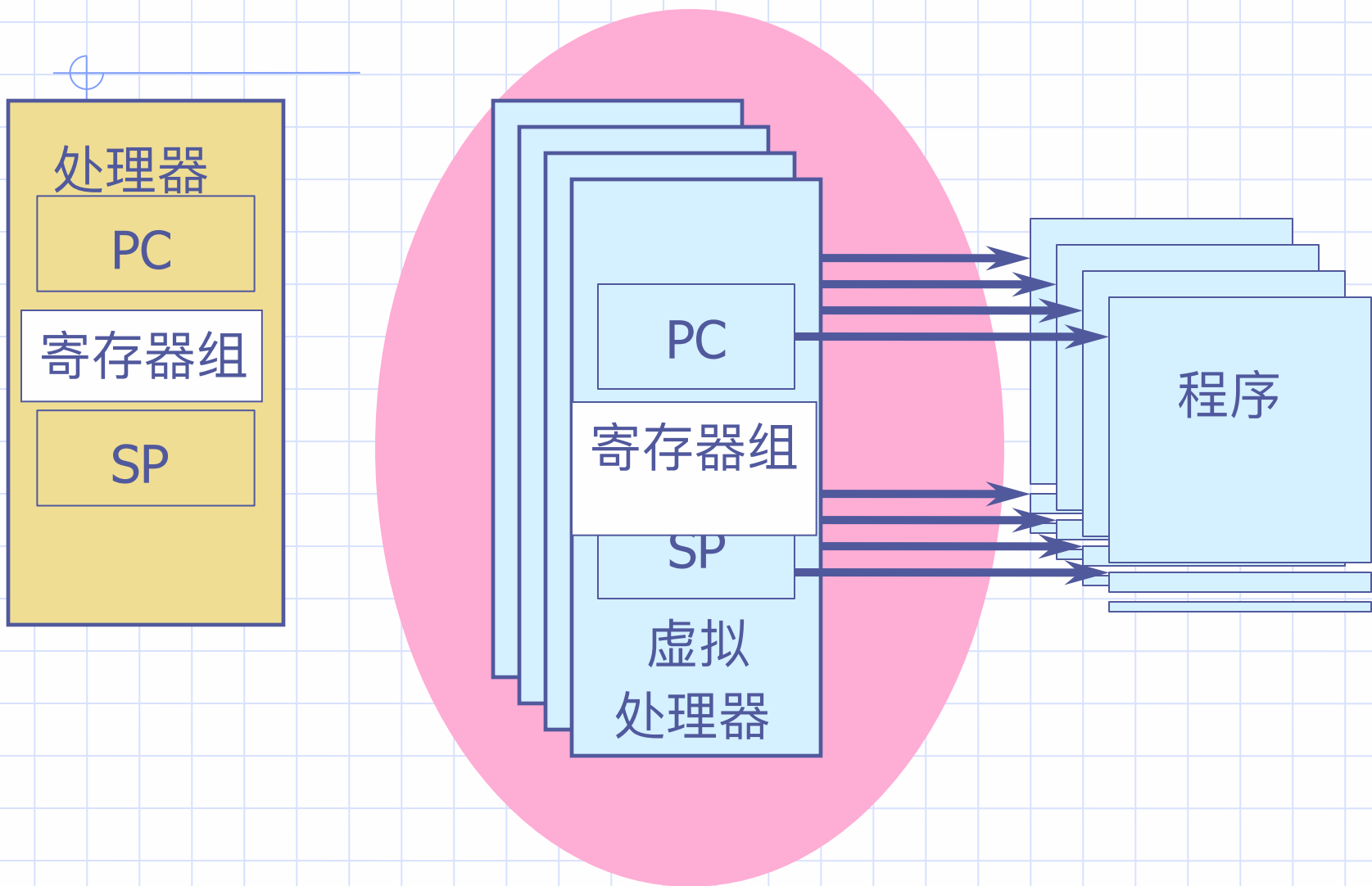
多任务时的的问题



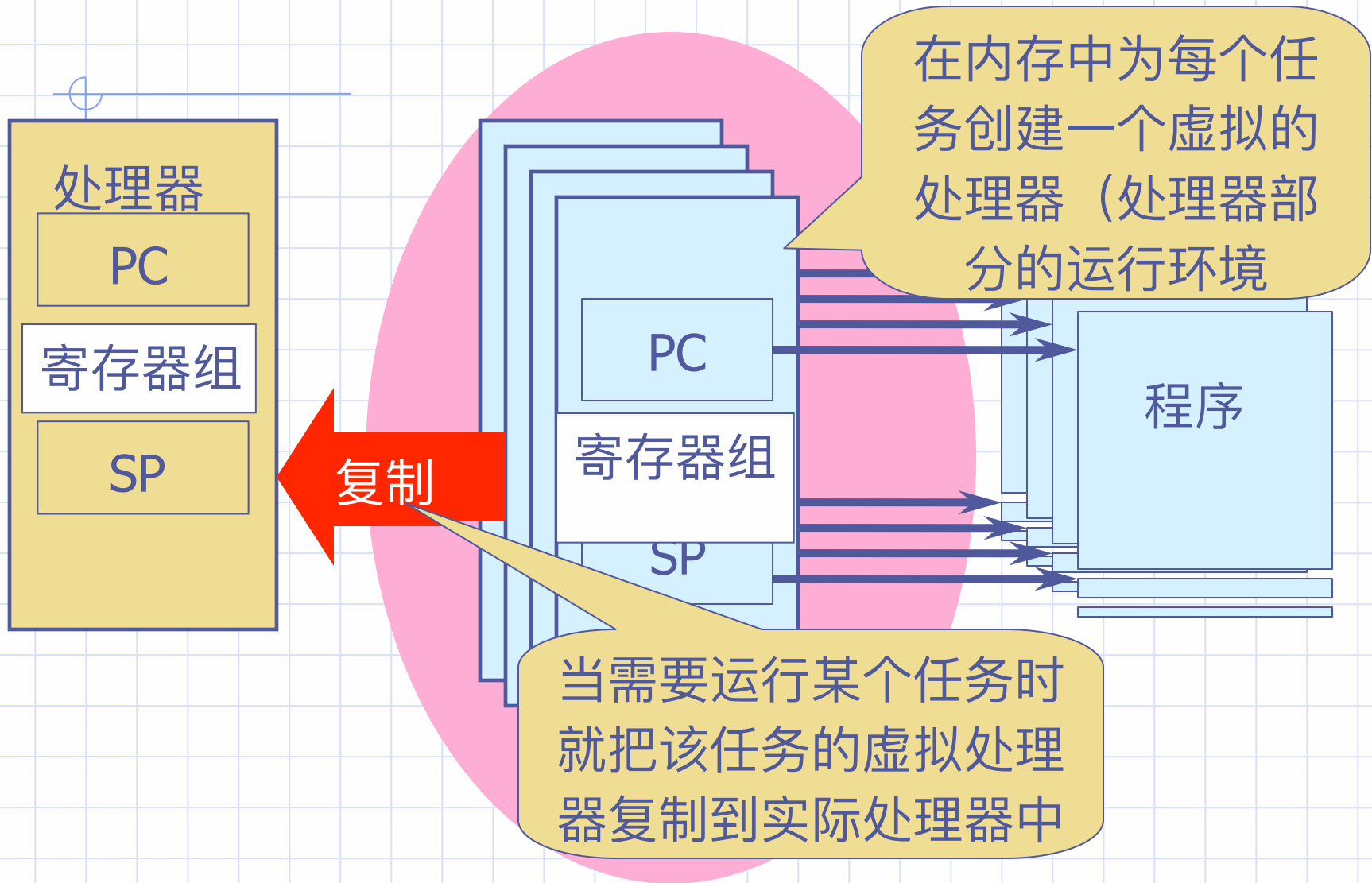
多任务时的的问题



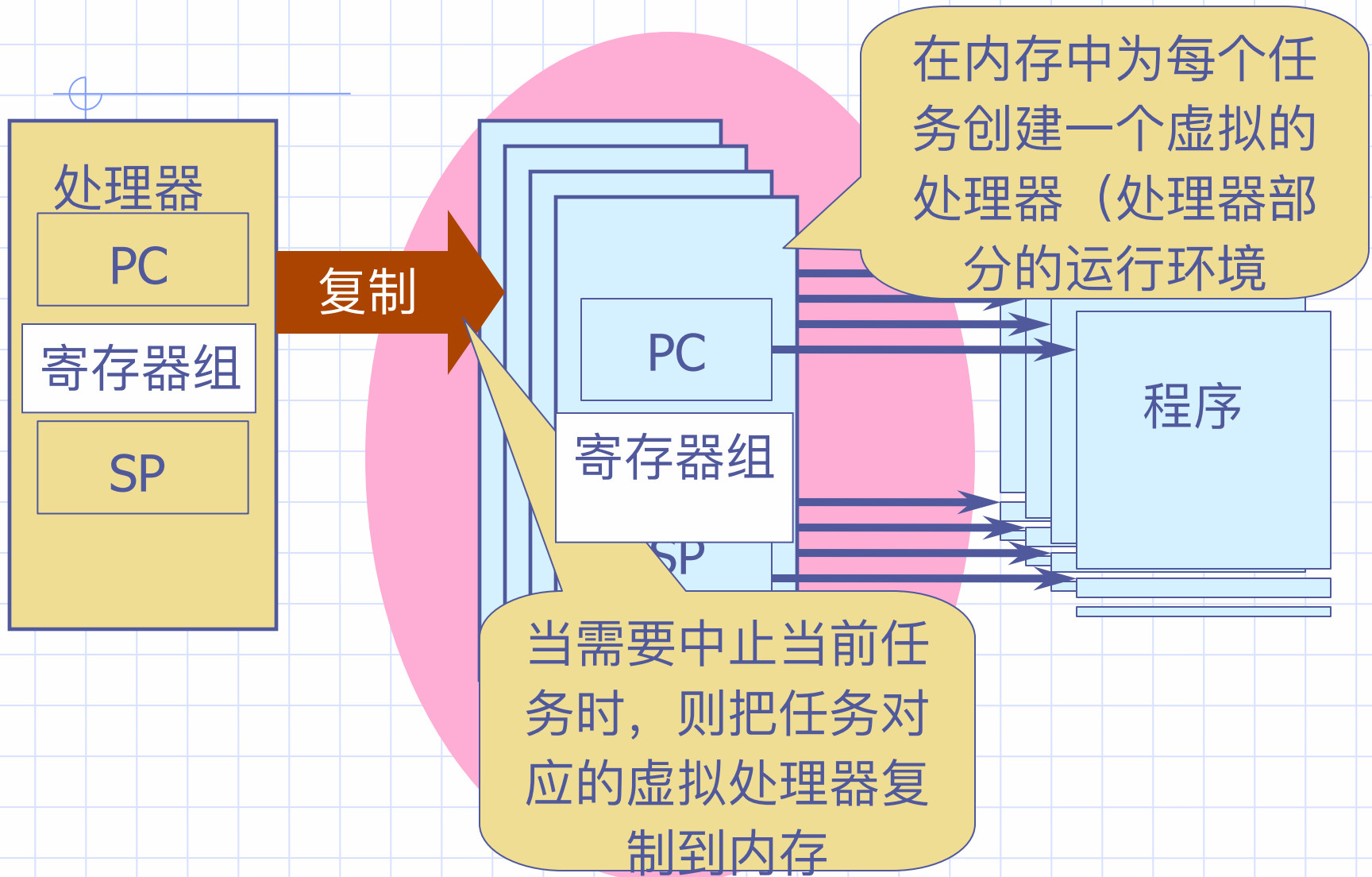
多任务的处理



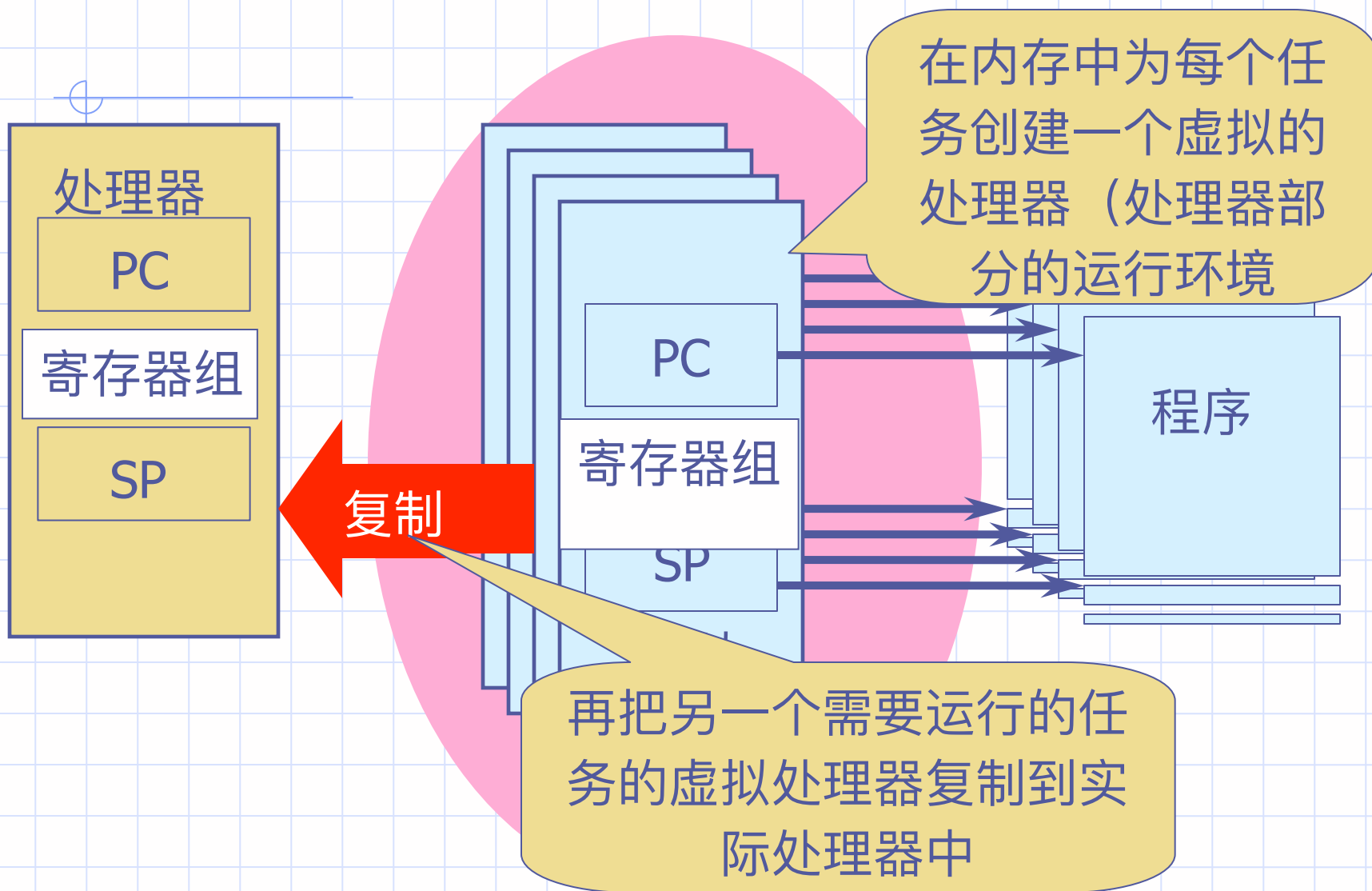
多任务的处理



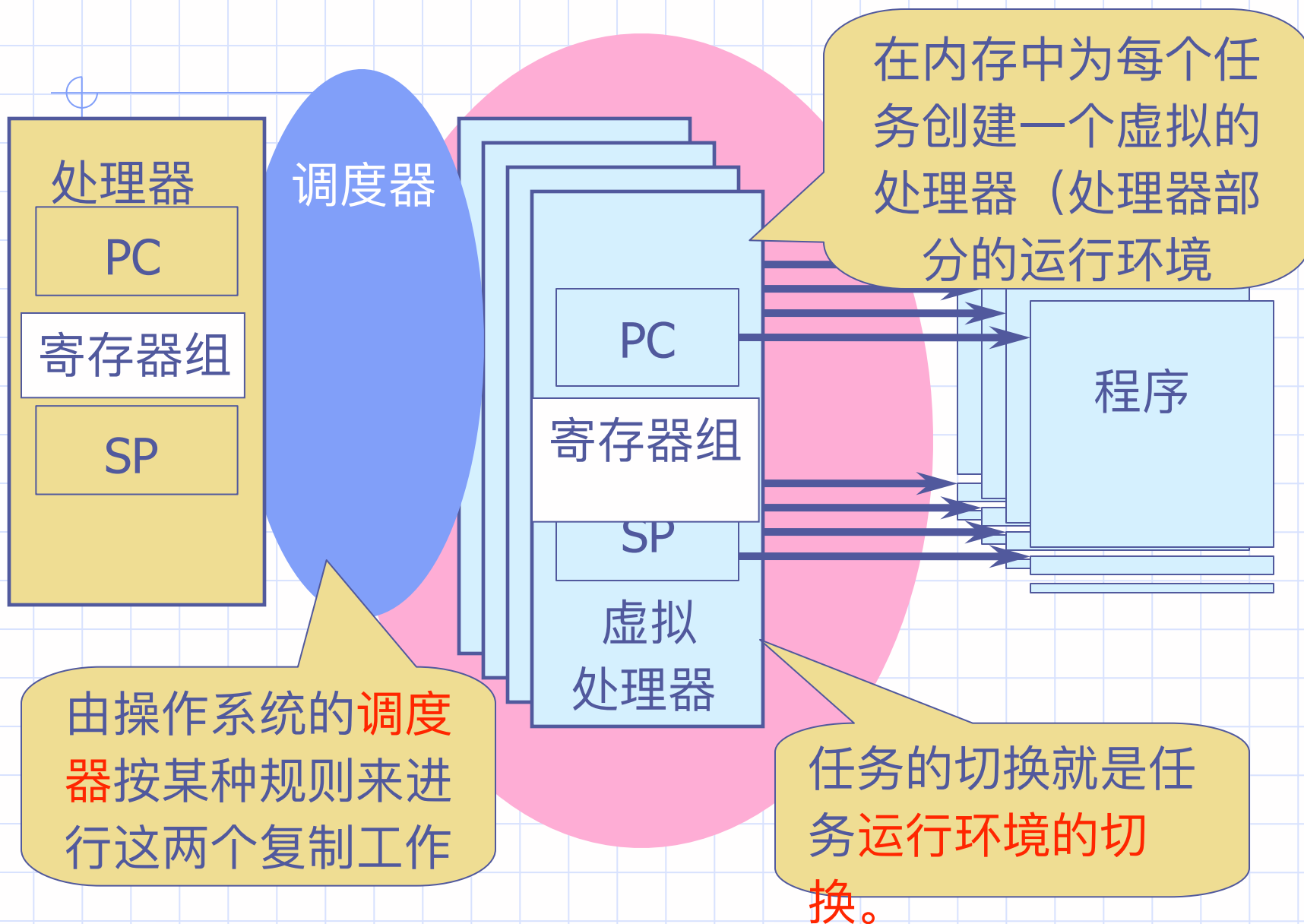
多任务的处理



多任务的处理



多任务的处理



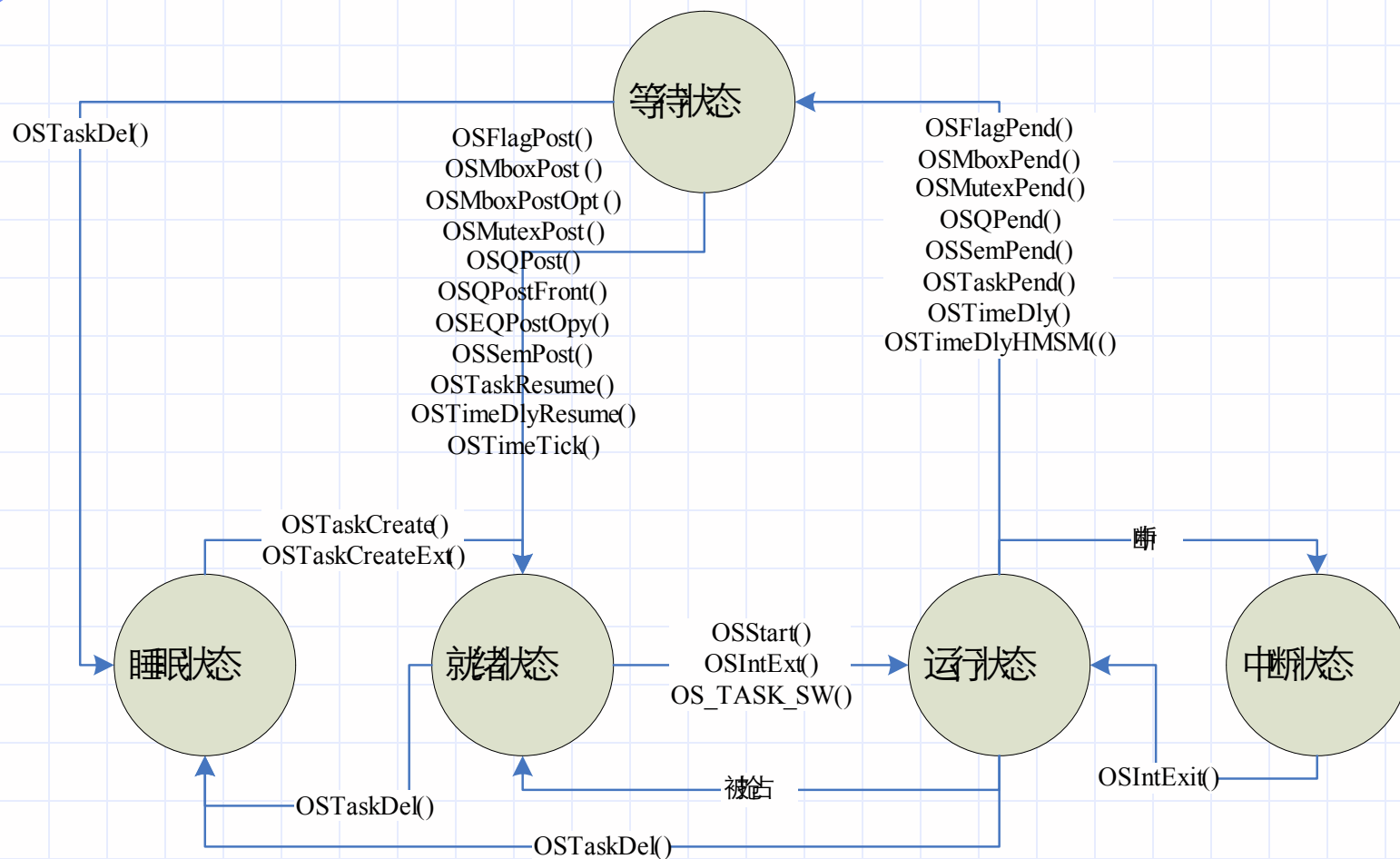
μC/OSII任务函数

- p μC/OSII的任务看起来像其它C的函数一样，有函数返回类型，有形式参数变量，但是任务是绝不会返回的，故返回参数必须定义成**void**。
- p 任务完成以后，任务可以自我删除，任务代码并非真的删除了，只是μC/OSII不再调度这个任务。

任务代码：任务完成后，可自我删除

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

μC/OSII任务状态转换图



任务运行环境

- 任务的运行环境包含以下主要信息：
 - 程序的断点地址 (PC)
 - 任务堆栈指针 (SP)
 - 程序状态字寄存器 (PSW)
 - 通用寄存器内容
 - 函数调用信息 (已存在于堆栈)
- 这些内容通常保存在任务堆栈中，它们也常叫做任务的上下文
- 任务堆栈指针(SP)保存在任务控制块中

任务控制块

- 任务控制块是由操作系统另行构造的一个数据结构，每个任务都有一个。它除了保存任务堆栈指针之外还要负责保存任务其他信息，例如任务的优先级、状态等。
- 具有控制块的程序才是一个可以被系统所运行的任务。
程序代码、私有堆栈、任务控制块是任务的三要素。

任务控制块

- 任务控制块是由操作系统另行构造的一个数据结构，每个任务都有一个。它除了保存任务堆栈指针之外还要负责保存任务其他信息，例如任务的优先级、状态等。
- 具有控制块的程序才是一个可以被系统所运行的任务。

程序代码 私有堆栈 任务控制块 任务的其他信息

```
typedef struct os_tcb {  
    OS_STK *OSTCBStkPtr;      //指向任务堆栈栈顶的指针  
    .....  
    INT8U    OSTCBStat; //任务的当前状态标志  
    INT8U    OSTCBPrio; //任务的优先级别  
    .....  
} OS_TCB;
```

任务切换过程

获得待运行任务的任务控制块

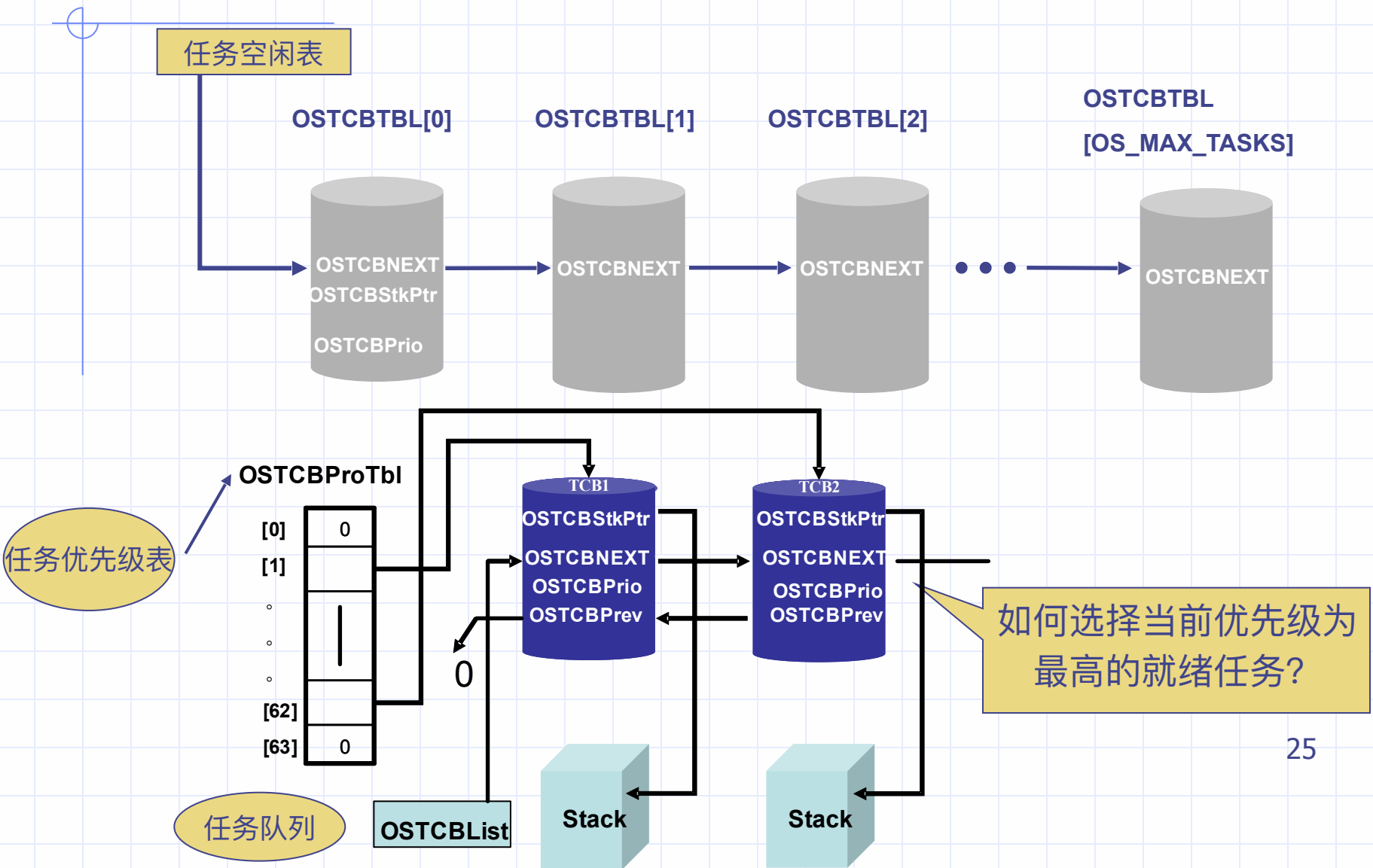
↓
处理器的SP=任务块中保存的SP

恢复待运行任务的运行环境

↓
处理器的PC=任务堆栈中的断点地址

如何获得待运行任务的任务控制块？（后面的任务调度中将会提到）

μC/OSII – 任务控制块 (TCB) 管理



μC/OSII – 独特的就绪表结构 (1/2)

p 就绪表 (Ready List)

✓ 两个变量

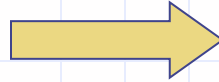
OSRdyGrp

OSRdyTbl []

✓ 两张映射表

OSMapTbl []

OSUnMapTbl []



Index	
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

μ C/OSII – 独特的就绪表结构 (2/2)

优先级为23、35、60的三个任务如何进入就绪表？

$$23 / 8 = 2$$

$$35 / 8 = 4$$

$$60 / 8 = 7$$

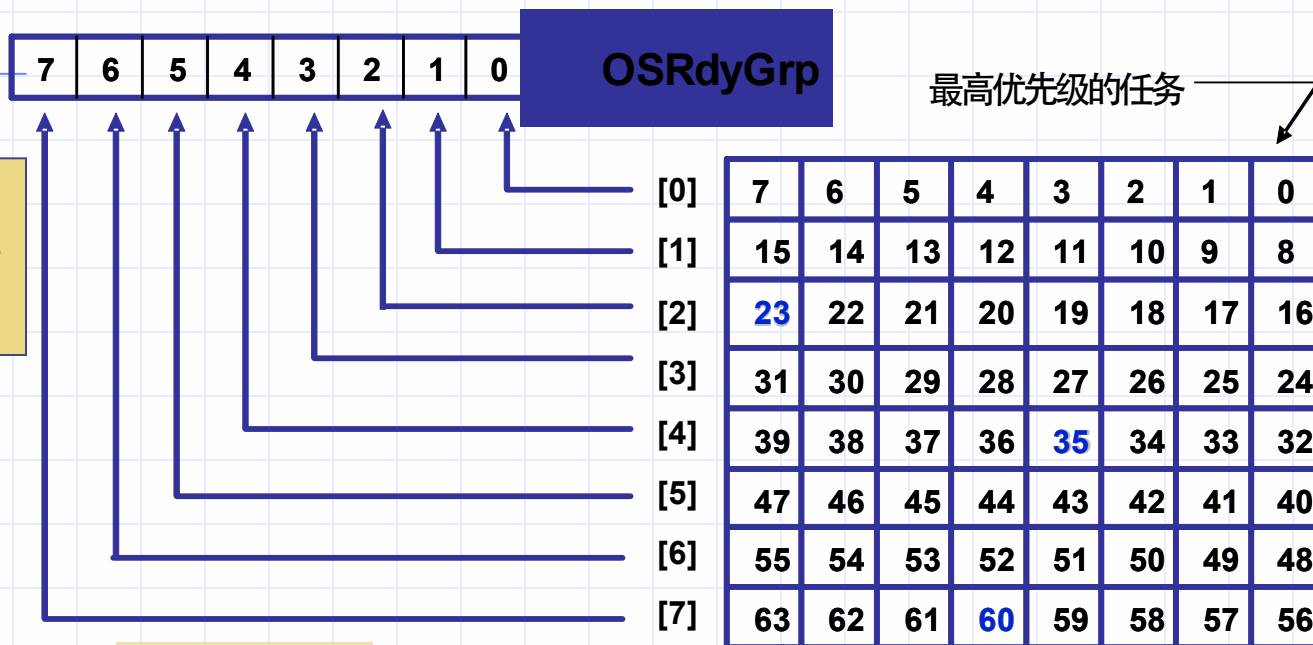
$$23 \% 8 = 7$$

$$35 \% 8 = 3$$

$$60 \% 8 = 4$$

OSMapTbl

OSRdyGrp = 10010100



最高优先级的任务

最低优先级的任务

OSRdyTbl[64]

μ C/OSII – 就绪表的操作算法

使任务进入
就绪态

```
OSRdyGrp |= OSMapTbl [ prio >> 3 ];  
OSRdyTbl [prio >> 3] |=  
OSMapTbl [prio & 0x07];
```

从就绪表中删
除一个任务

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

找出进入就绪
态优先级最高
的任务

```
y = OSUnMapTbl [OSRdyGrp];  
x = OSUnMapTbl [ OSRdyTbl [y] ];  
prio = ( y << 3 ) + x;
```

μC/OSII – 关于优先级表OSUnMapTbl

p OSUnMapTbl (优先级表)

```
INT8U const OSUnMapTbl[256] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x00 to 0x0F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x10 to 0x1F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x20 to 0x2F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x30 to 0x3F */
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x40 to 0x4F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x50 to 0x5F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x60 to 0x6F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x70 to 0x7F */
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x80 to 0x8F */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0x90 to 0x9F */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0xA0 to 0xAF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0xB0 to 0xBF */
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0xC0 to 0xCF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0xD0 to 0xDF */
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0xE0 to 0xEF */
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, /* 0xF0 to 0xFF */
};
```

- p 如果OSRdyGrp 的值为二进制01101000(0x68), 查OSUnMapTbl[0x68] = 3, 它相应于OSRdyGrp 中的第3位
- p 如果OSRdyTbl[3]的值是二进制11100100(0xe4), 则OSUnMapTbl[OSRdyTbc[3]] = 2
- p 任务的优先级 Prio = 3×8+2 = 26

μC/OSII – 任务调度器实例

```
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();
```

```
}
```

进入临界区

获取最高优先级

任务切换

退出临界区

小结



小结

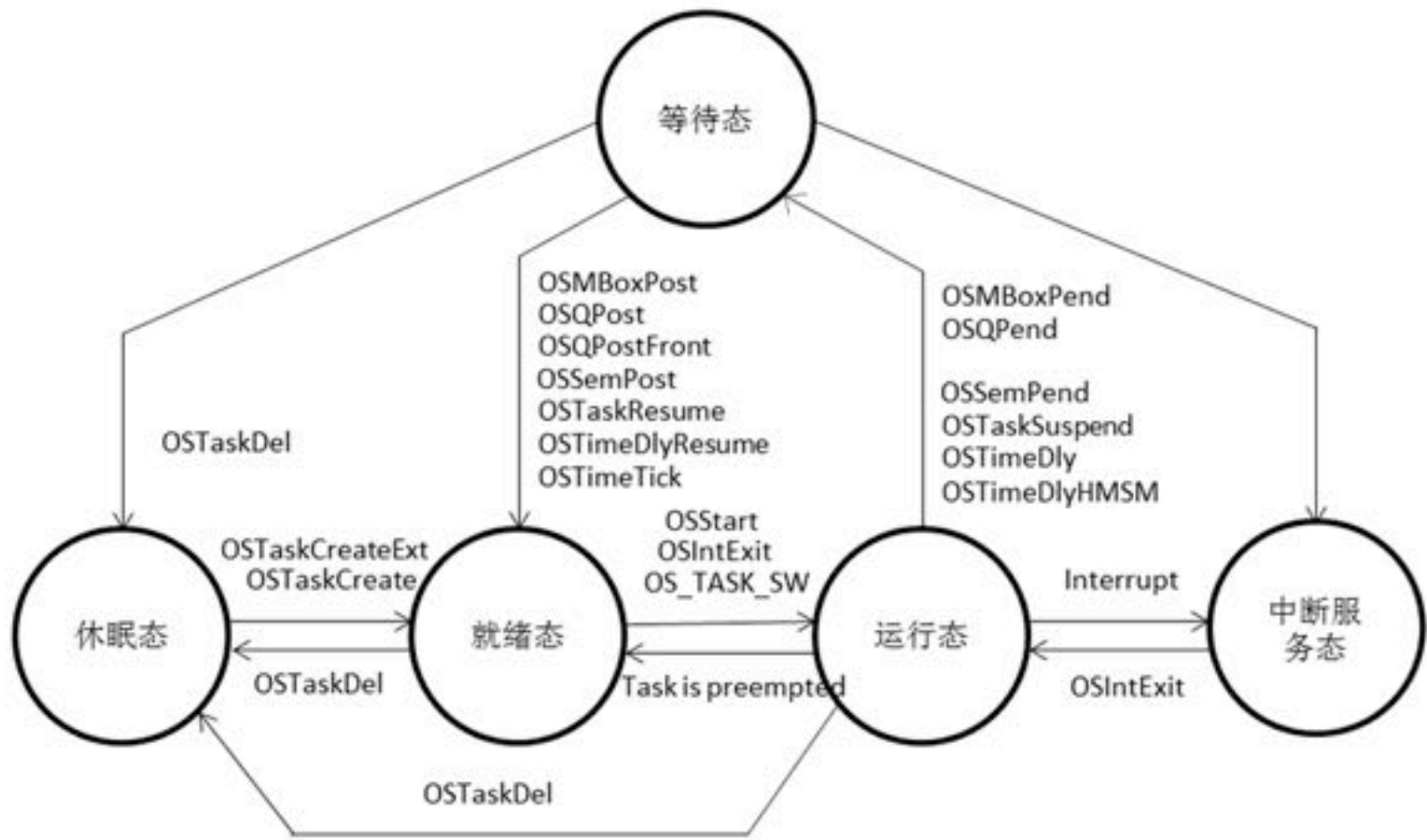
一个完整的任务应该有如下三部分：

- 任务代码（程序）
- 任务的私有堆栈（用以保护运行环境）
- 任务控制块（提供私有堆栈的位置）

这些都是任务方应该提供的基本信息。

μ C/OS-II的任务管理

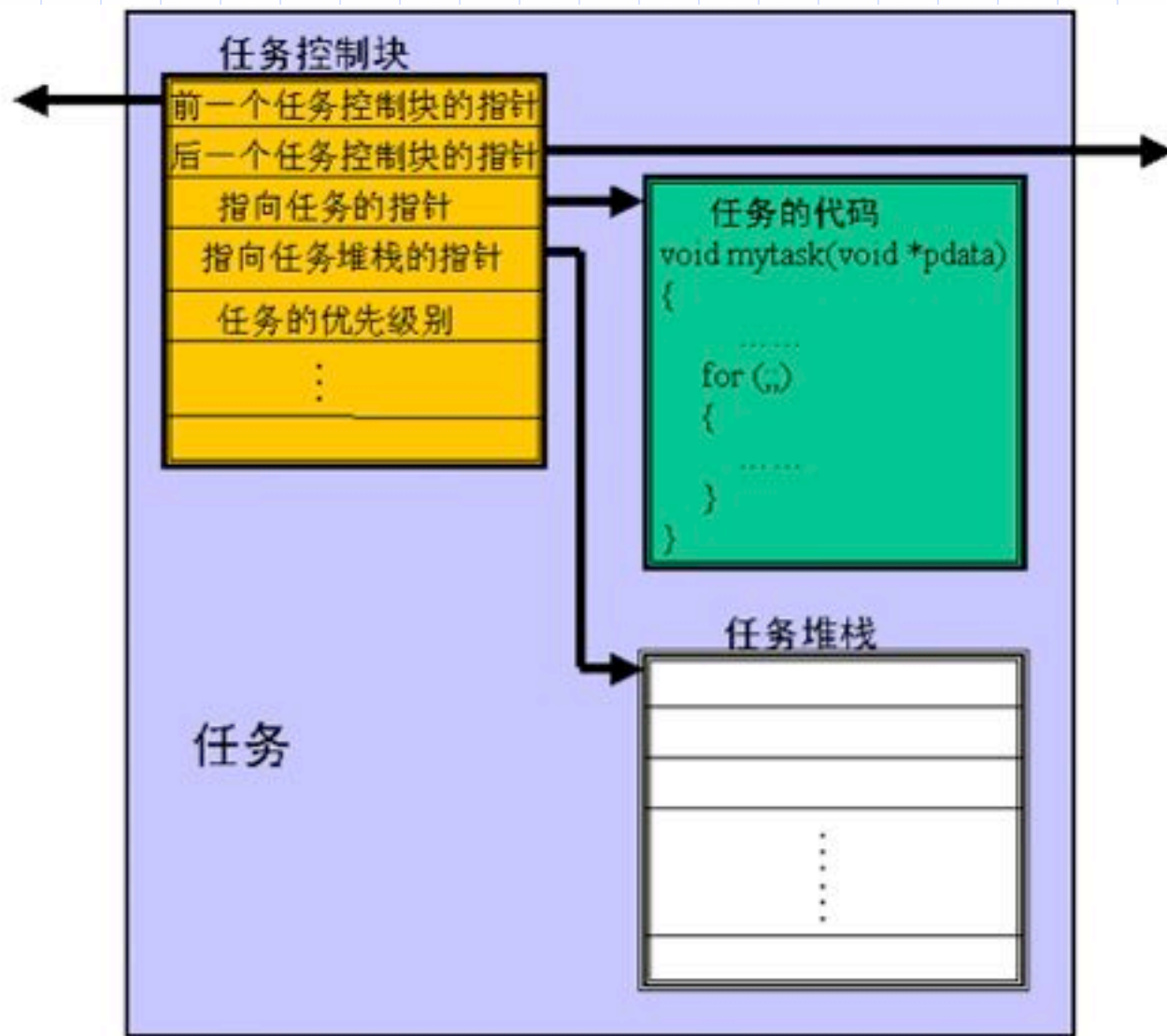
任务的状态



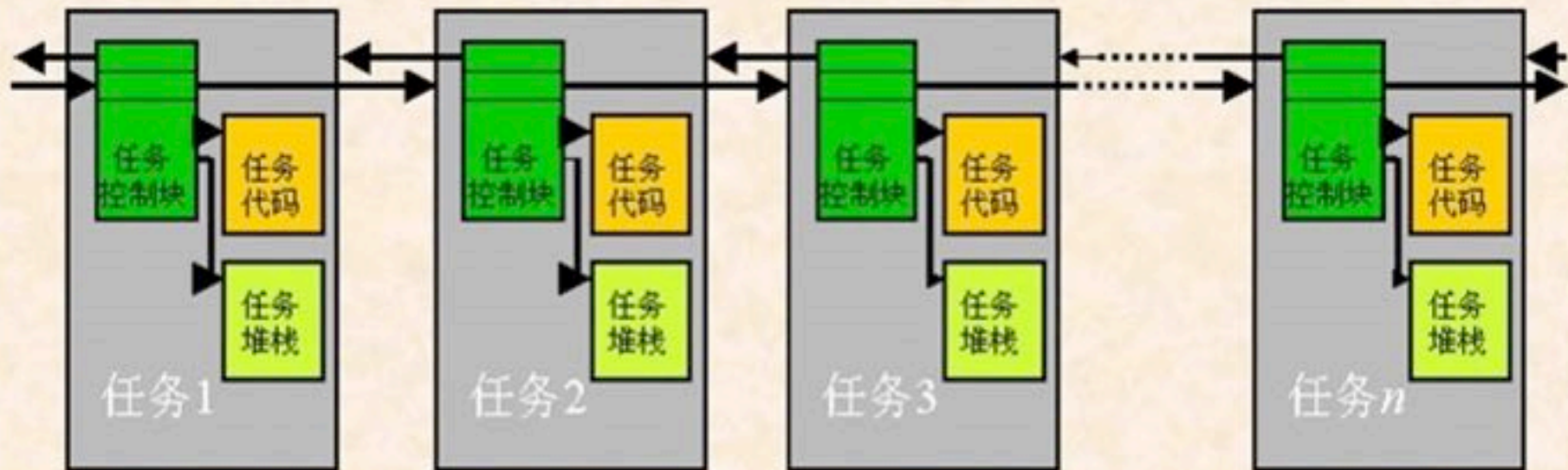
任务在内存中的结构



任务在内存中的结构



任务在内存中的结构



多个任务靠任务控制块组成了一个任务链表

用户任务代码的一般结构



用户任务代码的一般结构

```
void MyTask(void *pdata)
{
    for (;;)
    {
        可以被中断的用户代码;
        OS_ENTER_CRITICAL(); //进入临界段（关中断）
        不可以被中断的用户代码;
        OS_EXIT_CRITICAL();  //退出临界段（开中断）
        可以被中断的用户代码;
    }
}
```

用户任务代码的一般结构

```
void MyTask(void *pdata)
```

```
{
```

```
    for (;;)
    {
```

```
        可以被中断的用户代码;
```

```
        OS_ENTER_CRITICAL(); //进入临界段（关中断）
```

```
        不可以被中断的用户代码;
```

```
        OS_EXIT_CRITICAL(); //退出临界段（开中断）
```

```
        可以被中断的用户代码;
```

```
    }
```

```
}
```

用户任务代码的一般结构

```
void MyTask(void *pdata)
```

```
{
```

```
    for (;;)
    {
```

```
        可以被中断的用户代码;
```

```
        OS_ENTER_CRITICAL(); //进入临界段 (关中断)
```

```
        不可以被中断的用户代码;
```

```
        OS_EXIT_CRITICAL(); //退出临界段 (开中断)
```

```
        可以被中断的用户代码;
```

```
    }
```

```
}
```



无限循环

用户任务代码的一般结构

```
void MyTask(void *pdata)
```

```
{
```

```
    for (;;)
    {
```

```
        可以被中断的用户代码;
```

```
        OS_ENTER_CRITICAL();
```

```
        //进入临界段 (关中断)
```

```
        不可以被中断的用户代码;
```

```
        OS_EXIT_CRITICAL();
```

```
        //退出临界段 (开中断)
```

```
        可以被中断的用户代码;
```

```
    }
```

```
}
```

临界段

无限循环

用户任务代码的一般结构

```
void MyTask(void *pdata)
```

```
{
```

```
    for (;;)
    {
```

```
        可以被中断的用户代码;
```

```
        OS_ENTER_CRITICAL(); //进入临界段 (关中断)
```

```
        不可以被中断的用户代码;
```

```
        OS_EXIT_CRITICAL(); //退出临界段 (开中断)
```

```
        可以被中断的用户代码;
```

```
    }
```

```
}
```

临界段

无限循环

空闲任务

- 在多任务系统运行时，系统经常会在某个时间内无用户任务可运行而处于所谓的空闲状态，为了使CPU在没有用户任务可执行的时候有事可做， $\mu\text{C}/\text{OS-II}$ 提供了一个叫做空闲任务OSTaskIdle()的系统任务
- 在 $\mu\text{C}/\text{OS-II}$ 中，一个用户应用程序必须使用这个空闲任务，而且这个任务是不能用软件来删除的
- 空闲任务只是做了一个计数工作

空闲任务



空闲任务

```
void OSTaskIdle(void* pdata)
{
    # if OS_CRITICAL_METHOD == 3
        OS_CPU_SR cpu_sr;
    #endif

    pdata = pdata;           //防止某些编译器报错
    for(;;)
    {
        OS_ENTER_CRITICAL();//关闭中断
        OSdleCtr++;         //计数
        OS_EXIT_CRITICAL();  //开放中断
    }
}
```

统计任务

- μ C/OS-II提供的另一个系统任务是统计任务OSTaskStat()。这个统计任务每秒计算一次CPU在单位时间内被使用的时间，并把计算结果以百分比的形式存放在变量OSCPUUsage中，以便应用程序通过访问它来了解CPU的利用率，所以这个系统任务OSTaskStat()叫做统计任务

```

void OS_TaskStat (void *pdata)
{
#ifdef OS_CRITICAL_METHOD == 3           /* Allocate storage for CPU status register */
    OS_CPU_SR cpu_sr;
#endif
    INT32U run;
    INT32U max;
    INT8S usage;

    pdata = pdata;                       /* Prevent compiler warning for not using 'pdata' */
    while (OSSStatRdy == FALSE) {
        OSTimeDly(2 * OS_TICKS_PER_SEC); /* Wait until statistic task is ready */
    }
    max = OSIdleCtrMax / 100L;
    for (;;) {
        OS_ENTER_CRITICAL();
        OSIdleCtrRun = OSIdleCtr;        /* Obtain the of the idle counter for the past second */
        run = OSIdleCtr;
        OSIdleCtr = 0L;                  /* Reset the idle counter for the next second */
        OS_EXIT_CRITICAL();
        if (max > 0L) {
            usage = (INT8S)(100L - run / max);
            if (usage >= 0) {             /* Make sure we don't have a negative percentage */
                OSCPUUsage = usage;
            } else {
                OSCPUUsage = 0;
            }
        } else {
            OSCPUUsage = 0;
            max = OSIdleCtrMax / 100L;
        }
        OSTaskStatHook();                /* Invoke user definable hook */
        OSTimeDly(OS_TICKS_PER_SEC);     /* Accumulate OSIdleCtr for the next second */
    }
}

```

任务的优先级

- $\mu\text{C}/\text{OS-II}$ 把任务的优先权分为64个优先级别，每一个级别都用一个数字来表示。数字0表示任务的优先级别最高，数字越大则表示任务的优先级别越低
- 在文件OS_CFG.H中通过宏定义OS_LOWEST_PRIO来配置最低优先级。该常数一旦被定义，则意味着系统中可供使用的优先级别范围为0到OS_LOWEST_PRIO。
- 系统总是把最低优先级别OS_LOWEST_PRIO自动赋给空闲任务。如果还使用了统计任务，系统则会把优先级别OS_LOWEST_PRIO-1自动赋给统计任务，因此用户任务可以使用的优先级别范围为0到OS_LOWEST_PRIO-2

任务的堆栈

- 保存CPU寄存器中的内容及存储任务私有数据的需要，每个任务都应该配有自己的堆栈，任务堆栈是任务的重要组成部分
- 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。例如：

任务的堆栈

- 保存CPU寄存器中的内容及存储任务私有数据的需要，每个任务都应该配有自己的堆栈，任务堆栈是任务的重要组成部分
- 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。例如：

```
//定义堆栈的长度
```

```
#define      TASK_STK_SIZE    512
```

```
//定义一个数组来作为任务堆栈
```

```
OS_STK TaskStk[TASK_STK_SIZE];
```

任务的堆栈初始化

- 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
- $\mu\text{C}/\text{OS-II}$ 在创建任务函数`OSTaskCreate()`中通过调用任务堆栈初始化函数`OSTaskStkInit()`来完成任务堆栈初始化工作的
- 由于各种处理器的寄存器及对堆栈的操作方式不尽相同，因此该函数需要用户在进行 $\mu\text{C}/\text{OS-II}$ 的移植时，按所使用的处理器由用户来编写

树莓派移植的堆栈初始化函数

```
OS_STK *OSTaskStkInit (void (*task)(void *pd), void *p_arg, OS_STK *ptos, INT16U opt)
{
    OS_STK *stk;

    opt    = opt;           /* 'opt' is not used, prevent warning */
    stk    = ptos;          /* Load stack pointer */
    *(stk) = (OS_STK)task;   /* Entry Point */
    *(--stk) = (INT32U)0x14141414L; /* R14 (LR) */
    *(--stk) = (INT32U)0x12121212L; /* R12 */
    *(--stk) = (INT32U)0x11111111L; /* R11 */
    *(--stk) = (INT32U)0x10101010L; /* R10 */
    *(--stk) = (INT32U)0x09090909L; /* R9 */
    *(--stk) = (INT32U)0x08080808L; /* R8 */
    *(--stk) = (INT32U)0x07070707L; /* R7 */
    *(--stk) = (INT32U)0x06060606L; /* R6 */
    *(--stk) = (INT32U)0x05050505L; /* R5 */
    *(--stk) = (INT32U)0x04040404L; /* R4 */
    *(--stk) = (INT32U)0x03030303L; /* R3 */
    *(--stk) = (INT32U)0x02020202L; /* R2 */
    *(--stk) = (INT32U)0x01010101L; /* R1 */
    *(--stk) = (INT32U)p_arg; /* R0 : argument */
    *(--stk) = (INT32U)ARM_SYS_MODE; /* CPSR (Enable both IRQ and FIQ interrupts) */

    return (stk);
}
```

任务调度

- 多任务操作系统的核心工作就是任务调度。所谓调度，就是通过一个算法在多个任务中确定该运行的任务，做这项工作的函数就叫做调度器。
- $\mu\text{C}/\text{OS-II}$ 进行任务调度的思想是“近似地每时每刻总是让优先级最高的就绪任务处于运行状态”。为了保证这一点，它在系统或用户任务调用系统函数及执行中断服务程序结束时总是调用调度器，来确定应该运行的任务并运行它。 $\mu\text{C}/\text{OS-II}$ 进行任务调度的依据就是任务就绪表。

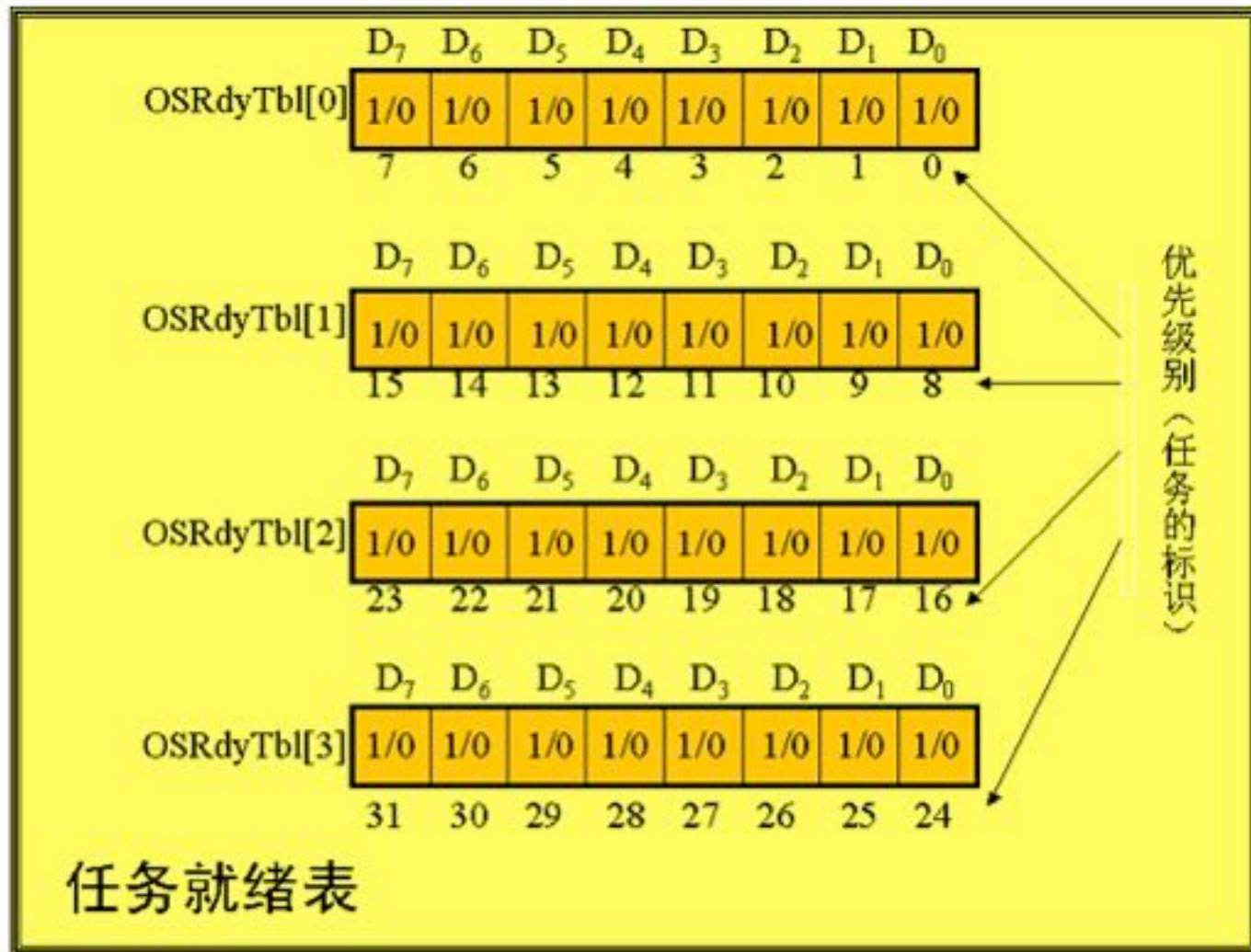
任务就绪表

- 为了能够使系统清楚地知道，系统中哪些任务已经就绪，哪些还没有就绪， $\mu\text{C}/\text{OS-II}$ 在内存中设立了一个记录表，系统中的每个任务都在这个表中占据一个位置，并用这个位置的状态（1或者0）来表示任务是否处于就绪状态，这个表就叫做任务就绪状态表，简称叫任务就绪表

任务就绪表



任务就绪表



任务就绪表就是一个二维数组OSRdyTbl[]

任务就绪表

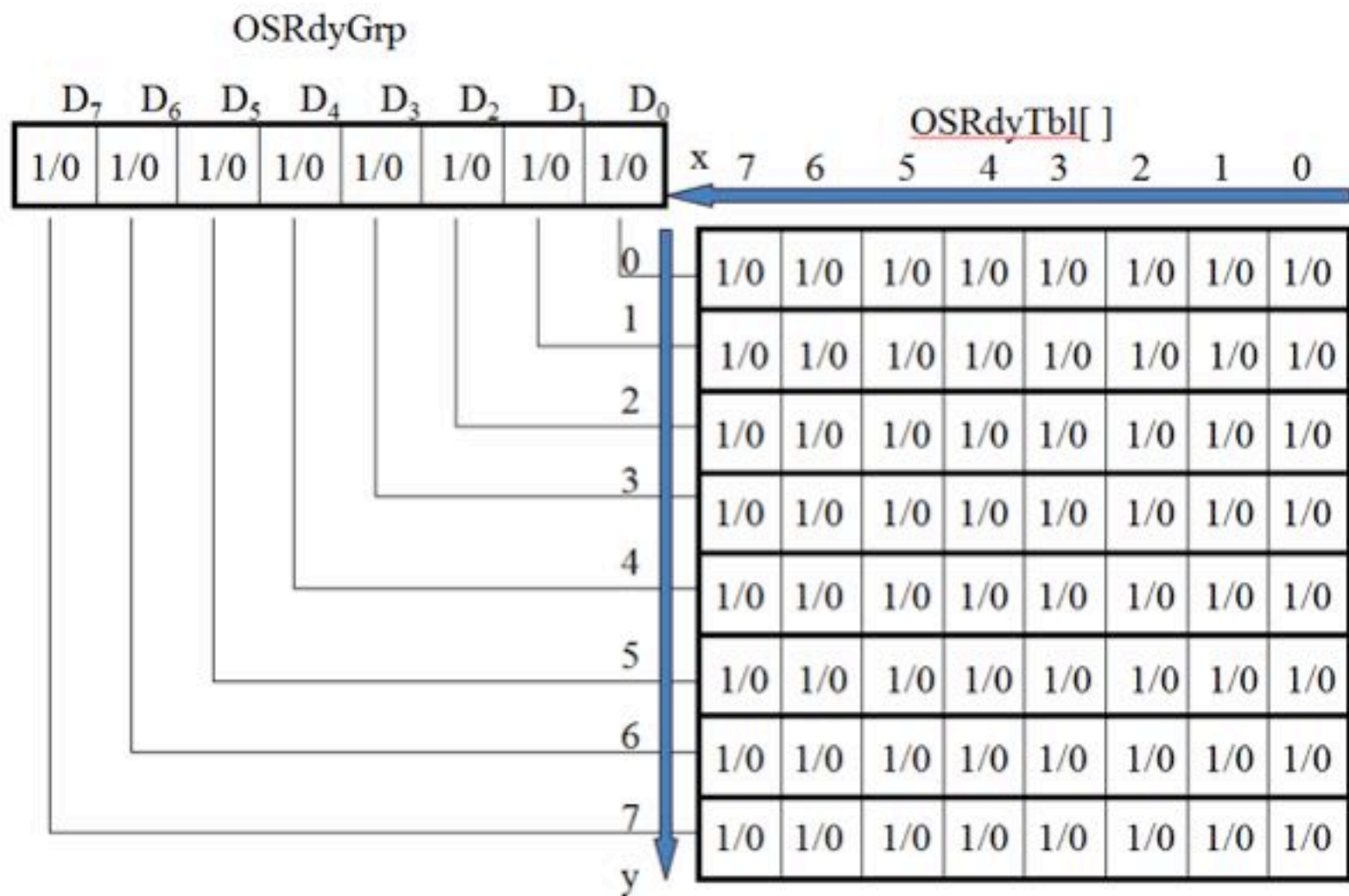
OSRdyGrp							
D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1/0	1/0	1/0	1/0	1/0	1/0	1/0	1/0

为加快访问任务就绪表的速度，系统定义了一个变量OSRdyGrp来表明就绪表每行中是否存在就绪任务。

- 该位为1表明OSRdyTbl[0]组有任务就绪
- 该位为1表明OSRdyTbl[1]组有任务就绪
- 该位为1表明OSRdyTbl[2]组有任务就绪
- 该位为1表明OSRdyTbl[3]组有任务就绪
- 该位为1表明OSRdyTbl[4]组有任务就绪
- 该位为1表明OSRdyTbl[5]组有任务就绪
- 该位为1表明OSRdyTbl[6]组有任务就绪
- 该位为1表明OSRdyTbl[7]组有任务就绪

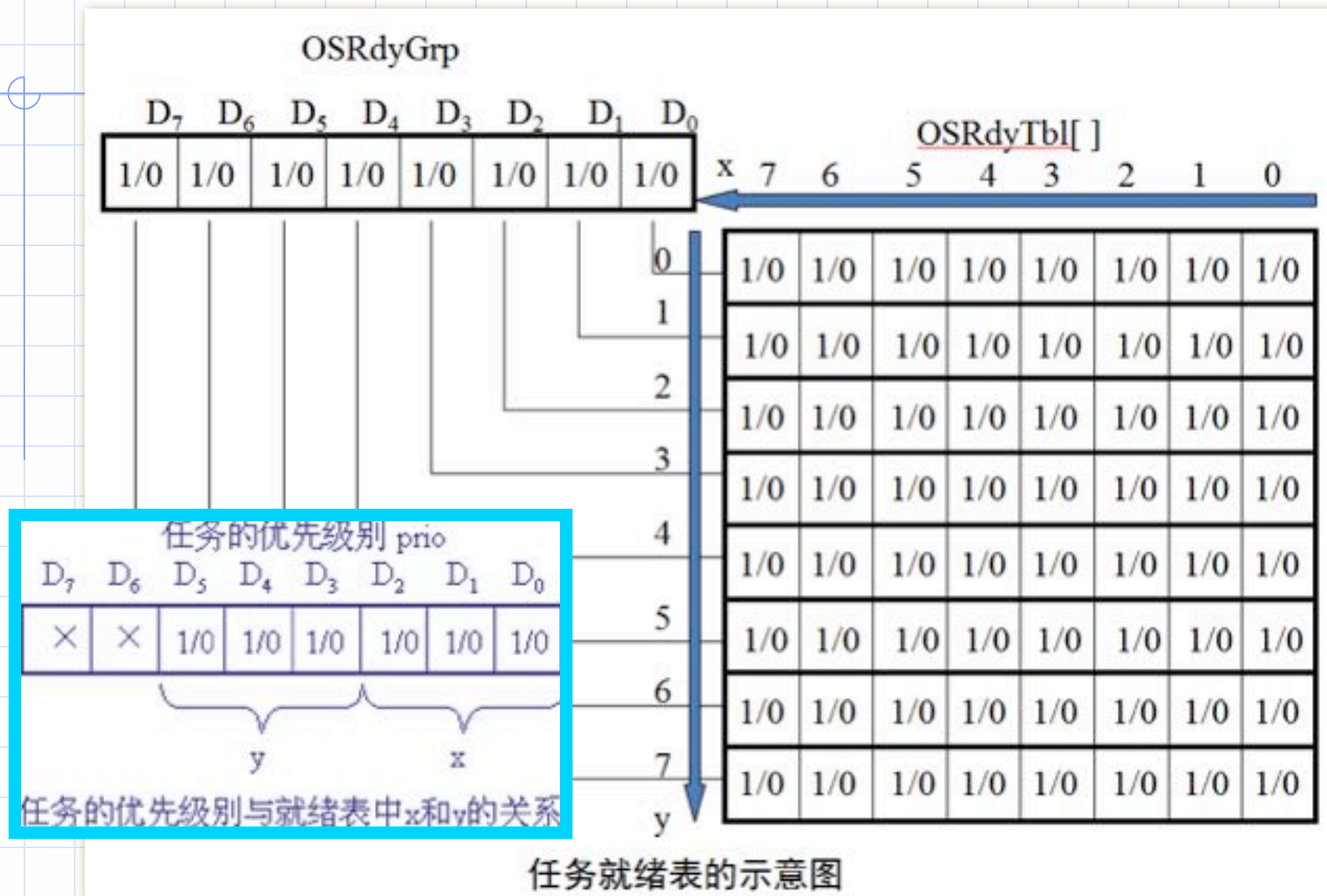
变量OSRdyGrp的格式及含义

任务就绪表

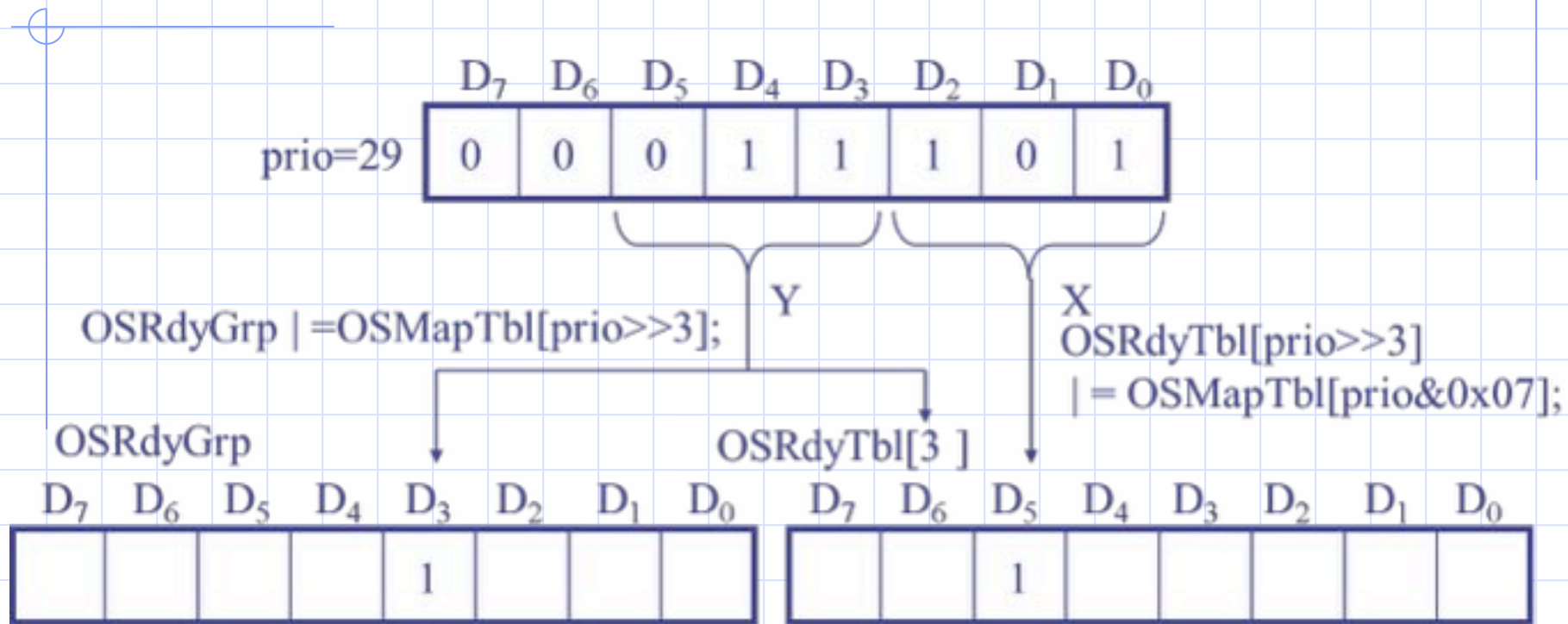


任务就绪表的示意图

任务就绪表



任务就绪表



把prio为29的任务置为就绪状态

$\text{OSMapTbl}[] = \{0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80\};$

任务就绪表

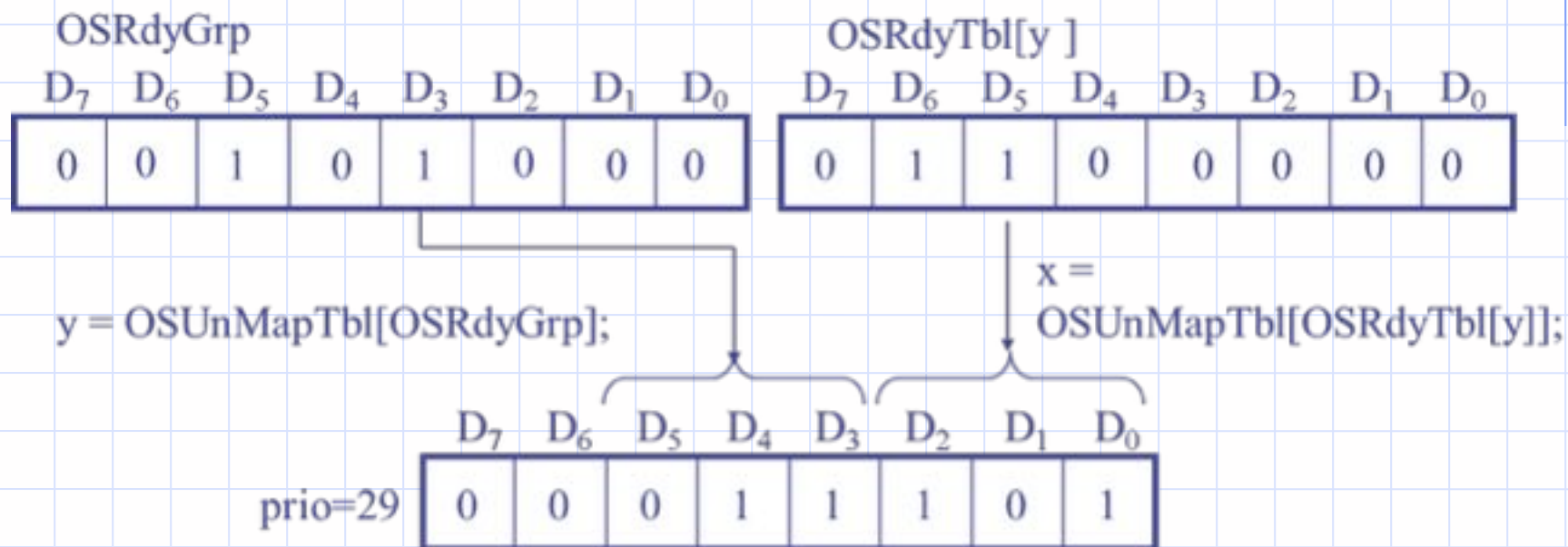
- 可以用类似下面的代码把优先级为prio的任务置为就绪状态：

```
OSRdyGrp |= OSMapTbl[prio>>3];  
OSRdyTbl[prio>>3] |= OSMapTbl[prio&0x07];
```

- 如果要使一个优先级为prio的任务脱离就绪状态则可使用如下类似代码：

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

任务就绪表



在就绪表中查找最高优先级别任务的过程

任务就绪表

- 优先级判定表OSUnMapTbl[256] (os_core.c)

```
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
};
```

例如：

如果OSRdyGrp的值为00101000B，即0X28，则查得OSUnMapTbl[OSRdyGrp]的值是3，它相应于OSRdyGrp中的第3位置1；

如果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

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

任务就绪表

- 从任务就绪表中获取优先级别最高的就绪任务可用如下类似的代码：

```
y = OSUnMapTal[OSRdyGrp]; //D5、D4、D3位  
x = OSUnMapTal[OSRdyTbl[y]]; //D2、D1、D0位  
prio = (y<<3)+x;           //优先级别
```

任务就绪表

根据**就绪表**获得待运行任务的**任务控制块指针**

如何获得待运行任务的任务控制块？

↓
处理器的SP=任务块中保存的SP

恢复待运行任务的运行环境

↓
处理器的PC=任务堆栈中的断点地址

任务切换函数

- 任务切换就是中止正在运行的任务（当前任务），转而去运行另外一个任务的操作，当然这个任务应该是就绪任务中优先级别最高的那个任务
- μ C/OS-II中通过调用任务切换函数OSCtxSw完成任务上下文切换

树莓派移植的任务切换函数

OSCtxSw

```
                                ; SAVE CURRENT TASK'S CONTEXT
                                ;   Return address
STR    LR, [SP, #-4]!
STR    LR, [SP, #-4]!
STR    R12, [SP, #-4]!
STR    R11, [SP, #-4]!
STR    R10, [SP, #-4]!
STR    R9, [SP, #-4]!
STR    R8, [SP, #-4]!
STR    R7, [SP, #-4]!
STR    R6, [SP, #-4]!
STR    R5, [SP, #-4]!
STR    R4, [SP, #-4]!
STR    R3, [SP, #-4]!
STR    R2, [SP, #-4]!
STR    R1, [SP, #-4]!
STR    R0, [SP, #-4]!
MRS    R4, CPSR                ;   push current CPSR
STR    R4, [SP, #-4]!

LDR    R4, ??OS_TCBCur        ; OSTCBCur->OSTCBStkPtr = SP;
LDR    R5, [R4]
STR    SP, [R5]

BL     OSTaskSwHook            ; OSTaskSwHook();
```

```

LDR    R4, ??OS_PrioCur          ; OSPrioCur = OSPrioHighRdy
LDR    R5, ??OS_PrioHighRdy
LDRB   R6, [R5]
STRB   R6, [R4]

LDR    R4, ??OS_TCBCur          ; OSTCBCur = OSTCBHighRdy;
LDR    R6, ??OS_TCBHighRdy
LDR    R6, [R6]
STR    R6, [R4]

LDR    SP, [R6]                  ; SP = OSTCBHighRdy->OSTCBStkPtr;

; RESTORE NEW TASK'S CONTEXT
;   pop new task's CPSR
LDR    R4, [SP], #4
MSR    CPSR_cxsf, R4
;   pop new task's context
LDR    R0, [SP], #4
LDR    R1, [SP], #4
LDR    R2, [SP], #4
LDR    R3, [SP], #4
LDR    R4, [SP], #4
LDR    R5, [SP], #4
LDR    R6, [SP], #4
LDR    R7, [SP], #4
LDR    R8, [SP], #4
LDR    R9, [SP], #4
LDR    R10, [SP], #4
LDR    R11, [SP], #4
LDR    R12, [SP], #4
LDR    LR, [SP], #4
LDR    PC, [SP], #4

```

μC/OS-II的初始化

- 在使用μC/OS-II的所有服务之前，必须要调用μC/OS-II的初始化函数OSInit()对μC/OS-II自身的运行环境进行初始化
- 函数OSInit()将对μC/OS-II的所有的全局变量和数据结构进行初始化，同时创建空闲任务OSTaskIdle，并赋之以最低的优先级别和永远的就绪状态。如果用户应用程序还要使用统计任务的话
(OS_TASK_STAT_EN=1)，则OSInit()还要以优先级别为OS_LOWEST_PRIO-1来创建统计任务

μC/OS-II的启动

- μC/OS-II进行任务的管理是从调用启动函数OSStart()开始的，前提条件是在调用该函数之前至少创建了一个用户任务。在OSStart中会调用OSStartHighRdy来启动第一个任务

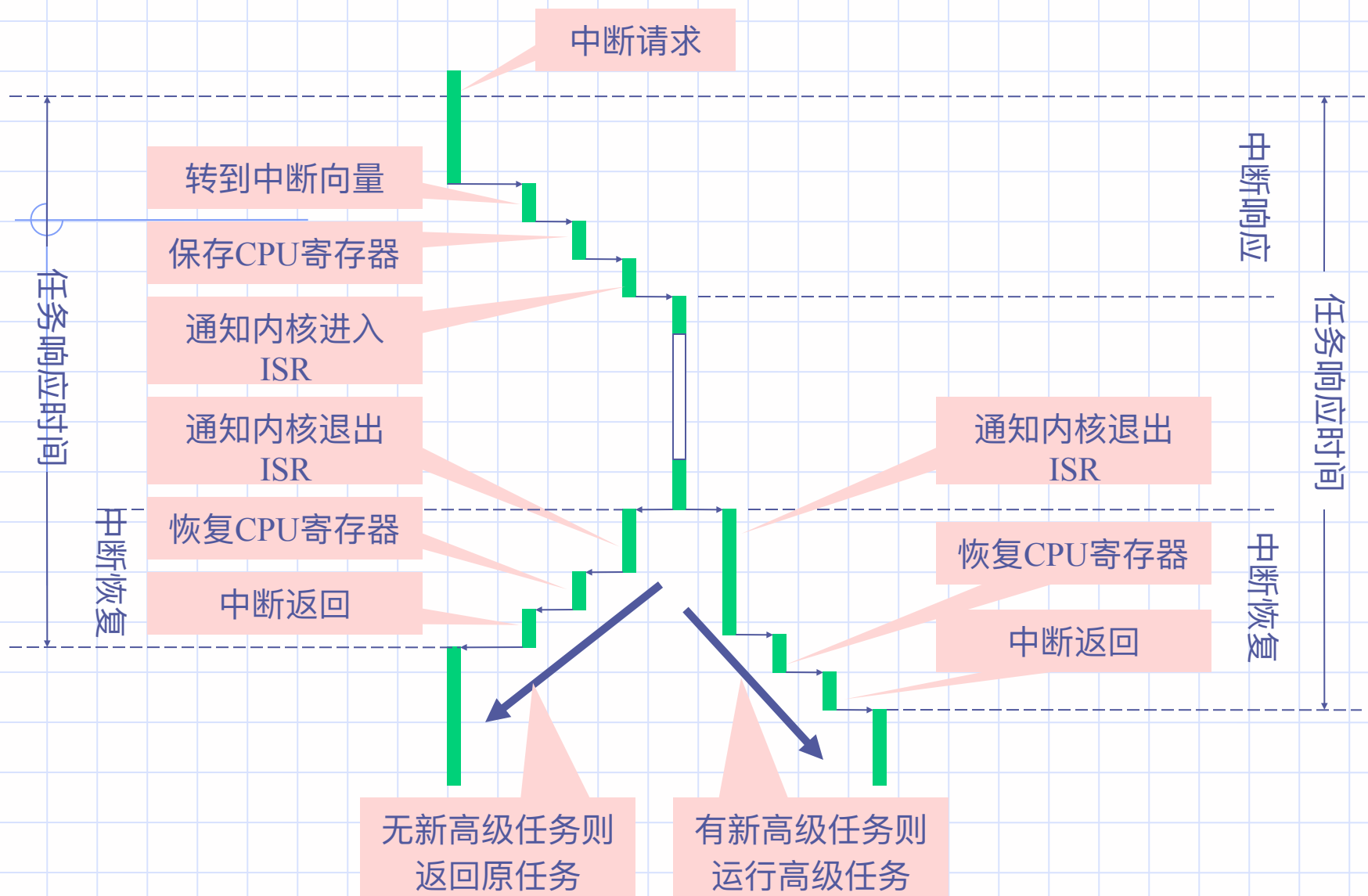
```
void main(void)
{
    .....           //在这个位置安装并启动μC/OS-II的时钟

    OSInit( );       //对μC/OS-II进行初始化
    .....
    OSTaskCreate (TaskStart,.....); //创建任务TaskStart
    OSStart( );      //开始多任务调度
}
void TaskStart(void*pdata)
{
    .....           //在这个位置创建其他任务
    for(;;)
    {
        .....       起始任务TaskStart的代码
    }
}
```

$\mu\text{C}/\text{OS-II}$ 的中断和时钟

响应中断的过程

- $\mu\text{C}/\text{OS-II}$ 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。
- 中断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，而不是一定要接续运行被中断的任务的。



中断的响应过程

OSIntEnter和OSIntExit

- OSIntEnter: 进入中断之前被调用, 将中断嵌套层次加一
- OSIntExit: 离开中断之后被调用, 将中断嵌套层次减一并且切换到当前就绪的最高优先级任务

```
void OSIntEnter (void)
{
    if (OSRunning == TRUE)
    {
        if (OSIntNesting < 255)
        {
            OSIntNesting++; //中断嵌套层数计数器加一
        }
    }
}
```

OSIntEnter和OSIntExit

```
void OSIntExit (void)
{
    #if OS_CRITICAL_METHOD == 3
        OS_CPU_SR cpu_sr;
    #endif
    if (OSRunning == TRUE)
    {
        OS_ENTER_CRITICAL();
        if (OSIntNesting > 0)
        {
            OSIntNesting--;    //中断嵌套层数计数器减一
        }
        if ((OSIntNesting == 0) && (OSLockNesting == 0))
        {
            OSIntExitY = OSUnMapTbl[OSRdyGrp];
            OSPrioHighRdy = (INT8U)((OSIntExitY << 3) + OSUnMapTbl[OSRdyTbl[OSIntExitY]]);
            if (OSRdyHighRdy != OSPrioCur)
            {
                OSTCBHighRdy = OSTCBPrioTbl[OSRdyHighRdy];
                OSCtxSwCtr++;
                OSIntCtxSw();
            }
        }
        OS_EXIT_CRITICAL();
    }
}
```

OSIntCtxSw

- 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

```
OSIntCtxSw( )
{
    OSTCBCur = OSTCBHighRdy; //任务控制块的切换
    OSPrioCur=OSPrioHighRdy;
    SP = OSTCBHighRdy->OSTCBStkPtr; //SP指向待运行任务堆栈

    用出栈指令把R1,R2,.....弹入CPU的通用寄存器;

    RETI;                                //中断返回, 使PC指向待运行任务
}
```

树莓派移植的OSIntCtxSw函数

OSIntCtxSw

```
BL    OSTaskSwHook                ; OSTaskSwHook();

LDR    R4, ??OS_PrioCur           ; OSPrioCur = OSPrioHighRdy
LDR    R5, ??OS_PrioHighRdy
LDRB   R6, [R5]
STRB   R6, [R4]
LDR    R4, ??OS_TCBCur            ; OSTCBCur = OSTCBHighRdy;
LDR    R6, ??OS_TCBHighRdy
LDR    R6, [R6]
STR    R6, [R4]
LDR    SP, [R6]                   ; SP = OSTCBHighRdy->OSTCBStkPtr;
                                        ; RESTORE NEW TASK'S CONTEXT

LDR    R4, [SP], #4                ; pop new task's CPSR
MSR    CPSR_cxsf, R4
LDR    R0, [SP], #4                ; pop new task's context
LDR    R1, [SP], #4
LDR    R2, [SP], #4
LDR    R3, [SP], #4
LDR    R4, [SP], #4
LDR    R5, [SP], #4
LDR    R6, [SP], #4
LDR    R7, [SP], #4
LDR    R8, [SP], #4
LDR    R9, [SP], #4
LDR    R10, [SP], #4
LDR    R11, [SP], #4
LDR    R12, [SP], #4
LDR    LR, [SP], #4
LDR    PC, [SP], #4
```

应用程序中的临界段

- 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。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，为增强可移植性， $\mu\text{C}/\text{OS-II}$ 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()

树莓派移植的开关中断宏

- 在树莓派的移植中，通过设置处理器CPSR的IRQ和FIQ位来关中断。这两个宏定义如下：

```
#define OS_ENTER_CRITICAL() {cpu_sr = OS_CPU_SR_Save();}
```

```
#define OS_EXIT_CRITICAL() {OS_CPU_SR_Restore(cpu_sr);}
```

OS_CPU_SR_Save

```
MRS    R0,CPSR                ; Set IRQ and FIQ bits in CPSR to disable all interrupts
ORR     R1,R0,#NO_INT
MSR     CPSR_c,R1
MRS     R1,CPSR                ; Confirm that CPSR contains the proper interrupt disable flags
AND     R1,R1,#NO_INT
CMP     R1,#NO_INT
BNE     OS_CPU_SR_Save         ; Not properly disabled (try again)
MOV     PC,LR                  ; Disabled, return the original CPSR contents in R0
```

OS_CPU_SR_Restore

```
MSR     CPSR_c,R0
MOV     PC,LR
```

μC/OS-II的系统时钟

- μC/OS-II与大多数计算机系统一样，用硬件定时器产生一个周期为ms级的周期性中断来实现系统时钟，最小的时钟单位就是两次中断之间相间隔的时间，这个最小时钟单位叫做**时钟节拍**（Time Tick）。
- 硬件定时器以时钟节拍为周期定时地产生中断。中断服务程序通过调用函数OSTimeTick()来完成系统在每个时钟节拍时需要做的工作。

OSTimeTick函数

```
void OSTimeTick (void)
{
    .....
    OSTimeTickHook( );
    .....
    OSTime++;                                //记录节拍数
    .....
    if (OSRunning == TRUE) {
        ptcb = OSTCBLList;
        while (ptcb->OSTCBPrio != OS_IDLE_PRIO)    //处理延时任务及调度
        {
            OS_ENTER_CRITICAL();
            if (ptcb->OSTCBDly != 0)
            {
                if (--ptcb->OSTCBDly == 0)    //任务的延时时间减一
                {
                    if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) == OS_STAT_RDY)
                    {
                        OSRdyGrp |= ptcb->OSTCBBitY;
                        OSRdyTbl[ptcb->OSTCBBY] |= ptcb->OSTCBBitX;
                    } else {
                        ptcb->OSTCBDly = 1;
                    }
                }
            }
            ptcb = ptcb->OSTCBNext;
            OS_EXIT_CRITICAL();
        }
    }
}
```


任务的延时

- 任务的延时函数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。

```
void OSTimeDly (INT16U ticks)
{
    #if OS_CRITICAL_METHOD == 3
        OS_CPU_SR cpu_sr;
    #endif
    if (ticks > 0)
    {
        OS_ENTER_CRITICAL();
        if ((OSRdyTbl[OSTCBCur->OSTCBBY] &= ~OSTCBCur->OSTCBBitX) == 0)
        {
            OSRdyGrp &= ~OSTCBCur->OSTCBBitY;    //取消当前任务的就绪状态
        }
        OSTCBCur->OSTCBDly = ticks;                //延时节拍数存入任务控制块
        OS_EXIT_CRITICAL();
        OS_Sched();                                //调用调度函数
    }
}
```

其他时间函数

- `INT8U OSTimeDlyResume(INT8U prio);`
- `INT32U OSTimeGet(void);`
- `void OSTimeSet(INT32U ticks);`



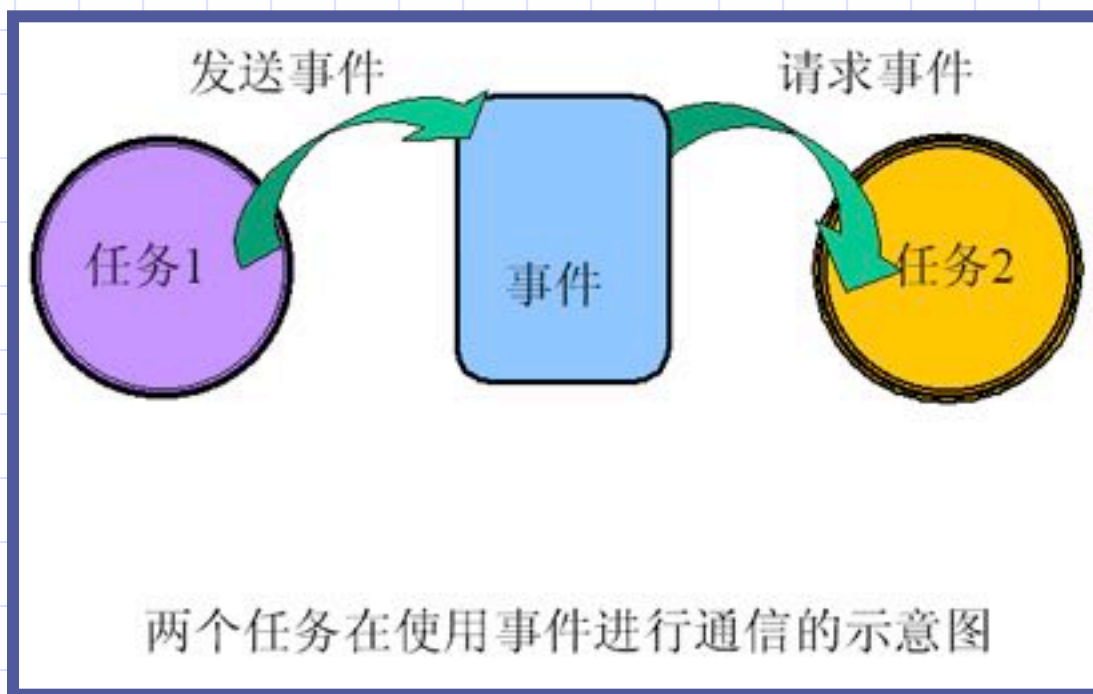
任务的同步与通信

必要性

- 系统中的多个任务在运行时，经常需要互相无冲突地访问同一个共享资源，或者需要互相支持和依赖，甚至有时还要互相加以必要的限制和制约，才能保证任务的顺利运行。因此，操作系统必须具有对任务的运行进行协调的能力，从而使任务之间可以无冲突、流畅地同步运行，而不致导致灾难性的后果。
- 计算机系统是依靠任务之间的良好通信来保证任务与任务的同步的。

事件

- 任务间的同步依赖于任务间的通信。在 $\mu\text{C}/\text{OS-II}$ 中，程序使用信号量、邮箱（消息邮箱）和消息队列这些被称作事件的中间环节来实现任务之间的通信。



事件控制块

- 为了把描述事件的数据结构统一起来， $\mu\text{C}/\text{OS-II}$ 使用叫做事件控制块ECB的数据结构来描述诸如信号量、邮箱（消息邮箱）和消息队列这些事件。事件控制块中包含包括等待任务表在内的所有有关事件的数据。

```
typedef struct
{
    INT8U  OSEventType;           //事件的类型
    INT16U  OSEventCnt;           //信号量计数器
    void *OSEventPtr;             //消息或消息队列的指针
    INT8U  OSEventGrp;            //等待事件的任务组
    INT8U  OSEventTbl[OS_EVENT_TBL_SIZE]; //任务等待表
} OS_EVENT;
```

空事件控制块链表

- 在 μ C/OS-II初始化时，系统会在初始化函数OSInit()中按应用程序使用事件的总数OS_MAX_EVENTS（在文件OS_CFG.H中定义），创建OS_MAX_EVENTS个空事件控制块并借用成员OSEventPtr作为链接指针，把这些空事件控制块链接成一个单向链表。由于链表中的所有控制块尚未与具体事件相关联，故该链表叫做空事件控制块链表。
- 每当应用程序创建一个事件时，系统就会从链表中取出一个空事件控制块，并对它进行初始化以描述该事件。而当应用程序删除一个事件时，就会将该事件的控制块归还给空事件控制块链表。

空事件控制块链表

OSEventFreeList

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

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

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

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

0

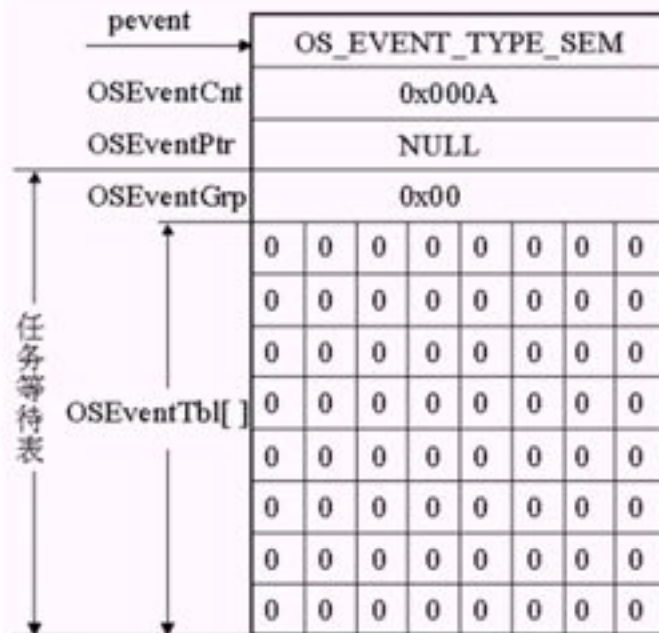
共OS_MAX_EVENTS个 事件控制块

空事件控制块链表

信号量及其操作

- 这里的信号量是指计数信号量。它的OSEventType为OS_EVENT_TYPE_Sem, OSEventCnt用来存储计数值。

一个刚创建且计数器初值 OSEventCnt 为 10 的信号量的示意图如图所示。



一个刚创建且计数器初值为10的信号量

信号量及其操作

- 在使用信号量之前，应用程序必须调用函数 `OSSemCreate()` 来创建一个信号量，`OSSemCreate()` 的原型为：

`OS_EVENT *OSSemCreate (INT16U cnt) ;`

- `cnt` 为信号量计数器初值
- 函数的返回值为已创建的信号量的指针。

信号量及其操作

- 任务通过调用函数OSSemPend()请求信号量，函数OSSemPend()的原型如下：

```
void OSSemPend (  
    OS_EVENT *pevent,      //信号量的指针  
    INT16U timeout,        //等待时限  
    INT8U *err);           //错误信息
```

- 为防止任务因得不到信号量而处于长期的等待状态，函数OSSemPend允许用参数timeout设置一个等待时间的限制，当任务等待的时间超过timeout时可以结束等待状态而进入就绪状态。如果参数timeout被设置为0，则表明任务的等待时间为无限长。

信号量及其操作

- 任务释放信号量需调用函数OSSemPost()。OSSemPost()函数在对信号量的计数器操作之前，首先要检查是否还有等待该信号量的任务。如果没有，就把信号量计数器OSEventCnt加一；如果有，则调用调度器OS_Sched()去运行等待任务中优先级别最高的任务。函数OSSemPost()的原型为：

```
INT8U OSSemPost      (OS_EVENT *pevent);
```

- 调用函数成功后，函数返回值为OS_ON_ERR，否则会根据具体错误返回OS_ERR_EVENT_TYPE（错误的事件类型）、OS_SEM_OVF（计数溢出）等。

信号量及其操作

- 应用程序如果不需要某个信号量了，那么可以调用函数 `OSSemDel()` 来删除该信号量，这个函数的原型为：

```
OS_EVENT *OSSemDel (  
    OS_EVENT *pevent, //信号量的指针  
    INT8U opt,         //删除条件选项  
    INT8U *err         //错误信息  
);
```

互斥信号量及其操作

- 在互斥型信号量的事件控制块中，OSEventType赋值为OS_EVENT_TYPE_MUTEX，OSEventCnt被分成了低8位和高8位两部分：低8位用来存放信号值（该值为0xFF时，信号为有效，否则信号为无效），高8位用来存放为了避免出现优先级反转现象而要提升的优先级别prio。



互斥信号量及其操作

- 创建互斥型信号量需要调用函数OSMutexCreate()。函数OSMutexCreate()的原型如下：

```
OS_EVENT *OSMutexCreate (
                INT8U prio, //优先级别
                INT8U *err    //错误信息
);
```

- 函数OSMutexCreate()在创建互斥信号量时，需要指定一个优先级prio，占有该互斥信号量的任务的优先级会被提升至prio，以防止优先级逆转。

互斥信号量及其操作

- 当任务需要访问一个独占式共享资源时，就要调用函数 `OSMutexPend()` 来请求管理这个资源的互斥型信号量，如果信号量有信号（`OSEventCnt`的低8位为 `0xFF`），则意味着目前尚无任务占用资源，于是任务可以继续运行并对该资源进行访问，否则就进入等待状态，直至占用这个资源的其他任务释放了该信号量。函数 `OSMutexPend()` 的原型为：

```
void OSMutexPend (
    OS_EVENT *pevent,      //互斥型信号量指
    INT16U timeout,        //等待时限
    INT8U *err              //错误信息
);
```


互斥信号量及其操作

- 任务可以通过调用函数OSMutexPost()发送一个互斥型信号量，这个函数的原型为：

```
INT8U OSMutexPost (OS_EVENT *pevent);
```

- 如果不需要某个互斥信号量了，那么可以调用函数OSMutexDel()来删除该互斥信号量，这个函数的原型为：

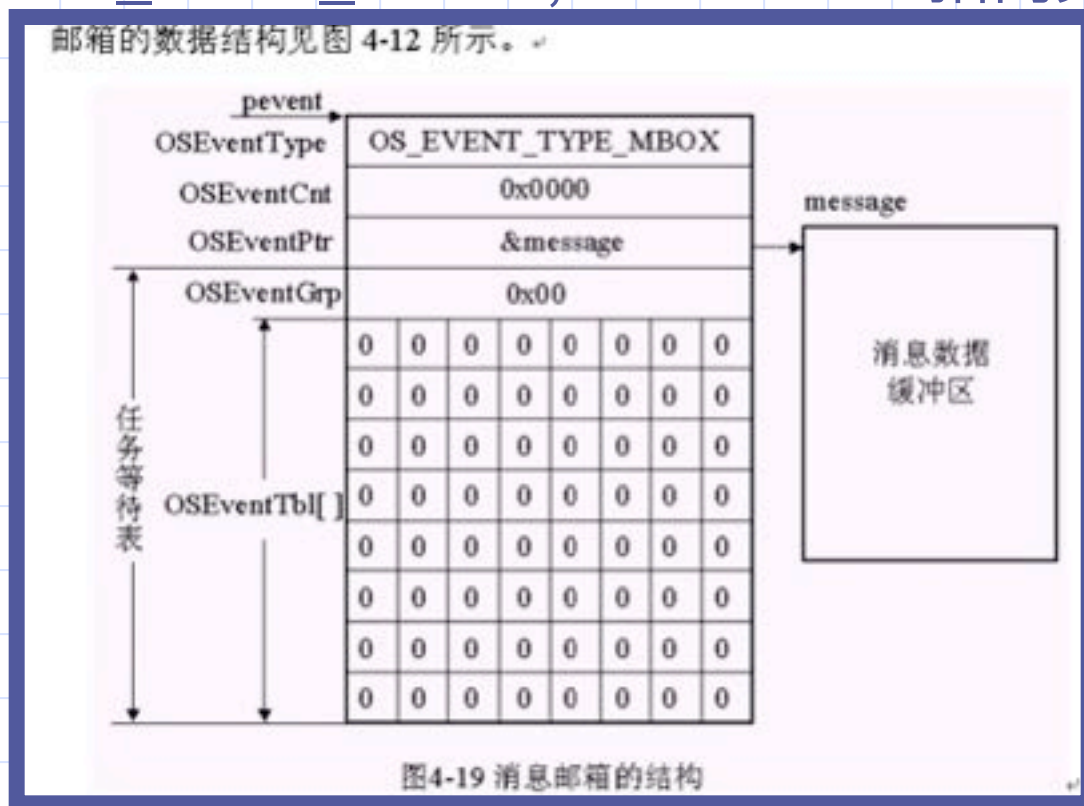
```
OS_EVENT *OSMutexDel (  
    OS_EVENT *pevent, //互斥信号量的指针  
    INT8U opt,         //删除条件选项  
    INT8U *err         //错误信息
```

消息邮箱及其操作

- 消息邮箱是在两个需要通信的任务之间通过传递数据缓冲区指针的方法来通信的。它的OSEventType为OS_EVENT_TYPE_MBOX, OSEvenPtr指向数据缓冲区。

消息邮箱及其操作

- 消息邮箱是在两个需要通信的任务之间通过传递数据缓冲区指针的方法来通信的。它的OSEventType为OS_EVENT_TYPE_MBOX，OSEventPtr指向数据缓冲区。



消息邮箱及其操作

- 创建邮箱需要调用函数OSMboxCreate (), 这个函数的原型为:

```
OS_EVENT *OSMboxCreate (void *msg);
```

- 函数中的参数msg为数据缓冲区的指针, 初值可一般为NULL
- 函数的返回值为消息邮箱的指针。

消息邮箱及其操作

- 当一个任务请求邮箱时需要调用函数OSMboxPend(), 这个函数的先查看邮箱指针OSEventPtr是否为NULL, 如果不是NULL就把邮箱中的消息指针返回给调用函数的任务, 同时用OS_NO_ERR通过函数的参数err通知任务获取消息成功; 如果邮箱指针OSEventPtr是NULL, 则使任务进入等待状态, 并引发一次任务调度。函数OSMboxPend()的原型为:

```
void *OSMboxPend (  
    OS_EVENT *pevent,           //请求消息邮箱指针  
    INT16U timeout,             //等待时限  
    INT8U *err                  //错误信息  
);
```

消息邮箱及其操作

- 任务可以通过调用函数OSMboxPost ()向消息邮箱发送消息，这个函数的原型为：

```
INT8U      OSMboxPost (OS_EVENT *pevent void  
*msg);
```

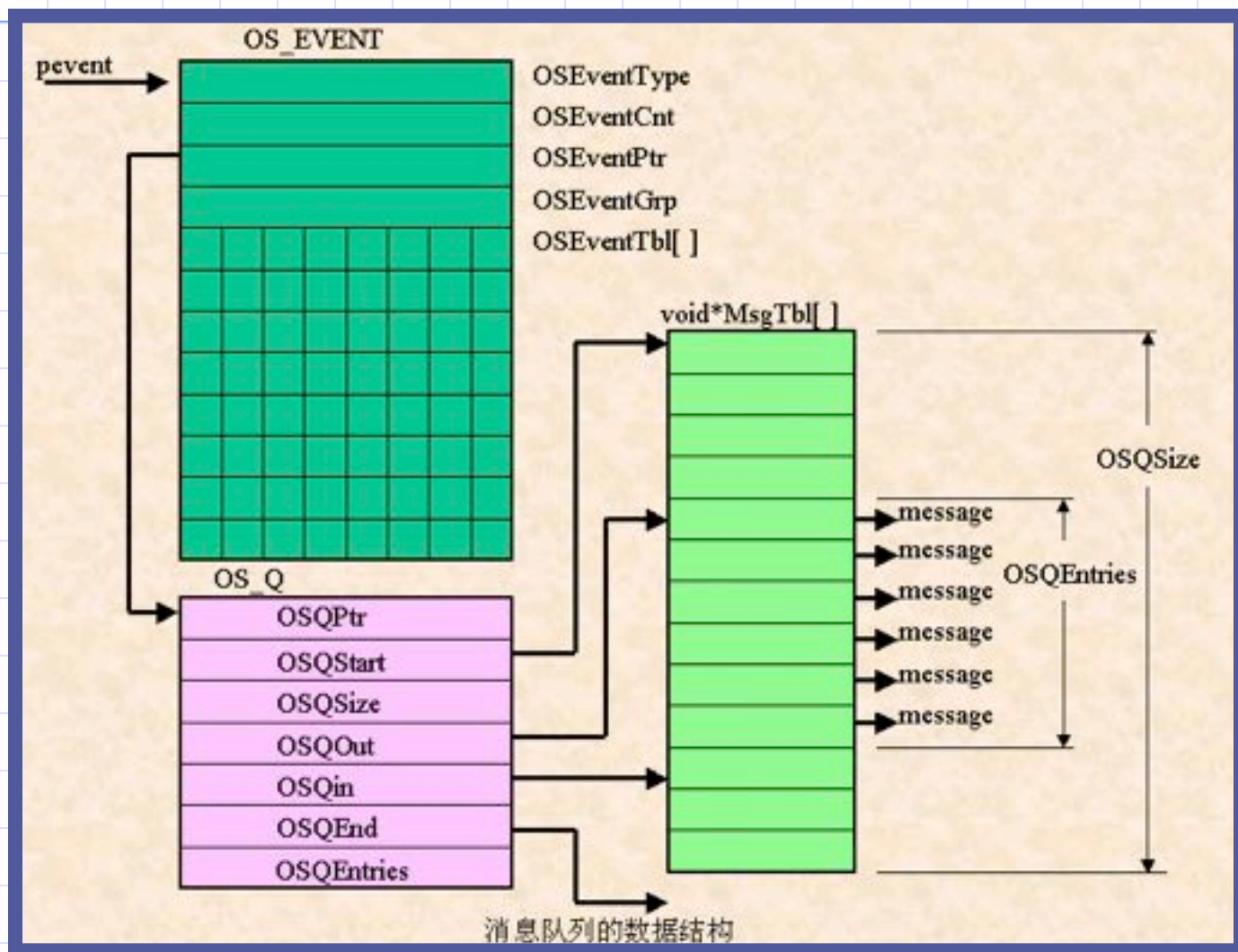
- 如果不需要某个消息邮箱了，那么可以调用函数OSMboxDel()来删除该消息邮箱，这个函数的原型为：

```
OS_EVENT *OSMboxDel (  
    OS_EVENT *pevent, //消息邮箱的指针  
    INT8U opt,         //删除条件选项  
    INT8U *err         //错误信息
```

消息队列及其操作

- 使用消息队列可以在任务之间传递多条消息。消息队列由三个部分组成：事件控制块、消息队列和消息。
- 当把事件控制块成员 `OSEventType` 的值置为 `OS_EVENT_TYPE_Q` 时，该事件控制块描述的就是一个消息队列。消息队列的数据结构如图所示。从图中可以看到，消息队列相当于一个共用一个任务等待列表的消息邮箱数组，事件控制块成员 `OSEventPtr` 指向了一个叫做队列控制块（`OS_Q`）的结构，该结构管理了一个数组 `MsgTbl[]`，该数组中的元素都是一些指向消息的指针。

消息队列及其操作



消息队列及其操作

● 队列控制块

消息队列的核心是消息指针数组，上图中的OS_Q就是消息指针数组的队列控制块，其各参数的含义如下表：

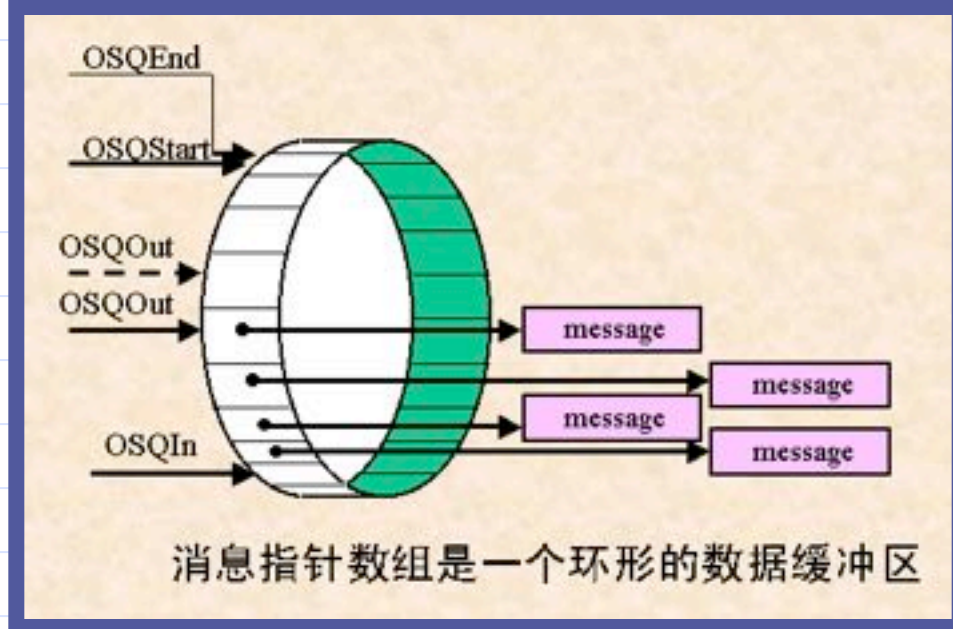
参 数	说 明
OSQSize	数组的长度。
OSQEntres	已存放消息指针的元素数目。
OSQStart	指针，指向消息指针数组的起始地址。
OSQEnd	指针，指向了消息指针数组结束单元的下一个单元。它使得数组构成了一个循环的缓冲区。
OSQIn	指针，指向了插入一条消息的位置。当它移动到与 OSQEnd 相等时，被调整到指向数组的起始单元。
OSQOut	指针，指向了被取出消息的位置。当它移动到与 OSQEnd 相等时，被调整到指向数组的起始单元。

消息队列及其操作

- 指针OSQIn和OSQOut可以移动。当OSQIn或OSQOut移动到数组末尾，也就是与OSQEnd相等时，OSQIn或OSQOut将会被调整到数组的起始位置OSQStart。也就是说，其实指针OSQEnd与OSQStart等值。于是，这个由消息指针构成的数组就头尾衔接起来形成了一个循环队列。

消息队列及其操作

- 指针OSQIn和OSQOut可以移动。当OSQIn或OSQOut移动到数组末尾，也就是与OSQEnd相等时，OSQIn或OSQOut将会被调整到数组的起始位置OSQStart。也就是说，其实指针OSQEnd与OSQStart等值。于是，这个由消息指针构成的数组就头尾衔接起来形成了一个循环队列。



消息队列及其操作

- 队列控制块的代码如下

```
typedef struct os_q
{
    struct os_q *OSQPtr;
    void **OSQStart;
    void **OSQEnd;
    void **OSQIn;
    void **OSQOut;
    INT16U OSQSize;
    INT16U OSQEntries;
} OS_Q;
```

- 空队列控制块链表

在 μ C/OS-II初始化时，系统将按文件OS_CFG.H中的配置常数OS_MAX_QS定义OS_MAX_QS个队列控制块，并用队列控制块中的指针OSQPtr将所有队列控制块链接为链表。由于这时还没有使用它们，故这个链表叫做空

消息队列及其操作

- 创建一个消息队列首先需要定义一指针数组，然后把各个消息数据缓冲区的首地址存入这个数组中，然后再调用函数OSQCreate()来创建消息队列。创建消息队列函数OSQCreate()的原型为：

```
OS_EVENT OSQCreate(void **start, INT16U size);
```

- 请求消息队列需要调用函数OSQPend()，该函数的原型为：

```
void *OSQPend(  
    OS_EVENT*pevent,    //消息队列的指针  
    INT16U timeout, //等待时限
```

```
    INT16U *pmsg, //消息内容
```

```
    //错误信息
```

消息队列及其操作

- 任务通过调用函数OSQPost()或OSQPostFront()来向消息队列发送消息。OSQPost()以FIFO（先进先出）的方式组织消息队列，OSQPostFront()以LIFO（后进先出）的方式组织消息队列。这两个函数的原型分别为：

```
INT8U OSQPost(  
    OS_EVENT*pevent,          //消息队列的指针  
    void*msg                  //消息指针  
);
```

```
INT8U OSQPostFront(  
    OS_EVENT*pevent,          //消息队列的指针  
    void*msg                  //消息指针  
);
```

消息队列及其操作

- 如果不需要某个消息队列了，那么可以调用函数 OSQDel() 来删除该消息队列，这个函数的原型为：

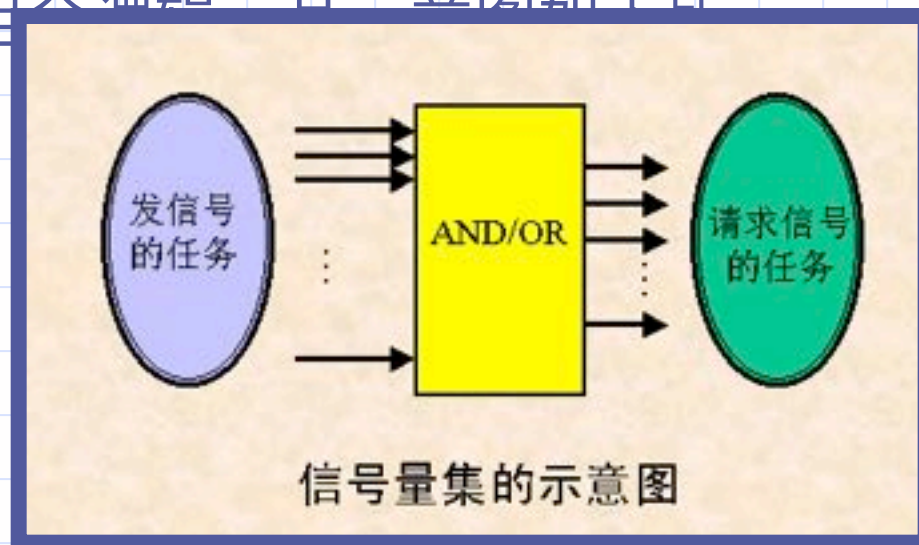
```
OS_EVENT *OSQDel (  
    OS_EVENT *pevent, //消息队列的指针  
    INT8U opt,         //删除条件选项  
    INT8U *err         //错误信息  
);
```

信号量集及其操作

- 在实际应用中，任务常常需要与多个事件同步，即要根据多个信号量组合作用的结果来决定任务的运行方式。μC/OS-II为了实现多个信号量组合功能定义了信号量集。
- 信号量集所能管理的信号量都是一些二值信号，所有信号量集实质上是一种可以对多个输入的逻辑信号进行基本逻辑运算的组合逻辑，其示意图如下所示

信号量集及其操作

- 在实际应用中，任务常常需要与多个事件同步，即要根据多个信号量组合作用的结果来决定任务的运行方式。 $\mu\text{C}/\text{OS-II}$ 为了实现多个信号量组合功能定义了信号量集。
- 信号量集所能管理的信号量都是一些二值信号，所有信号量集实质上是一种可以对多个输入的逻辑信号进行基本逻辑运算的组合逻辑，其示意图如下所示。



信号量集及其操作

- 信号量集的标志组

不同于信号量、消息邮箱、消息队列等事件， $\mu\text{C}/\text{OS-II}$ 不使用事件控制块来描述信号量集，而使用了一个叫做标志组的结构`OS_FLAG_GRP`。`OS_FLAG_GRP`结构如下：

```
typedef struct{
    INT8U   OSFlagType;    //识别是否为信号量集的标志
    void     *OSFlagWaitList;//指向等待任务链表的指针
    OS_FLAGS OSFlagFlags;   //所有信号列表
}OS_FLAG_GRP;
```

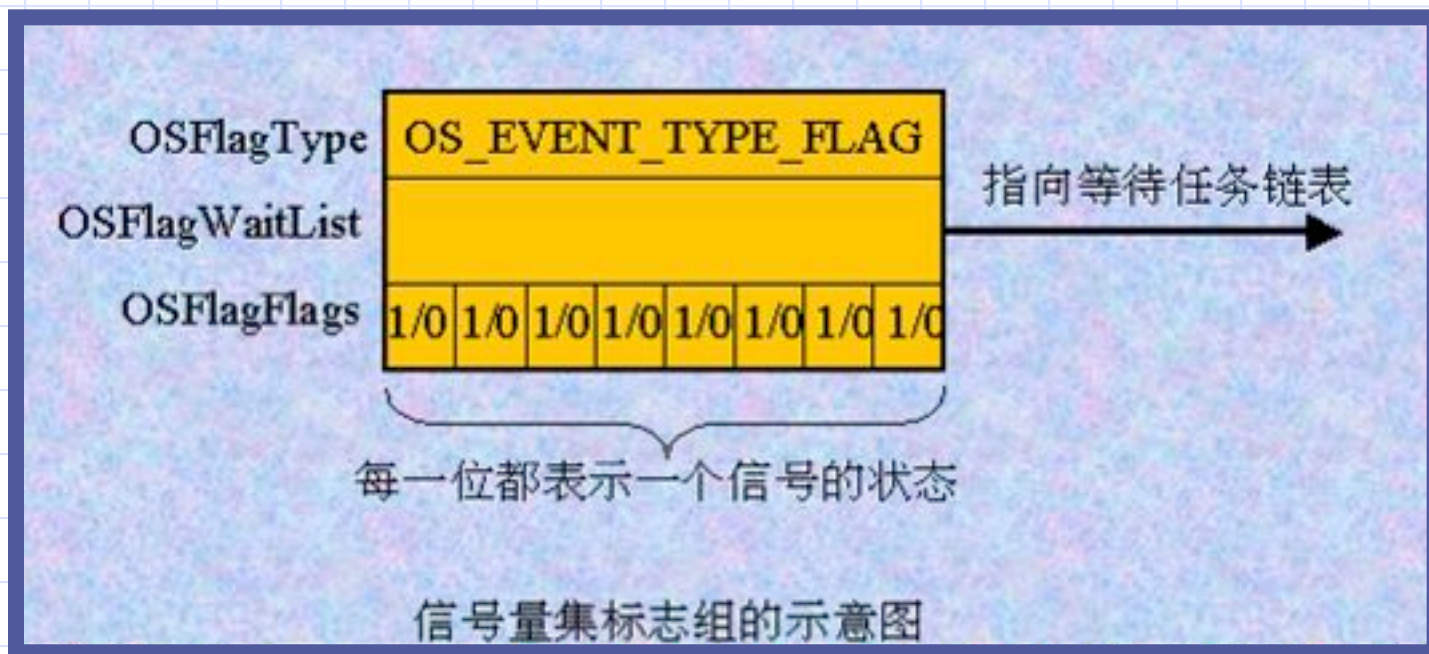
成员`OSFlagWaitList`是一个指针，当一个信号量集被创建后，这个指针指向了这个信号量集的等待任务链表。

信号量集及其操作

- 信号量集的标志组

信号量集及其操作

- 信号量集的标志组



信号量集及其操作

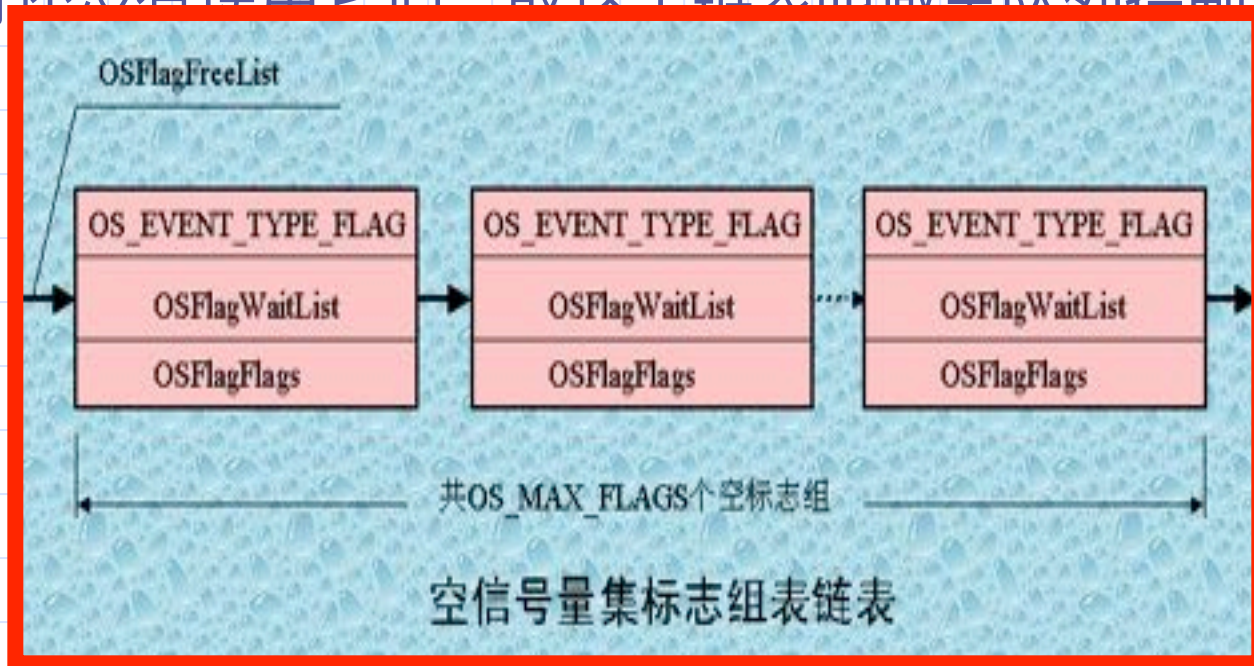
- 空信号量集的标志组链表

在 $\mu\text{C}/\text{OS-II}$ 初始化时，系统将按文件`OS_CFG.H`中的配置常数`OS_MAX_FLAGS`定义`OS_MAX_FLAGS`个信号量集的标志组，并将所有信号量集的标志组链接为链表。由于这时还没有使用它们，故这个链表叫做空队列控制块链表。

信号量集及其操作

- 空信号量集的标志组链表

在 $\mu\text{C}/\text{OS-II}$ 初始化时，系统将按文件`OS_CFG.H`中的配置常数`OS_MAX_FLAGS`定义`OS_MAX_FLAGS`个信号量集的标志组，并将所有信号量集的标志组链接为链表。由于这时还没有使用它们，故这个链表叫做空队列控制块链表。



信号量集及其操作

- 等待任务链表

与其他前面介绍过的事件不同，信号量集用一个双向链表来组织等待任务，每一个等待任务都是该链表中的一个节点（Node）。标志组 `OS_FLAG_GRP` 的成员 `OSFlagWaitList` 就指向了信号量集的这个等待任务链表。

等待任务链表节点 `OS_FLAG_NODE` 的结构如下：

```
typedef struct {  
    void *OSFlagNodeNext; //指向下一个节点的指针  
    void *OSFlagNodePrev; //指向前一个节点的指针  
    void *OSFlagNodeTCB; //指向对应任务控制块的指针  
    void *OSFlagNodeFlagGrp; //反向指向信号量集的指针  
    OS_FLAGS OSFlagNodeFlags; //信号过滤器  
    INT8U OSFlagNodeWaitType; //定义逻辑运算关系的数据  
} OS_FLAG_NODE;
```

信号量集及其操作

- 等待任务链表

信号量集及其操作

- 等待任务链表

定义信号的有效状态及等待任务与信号之间的逻辑关系的常数

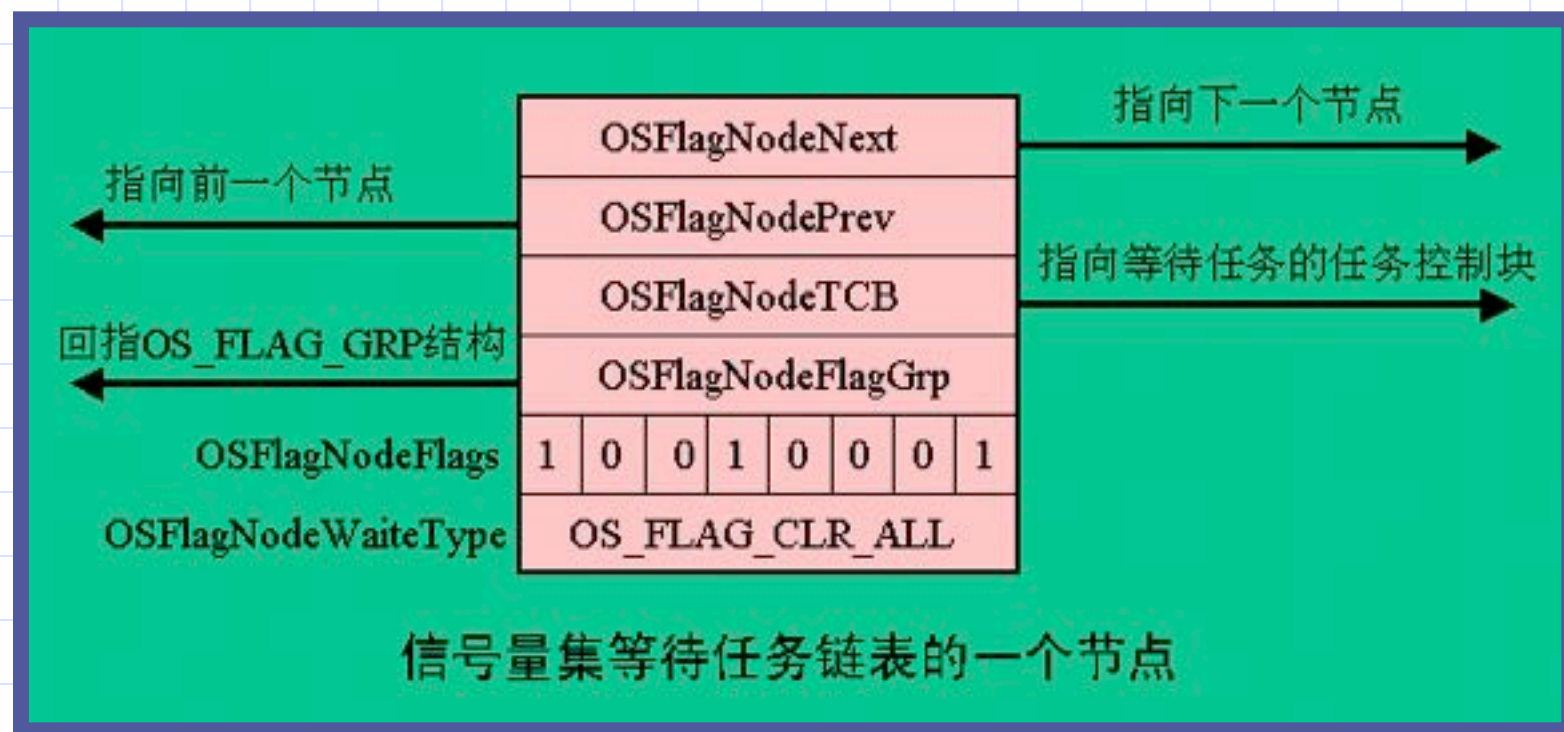
常 数	信号有效状态	等待任务的就绪条件
WAIT_CLR_ALL 或 WAIT_CLR_AND	0	信号全部有效 (全 0)。
WAIT_CLR_ANY 或 WAIT_CLR_OR	0	信号有一个或一个以上有效 (有 0)。
WAIT_SET_ALL 或 WAIT_SET_AND	1	信号全部有效 (全 1)。
WAIT_SET_ANY 或 WAIT_SET_OR	1	信号有一个或一个以上有效 (有 1)。

信号量集及其操作

- 等待任务链表

信号量集及其操作

- 等待任务链表

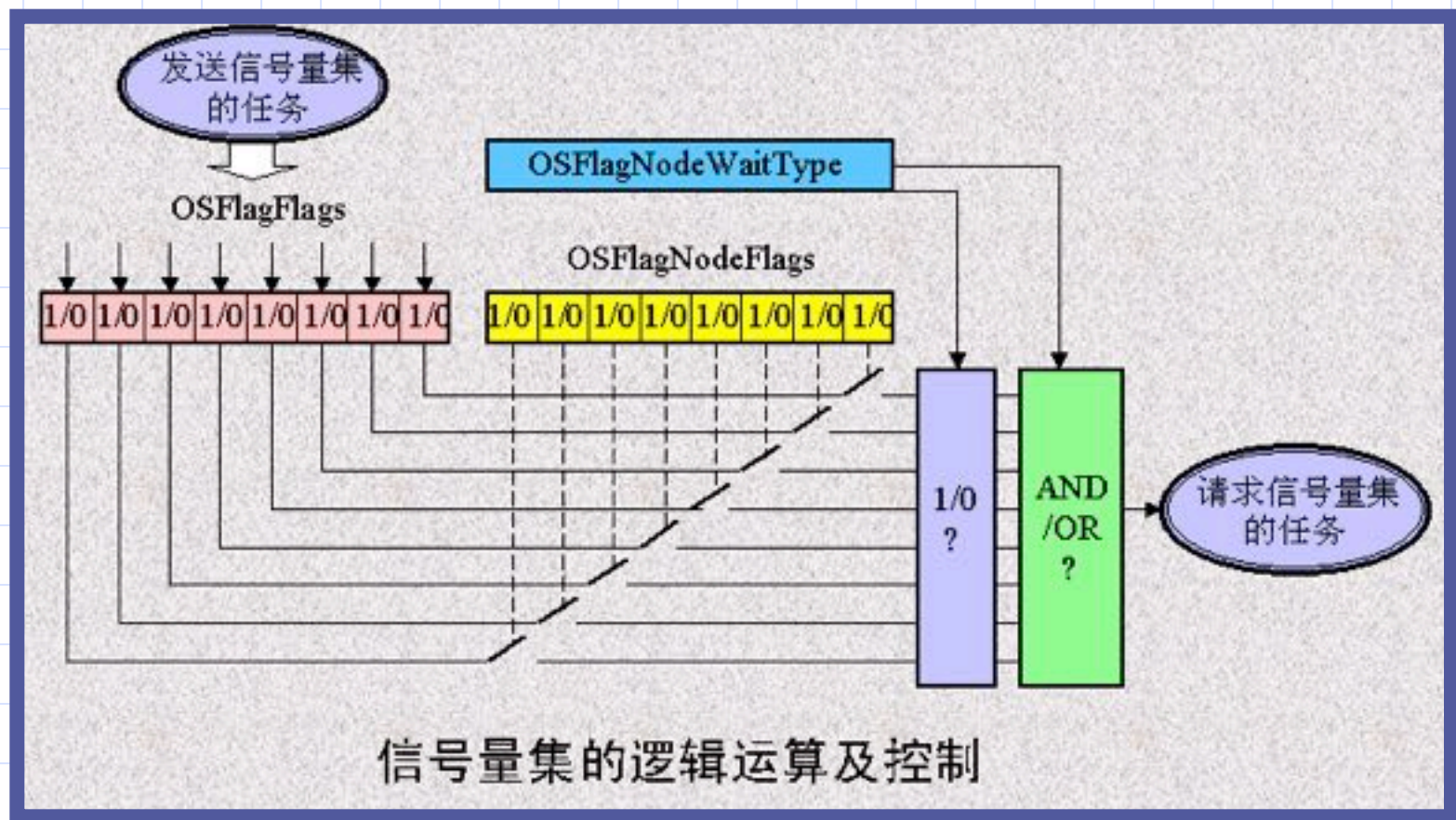


信号量集及其操作

- 等待任务链表

信号量集及其操作

- 等待任务链表



信号量集及其操作

- 给等待任务链表添加节点的函数为OS_FlagBlock(), 它会在请求信号量集函数OSFlagPend ()中被调用。原型为

```
static void OS_FlagBlock (  
    OS_FLAG_GRP *pgrp, //信号量集指针  
    OS_FLAG_NODE *pnode, //待添加的等待任务节点指针  
    OS_FLAGS flags,      //指定等待信号的数据  
    INT8U wait_type,     //信号与等待任务之间的逻辑  
    INT16U timeout       //等待时限  
);
```
- 从等待任务链表中删除节点的函数为OS_FlagUnlink(), 它会在发送信号量集函数OSFlagPost()中被调用。原型为

信号量集及其操作

- 任务可以通过调用函数OSFlagCreate ()来创建一个信号量集。OSFlagCreate ()的函数原型为：

```
OS_FLAG_GRP *OSFlagCreate (  
    OS_FLAGS flags, //信号的初始值  
    INT8U *err      //错误信息  
);
```

信号量集及其操作

- 任务可以通过调用函数OSFlagPend()请求一个信号量集，OSFlagPend()函数的原型为：

```
OS_FLAGS OSFlagPend (  
    OS_FLAG_GRP *pgrp,           //所请求的信号量集指针  
    OS_FLAGS flags,              //滤波器  
    INT8U wait_type,             //逻辑运算类型  
    INT16U timeout,              //等待时限  
    INT8U *err                    //错误信息  
);
```


信号量集及其操作

- 任务可以通过调用函数OSFlagPost ()向信号量集发信号，OSFlagPost ()函数的原型为：

```
OS_FLAGS OSFlagPost (  
    OS_FLAG_GRP *pgrp, //信号量集指针  
    OS_FLAGS flags,      //选择所要发送的信号  
    INT8U opt,           //信号有效的选项  
    INT8U *err           //错误信息  
);
```

任务向信号量集发信号，就是对信号量集标志组中的信号进行置“1”（置位）或置“0”（复位）的操作。Flags用来指定对信号量集中的哪些信号进行操作； opt用来指定对指

信号量集及其操作

- 如果不需要某个信号量集了，那么可以调用函数 `OSFlagDel()` 来删除该信号量集，这个函数的原型为：

```
OS_FLAG_GRP * OSFlagDel (  
    OS_FLAG_GRP *pgrp,    //信号量集的指针  
    INT8U opt,             //删除条件选项  
    INT8U *err             //错误信息  
);
```



内存的动态分配

μC/OS-II分配内存的特点

- μC/OS-II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。
- μC/OS-II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。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内存控制块来记录。

可动态分配内存的划分

- 应用程序如果要使用动态内存的话，首先在内存中划分出可以进行动态分配的区域，这个区域叫做内存分区，每个分区要包含若干个内存块。同一个分区中的内存块的字节数必须相等。
- 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，代码如下：

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

可动态分配内存的划分

- 为了使系统能够感知和有效地管理内存分区， $\mu\text{C}/\text{OS-II}$ 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。内存控制块的结构如下：

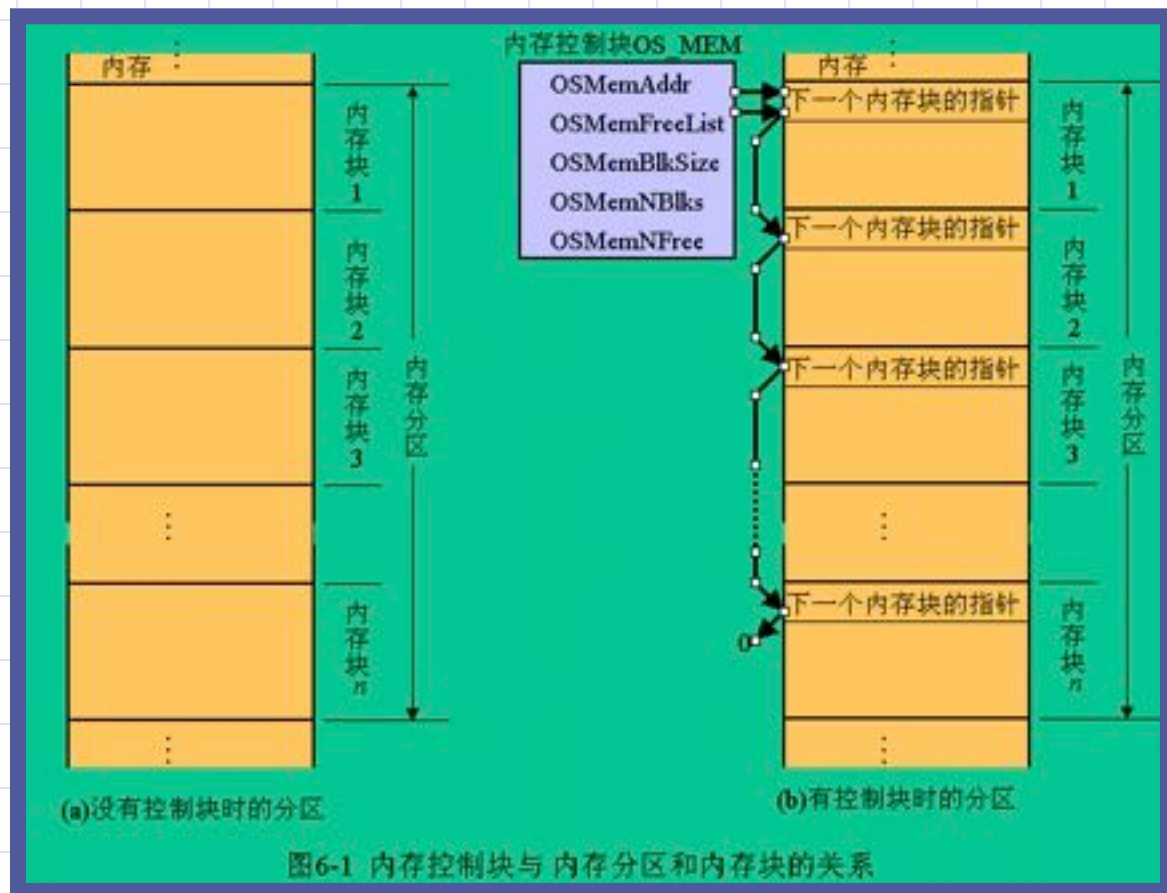
```
typedef struct {  
    void *OSMemAddr;           //内存分区的指针  
    void *OSMemFreeList; //内存控制块链表的指针  
    INT32U OSMemBlkSize;       //内存块的长度  
    INT32U OSMemNBlks; //分区内内存块的数目  
    INT32U OSMemNFree;         //分区内当前可分配的内存块的数目  
} OS_MEM;
```

可动态分配内存的划分

- 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

可动态分配内存的划分

- 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



可动态分配内存的划分

- 和管理其他的控制块一样， $\mu\text{C}/\text{OS-II}$ 使用一个空内存控制块链表来管理空内存控制块



动态内存的管理

- 划分了欲使用的分区和内存块之后，应用程序可以通过调用函数 `OSMemCreate()` 来建立一个内存分区，`OSMemCreate()` 函数的原型为：

```
OS_MEM *OSMemCreate(  
    void*addr,           //内存分区的起始地址  
    INT32U nblks,        //分区中内存块的数目  
    INT32U blksize,      //每个内存块的字节数  
    INT8U *err           //错误信息  
);
```

动态内存的管理

- 在应用程序需要一个内存块时，应用程序可以通过调用函数OSMemGet()向某内存分区请求获得一个内存块，OSMemGet()函数的原型为：

```
void *OSMemGet (  
    OS_MEM *pmem, //内存分区的指针  
    INT8U *err      //错误信息  
);
```

动态内存的管理

- 当应用程序不再使用一个内存块时，必须要及时地将它释放。应用程序通过调用函数OSMemPut()来释放一个内存块，OSMemPut()函数的原型为：

```
INT8U OSMemPut (  
    OS_MEM *pmem, //内存块所属内存分区的指针  
    void *pblk     //待释放内存块的指针  
);
```

动态内存的管理

- 应用程序可以通过调用函数OSMemQuery()来查询一个分区目前的状态信息，函数OSMemQuery()的原型为：

```
INT8U OSMemQuery (  
    OS_MEM *pmem, //待查询的内存控制块的指针  
    OS_MEM_DATA *pdata//存放分区状态信息结构的指  
    针  
)
```

```
typedef struct {  
    void *OSAddr;                //内存分区的指针  
    void *OSFreeList;            //分区内内存块链表的头指针  
    INT32U OSBlkSize;            //内存块的长度  
    INT32U OSNBls;               //分区内内存块的数目  
    INT32U OSNFree;              //分区内空闲内存块的数目  
    INT32U OSNUSED;              //已被分配的内存块数目  
} OS_MEM_DATA;
```

课程大纲

 μ C/OSII操作系统简介

 操作系统内核移植简介

课程大纲

 μ C/OSII操作系统简介

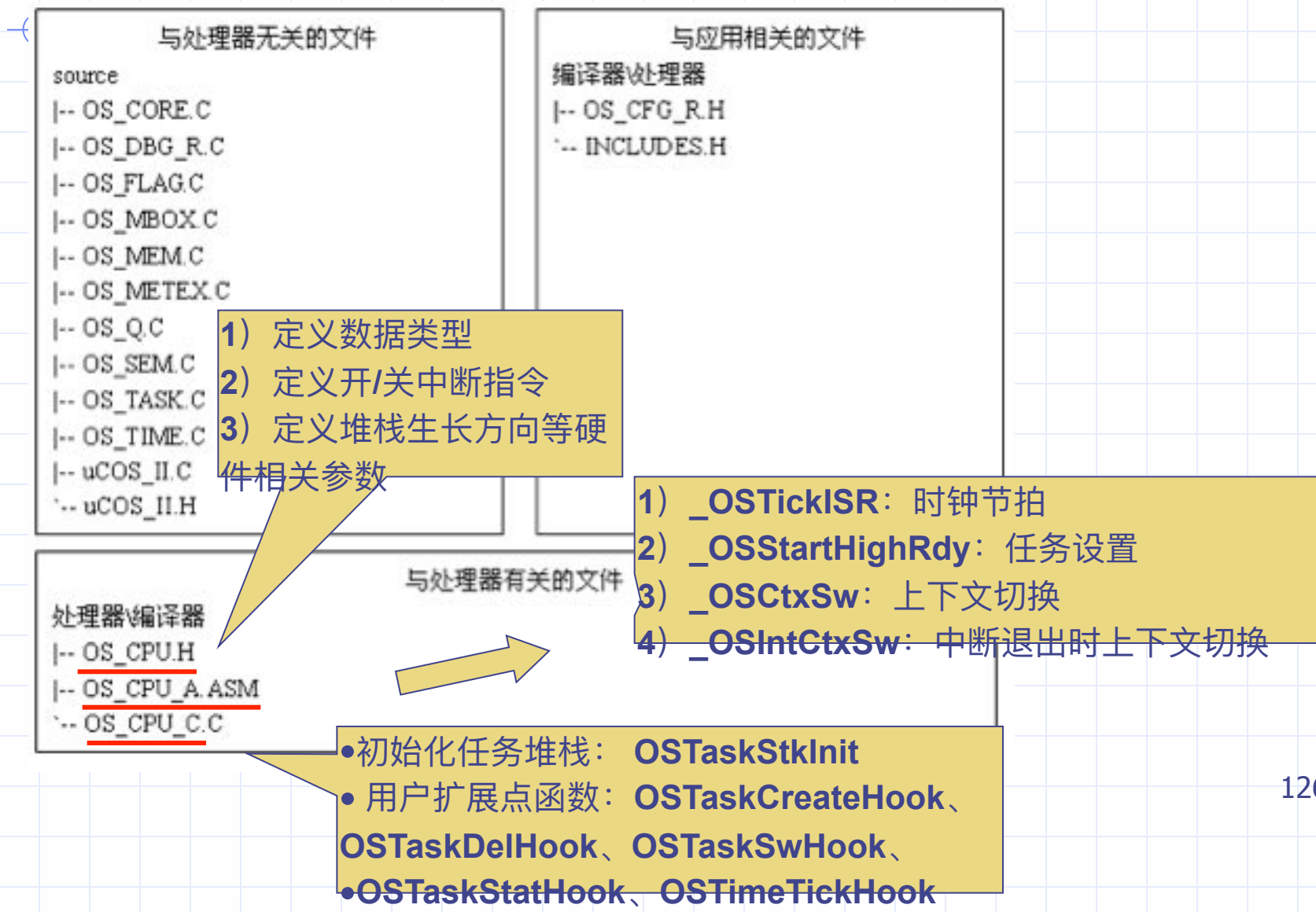
 操作系统内核移植简介

μC/OSII操作系统内核移植步骤(1/2)

p 要使μC/OSII正常运行，必须满足以下要求

- C 编译器能产生可重入代码
- 处理器支持中断，并且能产生定时中断，通常在10~100Hz
- 提供打开和关闭中断的指令
- 处理器支持能够容纳一定量数据的堆栈
- 处理器有将堆栈指针，以及寄存器读出、存储到堆栈，或内存中的指令

μC/OSII操作系统内核移植步骤(2/2)



Linux操作系统内核移植的主要步骤(1/3)

p 开发环境选择

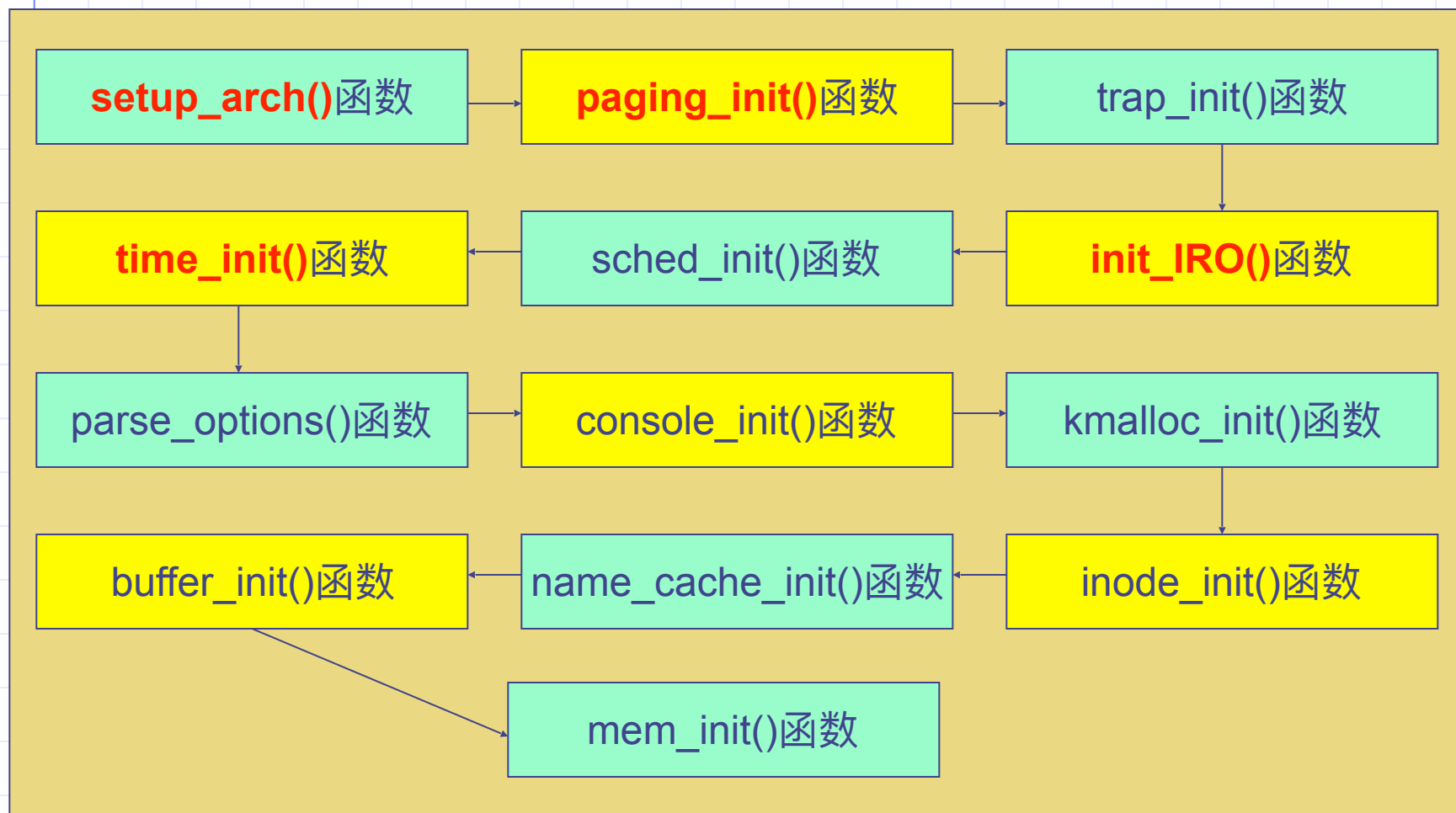
- 交叉编译器、交叉链接器选择
- 调试器选择（模拟器）
- **Makefile**修改

p 机型号、机型描述信息增加

p 启动代码添加（arch/kernel/head_***.S）

Linux操作系统内核移植的主要步骤(2/3)

p Start_kernel()的初始化步骤



Linux操作系统内核移植的主要步骤(3/3)

- p 修改与体系结构环境设置函数 (**setup_arch**)
- p 修改页表结构初始化函数 (**paging_init**)
- p 修改中断初始化函数 (**init_IRQ**)
- p 修改基本驱动程序 (时钟、定时器、串口通讯)
- p 内核编译、生成

