Ch3 进程同步问题

**预备知识**

3.1 同步与事件

3.1.1 任务间的同步

uC/OS-II中的任务（对应进程）在运行过程中，有时会受到来自其它任务的影响。当一件工作需要多个任务互相协作来完成时，或者当多个任务需要竞争系统中有限的资源时，任务之间便形成某种制约关系。考虑如下两种情形：

任务A与任务B进行合作。A负责向缓冲区写入数据，B负责从缓冲区中读取已写入的数据。显然，当A未写入时，B应该处于等待状态；只有当A完成写入才能通知B去读取数据。A和B之间形成了直接制约关系，这是第一种情形。

第二种情形是任务A与任务B共享系统的一台打印机。如果A正在使用打印机，则B不能再获得打印机的使用权而应等待；只有当A完成打印释放使用权后，等待打印机的B才能被唤醒并争取使用权。任务A和B因竞争同一资源而形成了间接的制约关系。

操作系统内核必须提供一种机制，使得在上述A未完成写入或A未释放使用权的时间段内，B必须要等待；而当A完成写入或A释放使用权的时刻，及时通知B以使其可由等待状态转为就绪。这种机制通常称同步机制，目的是使得存在制约关系的进程运行步调互相协调，以免出现混乱。

在uC/OS-II中，可以使用信号量、消息邮箱、消息队列等方式构建任务间的同步，它们被定义在一个统一的“事件”（Event）结构框架内。

3.1.2 事件

事件是操作系统内核对任务之间直接或间接制约关系的抽象，它表达了不同任务因需同步执行而必须建立的联系。

uC/OS-II事件结构体定义在ucos\_ii.h中：

typedef struct os\_event {

INT8U OSEventType; //事件类型

void \*OSEventPtr; //消息或消息队列指针

INT16U OSEventCnt; //信号量计数（仅信号量类型）

OS\_PRIO OSEventGrp; //等待事件的任务组

OS\_PRIO OSEventTbl[OS\_EVENT\_TBL\_SIZE]; //等待事件的任务表

} OS\_EVENT;

系统启动时，uC/OS-II会开辟一个大小为OS\_MAX\_EVENTS（默认是128）的OS\_EVENT结构体数组，作为全局共享的资源存在。即uC/OS-II中允许同时存在的事件数之和不能超过OS\_MAX\_EVENTS。

下面开始解释OS\_EVENT结构体。

首先是事件类型。成员变量OSEventType有如下取值范围：

#define OS\_EVENT\_TYPE\_UNUSED 0u //未使用（空事件）

#define OS\_EVENT\_TYPE\_MBOX 1u //事件是消息邮箱

#define OS\_EVENT\_TYPE\_Q 2u //事件是消息队列

#define OS\_EVENT\_TYPE\_SEM 3u //事件是信号量

#define OS\_EVENT\_TYPE\_MUTEX 4u //事件是互斥信号量

#define OS\_EVENT\_TYPE\_FLAG 5u //事件是信号量集（标志组）

可见事件种类涵盖了主要的同步机制。本章我们将主要关注信号量（包括互斥信号量）事件。

当事件是信号量时，成员变量OSEventPtr不再有意义，直接赋空值；成员变量OSEventCnt即为信号量的计数值。相关代码体现在创建信号量内核函数OSSemCreate()中：

OS\_EVENT \*OSSemCreate (INT16U cnt)

{

OS\_EVENT \*pevent;

……

OS\_ENTER\_CRITICAL(); //涉及全局指针需关中断

pevent = OSEventFreeList; //从空闲链中分配

……

OS\_EXIT\_CRITICAL();

if (pevent != (OS\_EVENT \*)0) { //非零说明分配成功

pevent->OSEventType = OS\_EVENT\_TYPE\_SEM; //设置事件类型是信号量

pevent->OSEventCnt = cnt; //设置信号量计数值

pevent->OSEventPtr = (void \*)0; //赋空值

OS\_EventWaitListInit(pevent); //初始化事件等待相关

}

return (pevent);

}

其中的内核函数OS\_EventWaitListInit()如下：

void OS\_EventWaitListInit (OS\_EVENT \*pevent)

{

INT8U i;

pevent->OSEventGrp = 0u;

for (i = 0u; i < OS\_EVENT\_TBL\_SIZE; i++) {

pevent->OSEventTbl[i] = 0u;

}

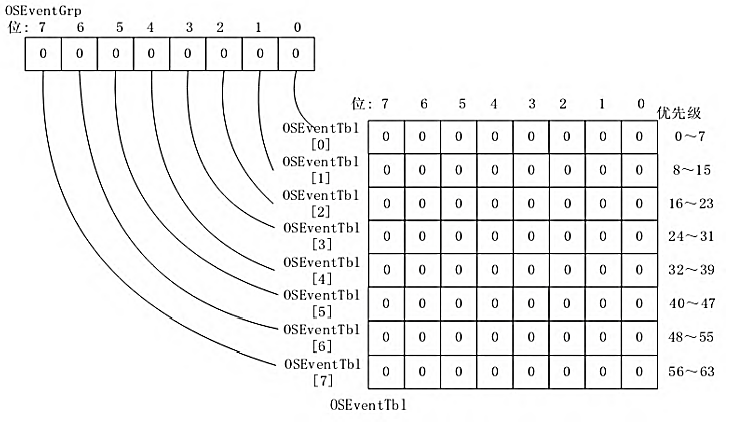
}

它的作用是将成员变量OSEventGrp以及数组OSEventTbl[]全部清零。那么为什么要这样做？回答这个问题还需要进一步了解事件控制块的概念。

3.2 事件控制块

通常将OS\_EVENT结构体称为事件控制块。与任务控制块TCB类似，体现了操作系统进程相关控制的思想。通过上一节我们已了解到成员变量OSEventType和OSEventCnt的作用。以下围绕剩余3个成员变量——OSEventPtr、OSEventGrp和OSEventTbl[]来讨论具体如何控制。

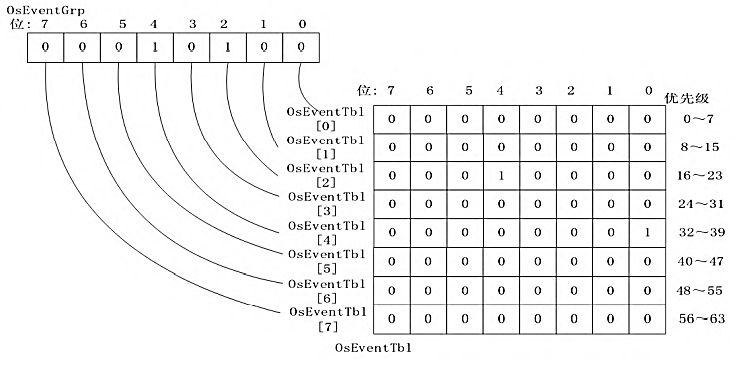
3.2.1 事件等待组和事件等待表

称成员变量OSEventGrp为事件等待组，而OSEventTbl[]为事件等待表。当一个事件开始产生作用，系统中将会有任务等待该事件的发生，那么就需要能够完整地体现等待该事件的所有任务，这项工作由OSEventGrp与OSEventTbl[]配合完成。下图说明了它们的关系。

在创建一个事件时，会将上述成员初始化为全是**0**的情形（对应了OS\_EventWaitListInit()函数调用），表明没有任务等待此事件的发生。这符合事件创建后的实际情况。事件真正产生作用仍然需要用户任务的“激活”行为。

思考：OSEventTbl[]数组的大小如何确定？（提示：参见符号常量OS\_EVENT\_TBL\_SIZE的定义）

假设现在事件已被激活，系统中有优先级为**20**和**32**的两个任务正在等待此事件的发生，在表中对应位置显示为1。此时事件等待组和事件等待表如下图：



这种结构的意义在于，内核能够整体掌握当前正在等待事件的所有任务。当事件发生时，通过效率很高的**位运算**，内核可以快速获取所有等待任务中优先级最高的任务，通知该任务转为就绪态等待调度。

思考：你能写出单字节的按位或和按位与吗？

在后续章节，我们将了解到任务就绪组和任务就绪表，它们使用了同样的结构以提升执行效率。

3.2.2 事件控制块空闲链表

系统启动时，uC/OS-II会在内存中开辟一片属于事件控制块的区域，默认含128个控制块。在操作系统初始化OSInit()中（main函数中第一句），有如下调用：

……

OS\_InitRdyList(); /\* Initialize the Ready List \*/

OS\_InitTCBList(); /\* Initialize the free list of OS\_TCBs \*/

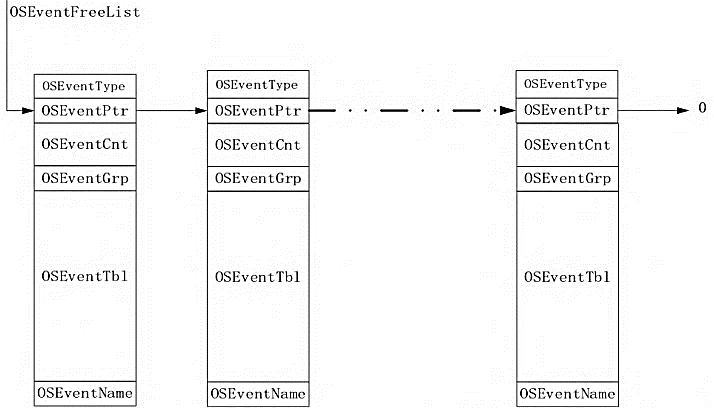
OS\_InitEventList(); /\* Initialize the free list of OS\_EVENTs \*/

……

可以看到，在完成对任务就绪链和TCB链的初始化后，接着就会对事件控制块链表进行初始化。

在内核函数OS\_InitEventList()中，主要做了两件事：1）将所有的事件控制块通过指针OSEventPtr链接成一个空闲链表；2）将全局指针OSEventFreeList指向第一个事件控制块。

下图反映了空闲链表的结构：



这里借用了成员变量OSEventPtr来构建链表。每当创建一个新事件，从以上空闲链表头部“摘下”一个块，全局指针OSEventFreeList后移；同时根据事件类型对OSEventPtr指针赋新值。具体来说，创建信号量时OSEventPtr直接赋空值，而创建消息队列时将被赋予指向一个队列。

3.3 uC/OS-II的信号量

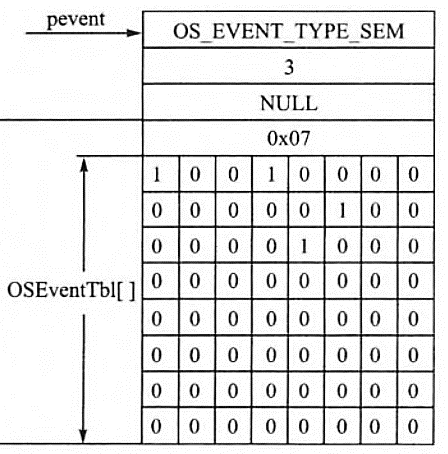
uC/OS-II内核文件os\_sem.c为单独的信号量模块，它主要有以下管理函数：

|  |  |
| --- | --- |
| **函数名** | **作用** |
| OSSemCreate | 创建信号量 |
| OSSemAccept | 无等待请求信号量 |
| OSSemPend | 有等待请求信号量 |
| OSSemPendAbort | 放弃请求信号量 |
| OSSemPost | 发送信号量 |

创建信号量的过程已在第1节说明。信号量被创建后并不意味着立刻产生作用，应在任务中使用“请求”和“发送”来激活信号量。下图给出了一个信号量应用的场景：计数值**3**表示当前资源数，等待组**0x07**表示低3组中有任务正在等待，等待表进一步表明等待任务优先级分别是**4**、**7**、**10**、**19**。

除基本的信号量请求OSSemPend与发送OSSemPost外，uC/OS-II还允许一个任务无等待的请求信号量，即当信号量无效时不进入阻塞态而可继续运行。此外，当系统发生故障时，允许任务放弃请求信号量。

本次实验将主要使用函数OSSemPend()与OSSemPost()。它们对应了最基本的**P**、**V**操作。



下例展示了两个任务间使用信号量同步的基本方法。

**例3-1** 使用信号量同步Task1和Task2。

void Task1(void \*pdata)

{

……

while (1) {

执行计算

err = OSSemPost(Semp); //发送信号量（**V**操作）

OSTimeDly(150);

}

}

void Task2(void \*pdata)

{

……

while (1) {

OSSemPend(Semp, 0, &err); //请求信号量（**P**操作）

处理结果

OSTimeDly(120);

}

}

计算任务Task1在完成计算后，利用信号量Semp通知Task2处理结果。使用信号量Semp之前需要在全局变量处定义，并在main函数中完成创建：

Semp = OSSemCreate(0);

说明是创建了一个初值为0的信号量。

那么为什么要把初值设为0呢？因为一开始并没有可供处理的结果，只有当Task1计算出来一个结果，才能让Semp加1并通知Task2；Task2请求到Semp后，会将Semp重新置0。可以把本例看作简单的生产者-消费者问题。在本章的实验部分，我们将着手较为规范地解决生产者-消费者问题。

**注意uC/OS操作系统中的信号量（包括互斥信号量）取值不会为负，即不需要通过负值判断当前是否有等待信号量的任务。如3.2节所述，在事件控制块中包含了等待信号量的所有任务信息。**

3.4 uC/OS-II的互斥信号量

uC/OS-II规定互斥信号量（Mutex）是二值信号量，即计数值只能是0或1，属于普通信号量的特例。它主要用于表达任务在访问某些共享资源时的“独占性”特征。计算机系统中存在大量需要独占访问的资源，如串口、USB、网络端口等。

内核文件os\_mutex.c是单独的互斥信号量模块，其主要管理函数与os\_sem.c类似，不再赘述。

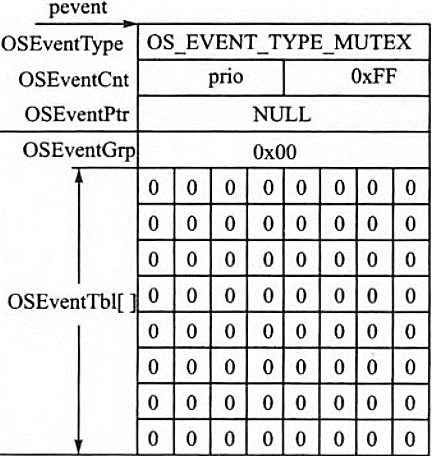
创建普通信号量时，如果使用以下参数：

Semp = OSSemCreate(1);

即表明此信号量在0和1之间取值，等同于互斥信号量。那么为什么还要专门提供一个单独的互斥信号量类型呢？

这是因为多任务在互斥地访问系统独占资源时，会出现任务**优先级反转**（Priority Inversion）的现象。简单来说是高优先级任务可能因为等待信号量，被低优先级任务所阻塞，在相当长一段时间内总是得不到CPU使用权，实际执行频度还不如那些低优先级的任务，看起来像是优先级被“反转”了一样。

有两种解决优先级反转的方法——优先级天花板（Priority Ceiling）和优先级继承（Priority Inheritance）。uC/OS-II使用实现较为简单的第一种方法：当任务A已请求到Mutex时，把A的优先级提升到比当前最高优先级用户任务还要高的级别，以期尽快执行并释放Mutex，称此优先级为优先级天花板。

我们看一下uC/OS-II具体是怎么做的。下图为Mutex型事件控制块结构：

根据图中所示，Mutex的结构与普通信号量的最大区别在成员变量**OSEventCnt**上。它被分为高8位和低8位两个部分：低8位存放计数值（仅检测是否是有效值0xFF，其它值视为无效）；高8位用来存放优先级天花板prio。当创建Mutex时，prio值由用户根据实际情况指定。

Mutex创建函数如下：

OS\_EVENT \*OSMutexCreate (INT8U prio, INT8U \*perr)

{

OS\_EVENT \*pevent;

……

pevent = OSEventFreeList; //从空闲链中分配事件控制块

pevent->OSEventType = OS\_EVENT\_TYPE\_MUTEX; //设置事件类型

pevent->OSEventCnt = (INT16U)((INT16U)prio << 8u) | OS\_MUTEX\_AVAILABLE;

pevent->OSEventPtr = (void \*)0; //赋空值 脱链

OS\_EventWaitListInit(pevent); //初始化事件等待相关

\*perr = OS\_ERR\_NONE;

return (pevent);

}

蓝色语句设置了OSEventCnt的高/低8位两个部分。

例3-2展示了两个任务使用互斥信号量访问同一临界资源的方法。

**例3-2** Task1和Task2互斥访问共享变量。

void Task1(void \*pdata)

{

……

while (1) {

OSMutexPend(Mutex, 0, &err); //请求互斥信号量（关锁）

访问共享变量

err = OSMutexPost(Mutex); //发送互斥信号量（开锁）

OSTimeDly(150);

}

}

void Task2(void \*pdata)

{

……

while (1) {

OSMutexPend(Mutex, 0, &err); //请求互斥信号量（关锁）

访问共享变量

err = OSMutexPost(Mutex); //发送互斥信号量（开锁）

OSTimeDly(120);

}

}

**实验内容**

3.5 建立工程APP03

在工程APP03中，将继续完善工程APP02的显示方式。我们隐藏了除用户任务外的其它任务，将关注点集中于用户任务的同步以及通信情况。

3.5.1 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APP03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，在“头文件”下添加现有项app\_cfg.h、INCLUDE.h、OS\_CFG.h。

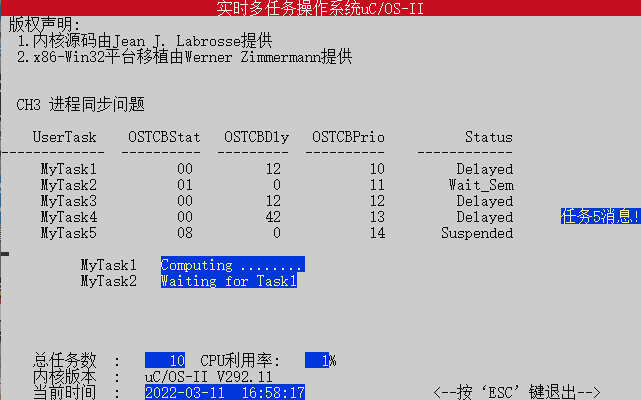
将Port文件夹以及Source文件夹完整拷入工程目录下。在项目中新建筛选器“Port”以及“Source”，对应分别将上述二个文件夹中的所有文件按现有项添加。

3、在项目属性配置“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“链接器”-“输入”-“附件依赖项”下添加winmm.lib。

3.5.2 运行界面

工程APP03运行界面如下：



可以看到，当中只显示了5个用户任务MyTask1~ MyTask5的相关信息。

3.5.3 对工程APP03的说明

1、从本章开始，起始任务TaskStart与时钟任务TaskClock将不再显示。

我们更改了main函数和TaskStart函数的写法：main函数在操作系统初始化后只创建TaskStart任务，然后直接OSStart()开始多任务调度；更多的任务创建交由TaskStart完成，这符合主流操作系统的习惯，即有一个起始进程的存在，它具有最高优先级并创建了所有用户任务，相应地，它在进入循环后执行频度很低。

2、TaskStart在进入循环前，还负责打印界面的固定部分。界面中活动部分刷新交由任务TaskDisp负责，它每35 ticks将全局数组TaskUserData[5]中的信息打印输出。

TaskUserData[5]维护了5个用户任务的当前信息。事实上，TaskUserData[5]的更新由统计任务负责，统计任务执行频率很高不适合用来刷新显示。**我们将在之后的工程中采用这种更新数组与刷新界面分离的方式**。

通过修改TaskUserData[]数组元素的结构体可以显示更多的任务信息，如这里还显示了各个任务的优先级。此外，基于对OSTCBStat不同取值的判断，本例可使Status栏输出更多的状态类型。

3、在本例程中，用户任务MyTask1负责某项作业的计算，MyTask2负责处理计算结果，它们构成了直接的制约关系。类似**例3-1**，由信号量Semp来对二者进行同步。如3.3节所述，我们可把上述任务看成简单的生产者、消费者。**但仅仅这样不足以解决生产者-消费者问题**。

4、用户任务MyTask4通过消息队列MsgQ接收来自MyTask5的消息。发送消息的MyTask5会被起始任务定期挂起和恢复。请注意Status栏的变化。

3.6 生产者-消费者问题

经典的进程同步问题——生产者-消费者问题考虑到了更广泛的情形：生产者将产品放入缓冲池，消费者从缓冲池中取出产品；缓冲池作为临界资源而存在，对其访问需要互斥进行。

app\_queue模块提供了典型的循环队列数据结构，用来具体表述对缓冲池的相关操作：

①建立缓冲池：对应队列初始化APPInitQueue()，它将完成循环队列的实例化；

②将1件产品放入缓冲池中：对应入队APPEnQueue()；

③从缓冲池中取出1件产品：对应出队APPDeQueue()。

**使用说明：**

1. 将app\_queue.c和app\_queue.h拷入App目录，并分别于“源文件”以及“头文件”下按现有项添加。
2. 在全局变量处定义缓冲池，如：

APP\_QUEUE Buffer;

在main函数中初始化缓冲池，如：

APPInitQueue(&Buffer);

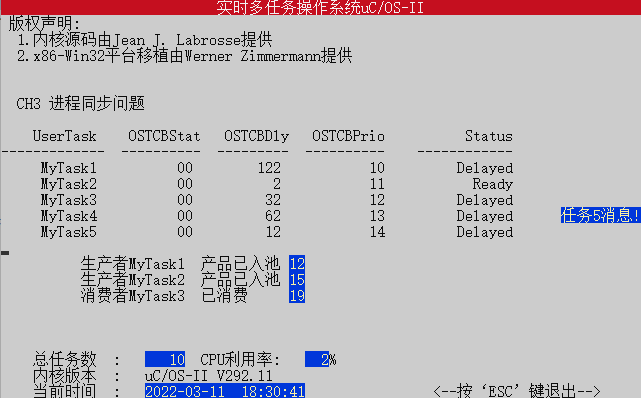
另外，为了可使用模块中的相关函数，需要增加以下声明：

#include "app\_queue.h"

1. 互斥信号量在创建时需要指定优先级天花板，本例可以这样创建：

Mutex = OSMutexCreate(1, &err);

即指定优先级天花板为1，仅低于起始任务的优先级，高于所有用户任务。

下图展示了系统中含两个生产者、一个消费者的情形：

**附：实验报告——实验3**



**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： | 3 | |
| 实验名称： | 进程同步问题 | |
| 实验人员： | 学号 | 21111301083 |
| 姓名 | 房尚鹏 |
| 班级 | 21人工智能 |
| 指导教师： | 李炳火 | |
| 实验室： | 2060502 | |
| 实验日期： | 2023年3月17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．理解μC/OS-II中事件的概念。

2．掌握信号量事件创建、请求和发送的基本方法。

3．使用信号量解决经典进程同步问题。

4．理解普通信号量与互斥信号量的区别。

二、实验内容与要求

1．阅读实验指导书，新建工程app03，运行并观察结果，回答以下问题：

1）信号量、互斥信号量以及消息队列在创建时，成员变量OSEventPtr分别取什么值？

2）定义了初值是1的普通信号量能否代替互斥信号量？为什么？

2．根据3.6节所述，编程解决生产者-消费者问题，调试运行并回答：

1）需要定义几个信号量？几个互斥信号量？它们在创建时初值分别设为多少？

2）按3.6节中的截图完成2个生产者、1个消费者的同步。

3）修改为1个生产者、2个消费者。在不改变MyTask1~ MyTask3延时参数的前提下，观察到什么结果？为什么会这样？

3．在控制台适当位置打印开发者信息（学号、姓名）。

4．仅提交最终生成的exe可执行文件。

三、设计步骤（学生填写）

1．概要设计（文字）

1）为实现展示生产者和消费者界面功能，需要修改MyTask1~3函数的PC\_DispStr函数部分，理由是需要修改输出的字符串。

2）为实现记录生产者和消费者各自操作的次数功能，需要增加源文件test.c的MyTask1~3函数，理由是通过记录函数运行次数来判断操作次数并输出

3）为实现三者之间互不干扰，需要为他们每个函数内部添加互斥信号量，来保证他们之中只有一个函数能够操作队列

4）为了实现消费者的操作数小于两位生产者的操作数总和，设置一个信号量来保证消费者的操作必定是在消费者的操作之后。

2．详细设计（文字+关键代码）

为了添加互斥信号量,在main函数中添加:

INT8U err;

PC\_DispClrScr(DISP\_BGND\_BLACK);

/\* Clear the screen \*/

OSInit(); /\* Initialize uC/OS-II \*/

/\* ToDo: Create semaphores, mailboxes etc. here \*/

SempFull=OSSemCreate(0);

SempEmpty=OSSemCreate(APP\_QUEUE\_SIZE);

Mutex = OSMutexCreate(1, &err);

修改mytask1,2,3函数如下:

void MyTask1(void \*pdata){

INT8U err,count=0;

char s[20];

pdata=pdata;

PC\_DispStr(10,16,"生产者MyTask1 产品已入池",DISP\_FGND\_BLACK+DISP\_BGND\_LIGHT\_GRAY);

while(1){

OSSemPend(SempEmpty,0,&err);

count++;

sprintf(s,"%2d",count);

PC\_DispStr(36,16,s,DISP\_FGND\_YELLOW+DISP\_BGND\_BLUE);

OSMutexPend(Mutex,0,&err);

APPEnQueue(&Buffer,1);

err=OSMutexPost(Mutex);

err=OSSemPost(SempFull);

OSTimeDly(150);

}

}

void MyTask2(void \*pdata){

INT8U err,count=0;

char s[20];

pdata=pdata;

PC\_DispStr(10,17,"生产者MyTask2 产品已入池",DISP\_FGND\_BLACK+DISP\_BGND\_LIGHT\_GRAY);

while(1){

OSSemPend(SempEmpty,0,&err);

count++;

sprintf(s,"%2d",count);

PC\_DispStr(36,17,s,DISP\_FGND\_YELLOW+DISP\_BGND\_BLUE);

OSMutexPend(Mutex,0,&err);

APPEnQueue(&Buffer,2);

err=OSMutexPost(Mutex);

err=OSSemPost(SempFull);

OSTimeDly(120);

}

}

void MyTask3(void \*pdata){

INT8U err,count=0;

char s[20];

pdata=pdata;

PC\_DispStr(10,18,"消费者MyTask3 已消费",DISP\_FGND\_BLACK+DISP\_BGND\_LIGHT\_GRAY);

while(1){

OSSemPend(SempFull,0,&err);

count++;

sprintf(s,"%2d",count);

PC\_DispStr(36,18,s,DISP\_FGND\_YELLOW+DISP\_BGND\_BLUE);

OSMutexPend(Mutex,0,&err);

APPDeQueue(&Buffer);

err=OSMutexPost(Mutex);

err=OSSemPost(SempEmpty);

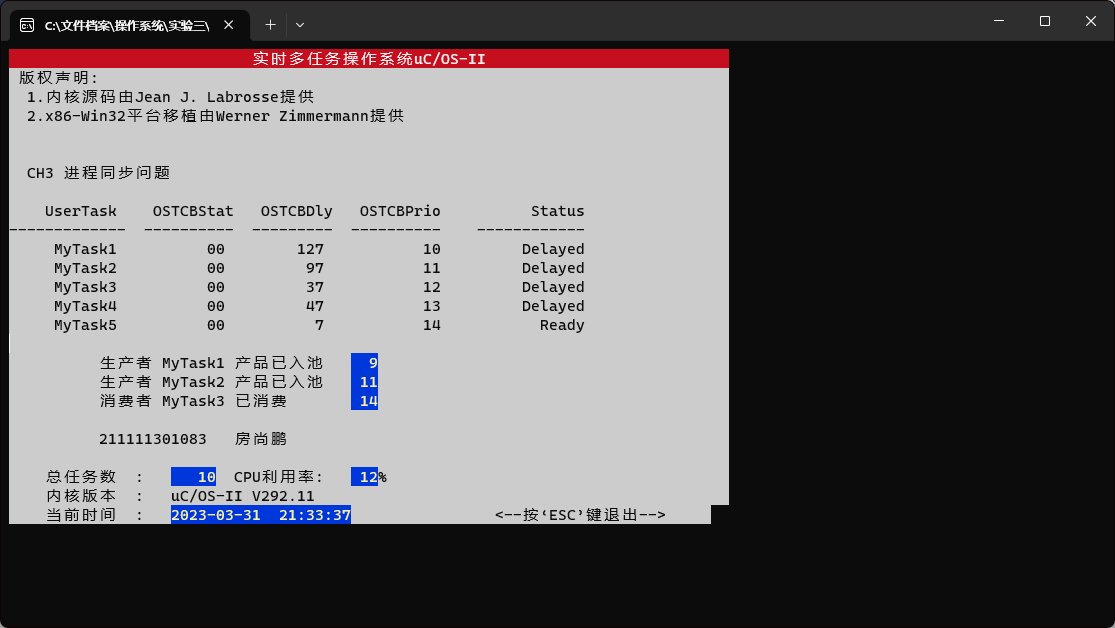
OSTimeDly(90);

}

}

四、结果分析（学生填写）

1．实现基本功能（输出截图）



2．问题分析（文字+部分代码 简要总结）

1）对问题1 - 1），

创建信号量、互斥信号量时OSEventPtr直接赋空值，而创建消息队列时将被赋予指向一个队列

2）对问题1 - 2），

不能，这是因为多任务在互斥地访问系统独占资源时，会出现任务优先级反转（Priority Inversion）的现象。会导致高优先级的任务被低优先级的任务卡住，导致高优先级的任务的执行频率反而不如低优先级的任务。

2）对问题2 - 1），

定义一个信号量，一个互斥信号量，信号量初始值为0，互斥信号量初始值为1。

2）对问题2 - 3），

因为设定了一个信号量来保证生产者来做出v操作，消费者做p操作，那么现在只存在一个生产者，那么所有的p操作全部都被消费者2抢去了，消费者3完全没有机会进行p操作，这导致消费者3的操作数始终为0，并未显示出来。

对于添加互斥信号量时的报错内容，经过代码排查发现是代码不完整导致的逻辑缺失，添加了包括字符以内的一些代码之后能正常运行。