ucos学习篇之信号量

一、相关背景知识

“信号量”为操作系统用于处理临界区问题和实现进程间同步提供了一种有效的机制。在很多操作系统原理书中都提到了信号量的概念，常用Ｐ操作与Ｖ操作来表明信号量的行为。ＰＶ操作的伪代码如下：

设s为一整数型变量:

```
Ｐ操作：while( s==0); s--;
V操作：s++
```

设任务A有如下代码：

```
P( s )
      //对s 代表的资源进行操作的代码
      .......
V( s )
```

如果任务Ａ进行Ｐ操作，则任务Ａ轮循s直至s==0,再将s--.当进行Ｖ操作时，对s++.可以将信号量理解为一种资源，初始资源数s=Ｎ。任务对信号量进行Ｐ操作，即申请占有一个资源量。如果资源量s=0,则任务等待；而一旦资源量s!=0,则将资源量s--,意味着任务申请到资源，可以执行后续操作。任务对信号量的Ｖ操作，即释放所占有资源，资源量s++。这里的资源是可重用资源，即任务使用资源后可将资源放回，供其它任务使用。

特别的，当初始资源数s = 1时，该资源为互斥型，即一次只允许一个任务使用。

关于信号量具体的概念，在任何一本操作系统原理书上都有介绍。这里只指出。对于信号量的理解，只看操作系统书上介绍的那点概念是不够的。即使能够在像windows等ＯＳ下使用ＡＰＩ编写代码，也仅仅是利用接口编程。对于想了解底层，了解信号量的实现，只有通过阅读实现代码才能够理解。

二、ucos对信号量的支持

１、代码结构整体分析：

ucos对信号量的支持由os_sem.c， os_core.c支持。其中os_core.c提供OS_EVENT数据结构的一些基本操作，os_sem.c则实现具体的信号量。信号量实现的分析，主要是**数据结构**的问题。

1)、OS_EVENT结构的实现分析

```
typedef struct {
    INT8U    OSEventType;           // 事件控制块的类型
    INT8U    OSEventGrp;            // 等待的任务组
    INT16U   OSEventCnt;            // 信号量计数
    void    *OSEventPtr;            // 此处用作链表的链接指针
    INT8U    OSEventTbl[OS_EVENT_TBL_SIZE];      // 等待任务表
} OS_EVENT; ( ucos_II.H )
```

OS_EVENT结构体中，信号量主要涉及的域为.OSEventCnt（信号量值）,OSEventGrp,OSEventTbl。其中.OSEventCnt用于计数，OSEventGrp,.OSEventTbl用于支持任务的挂起。参照前面的分析，.OSEventCnt相当于s.

由该结构定义了以下变量用于支持多个信号量的连接。

```
OS_EXT  OS_EVENT         *OSEventFreeList;
OS_EXT  OS_EVENT          OSEventTbl[OS_MAX_EVENTS];
```

OSEventTbl构成由OSEventFreeList为头指针，以.OSEventPtr为链接指针的**单链表**。对于

OS_EVENT项的获取和回收都是基于该链表.由链表由OSStart()调用OS_InitEventList()构造.

对OS_EVENT的基本操作有:(os_core.c)
static void OS_InitEventList (void);
功能：初始化信号量列表，将OSEventTbl中的各项构成以OSEventFreeList为头指针的单链表，以进行各项的插入与删除操作。

void OS_EventWaitListInit (OS_EVENT *pevent);
功能：仅初始化OS_EVENT中的.OSEventGrp, .OSEventTbl。初始化等待的任务列表为空。这里的等待列表和任务就绪表的结构和功能类似。

INT8U OS_EventTaskRdy (OS_EVENT *pevent, void *msg, INT8U msk);
功能：当事件发生时将最高优先级任务置为就绪。即从OS_EVENT中的等待任务列表中取一最高优先级任务，可能将任务插入就绪列表中。

void OS_EventTaskWait (OS_EVENT *pevent);
功能：将任务挂起，将任务挂起，插入到OS_EVENT中等待任务列表中。

void OS_EventTO (OS_EVENT *pevent);
功能：将任务置为将就绪，但原因是超时。

具体的代码分析可自行查看源文件和后面列出的参考书。
分析这些代码，会发现，代码的实现还是比较简单的，主要涉及到一些任务的挂起和置为就绪的操作。此时,可能有这样的问题:为什么为要单独列出OS_EVENT这样的结构？实际上，不仅是信号量，邮箱等模块也用到了OS_EVENT结构。上面的操作是信号量与邮箱**共享的一组操作。需要注意的是，代码涉及到了一些任务管理相关的操作，如任务的调度，挂起和就绪。**

2)、基于OS_EVENT的信号量实现

在os_sem.C中，实现了完整信号量操作:

OS_EVENT *OSSemCreate (INT16U cnt);
功能：创建一个信号量。从OSEventFreeList中取出一空闲OS_EVENT，进行必要的初始化。对信号量而言，主要是初始化.OSEventCnt域和调用**OS_EventWaitListInit**()对初始等待任务列表清空。

OS_EVENT *OSSemDel (OS_EVENT *pevent, INT8U opt, INT8U *err);
功能：删除一个信号量。回收OS_EVENT，如果OS_EVENT有等待的任务，则调用**OS_EventTaskRdy ()** 将等待的任务列表中任务置为就绪态。

void OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err);
功能：申请一个信号量。任务在无法获得信号量时会调用**OS_EventTaskWait()**挂起，再调用**OSSched()**进行任务切换。在超时返回时,调用**OS_EventTo ()** 将任务置为就绪.

INT8U OSSemPost (OS_EVENT *pevent);
功能：发送一个信号量。如果有任务等待，调用**OS_EventTaskRdy**()将任务置为就绪，调度。

INT8U OSSemQuery (OS_EVENT *pevent, OS_SEM_DATA *pdata);
功能：查询信号量。可以直接通过pevent直接获取信号量的信息。但是pevent很多时候是被多个任务共享的。会随时发生改变。使用额外的pdata可以快速的获取当前信号量的副本。

INT16U OSSemAccept (OS_EVENT *pevent);
功能：获取信号量。如果无法获取，立即退出。

参考上面的说明，可以明确信号量的实现与OS_EVENT结构操作函数的调用关系。一旦理解了这种调用关系，会发现信号量的实现其实是很容量理解的。需要指出的是，在信号量的实现中，对.OSEventCnt的操作，是通过短暂的开关中断实现。在关中断期间，可以对OS_EVENT结构进行操作，而不用担心被中断。换句话说，通过短暂的开关中断，实现了内核内部的临界区操作。

个人的体会，信号量相关的代码不大，分析时首先明确涉及到的数据结构及提供的操作接口，明确相互之间的联系后，再了解操作的具体实现。这样能够先有个总体的理解，分析细节时会容易的多。不至于因陷入细节而“只见树木，不见森林”。另外，在分析具体操作时，可以先针对接口，自己考虑如何实现，再与实际的代码对比。实际中会发现具体的代码实现和自己的想法有很多相似。

三、信号量的应用

以下将通过两个例子，实现使用信号量实现资源共享和任务间的同步.

1 、简单的交通灯控制 - 信号量进行任务同步的演示。

这里交通灯由 6 个简单的led构成。由两个任务分别控制红、绿、黄三个ＬＥＤ。
task1: 绿--黄－红－黄－绿-....
task0: 红－黄--绿--黄--红-....

```
void   task1( void * pdata )
{
       INT8U error;
       while(1)
       {
              OSSemPend( sem, 0, &error );       / /   申请信号量。只有在task0发送信号量后才可以继续操作
              LED1_ON( GREEN );                 // 绿
              LED1_OFF( YELLOW );

              OSSemPend( sem, 0, &error );
              LED1_ON( YELLOW );                 / / 黄
              LED1_OFF( GREEN );

              OSSemPend( sem, 0, &error );
              LED1_ON( RED );                   // 红
              LED1_OFF( YELLOW );

              OSSemPend( sem, 0, &error );
              LED1_ON( YELLOW );                // 黄
              LED1_OFF( RED );
```

```
        }
}

void    task0( void * pdata )
{
        INT8U  error;
        t2init();
        while(1)
        {
                LED0_ON( RED );                     // 红
                LED0_OFF( YELLOW );
                OSSemPost( sem );                   // 发送信号量，通知task1可以继续执
行代码
                OSTimeDlyHMSM( 0, 0, DELAY0, 0 );

                LED0_ON( YELLOW );                  // 黄
                LED0_OFF( RED );
                OSSemPost( sem );
                OSTimeDlyHMSM( 0, 0, DELAY1, 0 );

                LED0_ON( GREEN );                   // 绿
                LED0_OFF( YELLOW );
                OSSemPost( sem );
                OSTimeDlyHMSM( 0, 0, DELAY2, 0 );

                LED0_ON( YELLOW );                  // 黄
                LED0_OFF( GREEN );
                OSSemPost( sem );
                OSTimeDlyHMSM( 0, 0, DELAY3, 0 );
        }
}

int     main()
{
        OSInit();
        LED0_INIT();
        LED1_INIT();
        sem = OSSemCreate(0);               // 初始信号量值为0,使得task0首先被挂起
        OSTaskCreate( task0, (void *)0, &stk0[99], 3 );
        OSTaskCreate( task1, ( void *)0, &stk1[99], 2 );
        OSStart();
        return 0;
}
```

如上面的代码所示。创建了task1,task0任务，用信号量进行同步。

交通灯的控制，当然可以用一个任务来进行控制，但就用不着信号量了。如果采来两个任务，各个任务依次点亮灯，然后延时。这样两个任务就各自点灯，没有同步。此种方法存在的

问题是，在系统运行一些时间后，两个任务中其中某个可能会运行过快或过慢；两边的交通灯显示的变化就不是同时的。

task0每次在点灯后，向task1发送信号量，通知task1可以进行后续点灯操作了。task1在点灯后，会继续申请信号量，这时会被挂起，不能继续执行。而当task0在延时返回后，更换led显示，再向task1发送信号量，这样task1又可以运行，更改其led显示，然后再挂起.如此反复....

因为task1的点灯操作都是在task0发送信号量后进行，所以二者的动作能做到同步。

2)使用信号量闪烁LED - 信号量实现共离资源访问

```
void   task1( void * pdata )
{
       INT8U error;
       while(1)
       {
              OSSemPend( sem, 0, &error );
              LED1_ON( RED );                    // 点灯
              OSTimeDlyHMSM( 0, 0, 1, 0 );
              OSSemPost( sem );
       }
}

void   task0( void * pdata )
{
       INT8U  error;
       t2init();
       while(1)
       {
              OSSemPend( sem, 0, &error );
              LED1_OFF( RED );                   //  关灯
              OSTimeDlyHMSM( 0, 0, 1, 0 );
              OSSemPost( sem );
       }
}

int    main()
{
       OSInit();
       LED0_INIT();
       LED1_INIT();
       sem = OSSemCreate(1);
       OSTaskCreate( task0, (void *)0, &stk0[99], 3 );
       OSTaskCreate( task1, ( void *)0, &stk1[99], 2 );
       OSStart();
       return 0;
}
```

与前面的代码不同，上面的代码则使用两个任务来控制ＬＥＤ闪烁。task１负责点灯,task０负责熄灯。

因为led只有一盏，在半个周期(1s)内只允许有点灯或者熄灯，即只有task0 / task1运行。led此时作为临界资源，一次只允许一个任务占有，直至其放弃使用。

代码中的sem实际上是作为通行证，因为只有一盏led,所以通行证在创建时值 1 。task1,task0首行通过OSSemPend()申请通行证，当早请到时对LED操作，延时，然后放弃通行证，此时另外一个任务可以获取通行证再对led操作。可以看出，通过信号量，不会出现多个任务在操作led时的冲突；同时task0和task1在操作LED时也实现了操作的同步.

三、关于信号量的实现的体会

1、Ucos的信号量实现还是比较简单的。这里的简单指的是其代码易读，易理解。简单来说，需要是实现信号量计数与任务挂起列表的实现。对于任务代码而言，信号量提供了共享资源与同步的支持。而信号量本身的实现，则是通过开断中断实现临界区的访问。

2、信号量可以用于共享资源的访问和任务间的同步，实际在应用中如果处理不当，可能会产生“死锁”。死锁问题，在ucos内核中没有提供相应的解决方案，需要用户代码支持。

3、并不是ucos相关的ＡＰＩ可在所有情况下可用：ISR不可调用OSSemPend()，OSSemCreate().在使用时需要特别注意.

四、参考资源

1、ucos源码 V2.52
2、《嵌入式实时操作系统uc/os ii》