Ch5 死锁

**预备知识**

5.1 死锁产生

系统中有许多不同类型的资源，其中可能引起死锁的主要是需要互斥方式访问的不可抢占资源（临界资源）。这类资源有很多，如打印机、数据文件、缓冲队列等。

典型的死锁是下面这种情形。

进程P1和P2在执行时都需要写两个数据文件F1、F2。使用两个互斥信号量mutex1、mutex2访问这两个文件。如果操作系统调度进程按如下次序——

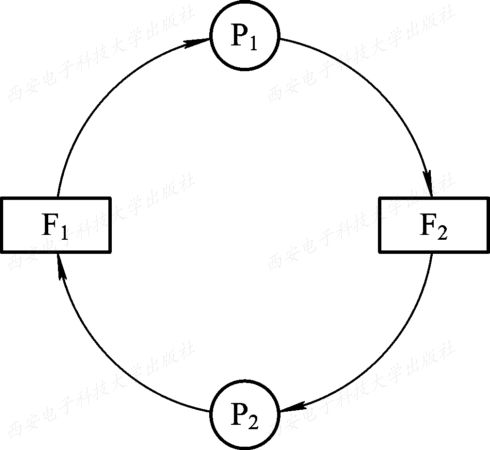
P1: wait(mutex1); // P1请求成功

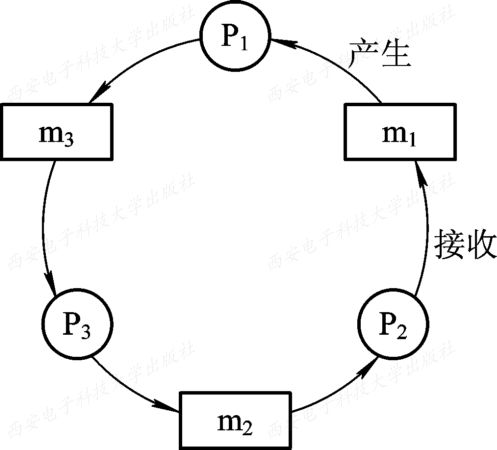
P2: wait(mutex2); // P2请求成功

P1: wait(mutex2); // P1请求失败

P2: wait(mutex1); // P2请求失败

……

则进程P1和P2都将阻塞，处于如下图所示的死锁状态。

有时可消耗资源也会引起死锁。可消耗性资源又称为临时性资源，它在进程运行期间动态地创建和消耗，例如进程通信中的消息。三个进程P1、P2和P3都需要接收来自其它进程的消息后才能继续执行，如果执行次序形成下图中的环路，则处于死锁状态。

在生产者-消费者问题中，缓冲池属于临界资源，对其访问需使用互斥信号量。而分别表示产品和空位的信号量Full、Empty则更接近于可消耗资源的情况。显然，生产者和消费者进程如果编写不当可能导致死锁，即执行次序出现上述图中的环路现象。

思考：如何改写生产者-消费者程序使其产生死锁？

我们将在本章例程中展现这种现象。

5.2 死锁检测

因为无法预知应用程序编写是否规范，操作系统不能完全避免死锁的发生。但当死锁发生后，操作系统应当立即检测到并指明位置和原因。

通常有两种检测死锁的思路。第一种用到了银行家算法中的安全性算法。

安全性算法描述如下：

**Step1**: 向量初始化 Work← Available Finish[*i*]← false

**Step2**: 查找符合如下条件的进程P*i*

(Finish[*i*] = false) ∧ (Need[*i*, *j*] ≤ Work[*j*])

如果没有这种进程存在, 转**Step4**

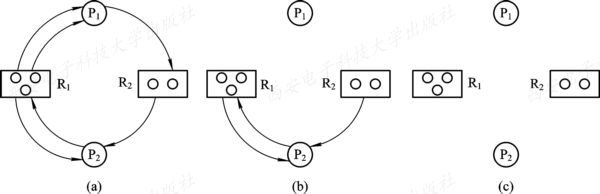
**Step3**: 让 Work[*j*]← Work[*j*] + Allocation[*i*, *j*]

Finish[*i*]← true 转**Step2**

**Step4**: 若所有进程Finish[*i*] = true , 则系统处在安全状态; 否则有死锁

这种检测方法需要操作系统定义并适时更新全局向量和矩阵Available、Allocation、Need以及Finish。适用于进程对资源的访问相对固定的场合。

第二种思路是检测资源分配图中的环路。如果资源分配图经过简化仍无法消除环路，则认为存在死锁。下图展示了经简化后可消除环路的情形。



这种检测方法适用性更广，代价是操作系统需要维护有向图的信息。

本章对生产者-消费者程序的死锁检测基于第一种方法，即利用安全性算法来完成。读者可以自行考虑第二种方法的实现方式。

**实验内容**

5.3 建立工程APP05

例程APP05展现了生产者-消费者发生死锁时的现象。同时实现了上一章提出的时间片轮转算法。

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 | **实现时间片轮转** | **10** | **150** |
| MyTask2 | **实现时间片轮转** | **11** | **120** |
| MyTask3 | **实现时间片轮转** | **12** | **90** |
| MyTask4 | **生产者** | **13** | **70** |
| MyTask5 | **消费者** | **14** | **30** |
| TaskClock | 时钟任务 | 30 | 50 |

5.3.1 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APP05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、app\_queue.c和app\_linked\_list.c，在“头文件”下添加现有项app\_cfg.h、include.h、os\_cfg.h、app\_queue.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。

5.3.2 运行界面

工程APP05运行界面如下：

5.3.3 对工程APP05的说明

1、在APP04基础上，APP05增加了上台间隔的显示。为了实现这一功能，需要调用Win32 API提供的高精度定时器。我们分别为MyTask1~ MyTask3设置了3个定时器，请留意相关代码。

2、为了解决偶尔会显示错误的问题，CPU利用率现在由整数改成浮点数。并基于Win32高精度定时器实现了第二种CPU利用率。请通过阅读代码自行比较两种CPU利用率的计算方式。

3、如上图所示，APP05例程会发生死锁：MyTask4处于请求互斥信号量状态，而MyTask5处于请求普通信号量状态，二者的上台次数都不再递增，很明显在MyTask4与MyTask5之间产生了死锁。

本章实验要求实现一个用户级的死锁检测函数**AppDeadLock**()，它将捕捉到以上死锁的发生以及位置。

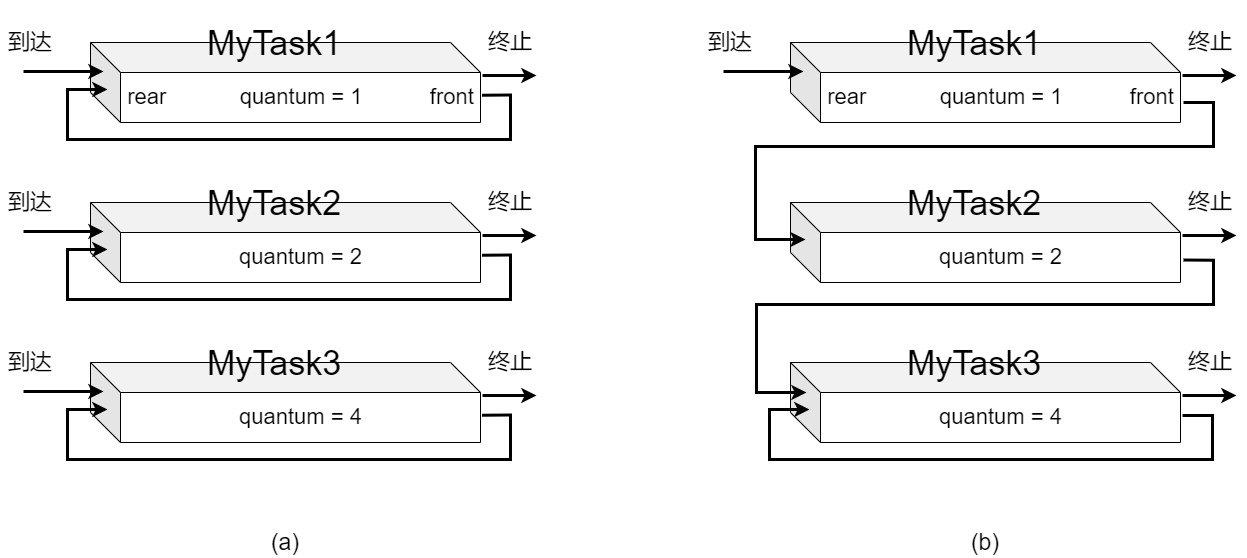
5.4 准备改造uC/OS-II调度器（二）

本节我们将讨论如何在时间片轮转调度基础上进一步扩展成多级反馈队列。另外，我们将继续讨论修改成允许优先级相同后涉及的中断处理函数问题。

5.4.1 多级反馈队列

按上一章4.4.4节，我们已在用户程序中尝试实现了一个时间片轮转算法AppSchedRR()，它运行在MyTask3函数内，以MyTask3上台间隔作为时间片基本单位。为了算法能够正常运行，仿照OS\_TCB构造双向链表的结点APP\_TCB，并于APP\_TCB中增加了时间片相关的成员变量。

APP05例程实现了分别在MyTask1~ MyTask3函数内按时间片轮转一定数量的APP\_TCB块，如下图(a)所示。注意MyTask1~ MyTask3函数内的轮转调度是彼此无关的。如果高优先级队列中的控制块在时间片耗尽后仍需执行，则需插入低一级优先级队列的队尾，如下图(b)所示，那么这个调度模型称为多级反馈队列。



本例程中，需要将MyTask1和MyTask2的调度算法修改为适用于多级队列的时间片轮转算法，而对MyTask3中的块则无需更改调度算法，因为MyTask3位于多级队列的最低一级。

本章实验要求实现一个基于3级队列的时间片轮转算法**AppSchedRR2**()，用在MyTask1和MyTask2函数内以替换AppSchedRR()。

5.4.2 再看活动链OSTCBList

在4.4.2节，我们讨论了全局数组OSTCBTbl[]的作用。有两个链表头指针与它有关——活动链OSTCBList和空闲链OSTCBFreeList。如果允许优先级相同，对空闲链没有实质影响，对活动链的影响较大。

最主要的影响在OSTimeTick()中。来看内核函数OSTimeTick()：

void OSTimeTick (void)

{

OS\_TCB \*ptcb;

……

if (OSRunning == OS\_TRUE) {

ptcb = OSTCBList; //取活动链头指针

while (ptcb->OSTCBPrio != OS\_TASK\_IDLE\_PRIO) { //查找所有TCB块

OS\_ENTER\_CRITICAL();

if (ptcb->OSTCBDly != 0u) {

ptcb->OSTCBDly--; //延时减1

if (ptcb->OSTCBDly == 0u) { //若减1后为0 需分情况处理

if ((ptcb->OSTCBStat & OS\_STAT\_PEND\_ANY) != OS\_STAT\_RDY) {

ptcb->OSTCBStat &= (INT8U)~(INT8U)OS\_STAT\_PEND\_ANY;

ptcb->OSTCBStatPend = OS\_STAT\_PEND\_TO;

} else {

ptcb->OSTCBStatPend = OS\_STAT\_PEND\_OK;

}

if ((ptcb->OSTCBStat & OS\_STAT\_SUSPEND) == OS\_STAT\_RDY) {

OSRdyGrp |= ptcb->OSTCBBitY;

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

}

}

}

ptcb = ptcb->OSTCBNext;

OS\_EXIT\_CRITICAL();

}

}

}

蓝色语句部分说明需要遍历整个活动链，以使所有活动任务块的OSTCBDly减1（仅当OSTCBDly>0）。为什么要减1？这里需要先理解OSTimeTick()是做什么的。

5.4.3 内核函数OSTimeTick()

我们知道操作系统需要一个周期性的节拍（Tick）信号，基于它操作系统才具有了系统时间、延时、超时等概念，更重要的是，节拍周期性地提供给操作系统“介入”程序运行和软硬件资源管理的时机。

一般情况下，节拍中断属于硬件中断——硬件按毫秒级周期产生节拍信号，信号到来会引发一次中断。中断需要遵照标准的处理流程，即先根据中断向量找到相应的中断服务子程序（Interrupt Service Routine, ISR）的入口地址，再转而去执行一趟ISR。

那么节拍中断的ISR是什么样子的？为了具体说明，不妨先来看运行在ARM平台的uC/OS-II如何调用OSTimeTick()。通过分析可知仅有一处调用：

void OS\_CPU\_SysTickHandler (void)

{

OS\_CPU\_SR cpu\_sr; //定义局部变量用来存放CPU现场信息

OS\_ENTER\_CRITICAL(); //关中断

OSIntNesting++; //中断嵌套层数加1

OS\_EXIT\_CRITICAL(); //开中断

OSTimeTick(); /\* Call uC/OS-II's OSTimeTick() \*/

OSIntExit(); /\* Tell uC/OS-II that we are leaving the ISR \*/

}

这里OS\_CPU\_SysTickHandler()即ARM平台的节拍ISR。（为了证明这一点，可以查看ARM启动汇编代码，如startup\_stm32f10x\_cl.s的第78行。）

可以看到，除去保存CPU现场，开关中断等基本步骤外，**真正处理中断的代码位于OSTimeTick()中，称OSTimeTick()为节拍中断处理函数**。

思考：本书中的实验运行在Windows上，节拍中断怎么产生？对应的ISR在哪里？（提示：uC/OS-II运行在Windows平台上，则由Windows提供节拍信号。在os\_cpu\_c.c文件的第416行有

timeSetEvent(1000 / OS\_TICKS\_PER\_SEC, 0, (LPTIMECALLBACK)hInterruptEvent[0], 0, TIME\_PERIODIC | TIME\_CALLBACK\_EVENT\_SET)

此Win32调用虚拟了节拍中断的产生，而非通过硬件。

os\_cpu\_c.c在Zimmermann提供的WIN32移植Port文件夹内，读者可以自行分析其中的函数OSInterruptThread()，它即是节拍ISR，流程与ARM平台的节拍ISR类似。）

5.4.4 如何修改OSTimeTick()

首先来回答5.4.2节提出的问题。OSTimeTick()的主要工作是使所有活动任务块的OSTCBDly减1，并根据减1后是否归零分情况处理。

通过上一节已知，每当一个新的节拍到来，意味着系统时间已经消耗完了一个节拍，故所有处于延时等待状态的任务块（OSTCBDly>0）延时减1；若某任务块仅因延时而阻塞，则当OSTCBDly归零时被唤醒转为就绪态。

接着我们讨论如何修改OSTimeTick()。

因为允许优先级相同的话，活动链OSTCBList无法使用将被删去，那么，基于OSTCBList的遍历将失效。要遍历系统当前所有活动任务块，必须使用其它方式。

第一种方式是通过搜索OSTCBPrioTbl数组实现。因空闲任务无延时，默认情况下搜索范围是OSTCBPrioTbl[0]至OSTCBPrioTbl[62]。这些数组元素可能有三种情况：

1. 空指针——对应优先级下无任务；
2. 指向一个块——对应优先级下只有一个任务，如OSTCBPrioTbl[62]（统计任务）；
3. 指向一个双向链表——对应优先级下有多个任务。

读者可根据这三种情况编写适用的遍历活动任务块算法。可以考虑首先在app\_linked\_list中增加遍历操作。

第二种方式是通过搜索OSTCBTbl数组实现。在上一章4.4.4节，我们提出了在用户程序上的尝试，需要定义3个全局量。其中AppTCBTbl是比照OSTCBTbl定义的全局APP\_TCB块数组。

每从AppTCBTbl[]中分配一个块给新任务（由AppTCBInit()实现），其中一个细节是任务块Id变成了4位的十进制整数（≥1000）。因此，在AppTCBTbl[]中，块Id是否是4位整数成为该任务块是否是活动任务块的标志。读者可以基于此搜索AppTCBTbl[]以实现对所有活动块的遍历。

5.5 本章实验要求

在Test.c中增加了如下2个新函数：

1）AppDeadLock()，用来实现死锁检测。其主体可照搬银行家算法中的安全性算法isSafe()，当检测出死锁时输出信息，当安全时不用输出。在Test.c开始已定义了银行家算法需要使用的矩阵和向量，在main函数中已完成相应的初始化。

**此函数留待读者实现**。你应当在MyTask4和MyTask5中合适位置显式调用AppDeadLock()进行死锁检测（如在每次请求信号量时）。并且在调用AppDeadLock()前后应有预分配和完成分配对应的动作（更新相关矩阵中的数值）。

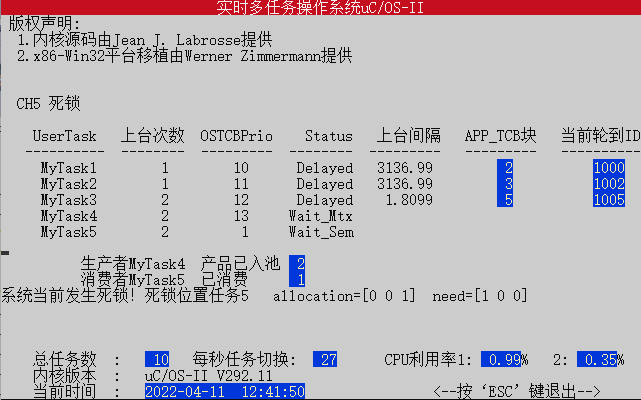
2）AppSchedRR2()，用来实现在时间片轮转基础上的多级反馈队列调度算法。请在MyTask1和MyTask2中调用此函数，仿真出一个三级反馈队列。

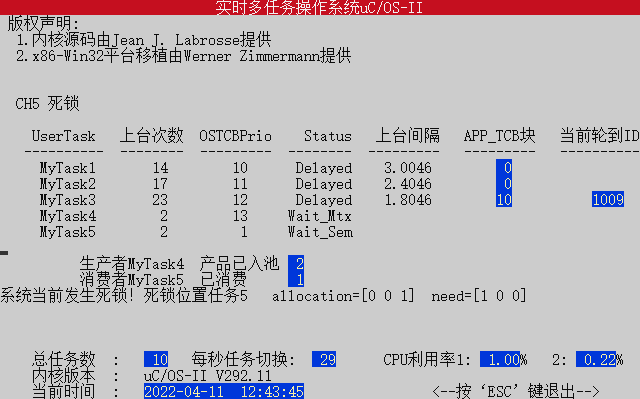
**此函数留待读者实现**。

基本的时间片轮转算法AppSchedRR()已实现，供读者借鉴。

下图反映了系统开始运行时的情形：在MyTask1上有2个任务块1000、1001，MyTask2上有3个任务块1002至1004，MyTask3上有5个任务块1005至1009。在MyTask1至MyTask3上形成三级反馈队列。

注意到运行伊始即检测出系统发生死锁，并已输出死锁的相关信息。

运行一段时间后，MyTask1和MyTask2队列分别向下一级跌落了一个块：

运行到最后，全部跌落至MyTask3队列：

**附：实验报告——实验5**



**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： | 5 | |
| 实验名称： | 死锁 | |
| 实验人员： | 学号 |  |
| 姓名 |  |
| 班级 |  |
| 指导教师： |  | |
| 实验室： |  | |
| 实验日期： | 2022年4月21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．理解系统产生死锁的原因。

2．掌握死锁检测的具体方法。

3．掌握多级反馈队列调度的具体实现方法。

二、实验内容与要求

1．阅读实验指导书。新建工程App05，调试运行。观察现象后回答MyTask4和MyTask5为什么会发生死锁。

2．根据5.5节要求，在指定位置实现函数AppDeadLock()，并正确调用该函数，可检测死锁并输出信息，如5.5节图中所示。

3．根据5.5节要求，在指定位置实现函数AppSchedRR2(INT8U prio)，可正确仿真三级反馈队列调度，如5.5节图中所示。

4．在界面中适当位置输出开发者信息（学号、姓名）。提交修改后的源码文件和项目生成的exe。

三、设计步骤（学生填写）

1．概要设计（文字）

1）打开VS2010，建立工程App05。将App05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/补充：

**……**

四、结果分析（学生填写）

1．问题分析（文字）

对问题1，

2．输出结果（截图）

对问题2

对问题3