西安电子科技大学
XIDIAN UNIVERSITY

嵌入式操作系统

实验报告

任课教师：张亮老师

学号姓名：

实验 1 任务的基本管理

1 实验目的

- ✓ 理解任务管理的基本原理，了解任务的各个基本状态及其变迁过程；
- ✓ 掌握 μC/OS-II 中任务管理的基本方法（创建、启动、挂起、解挂任务）；
- ✓ 熟练使用 μC/OS-II 任务管理的基本系统调用。

2 实验运行流程及代码分析

2.1 实验运行流程

初始创建两个应用任务，先运行就绪任务中优先级最高的人任务 Task0，但是 Task0 不断挂起自己，再运行剩余就绪任务最高优先级任务 Task1，然后 Task0 被 Task1 解挂，Task0 抢占 CPU，两个任务交替运行，过程中记录任务交替运行的次数

一个任务通常是一个无限的循环 ，由于任务的执行是由操作系统内核调度的，因此任务是绝不会返回的，其返回参数必须定义成 void。在 μC/OS-II中，当一个运行着的任务使一个比它优先级高的任务进入了就绪态，当前任务的 CPU 使用权就会被抢占，高优先级任务会立刻得到 CPU 的控制权（在系统允许调度和任务切换的前提下）。μC/OS-II可以管理多达 64 个任务，但目前版本的 μC/OS-II有两个任务已经被系统占用了（即空闲任务和统计任务）。必须给每个任务赋以不同的优先级，任务的优先级号就是任务编号（ID），优先级可以从 0 到 OS_LOWEST_PR10-2。优先级号越低，任务的优先级越高。μC/OS-II总是运行进入就绪态的优先级最高的任务。

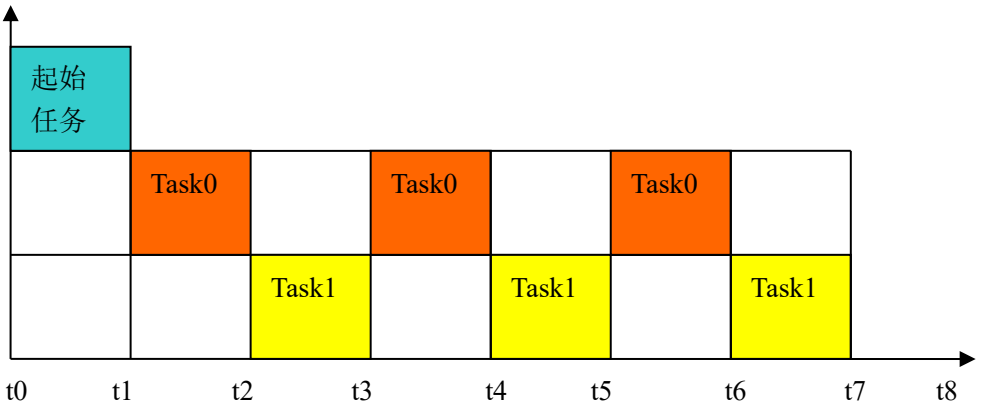


图 1

注意：图中的栅格并不代表严格的时间刻度，而仅仅表现各任务启动和执行的相对先后关系。

2.2 代码分析

系统启动后，经历一系列的初始化过程，进入 main()函数，这是我们编写实现应用程序的起点。首先需要在 main()函数里创建起始任务 TaskStart：

```
OSTaskCreate(TaskStart, (void *)0, &TaskStartStk[TASK_STK_SIZE - 1], 4);
```

TaskStart 任务

TaskStart 任务负责安装操作系统的时钟中断服务例程、初始化操作系统时钟，并创建所有的应用任务：

```
ucos_x86_idt_set_handler(0x20,(void *)OSTickISR,0x8e00);
                                        /* Install uC/OS-II's clock tick ISR       */
ucos_timer_init();                      /*Timer  初始化*/
TaskStartCreateTasks();                 /* Create all the application tasks        */
OSTaskSuspend(OS_PRIO_SELF);
```

具体负责应用任务创建的 TaskStartCreateTasks 函数代码如下，它创建了两个应用任务 Task0 和 Task1：

```
static    void    TaskStartCreateTasks (void)
{
    INT8U    i;
    for (i = 0; i < N_TASKS; i++) {          /* Create N_TASKS identical tasks       */
      TaskData[i] =    i;                   /* Each task will display its own letter    */
      }
      OSTaskCreate(Task0, (void *)&TaskData[i], &TaskStk[i][TASK_STK_SIZE - 1], 5);
      OSTaskCreate(Task1, (void *)&TaskData[i], &TaskStk[1][TASK_STK_SIZE - 1], 6);
   }
```

TaskStart 任务完成上述操作后将自己挂起，操作系统将调度当前优先级最高的应用任务 Task0 运行。

应用任务

应用任务 Task0 运行后将自己挂起，之后操作系统就会调度处于就绪状态的优先级最高的任务，具体代码如下：

```
void    Task0 (void *pdata)
{
    INT8U i;
    INT8U err;
    i=*(int *)pdata;
     for (;;) {
```

```
            ……                    /*此处为输出信息,显示任务运行的状态      */
            err=OSTaskSuspend(5);     /* suspend   itself              */
          }
}
```

应用任务 Task1 运行后将 Task0 唤醒，使其进入到就绪队列中：

```
void   Task1 (void *pdata)
{
    INT8U i;
    INT8U err;
    i=*(int *)pdata;

      for (;;) {
        OSTimeDly(150);
          ……                          /*此处为输出信息,显示任务运行的状态      */
        OSTimeDly(150);
        err=OSTaskResume(5);        /* resume    task0         */
            }
}
```

OSTaskCreate（）

建立一个新任务。任务的建立可以在多任务环境启动之前，也可以在正在运行的任务中建立。中断处理程序中不能建立任务。一个任务可以为无限循环的结构。

函数原型：INT8U OSTaskCreate(void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio);

参数说明：task 是指向任务代码首地址的指针。

Pdata 指向一个数据结构，该结构用来在建立任务时向任务传递参数。

返回值：

OSTaskCreate（）的返回值为下述之一：

- OS_NO_ERR：函数调用成功。
- OS_PRIO_EXIST：具有该优先级的任务已经存在。
- OS_PRIO_INVALID：参数指定的优先级大于 OS_LOWEST_PRIO。
- OS_NO_MORE_TCB：系统中没有 OS_TCB 可以分配给任务了。

OSTaskSuspend（）

无条件挂起一个任务。调用此函数的任务也可以传递参数 OS_PRIO_SELF，挂起调用任务本身。当前任务挂起后，只有其他任务才能唤醒被挂起的任务。任务挂起后，系统会重新进行任务调度，运行下一个优先级最高的就绪任务。唤醒挂起任务需要调用函数 OSTaskResume（）。

任务的挂起是可以叠加到其他操作上的。例如，任务被挂起时正在进行延时操作，那么任务的

唤醒就需要两个条件：延时的结束以及其他任务的唤醒操作。又如，任务被挂起时正在等待信号量，当任务从信号量的等待对列中清除后也不能立即运行，而必须等到被唤醒后。

函数原型：INT8U OSTaskSuspend **(INT8U prio);**

参数说明：prio 为指定要获取挂起的任务优先级，也可以指定参数 OS_PRIO_SELF，挂起任务本身。此时，下一个优先级最高的就绪任务将运行。

返回值：

OSTaskSuspend（）的返回值为下述之一：

- OS_NO_ERR：函数调用成功。
- OS_TASK_ SUSPEND_IDLE：试图挂起 μC/OS-II 中的空闲任务（Idle task）。此为非法操作。
- OS_PRIO_INVALID：参数指定的优先级大于 OS_LOWEST_PRIO 或没有设定 OS_PRIO_SELF 的值。
- OS_TASK_ SUSPEND _PRIO：要挂起的任务不存在。

OSTaskResume（）

唤醒一个用 OSTaskSuspend（）函数挂起的任务。OSTaskResume（）也是唯一能“解挂”挂起任务的函数。

函数原型：NT8U **OSTaskResume (INT8U prio);**

参数说明：prio 指定要唤醒任务的优先级。

返回值：

OSTaskResume （）的返回值为下述之一：

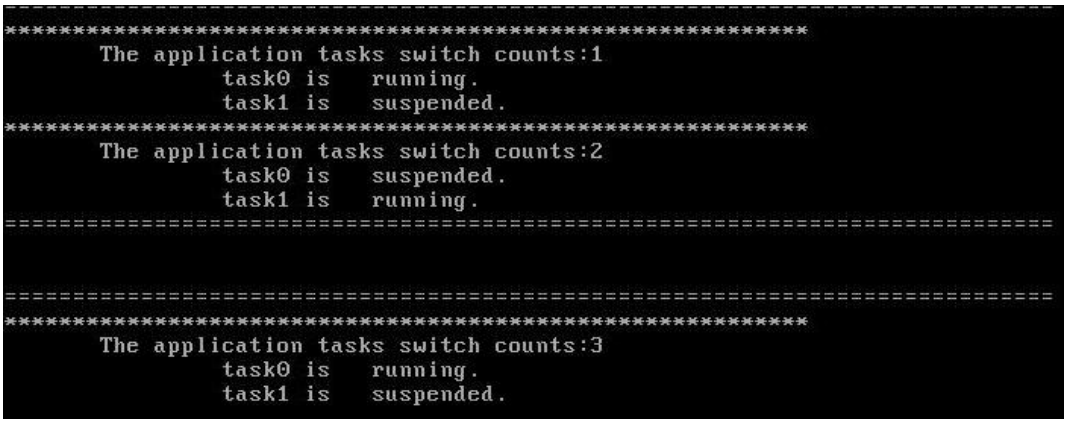
- OS_NO_ERR：函数调用成功。
- OS_TASK_RESUME_PRIO：要唤醒的任务不存在。
- OS_TASK_NOT_SUSPENDED：要唤醒的任务不在挂起状态。
- OS_PRIO_INVALID：参数指定的优先级大于或等于 OS_LOWEST_PRIO。

3 实验实现与结果分析

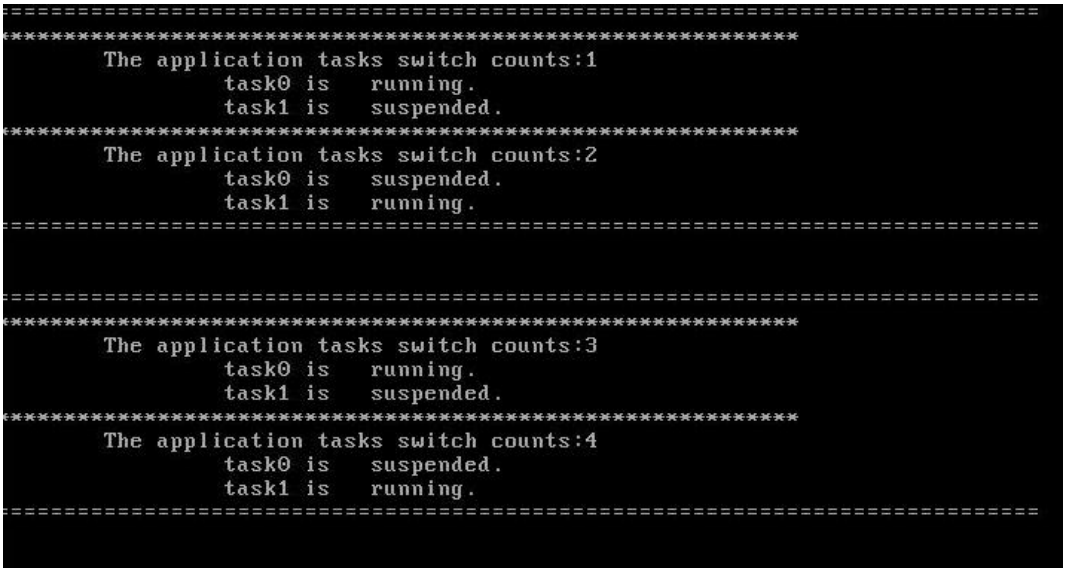
当我们在 LambdaTOOL 调试器的控制下运行构建好的程序后，将看到在 μC/OS-II内核的调度管理下，两个应用任务不断切换执行的情形：

T_1 时刻的截图

T_2 时刻的截图



T_3 时刻的截图



T_4 时刻的截图

在 μC/OS-II中，当一个运行着的任务使一个比它优先级高的任务进入了就绪态，当前任务的 CPU 使用权就会被抢占，高优先级任务会立刻得到 CPU 的控制权。初始创建两个应用任务，先运行就绪任务中优先级最高的人任务 Task0，但是 Task0 不断挂起自己，再运行剩余就绪任务最高优先级任务 Task1，然后 Task0 被 Task1 解挂，Task0 抢占 CPU，两个任务交替运行，过程中记录任务交替

运行的次数

实验 2 优先级反转

1 实验目的

掌握在基于优先级的可抢占嵌入式实时操作系统的应用中，出现优先级反转现象的原理。

2 实验运行流程及代码分析

2.1 实验运行流程

任务 TA0-2 的优先级依次降低，TA0 的优先级最高
TA0 和 2 互斥使用同一个资源，通过二值信号量 mutex 来管理（不使用防止优先级反转的策略）

初始化创建三个任务，延时 TA0 与 1 以便于先运行 TA2，那么 Ta2 运行并先申请到 mutex，
接下来 Ta1 的延时到期，优先级高于 Ta2，所以就抢占 CPU 而运行，而 Ta2 由执行态转换为就绪态，
接下来 TA0 的延时到期，Ta0 的优先级高于 TA1，所以 TA0 抢占 CPU，TA1 从运行态转为就绪态，但是此时 Ta0 申请 mutex 就会被阻塞（已经先被 TA2 申请到了）
时间到了后，TA1 由就绪态转为运行态，此时 TA0 在因为等待 TA2 手上的 mutex 而被阻塞，Ta2 因为优先级比 TA1 低而阻塞，这个时候如果 TA1 一直执行而 TA2 没有机会被调度，Ta2 就必须等待 TA1 执行完才能执行，TA0 更是要等到 TA2 释放 mutex 后才能执行，所以就出现了低优先级的 TA1 阻塞了高优先级的 TA0 的优先级反转现象

1）设计了 3 个应用任务 TA0～TA2，其优先级逐渐降低，任务 TA0 的优先级最高。

2）除任务 TA1 外，其它应用任务都要使用同一种资源，该资源必须被互斥使用。为此，创建一个二值信号量 mutex 来模拟该资源。虽然 μC/OS-II在创建信号量时可以选择采用防止优先级反转的策略，但在本实验中我们不使用这种策略。

3）应用任务的执行情况如图 2-1 所示：

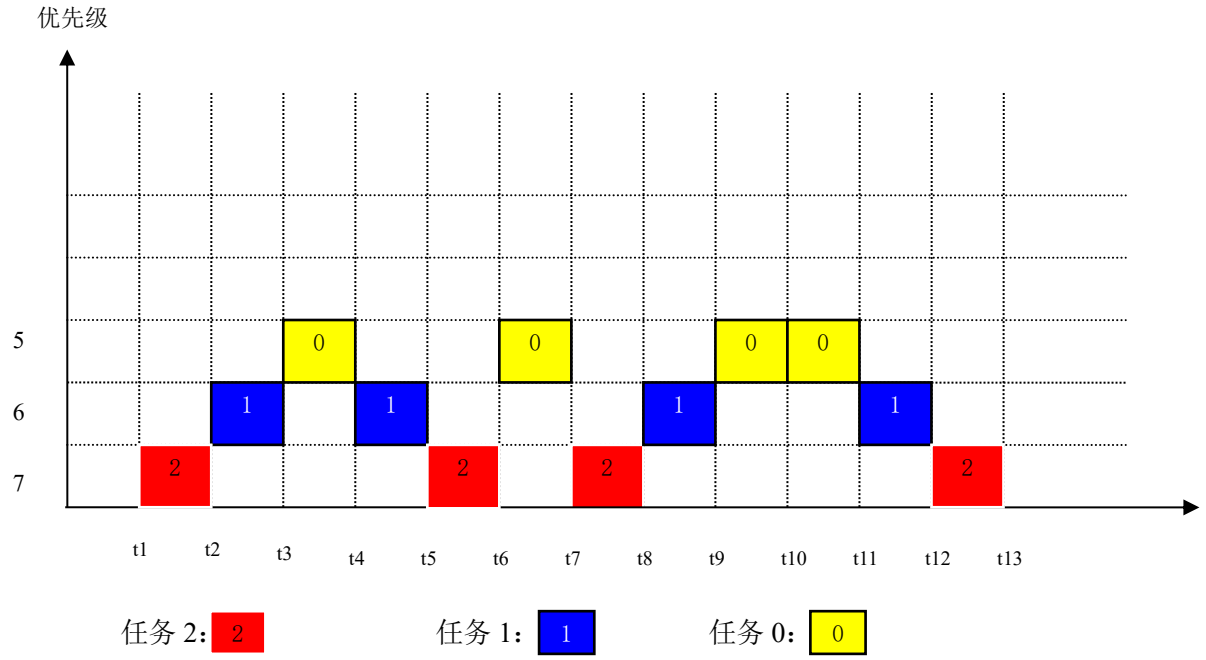


图 2-1

注意：图中的栅格并不代表严格的时间刻度，而仅仅表现各个任务启动和执行的相对先后关系。

2.2 代码分析

首先，在 main()函数中创建一个二值信号量：

mutex=OSSemCreate(1);

然后，在 main()函数中创建 TaskStart 任务。

TaskStart 任务

在 TaskStart 任务中创建并启动所有的应用任务 TA0, TA1,TA2。

```
static   void   TaskStartCreateTasks (void)
    {
        INT8U   i;
        for (i = 0; i <N_TASKS; i++) {          /* Create N_TASKS identical tasks   */
        TaskData[i] = i;
        }                                       /* Each task will pass its own id       */
        OSTaskCreate(Task0, (void *)&TaskData[0], &TaskStk[0][TASK_STK_SIZE - 1], 5);
        OSTaskCreate(Task1, (void *)&TaskData[1], &TaskStk[1][TASK_STK_SIZE - 1], 6);
        OSTaskCreate(Task2, (void *)&TaskData[2], &TaskStk[2][TASK_STK_SIZE - 1], 7);
    }
```

任务 TA0 的优先级最高，它需要使用信号量 mutex：

```
void    Task0 (void *pdata)
{
      INT8U    err;
      INT8U    id;
      INT16U value;
      id=*(int *)pdata;
      for (;;)
      {
         printk("Task %d is waitting a event.\n",id);
         OSTimeDly(200);                                    /* Delay 200 clock tick          */
         printk("The event of Task %d come.\n",id);
         printk("Task %d is try to get mutex.\n",id);
          OSSemPend(mutex,0,&err);                          /* Acquire mutex */
           switch(err)
           {
                case OS_NO_ERR:
                     printk("Task %d has got the mutex.\n",id);
                     printk("\n");
                     break;
                default:
                     printk("Task %d is suspended.\n",id);
                     printk("\n");
           }

           OSTimeDly(200);                                  /* Delay 200 clock tick          */
           printk("Task %d release mutex.\n",id);

           OSSemPost(mutex);                                /* Release mutex              */

       }
}
```

任务 TA1 具有中等优先级，它不使用信号量：

```
 void    Task1 (void *pdata)
 {
      INT8U    err;
      INT8U    id;
      int i;
      id=*(int *)pdata;

       for (;;) {
          printk("Task %d is waitting a event.\n",id);
          OSTimeDly(100);                                   /* Delay 100 clock tick          */
          printk("The event of Task %d come.\n",id);
          OSTimeDly(100);
```

```
    }
}
```

任务 TA2 的优先级最低，和高优先级任务 TA0 共用信号量 mutex：

```
    void    Task2 (void *pdata)
    {
        INT8U    err;
        INT8U    id;
        INT16U value;
        id=*(int *)pdata;
        int i;
        for (;;)
        {
            printk("Task %d is trying to get mutex.\n",id);
            OSSemPend(mutex,0,&err);                    /* Acquire mutex */
            switch(err)
            {
                case OS_NO_ERR:
                {
                    printk("\n");
                    printk("---------------------------------\n");
                    printk("Task %d has got the mutex.\n",id);
                    OSTimeDly(200);                     /* Delay 100 clock tick          */
                    break;
                }
                default :
                {
                    printk("Task %d    is failed to get mutex.\n",id);
                    printk("\n");
                    OSTimeDly(200);                     /* Delay 100 clock tick          */
                    break;
                }
            }
        printk("Task %d release mutex.\n",id);
         printk("\n");
```

```
        OSSemPost(mutex);                                   /* Release mutex              */
        }
    }
```

OSSemCreate（）

该函数建立并初始化一个信号量，信号量的作用如下：

- 允许一个任务和其他任务或者中断同步
- 取得设备的使用权
- 标志事件的发生

函数原型：OSSemCreate(INT16U value);

参数说明：value 参数是所建立的信号量的初始值，可以取 0 到 65535 之间的任何值。

返回值：OSSemCreate（）函数返回指向分配给所建立的信号量的控制块的指针。如果没有可用的控制块，OSSemCreate（）函数返回空指针。

OSSemPend（）

该函数用于任务试图取得设备的使用权、任务需要和其他任务或中断同步、任务需要等待特定事件的发生的场合。如果任务调用 OSSemPend（）函数时，信号量的值大于零，OSSemPend（）函数递减该值并返回该值。如果调用时信号量值等于零，OSSemPend（）函数将任务加入该信号量的等待队列。OSSemPend（）函数挂起当前任务直到其他的任务或中断设置信号量或超出等待的预期时间。如果在预期的时钟节拍内信号量被设置，μC/OS-II默认让最高优先级的任务取得信号量并回到就绪状态。一个被 OSTaskSuspend（）函数挂起的任务也可以接受信号量，但这个任务将一直保持挂起状态直到通过调用 OSTaskResume（）函数恢复该任务的运行。

函数原型：Void OSSemPend (OS_EVNNT　*pevent, INT16U　timeout,　int8u *err);

参数说明：

pevent　是指向信号量的指针。该指针的值在建立该信号量时可以得到。（参考 OSSemCreate（）函数）。

Timeout　允许一个任务在经过了指定数目的时钟节拍后还没有得到需要的信号量时恢复就绪状态。如果该值为零表示任务将持续地等待信号量，最大的等待时间为 65535 个时钟节拍。这个时间长度并不是非常严格的，可能存在一个时钟节拍的误差。

Err 是指向包含错误码的变量的指针。

返回值：

OSSemPend（）函数返回的错误码可能为下述几种：

- OS_NO_ERR ：信号量不为零。
- OS_TIMEOUT ：信号量没有在指定数目的时钟周期内被设置。
- OS_ERR_PEND_ISR ：从中断调用该函数。虽然规定了不允许从中断调用该函数，但 μC/OS-II 仍然包含了检测这种情况的功能。
- OS_ERR_EVENT_TYPE　：pevent 不是指向信号量的指针。

OSemPost（）

该函数用于设置指定的信号量。如果指定的信号量是零或大于零，OSSemPost（）函数递增该信号量的值并返回。如果有任何任务在等待该信号量，则最高优先级的任务将得到信号量并进入就绪状态。任务调度函数将进行任务调度，决定当前运行的任务是否仍然为最高优先级的就绪任务。

函数原型：INT8U OSSemPost（OS_EVENT *pevent）；

参数说明：pevent 是指向信号量的指针。该指针的值在建立该信号量时可以得到。（参考OSSemCreate（）函数）。

返回值：

OSSemPost（）函数的返回值为下述之一：

- OS_NO_ERR ：信号量被成功地设置
- OS_SEM_OVF ：信号量的值溢出
- OS_ERR_EVENT_TYPE ：pevent 不是指向信号量的指针

OSTimeDly（）

该函数用于将一个任务延时若干个时钟节拍。如果延时时间大于 0，系统将立即进行任务调度。延时时间的长度可从 0 到 65535 个时钟节拍。延时时间 0 表示不进行延时，函数将立即返回调用者。延时的具体时间依赖于系统每秒钟有多少个时钟节拍（由文件 SO_CFG.H 中的 OS_TICKS_PER_SEC 宏来设定）。

函数原型：void OSTimeDly (INT16U ticks);

参数说明：ticks 为要延时的时钟节拍数。

返回值：无

3 实验实现与结果分析

```
Welcome to ucos-II
Task 0 is waitting a event.
Task 1 is waitting a event.
Task 2 is trying to get mutex.

--------------------------------
Task 2 has got the mutex.
The event of Task 1 come.
The event of Task 0 come.
Task 0 is try to get mutex.
Task 1 is waitting a event.
Task 2 release mutex.

Task 0 has got the mutex.

Task 2 is trying to get mutex.
The event of Task 1 come.
Task 0 release mutex.
Task 0 is waitting a event.
Task 1 is waitting a event.

--------------------------------
Task 2 has got the mutex.
```

优先级反转发生在有多个任务需要使用共享资源的情况下，可能会出现高优先级任务被低优先级任务阻塞，并等待低优先级任务执行的现象。高优先级任务需要等待低优先级任务释放资源，而低优先级任务又正在等待中等优先级任务，这种现象就被称为优先级反转。两个任务都试图访问共享资源是出现优先级反转最通常的情况。为了保证一致性，这种访问应该是顺序进行的。如果高优先级任务首先访问共享资源，则会保持共享资源访问的合适的任务优先级顺序；但如果是低优先级任务首先获得共享资源的访问，然后高优先级任务请求对共享资源的访问，则高优先级任务被阻塞，直到低优先级任务完成对共享资源的访问。

任务 TA0-2 的优先级依次降低，TA0 的优先级最高

TA0 和 2 互斥使用同一个资源，通过二值信号量 mutex 来管理（不使用防止优先级反转的策略）

初始化创建三个任务，延时 TA0 与 1 以便于先运行 TA2，那么 Ta2 运行并先申请到 mutex，

接下来 Ta1 的延时到期，优先级高于 Ta2，所以就抢占 CPU 而运行，而 Ta2 由执行态转换为就绪态，

接下来 TA0 的延时到期，Ta0 的优先级高于 TA1，所以 TA0 抢占 CPU，TA1 从运行态转为就绪态，但是此时 Ta0 申请 mutex 就会被阻塞（已经先被 TA2 申请到了）

时间到了后，TA1 由就绪态转为运行态，此时 TA0 在因为等待 TA2 手上的 mutex 而被阻塞，Ta2 因为优先级比 TA1 低而阻塞，这个时候如果 TA1 一直执行而 TA2 没有机会被调度，Ta2 就必须等待 TA1 执行完才能执行，TA0 更是要等到 TA2 释放 mutex 后才能执行，所以就出现了低优先级的 TA1 阻塞了高优先级的 TA0 的优先级反转现象

实验 3　优先级继承

1 实验目的

掌握嵌入式实时操作系统 μC/OS-II 解决优先级反转的策略——优先级继承的原理。

2 实验运行流程及代码分析

2.1　实验运行流程

优先级继承是指高优先级任务因为申请某共享资源失败而被阻塞的时候，把当前拥有该共享资源的任务的优先级提升到等于这个高优先级任务的优先级，操作系统可指定 PIP

三个任务 Task0-2 竞争同一个资源 mutex 而相互制约，0-2 的优先级依次升高，而使用 mutex 时采用优先级继承机制，设置在使用 mutex 时的 PIP 为 8

Task2 最高优先级 10，首先运行，先获取到 mutex，下一时刻因为被执行了延时函数而挂起，此时调度当前最高优先级的 Task1

但是 Task1 运行时由于申请不到 mutex 而被阻塞，此时调度优先级最低的 Task0，同样因为申请不到 mutex 而被阻塞，但是当 01 被阻塞的时候，Task2 的优先级最高，所以保持优先级不变（当所有任务被阻塞的时候，系统有一个空闲任务在运行，优先级最低）

下一时刻 Task2 的延时到期，开始执行，释放 mutex，Task1 就申请到了 mutex，Task2 又被延时一段时间，Task1 也被延时，空闲任务开始运行

下一时刻 Task2 延时到期，开始运行并申请 mutex，申请失败而被阻塞，此时操作系统发现拥有 mutex 的 Task1 优先级低于 task2，因此提升 Task1 的优先级到 PIP，但是此时 task1 正在被延时而没有运行

等到 Task1 延时时间到，它在 PIP 下运行，释放信号量，而 task2 就申请到了 mutex，Task1 就恢复原来的优先级高度，Task2 优先级高于 1 而抢占 CPU，12 同时延时

2 的延时先到期，释放 mutex，而被 0 获取，2 再延时，0 被调度，在 0 打印信息后把自己延时，而后 1 的延时到期，申请 mutex 而被阻塞，又开始优先级继承机制而提升 task0 的优先级到 PIP，空闲任务运行，

2 的延时到期，申请信号量被阻塞，但是 task0 的优先级现在是高于 2 的，所以不会触发优先级继承机制

在本实验中设计了处于不同优先级的应用任务，如下图所示：

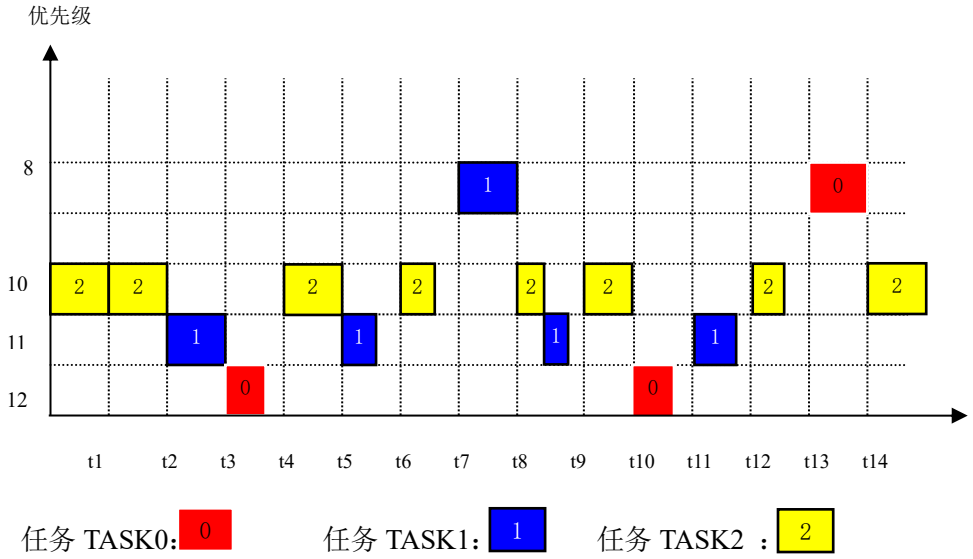


图 3-1

注意：图中的栅格并不代表严格的时间刻度，而仅仅表现各个任务启动和执行的相对先后关系。

这 3 个应用任务因为要竞争同一互斥资源 mutex 而相互制约。其中，任务 TASK0 的原始优先级最低，任务 TASK1 的原始优先级中等，任务 TASK2 的原始优先级最高。在使用 mutex 时采用优先级继承策略，并指定各任务在使用 mutex 时的 PIP（优先级继承优先级）为 8。

2.2 代码分析

TaskStart 任务首先运行，由它创建其他 3 个应用任务：

```
void    TaskStartCreateTasks (void)
{
    INT8U    i;
    for (i = 0; i <N_TASKS; i++) {                /* Create N_TASKS identical tasks      */
    TaskData[i] = i;                              /* Each task will pass its own id          */
    OSTaskCreate(Task, (void *)&TaskData[i], &TaskStk[i][TASK_STK_SIZE - 1], 12-i);
    }
}
```

每个任务的代码均如下所示：

```
void    Task (void *pdata)
{
      INT8U    err;
      INT8U    id;
     id=*(int *)pdata;
```

```
    for (;;)
    {
        printk("task %d try to get the mutex. \n", id);
        OSMutexPend(mutex, 0, &err);                /* Acquire mutex to get continue */
        printk("task %d is getting the mutex. \n", id);
        OSTimeDlyHMSM(0, 0, 0, 200);                /* Wait 200 minisecond           */
         printk("task %d     releases the mutex. \n", id);
        OSMutexPost(mutex);                         /* Release mutex                 */
        OSTimeDlyHMSM(0, 0, 0, (3-id)*150);         /* Wait (3-id)*150 minisecond    */
    }
}
```

OSMutexCreate()

本实验通过调用 OSMutexCreate(8,&err)，设置了一个互斥信号量，其中 8 为 PIP（优先级继承优先级）的值。

OSMutexPend()

该函数用于获得互斥信号量，其具体执行过程如下（关键代码解释）：

```
If ((INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8) == OS_MUTEX_AVAILABLE){}
```

判断互斥信号量是否被占用

```
pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;      /*Yes, Acquire the resource    */
pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;        /* Save priority of owning task */
pevent->OSEventPtr  = (void *)OSTCBCur;           /*     Point to owning task's OS_TCB */
```

如果互斥信号量没有被占有，则获得互斥信号量，任务继续运行。

```
pip   = (INT8U)(pevent->OSEventCnt >> 8);   /* No, Get PIP from mutex                */
mprio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);   /*        Get priority of
mutex owner     */
ptcb  = (OS_TCB *)(pevent->OSEventPtr);           /*        Point to TCB of mutex owner     */
if (ptcb->OSTCBPrio != pip && mprio > OSTCBCur->OSTCBPrio) {   /*        Need to promote prio
of owner?*/
```

互斥信号量被占用，但是占有互斥信号量的任务的优先级高于请求互斥信号量的任务的优先级，则不改变占有互斥信号量的任务的优先级。

```
ptcb->OSTCBPrio             = pip;      /* Change owner task prio to PIP     */
ptcb->OSTCBY                = ptcb->OSTCBPrio >> 3;
ptcb->OSTCBBitY             = OSMapTbl[ptcb->OSTCBY];
ptcb->OSTCBX                = ptcb->OSTCBPrio & 0x07;
```

```
ptcb->OSTCBBitX              = OSMapTbl[ptcb->OSTCBX];
if (rdy == TRUE) {                                          /* If task was ready at owner's priority ...*/
        OSRdyGrp    |= ptcb->OSTCBBitY;         /* ... make it ready at new priority.          */
        OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
        }
        OSTCBPrioTbl[pip]    = (OS_TCB *)ptcb;
    }
```

占有互斥信号量的任务的优先级低于请求互斥信号量的任务的优先级，改变占有互斥信号量的任务的优先级到 PIP。

OSMutexPost()

该函数用于释放互斥信号量，具体代码说明如下：

```
if (OSTCBCur->OSTCBPrio == pip) {           /* Did we have to raise current task's priority? */
```

判断任务的优先级是否在申请获得互斥信号量的过程中被改变。

```
if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
    }
    OSTCBCur->OSTCBPrio              = prio;
    OSTCBCur->OSTCBY                 = prio >> 3;
    OSTCBCur->OSTCBBitY              = OSMapTbl[OSTCBCur->OSTCBY];
    OSTCBCur->OSTCBX                 = prio & 0x07;
    OSTCBCur->OSTCBBitX              = OSMapTbl[OSTCBCur->OSTCBX];
    OSRdyGrp                        |= OSTCBCur->OSTCBBitY;
    OSRdyTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;
    OSTCBPrioTbl[prio]               = (OS_TCB *)OSTCBCur;
}
OSTCBPrioTbl[pip] = (OS_TCB *)1;                    /* Reserve table entry */
```

如果任务的优先级在申请获得互斥信量的过程中被改变，则还原其初始优先级。

3 实验实现与结果分析

本实验运行后，从虚拟机的窗口中可以看到如下信息：

```
task 2 try to get the mutex.
task 2 is getting the mutex.
task 1 try to get the mutex.
task 0 try to get the mutex.
task 2   releases the mutex.
task 1 is getting the mutex.
task 2 try to get the mutex.
task 1   releases the mutex.
task 2 is getting the mutex.
task 2   releases the mutex.
task 0 is getting the mutex.
task 1 try to get the mutex.
task 2 try to get the mutex.
task 0   releases the mutex.
task 2 is getting the mutex.
task 2   releases the mutex.
```

优先级继承的主要思想是：当高优先级任务因申请某共享资源失败被阻塞时，把当前拥有该资源的、且优先级较低的任务的优先级提升，提升的高度等于这个高优先级任务的优先级。在 μC/OS-II 中，在创建管理共享资源的互斥信号量时，可以指定一个 PIP（优先级继承优先级），之后可以把拥有共享资源的任务优先级提升到这个高度。具体过程如下：

1. 当任务 A 申请共享资源 S 时，首先判断是否有别的任务正在占用资源 S，若无，则任务 A 获得资源 S 并继续执行；
2. 如果任务 A 申请共享资源 S 时任务 B 正在使用该资源，则任务 A 被挂起，等待任务 B 释放该资源；同时判断任务 B 的优先级是否低于任务 A 的，若高于任务 A，则维持任务 B 的优先级不变；
3. 如果任务 B 的优先级低于任务 A 的，则提升任务 B 的优先级到 PIP，当任务 B 释放资源后，再恢复其原来的优先级。

优先级继承是指高优先级任务因为申请某共享资源失败而被阻塞的时候，把当前拥有该共享资源的任务的优先级提升到等于这个高优先级任务的优先级，操作系统可指定 PIP

三个任务 Task0-2 竞争同一个资源 mutex 而相互制约，0-2 的优先级依次升高，而使用 mutex 时采用优先级继承机制，设置在使用 mutex 时的 PIP 为 8

Task2 最高优先级 10，首先运行，先获取到 mutex，下一时刻因为被执行了延时函数而挂起，此时调度当前最高优先级的 Task1

但是 Task1 运行时由于申请不到 mutex 而被阻塞，此时调度优先级最低的 Task0，同样因为申请不到 mutex 而被阻塞，但是当 01 被阻塞的时候，Task2 的优先级最高，所以保持优先级不变（当所有任务被阻塞的时候，系统有一个空闲任务在运行，优先级最低）

下一时刻 Task2 的延时到期，开始执行，释放 mutex，Task1 就申请到了 mutex，Task2 又被延时一段时间，Task1 也被延时，空闲任务开始运行

下一时刻 Task2 延时到期，开始运行并申请 mutex，申请失败而被阻塞，此时操作系统发现拥有 mutex 的 Task1 优先级低于 task2，因此提升 Task1 的优先级到 PIP，但是此时 task1 正在被延时而没有运行

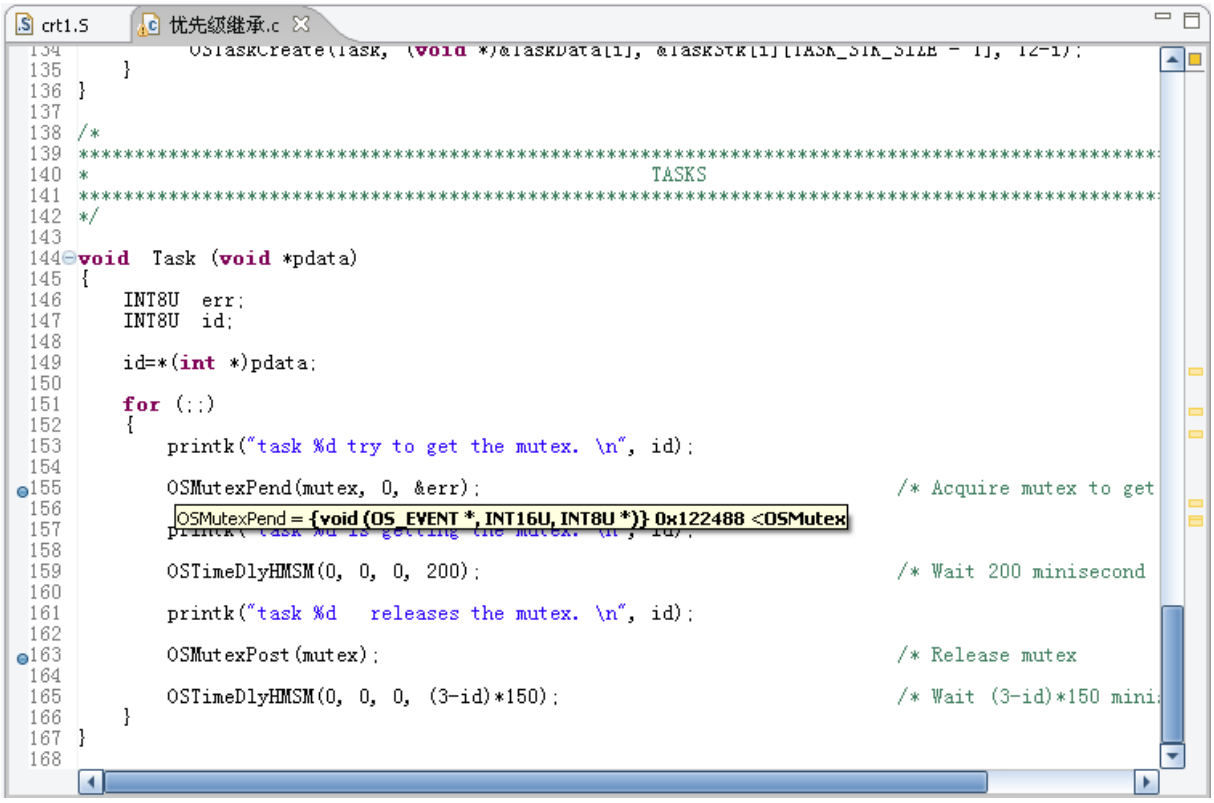
等到 Task1 延时时间到，它在 PIP 下运行，释放信号量，而 task2 就申请到了 mutex，Task1 就恢复原来的优先级高度，Task2 优先级高于 1 而抢占 CPU，12 同时延时

2 的延时先到期，释放 mutex，而被 0 获取，2 再延时，0 被调度，在 0 打印信息后把自己延时，而后 1 的延时到期，申请 mutex 而被阻塞，又开始优先级继承机制而提升 task0 的优先级到 PIP，空闲任务运行，

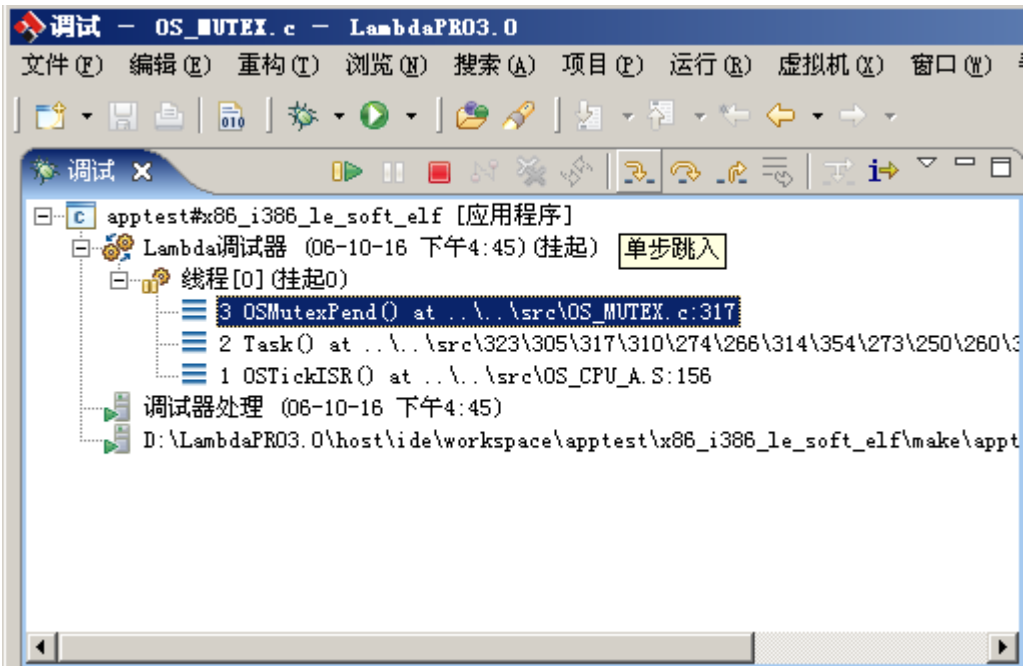
2 的延时到期，申请信号量被阻塞，但是 task0 的优先级现在是高于 2 的，所以不会触发优先级继承机制

4 应用调试过程

在 LambdaTOOL 的调试界面中，在“优先级继承.c”文件中 Task 任务代码调用函数 OSMutexPend() 和 OSMutexPost()处设置断点。



当程序运行至断点处时，选择“单步跳入”运行模式，可以进入到 OS_MUTEX.c 文件中，查看 OSMutexPend()和 OSMutexPost（）函数的运行过程，深入了解 μC/OS-II 操作系统的内核代码。



实验 4 信号量：哲学家就餐问题的实现

1 实验目的

掌握在基于嵌入式实时操作系统 μC/OS-II 的应用中，任务使用信号量的一般原理。通过经典的哲学家就餐实验，了解如何利用信号量来对共享资源进行互斥访问。

2 代码分析

五个哲学家任务（ph1、ph2、ph3、ph4、ph5）主要有两种过程：思考（即睡眠一段时间）和就餐。每个哲学家任务在就餐前必须申请并获得一左一右两支筷子，就餐完毕后释放这两支筷子。五个哲学家围成一圈，每两人之间有一支筷子。一共有五支筷子，在该实验中用了五个互斥信号量来代表。如图 4-1 所示：

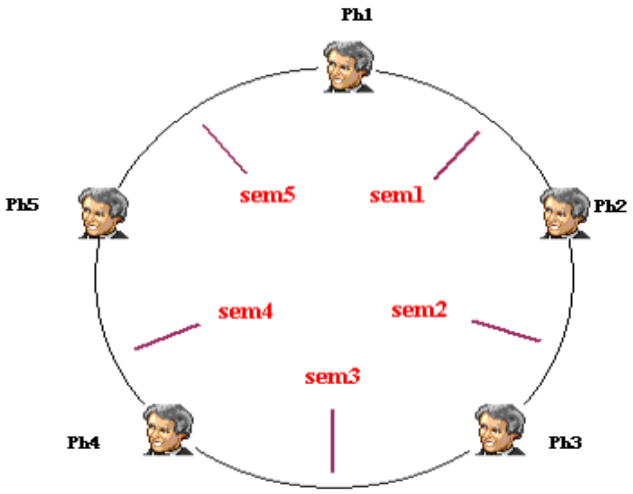


图 4-1

每个任务的代码都一样，如下所示：

```
INT8U    err;
INT8U    i;
INT8U    j;

i=*(int *)pdata;
j=(i+1) % 5;
for (;;) {
```

TASK_Thinking_To_Hungry(i); /* 首先哲学家处于 thinking 状态，然后执行 TASK_Thinking_To_Hungry()函数进入 hungry 状态*/

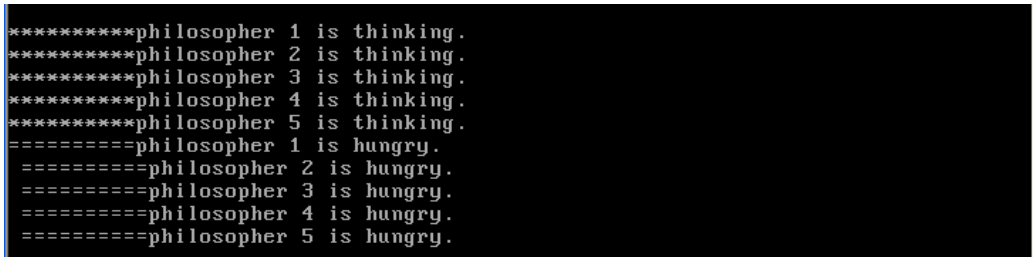
```
printk("\n");
```

/*申请信号量，在获得两个信号量后执行 TASK_Eat()函数进入 eating 状态*/

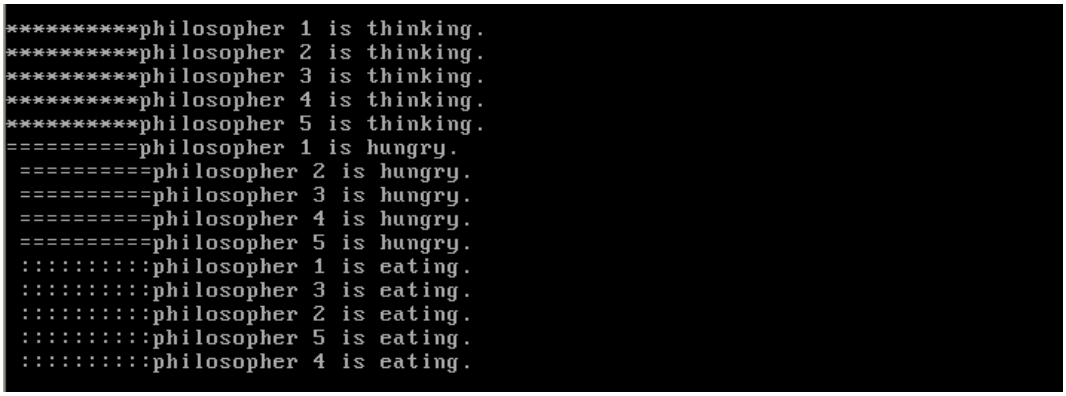
```
        OSSemPend(fork[i], 0, &err);
        OSSemPend(fork[j], 0, &err);            /* Acquire semaphores to eat */
        TASK_Eat(i);
        printk("\n");
    /*释放信号量*/
        OSSemPost(fork[j]);
        OSSemPost(fork[i]);                     /* Release semaphore            */
        OSTimeDly(200);                         /* Delay 200 clock tick         */
        }
```

3 运行及观察应用输出信息

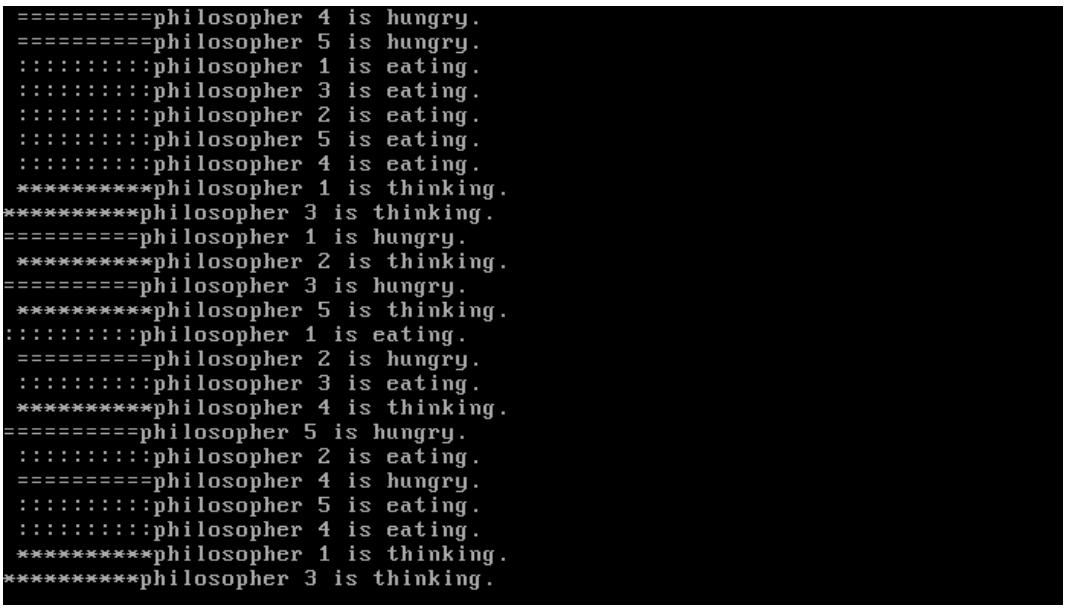
开始，所有的哲学家先处于 thinking 状态，然后都进入 hungry 状态：



之后首先获得两个信号量的 1、3 号哲学家开始 eating，待他们释放相关信号量之后，哲学家 2、5、4 获得所需的信号量并 eating：



应用如此这般地循环执行程序下去……


==========philosopher 4 is hungry.
==========philosopher 5 is hungry.
::::::::::philosopher 1 is eating.
::::::::::philosopher 3 is eating.
::::::::::philosopher 2 is eating.
::::::::::philosopher 5 is eating.
::::::::::philosopher 4 is eating.
**********philosopher 1 is thinking.
**********philosopher 3 is thinking.
==========philosopher 1 is hungry.
**********philosopher 2 is thinking.
==========philosopher 3 is hungry.
**********philosopher 5 is thinking.
::::::::::philosopher 1 is eating.
==========philosopher 2 is hungry.
::::::::::philosopher 3 is eating.
**********philosopher 4 is thinking.
==========philosopher 5 is hungry.
::::::::::philosopher 2 is eating.
==========philosopher 4 is hungry.
::::::::::philosopher 5 is eating.
::::::::::philosopher 4 is eating.
**********philosopher 1 is thinking.
**********philosopher 3 is thinking.

实验 5　μC/OS-II 的内存管理

1 实验目的

掌握嵌入式实时操作系统 μC/OS-II 内存管理中内存分配和回收的功能。

2 实验运行流程及代码分析

在该实验中，需要用到 μC/OS-II 内存管理中内存分配和回收的功能。为此，设计了如下图所示的应用：

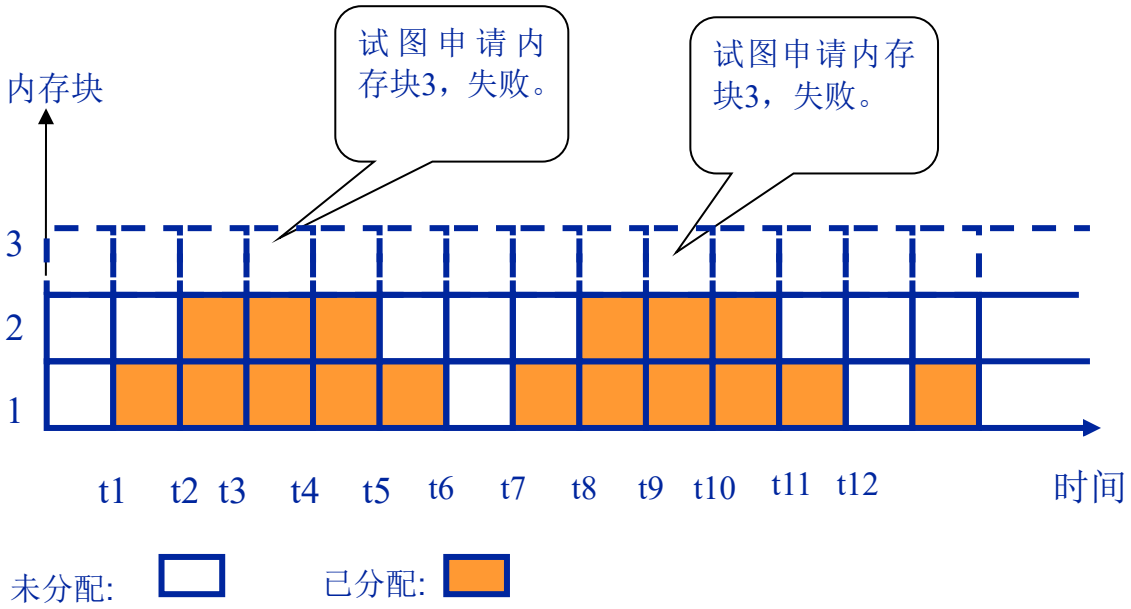


图 5-1

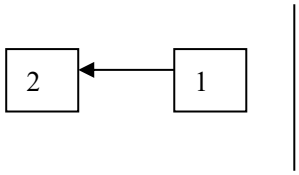
注意：根据程序中设定的时间延迟，图 5-1 中的每个栅格对应 100 个系统时钟周期。为了防止内存申请和释放的不合理导致的大块连续内存被分割成可用性小的小片的问题，μCos-II 将用于动态内存分配的空间分成一些固定大小的内存块，根据应用申请的内存大小，分配适当数量的内存块。图 5-1 的纵坐标就代表内存块。

在 main()函数中，使用 μCos-II 的 OSMemCreate（）函数创建一个用于动态内存分配的区域。通过传递适当的调用参数，我们在该区域中划分了 2 个 128B 的内存块。如果成功创建这些内存块，

μCos-II 会在内部建立并维护一个单向链表。

```
static void MemoryCreate()
{
        INT8U err;     //为 OSMemCreate 函数设置包含错误码的变量的指针。
        CommMem = OSMemCreate(&CommBuf[0][0], 2, 128, &err);          //OSMemCreate()函
                        数建立并初始化一块内存区。2 块，每块 128 字节。
}
```

以下就是刚刚初始化的内存区域的情况（为了表示方便，竖线右边表示已分配的内存块，竖线左边表示未被分配的内存块，下同）：



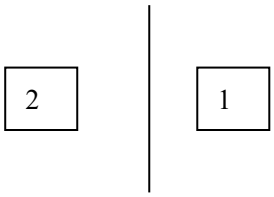
此时在虚拟机中可以观察到以下的情况：

应用任务使用函数 OSMemGet(CommMem,&err)来申请内存块，根据当前内存区域的情况来判断分配的结果：

```
if(i==0)
        {
        CommMsg1=OSMemGet(CommMem,&err);          //申请内存区中的一块。
            if (err == OS_NO_ERR)
            {
                printk("First memory application HAS accept.\n");
                    /*  内存块已经分配  */
            }
            else
            {
                printk("First memory alpplication NOT accept.\/n");
            }

        MemInfo(pdata);          //得到内存区的信息。
        DispShow();              //将内存区信息显示到屏幕上。
        OSTimeDly(100);         /* Wait 100tick                          */
```

```
i++;
}
```

申请过后，内存区域在 t2 时刻的状态如下：



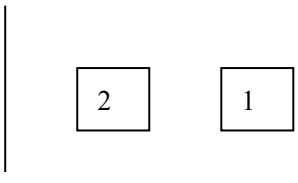
也就是说，内存块 1 已经被分配出去了，它的首地址储存在 OSMemGet（）函数的返回参数中。我们需要用一个变量（CommMsg1）记录这个返回值，以便在使用完这个内存块之后能正确地释放它。

在虚拟机中观察到的结果如下：

```
First memory application HAS accept.
the pointer to the begining of memory address is: 1201056
the size of blocks in this memory area is: 128
the number of free blocks in this memory area is: 1
the number of using blocks in this memory area is: 1
```

在 t3 时刻，内存块 2 也被分配了出去：

```
else if(i==1)
        {
        CommMsg2=OSMemGet(CommMem,&err);                    //申请内存区中的一块。
            if (err == OS_NO_ERR)
            {
                printk("Second memory application HAS accept.\n");
                                                            /*  内存块已经分配  */
            }
            else
            {
                printk("Second memory alpplication NOT accept.\n");
            }
```

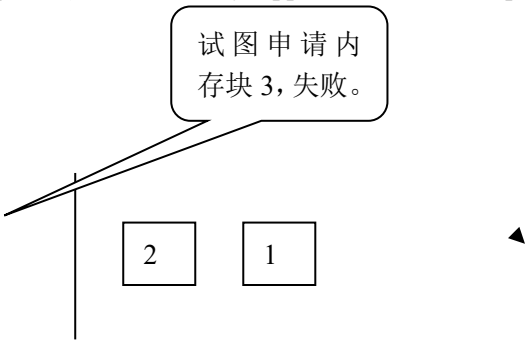


在虚拟机中看到的结果：

```
Second memory application HAS accept.
the pointer to the begining of memory address is: 1201056
the size of blocks in this memory area is: 128
the number of free blocks in this memory area is: 0
the number of using blocks in this memory area is: 2
```

接下来在 t4 时刻，应用试图申请内存块 3，但此时系统因为没有足够的内存块可以分配，所以失败：

```
CommMsg3=OSMemGet(CommMem,&err);
if (err == OS_NO_ERR)
{
    printk("Third memory application HAS accept.\n");
                                    /*  内存块已经分配  */
}
else
{
    printk("Third memory alpplication NOT accept.\n");
}
```



在虚拟机中看到的运行结果：

在 t5、t6 和 t7 时刻，应用相继将内存块 3、2、1 归还给系统，但是由于之前申请内存块 3 的操作是失败的，所以其归还操作是无效的：

```
for(i=3;i>0;i--)
{
ReleaseMem(i);              //释放第 i 个内存块。
MemInfo(pdata);             //函数得到内存区的信息
DispShow();                 //将内存区信息显示到屏幕上。
```

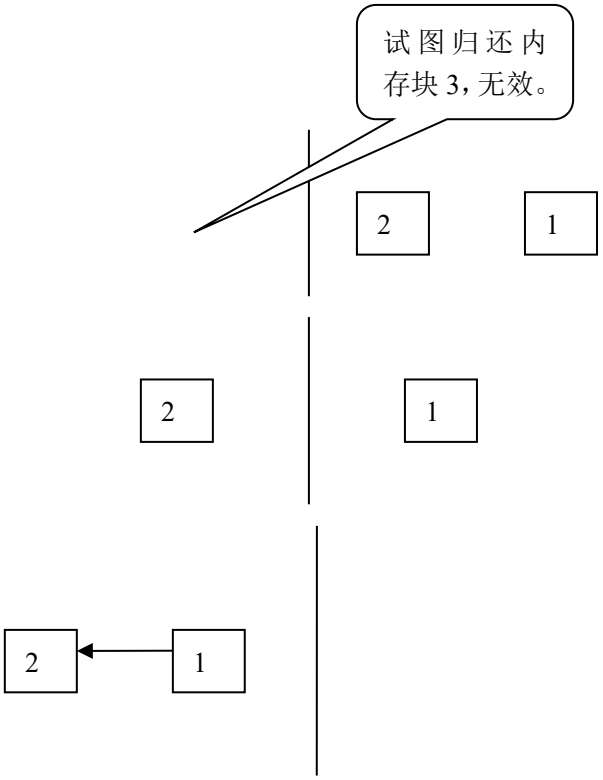
```
            OSTimeDly(10);              //延迟 10 个时钟周期
            }
```

释放内存块的函数代码如下：

```
void ReleaseMem(int i)
{
    INT8U err;
    switch(i)
    {
        case 3:OSMemPut(CommMem,CommMsg3);
                if (err == OS_NO_ERR)
                 {
                    printk("Third memory has been released.\n");
                                        /*  释放内存块        */
                 }
                else
                {
                    printk("Third memory didn't been releasd.\n");
                }
        break;

        case 2:OSMemPut(CommMem,CommMsg2);
                if (err == OS_NO_ERR)
                 {
                    printk("Second memory has been released.\n");
                                        /*  释放内存块        */
                 }
                else
                {
                    printk("Second memory didn't been releasd.\n");
                }
        break;

        case 1:OSMemPut(CommMem,CommMsg1);
                if (err == OS_NO_ERR)
                 {
                    printk("First memory has been released.\n");
                                        /*  释放内存块        */
                 }
                else
                {
                    printk("First memory didn't been releasd.\n");
                }
        break;
```

```
    }
}
```

过程如下：



在虚拟机中看到的运行结果：

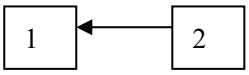
```
Third memory didn't exist.
the pointer to the begining of memory address is: 1200256
the size of blocks in this memory area is: 128
the number of free blocks in this memory area is: 0
the number of using blocks in this memory area is: 2
```

这里应该注意，μCos-II 的内存块归还函数是：OSMemPut(CommMem,CommMsg3);

它的返回值只有两种：OS_NO_ERR ：成功归还内存块

OS_MEM_FULL ：内存区已满，不能再接受更多释放的内存块。出现这种情况说明用户程序出现了错误，归还了多于用 OSMemGet（）函数得到的内存块。

如果先归还内存块 1，后归还内存块 2，则是如下情况：



虽然内存块的地址并没有变化，但是链表的结构发生了变化。也就是说，如果进行多次内存分配和归还的话，那么最终的链表和初始化时的链表会完全不同。

另外，在这个应用中使用了两个函数：

```
MemInfo(pdata);                    //函数得到内存区的信息
DispShow();                        //将内存区信息显示到屏幕上。
```

它们的代码如下：

```
static void DispShow()
    {
    INT8U          err;
    INT8U          s[4][50];

    OS_MEM_DATA mem_data;
    OSMemQuery(CommMem, &mem_data);

    printk( "the pointer to the begining of memory address is: %d\n",(int)(mem_data.OSAddr));
    printk( "the size of blocks in this memory area is: %d\n",(int)(mem_data.OSBlkSize));
    printk( "the number of free blocks in this memory area is: %d\n", (int)(mem_data.OSNFree));
    printk( "the number of using blocks in this memory area is: %d\n", (int)mem_data.OSNUsed);
    printk("\n\n\n");
    OSTimeDlyHMSM(0,0,8,0);
}
void MemInfo(void *pdata)   //内存空间信息。
{
      INT8U          err;      //为函数设置包含错误码的变量指针。
      pdata = pdata;
      OS_MEM_DATA mem_data;
      err = OSMemQuery(CommMem, &mem_data);
                              //OSMemQuery（）函数得到内存区的信息。
}
```

OSMemCreate（）

该函数建立并初始化一个用于动态内存分配的区域，该内存区域包含指定数目的、大小确定的内存块。应用可以动态申请这些内存块并在用完后将其释放回这个内存区域。该函数的返回值就是指向这个内存区域控制块的指针，并作为 OSMemGet（），OSMemPut（），OSMemQuery（）等相关调用的参数。

函数原型：OSMemCreate(void *addr, INT32U nblks ,INT32U blksize, INT8U *err);

参数说明：addr 建立的内存区域的起始地址。可以使用静态数组或在系统初始化时使用 malloc（）函数来分配这个区域的空间。

Nblks 内存块的数目。每一个内存区域最少需要定义两个内存块。

Blksize 每个内存块的大小，最小应该能够容纳一个指针变量。

Err 是指向包含错误码的变量的指针。Err 可能是如下几种情况：

- OS_NO_ERR ：成功建立内存区域。
- OS_MEM_INVALID_ADDR：非法地址，即地址为空指针。

- OS_MEM_INVALID_PART ：没有空闲的内存区域。
- OS_MEM_INVALID_BLKS ：没有为内存区域建立至少两个内存块。
- OS_MEM_INVALID_SIZE ：内存块大小不足以容纳一个指针变量。

返回值 ：OSMemCreate（）函数返回指向所创建的内存区域控制块的指针。如果创建失败，函数返回空指针。

OSMemGet（）

该函数用于从内存区域分配一个内存块。用户程序必须知道所建立的内存块的大小，并必须在使用完内存块后释放它。可以多次调用 OSMemGet（）函数。它的返回值就是指向所分配内存块的指针，并作为 OSMemPut（）函数的参数。

函数原型：OSMemGet(OS_MEM *pmem, INT8U *err);

参数说明：pmem 是指向内存区域控制块的指针，可以从 OSMemCreate（）函数的返回值中得到。

Err 是指向包含错误码的变量的指针。Err 可能是如下情况：

- OS_NO_ERR ：成功得到一个内存块。
- OS_MEM_NO_FREE_BLKS ：内存区域中已经没有足够的内存块。

返回值：OSMemGet（）函数返回指向所分配内存块的指针。如果没有可分配的内存块，OSMemGet（）函数返回空指针。

OSMemPut（）

该函数用于释放一个内存块，内存块必须释放回它原先所在的内存区域，否则会造成系统错误。

函数原型：OSMemPut(OS_MEM *pmem, void *pblk);

参数说明：pmem 是指向内存区域控制块的指针，可以从 OSMemCreate（）函数的返回值中得到。

Pblk 是指向将被释放的内存块的指针。

返回值 ：OSMemPut（）函数的返回值为下述之一：

- OS_NO_ERR ：成功释放内存块
- OS_MEM_FULL ：内存区域已满，不能再接受更多释放的内存块。这种情况说明用户程序出现了错误，释放了多于用 OSMemGet（）函数得到的内存块。

OSMemQuery（）

该函数用于得到内存区域的信息。

函数原型：OSMemQuery(OS_MEM *pmem, OS_MEM_DATA *pdata);

参数说明：

pmem 是指向内存区域控制块的指针，可以从 OSMemCreate（）函数的返回值中得到。

Pdata 是一个指向 OS_MEM_DATA 数据结构的指针，该数据结构包含了以下的域：

```
Void    OSAddr;          /*指向内存区域起始地址的指针          */
```

```
Void     OSFreeList;    /*指向空闲内存块列表起始地址的指针      */
INT32U   OSBlkSize;     /*每个内存块的大小                      */
INT32U   OSNBlks;       /*该内存区域中的内存块总数              */
INT32U   OSNFree;       /*空闲的内存块数目                      */
INT32U   OSNUsed;       /*已使用的内存块数目                    */
```

实验 6 时钟中断

1 实验目的

掌握嵌入式实时操作系统 μC/OS 中中断的使用情况。

2 实验运行流程及代码分析

2.1 实验流程

三个任务 123 的优先级依次降低，有一个初值为 1 的信号量 InterruptSem（12 竞争）

先调度 1,1 把自己延时，2 开始执行，2 获取信号量，2 又把自己延时，3 开始执行，打印输出后延时，1 延时时间到并投入运行，但申请信号量失败被阻塞，3 继续运行并打印输出，期间时钟中断不断产生，在中断处理程序中对任务 Task2 的睡眠时间进行计数

Task2 睡眠时间到后恢复运行，并释放信号量 InterruptSem，Task1 获得信号量 InterruptSem 后抢占 Task2 运行，Task1 使用完信号量 InterruptSem 后释放该信号量

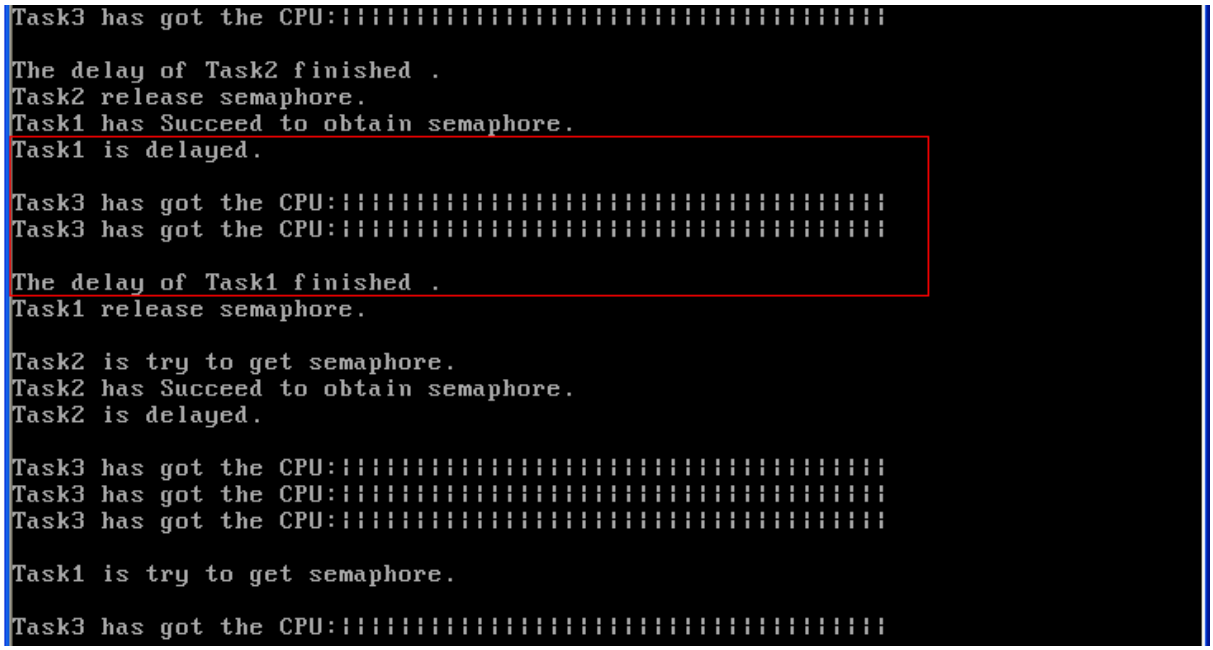
2.2 代码分析

程序代码如下：

```
void    Task1 (void *pdata)
         {
             INT8U    err;
             pdata=pdata;

        for (;;)
        {
       OSTimeDly(100);
       printk("\nTask1is try to get semaphore.\n\n");          /*task1 delay 100 clock ticks       */
       OSSemPend(InterruptSem, 0, &err);         /* Acquire semaphore to get into the room */
       printk("Task1 has Succeed to obtain semaphore.\n");
       printk("Task1 is delayed.\n\n");
       OSTimeDly(200);
       printk("\nThe delay of Task1 finished .\n");
       printk("Task1 release semaphore.\n");
       OSSemPost(InterruptSem);          /* Release semaphore                */
       OSTimeDly(200);
        }
            }

void Task2 (void *pdata)
         {   INT8U err;
             pdata=pdata;
```

```
    for (;;)
        {
        printk( "\nTask2 is try to get semaphore.\n");
        OSSemPend(InterruptSem, 0, &err);       /* Acquire semaphore to get into the room */
        printk( "Task2 has Succeed to obtain semaphore.\n");
        printk("Task2 is delayed.\n\n");
        OSTimeDly(500);          /*task2 delay 500 clock ticks            */
        printk("\nThe delay of Task2 finished .\n");
        printk("Task2 release semaphore.\n");
        OSSemPost(InterruptSem);              /* Release semaphore                    */
        OSTimeDly(200);
        }
}

void   Task3 (void *pdata)
{
    pdata=pdata;
    for (;;)
    {
        printk("Task3 has got the CPU:||||||||||||||||||||||||||||||||||||\n");
        OSTimeDly(100);

    }
}
```

3 实验实现与结果分析



三个任务 123 的优先级依次降低，有一个初值为 1 的信号量 InterruptSem（12 竞争）

先调度 1,1 把自己延时，2 开始执行，2 获取信号量，2 又把自己延时，3 开始执行，打印输出后延时，1 延时时间到并投入运行，但申请信号量失败被阻塞，3 继续运行并打印输出，期间时钟中断不断产生，在中断处理程序中对任务 Task2 的睡眠时间进行计数

Task2 睡眠时间到后恢复运行，并释放信号量 InterruptSem，Task1 获得信号量 InterruptSem 后抢占 Task2 运行，Task1 使用完信号量 InterruptSem 后释放该信号量

实验 7 消息队列

1 实验目的

掌握嵌入式实时操作系统 μC/OS-II 中消息队列机制的基本原理和使用方法。

2 实验运行流程及代码分析

2.1 实验运行流程

在本实验中，设计了 6 个普通应用任务：TA0（优先级为 1）、TA1（优先级为 2）、TA2（优先级为 3）、TA3（优先级为 4）、TA4（优先级为 5）、TA5（优先级为 6），以及一个控制任务 TaskCon（优先级为 7）。

创建一个等待属性为 FIFO 的消息队列 1；创建一个等待属性为 LIFO 的消息队列 2。

由任务 TA0、TA1、TA2 等待队列 1 中的消息。TA0、TA1、TA2 使用相同的任务代码（Taskq1 函数）。

由任务 TA3、TA4、TA5 等待队列 2 中的消息。TA3、TA4、TA5 使用相同的任务代码（Taskq2 函数）。

TaskCon 任务向队列 2 中连续发送 6 条消息，然后查询消息数；清空该队列后再查询。

在任务 TA3、TA4、TA5 等待队列 2 中的消息的过程中，让 TaskCon 删除队列 2；当队列 2 被删除后，检查任务 TA3、TA4、TA5 调用接收消息的函数是否返回错误码。

2.2 代码分析

在 main()函数中通过 q1 = OSQCreate(&Msg1[0],6);q2 = OSQCreate(&Msg2[0],6);　创建两个消息队列。

在 TaskStart 任务中创建并启动所有的应用任务。

```
static  void  TaskStartCreateTasks (void)
{
    INT8U  i;
    for (i = 0; i <N_TASKS; i++) {        /* Create N_TASKS identical tasks  */
        TaskData1[i] = i;                 /* Each task will pass its own id  */
        OSTaskCreate(Taskq1, (void *)&TaskData1[i], &TaskStk1[i][TASK_STK_SIZE - 1], i+1);
    }
    for (i = 0; i <N_TASKS; i++) {          /* Create N_TASKS identical tasks */
        TaskData2[i] = i;                   /* Each task will pass its own id  */
        OSTaskCreate(Taskq2, (void *)&TaskData2[i], &TaskStk2[i][TASK_STK_SIZE - 1], i+4);
```

```
        }
        OSTaskCreate(TaskCon, (void *)0, &TaskConStk[TASK_STK_SIZE - 1], i+4);
                                    /* Create control tasks    */
    }
```

应用任务 TA0 的代码（TA1，TA2 相同）：从队列 q1 中按 LIFO 方式取消息。

```
void    Taskq1 (void *pdata)
{
    INT8U    err;
    INT8U    id;
    char     s[30];
    void     *mg;

    id=*(int *)pdata;

    for (;;) {
        OSTimeDlyHMSM(0, 0, 2, 0);              /* Wait 2 second                                    */
        mg=OSQPend(q1,0,&err);                   /* apply for message                                */
        switch(err){
                case OS_NO_ERR:{
                    printk("       task %d has got the %s \n",id,(char *)mg);
                    OSTimeDlyHMSM(0, 0, 0, 200*(4-id));
                    break;
                }                                   /* If it is normally, just print the string.*/
                default :{
                    printk( "queue1 %d    is empty.                \n",id);
                    OSTimeDlyHMSM(0, 0, 0, 200*(4-id));
                    break;
                }           /* If the queue is empty or has been deleted, print another string.*/
        }
     }
}
```

应用任务 TA3 的代码（TA4，TA5 相同）：从队列 q2 中按 FIFO 方式取消息。

```
void    Taskq2 (void *pdata)
```

```
{
    INT8U    err;
    INT8U    id;
    char     s[30];
    void     *mg;

    id=*(int *)pdata;

    for (;;) {
        OSTimeDlyHMSM(0, 0, 2, 0);              /* Wait 2 second                    */
        mg=OSQPend(q2,0,&err);                  /* apply for message                */
        switch(err){
                case OS_NO_ERR:{
                    printk("       task %d has got the %s. \n", id+3, (char *)mg);
                    OSTimeDlyHMSM(0, 0, 0, 200*(4-id));
                    break;
                }                                   /* If it is normally, just print the string.*/
                default :{
                    printk( "queue2    is empty,%d can't got the message.          \n",id+3);
                    OSTimeDlyHMSM(0, 0, 0, 200*(4-id));
                    break;
                }           /* If the queue is empty or has been deleted, print another string.*/
        }
     }
}
```

控制任务

```
void    TaskCon (void *pdata)
{
    INT8U    i,j;
    INT8U    err;
    INT8U    note=1;                        /* for flush the queue              */
    INT16U    del=3;                          /* for delete the queue             */
    OS_EVENT *q;
    char     ch[50];
```

```
OS_Q_DATA *data;
static char *s[]={                          /* queue1's message                    */
    "message0","message1","message2","message3","message4","message5"
};

static char *t[]={                          /* queue2's message                    */
    "messageA","messageB","messageC","messageD","messageE","messageF"
};

pdata=pdata;

for (;;)
 {
    printk("\n-------------Add message to queue1-------------\n");
    for( i = 0 ; i < 6 ; i++ )
    {
        err = OSQPostFront(q1,(void*)s[i]);          /* post message to q1 LIFO          */
        switch(err){
            case OS_NO_ERR:{
                printk("the queue1 %d add %s\n",i,s[i]);
                OSTimeDlyHMSM(0, 0, 0, 150);
                break;
            }
            case OS_Q_FULL:{
                printk("the queue1 is full, don't add.\n");
                OSTimeDlyHMSM(0, 0, 0, 150);
                break;
            }
            default :break;
        }
    }
    if(del>=0){
    printk("\n-------------Add message to queue2-------------\n");
    }
    for( j = 0 ; j < 6 ; j++ )
```

```
        {
            err = OSQPost(q2,(void*)t[j]);          /* post message to q2 FIFO              */
            switch(err){
                case OS_NO_ERR:{
                    printk("the queue2 %d add %s\n",j,t[j]);
                    OSTimeDlyHMSM(0, 0, 0, 150);
                    break;
                }
                case OS_Q_FULL:{
                    printk("the queue2 is full, don't add. \n");
                    OSTimeDlyHMSM(0, 0, 0, 150);
                    break;
                }
                default :break;
            }
        }

    if(del>=0){
    if(note==1)
        {
            OSQFlush(q2);
            printk("\n----------clear up the queue2----------\n");      /* clear up the queue 2. */
            note=0;
        }
        else
            note=1;
    }
        err=OSQQuery(q2,data);              /* get the information about q2                */
        if(err==OS_NO_ERR){
        printk("\n----------the queue2'information----------\n");
        printk(" NextMsg:%s, NumMsg:%d, QSize:%d.\n",(char )data->OSMsg, data->OSNMsgs,
data->OSQSize);
        printk("------------------------------------------------\n");
        }                               /* print the information about q2              */
        OSTimeDlyHMSM(0, 0, 0, 500);        /* Wait 500 minisecond                 */
```

```
        printk("\n-----------------------------------\n",);
        if(del==0)
        {
            q=OSQDel(q2,OS_DEL_ALWAYS,&err); /* delete the    q2              */
            if(q==(OS_EVENT *)0)
            {
                printk("        already successful delete queue2              \n");
            }
        }
        else
        {
            del--;
            printk( "          not successful delete queue2    \n");
        }
        printk("-----------------------------------\n");
    }
}
```

OSQCreate（）

该函数用于建立一个消息队列。任务或中断可以通过消息队列向一个或多个任务发送消息。消息的含义是和具体的应用密切相关的。

函数原型：OS_EVENT   *OSQCreate(void **start, INT8U size);

参数说明：

start　是消息内存区的首地址，消息内存区是一个指针数组。

Size　是消息内存区的大小。

返回值：

OSQCreate（）函数返回一个指向消息队列控制块的指针。如果没有空闲的控制块，OSQCreate（）函数返回空指针。

OSQPend（）

该函数用于任务等待消息。消息通过中断或任务发送给需要的任务。消息是一个指针变量，在不同的应用中消息的具体含义不同。如果调用 OSQPend（）函数时队列中已经存在消息，那么该消息被返回给 OSQPend（）函数的调用者，该消息同时从队列中清除。如果调用 OSQPend（）函数时队列中没有消息，OSQPend（）函数挂起调用任务直到得到消息或超出定义的超时时间。如果同时有多个任务等待同一个消息，μC/OS-II默认最高优先级的任务取得消息。一个由 OSTaskSuspend（）

函数挂起的任务也可以接受消息，但这个任务将一直保持挂起状态直到通过调用 OSTaskResume（）函数恢复任务的运行。

函数原型：Void *OSQPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);

参数:

pevent 是指向消息队列的指针，该指针的值在建立该队列时可以得到。（参考 OSMboxCreate（）函数）。

Timeout 允许一个任务以指定数目的时钟节拍等待消息。超时后如果还没有得到消息则恢复成就绪状态。如果该值设置成零则表示任务将持续地等待消息，最大的等待时间为 65535 个时钟节拍。这个时间长度并不是非常严格的，可能存在一个时钟节拍的误差。

Err 是指向包含错误码的变量的指针。OSQPend（）函数返回的错误码可能为下述几种：

- OS_NO_ERR ：消息被正确地接受。
- OS_TIMEOUT ：消息没有在指定的时钟周期数内接收到消息。
- OS_ERR_PEND_ISR ：从中断调用该函数。虽然规定了不允许从中断中调用该函数，但μC/OS-II仍然包含了检测这种情况的功能。
- OS_ERR_EVENT_TYPE ：pevent 不是指向消息队列的指针。

返回值：

OSQPend（）函数返回取得的消息并将*err 置为 OS_NO_ERR。如果没有在指定数目的时钟节拍内接受到消息，OSQPend（）函数返回空指针并将*err 设置为 OS_TIMEOUT。

OSQPostFront()

该函数用于向消息队列发送消息。OSQPostFront（）函数和 OSQPost（）函数非常相似，不同之处在于 OSQPostFront（）函数将发送的消息插到消息队列的最前端。也就是说，OSQPostFront（）函数使得消息队列按照后入先出（LIFO）的方式工作，而不是先入先出（FIFO）。消息是一个指针长度的变量，在不同的应用中消息的含义也可能不同。如果队列中已经存满消息，则此调用将返回错误码。OSQPost（）函数也是如此。在调用此函数时如果有任何任务在等待队列中的消息，则最高优先级的任务将得到这个消息。如果等待消息的任务优先级比发送消息的任务优先级高，那么高优先级的任务在得到消息后将立即抢占当前任务执行，也就是说，将发生一次任务切换。

函数原型：INT8U OSQPostFront(OS_EVENT *pevent, void *msg);

参数：

pevent 是指向即将接收消息的消息队列的指针。该指针的值在建立队列时可以得到。（参考 OSQCreate（）函数）。

Msg 是即将发送的消息的指针。不允许传递一个空指针。

返回值：

OSQPost（）函数的返回值为下述之一：

- OS_NO_ERR ：消息成功地放到消息队列中。
- OS_MBOX_FULL ：消息队列已满。
- OS_ERR_EVENT_TYPE ：pevent 不是指向消息队列的指针。

OSQPost()

该函数用于向消息队列发送消息。消息是一个指针长度的变量，在不同的应用中消息的含义也可能不同。如果队列中已经存满消息，则此调用返回错误码。如果有任何任务在等待队列中的消息，则最高优先级的任务将得到这个消息。如果等待消息的任务优先级比发送消息的任务优先级高，那么高优先级的任务将在得到消息后立即抢占当前任务执行，也就是说，将发生一次任务切换。消息是以先入先出（FIFO）方式进入队列的，即先进入队列的消息先被传递给任务。

函数原型：INT8U OSQPost(OS_EVENT *pevent, void *msg);

参数：

pevent 是指向即将接受消息的消息队列的指针。该指针的值在建立队列时可以得到。（参考 OSQCreate（）函数）。

Msg 是即将发送给队列的消息。不允许传递一个空指针。

返回值：

OSQPost（）函数的返回值为下述之一：

- OS_NO_ERR ：消息成功地放到消息队列中。
- OS_MBOX_FULL ：消息队列已满。
- OS_ERR_EVENT_TYPE ：pevent 不是指向消息队列的指针。

OSQFlush ()

该函数用于清空消息队列。

函数原型：INT8U *OSQFlush（OS_EVENT *pevent）;

参数：

pevent 是指向消息队列的指针。该指针的值在建立队列时可以得到。（参考 OSQCreate（）函数）。

返回值：

OSQFlush（）函数的返回值为下述之一：

- OS_NO_ERR ：消息队列被成功清空
- OS_ERR_EVENT_TYPE ：试图清除不是消息队列的对象

OSQQuery()

该函数用来取得消息队列的信息。用户程序必须建立一个 OS_Q_DATA 的数据结构，该结构用来保存从消息队列的控制块得到的数据。通过调用该函数可以知道是否有任务在等待消息、有多少个任务在等待消息、队列中有多少消息以及消息队列可以容纳的消息数。OSQQuery（）函数还可以得到即将被传递给任务的消息。

函数原型：INT8U OSQQuery(OS_EVENT *pevent, OS_Q_DATA *pdata);

参数：

pevent 是指向消息队列的指针。该指针的值在建立消息队列时可以得到。（参考 OSQCreate（）函数）。

Pdata 是指向 OS_Q_DATA 数据结构的指针，该数据结构包含下述成员：

```
Void        *OSMsg;                          /* 下一个可用的消息*/
INT16U      OSNMsgs;                         /*  队列中的消息数目*/
INT16U      OSQSize;                         /*  消息队列的大小  */
INT8U       OSEventTbl[OS_EVENT_TBL_SIZE];   /*  消息队列的等待队列*/
INT8U       OSEventGrp;
```

返回值：

OSQQuery（）函数的返回值为下述之一：

- OS_NO_ERR ：调用成功
- OS_ERR_EVENT_TYPE ：pevent 不是指向消息队列的指针。

OSQDel()

该函数用于删除指定的消息队列。

OSTimeDlyHMSM()

该函数用于将一个任务延时若干时间。延时的单位是小时、分、秒、毫秒。调用 OSTimeDlyHMSM（）后，如果延时时间不为 0，系统将立即进行任务调度。

函数原型: void OSTimeDlyHMSM(INT8U hours，INT8U minutes，INT8U seconds，INT8U milli);

参数：

hours 为延时小时数，范围从 0-255。

minutes 为延时分钟数，范围从 0-59。

seconds 为延时秒数，范围从 0-59

milli 为延时毫秒数，范围从 0-999。

需要说明的是，操作系统在处理延时操作时都是以时钟节拍为单位的，实际的延时时间是时钟

节拍的整数倍。如果系统时钟节拍的间隔是 10ms，而设定延时为 5ms 的话，则不会产生延时操作；而如果设定延时为 15ms，则实际的延时是两个时钟节拍，也就是 20ms。

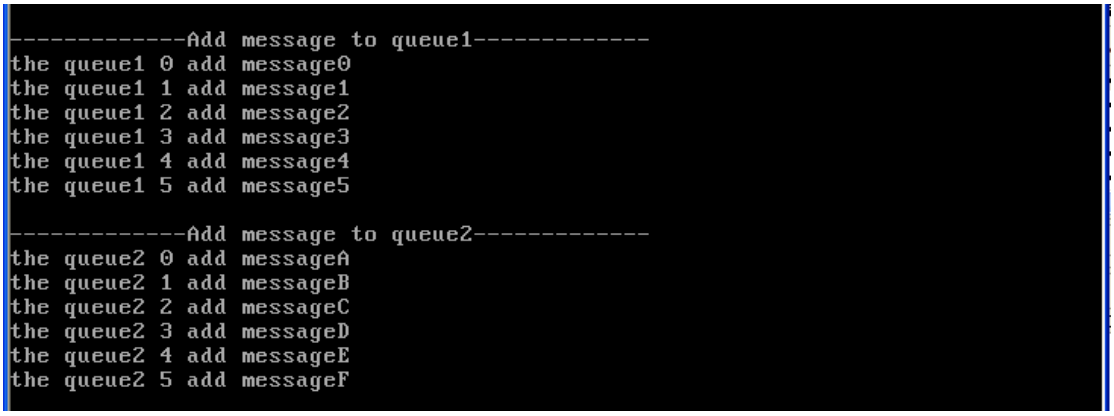
返回值：

OSTimeDlyHMSM（）的返回值为下述之一：

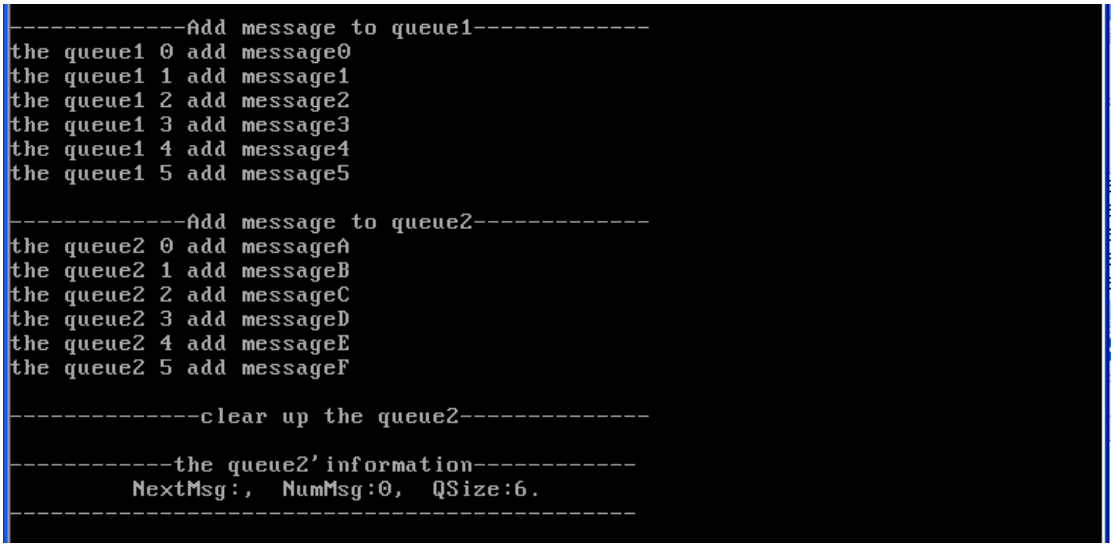
- OS_NO_ERR：函数调用成功。
- OS_TIME_INVALID_MINUTES：参数错误，分钟数大于 59。
- OS_TIME_INVALID_SECONDS：参数错误，秒数大于 59。
- OS_TIME_INVALID_MILLI：参数错误，毫秒数大于 999。
- OS_TIME_ZERO_DLY：四个参数全为 0。

3 实验实现与结果分析

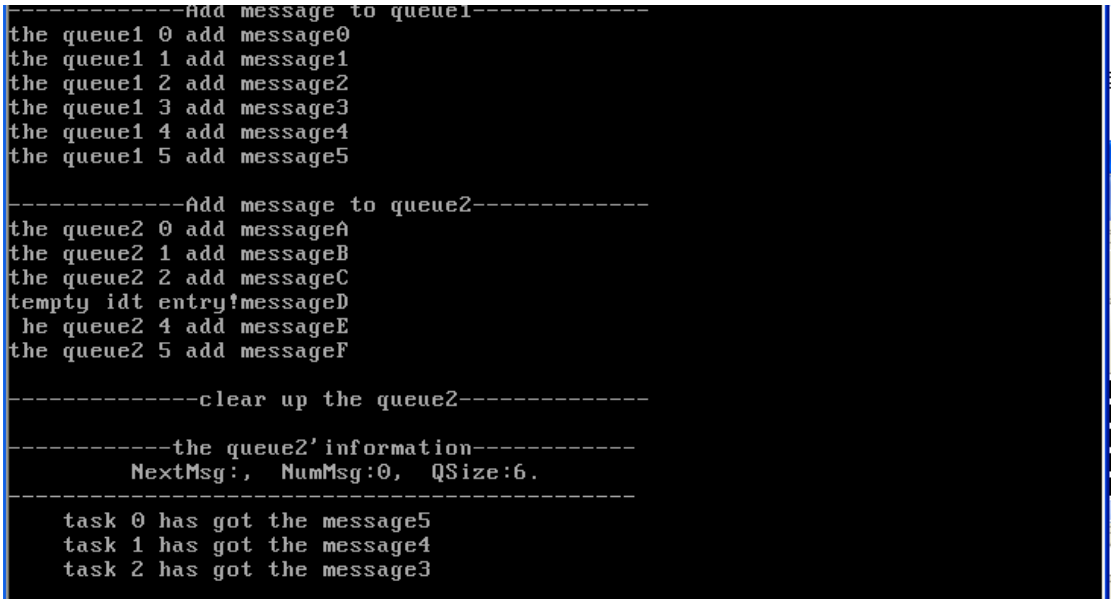
（1）首先，任务 TaskCon 将消息放入消息队列中：



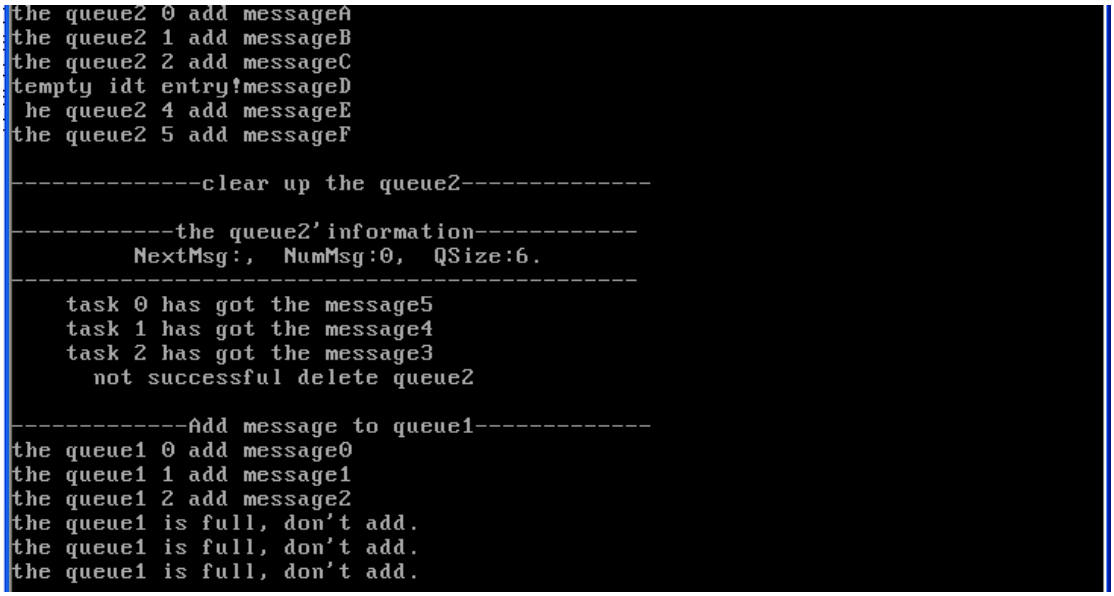
(2）任务 TaskCon 清空消息队列 2，之后查询消息队列 2 的内容：



（3）TA0—TA2 按照 LIFO 顺序获取消息：



（4）任务 TaskCon 再次将消息放入队列中，TA0—TA2 按照 LIFO 顺序获取消息，TA3—TA5 按照 FIFO 顺序获取消息。



```
-------------------------------------------
    task 0 has got the message5
    task 1 has got the message4
    task 2 has got the message3
      not successful delete queue2

-------------Add message to queue1-------------
the queue1 0 add message0
the queue1 1 add message1
the queue1 2 add message2
the queue1 is full, don't add.
the queue1 is full, don't add.
the queue1 is full, don't add.

-------------Add message to queue2-------------
    task 3 has got the messageA.
the queue2 0 add messageA
    task 4 has got the messageB.
the queue2 1 add messageB
    task 5 has got the messageC.
the queue2 2 add messageC
the queue2 3 add messageD
the queue2 4 add messageE
the queue2 5 add messageF
```

（5）任务 TaskCon 将消息队列 2 删除以后，TA3—TA5 就不能获取消息了。

```
the queue1 is full, don't add.
    task 1 has got the message0
the queue1 3 add message3
the queue1 is full, don't add.
the queue1 is full, don't add.

-------------Add message to queue2-------------

--------------clear up the queue2--------------
    task 0 has got the message3
queue2  is empty,5 can't got the message.

-------------Add message to queue1-------------
the queue1 0 add message0
the queue1 is full, don't add.
the queue1 is full, don't add.
queue2  is empty,4 can't got the message.
the queue1 is full, don't add.
the queue1 is full, don't add.
    task 2 has got the message0
the queue1 5 add message5
queue2  is empty,3 can't got the message.

-------------Add message to queue2-------------
```

小组协作：修改信号量相关代码

1 实验要求

Part1：OSSemPend 中能够判断 sem 是来自 post 还是 del。

Part2：信号量的 post 和 pend 操作中，调度任务就绪的逻辑从按照优先级改为先进先出。

Part3：create 里面为什么要判断是否在中断中

2 实现原理

Part1：

在 post 和 del 中，OS_EventTaskRdy 函数的第二个参数是没有意义的，取值 0/1 都不会影响逻辑

Part2：

通过在 TCB 结构体中新增两个成员指针 TCBFifoNext 和 TCBFifoPrev，用双向循环链表构造实现先进先出的队列。重用信号量中 OS_EVENT 结构体没有使用的 OSEventPtr，指向循环链表的链表头。

os_task:

OSTaskDel 函数中，在要删除的任务正在等待一个信号量时，处理新增的指针，把任务从等待队列中删除。

os_core：

OS_TCBInit 函数中新增对两个指针的初始化（初始化为空）。

新增函数 OS_EventTaskRdy_Fifo，把队列中的第一个任务出队，同时保留了 OS_EventTaskRdy 函数中将任务从事件等待表中移除、加入就绪表的操作。

新增函数 OS_EventTaskWait_Fifo，把任务插入队列尾部，同时保留了 OS_EventTaskWait 函数中将任务加入事件等待表、从就绪表中移除的操作。

新增函数 OS_EventTO_Fifo，把超时的任务从队列中移除，加入就绪表，TCB 中的两个指针置为空。

os_sem:

OSSemCreate 函数中初始化 OSEventPtr 为空。

OSSemDel 函数中，修改判断是否有任务等待该信号量的判断条件，改为根据 OSEventPtr 判断；调用 OS_EventTaskRdy 改为调用新增的 OS_EventTaskRdy_Fifo。

OSSemPend 函数中，调用 OS_EventTaskWait 和 OS_EventTO，改为调用新增的 OS_EventTaskWait_Fifo 和 OS_EventTO_Fifo。

OSSemPost 函数中，逻辑同上，修改调用的函数和判断条件。

Part3：

1.内核对象管理器的非可重入性

在 OSSemCreate() 中操作的 OSEventFreeList 链表是全局共享资源：

```
OS_ENTER_CRITICAL();
pevent = OSEventFreeList; // 获取空闲事件控制块
if (OSEventFreeList != (OS_EVENT *)0) {
    OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr; // 修改全局链表指针
}
```

若在 ISR 中执行这些操作，当中断嵌套发生时可能引发链表指针混乱。例如：高优先级中断打断低优先级中断的 OSEventFreeList 分配过程，导致链表状态不一致。

2.防止无效的任务切换

信号量创建后可能立即被使用，但 ISR 上下文无法进行完整的任务调度。当调用 OS_Sched() 时：

```
void OS_Sched (void)
{
    if (OSIntNesting > 0) {   // 在中断中直接返回
        return;
    }
    ...
}
```

在 ISR 中即使创建信号量后调用 OSSemPost() ，调度器也不会触发真正的任务切换，导致新创建信号量的使用出现延迟。

3.内存分配原子性保障

uC/OS-II 的 OSEventFreeList 管理依赖临界区保护：

```
void OSInit (void)
{
    ...
```

```
        // ECB 初始化代码
        for (i = 0; i < (OS_MAX_EVENTS - 1); i++) {
            pevent1->OSEventPtr = pevent2; //  建立空闲链表
            pevent1++;
            pevent2++;
        }
    }
```

若允许在 ISR 中创建信号量，当中断抢占正在执行 OSSemCreate() 的任务时，可能破坏链表指针。例如：任务 A 正在修改 OSEventFreeList 指针时被中断 B 打断，中断 B 同样尝试获取 OSEventFreeList，会导致链表节点丢失或重复分配。

3 代码实现

Part1：

post

```c
INT8U OSSemPost (OS_EVENT *pevent) {

if OSCRITICALMETHOD == 3
OS_CPU_SR    cpu_sr = 0;
endif
if OSARGCHK_EN > 0
if (pevent == (OS_EVENT *)0) {
    return (OS_ERR_PEVENT_NULL);
}
endif
if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {
    return (OS_ERR_EVENT_TYPE);
}
OS_ENTER_CRITICAL();
if (pevent->OSEventGrp != 0) {
    (void)OS_EventTaskRdy(pevent, (void *)1, OS_STAT_SEM);          //此处进行了修改，第二参数改为 1
    OS_EXIT_CRITICAL();
    OS_Sched();
    return (OS_NO_ERR);
}
if (pevent->OSEventCnt < 65535u) {
    pevent->OSEventCnt++;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
```

```
}
OS_EXIT_CRITICAL();
return (OS_SEM_OVF);
} ```

del 不变

```c OS_EVENT *OSSemDel (OS_EVENT *pevent, INT8U opt, INT8U *err) { BOOLEAN taskswaiting;
OSEVENT *pevent_return;

if OSCRITICALMETHOD == 3
OS_CPU_SR    cpu_sr = 0;
endif
if OSARGCHK_EN > 0
if (err == (INT8U *)0) {
    return (pevent);
}
if (pevent == (OS_EVENT *)0) {
    *err = OS_ERR_PEVENT_NULL;
    return (pevent);
}
endif
if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {
    *err = OS_ERR_EVENT_TYPE;
    return (pevent);
}
if (OSIntNesting > 0) {
    *err = OS_ERR_DEL_ISR;
    return (pevent);
}
OS_ENTER_CRITICAL();
if (pevent->OSEventGrp != 0) {
    tasks_waiting = OS_TRUE;
} else {
    tasks_waiting = OS_FALSE;
}
switch (opt) {
    case OS_DEL_NO_PEND:
        if (tasks_waiting == OS_FALSE) {
if OSEVENTNAME_SIZE > 1
            pevent->OSEventName[0] = '?';
            pevent->OSEventName[1] = OS_ASCII_NUL;
endif
            pevent->OSEventType       = OS_EVENT_TYPE_UNUSED;
```
```

```
            pevent->OSEventPtr         = OSEventFreeList;
            pevent->OSEventCnt         = 0;
            OSEventFreeList            = pevent;
            OS_EXIT_CRITICAL();
            *err                       = OS_NO_ERR;
            pevent_return              = (OS_EVENT *)0;
        } else {
            OS_EXIT_CRITICAL();
            *err                       = OS_ERR_TASK_WAITING;
            pevent_return              = pevent;
        }
        break;

    case OS_DEL_ALWAYS:
        while (pevent->OSEventGrp != 0) {
            (void)OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM);        //第二个参数为 0
        }
if OSEVENTNAME_SIZE > 1
        pevent->OSEventName[0] = '?';
        pevent->OSEventName[1] = OS_ASCII_NUL;
endif
        pevent->OSEventType        = OS_EVENT_TYPE_UNUSED;
        pevent->OSEventPtr         = OSEventFreeList;
        pevent->OSEventCnt         = 0;
        OSEventFreeList            = pevent;
        OS_EXIT_CRITICAL();
        if (tasks_waiting == OS_TRUE) {
            OS_Sched();
        }
        *err                       = OS_NO_ERR;
        pevent_return              = (OS_EVENT *)0;
        break;

    default:
        OS_EXIT_CRITICAL();
        *err                       = OS_ERR_INVALID_OPT;
        pevent_return              = pevent;
        break;
}
return (pevent_return);
} ```
```

Part2：

ucos_ii.h：

```
typedef struct os_tcb {
/*...其余未改动代码...*/
/*新增部分*/
    struct os_tcb *TCBFifoNext;
    struct os_tcb *TCBFifoPrev;
} OS_TCB;
```

os_task.c:

```
INT8U  OSTaskDel (INT8U prio)
/*...其余未改动代码...*/
#if OS_EVENT_EN
    pevent = ptcb->OSTCBEventPtr;
    if (pevent != (OS_EVENT *)0) {     /* If task is waiting on event         */
        pevent->OSEventTbl[y] &= ~ptcb->OSTCBBitX;
        if (pevent->OSEventTbl[y] == 0) {     /* ... remove task from ...            */
            pevent->OSEventGrp &= ~ptcb->OSTCBBitY;/* ... event ctrl block         */
        }
        if (pevent->OSEventType == OS_EVENT_TYPE_SEM) {     //新增：处理新增的指针,逻辑和
EventTO一样
            ptcb->TCBFifoPrev->TCBFifoNext = ptcb->TCBFifoNext;
            ptcb->TCBFifoNext->TCBFifoPrev = ptcb->TCBFifoPrev;
            if (pevent->OSEventPtr == ptcb) {
                if (ptcb->TCBFifoNext == ptcb) {
                    pevent->OSEventPtr = NULL;
                }
                else
                    pevent->OSEventPtr = ptcb->TCBFifoNext;
            }
            ptcb->TCBFifoNext = NULL;
            ptcb->TCBFifoPrev = NULL;
        }
    }
#endif
```

os_core：

```
INT8U  OS_TCBInit (INT8U prio, OS_STK *ptos, OS_STK *pbos, INT16U id, INT32U stk_size, void
*pext, INT16U opt)
{
/*...其余未改动代码...*/
if (ptcb != (OS_TCB *)0) {
    OSTCBFreeList        = ptcb->OSTCBNext;
    OS_EXIT_CRITICAL();
    ptcb->OSTCBStkPtr    = ptos;
    ptcb->OSTCBPrio      = prio;
```

```
    ptcb->OSTCBStat      = OS_STAT_RDY;
    ptcb->OSTCBPendTO    = OS_FALSE;
    ptcb->OSTCBDly       = 0;
    ptcb->TCBFifoNext = NULL;   //新增：初始化新加的指针
    ptcb->TCBFifoPrev = NULL;  //新增：初始化新加的指针
/*...其余未改动代码...*/
}

/*新增函数 */
#if OS_EVENT_EN > 0
void  OS_EventTaskRdy_Fifo (OS_EVENT *pevent, void *msg, INT8U msk)
{
    OS_TCB  *ptcb;
    OS_TCB* head; //链表头（pevent->OSEventPtr）
    OS_TCB* tail; //链表尾（pevent->OSEventPtr->TCBFifoPrev）
    INT8U    x;
    INT8U    y;
#if OS_LOWEST_PRIO <= 63
    INT8U    bitx;
    INT8U    bity;
#else
    INT16U   bitx;
    INT16U   bity;
    INT16U  *ptbl;
#endif
    ptcb = pevent->OSEventPtr;       /*用队列定位当前要置为就绪的任务*/

    pevent->OSEventTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX;                    /* Remove this
task from the waiting list      */
    if (pevent->OSEventTbl[ptcb->OSTCBY] == 0) {
        pevent->OSEventGrp &= ~ptcb->OSTCBBitY;                   /* Clr group bit if this was
only task pending */
    }
    ptcb->OSTCBDly       =  0;                          /* Prevent OSTimeTick() from readying
task     */
    ptcb->OSTCBEventPtr  = (OS_EVENT *)0;               /* Unlink ECB from this task
*/
#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0)
    ptcb->OSTCBMsg       = msg;                         /* Send message directly to waiting
task       */
#else
    msg                  = msg;                         /* Prevent compiler warning if not used
*/
#endif
```

```
    ptcb->OSTCBPendTO    = OS_FALSE;                    /* Cancel 'any' timeout because of post
*/
    ptcb->OSTCBStat     &= ~msk;                        /* Clear bit associated with event type
*/
    if (ptcb->OSTCBStat == OS_STAT_RDY) {               /* See if task is ready (could be
susp'd)      */
        OSRdyGrp        |= ptcb->OSTCBBitY;                     /* Put task in the ready to
run list           */
        OSRdyTbl[ptcb->OSTCBY]     |= ptcb->OSTCBBitX;
    }
    /*维护队列-取出*/
    head = pevent->OSEventPtr;
    if (head != NULL) {
        tail = head->TCBFifoPrev;
        if (head->TCBFifoNext == head) {
            pevent->OSEventPtr = NULL;
        }
        else {
            head->TCBFifoNext->TCBFifoPrev = tail;
            tail->TCBFifoNext = head->TCBFifoNext;
            pevent->OSEventPtr = head->TCBFifoNext;
        }
        head->TCBFifoNext = NULL;
        head->TCBFifoPrev = NULL;
    }
    return;
}
#endif

/*新增函数*/
#if OS_EVENT_EN > 0
void  OS_EventTaskWait_Fifo (OS_EVENT *pevent)
{
    INT8U  y;
    OS_TCB* head; //链表头（pevent->OSEventPtr）
    OS_TCB* tail; //链表尾（pevent->OSEventPtr->TCBFifoPrev）

    OSTCBCur->OSTCBEventPtr = pevent;             /* Store pointer to event control block in
TCB       */
    y                       = OSTCBCur->OSTCBY;   /* Task no longer ready
*/
    OSRdyTbl[y]            &= ~OSTCBCur->OSTCBBitX;
    if (OSRdyTbl[y] == 0) {
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;         /* Clear event grp bit if this was only task
```

```
pending */
    }
    pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;          /* Put task in
waiting list  */
    pevent->OSEventGrp                   |= OSTCBCur->OSTCBBitY;

    /*维护链表-插入*/
    head = pevent->OSEventPtr;
    if (head == NULL) {
        OSTCBCur->TCBFifoPrev = OSTCBCur;
        OSTCBCur->TCBFifoNext = OSTCBCur;
        pevent->OSEventPtr = OSTCBCur;
    }
    else {
        tail = head->TCBFifoPrev;
        OSTCBCur->TCBFifoPrev = tail;
        OSTCBCur->TCBFifoNext = head;
        tail->TCBFifoNext = OSTCBCur;
        head->TCBFifoPrev = OSTCBCur;

    }
}
#endif

/*新增函数*/
#if OS_EVENT_EN > 0
void  OS_EventTO_Fifo (OS_EVENT *pevent)
{
    INT8U  y;


    y                      = OSTCBCur->OSTCBY;
    pevent->OSEventTbl[y] &= ~OSTCBCur->OSTCBBitX;     /* Remove task from wait list
*/
    if (pevent->OSEventTbl[y] == 0x00) {
        pevent->OSEventGrp &= ~OSTCBCur->OSTCBBitY;
    }
    OSTCBCur->OSTCBPendTO   = OS_FALSE;                /* Clear the Pend Timeout flag
*/
    OSTCBCur->OSTCBStat     = OS_STAT_RDY;             /* Set status to ready
*/
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;           /* No longer waiting for event
*/
    /*把当前任务从链表上拆下来*/
```

```
    OSTCBCur->TCBFifoPrev->TCBFifoNext = OSTCBCur->TCBFifoNext;
    OSTCBCur->TCBFifoNext->TCBFifoPrev = OSTCBCur->TCBFifoPrev;
    if(pevent->OSEventPtr == OSTCBCur){
        if(OSTCBCur->TCBFifoNext == OSTCBCur){
            pevent->OSEventPtr = NULL;
        }
        else
            pevent->OSEventPtr = OSTCBCur->TCBFifoNext;
    }
    OSTCBCur->TCBFifoNext = NULL;
    OSTCBCur->TCBFifoPrev = NULL;
}
#endif
```

os_sem:

```
OS_EVENT  *OSSemCreate (INT16U cnt)
{
/*...其余未改动代码...*/
pevent->OSEventPtr     = NULL;      /* 初始化 OSEventPtr，当前没有等待该信号量的任务*/
/*...其余未改动代码...*/
}

OS_EVENT  *OSSemDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
{
/*...其余未改动代码...*/
if (pevent->OSEventPtr != NULL) {/* 修改：通过OSEventPtr判断是否有任务等待信号量     */
    tasks_waiting = OS_TRUE;
} else {
    tasks_waiting = OS_FALSE;
}
/*...其余未改动代码...*/
case OS_DEL_ALWAYS:                                /* Always delete the semaphore
*/
      while (pevent->OSEventPtr != NULL) {/*通过OSEventPtr判断是否有任务等待信号量   */
         OS_EventTaskRdy_Fifo(pevent, (void *)0, OS_STAT_SEM);   // 改为调用新写的函数
      }
/*...其余未改动代码...*/
}

void  OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
/*...其余未改动代码...*/
OS_EventTaskWait_Fifo(pevent);                          /* 改为调用新增的函数     */
/*...其余未改动代码...*/
```

```
if (OSTCBCur->OSTCBPendTO == OS_TRUE) {  /* See if we timedout                    */
    OS_EventTO_Fifo(pevent);                        // 改为调用新增函数
    OS_EXIT_CRITICAL();
    *err = OS_TIMEOUT;           /* Indicate that didn't get event within TO      */
    return;
}
}

INT8U  OSSemPost (OS_EVENT *pevent)
{
/*...其余未改动代码...*/
if (pevent->OSEventPtr != NULL) {                              /* 改用队列判断 */
     OS_EventTaskRdy_Fifo(pevent, (void *)0, OS_STAT_SEM); /* 改用修改后的函数    */
     OS_EXIT_CRITICAL();
     OS_Sched();                      /* Find HPT ready to run                */
     return (OS_NO_ERR);
 }
/*...其余未改动代码...*/
}
```

测试用应用程序代码 test.c：

（由于将 ucosii 移植到 windows 上运行，所以部分内容可能与直接在 ucosii 上运行不符）

```
int  main(void)
{
    PC_DispClrScr(DISP_FGND_WHITE + DISP_BGND_BLACK);
    OSInit();
    MySem = OSSemCreate(0);                          //创建信号量
    OSTaskCreate(TaskStart, (void*)0, &TaskStartStk[TASK_STK_SIZE - 1], 0);
    OSStart();    /* Start multitasking                          */
    return 0;
}
static  void  TaskStartCreateTasks(void)
{
    OSTaskCreate(Task1, (void*)0, &TaskStk[0][TASK_STK_SIZE - 1], 7);
    OSTimeDlyHMSM(0, 0, 0, 10);
    OSTaskCreate(Task2, (void*)0, &TaskStk[1][TASK_STK_SIZE - 1], 6);
    OSTimeDlyHMSM(0, 0, 0, 10);
    OSTaskCreate(Task3, (void*)0, &TaskStk[2][TASK_STK_SIZE - 1], 5);
    OSTaskCreate(Task4, (void*)0, &TaskStk[3][TASK_STK_SIZE - 1], 4);
}
void  Task1(void* pdata)
{
    INT8U  err;
    int i = 0;
```

```
#ifdef __WIN32__
    srand(GetCurrentThreadId());
#endif
#ifdef __LINUX__
    srand(getppid());
#endif
    for (;;) {
        OSSemPend(MySem, 0, &err);
        if (err == OS_NO_ERR) {
            PC_DispStr(0, 8+i, "Task1 got semaphore\n", DISP_FGND_YELLOW + DISP_BGND_BLUE);
        }
        //printf("Task1 got semaphore at %lu ticks\r\n", OSTime);
        i = i + 3;
        OSTimeDlyHMSM(0, 0, 1, 0);
    }
}
void  Task2(void* pdata)
{
    INT8U  err;
    int i = 0;

#ifdef __WIN32__
    srand(GetCurrentThreadId());
#endif
#ifdef __LINUX__
    srand(getppid());
#endif
    for (;;) {
        OSSemPend(MySem, 0, &err);
        if (err == OS_NO_ERR) {
            PC_DispStr(0, 9+i, "Task2 got semaphore\n", DISP_FGND_YELLOW + DISP_BGND_BLUE);
        }
        i = i + 3;
        OSTimeDlyHMSM(0, 0, 1, 0);
    }
}
void  Task3(void* pdata)
{
    INT8U  err;
    int i = 0;

#ifdef __WIN32__
    srand(GetCurrentThreadId());
```

```
#endif
#ifdef __LINUX__
    srand(getppid());
#endif
    for (;;) {
        OSSemPend(MySem, 0, &err);
        if (err == OS_NO_ERR) {
            PC_DispStr(0, 10+i, "Task3 got semaphore\n", DISP_FGND_YELLOW + DISP_BGND_BLUE);
        }
        i = i + 3;
        OSTimeDlyHMSM(0, 0, 1, 0);
    }
}
void Task4(void* pdata) {
    for (;;) {
        OSSemPost(MySem);
        OSTimeDlyHMSM(0, 0, 3, 0);
    }
}
```

4 结果展示

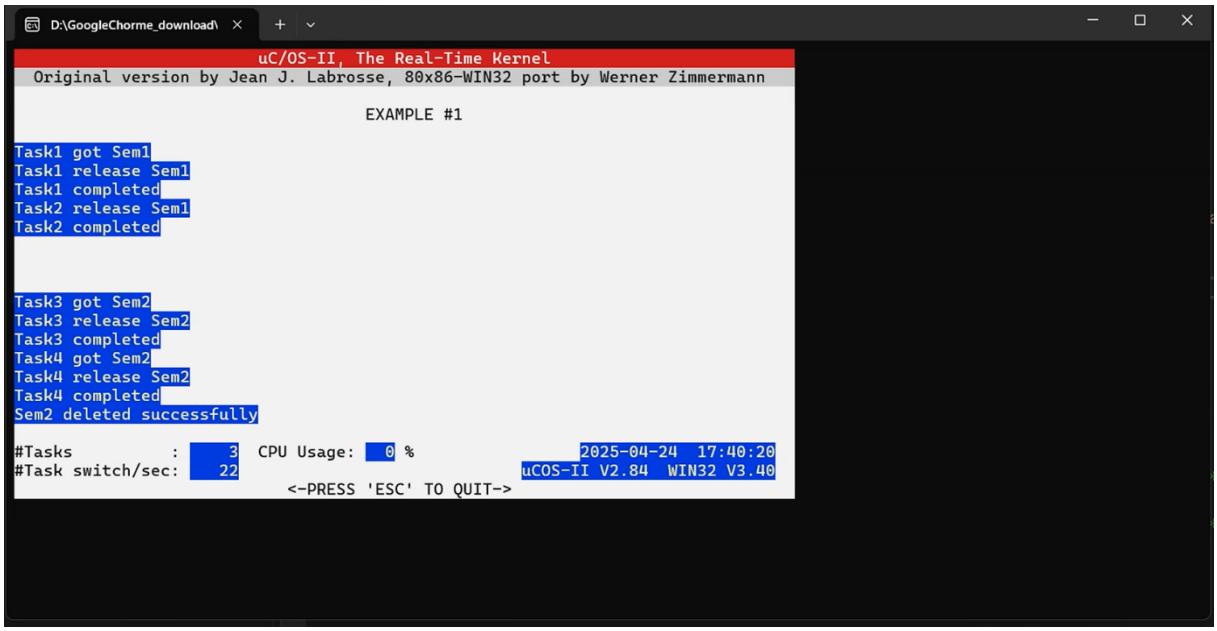
Part1：

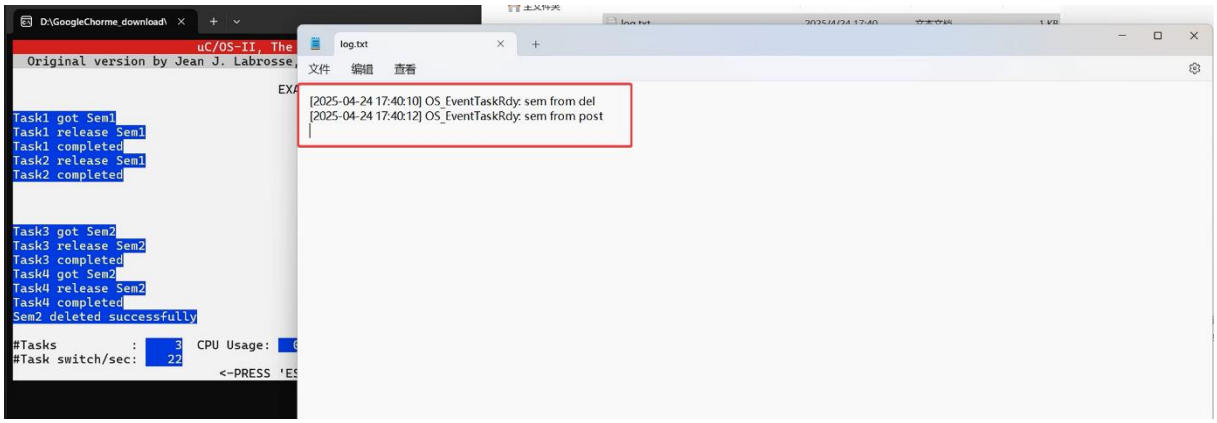
再加一些简单的 if 语句对传入的参数进行判断，LogMessage 函数用于测试：将判断结果保存在文本文件中

```
700         // 添加信号量事件的消息判断
701         if (msk == OS_STAT_SEM) {
702             if ((INT32U)msg == 0) {
703                 //printf("sem from del\n");
704                 LogMessage("OS_EventTaskRdy: sem from del");
705                 //PC_DispStr(0, 10, "Task2 release Sem2\n", DISP_FGND_YELLOW + DISP_BGND_BLUE);
706             }
707             else if ((INT32U)msg == 1) {
708                 //printf("sem from post\n");
709                 LogMessage("OS_EventTaskRdy: sem from post");
710                 //PC_DispStr(0, 10, "Task2 release Sem2\n", DISP_FGND_YELLOW + DISP_BGND_BLUE);
711             }
712         }
713
```

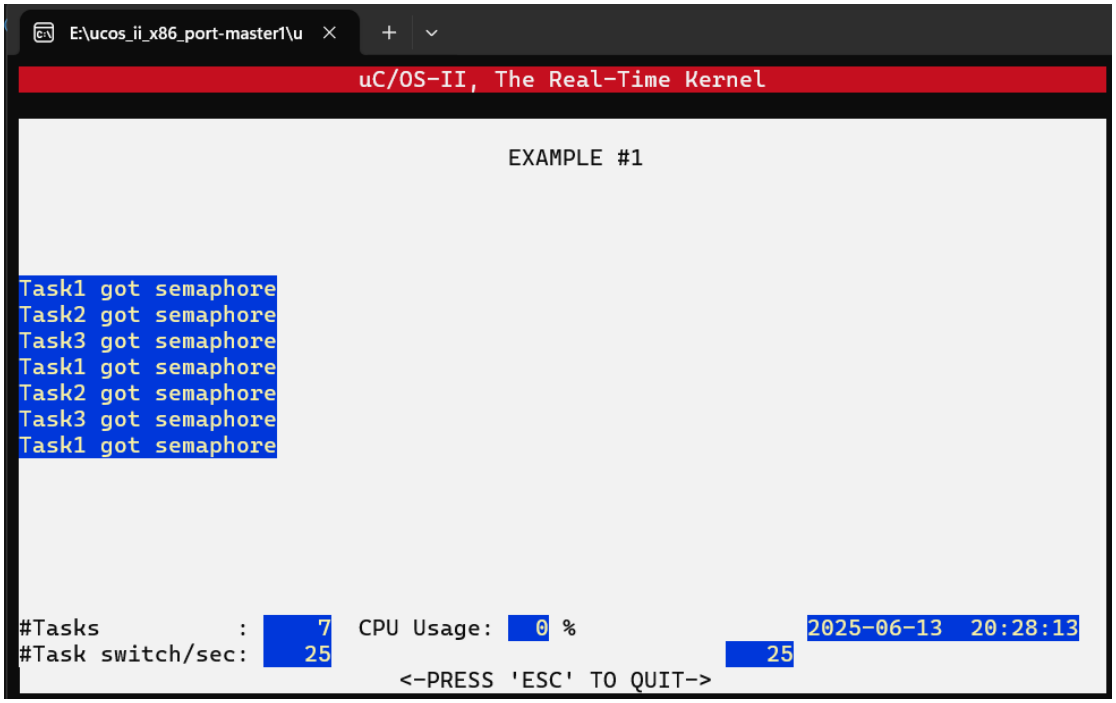
```
1659
1660
1661
1662  /*
1663  ********************************************************************************************************
1664  *                                          用户添加(lyc) START
1665  ********************************************************************************************************
1666  */
1667  // 添加日志函数
1668  void LogMessage(const char* message)
1669  {
1670      static FILE* logFile = NULL;
1671      time_t now;
1672      struct tm* timeinfo;
1673      char timeString[30];
1674
1675      // 获取当前时间
1676      time(&now);
1677      timeinfo = localtime(&now);
1678      strftime(timeString, sizeof(timeString), "%Y-%m-%d %H:%M:%S", timeinfo);
1679
1680      if (logFile == NULL) {
1681          logFile = fopen("C:\\Users\\lyc\\Desktop\\3\\3.2\\emOS\\ucosLog\\log.txt", "a");
1682          if (logFile == NULL) {
1683              return;
1684          }
1685      }
1686
1687      fprintf(logFile, "[%s] %s\n", timeString, message);
1688      fflush(logFile);
1689  }
1690
1691  /*
1692  ********************************************************************************************************
1693  *                                          用户添加(lyc) END
1694  ********************************************************************************************************
1695  */
```

结果展示





Part2：



task1 2 3 的优先级分别是 7 6 5，而申请信号量的顺序是 task1 task2 task3。从显示结果可以看出，获得信号量的顺序是 task1 task2 task3，任务成功按照先进先出的顺序获取信号量。