uC/OS-II: Task Synchronization

Embedded OS Implementation

Prof. Ya-Shu Chen
National Taiwan University
of Science and Technology

Objectives

- To know (and trace the codes of) the services:
  - event control block
  - semaphore
  - mutex

Event Control Block

- A building block to implement services such as
  - Semaphore
  - Mutual Exclusion Semaphore
  - Message Mailbox
  - Message Queue

- All ECB functions only consider how to manipulate data structures.

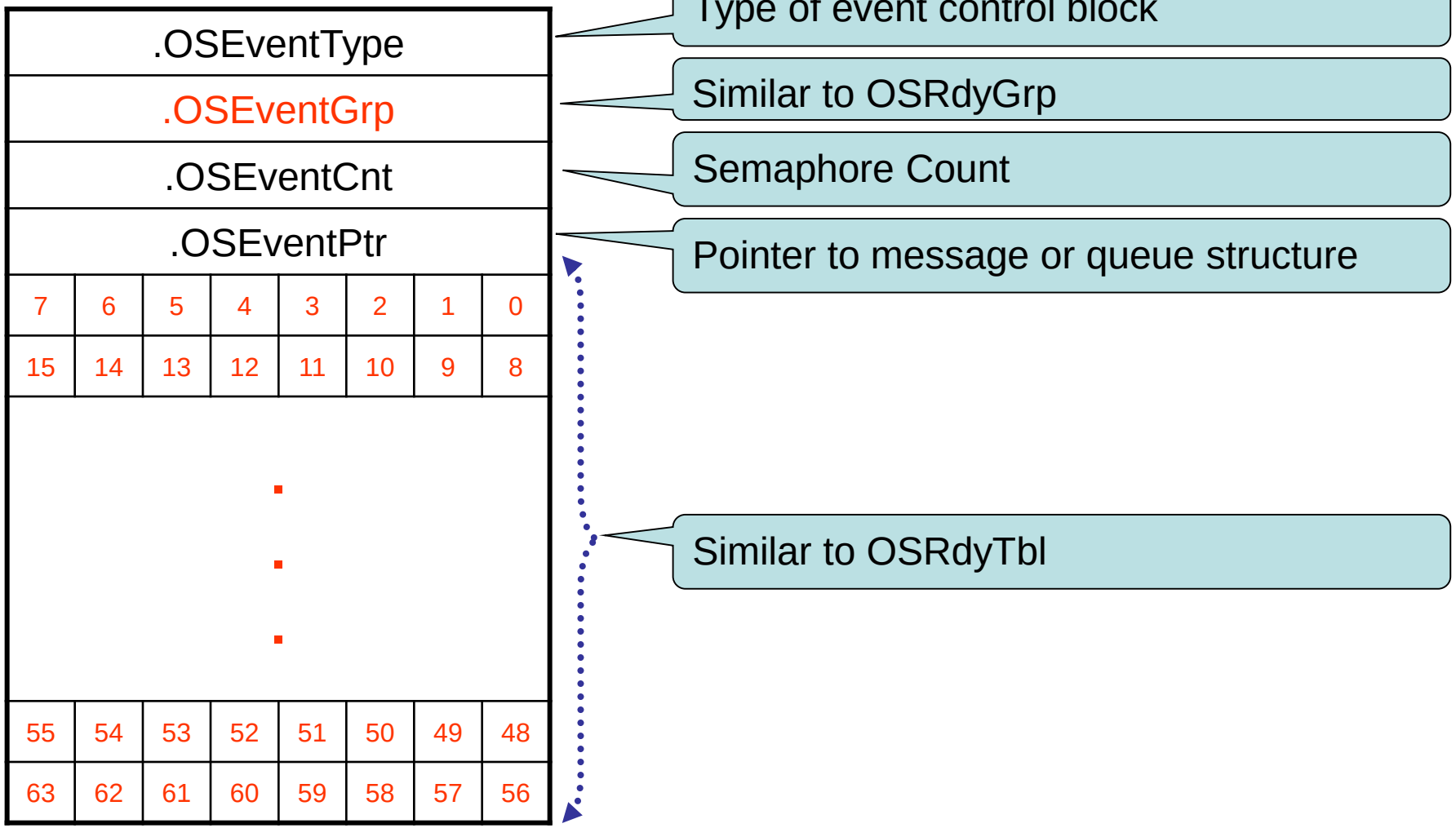
  Caller must consider synchronization issues

ECB data structure

```
/*
*********************************************************************************************
*                                          OS_EVENT types
*********************************************************************************************
*/
#define  OS_EVENT_TYPE_UNUSED      0
#define  OS_EVENT_TYPE_MBOX        1
#define  OS_EVENT_TYPE_Q           2
#define  OS_EVENT_TYPE_SEM         3
#define  OS_EVENT_TYPE_MUTEX       4
#define  OS_EVENT_TYPE_FLAG        5
```

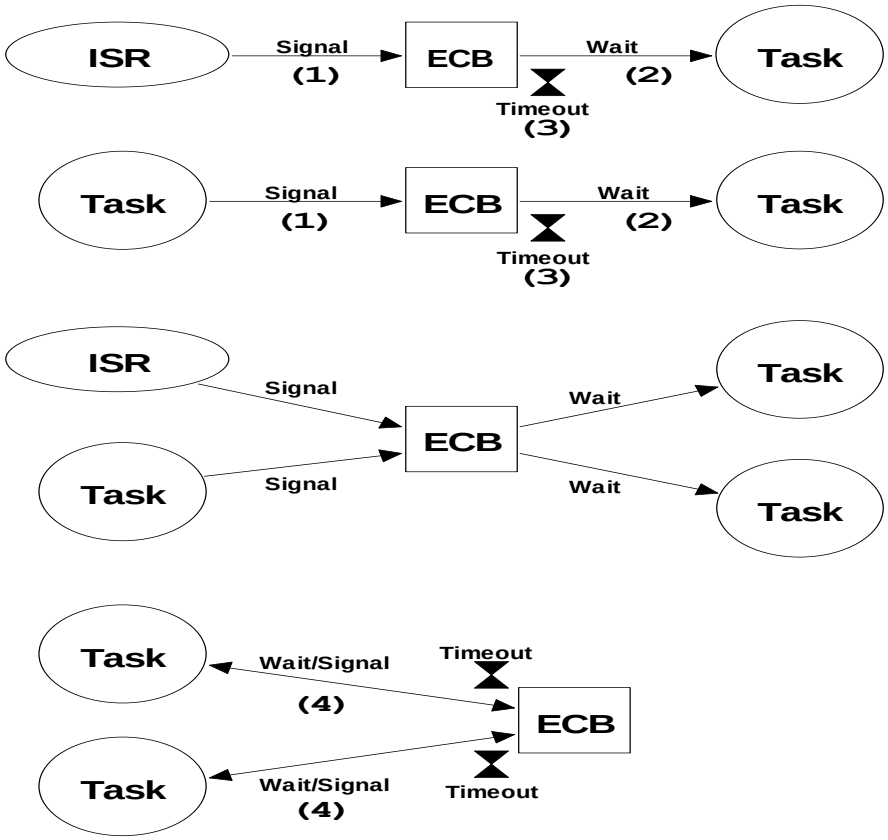

```
/*
*********************************************************************************************************
*                                              EVENT CONTROL BLOCK
*********************************************************************************************************
*/

#if (OS_EVENT_EN > 0) && (OS_MAX_EVENTS > 0)
typedef struct {
    INT8U    OSEventType;                    /* Type of event control block (see OS_EVENT_TYPE_???)      */
    INT8U    OSEventGrp;                     /* Group corresponding to tasks waiting for event to occur  */
    INT16U   OSEventCnt;                     /* Semaphore Count (not used if other EVENT type)           */
    void    *OSEventPtr;                     /* Pointer to message or queue structure                    */
    INT8U    OSEventTbl[OS_EVENT_TBL_SIZE]; /* List of tasks waiting for event to occur                 */
} OS_EVENT;
#endif
```

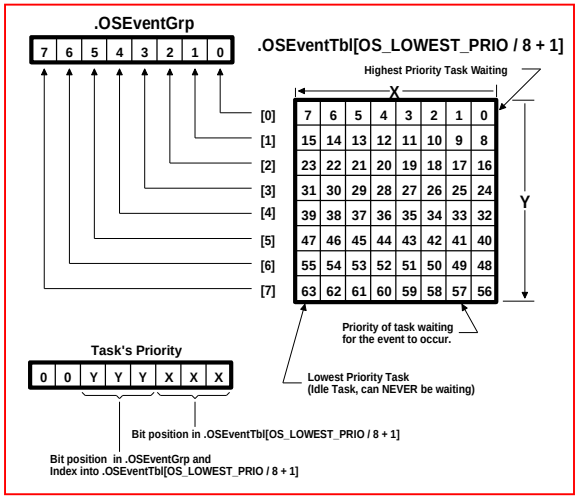
ECB data structure

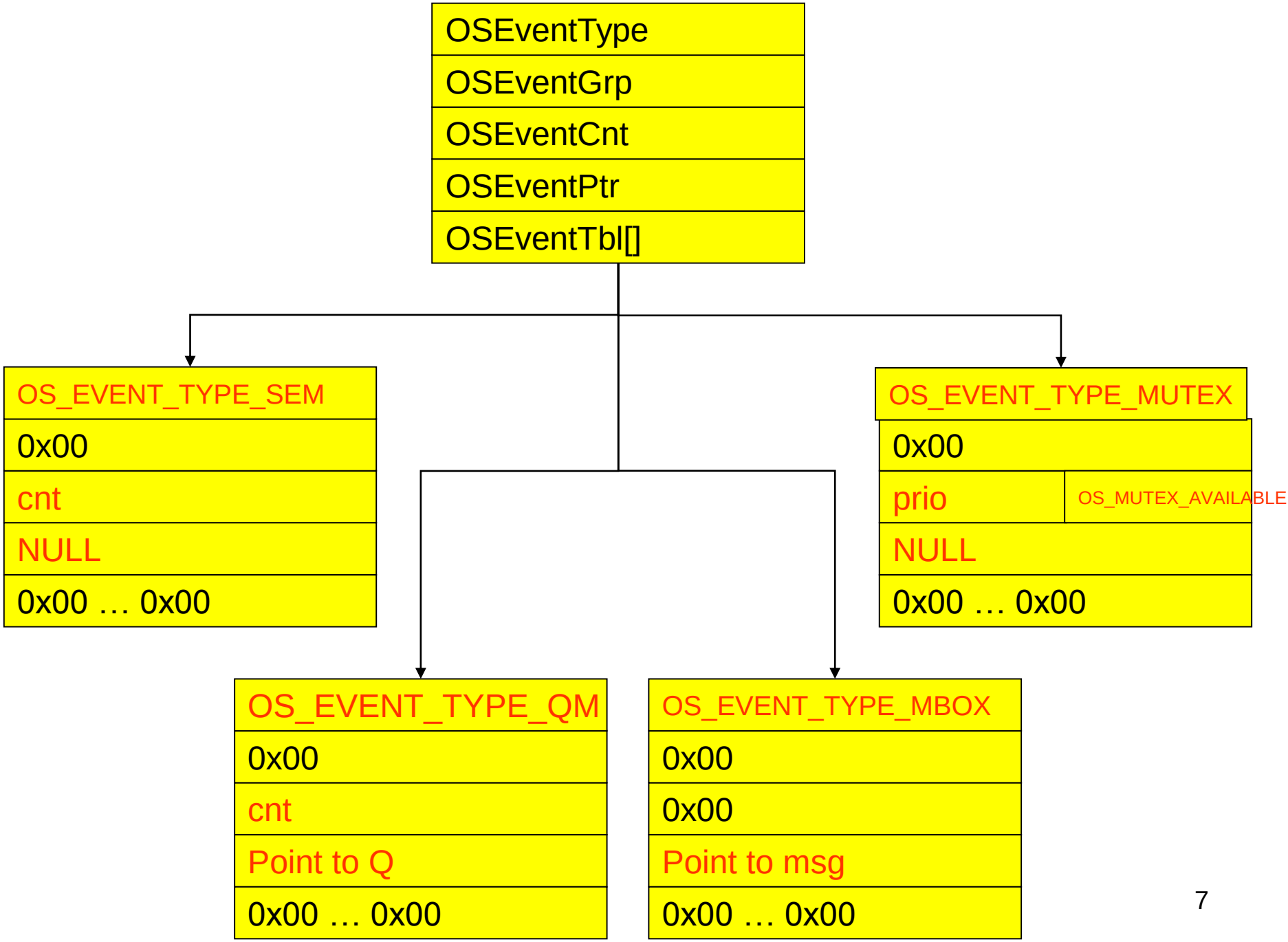


"red" fields are Initialized by OS_EventWaitListInit

```
typedef struct {
    INT8U   OSEventType;                    /* Event type                    */
    INT8U   OSEventGrp;                     /* Group for wait list           */
    INT16U  OSEventCnt;                     /* Count (when event is a semaphore) */
    void   *OSEventPtr;                     /* Ptr to message or queue structure */
    INT8U   OSEventTbl[OS_EVENT_TBL_SIZE]; /* Wait list for event to occur   */
} OS_EVENT;
```






OSEventType
OSEventGrp
OSEventCnt
OSEventPtr
OSEventTbl[]
OS_EVENT_TYPE_SEM
0x00
cnt
NULL
0x00 … 0x00
OS_EVENT_TYPE_MUTEX
0x00
prio
OS_MUTEX_AVAILABLE
NULL
0x00 … 0x00
OS_EVENT_TYPE_QM
0x00
cnt
Point to Q
0x00 … 0x00
OS_EVENT_TYPE_MBOX
0x00
0x00
Point to msg
0x00 … 0x00

Functions

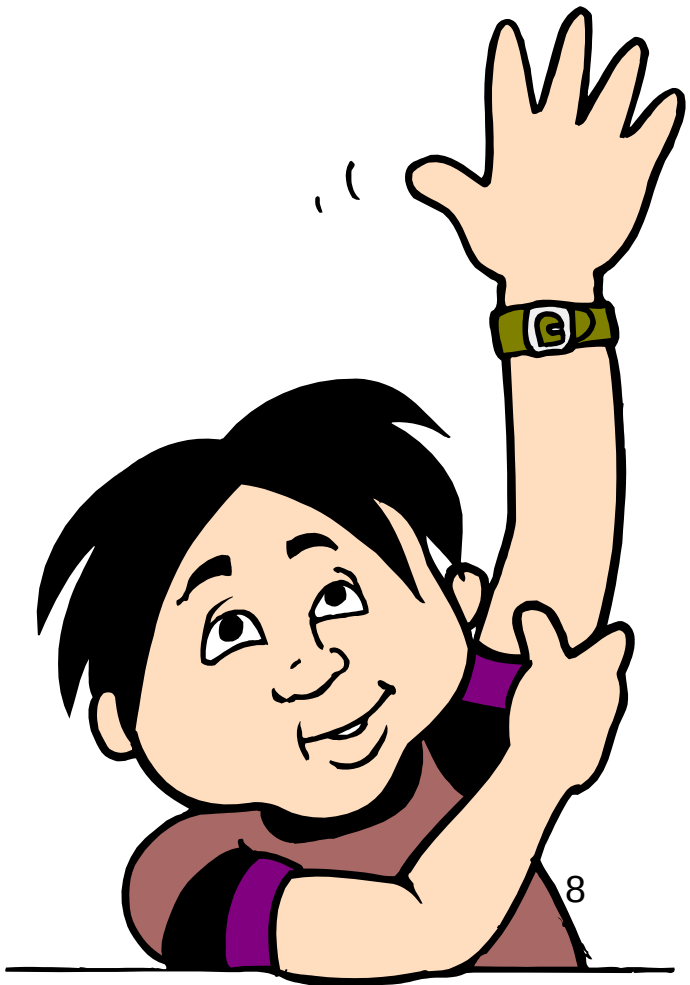
- OS_EventWaitListInit()

  Initialize an ECB
- OS_EventTaskRdy()

  Make a task ready
- OS_EventTaskWait()

  Put the task to sleep
- OS_EventTO()

  Make a task ready



OS_EventWaitListInit()

```
#if ((OS_Q_EN > 0) && (OS_MAX_QS > 0)) || (OS_MBOX_EN > 0) || (OS_SEM_EN > 0) || (OS_MUTEX_EN > 0)
void  OS_EventWaitListInit (OS_EVENT *pevent)
{
    INT8U  *ptbl;


    pevent->OSEventGrp = 0x00;                      /* No task waiting on event                           */
    ptbl               = &pevent->OSEventTbl[0];

#if OS_EVENT_TBL_SIZE > 0
    *ptbl++            = 0x00;
#endif

#if OS_EVENT_TBL_SIZE > 1
    *ptbl++            = 0x00;
#endif

#if OS_EVENT_TBL_SIZE > 2
    *ptbl++            = 0x00;
#endif

#if OS_EVENT_TBL_SIZE > 3
    *ptbl++            = 0x00;
#endif

#if OS_EVENT_TBL_SIZE > 4
    *ptbl++            = 0x00;
#endif

#if OS_EVENT_TBL_SIZE > 5
    *ptbl++            = 0x00;
#endif

#if OS_EVENT_TBL_SIZE > 6
    *ptbl++            = 0x00;
#endif

#if OS_EVENT_TBL_SIZE > 7
    *ptbl              = 0x00;
#endif
}
#endif
```

OS_EventTaskRdy()

- This function is called by the POST functions for a semaphore, a mutex, a message mailbox or a message queue when the ECB is signaled.

- OS_EventTaskRdy() removes the highest priority task from the wait list of the ECB and makes this task ready to run.

OS_EventTaskRdy()

```
INT8U  OS_EventTaskRdy (OS_EVENT *pevent, void *msg, INT8U msk) {
    OS_TCB *ptcb;
    INT8U   x, y, bitx, bity, prio;

    y    = OSUnMapTbl[pevent->OSEventGrp];
    bity = OSMapTbl[y];
    x    = OSUnMapTbl[pevent->OSEventTbl[y]];
    bitx = OSMapTbl[x];
    prio = (INT8U)((y << 3) + x);
    if ((pevent->OSEventTbl[y] &= ~bitx) == 0x00)
        pevent->OSEventGrp &= ~bity;
    ptcb                = OSTCBPrioTbl[prio];
    ptcb->OSTCBDly      = 0;
    ptcb->OSTCBEventPtr = (OS_EVENT *)0;
    ptcb->OSTCBMsg      = msg;
    ptcb->OSTCBStat    &= ~msk;
    if (ptcb->OSTCBStat == OS_STAT_RDY) {
        OSRdyGrp       |= bity;       OSRdyTbl[y]    |= bitx;
    }
    return (prio);
}
```

Find highest priority task waiting for message

Remove this task from the waiting list

Prevent OSTimeTick() from readying task

Unlink ECB from this task

Clear bit associated with event type

Set the task ready to run if the task is not suspended

OS_EventTaskWait()

- This function is called by the PEND functions for a semaphore, a mutex, a message mailbox or a message queue when a task must wait on an ECB.

- It removes the current task from the ready list and places it in the wait list of the ECB.

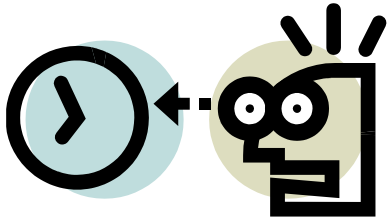
OS_EventTaskWait()

```
void  OS_EventTaskWait (OS_EVENT *pevent)
{
    OSTCBCur->OSTCBEventPtr = pevent;
    if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0x00) {
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
    }
    pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;
    pevent->OSEventGrp                   |= OSTCBCur->OSTCBBitY;
}
```

Task no longer ready

Put task in waiting list

OS_EventTO()



- OS_EventTO = OS-Event-Time-Out

- This function is called by the PEND functions for a semaphore, a mutex, a message mailbox or a message queue when the ECB is not signaled within the specified timeout period.

OS_EventTO()

```
void  OS_EventTO (OS_EVENT *pevent)
{
    if ((pevent->OSEventTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) ==
0x00) {
        pevent->OSEventGrp &= ~OSTCBCur->OSTCBBitY;
    }
    OSTCBCur->OSTCBStat     = OS_STAT_RDY;
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
}
```

Remove the task from wait list of the ECB

Clock Tick

```
void  OSTimeTick (void)
{
    OS_TCB    *ptcb;

    OSTimeTickHook();

    if (OSRunning == TRUE) {
        ptcb = OSTCBList;
        while (ptcb->OSTCBPrio != OS_IDLE_PRIO) {
            OS_ENTER_CRITICAL();
            if (ptcb->OSTCBDly != 0) {
                if (--ptcb->OSTCBDly == 0) {
                    if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) == OS_STAT_RDY) {
                        OSRdyGrp               |= ptcb->OSTCBBitY;
                        OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                    } else {
                        ptcb->OSTCBDly = 1;
                    }
                }
            }
            ptcb = ptcb->OSTCBNext;
            OS_EXIT_CRITICAL();
        }
    }
}
```

For all TCB's

Decrement delay-counter if needed

If the delay-counter reaches zero, make the task ready. Or, the task remains waiting.

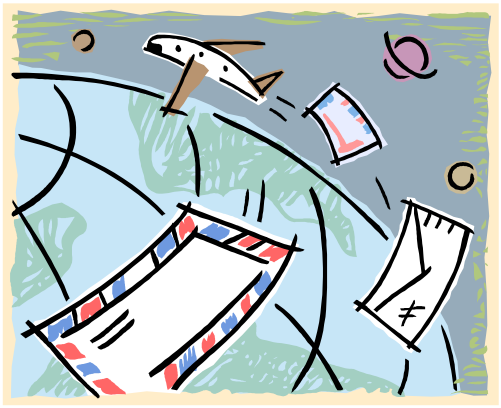
Semaphores

- A 16-bit unsigned integer used to hold the semaphore count (0 to 65535)

- Create a semaphore by calling OSSemCreate( )
  - Specify the initial value of a semaphore

Semaphore Management

- OSSemCreate()
  Creating a semaphore
- OSSemDel()
  Deleting a semaphore
- OSSemPend
  Waiting on a semaphore
- OSSemPost
  Signaling a semaphore
- OSSemAccept()
  Getting a Semaphore without waiting



OSSemCreate()

```
OS_EVENT  *OSSemCreate (INT16U cnt)
{
#if OS_CRITICAL_METHOD == 3                          /* Allocate storage for CPU status register */
    OS_CPU_SR  cpu_sr;
#endif
    OS_EVENT  *pevent;


    if (OSIntNesting > 0) {                          /* See if called from ISR ...               */
        return ((OS_EVENT *)0);                      /* ... can't CREATE from an ISR             */
    }
    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;                        /* Get next free event control block        */
    if (OSEventFreeList != (OS_EVENT *)0) {          /* See if pool of free ECB pool was empty   */
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {                   /* Get an event control block               */
        pevent->OSEventType = OS_EVENT_TYPE_SEM;
        pevent->OSEventCnt  = cnt;                   /* Set semaphore value                      */
        pevent->OSEventPtr  = (void *)0;             /* Unlink from ECB free list                */
        OS_EventWaitListInit(pevent);                /* Initialize to 'nobody waiting' on sem.   */
    }
    return (pevent);
}
```

All kernel objects need to be created from task-level code or before multitasking starting

Obtain a free ECB

Clear OSEventGrp and OSEventTbl[]

OSSemDel()

- Pevent is a pointer to the ECB associated with the desired semaphore.
- Opt determines delete options as follows:
  - OS_DEL_NO_PEND

    Delete semaphore ONLY if no task pending
  - OS_DEL_ALWAYS

    Deletes the semaphore even if tasks are waiting. In this case, all the tasks pending will be readied.

OSSemDel()

```
OS_EVENT  *OSSemDel (OS_EVENT *pevent, INT8U opt, INT8U *err) {
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {
        tasks_waiting = TRUE;
    } else tasks_waiting = FALSE;
    switch (opt) {
        case OS_DEL_NO_PEND:
            if (tasks_waiting == FALSE) {
                pevent->OSEventType = OS_EVENT_TYPE_UNUSED;
                pevent->OSEventPtr  = OSEventFreeList;
                OSEventFreeList     = pevent;
                OS_EXIT_CRITICAL();
                *err = OS_NO_ERR;
                return ((OS_EVENT *)0);
            } else {
                OS_EXIT_CRITICAL();
                *err = OS_ERR_TASK_WAITING;
                return (pevent);
            }
```

See if any tasks waiting on semaphore

Delete semaphore only if no task waiting

Return Event Control Block to free list

OSSemDel()

```
case OS_DEL_ALWAYS:
        while (pevent->OSEventGrp != 0x00) {
            OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM);
        }
        pevent->OSEventType = OS_EVENT_TYPE_UNUSED;
        pevent->OSEventPtr  = OSEventFreeList;
        OSEventFreeList     = pevent;
        OS_EXIT_CRITICAL();
        if (tasks_waiting == TRUE) {
            OS_Sched();
        }
        *err = OS_NO_ERR;
        return ((OS_EVENT *)0);

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

Always delete the semaphore

Ready aLL tasks waiting for semaphore

OSSemDel()

- This call can potentially disable interrupts for a long time.
  - The interrupt disable time is directly proportional to the number of tasks waiting on the semaphore.
- Because all tasks pending on the semaphore will be accessed, you must be careful in applications where the semaphore is used for mutual exclusion.

OSSemPend()

```
void  OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3                          /* Allocate storage for CPU status register      */
    OS_CPU_SR  cpu_sr;
#endif


    if (OSIntNesting > 0) {                          /* See if called from ISR ...                    */
        *err = OS_ERR_PEND_ISR;                      /* ... can't PEND from an ISR                    */
        return;
    }
#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                   /* Validate 'pevent'                             */
        *err = OS_ERR_PEVENT_NULL;
        return;
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {  /* Validate event block type                     */
        *err = OS_ERR_EVENT_TYPE;
        return;
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventCnt > 0) {                    /* If sem. is positive, resource available ...   */
        pevent->OSEventCnt--;                        /* ... decrement semaphore only if positive.     */
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
        return;
    }
```

The semaphore is available.

Decrement semaphore only if positive

OSSemPend()

The semaphore is not available.

```
    OSTCBCur->OSTCBStat |= OS_STAT_SEM;          /* Otherwise, must wait until event occurs       */
    OSTCBCur->OSTCBDly   = timeout;              /* Resource not available, pend on semaphore     */
    OS_EventTaskWait(pevent);                    /* Store pend timeout in TCB                     */
    OS_EXIT_CRITICAL();                          /* Suspend task until event or timeout occurs    */
    OS_Sched();                                  /* Find next highest priority task ready         */
    OS_ENTER_CRITICAL();
    if (OSTCBCur->OSTCBStat & OS_STAT_SEM) {     /* Must have timed out if still waiting for event*/
        OS_EventTO(pevent);
        OS_EXIT_CRITICAL();
        *err = OS_TIMEOUT;                       /* Indicate that didn't get event within TO      */
        return;
    }
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
}
```

Timeout

When the semaphore is signaled OSSemPend() resumes executing immediately after OS_Sched()

Clock Tick

```
void  OSTimeTick (void)
{
   OS_TCB    *ptcb;

   OSTimeTickHook();

   if (OSRunning == TRUE) {
      ptcb = OSTCBList;
      while (ptcb->OSTCBPrio != OS_IDLE_PRIO) {
         OS_ENTER_CRITICAL();
         if (ptcb->OSTCBDly != 0) {
            if (--ptcb->OSTCBDly == 0) {
               if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) == OS_STAT_RDY) {
                  OSRdyGrp               |= ptcb->OSTCBBitY;
                  OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
               } else {
                  ptcb->OSTCBDly = 1;
               }
            }
         }
         ptcb = ptcb->OSTCBNext;
         OS_EXIT_CRITICAL();
      }
   }
}
```

For all TCB's

Decrement delay-counter if needed

If the delay-counter reaches zero, make the task ready. Or, the task remains waiting.

OSSemPost()

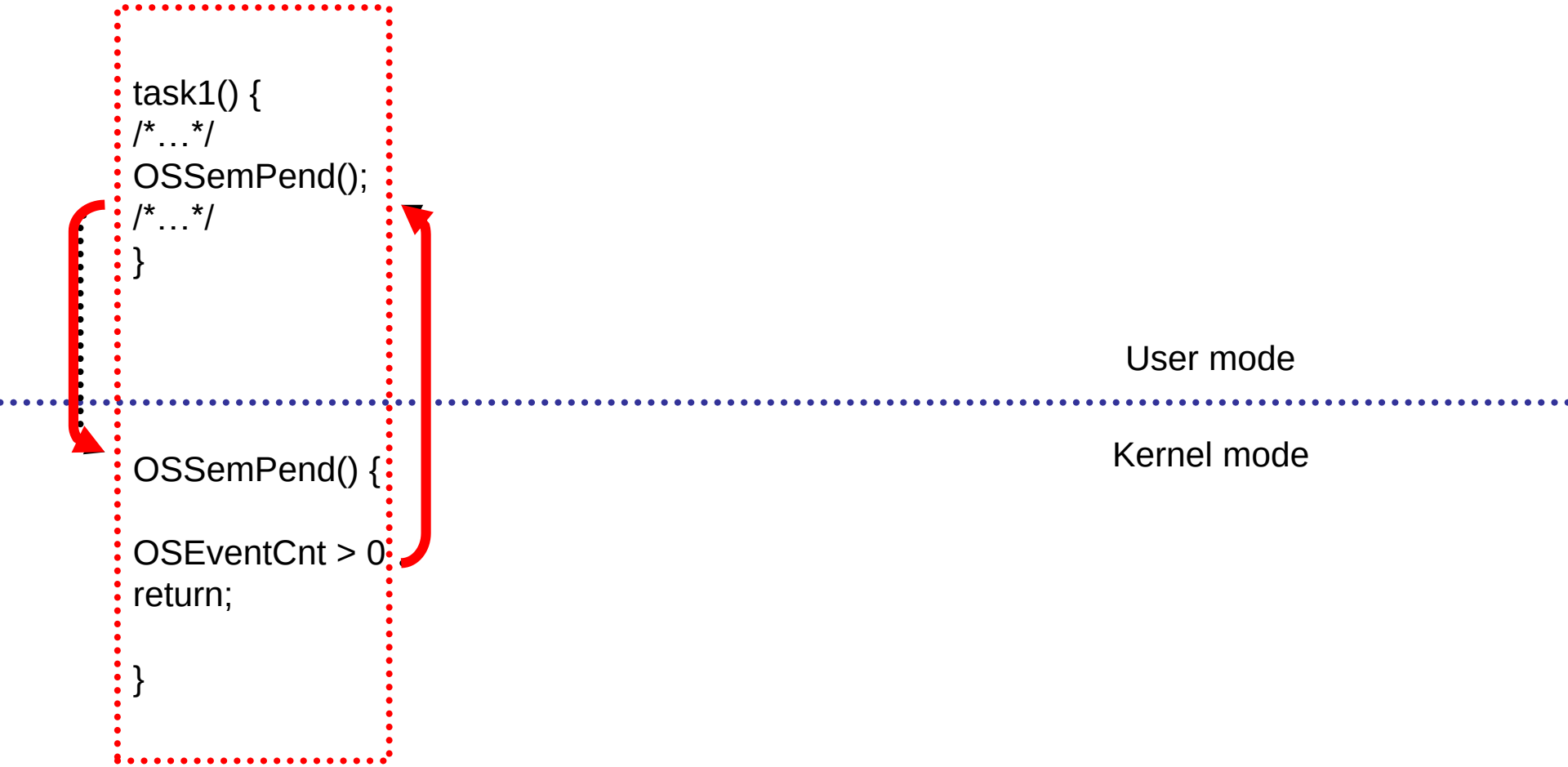
```
INT8U  OSSemPost (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3                                  /* Allocate storage for CPU status register */
    OS_CPU_SR  cpu_sr;
#endif


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                           /* Validate 'pevent'                        */
        return (OS_ERR_PEVENT_NULL);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {          /* Validate event block type                */
        return (OS_ERR_EVENT_TYPE);
    }
#endif
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                        /* See if any task waiting for semaphore    */
        OS_EventTaskRdy(pevent, (void *)0, OS_STAT_SEM);     /* Ready highest prio task waiting on event */
        OS_EXIT_CRITICAL();
        OS_Sched();                                          /* Find highest priority task ready to run  */
        return (OS_NO_ERR);
    }
    if (pevent->OSEventCnt < 65535) {                    /* Make sure semaphore will not overflow        */
        pevent->OSEventCnt++;                            /* Increment semaphore count to register event  */
        OS_EXIT_CRITICAL();
        return (OS_NO_ERR);
    }
    OS_EXIT_CRITICAL();                                  /* Semaphore value has reached its maximum      */
    return (OS_SEM_OVF);
}
```

See if any task waiting for semaphore

Ready the highest priority task waiting on the event

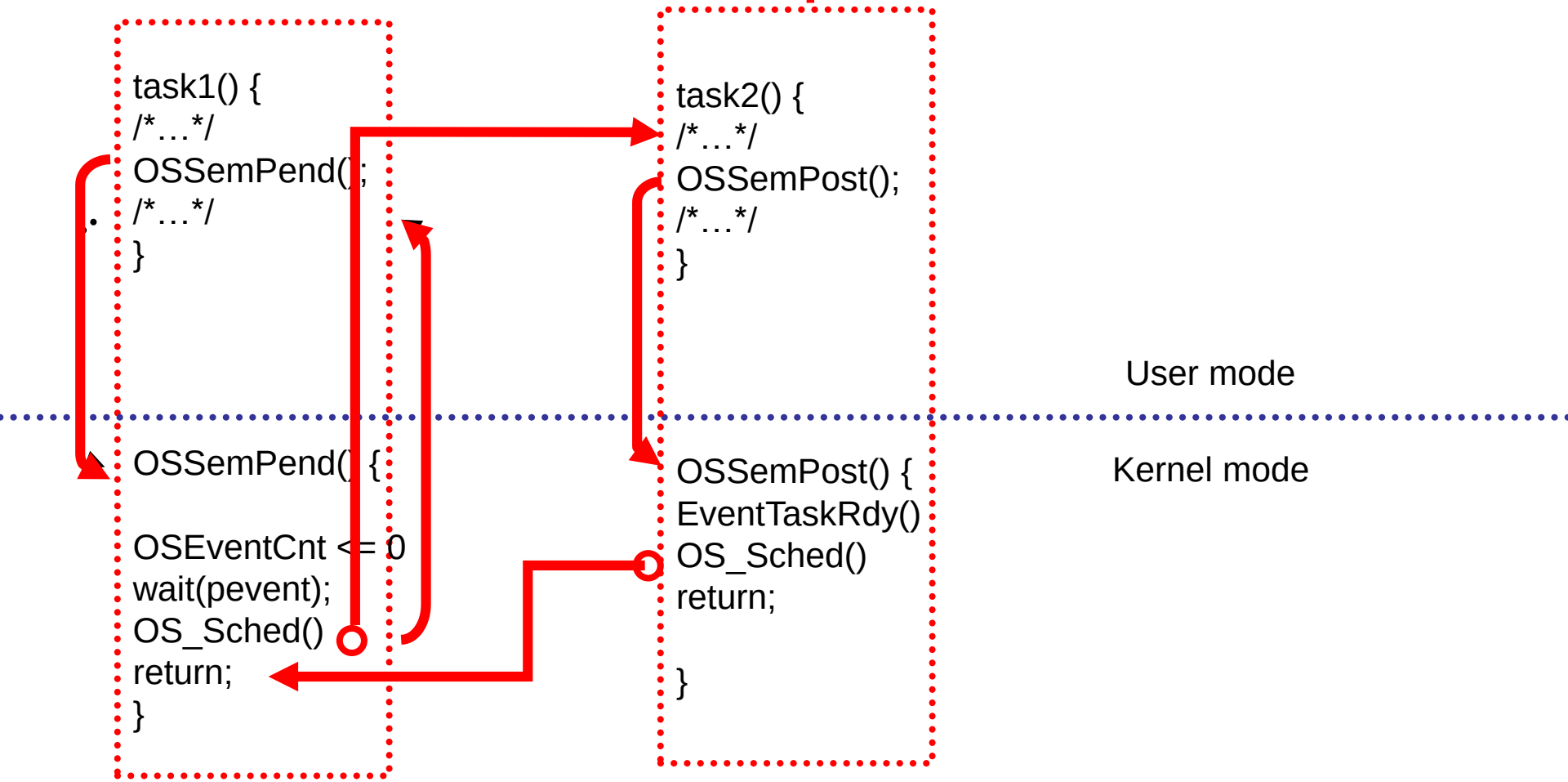
Increment the semaphore count

OSSemPend()
Example 1



OSSemPend()

Example 2



```
void OSSched (void)
{
    INT8U y;
    OS_ENTER_CRITICAL();
    if ((OSLockNesting | OSIntNesting) == 0) {                          (1)
        y             = OSUnMapTbl[OSRdyGrp];                           (2)
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);    (2)
        if (OSPrioHighRdy != OSPrioCur) {                               (3)
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];                 (4)
            OSCtxSwCtr++;                                               (5)
            OS_TASK_SW();                                               (6)
        }
    }
    OS_EXIT_CRITICAL();
}
```

(1) Rescheduling will not be performed if the scheduler is locked or some interrupt is currently serviced (why?).

(2) Find the highest-priority ready task.

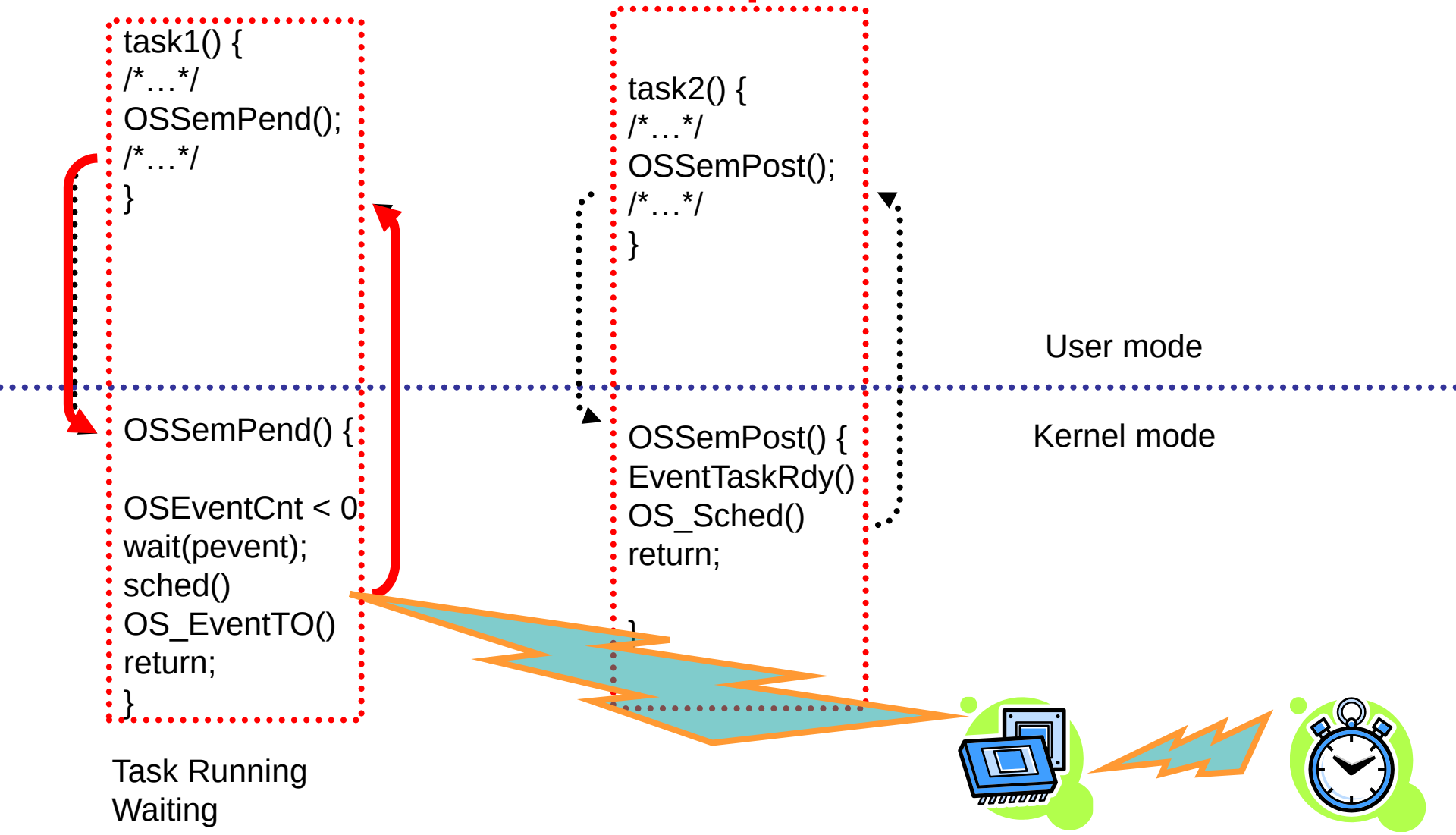
(3) If it is not the current task, then

(4) ~(6) Perform a context-switch.

OS_TASK_SW() is a macro: “asm int 0x80”

OSSemPend()

Example 3



Clock Tick

```
void  OSTimeTick (void)
{
   OS_TCB    *ptcb;

   OSTimeTickHook();

   if (OSRunning == TRUE) {
      ptcb = OSTCBList;
      while (ptcb->OSTCBPrio != OS_IDLE_PRIO) {
         OS_ENTER_CRITICAL();
         if (ptcb->OSTCBDly != 0) {
            if (--ptcb->OSTCBDly == 0) {
               if ((ptcb->OSTCBStat & OS_STAT_SUSPEND) == OS_STAT_RDY) {
                  OSRdyGrp               |= ptcb->OSTCBBitY;
                  OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
               } else {
                  ptcb->OSTCBDly = 1;
               }
            }
         }
         ptcb = ptcb->OSTCBNext;
         OS_EXIT_CRITICAL();
      }
   }
}
```

For all TCB's

Decrement delay-counter if needed

If the delay-counter reaches zero, make the task ready. Or, the task remains waiting.

Interrupts under uC/OS-2

```
void OSIntExit (void)
{
   OS_ENTER_CRITICAL();
   if ((--OSIntNesting | OSLockNesting) == 0) {
      OSIntExitY    = OSUnMapTbl[OSRdyGrp];
      OSPrioHighRdy = (INT8U)((OSIntExitY << 3) +
                      OSUnMapTbl[OSRdyTbl[OSIntExitY]]);
      if (OSPrioHighRdy != OSPrioCur) {
         OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
         OSCtxSwCtr++;
         OSIntCtxSw();
      }
   }
   OS_EXIT_CRITICAL();
}
```

If scheduler is not locked and no interrupt nesting

If there is another high-priority task ready

A context switch is performed.

Note that OSIntCtxSw() is called instead of calling OS_TASK_SW() because the ISR already saves the CPU registers onto the stack.

```
void OSIntEnter (void)
{
   OS_ENTER_CRITICAL();
   OSIntNesting++;
   OS_EXIT_CRITICAL();
}
```

OSSemAccept()

```
#if OS_SEM_ACCEPT_EN > 0
INT16U  OSSemAccept (OS_EVENT *pevent)
{
#if OS_CRITICAL_METHOD == 3                          /* Allocate storage for CPU status register      */
    OS_CPU_SR  cpu_sr;
#endif
    INT16U     cnt;


#if OS_ARG_CHK_EN > 0
    if (pevent == (OS_EVENT *)0) {                   /* Validate 'pevent'                             */
        return (0);
    }
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {  /* Validate event block type                     */
        return (0);
    }
#endif
    OS_ENTER_CRITICAL();
    cnt = pevent->OSEventCnt;
    if (cnt > 0) {                                   /* See if resource is available                  */
        pevent->OSEventCnt--;                        /* Yes, decrement semaphore and notify caller    */
    }
    OS_EXIT_CRITICAL();
    return (cnt);                                    /* Return semaphore count                        */
}
#endif
```

OSSemQuery()

```
typedef struct {
    INT16U  OSCnt;                          /* Semaphore count                                          */
    INT8U   OSEventTbl[OS_EVENT_TBL_SIZE];  /* List of tasks waiting for event to occur                 */
    INT8U   OSEventGrp;                     /* Group corresponding to tasks waiting for event to occur */
} OS_SEM_DATA;

INT8U  OSSemQuery (OS_EVENT *pevent, OS_SEM_DATA *pdata)
{
#if OS_CRITICAL_METHOD == 3                                 /* Allocate storage for CPU status register */
    OS_CPU_SR  cpu_sr;
#endif
    INT8U     *psrc;
    INT8U     *pdest;

#endif
    OS_ENTER_CRITICAL();
    pdata->OSEventGrp = pevent->OSEventGrp;                 /* Copy message mailbox wait list           */
    psrc              = &pevent->OSEventTbl[0];
    pdest             = &pdata->OSEventTbl[0];
#if OS_EVENT_TBL_SIZE > 0
    *pdest++          = *psrc++;
#endif
                                                            .
                                                            .
                                                            .
#if OS_EVENT_TBL_SIZE > 6
    *pdest++          = *psrc++;
#endif

#if OS_EVENT_TBL_SIZE > 7
    *pdest            = *psrc;
#endif
    pdata->OSCnt      = pevent->OSEventCnt;                 /* Get semaphore count                      */
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

Mutual Exclusion Semaphores

- A mutex is used by your application code to reduce the priority inversion problem

- The kernel can raise the priority of the lower priority task to shorten the blocking time suffered by the higher priority task
  - In order to implement priority inheritance, a kernel needs to provide the ability to support multiple tasks at the same priority

Mutual Exclusion Semaphores

- uC/OS-II uses an alternative approach to work around that every task must have unique priority
  - Let a priority is reserved for a mutex
  - The reserved priority (for the mutex) must be higher than all tasks that could use the mutex

Mutual Exclusion Semaphores

- uC/OS-II
  - It differs from ceiling priority protocol
    - CPP raises priority when resource is locked
    - uC/OS-II raises priority when contending resources
  - It differs from priority-inheritance protocol
    - Consider: priority: [mutex]→T1→T2→T3
      - T2 and T3 uses mutex but not T1
    - Transitive priority inheritance and inheriting priorities for multiple times is not possible

Mutual Exclusion Semaphores

- Be careful about the priority for mutex!!
  - It should be immediately higher than the highest priority of all tasks use the mutex
    - Or extra blocking time might be introduced!!

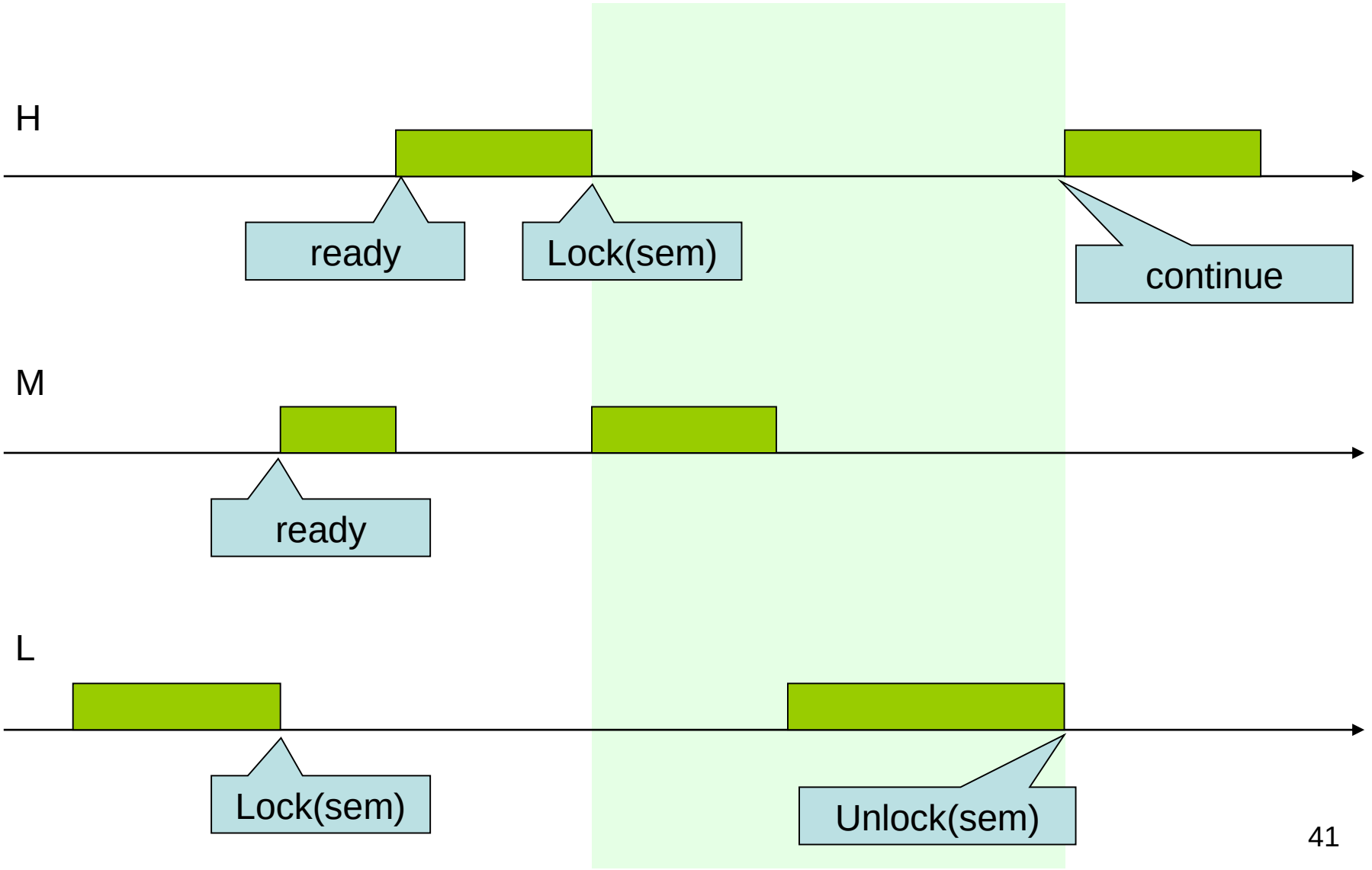
Example1

```
OSMutexCreate(VH, &err);

void taskPrioH {
        while(1) {
                /*…*/
                OSMutexPend(mutex, 0, &err);
                /*…*/
                OSMutexPost(Mutex);
        }
}
```

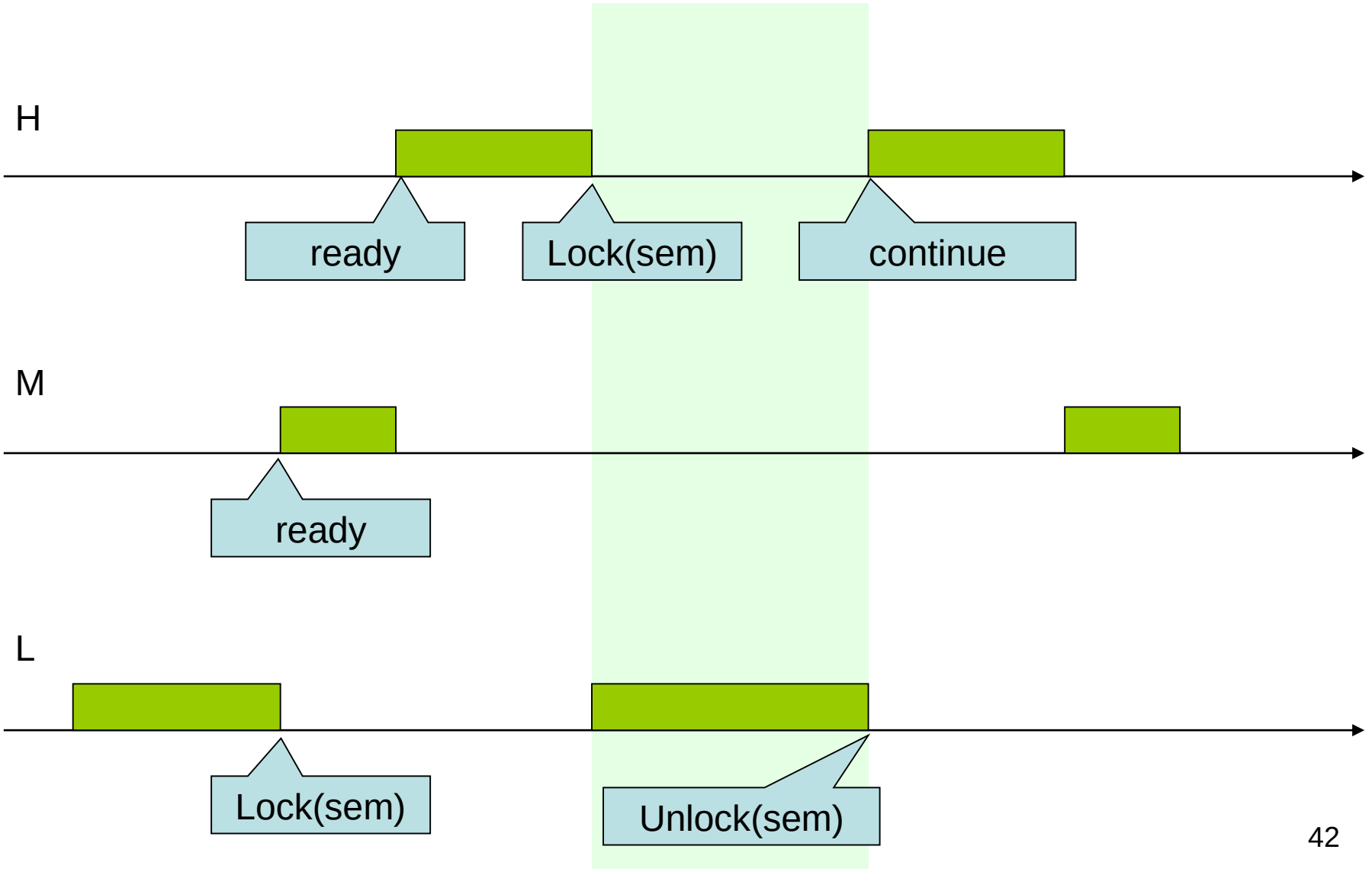
```
void taskPrioM {
        while(1) {
                /*…*/
        }
}
```

```
void taskPrioL {
        while(1) {
                /*…*/
                OSMutexPend(mutex, 0, &err);
                /*…*/
                OSMutexPost(Mutex);
        }
}
```

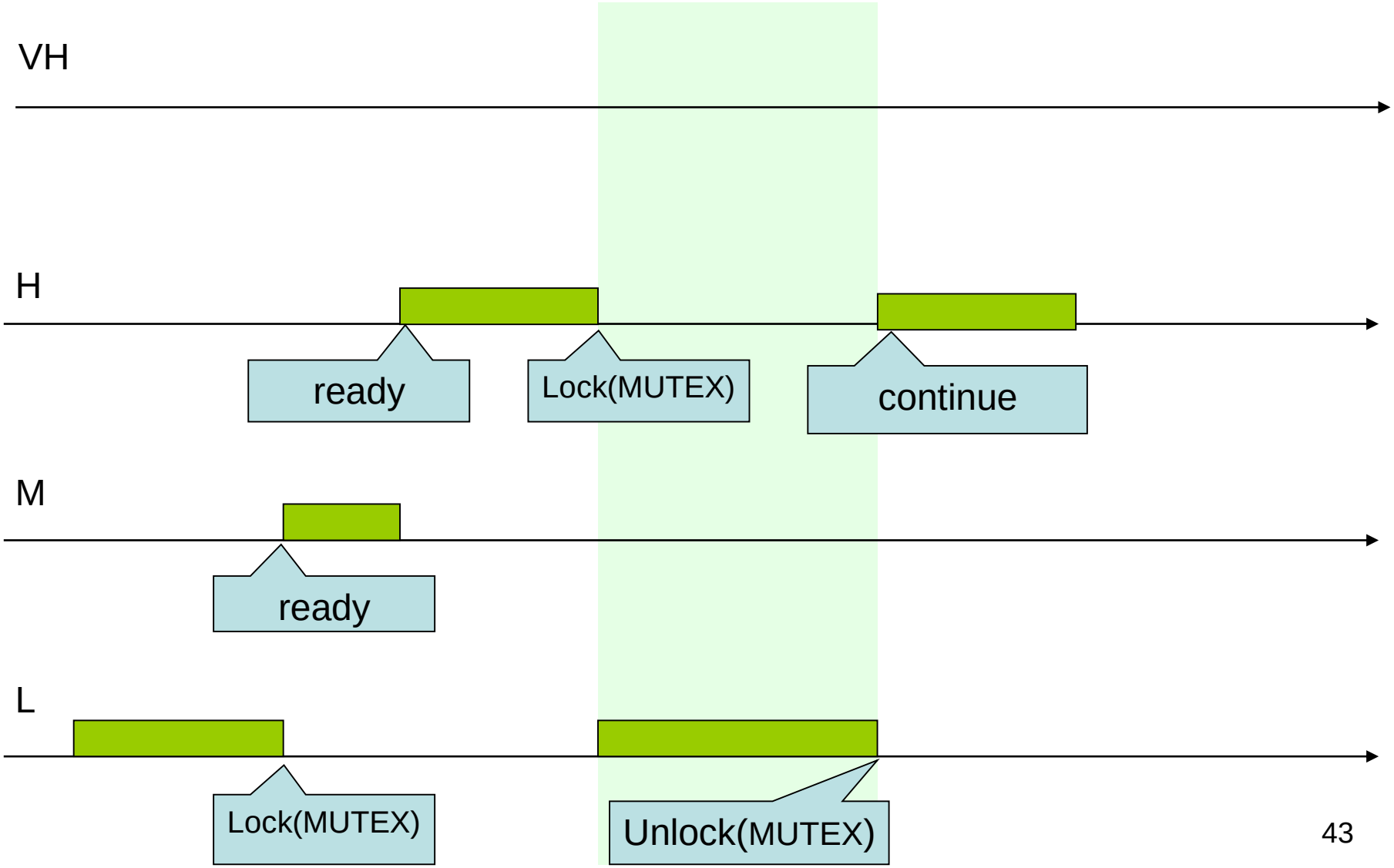
Example - without PIP



Example - with PIP



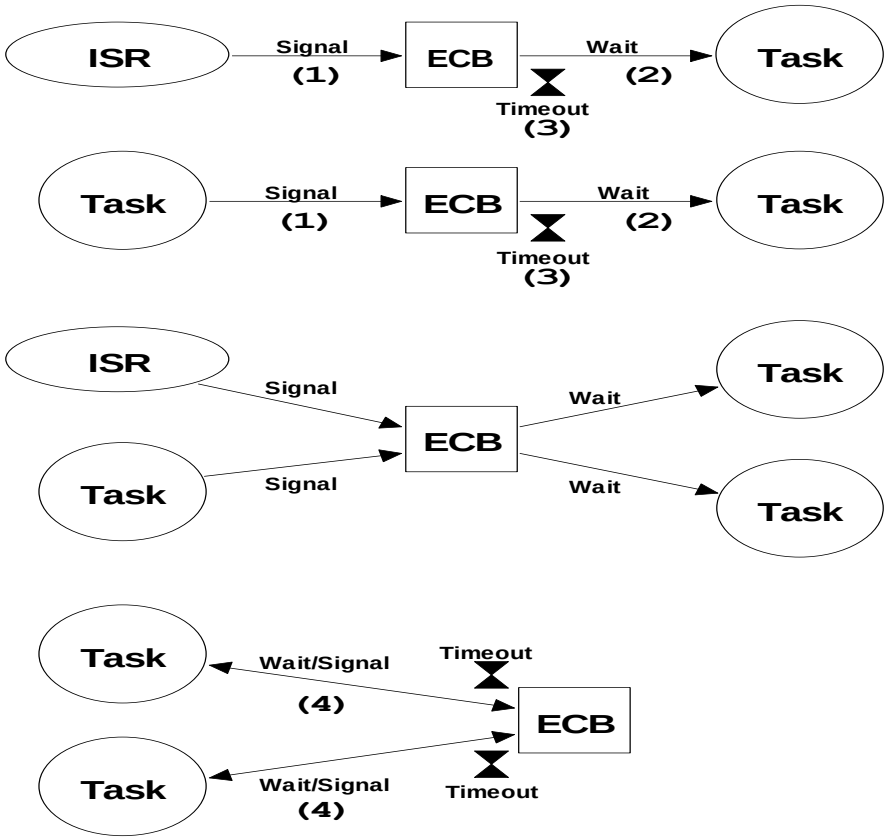
Example - µC/OS-II

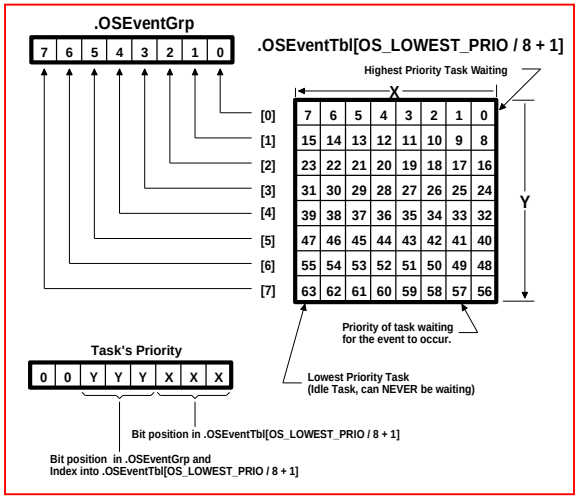


Functions

- OSMutexCreate()
- OSMutexDel()
- OSMutexPend()
- OSMutexAccept()
- OSMutexPost()
- OSMutexQuery()

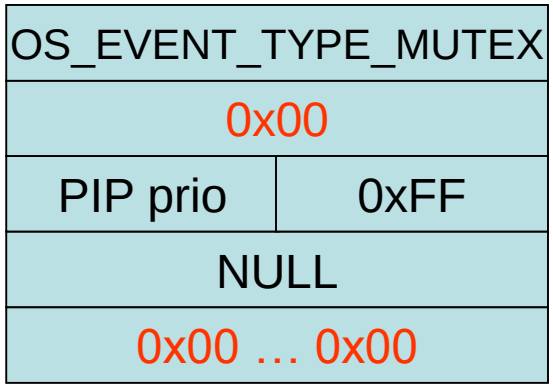
```
typedef struct {
    INT8U   OSEventType;                    /* Event type                    */
    INT8U   OSEventGrp;                     /* Group for wait list           */
    INT16U  OSEventCnt;                     /* Count (when event is a semaphore) */
    void   *OSEventPtr;                     /* Ptr to message or queue structure */
    INT8U   OSEventTbl[OS_EVENT_TBL_SIZE]; /* Wait list for event to occur   */
} OS_EVENT;
```



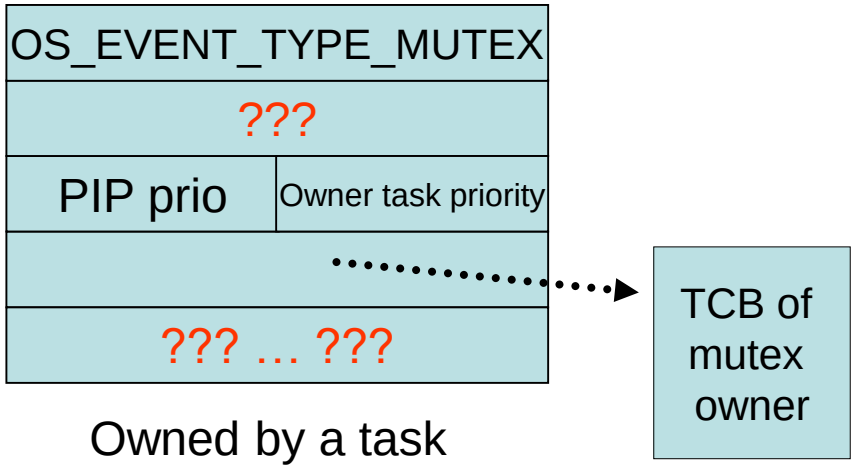


OSMutexCreate()

- The major difference between OSMutexCreate() and OSSemCreate()
  - OSMutexCreate() would reserve a priority slot for priority inheritance protocol



available



Owned by a task

```
OS_EVENT  *OSMutexCreate (INT8U prio, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3                                  /* Allocate storage for CPU status register */
    OS_CPU_SR  cpu_sr;
#endif
    OS_EVENT  *pevent;


    if (OSIntNesting > 0) {                                  /* See if called from ISR ...               */
        *err = OS_ERR_CREATE_ISR;                            /* ... can't CREATE mutex from an ISR       */
        return ((OS_EVENT *)0);
    }
#if OS_ARG_CHK_EN > 0
    if (prio >= OS_LOWEST_PRIO) {                            /* Validate PIP                             */
        *err = OS_PRIO_INVALID;
        return ((OS_EVENT *)0);
    }
#endif
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] != (OS_TCB *)0) {                 /* Mutex priority must not already exist    */
        OS_EXIT_CRITICAL();                                  /* Task already exist at priority ...       */
        *err = OS_PRIO_EXIST;                                /* ... inheritance priority                 */
        return ((OS_EVENT *)0);
    }
    OSTCBPrioTbl[prio] = (OS_TCB *)1;                        /* Reserve the table entry                  */
    pevent             = OSEventFreeList;                    /* Get next free event control block        */
    if (pevent == (OS_EVENT *)0) {                           /* See if an ECB was available              */
        OSTCBPrioTbl[prio] = (OS_TCB *)0;                    /* No, Release the table entry              */
        OS_EXIT_CRITICAL();
        *err               = OS_ERR_PEVENT_NULL;             /* No more event control blocks             */
        return (pevent);
    }
    OSEventFreeList     = (OS_EVENT *)OSEventFreeList->OSEventPtr;   /* Adjust the free list            */
    OS_EXIT_CRITICAL();
    pevent->OSEventType = OS_EVENT_TYPE_MUTEX;
    pevent->OSEventCnt  = (prio << 8) | OS_MUTEX_AVAILABLE;/* Resource is available                      */
    pevent->OSEventPtr  = (void *)0;                         /* No task owning the mutex                 */
    OS_EventWaitListInit(pevent);
    *err                = OS_NO_ERR;
    return (pevent);
}
```

Reserve a priority by placing a non-null value

OSMutexDel()

```
OS_EVENT  *OSMutexDel (OS_EVENT *pevent, INT8U opt, INT8U *err) {
    BOOLEAN    tasks_waiting;
    INT8U      pip;
    if (OSIntNesting > 0) {                                  /* See if called from ISR ...                */
        *err = OS_ERR_DEL_ISR;                               /* ... can't DELETE from an ISR              */
        return (pevent);
    }
    OS_ENTER_CRITICAL();
    if (pevent->OSEventGrp != 0x00) {                        /* See if any tasks waiting on mutex         */
        tasks_waiting = TRUE;                                /* Yes                                       */
    } else {
        tasks_waiting = FALSE;                               /* No                                        */
    }

    switch (opt) {
        case OS_DEL_NO_PEND:                                 /* Delete mutex only if no task waiting      */
             if (tasks_waiting == FALSE) {
                 pip                  = (INT8U)(pevent->OSEventCnt >> 8);
                 OSTCBPrioTbl[pip]    = (OS_TCB *)0;         /* Free up the PIP                           */
                 pevent->OSEventType = OS_EVENT_TYPE_UNUSED;
                 pevent->OSEventPtr  = OSEventFreeList;      /* Return Event Control Block to free list   */
                 OSEventFreeList     = pevent;
                 OS_EXIT_CRITICAL();
                 *err = OS_NO_ERR;
                 return ((OS_EVENT *)0);                     /* Mutex has been deleted                    */
             } else {
                 OS_EXIT_CRITICAL();
                 *err = OS_ERR_TASK_WAITING;
                 return (pevent);
             }
```

Notice that to setup the field to UNUSED can prevent user from miss using the kernel object

OSMutexDel()

```
        case OS_DEL_ALWAYS:                                    /* Always delete the mutex                    */
             while (pevent->OSEventGrp != 0x00) {              /* Ready ALL tasks waiting for mutex          */
                 OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MUTEX);
             }
             pip                  = (INT8U)(pevent->OSEventCnt >> 8);
             OSTCBPrioTbl[pip]    = (OS_TCB *)0;               /* Free up the PIP                            */
             pevent->OSEventType = OS_EVENT_TYPE_UNUSED;
             pevent->OSEventPtr  = OSEventFreeList;            /* Return Event Control Block to free list    */
             OSEventFreeList     = pevent;                     /* Get next free event control block          */
             OS_EXIT_CRITICAL();
             if (tasks_waiting == TRUE) {                      /* Reschedule only if task(s) were waiting    */
                 OS_Sched();                                   /* Find highest priority task ready to run    */
             }
             *err = OS_NO_ERR;
             return ((OS_EVENT *)0);                           /* Mutex has been deleted                     */

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

OSMutexPend()

```
void  OSMutexPend (OS_EVENT *pevent, INT16U timeout, INT8U *err) {
    INT8U      pip;                                          /* Priority Inheritance Priority (PIP)      */
    INT8U      mprio;                                        /* Mutex owner priority                     */
    BOOLEAN    rdy;                                          /* Flag indicating task was ready           */
    OS_TCB    *ptcb;
    if (OSIntNesting > 0) {                                  /* See if called from ISR ...               */
        *err = OS_ERR_PEND_ISR;                              /* ... can't PEND from an ISR               */
        return;
    }
    OS_ENTER_CRITICAL();                                                          /* Is Mutex available?
    if ((INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8) == OS_MUTEX_AVAILABLE) {
        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       */
        OS_EXIT_CRITICAL();
        *err  = OS_NO_ERR;
        return;
    }
    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 the mutex is free…

```
    if (ptcb->OSTCBPrio != pip && mprio > OSTCBCur->OSTCBPrio) {  /*     Need to promote prio of owner?*/
        if ((OSRdyTbl[ptcb->OSTCBY] & ptcb->OSTCBBitX) != 0x00) { /*     See if mutex owner is ready   */
                                                                  /*     Yes, Remove owner from Rdy ...*/
                                                                  /*          ... list at current prio */
            if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {
                OSRdyGrp &= ~ptcb->OSTCBBitY;
            }
            rdy = TRUE;
        } else {
            rdy = FALSE;                                          /* No                            */
        }
        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;
    }
    OSTCBCur->OSTCBStat |= OS_STAT_MUTEX;                     /* Mutex not available, pend current task   */
    OSTCBCur->OSTCBDly   = timeout;                           /* Store timeout in current task's 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();
    if (OSTCBCur->OSTCBStat & OS_STAT_MUTEX) {                /* Must have timed out if still waiting for event*/
        OS_EventTO(pevent);
        OS_EXIT_CRITICAL();
        *err = OS_TIMEOUT;                                    /* Indicate that we didn't get mutex within TO   */
        return;
    }
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
}
```

If the owner's priority has not been raised, do it.
Move the owner's ready bit to that of the new priority

No need to move its ready bit if it is not ready

Set the ECB bitmap

OSMutexPost()

```
INT8U  OSMutexPost (OS_EVENT *pevent) {
    INT8U      pip;                                    /* Priority inheritance priority                */
    INT8U      prio;
    if (OSIntNesting > 0) {                            /* See if called from ISR ...                   */
        return (OS_ERR_POST_ISR);                      /* ... can't POST mutex from an ISR             */
    }
    OS_ENTER_CRITICAL();
    pip  = (INT8U)(pevent->OSEventCnt >> 8);           /* Get priority inheritance priority of mutex   */
    prio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);  /* Get owner's original priority      */
    if (OSTCBCur->OSTCBPrio != pip &&
        OSTCBCur->OSTCBPrio != prio) {                 /* See if posting task owns the MUTEX           */
        OS_EXIT_CRITICAL();
        return (OS_ERR_NOT_MUTEX_OWNER);
    }
```

OSMutexPost()

```
    if (OSTCBCur->OSTCBPrio == pip) {                  /* Did we have to raise current task's priority? */
                                                       /* Yes, Return to original priority              */
                                                       /*      Remove owner from ready list at 'pip'    */
        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                           */
    if (pevent->OSEventGrp != 0x00) {                  /* Any task waiting for the mutex?               */
                                                       /* Yes, Make HPT waiting for mutex ready         */
        prio                 = OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MUTEX);
        pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;   /*      Save priority of mutex's new owner       */
        pevent->OSEventCnt |= prio;
        pevent->OSEventPtr  = OSTCBPrioTbl[prio];      /*      Link to mutex owner's OS_TCB             */
        OS_EXIT_CRITICAL();
        OS_Sched();                                    /*      Find highest priority task ready to run  */
        return (OS_NO_ERR);
    }
    pevent->OSEventCnt |= OS_MUTEX_AVAILABLE;          /* No,  Mutex is now available                   */
    pevent->OSEventPtr  = (void *)0;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

Move the ready bit of the current task back to its original priority

OSMutexAccept()

```
INT8U  OSMutexAccept (OS_EVENT *pevent, INT8U *err) {
    if (OSIntNesting > 0) {                                  /* Make sure it's not called from an ISR        */
        *err = OS_ERR_PEND_ISR;
        return (0);
    }
    OS_ENTER_CRITICAL();                                     /* Get value (0 or 1) of Mutex*/
    if ((pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8) == OS_MUTEX_AVAILABLE) {
        pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;     /*      Mask off LSByte (Acquire Mutex)         */
        pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;       /*      Save current task priority in LSByte    */
        pevent->OSEventPtr  = (void *)OSTCBCur;          /*      Link TCB of task owning Mutex           */
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
        return (1);
    }
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
    return (0);
}
```

Summary

- In realistic systems, compromise exists between simplicity and performance
  - [PCP]→[PIP]→[CPP]→[NCSP]
- You must be able to analyze the following when mutex is used
  - Blocking time
  - Number of blocking