Ch4 进程调度基础

本章将围绕操作系统进程调度相关问题，具体探讨uC/OS-II下调度的实现方式，所涉及的数据结构和算法，是读者深入理解内核调度行为的基础，也是后续开展综合性课程设计的前提。

**预备知识**

4.1 调度相关数据结构

4.1.1 再看任务控制块

在第二章，我们已初步认识了任务控制块结构体OS\_TCB。任务是uC/OS-II的基本调度单位，而TCB的作用类似进程控制块PCB，是任务管理以及调度产生作用的具体对象。那么调度主要涉及到哪些成员变量呢？

来看OS\_TCB的这部分：

typedef struct os\_tcb {

……

struct os\_tcb \*OSTCBNext; //指针指向下一个TCB

struct os\_tcb \*OSTCBPrev; //指针指向上一个TCB

……

INT8U OSTCBX; //用于快速访问任务就绪表

INT8U OSTCBY; //用于快速访问任务就绪表

OS\_PRIO OSTCBBitX; //用于快速访问任务就绪表

OS\_PRIO OSTCBBitY; //用于快速访问任务就绪表

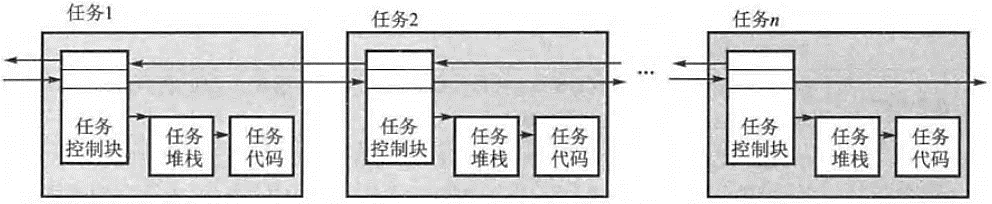
……

INT32U OSTCBCtxSwCtr; //任务被切换次数

INT32U OSTCBCyclesTot; //任务已运行时钟周期数

……

} OS\_TCB;

成员变量OSTCBNext和OSTCBPrev主要用于构建TCB双向链表：

TCB双向链表联结了系统中**当前正在执行的所有任务**，故称活动链，它是由内核实时维护的。它是如何建立的？

考虑最开始的情形。在ucos\_ii.h中，有相关全局变量定义：

OS\_TCB \*OSTCBCur; /\* Pointer to currently running TCB \*/

OS\_TCB \*OSTCBFreeList; /\* Pointer to list of free TCBs \*/

OS\_TCB \*OSTCBHighRdy; /\* Pointer to highest priority TCB R-to-R\*/

OS\_TCB \*OSTCBList; /\* Pointer to doubly linked list of TCBs\* /

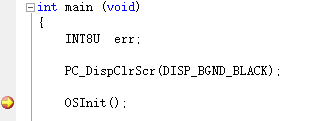
OS\_TCB \*OSTCBPrioTbl[OS\_LOWEST\_PRIO + 1u]; /\* Table of pointers to created TCBs \*/

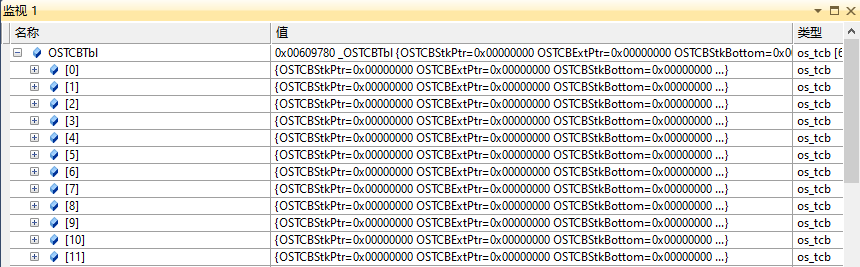
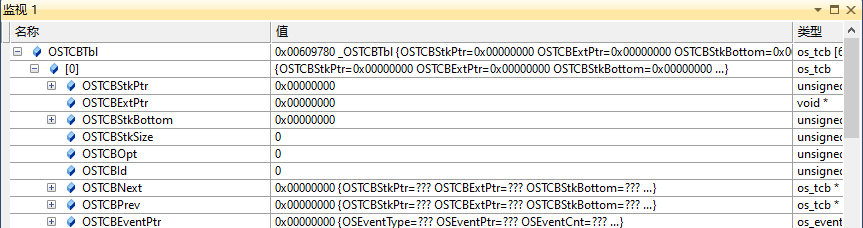
OS\_TCB OSTCBTbl[OS\_MAX\_TASKS + OS\_N\_SYS\_TASKS]; /\* Table of TCBs \*/

**系统启动时，在执行main函数前已完成所有全局变量的内存分配**。蓝色部分语句表明，在执行系统初始化OSInit()前，会开辟出一个长度为OS\_MAX\_TASKS + OS\_N\_SYS\_TASKS的TCB结构体数组，以及一个长度为OS\_LOWEST\_PRIO + 1u的指针数组。两个数组大小默认都是**64**。

思考：既然长度一样，为什么定义数组长度的常量不一样？

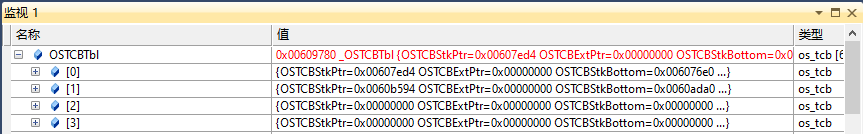
（OSTCBPrioTbl数组通过使用OS\_LOWEST\_PRIO强调了它与任务优先级是一一对应关系；而OSTCBTbl数组长度受限于最大任务数，其中元素下标与优先级没有必然联系。）

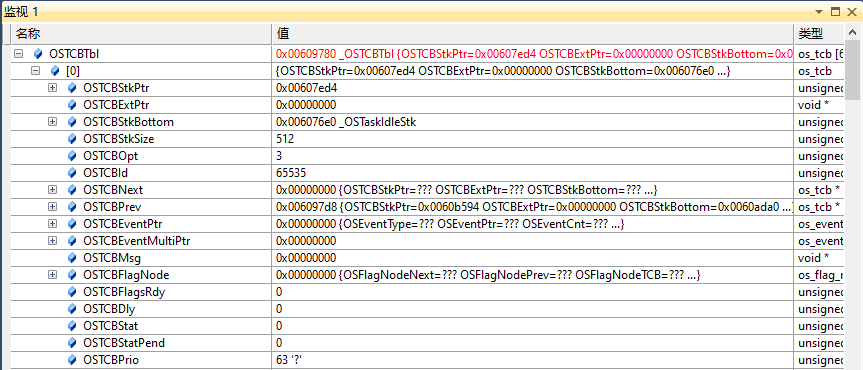
**双向链表在OSInit()前是没有的**。为了证明这一点，我们在OSInit()语句处设置断点（点击左侧），然后按F5启动调试：

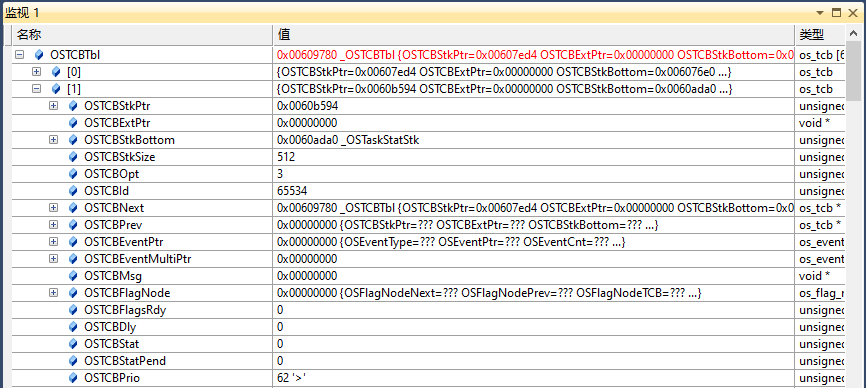
添加变量OSTCBTbl监视：

可以看到64个元素全为空，点进任一元素内OSTCBNext与OSTCBPrev指针都是空值，说明双向链表尚未建立。

此时监视全局指针OSTCBFreeList与OSTCBList也是空值。

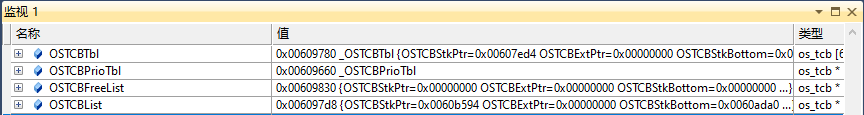
接着单步执行。按下F10后：

可以看出OSTCBTbl数组前两个元素已经被改写。分别点进去看。第一个元素：

它的优先级为63，说明是空闲任务。它的后向指针为空，前向指针有明确的值6097d8(=&OSTCBTbl[1])。第二个元素：

优先级为62，是统计任务。它的前向指针为空，后向指针有值609780，等于整个数组首地址，即&OSTCBTbl[0]。

此时监视OSTCBFreeList与OSTCBList为：

OSTCBFreeList指向了OSTCBTbl[2]，OSTCBList指向OSTCBTbl[1]。

我们得到结论：**OSTCBFreeList指向OSTCBTbl数组中下一个空白块，而OSTCBList指向最近使用的块**。

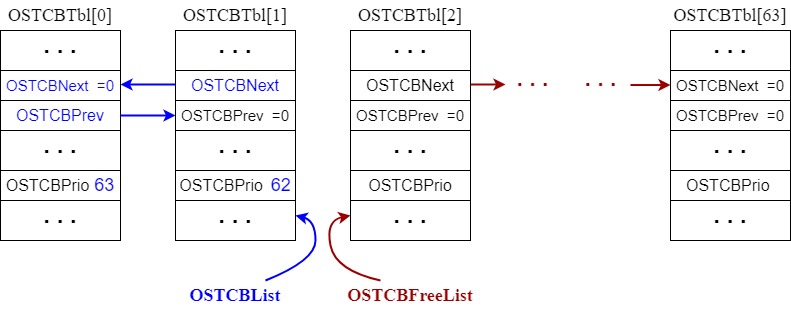
通过以上比较分析，可以梳理出内核函数OSInit()有关TCB链表的初始化动作有：

1）首先创建空闲任务，分配TCB，更改块中成员变量的值，空闲任务TCB从OSTCBFreeList中脱链；

2）接着创建统计任务（默认开启情况下），分配TCB，更改块中成员变量的值，使得与空闲任务构成双向链，统计任务TCB从OSTCBFreeList中脱链；

3）OSTCBFreeList指向下一个空白块OSTCBTbl[2]；

4）OSTCBList指向OSTCBTbl[1]即统计任务的TCB。

下图展现了此时OSTCBTbl[]数组的大致情况：

除当前活动块链表OSTCBList以及空闲块链表OSTCBFreeList外，指针数组OSTCBPrioTbl按优先级保存了已创建任务的TCB地址。可以在调试中添加OSTCBPrioTbl监视加以验证。

4.1.2 任务就绪组和任务就绪表

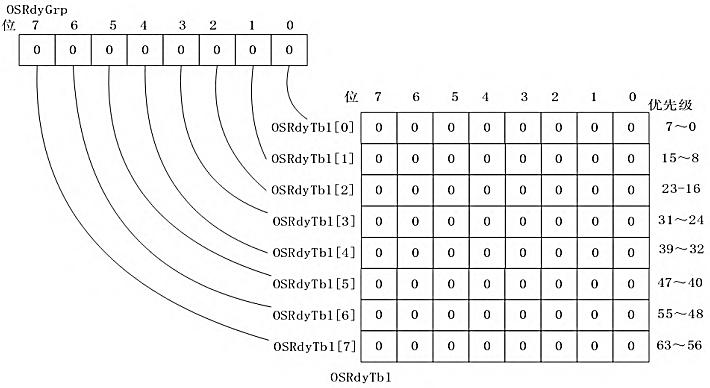
uC/OS-II通过链表实时维护所有任务的TCB，但是很多内核动作并不会直接基于链表完成。设想如下场景需求——内核要获取当前优先级最高的就绪任务，于是通过表头OSTCBList进行全链表遍历，不但要验证TCB的成员变量OSTCBStat（是否为0），还要记录已就绪TCB中的最小OSTCBPrio值（最高优先级）。**显然这样做是非常耗时的，内核不会采用这么笨重的办法。**

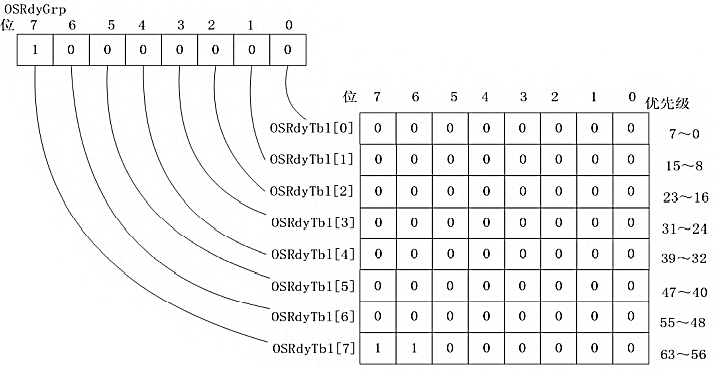
全局任务就绪组（表）被专门用来查找当前已就绪最高优先级任务。其结构与第三章事件就绪组（表）类似，可通过位运算来加速操作。在ucos\_ii.h中有如下定义：

OS\_PRIO OSRdyGrp; /\* Ready list group \*/

OS\_PRIO OSRdyTbl[OS\_RDY\_TBL\_SIZE]; /\* Table of tasks which are ready to run \*/

其中OS\_RDY\_TBL\_SIZE大小为OS\_LOWEST\_PRIO/8+1（=8）。

设置断点，进入调试。在调用OSInit()前，就绪组与就绪表为：

此时尚未创建空闲任务与统计任务，故对应表中位置都是0。单步**过OSInit()**后变成：

相应位置已被置1。

思考：如何根据任务的优先级在就绪表中登记？

（设某任务优先级为30。一般的想法是，先30/8=3，找到OSRdyTbl[3]行，再30%8=6，定位此行的D6位。**有没有更好的方法？**）

4.1.3 对任务就绪表的操作

uC/OS-II实际操作就绪表时，使用了特别的方法。在4.1.1节，留下4个成员变量——**OSTCBX**、**OSTCBY**、**OSTCBBitX**以及**OSTCBBitY**还没有说明具体用途。下面来看作者J. J. Labrosse是怎么使用它们的。

当新创建了一个任务时，OSTaskCreate()在进行一系列必要的检查后，主要做了两件事：

1）调用OSTaskStkInit()初始化任务栈；

2）调用OS\_TCBInit()初始化TCB。

关于任务栈，放到后续章节讨论，本章主要关注OS\_TCBInit()。这个函数位于内核核心模块os\_core.c的最后，这里给出它的主要部分：

INT8U OS\_TCBInit (INT8U prio, OS\_STK \*ptos, OS\_STK \*pbos, INT16U id,

INT32U stk\_size, void \*pext, INT16U opt)

{

……

ptcb = OSTCBFreeList; //从空闲TCB链表中取下一个块

if (ptcb != (OS\_TCB \*)0) {

OSTCBFreeList = ptcb->OSTCBNext; //空闲链表头后移

ptcb->OSTCBStkPtr = ptos; //更新任务栈

ptcb->OSTCBPrio = prio; //更新优先级

ptcb->OSTCBStat = OS\_STAT\_RDY; //新任务状态为就绪

ptcb->OSTCBStatPend = OS\_STAT\_PEND\_OK; //新任务无等待事件

ptcb->OSTCBDly = 0u; //新任务无延时

……

ptcb->OSTCBY = (INT8U)(prio >> 3u);

ptcb->OSTCBX = (INT8U)(prio & 0x07u);

ptcb->OSTCBBitY = (OS\_PRIO)(1uL << ptcb->OSTCBY);

ptcb->OSTCBBitX = (OS\_PRIO)(1uL << ptcb->OSTCBX);

……

OS\_ENTER\_CRITICAL();

OSTCBPrioTbl[prio] = ptcb; //更新全局指针数组（保存此TCB地址）

OS\_EXIT\_CRITICAL();

OS\_ENTER\_CRITICAL();

ptcb->OSTCBNext = OSTCBList; //接入全局活动TCB链表

ptcb->OSTCBPrev = (OS\_TCB \*)0; //前向指针为空

if (OSTCBList != (OS\_TCB \*)0) {

OSTCBList->OSTCBPrev = ptcb; //上一个块的前向指针指向此TCB

}

OSTCBList = ptcb; //活动TCB链表头指向此TCB

OSRdyGrp |= ptcb->OSTCBBitY;

OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;

OSTaskCtr++; //更新全局任务计数

OS\_EXIT\_CRITICAL();

return (OS\_ERR\_NONE);

}

}

涉及4个成员变量的语句为蓝色部分。大致可以看出，这4个成员变量与更新就绪表、就绪组有关。我们从上往下看起。

这里不妨设被创建的任务优先级为30，即prio=30，二进制是00011110。首先是右移3位，有prio >> 3u = 00000011，OSTCBY取值3。

然后取prio的最低3位，prio & 0x07u = 00000110，OSTCBX取值6。

大家马上会想到，X即30在就绪表中的横坐标，Y是纵坐标。事实上，**作者这样命名也是为了体现出在二维的位表中坐标的概念**。

接下来，00000001左移OSTCBY位赋给OSTCBBitY，00000001左移OSTCBX位赋给OSTCBBitX。OSTCBBitY取值00001000，OSTCBBitX取值01000000。这两个变量是为了最后的或运算：

1）就绪组OSRdyGrp与OSTCBBitY相或，置OSRdyGrp的D3位为1；

2）就绪表第3行OSRdyTbl[3]与OSTCBBitX相或，置此行的D6位为1。

这样，优先级为30的任务就在就绪组和就绪表中对应位置完成了登记。

回到刚刚那个问题，用4个成员变量、6次位运算的代价，真的要优于2次算术运算吗？如果只是定位登记，可能并不能体现这里的优势。作者这样做的主要目的，**在于快速查找优先级最高的任务，而这是调度器最频繁的动作**。

uC/OS-II是基于优先级抢占的实时操作系统，其调度准则为“**在每个节拍到来时尽可能让优先级最高的就绪任务上台运行**”。高优先级任务可以抢占低优先级任务的运行权，是实时性的体现，显然这需要高频次的查询任务优先级高低的动作。

4.2 uC/OS-II的任务调度器

本节将介绍任务调度中最重要的一环——调度器。

uC/OS-II调度器具体有2个——OS\_Sched和OSIntExit。前者面向**任务级别的调度**，后者面向**中断级别的调度**。所谓任务级别的调度，指在任务执行中主动触发的调度；而中断级别的调度，指在中断退出的时候触发的调度。二者主要区别在系统调度行为被触发的时机上。本章主要讨论前者。

OS\_Sched定义在os\_core.c中：

void OS\_Sched (void)

{

OS\_ENTER\_CRITICAL();

if (OSIntNesting == 0u) { //不处于中断嵌套中

if (OSLockNesting == 0u) { //调度器不是锁定状态

OS\_SchedNew();

OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy]; //更新当前最高优先级就绪任务指针（直接查指针数组）

if (OSPrioHighRdy != OSPrioCur) { //如果当前任务不是优先级最高的就绪任务

OSTCBHighRdy->OSTCBCtxSwCtr++; //优先级最高就绪任务上台次数加1

OSCtxSwCtr++; //系统总切换次数加1

OS\_TASK\_SW(); //执行任务上下文切换

}

}

}

OS\_EXIT\_CRITICAL();

}

调度器代码篇幅很小：当一系列全局变量被更新后，最终由**OS\_TASK\_SW**()具体执行任务的上下文切换。因为本书实验项目基于VC2010开发，项目整体是作为Windows下的一个进程来运行的。因此，所有的任务都是位于同一进程内的Windows线程。OS\_TASK\_SW()最终指向线程切换的Win32 API，具体细节留待读者去发现。

注意到如果当前任务（即发起本次调度行为的任务）优先级OSPrioCur就是所有就绪任务中的最高优先级OSPrioHighRdy，那么不需要进行切换。

内核静态函数OS\_SchedNew()的作用是更新全局量OSPrioHighRdy，定义如下：

static void OS\_SchedNew (void)

{

INT8U y;

y = OSUnMapTbl[OSRdyGrp];

OSPrioHighRdy = (INT8U)((y << 3u) + OSUnMapTbl[OSRdyTbl[y]]);

}

以上两条语句即查询当前优先级最高就绪任务的全部实现。

直观上，这两条语句显然与任务就绪组和就绪表有关。细看的话，就绪组与就绪表本身又作为查**OSUnMapTbl**表的索引而存在。

那么OSUnMapTbl表又是什么？在os\_core.c的开头有：

INT8U const OSUnMapTbl[256] = {

0u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,

4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,

5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,

4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,

6u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u,

……

};

得知OSUnMapTbl是一个常量数组，称为Priority Resolution Table。

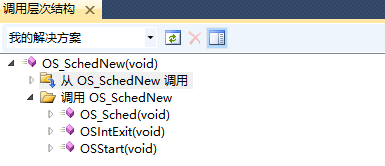
作者J. J. Labrosse在这里体现了他深厚的编程功力。他将查找就绪表中最高优先级等价为对常量数组的2次直接查表。读者可能会有疑问，这样做值得吗？

不妨来看通常的查找算法：就绪表OSRdyTbl[]按数组元素下标升序循环一遍，直到元素非零为止；接着再确认最低的“1”出现在哪个二进制位。且不论这样做的时间复杂度，一个更为关键的约束是，“**实时操作系统应保证其内核动作必须可确认完成时间**”。而含有循环的代码往往是无法满足这一约束的。

思考：为什么对就绪表的循环查找语句不能保证完成时间？

（因为就绪表中“1”的分布具有随机性，你可能在循环一开始就找到代表优先级最高的那个“1”，也可能在循环最后才找到。）

需要说明的是，在对位运算的使用上，ucos-ii是要优于后续版本ucos-iii的。ucos-iii并没有使用上述查常量表的方式获取当前就绪最高优先级任务（截止到版本3.08.01）。这也导致目前很多高实时性项目仍然采用ucos-ii。读者可以通过对比ucos-iii内核调度器OSSched()验证这一点。

另一个有意思的问题是，哪里需要用到OS\_SchedNew()？通过鼠标右键点击此函数-“查看调用层次结构”-“调用OS\_SchedNew”，

发现只有3处调用了OS\_SchedNew()：

1）OS\_Sched()函数内，这个前文已解释；

2）OSIntExit()函数内，显然中断级别的调度也需查询最高优先级就绪任务；

3）OSStart()函数内，由OS\_SchedNew()获取起始任务的优先级（因为它总是高于空闲和统计任务）并设置相关全局变量。

也就是说，除系统启动OSStart()外，两个调度器都必须使用OS\_SchedNew()。

**实验内容**

4.3 建立工程APP04

APP04给出了读者-写者问题的基本解决。各个任务说明及参数如下表：

|  |  |  |  |
| --- | --- | --- | --- |
| 任务名 | 说明 | 优先级 | OSTimeDly()参数 |
| TaskStart | 起始任务 | 0 | 400 |
| MyTask1 | **读者** | **10** | **150** |
| MyTask2 | **无** | **11** | **120** |
| MyTask3 | **待实现时间片轮转** | **12** | **90** |
| MyTask4 | **向Task5发送消息** | **13** | **70** |
| MyTask5 | **写者 收到Task4消息后再写** | **14** | **30** |
| TaskClock | 时钟任务 | 30 | 50 |

4.3.1 新建工程

1、在VC2010新建项目窗口选择“Win32控制台应用程序”，取消“预编译头”，勾选“空项目”。工程项目属性配置“常规”-“字符集”下选择“使用多字节字符集”。

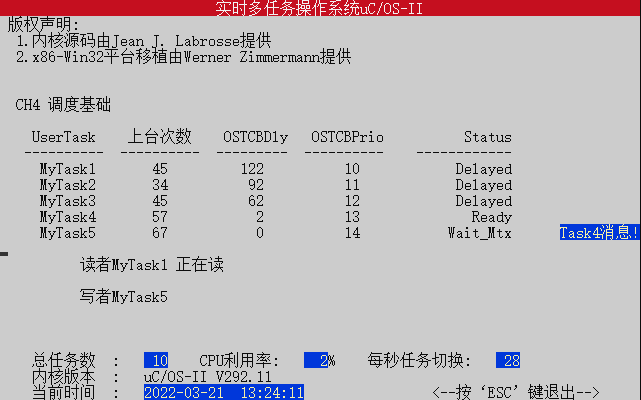
2、使用APP04工程的源码。将App文件夹拷入工程目录下。在“源文件”下添加现有项Test.c和app\_linked\_list.c，在“头文件”下添加现有项app\_cfg.h、include.h、os\_cfg.h以及app\_linked\_list.h。

将Port文件夹和Source文件夹完整拷入工程目录下。在项目中新建筛选器“Port”以及“Source”，分别将上述二个文件夹中的所有文件按现有项添加。

3、在项目配置属性“C/C++”-“常规”-“附加包含目录”下添加App、Port和Source目录的完整路径。

4、在配置属性“链接器”-“输入”-“附件依赖项”下添加winmm.lib。

4.3.2 运行界面

工程APP04运行界面如下：

4.3.3 对工程APP04的说明

1、APP04在上一章APP03基础上增加了调度相关信息的显示，包括：用户任务的累计上台次数、系统总任务切换次数。请留意涉及的全局变量。

2、APP04中解决读者-写者问题的方式并不完美。在读者较多情形下（≥2），写者得不到写入机会。可以增加一个读者来验证这一点。

3、在MyTask3中将初步实现一个简单的时间片轮转。实现细节与要求在4.4.4节。

4.4 准备改造uC/OS-II调度器（一）

前面已大致了解了uC/OS-II的任务级调度器OS\_Sched()，本节将讨论改造这个调度器的可能性，以及必要的准备工作。**我们的最终目标是在uC/OS-II中实现时间片轮转**。

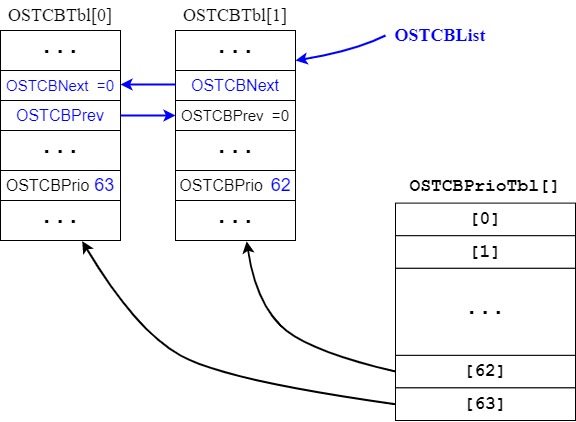
所谓时间片轮转，是指在相同优先级之间考虑执行均衡性的策略，不同优先级的任务是不存在时间片轮转的。在第二章讨论优先级时，一个重要的结论是**uC/OS-II中每个任务的优先级都是唯一的，任两个任务间优先级都有高低之分**。这种设定有它好的方面，就是实现起来相对简单。不足之处在于这导致了uC/OS-II调度策略的单一化。

那么，如果允许任务的优先级相同，会影响哪些地方？

4.4.1 对全局数组OSTCBPrioTbl[]的讨论

与优先级关系最紧密的数据结构是全局数组OSTCBPrioTbl[]。

从4.1.1节可知，结构体数组OSTCBTbl[]的索引与优先级无关，而指针数组OSTCBPrioTbl[]是按优先级索引的。

下图表示了数组OSTCBPrioTbl[]的作用。

当前时刻已完成系统初始化，但尚未创建用户任务。OSTCBPrioTbl[62]与OSTCBPrioTbl[63]已分别保存了统计任务与空闲任务的TCB地址。

OSTCBPrioTbl[]的作用就是按优先级保存已创建任务的TCB地址。其目的在于内核可根据某一优先级快速找到所对应的TCB。

可以通过查找引用的方式简单衡量内核使用两个数组的情况。在ucos\_ii.h中分别对两个数组定义处“右键点击”-“查找所有引用”，有：

|  |  |
| --- | --- |
| 全局数组 | 在内核主要文件中的引用 |
| OSTCBTbl | 共 6 处 |
| OSTCBPrioTbl | 共 41 处 |

OSTCBPrioTbl使用次数大大超过了OSTCBTbl。直接原因是，如果内核想要立即获取一个任务的TCB指针，最快的方法应该：

ptcb = OSTCBPrioTbl[prio];

即以优先级prio为索引找到对应的数组元素（而不是通过遍历图中蓝色的链表OSTCBList）。

根据任务优先级找到TCB的方式，在ucos-ii内核中被广泛使用。**但如果允许任务优先级相同，这种获取TCB的方式将无效。**事实上大多数主流操作系统并不会设定优先级唯一，用优先级索引任务是ucos-ii的特点。后续版本ucos-iii已经取消了优先级唯一的设定。

思考：主流操作系统中用什么来体现进程的唯一性？（提示：PID）

4.4.2 对全局数组OSTCBTbl[]的讨论

要改成优先级相同，还需明确数组OSTCBTbl[]的作用。上一节中提及了内核对此数组的6处引用，主要集中在OS\_InitTCBList()函数中：围绕活动TCB链OSTCBList以及空闲TCB链OSTCBFreeList的初始化显然与OSTCBTbl[]有关。

OSTCBTbl[]本身作为全局的资源看待。在其上的操作主要还是通过活动链和空闲链。

空闲链OSTCBFreeList在两处使用：

1）OS\_TCBInit()函数中——此函数只有创建新任务时才调用，新任务需要从空闲链上取下一个新的块。

2）OSTaskDel()函数中——删除任务要将TCB块归还给空闲链。

而活动链OSTCBList在以下四处使用：

1）OS\_TCBInit()中——从空闲链上取下的新块需要插入活动链。

2）OSTaskDel()中——被删任务TCB应从活动链上脱链。

3）OSTimeTick()中——时钟节拍中断服务函数。在每个节拍到来，遍历活动链中的所有TCB使其OSTCBDly减一。

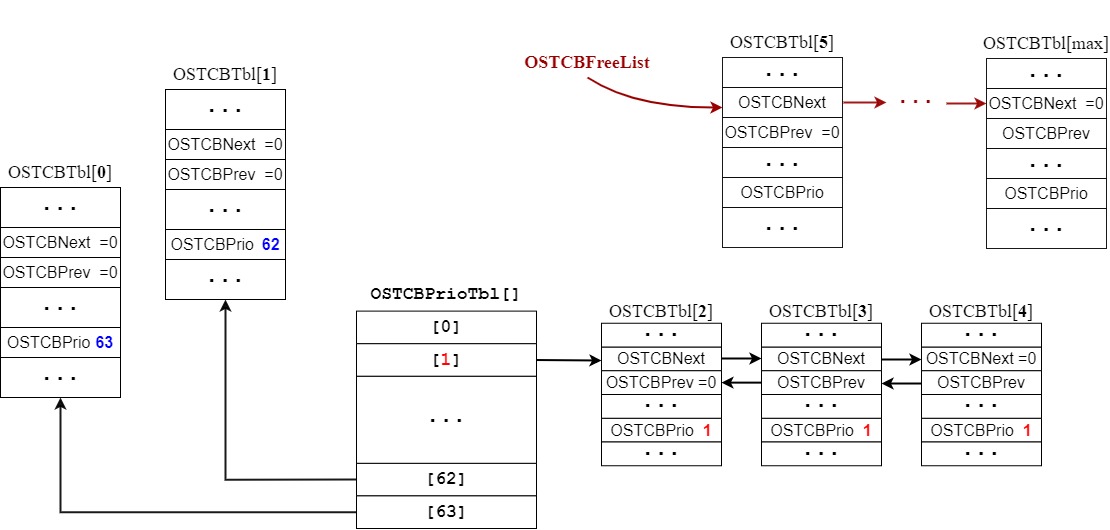
4）DumpTaskList()中——与移植平台有关，暂不讨论。

4.4.3 改成允许相同优先级

**在相同优先级下，OSTCBPrioTbl[]数组元素存放的不只是一个TCB的地址，而是由若干个TCB组成的链表**。换句话说，设有相同优先级*k*的*n*个任务TCB组成链表，指针OSTCBPrioTbl[*k*]指向了此链表的表头，链表含*n*个结点。

具体设计思路是每当创建一个新任务，从空闲链上取下一个块后根据优先级*k*插入对应的链表OSTCBPrioTbl[*k*]。活动链OSTCBList此时将失去作用，可以删除。

下图反映的时刻是，系统初始化后依次创建了3个优先级为1的任务。这些任务的TCB在OSTCBPrioTbl[1]处组成双向链表。



最后总结一下这样做的后果：

1）根据优先级找对应的TCB将失效。因为一个优先级下可能不只一个任务。可以妥协为先根据优先级找到对应的TCB链，然后基于链表的基本操作定位到目标TCB。

2）4.2节中函数OS\_SchedNew()的作用将被重新定义。即由“获取当前最高优先级就绪任务”改为“获取当前就绪任务中的最高优先级”。全局量OSPrioHighRdy只能索引到链表OSTCBPrioTbl[OSPrioHighRdy]。至于此优先级下哪个任务是需要调度上台的，应基于状态和剩余时间片等综合判断。

3）全局活动链OSTCBList将被删除，主要影响了OSTimeTick()中对全体活动任务的遍历。可以妥协为通过遍历OSTCBPrioTbl数组实现，代价是耗时增加。

以上后果只是粗略估计，更多细节问题留待后续章节进一步讨论。这种设计会大量需求结点的插入、删除以及遍历等基本链表操作。

4.4.4 在用户程序中的尝试

为了不影响uC/OS-II内核的正常功能，我们尝试在用户程序上实现以上设计。在Test.c中定义了3个全局量：

APP\_TCB AppTCBTbl[128]; //全局APP\_TCB数组 含128个块

APP\_TCB \*AppTCBFreeList; //空闲APP\_TCB链头

APP\_TCB \*AppTCBPrioTbl[64]; //对应64个优先级的指针数组 用来连接APP\_TCB双向链表

其中AppTCBTbl[]比照OSTCBTbl[]，AppTCBFreeList比照OSTCBFreeList，而AppTCBPrioTbl[]比照OSTCBPrioTbl[]。

APP\_TCB结构体用来仿真OS\_TCB，最主要的区别是它增加了一个成员变量用以表示当前剩余的时间片。

我们增加了一个双向链表模块app\_linked\_list。链表结点定义成APP\_TCB。链表模块内实现了一个基本方法——APPInsertListTail，即将结点插入链尾。

有了以上定义和方法后，在Test.c中增加了如下3个新函数：

1）InitAppTCBList()，作用类似内核函数OS\_InitTCBList()。它完成了AppTCBTbl[]数组的初始化，并提供空闲APP\_TCB链头AppTCBFreeList。

**此函数已经实现，并于系统初始化后调用**。

2）AppTCBInit()，作用类似内核函数OS\_TCBInit()。它完成新APP\_TCB块的划分以及初始化，并将新块插入对应优先级的双向链表链尾。

**此函数已经实现，并于MyTask3中进入循环前调用**。

3）AppSchedRR()，用来实现时间片轮转。具体地，如果当前位于链首的任务块时间片已用完，将其送往链尾，并取链首后一个任务块上台执行。

**此函数尚未实现**。

在MyTask3的循环中调用AppSchedRR()。也就是说，将MyTask3上台时间间隔当作时间片基本单位。每次上台，将执行一次时间片轮转。

下图反映了在10个任务块上进行时间片为2的轮转情况。这些任务块的ID由1000至1009，且优先级同是12。



**附：实验报告——实验4**



**计算机与信息 学院实验报告**

|  |  |  |
| --- | --- | --- |
| 实验课程： | 操作系统实验 | |
| 实验编号： | 4 | |
| 实验名称： | 进程调度基础 | |
| 实验人员： | 学号 |  |
| 姓名 |  |
| 班级 | 20\*\*\* |
| 指导教师： |  | |
| 实验室： |  | |
| 实验日期： | 2022年3月31日 | |

【注】每次实验后，将此电子版实验报告与指定文件打包整体上传。

一、实验目的

1．熟悉μC/OS-II任务调度相关数据结构，了解任务级调度器的基本流程。

2．掌握VC环境下分析与调试工程项目的基本方法。

3．理解时间片轮转算法思想，以及时间片与优先级之间的关系。

4．掌握双向链表的基本结构与操作方法。

二、实验内容与要求

1．阅读实验指导书。新建工程App04，调试运行并回答以下问题：

1）分别在main函数中OSInit();和OSStart();语句处设置断点，监视相关变量在语句执行前后的变化。

2）右键点击os\_core.c文件中OS\_Sched函数名-“查看调用层次结构”-“调用OS\_Sched”，分析哪些地方会使用调度器OS\_Sched。

3）在本例程中，我们增加了2个TASK\_USER\_DATA结构体的成员，对应了OS\_TCB的成员变量OSTCBCtxSwCtr和OSTCBCyclesTot。在输出界面中将OSTCBDly栏替换为OSTCBCyclesTot，观察有什么现象。

2．根据4.4.4节，在指定位置实现如下函数：

1）在双向链表头部插入结点——

APPInsertListHead (APP\_TCB \*\*head, APP\_TCB \*ptcb)

2）在双向链表上按时间片轮转——

AppSchedRR(INT8U prio)

实现时间片轮转应该在界面中有所体现，例如按4.4.4节图中的方式。

3．在控制台适当位置输出开发者信息（学号、姓名）。

4．提交app\_linked\_list.c、Test.c和项目生成的exe文件，实验报告。

三、设计步骤（学生填写）

1．概要设计（文字）

1）打开VS2010，建立工程app04。将app04工程的xxx属性值作如下调整

**……**

2）为实现xxx功能，需要修改xxx文件的xxx函数（部分），理由是

**……**

3）

**……**

**……**

2．详细设计（文字+关键代码）

为实现xxx功能，在xxx.c文件中xxx函数（部分）作如下关键代码更改：

**……**

四、结果分析（学生填写）

1．问题分析（简要文字+列表）

对问题1 - 1），

|  |  |  |
| --- | --- | --- |
| 变量 | 在 OSInit(); 处 | 在 OSStart(); 处 |
| OSTCBTbl |  |  |
| OSTCBPrioTbl |  |  |
| …… |  |  |

对问题1 - 2），

|  |  |
| --- | --- |
| 调用位置 | 分析调用原因 |
| OSTimeDly的最后 |  |
| …… |  |

对问题1 - 3），现象是

2．实现函数（部分代码）

2 - 1）

2 - 2）

输出结果（截图）