Mutual Exclusion Semaphores

 Mutex Utilization

 Mutual Exclusion Semaphores or simply *mutexes* are used by tasks to gain exclusive access to a resource. Mutexes are *binary semaphores* that have additional features beyond the normal semaphores mechanism provided by µC/OS-II.

 A mutex is used by your application code to reduce the priority inversion problem as described in *Real-Time Systems Concepts*. A priority inversion occurs when a low priority task owns a resource needed by a high priority task. In order to reduce priority inversion, the kernel can increase the priority of the low priority task to the priority of the higher priority task until the low priority task is done with the resource.

 In order to implement mutexes, a real-time kernel needs to provide the ability to support multiple tasks at the same priority. Unfortunately, µC/OS-II doesn’t allow multiple tasks at the same priority. However, there is a way around this problem. What if a priority just above the high priority task was *reserved* by the mutex to allow a low priority task to be raised in priority.

 Let’s use an example to illustrate how µC/OS-II mutexes work. Listing 8.1 shows three tasks that may need to access a common resource. To access the resource, each task must pend on the mutex ResourceMutex. Task #1 has the highest priority (10), task #2 has a medium priority (15) and task #3, the lowest (20). An unused priority just above the highest task priority (i.e., priority 9) will be reserved as the *Priority Ceiling Priority (PCP)*.
OS_EVENT *ResourceMutex;
OS_STK TaskPrio10Stk[1000];
OS_STK TaskPrio15Stk[1000];
OS_STK TaskPrio20Stk[1000];

void main (void)
{
    INT8U err;
    
    OSInit();                                                              (1)
    ---------- Application Initialization ----------
    ResourceMutex = OSMutexCreate(9, &err);                                (2)
    OSTaskCreate(TaskPrio10, (void *)0, &TaskPrio10Stk[999], 10);          (3)
    OSTaskCreate(TaskPrio15, (void *)0, &TaskPrio15Stk[999], 15);
    OSTaskCreate(TaskPrio20, (void *)0, &TaskPrio20Stk[999], 20);
    ---------- Application Initialization ----------
    OSStart();                                                             (4)
}

void TaskPrio10 (void *pdata)
{
    INT8U err;
    
    pdata = pdata;
    while (1) {
        --------- Application Code ---------
        OSMutexPend(ResourceMutex, 0, &err);
        --------- Access common resource ---------
        OSMutexPost(ResourceMutex);
        --------- Application Code ---------
    }
}

void TaskPrio15 (void *pdata)
{
    INT8U err;
    
    pdata = pdata;
    while (1) {
        --------- Application Code ---------
        OSMutexPend(ResourceMutex, 0, &err);
        --------- Access common resource ---------
        OSMutexPost(ResourceMutex);
        --------- Application Code ---------
    }
}

void TaskPrio20 (void *pdata)
{
    INT8U err;
    
    pdata = pdata;
    while (1) {
        --------- Application Code ---------
        OSMutexPend(ResourceMutex, 0, &err);
        --------- Access common resource ---------
        OSMutexPost(ResourceMutex);
(1) & (2) As shown in `main()`, µC/OS-II is initialized and a mutex is created by calling `OSMutexCreate()`. You should note that `OSMutexCreate()` is passed the PCP (i.e., 9).

(3) & (4) The three tasks are then created and µC/OS-II is started.

Suppose that this application has been running for a while and that, at some point, task #3 accesses the common resource first and thus acquires the mutex. Task #3 runs for a while and then gets preempted by task #1. Task #1 needs the resource and thus attempts to acquire the mutex (by calling `OSMutexPend()`). In this case, `OSMutexPend()` notices that a higher priority task needs the resource and thus raises the priority of task #3 to 9 which forces a context switch back to task #3. Task #3 will proceed and hopefully release the resource quickly. When done with the resource, task #3 will call `OSMutexPost()` to release the mutex. `OSMutexPost()` will notice that the mutex was owned by a lower priority task that got its priority raised and thus, will return task #3 to it’s original priority. `OSMutexPost()` will notice that a higher priority task (i.e., task #1) needs access to the resource and will give the resource to task #1 and perform a context switch to task #1.

µC/OS-II’s mutexes consist of three elements: a flag indicating whether the mutex is available (0 or 1), a priority to assign the task that owns the mutex in case a higher priority task attempts to gain access to the mutex, and a list of tasks waiting for the mutex.

µC/OS-II provides six services to access mutexes: `OSMutexCreate()`, `OSMutexDel()`, `OSMutexPend()`, `OSMutexPost()`, `OSMutexAccept()` and `OSMutexQuery()`.

To enable µC/OS-II mutex services, you must set the configuration constants in `OS_CFG.H`. Specifically, table 8.1 shows which services are compiled based on the value of configuration constants found in `OS_CFG.H`. You should note that NONE of the mailbox services are enabled when `OS_MUTEX_EN` is set to 0. To enable specific features (i.e., service) listed in Table 8.1, simply set the configuration constant to 1. You will notice that `OSMutexCreate()`, `OSMutexPend()` and `OSMutexPost()` cannot be individually disabled like the other services. That’s because they are always needed when you enable µC/OS-II mutual exclusion semaphore management.
<table>
<thead>
<tr>
<th>µC/OS-II Mutex Service</th>
<th>Enabled when set to 1 in OS_CFG.H</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSMutexAccept ()</td>
<td>OS_MUTEX_ACCEPT_EN</td>
</tr>
<tr>
<td>OSMutexCreate ()</td>
<td></td>
</tr>
<tr>
<td>OSMutexDel ()</td>
<td>OS_MUTEX_DEL_EN</td>
</tr>
<tr>
<td>OSMutexPend ()</td>
<td></td>
</tr>
<tr>
<td>OSMutexPost ()</td>
<td></td>
</tr>
<tr>
<td>OSMutexQuery ()</td>
<td>OS_MUTEX_QUERY_EN</td>
</tr>
</tbody>
</table>

Table - Table 8.1 Mutex configuration constants in OS_CFG.H

Figure 8.1 shows a flow diagram to illustrate the relationship between tasks and a mutex. A mutex can only be accessed by tasks. Note that the symbology used to represent a mutex is a ‘key’. The ‘key’ symbology shows that the mutex is used to access shared resources.

Creating a Mutex, OSMutexCreate()

A mutex needs to be created before it can be used. Creating a mutex is accomplished by calling `OSMutexCreate()`. The initial value of a mutex is always set to 1 indicating that the resource is available. The code to create a mutex is shown in listing 8.2.
OS_MUTEX *OSMutexCreate (INT8U prio, INT8U *err) 
{
#if OS_CRITICAL_METHOD == 3
OS_CPU_SR cpu_sr;
#endif
OS_MUTEX *pevent;

if (OSIntNesting > 0) {                                  (1)
  *err = OS_ERR_CREATE_ISR;
  return ((OS_MUTEX *)0);
}
#if OS_ARG_CHK_EN
if (prio >= OS_LOWEST_PRIO) {                            (2)
  *err = OS_PRIO_INVALID;
  return ((OS_MUTEX *)0);
}
#endif
OS_ENTER_CRITICAL();
if (OSTCBPrioTbl[prio] != (OS_TCB *)0) {                 (3)
  *err = OS_PRIO_EXIST;
  OS_EXIT_CRITICAL();
  return ((OS_MUTEX *)0);
}
OSTCBPrioTbl[prio] = (OS_TCB *)1;                        (4)
pevent             = OSEventFreeList;                    (5)
if (pevent == (OS_MUTEX *)0) {
  OSTCBPrioTbl[prio] = (OS_TCB *)0;
  OS_EXIT_CRITICAL();
  *err               = OS_ERR_PEVENT_NULL;
  return (pevent);
}
OSEventFreeList     = (OS_MUTEX *)OSEventFreeList->OSEventPtr;    (6)
OS_EXIT_CRITICAL();
pevent->OSEventType = OS_EVENT_TYPE_MUTEX;               (7)
pevent->OSEventCnt  = (prio << 8) | OS_MUTEX_AVAILABLE;  (8)
pevent->OSEventPtr  = (void *)0;                         (9)
OSEventWaitListInit(pevent);                            (10)
*err                = OS_NO_ERR;
return (pevent);                                        (11)
}

Listing - Listing 8.2, Creating a mutex.

(1) OSMutexCreate() starts by making sure it’s not called from an ISR because that’s not allowed.

(2) OSMutexCreate() then verifies that the PIP is within valid ranged based on what you determined the lowest priority is for your application as specified in OS_CFG.H.

(3) OSMutexCreate() then checks to see that there isn’t already a task assigned to the PIP. A NULL pointer in OSTCBPrioTbl[] indicates for the Priority Inheritance Priority
(PIP) is available.

(4) If an entry is available, `OSMutexCreate()` reserves the priority by placing a non-NULL pointer in `OSTCBPrioTbl[prio]`. This will prevent you from being able to use this priority to create other tasks or other mutexes using this priority.

(5) `OSMutexCreate()` then attempts to obtain an ECB (Event Control Block) from the free list of ECBs.

(6) The linked list of free ECBs is adjusted to point to the next free ECB.

(7) If there was an ECB available, the ECB type is set to `OS_EVENT_TYPE_MUTEX`. Other μC/OS-II services will check this field to make sure that the ECB is of the proper type. This prevents you from calling `OSMutexPost()` on an ECB that was created for use as a message mailbox, for example.

(8) `OSMutexCreate()` then set the mutex value to ‘available’ and the PIP is stored.

It is worth noting that the `.OSEventCnt` field is used differently. Specifically, the upper 8 bits of `.OSEventCnt` are used to hold the PIP and the lower 8 bits are used to hold either the value of the mutex when the resource is available (0xFF) or, the priority of the task that ‘owns’ the mutex (a value between 0 and 62). This prevents having to add extra fields in an `OS_EVENT` structure and thus reduces the amount of RAM needed by μC/OS-II.

(9) Because the mutex is being initialized, there are no tasks waiting for it.

(10) The wait list is then initialized by calling `OSEventWaitListInit()`.

(11) Finally, `OSMutexCreate()` returns a pointer to the ECB. This pointer MUST be used in subsequent calls to manipulate mutexes (`OSMutexPend()`, `OSMutexPost()`, `OSMutexAccept()`, `OSMutexDel()` and `OSMutexQuery()`). The pointer is basically used as the mutex’s handle. If there were no more ECBs, `OSMutexCreate()` would have returned a NULL pointer.

Figure 8.2 shows the ECB just before returning from `OSMutexCreate()`.
Deleting a Mutex, OSMutexDel()

The code to delete a mutex is shown in listing 8.3 and this service is available only if OS_MUTEX_DEL_EN is set to 1 in OS_CFG.H. This is a dangerous function to use because multiple tasks could attempt to access a deleted mutex. You should always use this function with great care. Generally speaking, before you would delete a mutex, you should first delete all the tasks that can access the mutex.
OS_EVENT  *OSMutexDel (OS_EVENT *pevent, INT8U opt, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
OS_CPU_SR  cpu_sr;
#endif
BOOLEAN    tasks_waiting;

if (OSIntNesting > 0) {                                   (1)
    *err = OS_ERR_DEL_ISR;
    return (pevent);
}
#if OS_ARG_CHK_EN
if (pevent == (OS_EVENT *)0) {                            (2)
    *err = OS_ERR_PEVENT_NULL;
    return (pevent);
}
if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {         (3)
    OS_EXIT_CRITICAL();
    *err = OS_ERR_EVENT_TYPE;
    return (pevent);
}
#endif
OS_ENTER_CRITICAL();
if (pevent->OSEventGrp != 0x00) {                         (4)
    tasks_waiting = TRUE;
} else {
    tasks_waiting = FALSE;
}
switch (opt) {
    case OS_DEL_NO_PEND:                                  (9)
        if (tasks_waiting == FALSE) {                    (5)
            pevent->OSEventType = OS_EVENT_TYPE_UNUSED;  (6)
            pevent->OSEventPtr = OSEventFreeList;       (7)
            OSEventFreeList = pevent;
            OS_EXIT_CRITICAL();
            *err = OS_NO_ERR;
            return ((OS_EVENT *)0);                      (8)
        } else {
            OS_EXIT_CRITICAL();
            *err = OS_ERR_TASK_WAITING;
            return (pevent);
        }
        break;
    case OS_DEL_ALWAYS:                                  (14)
        while (pevent->OSEventGrp != 0x00) {           (10)
            OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MUTEX);
        }
        pevent->OSEventType = OS_EVENT_TYPE_UNUSED;    (11)
        pevent->OSEventPtr = OSEventFreeList;          (12)
        OSEventFreeList = pevent;
        OS_EXIT_CRITICAL();
        if (tasks_waiting == TRUE) {                   (13)
            OS_Sched();
        }
        *err = OS_NO_ERR;
        return ((OS_EVENT *)0);                      (14)
    default:                                             (14)
        OS_EXIT_CRITICAL();
        *err = OS_ERR_INVALID_OPT;
        return (pevent);
}
}
Listing - Listing 8.3, Deleting a mutex.

(1) OSMutexDel() starts by making sure that this function is not called from an ISR because that’s not allowed.

(2)

(3) We then check the arguments passed to it—pevent cannot be a NULL pointer and pevent needs to point to a mutex.

(4) OSMutexDel() then determines whether there are any tasks waiting on the mutex. The flag tasks_waiting is set accordingly.

Based on the option (i.e., opt) specified in the call, OSMutexDel() will either delete the mutex only if no tasks are pending on the mutex (opt == OS_DEL_NO_PEND) or, delete the mutex even if tasks are waiting (opt == OS_DEL_ALWAYS).

(5)

(6)

(7) When opt is set to OS_DEL_NO_PEND and there is no task waiting on the mutex, OSMutexDel() marks the ECB as unused and the ECB is returned to the free list of ECBs. This will allow another mutex (or any other ECB based object) to be created. You will note that OSMutexDel() returns a NULL pointer L8.3(8) since, at this point, the mutex should no longer be accessed through the original pointer.

(9)

(10) When opt is set to OS_DEL_ALWAYS then all tasks waiting on the mutex will be readied. Each task will think it has access to the mutex. Of course, that’s a dangerous outcome since the whole point of having a mutex is to protect against multiple access of a resource. Again, you should delete all the tasks that can access the mutex before you delete the mutex.

(11)

(12) Once all pending tasks are readied, OSMutexDel() marks the ECB as unused and
the ECB is returned to the free list of ECBs.

(13) The scheduler is called only if there were tasks waiting on the mutex.

(14) You will note that OSMutexDel() returns a NULL pointer since, at this point, the mutex should no longer be accessed through the original pointer.

**Waiting on a Mutex (blocking), OSMutexPend()**

The code to wait on a mutex is shown in listing 8.4.
void OSMutexPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
#if OS_CRITICAL_METHOD == 3
  OS_CPU_SR cpu_sr;
#endif
INT8U pip;
INT8U mprrio;
BOOLEAN rdy;
OS_TCB *ptcb;

if (OSIntNesting > 0) {                                 (1)
  *err = OS_ERR_PEND_ISR;
  return;
}
#endif
#if OS_ARG_CHK_EN
if (pevent == (OS_EVENT *)0) {                           (2)
  *err = OS_ERR_PEVENT_NULL;
  return;
}
#endif
OS_ENTER_CRITICAL();
#if OS_ARG_CHK_EN
if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {        (3)
  OS_EXIT_CRITICAL();
  *err = OS_ERR_EVENT_TYPE;
  return;
}
#endif (4)
if ((INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8) == OS_MUTEX_AVAILABLE) {
  pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;         (5)
  pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;           (6)
  pevent->OSEventPtr  = (void *)OSTCBCur;              (7)
  OS_EXIT_CRITICAL();
  *err  = OS_NO_ERR;
  return;
}

pip   = (INT8U)(pevent->OSEventCnt >> 8);                       (8)
mprio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);    (9)
ptcb  = (OS_TCB *)(pevent->OSEventPtr);                        (10)
if (ptcb->OSTCBPrio != pip && mprrio > OSTCBCur->OSTCBPrio) {   (11)
  if ((OSRdyTbl[ptcb->OSTCBY] & ptcb->OSTCBBitX) != 0x00) {  (12)
    if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0x00) {
      OSRdyGrp &= ~ptcb->OSTCBBitY;
    } else {
      rdy = FALSE;                                           (15)
    }
  }
  OSTCBPrioTbl[pip]       = (OS_TCB *)ptcb;
  rdy = TRUE;                                              (14)
} else {
  rdy = FALSE;
}
ptcb->OSTCBPrio         = pip;                                 (16)
ptcb->OSTCBY            = ptcb->OSTCBPrio >> 3;
ptcb->OSTCBBitY         = OSMapTbl[ptcb->OSTCBY];
ptcb->OSTCBX            = ptcb->OSTCBPrio & 0x07;
ptcb->OSTCBBitX         = OSMapTbl[ptcb->OSTCBX];
if (rdy == TRUE) {                                             (17)
  OSRdyGrp |= ptcb->OSTCBBitY;
  OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
}
OSTCBPrioTbl[pip]       = (OS_TCB *)ptcb;
Like all µC/OS-II pend calls, OSMutexPend() cannot be called from an ISR and thus, OSMutexPend() checks for this condition first.

Assuming that the configuration constant OS_ARG_CHK_EN is set to 1, OSMutexPend() makes sure that the ‘handle’ pevent is not a NULL pointer and that the ECB being pointed to has been created by OSMutexCreate().

The mutex is available if the lower 8 bits of .OSEventCnt are set to 0xFF (i.e., OS_MUTEX_AVAILABLE). If this is the case, OSMutexPend() will grant the mutex to the calling task and, OSMutexPend() will set the lower 8 bits of .OSEventCnt to the calling’s task priority.

OSMutexPend() then sets .OSEventPtr to point to the TCB of the calling task and returns. At this point the caller can proceed with accessing the resource since the return error code is set to OS_NO_ERR. Obviously, if you want the mutex, this is the outcome you are looking for. This also happens to be the fastest (normal) path through OSMutexPend().

If the mutex is owned by another task, the calling task needs to be put to sleep until the other task relinquishes the mutex (see OSMutexPost()). OSMutexPend() allows you to specify a timeout value as one of its arguments (i.e., timeout). This feature is useful to avoid waiting indefinitely for the mutex. If the value passed is non-zero, then
OSMutexPend() will suspend the task until the mutex is signaled or the specified timeout period expires. Note that a timeout value of 0 indicates that the task is willing to wait forever for the mutex to be signaled.

(8) & (9) & (10) Before the calling task is put to sleep, OSMutexPend() extracts the PIP of the mutex, the priority of the task that owns the mutex and a pointer to the TCB of the task that owns the mutex.

(11) If the owner’s priority is lower (a higher number) than the task that calls OSMutexPend() then, the priority of the task that owns the mutex will be raised to the mutex’s priority inheritance priority (PIP). This will allow the owner to relinquish the mutex sooner.

(12) OSMutexPend() then determines if the task that owns the mutex is ready-to-run.

(13) & (14) If it is, that task will be made no longer ready-to-run at the the owner’s priority and the flag rdy will be set indicating that the mutex owner was ready-to-run.

(15) If the task was not ready-to-run, rdy is set accordingly. The reason the flag is set is to determine whether we need to make the task ready-to-run at the new, higher priority (i.e., at the PIP).

(16) OSMutexPend() then computes TCB (Task Control Block) elements at the PIP. You should note that I could have saved this information in the OS_EVENT data structure when the mutex was created in order to save processing time. However, this would have meant additional RAM for each OS_EVENT instantiation.

(17) From this information and the state of the rdy flag, we determine whether the mutex owner needs to be made ready-to-run at the PIP.

(18) To put the calling task to sleep, OSMutexPend() sets the status flag in the task’s TCB to indicate that the task is suspended waiting for a mutex.

(19) The timeout is also stored in the TCB so that it can be decremented by OSTimeTick(). You should recall that OSTimeTick() decrements each of the created tasks OSTCBDly fields if they are non-zero.

(20) The actual work of putting the task to sleep is done by OS_EventTaskWait().
(21) Because the calling task is no longer ready-to-run, the scheduler is called to run the next highest priority task that is ready-to-run.

When the mutex is signaled (or the timeout period expires) and the task that called `OSMutexPend()` is again the highest priority task, `OS_Sched()` returns.

(22) `OSMutexPend()` then checks to see if the TCB’s status flag is still set to indicate that the task is waiting for the mutex. If the task is still waiting for the mutex then it must not have been signaled by an `OSMutexPost()` call. Indeed, the task must have be readied by `OSTimeTick()` indicating that the timeout period has expired.

(23) & (24) In this case, the task is removed from the wait list for the mutex by calling `OS_EventTO()`, and an error code is returned to the task that called `OSMutexPend()` to indicate that a timeout occurred.

If the status flag in the task’s TCB doesn’t have the `OS_STAT_MUTEX` bit set then the mutex must have been signaled and the task that called `OSMutexPend()` can now conclude that it has the mutex.

(25) Finally, the link to the ECB is removed.

**Signaling a Mutex, OSMutexPost()**

The code to signal a mutex is shown in listing 8.5.
INT8U OSMutexPost (OS_EVENT *pevent)
{
    #if OS_CRITICAL_METHOD == 3
        OS_CPU_SR cpu_sr;
    #endif
    INT8U    pip;
    INT8U    prio;

    if (OSIntNesting > 0) {                                     (1)
        return (OS_ERR_POST_ISR);
    }
    #if OS_ARG_CHK_EN
    if (pevent == (OS_EVENT *)0) {                              (2)
        return (OS_ERR_PEVENT_NULL);
    }
    #endif
    OS_ENTER_CRITICAL();
    pip  = (INT8U)(pevent->OSEventCnt >> 8);
Prio = (INT8U)(pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8);
    #if OS_ARG_CHK_EN
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {          (3)
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (OSTCBCur->OSTCBPrio != pip ||
Prio != prio) {                                               (4)
        OS_EXIT_CRITICAL();
        return (OS_ERR_NOT_MUTEX_OWNER);
    }
    #endif
    if (OSTCBCur->OSTCBPrio == pip) {                             (5)
    
        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;
    }
    OSRdyTbl[pip] = (OS_TCB *)1;
    if (pevent->OSEventGrp != 0x00) {                            (6)
        prio                = OS_EventTaskRdy(pevent, (void *)0, OS_STAT_MUTEX);
        pevent->OSEventCnt &= 0xFF00;                   (9)
        pevent->OSEventCnt |= prio;
        pevent->OSEventPtr  = OSTCBPrioTbl[prio];
        OS_EXIT_CRITICAL();
        OS_Sched();                                    (10)
        return (OS_NO_ERR);
    }
    pevent->OSEventCnt |= 0x00FF;                      (11)
    pevent->OSEventPtr  = (void *)0;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
(1) Mutual exclusion semaphores must only be used by tasks and thus, a check is performed to make sure that `OSMutexPost()` is not called from an ISR.

(2) & (3) Assuming that the configuration constant `OS_ARG_CHK_EN` is set to 1, `OSMutexPost()` checks that the ‘handle’ pointer is not a NULL pointer and that the ECB being pointed to has been created by `OSMutexCreate()`.

(4) `OSMutexPost()` makes sure that the task that is signaling the mutex actually owns the mutex. The owner’s priority must either be set to the pip (an `OSMutexPend()` could have raised the owner’s priority) or the priority stored in the mutex itself.

(5) `OSMutexPost()` then checks to see if the priority of the mutex owner had to be raised to the PIP because a higher priority task attempted to access the mutex. In this case, the priority of the owner is reduced back to its original value. The original task priority is extracted from the lower 8 bits of `OSEventCnt`.

(6) The calling task is removed from the ready list at the PIP and placed in the ready list at the task’s original priority. Note that the TCB fields are recomputed for the original task priority.

(7) Next, we check to see if any tasks are waiting on the mutex. There are tasks waiting when the `.OSEventGrp` field in the ECB contains a non-zero value.

(8) The highest priority task waiting for the mutex will be removed from the wait list by `OS_EventTaskRdy()` (see section 6.02, *Making a task ready, OS_EventTaskRdy()*) and this task will be made ready-to-run.

(9) The priority of the new owner is saved in the mutex’s ECB.

(10) `OS_Sched()` is then called to see if the task made ready is now the highest priority task ready-to-run. If it is, a context switch will result and the readied task will be executed. If the readied task is not the highest priority task then `OS_Sched()` will return and the task that called `OSMutexPost()` will continue execution.

(11) If there were no tasks waiting on the mutex, the lower 8 bits of `.OSEventCnt` would be set to 0xFF indicating that the mutex is immediately available.
Getting a Mutex without waiting (non-blocking), OSMutexAccept()

It is possible to obtain a mutex without putting a task to sleep if the mutex is not available. This is accomplished by calling `OSMutexAccept()` and the code for this function is shown in listing 8.6.

```c
INT8U OSMutexAccept (OS_EVENT *pevent, INT8U *err) {  
#if OS_CRITICAL_METHOD == 3
    OS_CPU_SR cpu_sr;
#endif
    if (OSIntNesting > 0) {  
        *err = OS_ERR_PEND_ISR;
        return (0);
    }
#if OS_ARG_CHK_EN
    if (pevent == (OS_EVENT *)0) {
        *err = OS_ERR_PEVENT_NULL;
        return (0);
    }
#endif
    OS_ENTER_CRITICAL();
#if OS_ARG_CHK_EN
    if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return (0);
    }
#endif
    OS_ENTER_CRITICAL();
    if ((pevent->OSEventCnt & OS_MUTEX_KEEP_LOWER_8) == OS_MUTEX_AVAILABLE) {
        pevent->OSEventCnt &= OS_MUTEX_KEEP_UPPER_8;    
        pevent->OSEventCnt |= OSTCBCur->OSTCBPrio;
        pevent->OSEventPtr  = (void *)OSTCBCur;         
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
        return (1);
    }
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
    return (0);
}
```

**Listing - Listing 8.6, Getting a mutex without waiting.**

(1) As with the other calls, if `OS_ARG_CHK_EN` is set to 1 in `OS_CFG.H`, `OSMutexAccept()` start by ensuring that it’s not called from and ISR and performs boundary checks.
(2) `OSMutexAccept()` then checks to see if the mutex is available (the lower 8 bits of `.OSEventCnt` would be set to 0xFF).

(3) & (4) If the mutex is available, `OSMutexAccept()` would acquire the mutex by writing the priority of the mutex owner in the lower 8 bits of `.OSEventCnt` and by linking the the owner’s TCB.

The code that called `OSMutexAccept()` will need to examine the returned value. A returned value of 0 indicates that the mutex was not available while a return value of 1 indicates that the mutex was available and the caller can access the resource.

**Obtaining the status of a mutex, `OSMutexQuery()`**

`OSMutexQuery()` allows your application to take a ‘snapshot’ of an ECB that is used as a mutex. The code for this function is shown in listing 8.7.
```c
INT8U OSMutexQuery (OS_EVENT *pevent, OS_MUTEX_DATA *pdata)
{
    #if OS_CRITICAL_METHOD == 3
        OS_CPU_SR cpu_sr;
    #endif
    INT8U *psrc;
    INT8U *pdest;

    if (OSIntNesting > 0) {                                 (1)
        return (OS_ERR_QUERY_ISR);
    }
    #if OS_ARG_CHK_EN
        if (pevent == (OS_EVENT *)0) {                          (2)
            return (OS_ERR_PEVENT_NULL);
        }
        OS_ENTER_CRITICAL();
        #if OS_ARG_CHK_EN
            if (pevent->OSEventType != OS_EVENT_TYPE_MUTEX) {       (3)
                OS_EXIT_CRITICAL();
                return (OS_ERR_EVENT_TYPE);
            }
        #endif
        pdata->OSMutexPIP  = (INT8U)(pevent->OSEventCnt >> 8);  (4)
        pdata->OSOwnerPrio = (INT8U)(pevent->OSEventCnt & 0x00FF);
        if (pdata->OSOwnerPrio == 0xFF) {
            pdata->OSValue = 1;                                 (5)
        } else {
            pdata->OSValue = 0;                                 (6)
        }
        pdata->OSEventGrp  = pevent->OSEventGrp;                (7)
        psrc               = &pevent->OSEventTbl[0];
        pdest              = &pdata->OSEventTbl[0];
        #if OS_EVENT_TBL_SIZE > 0
            *pdest++           = *psrc++;
        #endif
        #if OS_EVENT_TBL_SIZE > 1
            *pdest++           = *psrc++;
        #endif
        #if OS_EVENT_TBL_SIZE > 2
            *pdest++           = *psrc++;
        #endif
        #if OS_EVENT_TBL_SIZE > 3
            *pdest++           = *psrc++;
        #endif
        #if OS_EVENT_TBL_SIZE > 4
            *pdest++           = *psrc++;
        #endif
        #if OS_EVENT_TBL_SIZE > 5
            *pdest++           = *psrc++;
        #endif
        #if OS_EVENT_TBL_SIZE > 6
            *pdest++           = *psrc++;
        #endif
        #if OS_EVENT_TBL_SIZE > 7
            *pdest             = *psrc;
        #endif
```

As with all mutex calls, `OSMutexQuery()` determines whether the call is made from an ISR.

(2) & (3) If the configuration constant `OS_ARG_CHK_EN` is set to 1, `OSMutexQuery()` checks that the ‘handle’ `pevent` is not a NULL pointer and that the ECB being pointed to has been created by `OSMutexCreate()`. `OSMutexQuery()` then loads the `OS_MUTEX_DATA` structure with the appropriate fields.

(4) First, we extract the Priority Inheritance Priority (PIP) from the upper 8 bits of the `.OSEventCnt` field of the mutex.

(5) Next, we obtain the mutex value from the lower 8 bits of the `.OSEventCnt` field of the mutex. If the mutex is available (i.e., lower 8 bits set to 0xFF) then the mutex value is assumed to be 1.

(6) Otherwise, the mutex value is 0 (i.e., unavailable because it’s owned by a task).

(7) Finally, the mutex wait list is copied into the appropriate fields in `OS_MUTEX_DATA`. For performance reasons, I decided to use inline code instead of using a for loop.

`OSMutexQuery()` is passed two arguments: `pevent` contains a pointer to the mutex which is returned by `OSMutexCreate()` when the mutex is created and, `pdata` which is a pointer to a data structure (`OS_MUTEX_DATA`, see `uCOS_II.H`) that will hold information about the mutex. Your application will thus need to allocate a variable of type `OS_MUTEX_DATA` that will be used to receive the information about the desired mutex. I decided to use a new data structure because the caller should only be concerned with mutex specific data as opposed to the more generic `OS_EVENT` data structure. `OS_MUTEX_DATA` contains the mutex PIP (Priority Inheritance Priority) (`.OSMutexPIP`), the priority of the task owning the mutex (`.OSMutexPrio`) and the value of the mutex (`.OSMutexValue`) which is set to 1 when the mutex is available and 0 if it’s
not. Note that .OSMutexPrio contains 0xFF if no task owns the mutex. Finally, 
OSEventGrp).