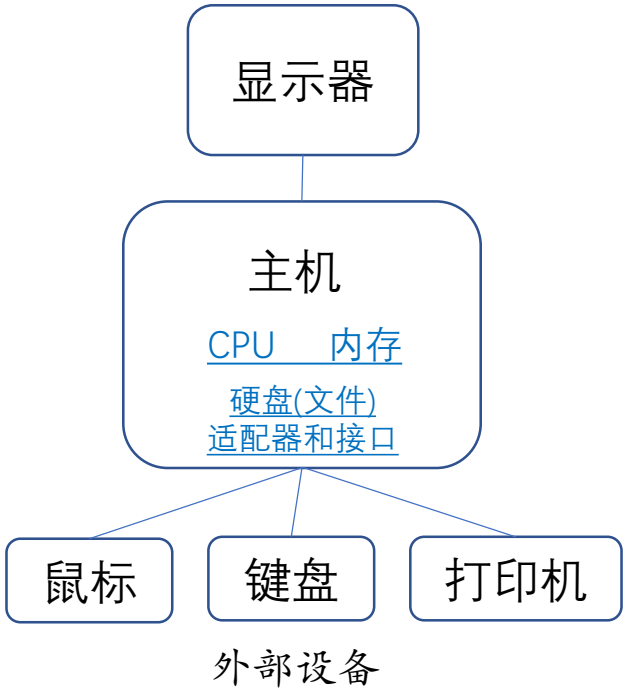
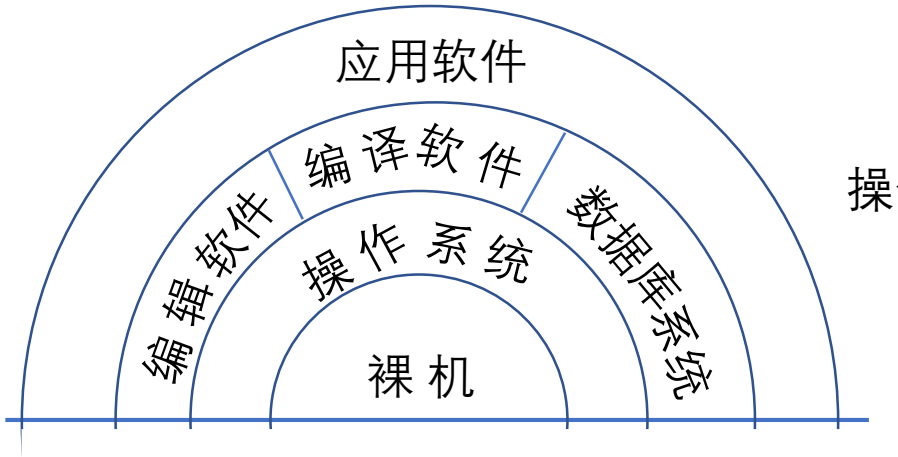
第十章、uCOS-II操作系统

官方网站 参考 移植

什么是操作系统

- 操作系统(Operation System，简称OS)是管理计算机硬件与软件资源的计算机程序。



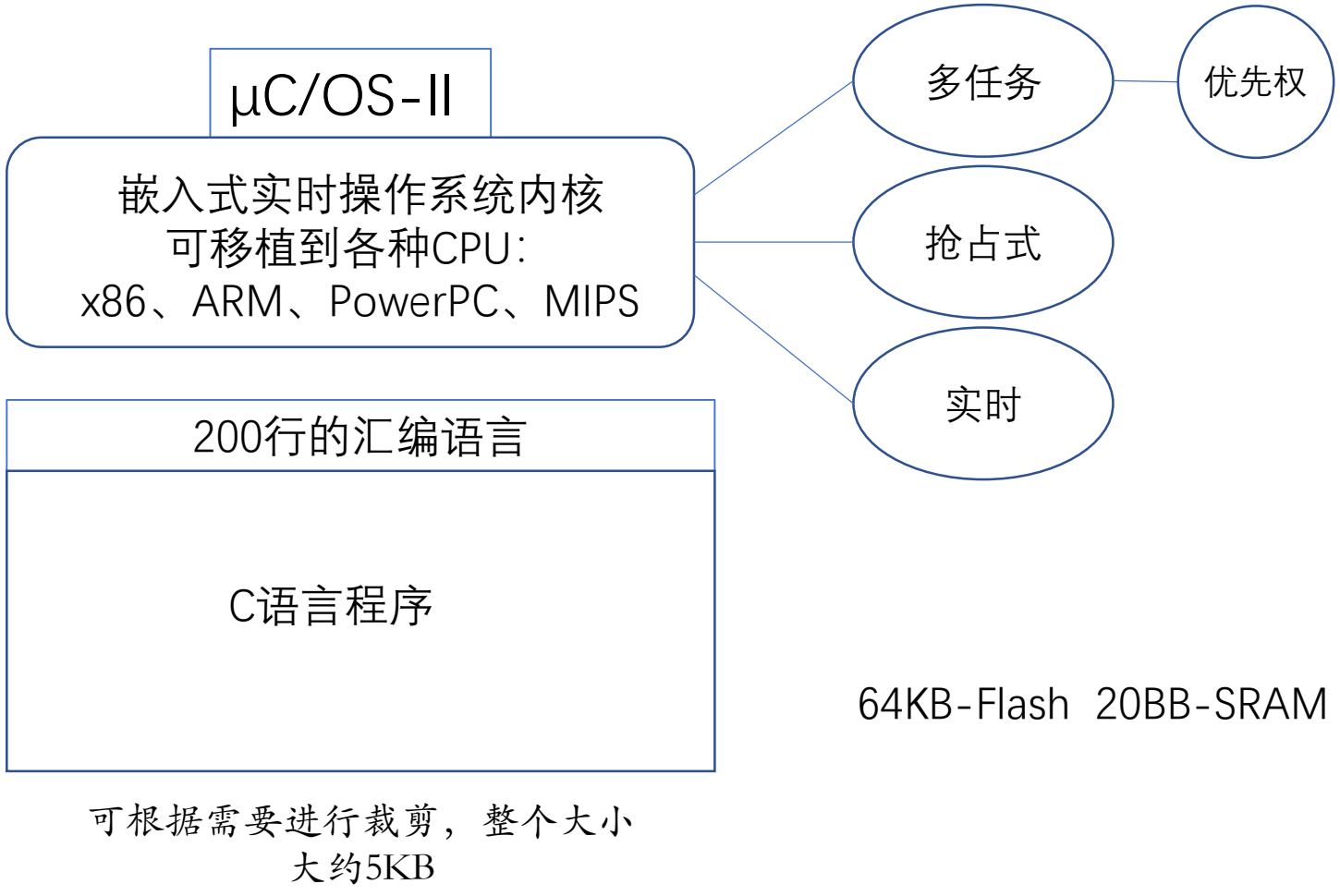


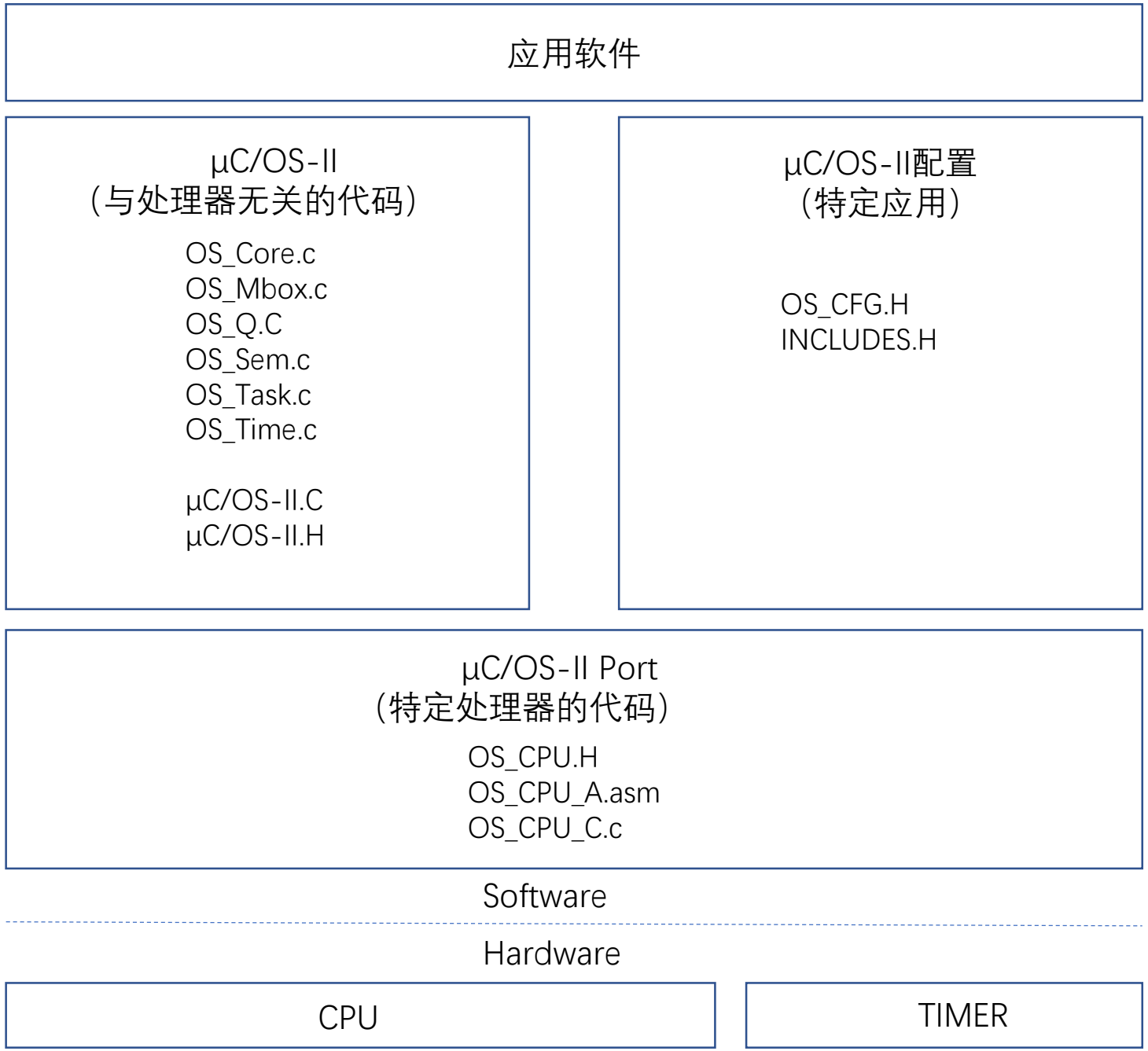
操作系统的功能
- ①任务管理(进程管理)
- ②存储管理
- ③设备管理
- ④文件管理
- ⑤作业管理

任务--task
存储--memory
设备--peripheral device
文件--file
作业--job

- µC/OS-II操作系统由Jean J. Labrosse在1992年开发
- 在全世界有数百种产品在应用:
  - 医疗器械
  - 移动电话
  - 路由器
  - 工业控制
  - GPS 导航系统
  - 智能仪器
  - 更多

μC/OS-II操作系统


μC/OS-II
嵌入式实时操作系统内核
可移植到各种CPU:
x86、ARM、PowerPC、MIPS
多任务
优先权
抢占式
实时
200行的汇编语言
C语言程序
64KB-Flash 20BB-SRAM
可根据需要进行裁剪，整个大小
大约5KB

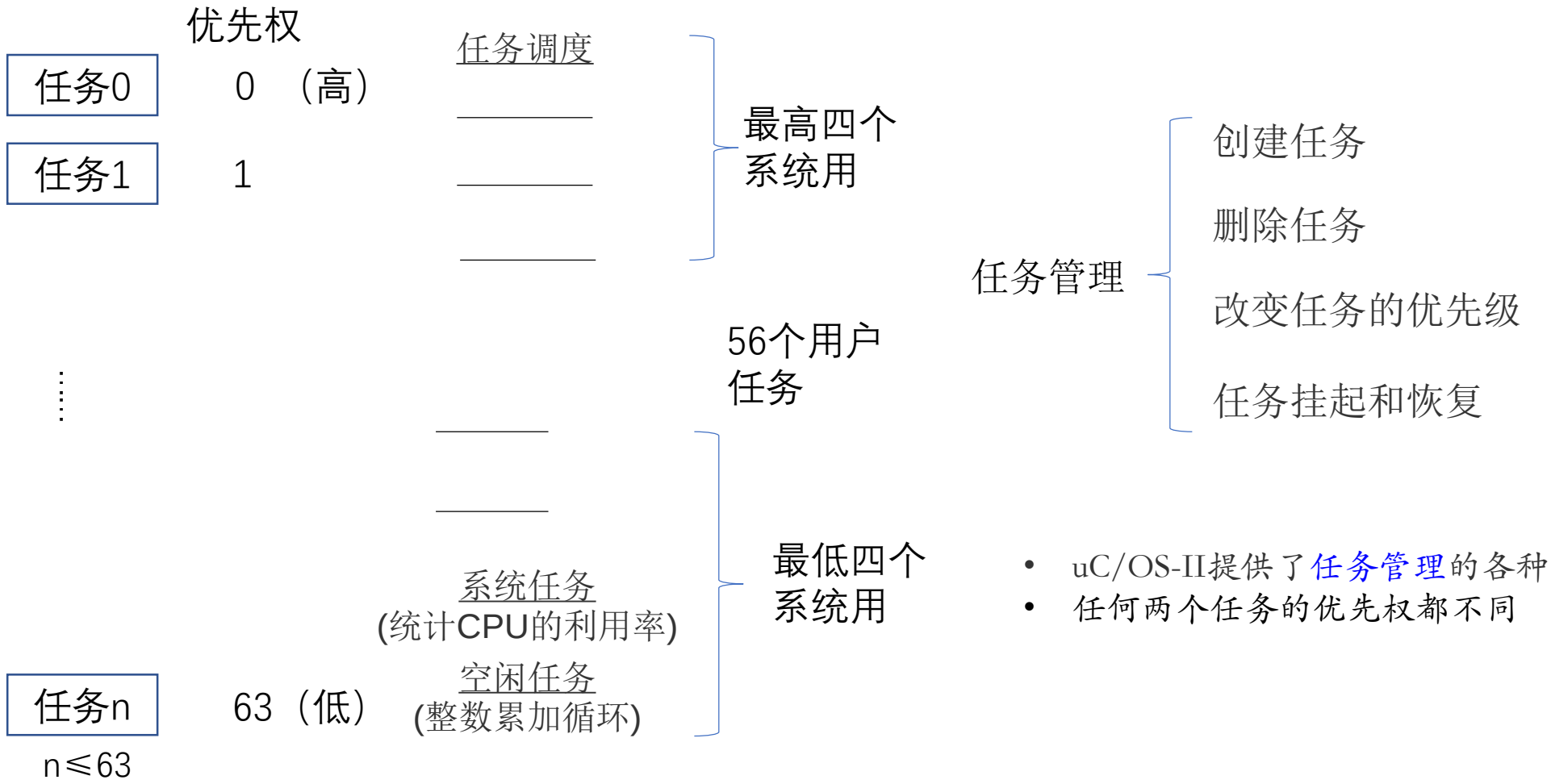


µC/OS-II可以大致分成核心、任务处理、时间处理、任务同步与通信和CPU的移植等5个部分：

- **核心部分**(OS_Core.c)用于维持系统基本工作，包括操作系统初始化、操作系统运行、中断进出的前导、时钟节拍、任务调度、事件处理等。
- **任务处理部分**(OS_Task.c) 负责任务的建立、删除、挂起、恢复等。µC/OS-II是以任务为基本单位进行调度的。
- **时钟部分**(OS_Time.c) 负责任务延时等操作。µC/OS-II中的最小时钟单位是timetick（时钟节拍）。
- **任务同步和通信部分**(OS_Mbox.c, OS_Q.c, OS_Sem.c)为事件处理部分，包括信号量、邮箱、邮箱队列、事件标志等部分，用于任务间的互相联系和对临界资源的访问。
- **与CPU的接口部分**(OS_CPU_C.c, OS_CPU_A.asm) 包括特定CPU的移植代码，包括中断级任务切换的底层实现、任务切换的底层实现、时钟节拍的产生和处理、中断的相关处理等内容。

主要功能

任务管理



- uC/OS-II提供了任务管理的各种函数调用
- 任何两个任务的优先权都不同

μC/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μC/OS-Ⅱ之前，用户至少要建立一个应用任务。

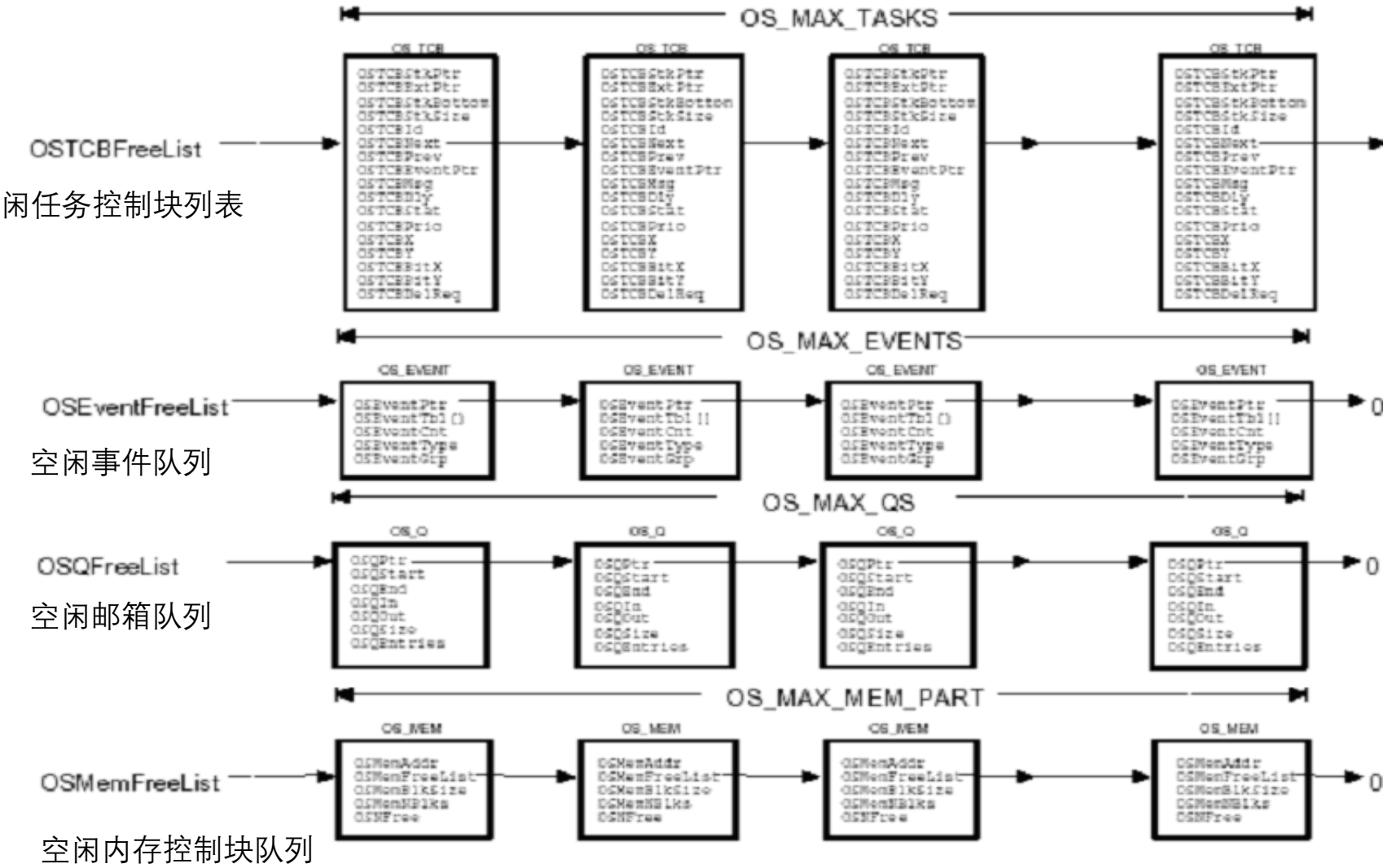
```
OSInit();     /* 初始化uC/OS-II*/
……
调用OSTaskCreate()或OSTaskCreateExt();
……
OSStart();   /*开始多任务调度!永不返回 */
```

OSInit()建立空闲任务idle task，这个任务总是处于就绪态的。空闲任务OSTaskIdle（）的优先级总是设成最低，即OS_LOWEST_PRIO。

OSStart：

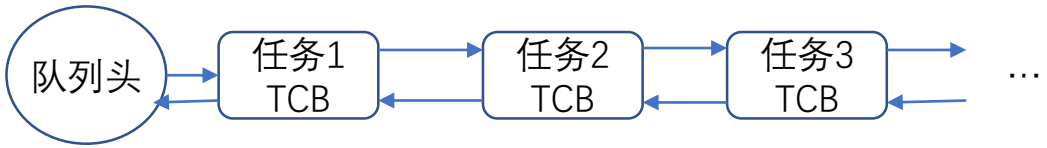
```
if (OSRunning == FALSE) {
    y             = OSUnMapTbl[OSRdyGrp];
    x             = OSUnMapTbl[OSRdyTbl[y]];
    OSPrioHighRdy = (INT8U)((y << 3) + x);
    OSPrioCur     = OSPrioHighRdy;
    OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];

    OSTCBCur      = OSTCBHighRdy;
    OSStartHighRdy();
}
```

μC/OS-Ⅱ还初始化了4个空数据结构缓冲区。

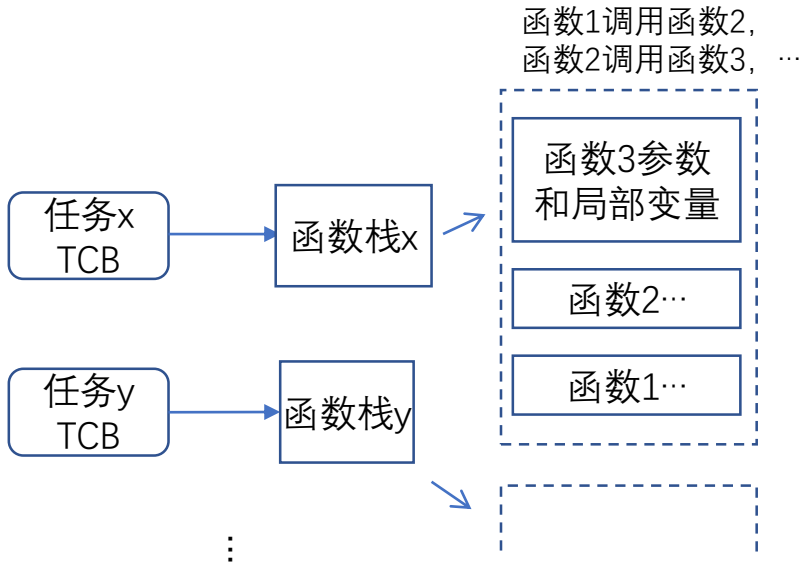


任务控制块（TCB）

```
struct os_tcb {
  OS_STK          *OSTCBStkPtr;    /*当前任务栈顶的指针*/
  struct os_tcb   *OSTCBNext;      /*任务控制块的双重链接指针*/
  struct os_tcb   *OSTCBPrev;      /*任务控制块的双重链接指针*/
  OS_EVENT        *OSTCBEventPtr;  /*事件控制块的指针*/
  void            *OSTCBMsg;       /*消息的指针*/
  INT16U           OSTCBDly;       /*任务延时*/
  INT8U            OSTCBStat;      /*任务的状态字*/
  INT8U            OSTCBPrio;      /*任务优先级*/
  INT8U            OSTCBX;         /*用于加速进入就绪态的过程*/
  INT8U            OSTCBY;         /*用于加速进入就绪态的过程*/
  INT8U            OSTCBBitX;      /*用于加速进入就绪态的过程*/
  INT8U            OSTCBBitY;      /*用于加速进入就绪态的过程*/
}
```



- 任务控制块 OS_TCB是一个数据结构，保存该任务的相关参数，包括任务堆栈指针，状态，优先级，任务表位置，任务链表指针等。
- 一旦任务建立了，任务控制块OS_TCBs将被赋值。
- 所有的任务控制块分为两条链表，空闲链表和使用链表。

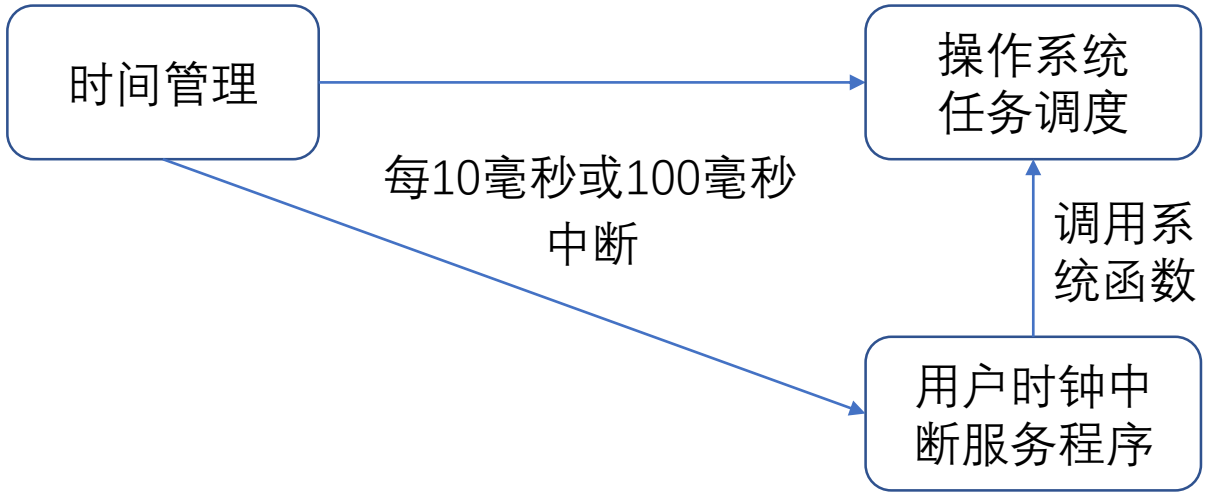


一个任务通常是一个无限的循环:

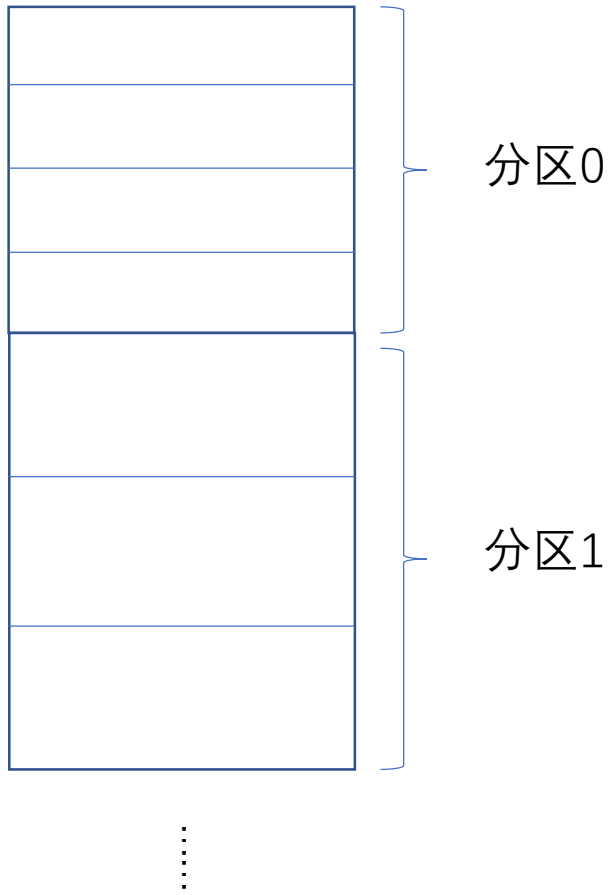
```
void mytask(void *pdata)
{
        do init
        while (1) {
            do something;
            waiting;
            do something;
        }
}
```

时间管理

- μC/OS-II的时间管理是通过定时中断来实现的，该定时中断一般为10毫秒或100毫秒发生一次，时间频率取决于用户对硬件系统的定时器编程来实现。中断发生的时间间隔是固定不变的，该中断也成为一个**时钟节拍**。
- μC/OS-II要求用户在定时中断的服务程序中，调用系统提供的与时钟节拍相关的系统函数，例如中断级的任务切换函数，系统时间函数。



内存管理



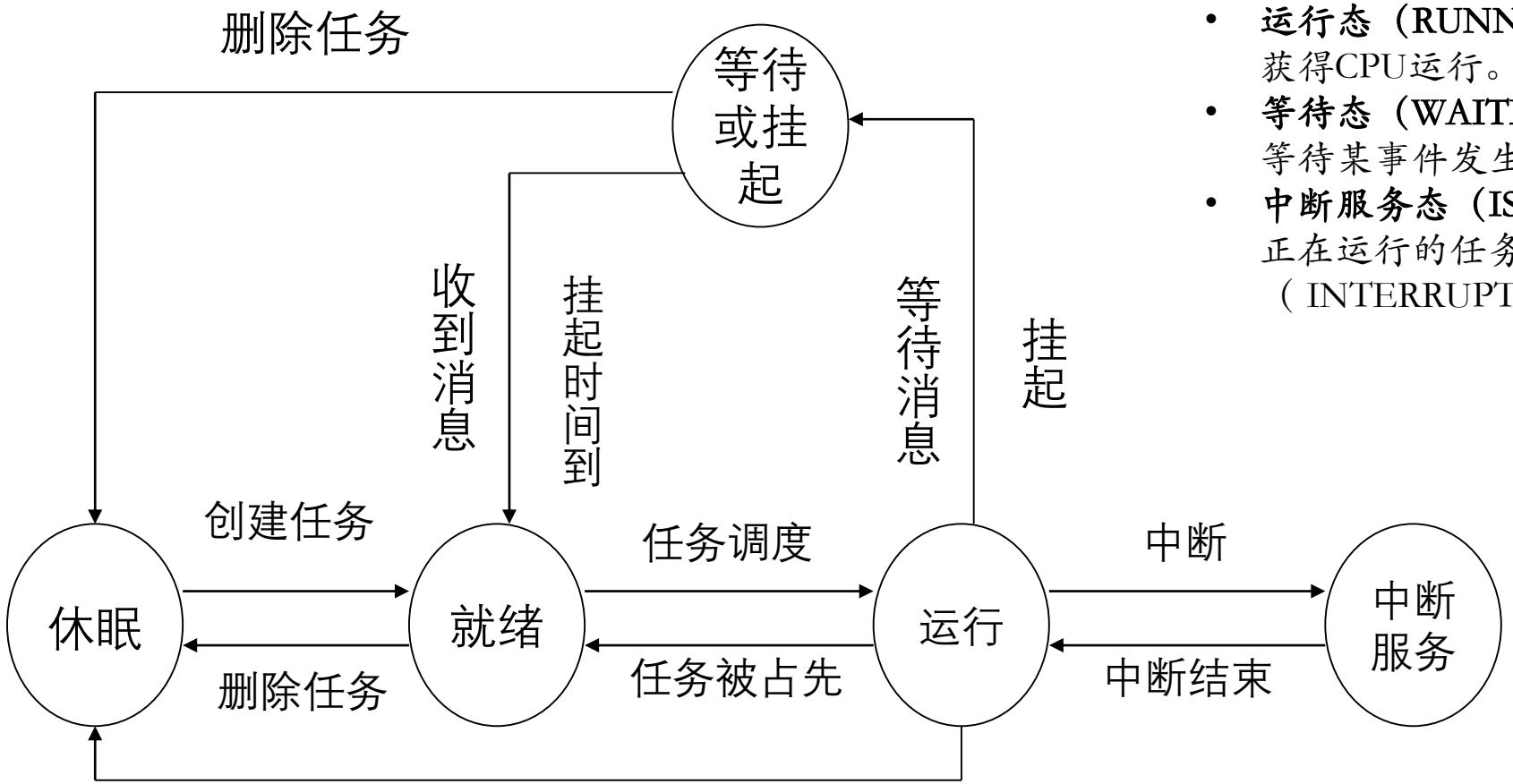
- 在ANSI C中是使用malloc和free两个函数来动态分配和释放内存。但在嵌入式实时系统中，多次这样的操作会导致内存碎片化，且由于内存管理算法的原因，malloc和free的执行时间也是不确定。
- μC/OS-II中把连续的大块内存按分区（partition）管理。每个分区中包含整数个大小相同的内存块，但不同分区之间的内存块大小可以不同。用户需要动态分配内存时，系统选择一个适当的分区，按块来分配内存。释放内存时将该块放回它以前所属的分区，这样能有效解决碎片问题，同时执行时间也是固定的。
- 为了便于内存的管理，在μC/OS-II中使用内存控制块（memory control blocks）的数据结构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内存控制块。

```
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;
```

*Linux需要MCU具有MMU(Memory Manangement Unit)功能， 而uC/OS-II不需要。

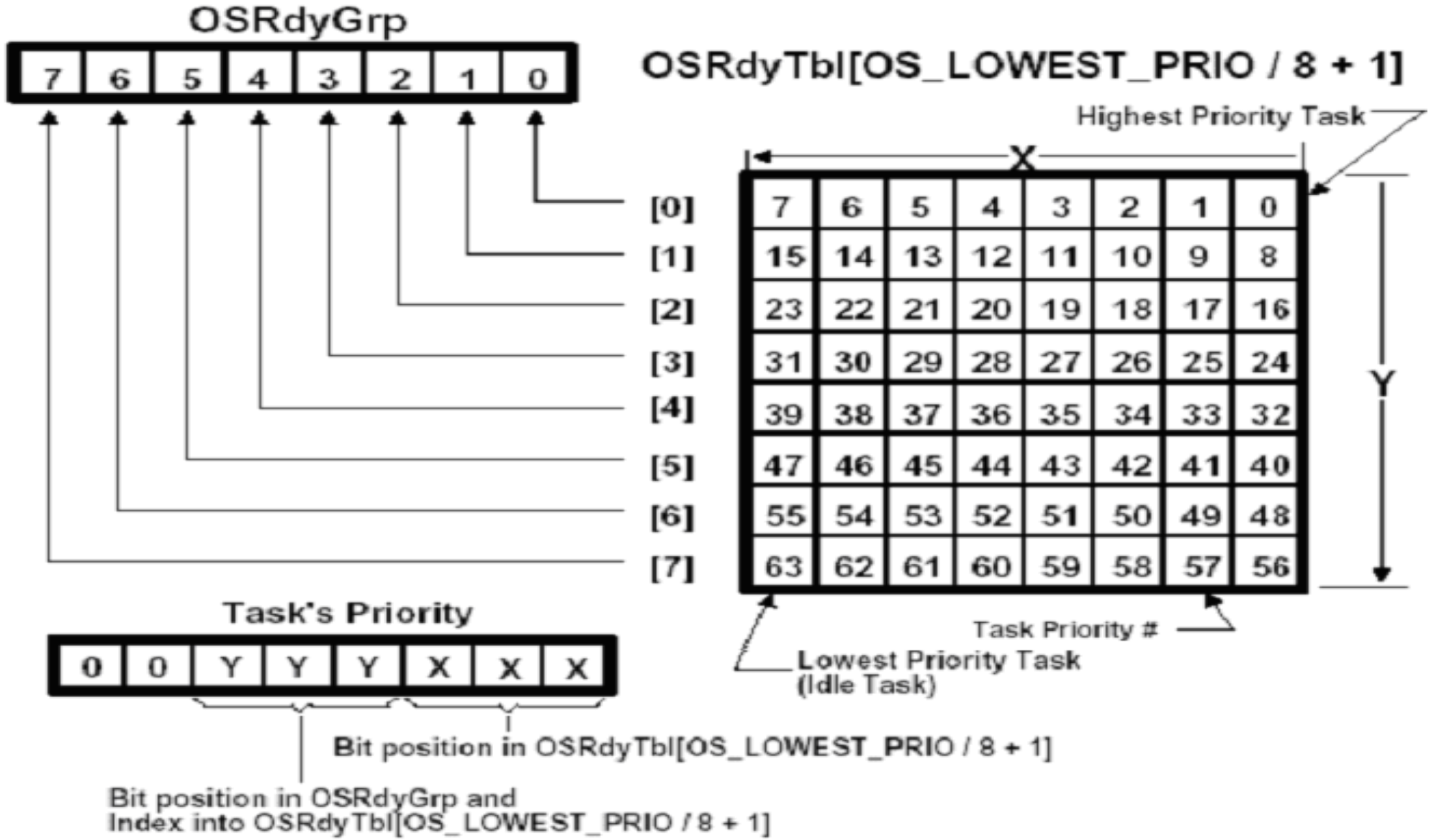
任务管理

任务状态



- **休眠态（DORMANT）**
  任务驻留在程序空间，还没有交给UCOSII管理，即还没有配备任务控制块和被创建。
- **就绪态（READY）**
  任务一旦创建，就进入就绪态准备获得CPU运行。
- **运行态（RUNNING）**
  获得CPU运行。
- **等待态（WAITING）**
  等待某事件发生的状态。
- **中断服务态（ISR）**
  正在运行的任务被中断就进入了中断服务态（INTERRUPT SERVICE ROUTINE）。

任务就绪表


OSRdyGrp
7 6 5 4 3 2 1 0
OSRdyTbl[OS_LOWEST_PRIO / 8 + 1]
Highest Priority Task
X
Y
[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
Task Priority #
Lowest Priority Task
(Idle Task)
Task's Priority
0 0 Y Y Y X X X
Bit position in OSRdyTbl[OS_LOWEST_PRIO / 8 + 1]
Bit position in OSRdyGrp and
Index into OSRdyTbl[OS_LOWEST_PRIO / 8 + 1]

任务创建

- 想让μC/OS-Ⅱ管理用户的任务，用户必须要先建立任务。
- 用户可以通过传递任务地址和其它参数到以下两个函数之一来建立任务:
  - OSTaskCreate()
  - OSTaskCreateExt()
- 任务不能由中断服务程序(ISR)来建立。

任务切换

- μC/OS-Ⅱ总是运行进入就绪态任务中优先级最高的那一个。确定哪个任务优先级最高，下面运行该哪个任务是由调度器（Scheduler）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。

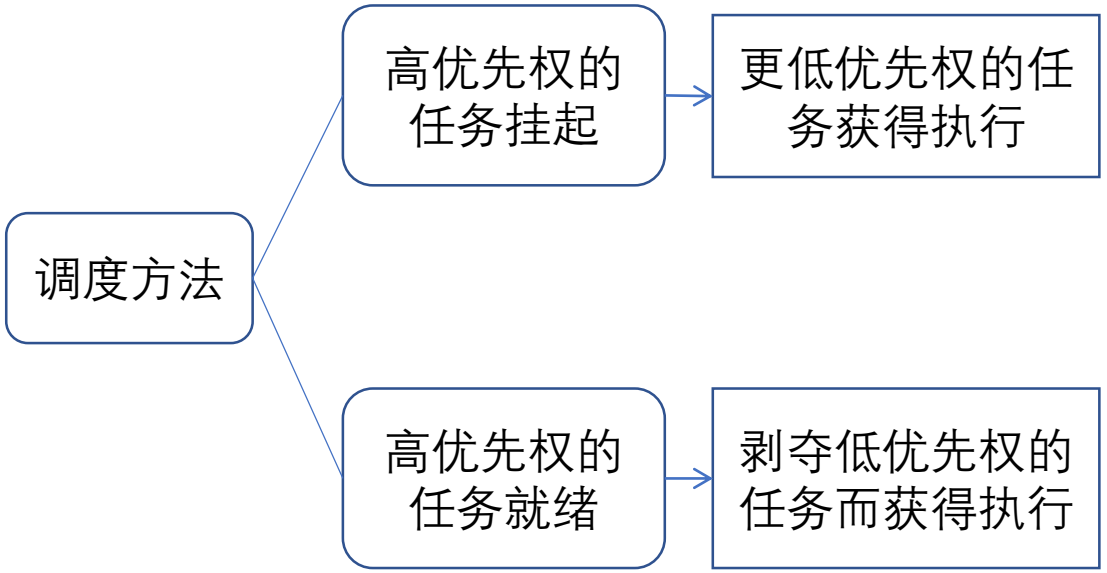
任务执行

- μC/OS是抢占式实时多任务内核，优先级最高的任务一旦准备就绪，则拥有CPU的所有权开始投入运行。
- μC/OS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。

调度过程

- 调度工作的内容可以分为两部分：最高优先级任务的寻找和任务切换。
- 最高优先级任务的寻找是通过建立就绪任务表来实现的。 μC/OS中的每一个任务都有独立的堆栈空间，并有一个称为任务控制块TCB(Task Control Block)的数据结构，其中第一个成员变量就是保存的任务堆栈指针。
- 任务调度模块首先用变量OSTCBHighRdy记录当前最高级就绪任务的TCB地址，然后调用OS_TASK_SW()函数来进行任务的上下文切换。

任务调度

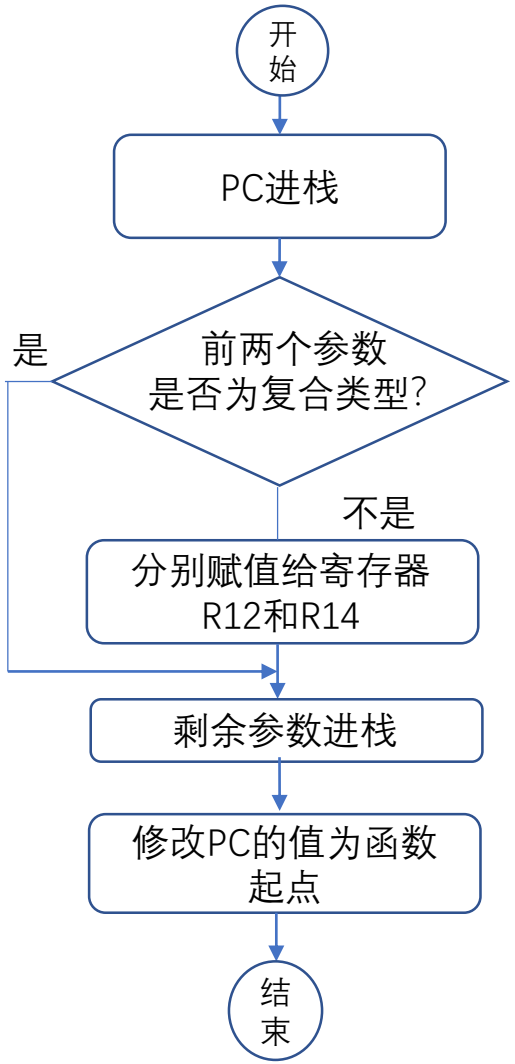


- μC/OS-Ⅱ采用的是可抢夺型实时多任务内核。可抢夺型的实时内核在任何时候都运行就绪了的最高优先级的任务。
- μC/OS-Ⅱ的任务调度是完全基于任务优先级的**抢占式调度**，也就是最高优先级的任务一旦处于就绪状态，则立即抢占正在运行的低优先级任务的处理器资源。为了简化系统设计，μC/OS-Ⅱ规定所有任务的优先级不同，因为任务的优先级也同时唯一标志了该任务本身。
- 分别发生在系统服务和时钟中断服务程序中的**两种调度方法**：

  （1）高优先级的任务因为需要某种**临界资源**，主动请求挂起，让出处理器，此时将调度**就绪状态**的低优先级任务获得执行，这种调度也称为任务级的**上下文切换**。

  （2）高优先级的任务因为时钟节拍到来，在**时钟中断**的处理程序中，内核发现高优先级任务获得了执行条件(如休眠的时钟到时)，则在中断态直接切换到高优先级任务执行。这种调度也称为中断级的**上下文切换**。

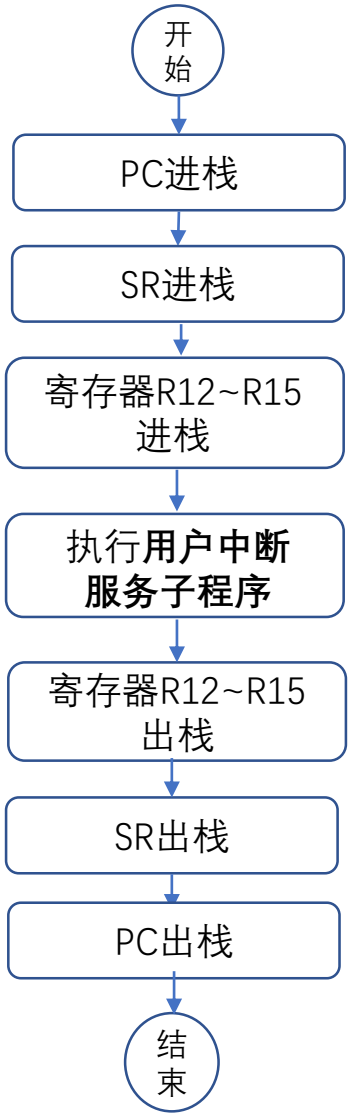
函数调用



- 如果是函数级调用，编译器会在函数调用时先把当前函数PC压栈，然后调用函数，PC值改变。
- 如果被调用的函数带有参数，那么，编译器按照以下的规则进行。
- 最左边的两个参数如果不是struct或者union类型，将被赋值到寄存器，否则将被压栈。函数剩下的参数都将被压栈。
- 根据最左边的那两个参数的类型，它们分别赋值给R12（对于32位类型赋值给R12:R13）和R14(对于32位类型赋值给R14:R15)。

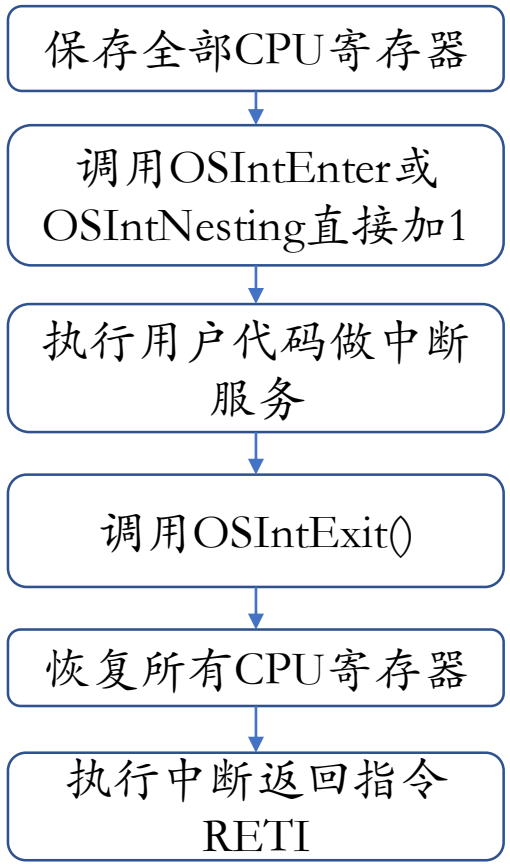
* 这里假设采用的C编译器为IAR Embedded WorkBench

中断调用



- 如果是在中断中调用中断服务子程序的话，编译器将把当前执行语句的**PC压栈**，同时再把**状态寄存器SR压栈**。接着，根据中断服务子程序的复杂程度，选择把**寄存器R12~R15压栈**。
- 然后，执行中断服务子程序。
- 中断处理结束后把**寄存器R12~R15 出栈**，**SR出栈**，**PC出栈**。把系统恢复到中断前的状态，使程序接着被中断的部分继续运行。

- μC/OS中，中断服务子程序要用汇编语言来写。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。
- **用户中断服务子程序**框架:



定时中断的服务程序:

```
void OSTickISR(void)
{
    保存处理器寄存器的值;
    调用OSIntEnter()或是将OSIntNesting加1;
    调用OSTimeTick();
     调用OSIntExit();
    恢复处理器寄存器的值;
    执行中断返回指令RETI;
}
```

- 为了实现资源共享，一个操作系统必须提供临界段(Critical Sections)操作的功能。
- μC/OS-Ⅱ为了处理临界段代码需要关中断，处理完毕后再开中断。这使得μC/OS-Ⅱ能够避免同时有其它任务或中断服务进入临界段代码。
- μC/OS-Ⅱ定义两个宏(macros)来开关中断。分别是：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。

```
void OSIntEnter (void)
{
    OS_ENTER_CRITICAL();
    OSIntNesting++;
    OS_EXIT_CRITICAL();
}
```

OSIntNesting为中断嵌套的层数，只有OSIntNesting为0，系统才会进行任务调度。

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

任务切换

任务级的任务切换原理

```
void OSSched(void){
  关中断;
  if (不是中断嵌套并且系统可以被调度)
  {
      确定优先级最高的任务;
      if (最高级的任务不是当前的任务)
      {
          调用OSCtxSw();
      }
   }
  开中断;
}
```

```
void OSCtxSw(void)
{
   将SR及R1~R4压栈;
   OSTCBCur→OSTCBStkPtr = SP;
   OSTCBCur  = OSTCBHighRdy;
   SP  = OSTCBHighRdy →OSTCBSTKPtr;
   将SR及R4~R1从新堆栈中弹出;
   执行中断返回指令RETI;
}
```

源码:

```
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++;
            OSCtxSw();                  //OS_TASK_SW();
         }
    }
    OS_EXIT_CRITICAL();
}
```

== 谢谢 ==