实践视角下操作系统中若干概念或原理的探讨

韩明峰

烟台理工学院

2022. 11. 27

目的

- 教材是学生学习的主要依据，是培养学生自学能力的主要载体。
- 在担任我国经典教材《计算机网络(第八版)》(谢希仁)主审的过程中，深深地认识到教材进步的复杂性和艰巨性。
  - 谢希仁教授是全国和全军优秀教师，解放军“人梯奖”获得者。
  - 自2016年以来，双方往来近400封邮件探讨教学内容与教材改进等问题。
- 本人也希望为国内开源《操作系统》教材的建设贡献一点绵薄之力。

目录

OS实践与OS教材

- 概念和原理是OS教材的主要组成部分之一，从实践中提炼概念和原理是OS教材的主要任务之一，但这并不是要求通用操作系统的原理详尽到实例的程度。
- 一些国外著名操作系统教材，如William Stallings的《操作系统：精髓与设计原理(第9版)》，用专门的章节来介绍实时嵌入式操作系统，而一些实时内核的官方文档也蕴含着丰富的设计思想。

关于MiCriμm

- MiCriμm((从软件的眼中看)微处理器世界)公司是全球RTOS的领导者，2016年被Silicon Labs收购。
- 其uc/OS-II通过了美国联邦航空管理局(FAA)的认证。
- 工业调查一致显示uc/OS-III是嵌入式领域中最为流行的操作系统之一。
- MiCriμm公司总裁JeanJ.Labrosse所著的《uc/OS-III:The Real-Time Kernel》深刻地揭示了内核的设计原理。

目录

原语和系统调用

- 所谓“原语”一般是指由若干条指令所组成的程序段，用来实现某个特定功能，在执行过程中不可被中断，或者中断但不引起race condition。
- 原语和系统调用是两个不同的概念。
  - 原语主要强调操作的不可分割性，可以认为是一个不可中断的子程序调用。
  - 操作系统中有些系统调用是以原语的形式出现的，但并不是所有系统调用都是原语。因为如果那样的话，整个系统的并发性就不可能得到充分的发挥。

uc/OS-II (V2.91) 中的P操作


```
void  OSSemPend (OS_EVENT  *pevent, INT32U  timeout, INT8U  *perr)
{
    OS_ENTER_CRITICAL( );
    if (pevent->OSEventCnt > 0u) {    /* If sem. is positive,
                                         resource available ... */
        pevent->OSEventCnt--;         /* ... decrement   semaphore
                                         only if positive. */
        OS_EXIT_CRITICAL( );
        *perr = OS_ERR_NONE;
        return;
    }
```

uc/OS-II (V2.91) 中的P操作 (续1)


```
                                /* Otherwise, must wait until event occurs */
OSTCBCur->OSTCBStat     |= OS_STAT_SEM;      /* Resource not
                                                available, pend on semaphore */
OSTCBCur->OSTCBStatPend  = OS_STAT_PEND_OK;
OSTCBCur->OSTCBDly       = timeout;    /* Store pend timeout in TCB  */
OS_EventTaskWait(pevent);                    /* Suspend task until event or
                                                timeout occurs */
OS_EXIT_CRITICAL( );
OS_Sched( );                          /* Find next highest priority task ready */
OS_ENTER_CRITICAL( );
switch (OSTCBCur->OSTCBStatPend) { /* See if we timed-out or aborted*/
    case OS_STAT_PEND_OK:
        *perr = OS_ERR_NONE;
        break;
```

uc/OS-II (V2.91) 中的P操作 (续2)


```
        case OS_STAT_PEND_ABORT:
            *perr = OS_ERR_PEND_ABORT;    /* Indicate that we aborted */
            break;
        case OS_STAT_PEND_TO:
        default:
            OS_EventTaskRemove(OSTCBCur, pevent);
            *perr = OS_ERR_TIMEOUT;       /* Indicate that we didn't get event
                                             within TO */
            break;
    }
    OSTCBCur->OSTCBStat =  OS_STAT_RDY;   /* Set   task  status to ready */
    OSTCBCur->OSTCBStatPend= OS_STAT_PEND_OK;  /* Clear pend  status*/
    OSTCBCur->OSTCBEventPtr = (OS_EVENT  *)0;   /* Clear event pointers */
    OS_EXIT_CRITICAL( );
}
```

uc/OS-II (V2.91)中的V操作


```
INT8U  OSSemPost (OS_EVENT *pevent)
{
    OS_ENTER_CRITICAL( );
    if (pevent->OSEventGrp != 0u) { /* See if any task waiting for semaphore*/
                                    /* Ready HPT waiting on event */
        (void)OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM,
                              OS_STAT_PEND_OK);
        OS_EXIT_CRITICAL( );
        OS_Sched( );                /* Find HPT ready to run */
        return (OS_ERR_NONE);
    }
```

uc/OS-II (V2.91) 中的V操作(续)


```
    if (pevent->OSEventCnt < 65535u) {  /* Make sure semaphore will not
                                           overflow */
        pevent->OSEventCnt++;           /* Increment semaphore count to
                                           register event */
        OS_EXIT_CRITICAL( );
        return (OS_ERR_NONE);
    }
    OS_EXIT_CRITICAL( );     /* Semaphore value has reached its maximum */
    return (OS_ERR_SEM_OVF);
}
```

思考1

- 以上P/V操作的主要实现代码表明P/V操作中存在着多个原语，因此，不能把P/V操作甚至其它一些系统调用轻易说成是一个原语。
- William Stallings的《操作系统：精髓与设计原理(第9版)》的相关表述：
  - “要通过信号量s传送信号，进程须执行原语semSignal(s)；”
  - “要通过信号量s接收信号，进程须执行原语semWait(s)。”
- Abraham Silberschatz等的《操作系统概念(第9版)》的相关表述：
  - “对同一信号量，没有两个进程可以同时执行wait( )和signal( )操作。”

目录

同步中的瞬时事件和状态事件

- 同步中的瞬时事件举例：
  - 高铁服务员们关门后司机才能启动高铁运行，高铁停止后服务员们才能开门；
  - 开关被按下；
  - 运动传感器探测到一个目标；
  - 发生爆炸等。
- 获取瞬时事件的任务要消耗掉该事件。
- 同步中的状态事件举例：
  - 温度超出阈值；
  - 引擎转速为零；
  - 电池低电压；
  - 燃料箱中燃料不足等。
- 状态事件不应该被等待这些事件的任务消耗掉，因为这些状态信息是由其它任务或ISR来管理的，如：
  - 电池监控任务报告电池电压正常。

信号量能否解决状态事件的问题

- 信号量用一个整型变量来累计唤醒次数。具体使用时，
  - 信号量可以表示资源数目，用于解决资源同步问题；
  - 信号量也可以表示事件的发生或事件发生的次数，用于解决活动(行为)同步问题。
- P操作中信号量计数值的递减可用于消耗瞬时事件。
- 然而，P/V操作没有支持状态事件的功能。

Windows中的自动/人工复位事件

- Windows中利用API CreateEvent( )来创建一个事件。

```
HANDLE    CreateEvent(
        LPSECURITY_ATTRIBUTES   lpEventAttributes,// 安全属性
        BOOL     bManualReset,   // 复位方式
        BOOL    bInitialState,   // 初始状态
        LPCTSTR          lpName); // 对象名称
```

- bManualReset为TRUE表示通过API ResetEvent( )来人工复位事件，V操作SetEvent( )唤醒所有等待线程；
- bManualReset为FALSE表示通过P操作WaitForSingleObject( )获取到事件后会自动复位事件，V操作SetEvent( )唤醒一个等待线程。

uc/OS-II中的消耗/非消耗事件

- 任务使用系统函数OSFlagPend( )来请求一个事件组，其函数原型：

```
OS_FLAGS     OSFlagPend(
                OS_FLAG_GRP   *pgrp,        //请求的事件组指针
                OS_FLAGS      flags,        //等待的位
                INT8U     wait_type,        //等待的类型
                INT16U      timeout,        //等待时限
                INT8U         *err);        //错误信息
```

- 如果在OSFlagPend( )中使用OS_FLAG_CONSUME系统常数，获取后的事件值将被消耗(清除)。
- 使用方式：wait_type + OS_FLAG_CONSUME

同步中的条件

- 信号量、事件组和条件变量的条件表达能力依次增强。
- 一般的信号量机制不太容易处理多个事件源。
  - 实现多个事件逻辑“与”的效果一般需要顺序调用多个P操作。
  - 不太容易实现多个事件逻辑“或”的效果。
  - 信号量集主要解决多类资源管理的问题。
- 而条件变量又是在用户态由应用进行条件的判断。
- 事件组机制使任务可以按多个事件的逻辑“与”或者逻辑“或”进行等待，如：
  - 如果两个键都按下，则点亮LED灯。
  - 如果两个键中有一个按下，则点亮LED灯。

Windows中的多事件等待

- Windows的API WaitForMultipleObjects( )可以实现多事件等待。

```
DWORD WaitForMultipleObjects(
DWORD nCount,   // number of handles in the handle array
CONST HANDLE *lpHandles,// pointer to the object-handle
                        // array
BOOL fWaitAll,          // wait flag
DWORD dwMilliseconds); // time-out interval in milliseconds
```

- fWaitAll为TRUE，表示逻辑“与”；
- fWaitAll为FALSE，表示逻辑“或”。

uc/OS-II中的多事件等待

- uc/OS-II的OSFlagPend( )可以实现多事件等待。

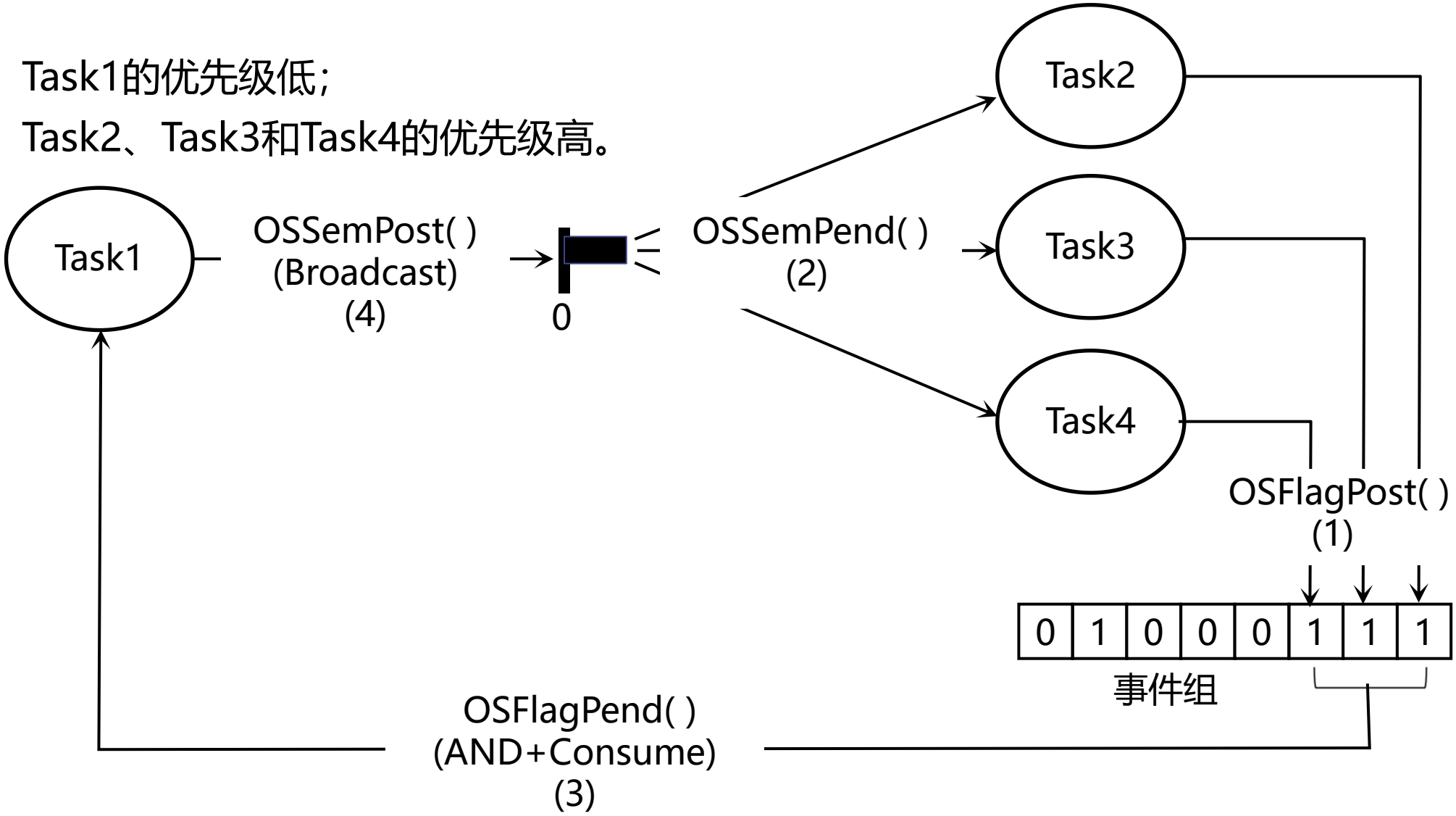
```
OS_FLAGS   OSFlagPend(OS_FLAG_GRP   *pgrp,//请求的事件组指针
                      OS_FLAGS      flags,   //等待的位
                      INT8U     wait_type,   //等待的类型
                      INT16U       timeout,  //等待时限
                      INT8U           *err);  //错误信息
```

wait_type	描述
OS_FLAG_WAIT_SET_ALL(AND)	全1
OS_FLAG_WAIT_SET_ANY(OR)	有1
OS_FLAG_WAIT_CLR_ALL(AND)	全0
OS_FLAG_WAIT_CLR_ANY(OR)	有0

事件组的应用(1)

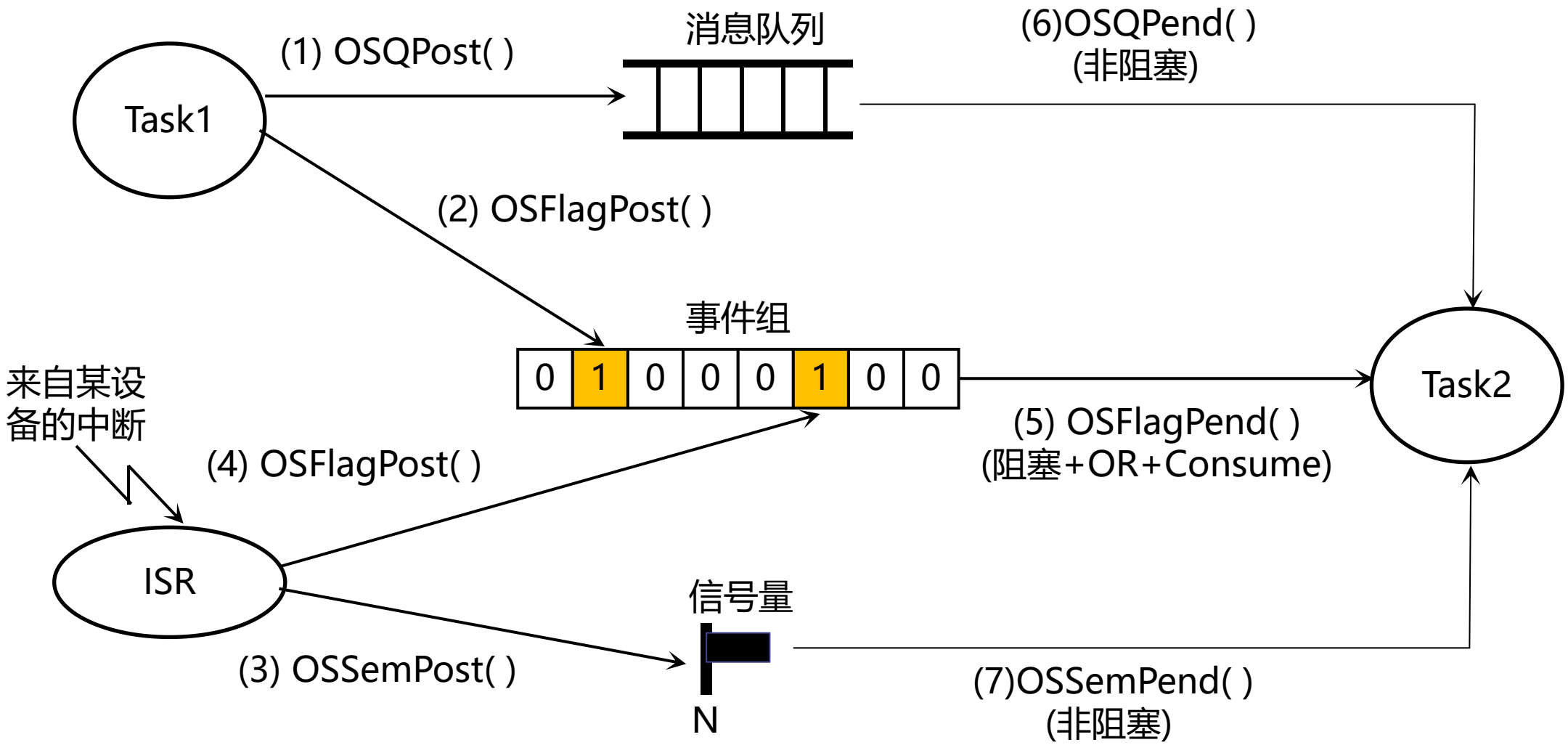
Task1的优先级低；

Task2、Task3和Task4的优先级高。



利用事件组和二值信号量使多个任务同时开始执行

事件组的应用(2)



利用事件组同时等待多个内核对象

思考2

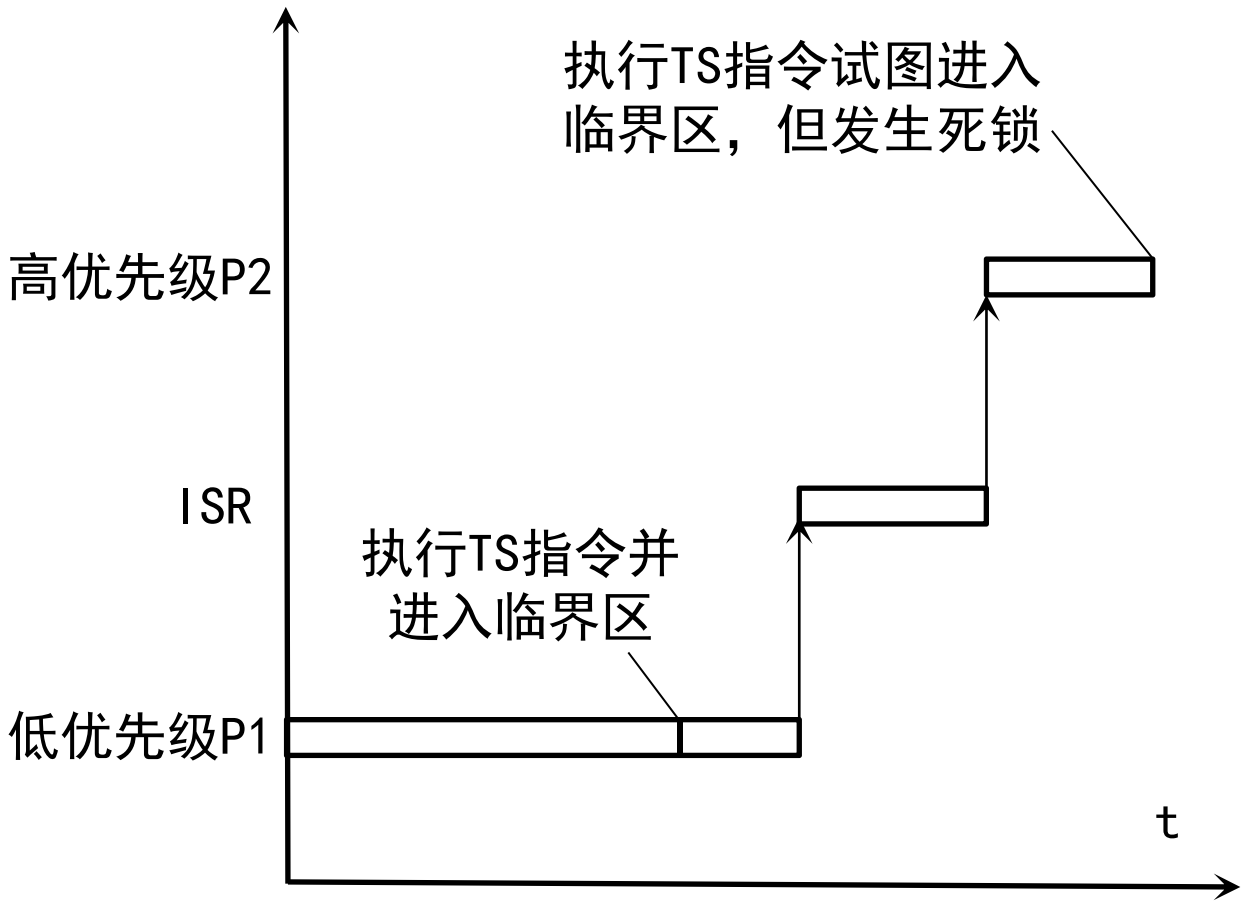
- 一些国外著名操作系统教材在原理中均无事件组机制的详细表述。
  - William Stallings的《操作系统：精髓与设计原理(第9版)》只是在原理中简单提及事件标志。
- 为解决活动(行为)同步中的状态事件以及逻辑“与”或者逻辑“或”的相关问题，建议在通用OS原理中增加事件组机制。

目录

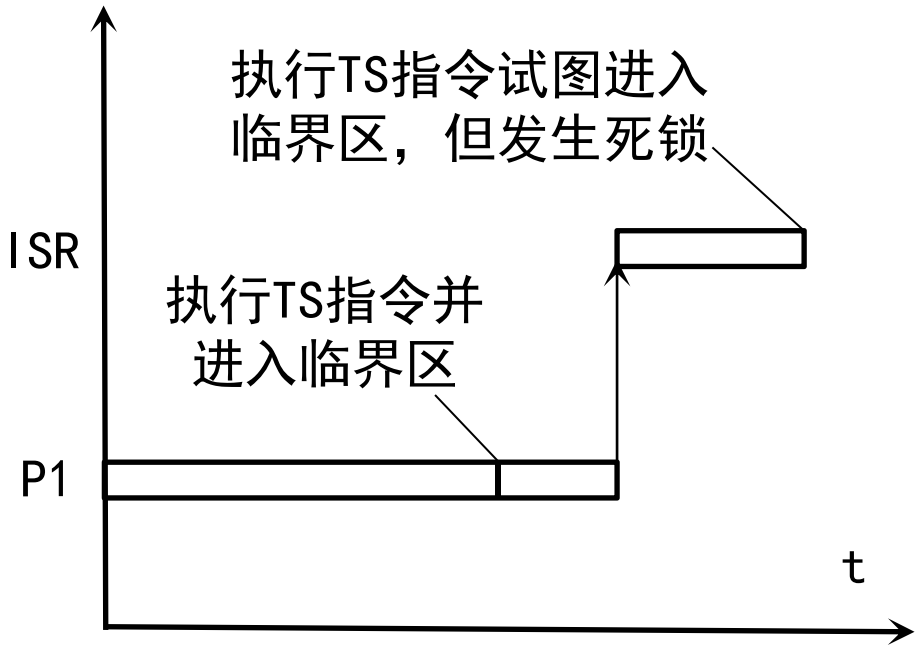
UP下利用硬件指令忙等待实现互斥的分析

- 硬件的test-and-set、compare-and-swap和exchange等指令可用于实现互斥。
- 在基于时间片轮转的调度算法下，假设N个任务竞争同一个临界资源，当一个任务在临界区时，最坏的情况下其它(N-1)个任务共浪费(N-1)个时间片，临界区中的任务才能得以调度运行，从而离开临界区，代价很高。
- 在基于优先级的抢占式调度算法下，不同优先级的任务竞争同一个临界资源，存在着死锁(更准确地说是livelock，活锁)的可能。
- 如果一个ISR试图申请一个线程拥有的自旋锁，也会发生死锁。

UP下使用自旋锁引起的死锁



任务间的死锁示意图



任务与ISR间的死锁示意图

UP下能否使用忙等待的同步

- 基于以上分析，即使在UP下提供了硬件的test-and-set、compare-and-swap和exchange等指令，基于OS的编程也不宜采用。
- 类似地，OS实例在UP下一般也不会提供忙等待的软件同步工具，如：
  - Linux和VxWorks的自旋锁在UP下都退化为禁止本地CPU任务抢占或禁止本地CPU中断。

思考3

【2021年科目408真题第45题】下表给出了整型信号量S的wait( )和signal( )操作的功能描述，以及采用开/关中断指令实现信号量操作互斥的两种方法。

请回答下列问题。

(1)为什么在wait( )和signal( )操作中对信号量S的访问必须互斥执行?

(2)分别说明方法1和方法2是否正确。若不正确，请说明理由。

(3)用户程序能否使用开/关中断指令实现临界区互斥？为什么?

功能描述

```
Semaphore S;
wait(S){
  while(S<=0)
     ;
  S=S-1;
}

signal(S){
  S=S+1;
}
```

方法1

```
Semaphore S;
wait(S){
   关中断;
   while(S<=0)
      ;
   S=S-1;
   开中断;
}

signal(S){
   关中断;
   S=S+1;
   开中断;
}
```

方法2

```
Semaphore S;
wait(S){
   关中断;
   while(S<=0){
      开中断;
      关中断;
   }
   S=S-1;
   开中断;
}
signal(S){
   关中断;
   S=S+1;
   开中断;
}
```

两种实现方法的正确性与可行性分析

- 本题针对的是UP环境。
- 方法1存在着while死循环的情况，不正确。
- 方法2虽然解决了方法1的问题，但所实现的信号量在二进制用于互斥的情况下，存在着前述的代价很高以及可能死锁的问题。
- 在2021年的科目408真题题解中方法2被认为是正确的。

SMP下的自旋锁

- SMP下一般要考虑CPU的亲和性，大多数SMP系统每个CPU都有自己的私有就绪队列，前述UP下使用自旋锁的问题在本地CPU也可能发生，因此需要采取一些方法来解决上述问题。
- SMP下线程持有自旋锁期间应禁止本地CPU任务抢占，如：
  - Linux的spin_lock( )和spin_lock_irqsave( )自动调用preempt_disable( );
  - VxWorks的spinLockTaskTake( )和spinLockIsrTake( )禁止本地CPU任务抢占。
- 如果涉及到中断上下文申请自旋锁，还需要和禁止本地CPU中断联合使用，如：
  - Linux的spin_lock_irqsave( )调用local_irq_save( );
  - VxWorks的spinLockIsrTake( )禁止本地CPU中断。

思考4

- 一些国外著名操作系统教材在利用test-and-set、compare-and-swap和exchange等指令实现SMP下的信号量时，都没有考虑持有自旋锁期间本地CPU发生中断时可能带来的问题，如：
  - William Stallings的《操作系统：精髓与设计原理(第9版)》；
  - Abraham Silberschatz的《操作系统概念(第9版)》；
  - Andrew S.Tanenbaum的《现代操作系统(第4版)》。

```
semWait(s)
{
    while(compare_and swap(s.flag,0,1)
            ==1)
        /*不做任何事*/;
    s.count--;
    if(s.count<0){
        /*该进程进入s.queue队列*/;
        /*阻塞该进程(还须将s.flag设置为0)*/;
    }
    s.flag=0;
}
```

```
semSignal(s)
{
    while(compare_and swap(s.flag,0,1)
            ==1)
        /*不做任何事*/;
    s.count++;
    if(s.count<=0){
        /*从s.queue队列中移出进程P*/;
        /*进程P进入就绪队列*/;
    }
    s.flag=0;
}
```

SMP下信号量的实现。引自William Stallings的《操作系统：精髓与设计原理(第9版)》

感谢观看
欢迎指正

Thank You