

NANYANG TECHNOLOGICAL UNIVERSITY

SCE13-0026

Micrium μ C/OS-III++

Submitted in Partial Fulfillment of the Requirements for the
Degree of Bachelor of Computer Engineering of the Nanyang
Technological University

by

Do Nhat Minh

School of Computer Engineering

March 28, 2014

Abstract

Real-time operating systems play an important role in time and safety-critical software systems used in many fields, such as avionics, automotives and defense applications. $\mu\text{C}/\text{OS-III}$ is an open-source real-time operating system which aims to be used on embedded devices with restricted resources. $\mu\text{C}/\text{OS-III}$ has many useful features; however, it does not have better implementations proven in more recent advances in real-time systems research.

In this thesis, Earliest Deadline First scheduling algorithm and Stack Resource Policy for time-guaranteed resource sharing were implemented.

Acknowledgements

I am particularly grateful to my supervisor, Dr. Arvind Easwaran from the School of Computer Engineering, for being extremely supportive of my project choice and for giving invaluable advice during the course of this final year project.

I would like to thank Tran Ngoc Khanh Thy for the continued emotional support during the course of this final year project.

Contents

1	Introduction	5
2	Background	6
2.1	Scheduling Algorithms	6
2.1.1	Rate Monotonic Scheduling	6
2.1.2	Utilization Analysis	7
2.1.3	Utilization Analysis of RMS	7
2.1.4	Earliest Deadline First Scheduling	8
2.1.5	Utilization Analysis for EDF	8
2.2	Resource Sharing with Mutual Exclusion	8
2.2.1	Priority Inversion	8
2.2.2	Priority Inheritance Protocol	10
2.2.3	Deadlock	11
2.2.4	Priority Ceiling Protocol	12
3	Overview of Micrium μC/OS-III	13
3.1	Task Model	13
3.2	Scheduling Algorithm	13
3.3	Resource Sharing	14
4	Modifications	15
4.1	Earliest Deadline First Scheduling	15
4.1.1	Notes	15
4.2	Priority Ceiling Protocol	16
4.2.1	OSMutexPend()	16

4.2.2	OSMutexPost()	22
-------	---------------	----

List of Figures

2.1	Priority Inversion	9
2.2	Priority Inheritance Protocol	10
2.3	Deadlock - Resource Allocation Graph	11
2.4	Deadlock	11

Chapter 1

Introduction

An operating system is a collection of software, or software components, which can be characterized as serving the following purposes, (1) interfacing with the underlying hardware to provide convenient abstractions for application programmers, and (2) managing the programs running on the system so that misbehaving programs do not impede others (Witchel, E. , 2009).

A real-time operating system is an operating system which must adhere to a real-time constraint. In such a system, the timeliness of the results from programs are as important as the correctness of such solutions. The system risks catastrophic failures if deadlines are missed. Some important applications for real-time operating systems are in avionics and automotives, where missing task deadlines leads to lives lost.

μ C/OS-III is an open-source priority-based pre-emptive real-time multitasking operating system for embedded systems. It has many useful features which aims to cut development time (Labrosse, J. J. , 2010, pp. 27-31).

However, μ C/OS-III does not implement deadline management but defer this task to the programmer. Moreover, μ C/OS-III has no facility to specify nor keep track of the deadlines for running tasks of the system.

The goal of this thesis is to implement better algorithms for task scheduling and resource sharing for μ C/OS-III. For task scheduling, Earliest Deadline First scheduling is implemented, and for resource sharing, Stack Resource Policy is implemented.

Chapter 2

Background

2.1 Scheduling Algorithms

A scheduling algorithm is an algorithm to determine the ordering of task executions in order to maximize resource utilization, while satisfying safety and correctness (Liu, C. L. and Layland, J. W. , 1973).

A real-time system consists of a number of tasks, each of which has a deadline and a period. Tasks must be completed before their deadlines or risk catastrophic consequences, e.g. a plane might crash if the task that sends sensor failure status misses deadlines. Task period is the interval between release times of instances of a task. Request rate is the reciprocal of task period. A task set is a collection of tasks to be scheduled. A task set is feasible when there exists an ordering where no deadline is missed.

2.1.1 Rate Monotonic Scheduling

Rate Monotonic Scheduling belongs to the class of fixed priority scheduling algorithms. A fixed priority scheduling algorithm is a scheduling algorithm where task orderings are based on statically assigned priorities of the tasks. The rate monotonic priority assignment assumes that a task with higher request rate is assigned a higher priority. C. L. Liu and J. W. Layland in their seminal paper have proven that for fixed priority scheduling, if there exists a feasible assignment, the

rate monotonic assignment is also feasible (Liu, C. L. and Layland, J. W. , 1973). In other words, rate monotonic scheduling algorithm (RMS) is optimal.

2.1.2 Utilization Analysis

CPU utilization for a task is the ratio between the time spent in execution and its period. CPU utilization for a task set is the summation of CPU utilization of each task in the set.

$$U = \sum_{i=1}^n \frac{C_i}{T_i}$$

where C_i is execution time and T_i is period for task i , n is the number of tasks in the task set and U is CPU utilization.

2.1.3 Utilization Analysis of RMS

It is proven that CPU utilization for a task set in RMS must be kept below an upper bound in order to guarantee feasibility. This upper bound is

$$U_{RMS} \leq n(2^{\frac{1}{n}} - 1)$$

where n is the number of tasks in that task set and U_{RMS} is CPU utilization (Liu, C. L. and Layland, J. W. , 1973).

As n tends towards infinity, this expression will tend towards

$$\lim_{n \rightarrow \infty} n(2^{\frac{1}{n}} - 1) = \ln 2 \approx 0.693147 \dots$$

As a rule of thumb, a task set is feasible under fixed priority scheduling when its CPU utilization is below 69.3%. The other 30.7% of CPU time can be reserved for other non real-time tasks.

However, this upper bound is pessimistic. It has been shown that a randomly generated task set will meet all deadlines when utilization is below 85% if exact task deadlines and periods are known, which might be difficult to achieve (Lehoczky, J.; Sha, L. and Ding, Y. , 1989).

2.1.4 Earliest Deadline First Scheduling

Earliest Deadline First Scheduling (EDF) is a scheduling algorithm where task orderings are based on deadlines of task instances. Under this scheme, an instance of a task is assigned highest priority if its deadline is nearest, while an instance of a task with a deadline that is farthest is assigned the lowest priority.

2.1.5 Utilization Analysis for EDF

The utilization bound for EDF is

$$U_{EDF} \leq 1$$

where U_{EDF} is the CPU utilization (Liu, C. L. and Layland, J. W. , 1973).

From the utilization bounds for RMS and EDF, it is trivial to see that EDF guarantees all deadlines of a task set at a higher load than RMS. Therefore, EDF is more desirable from a resource utilization standpoint.

2.2 Resource Sharing with Mutual Exclusion

In order to maximize utility of resources in computer systems, resources are shared among the tasks that needs them. These resources include hardware such as printers or software such as a region of computer memory. However, some resources must only be accessed by one task at a time. Examples include printers where unprotected, concurrent access to the print queue will result in sentences from different documents interleaving each other.

Mutual exclusion, or mutex, is a technique to protect such resources from concurrent access. Mutex was first identified and an implementation for which was first introduced by Dijkstra in his seminal 1965 paper (Dijkstra, E. W. , 1965).

2.2.1 Priority Inversion

In the context of real-time systems, there is a problem of priority inversion in the use of mutexes for resource sharing. Priority inversion occurs when a high

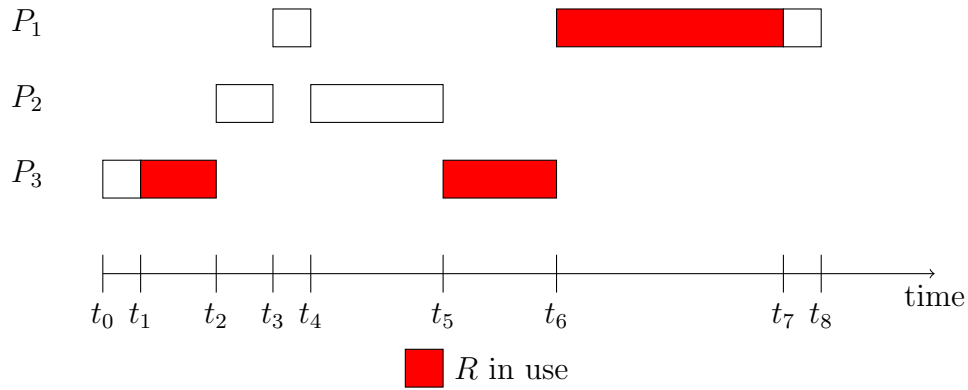


Figure 2.1: Priority Inversion

priority task wants to access a shared resource and is blocked by a lower priority task holding the mutex for that resource.

Figure 2.1 shows an example of priority inversion. In this example, P_1 has highest priority, while P_3 's priority is the lowest. At t_0 , P_3 begins execution. At t_1 , P_3 acquires the mutex for resource R . At t_2 , P_3 is preempted by P_2 . At t_3 , P_1 preempts P_2 and then at t_4 tries to acquire the mutex for R which blocks P_1 and P_2 resumes execution. P_2 finishes execution at t_5 . P_3 then resumes execution and releases the mutex for R and finishes execution at t_6 . P_1 then resumes execution, having successfully acquired mutex for R . At t_7 , P_1 releases the mutex for R and finishes execution at t_8 .

In the above example, the highest priority task, namely P_1 , is preempted by a lower priority task, namely P_2 . It is easy to see that this example can be expanded to include multiple medium priority tasks, where the highest priority task is preempted consecutively by those tasks. In that case, the highest priority task may even miss its deadline because of the chain blocking of those lower priority tasks.

A real-life example of priority inversion is the incident of Mars Pathfinder spacecraft. There were many software tasks running on the Mars Pathfinder's VxWorks real-time operating system. The high priority task `bc_dist` was blocked by the much lower priority task `ASI/MET`, which in turn was blocked by other medium priority tasks. `bc_dist` therefore missed its deadline, which was before

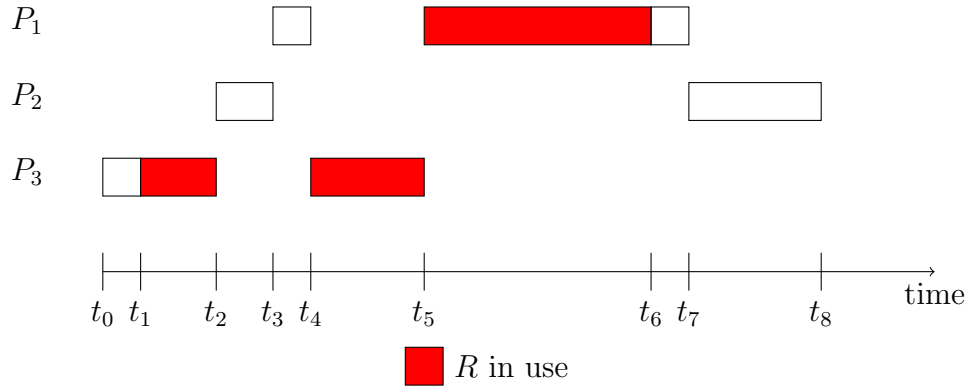


Figure 2.2: Priority Inheritance Protocol

the execution of the `bc_sched` task. The software on Mars Pathfinder dealt with this missed deadline by rebooting itself. Although no catastrophe resulted, the rest of the activities for that day was only accomplished the next day (Reeves, G. E. , 1997).

2.2.2 Priority Inheritance Protocol

In order to solve the problem of Priority Inversion, Lui S. et. al. proposed a class of Priority Inheritance Protocols in their seminal 1990 paper (Lui S.; Rajkumar, R.; Lehoczky, J.P. , 1990). The basic idea for Priority Inheritance Protocols is when a lower priority task blocks a higher priority task inside its critical section, it is promoted to a higher priority for the duration of its critical section.

Figure 2.2 shows how priority inheritance can help mitigate the problem of priority inversion. The task set illustrated is the same as in figure 2.1. The tasks are scheduled the same way as in figure 2.1 from t_0 to t_4 . However, when P_1 tries to take the mutex for R and is blocked by P_3 , P_3 's priority is raised to be the same as P_1 's. Thus, P_2 is no longer able to preempt P_3 and P_3 can run until it releases the mutex for R and finishes at t_5 . As P_1 's priority is higher than P_2 's, it is scheduled to run at t_5 . P_1 releases the mutex for R at t_6 and finishes at t_7 , as which point P_2 is scheduled to run till finish.

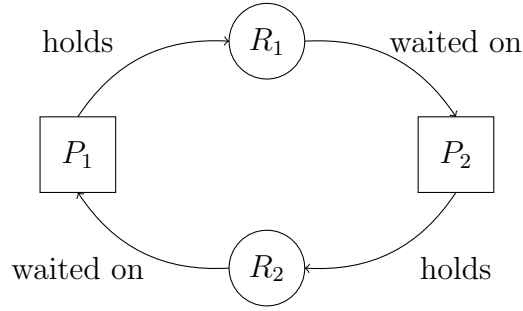


Figure 2.3: Deadlock - Resource Allocation Graph

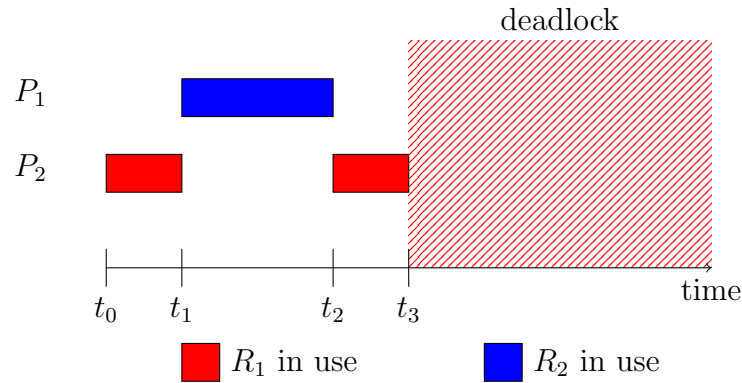


Figure 2.4: Deadlock

2.2.3 Deadlock

In the use of mutexes for resource sharing, a deadlock occurs when a task waits on a mutex to a resource held by another task, which in turn is waiting for another mutex held by the first task, as illustrated in figure 2.3.

Although Priority Inheritance Protocol is well-suited to handle the problem of priority inversion, it does not help to prevent deadlocks, as illustrated in figure 2.4. At t_0 , P_2 starts executing and acquires the mutex for R_1 . At t_1 , P_1 becomes ready and starts executing immediately, as it has higher priority than P_2 . P_1 acquires the mutex for R_2 at the start of its execution. At t_2 , P_1 tries to take the mutex for R_1 but is blocked by P_2 , which holds that mutex. With P_1 blocked, P_2 then resumes execution and at t_3 tries to take the mutex for R_2 , which is currently held by P_1 . As P_1 and P_2 are now waiting for each other, no progress can be made and a deadlock occurs.

2.2.4 Priority Ceiling Protocol

Priority Ceiling Protocol (PCP) extends Priority Inheritance Protocol by adding more restrictions on mutex acquisition. Under this scheme, each mutex is assigned a priority ceiling which is the same priority as the highest priority task that will use that mutex. A task is allowed to acquire a mutex when its priority is higher than all priority ceilings of mutexes currently locked by tasks other than that task. When a low priority task blocks higher priority tasks from acquiring a mutex, the priority of the low priority task is raised to the highest priority among the higher priority tasks, (Lui S.; Rajkumar, R.; Lehoczky, J.P. , 1990).

Chapter 3

Overview of Micrium μ C/OS-III

3.1 Task Model

Tasks in μ C/OS-III are implemented as normal C functions with their own accompanying Task Control Blocks and stacks. However, unlike normal C functions, tasks are not allowed to return (Labrosse, J. J. , 2010, p. 83).

Task Control Block (TCB) is a C **struct** which holds necessary task-related information on which the whole of μ C/OS-III depends in order to function properly. The information contained in a TCB includes a pointer to the top of stack, a pointer to the C function underlying this task, the current state of this task, the priority of this task and many more.

There are two type of tasks, namely run-to-completion and infinite loop. Run-to-completion tasks must call OSTaskDel() at the end of the function (Labrosse, J. J. , 2010, p. 84).

3.2 Scheduling Algorithm

μ C/OS-III has a priority-based, preemptive scheduler. (Labrosse, J. J. , 2010, p. 141).

Priority-based

Each task are assigned a static priority. The kernel schedule them based on their priorities.

Preemptive

Higher priority tasks can preempt lower priority tasks, which means that during execution of a low priority task, if a high priority task is ready, the low priority task may be suspended so as to give CPU time to the high priority task.

3.3 Resource Sharing

Chapter 4

Modifications

4.1 Earliest Deadline First Scheduling

Earliest Deadline First Scheduling (EDF) is a dynamic scheduling algorithm.

4.1.1 Notes

EDF does not care about the relative deadline of the task but cares about its absolute deadline and its TCB.

The task spawner which spawns jobs for recurrent tasks cares about their relative deadlines and periodicities.

Because of task periodicities, the task spawner must allocate memory dynamically (for TCBs, stacks) to create job instances for tasks.

The OSRdyHeap is used to keep track of ready tasks with deadlines. The heap is a min heap w.r.t absolute deadlines.

The SpawnerHeap is used to keep track of which task to spawn next; this heap is also a min heap w.r.t. absolute spawn time.

⇒ Need for coarser time tick management: running spawner on every tick is expensive → cannot make guarantees about timeliness of task spawning.

Dynamic memory allocation problem:

Let the user allocate a block of memory for TCBs (this leads to problems with how tasks can communicate)

4.2 Priority Ceiling Protocol

The two basic operations on a mutex are (1) waiting on it and (2) releasing it. To wait on a mutex, `OSMutexPend()` must be called. To release a mutex, `OSMutexPost()` must be called.

4.2.1 OSMutexPend()

The function starts out with checking if the use of the mutex is legal, by checking if the current task's ceiling not higher than that of the mutex ceiling.

```
if ( OSCompareCeilings(OSTCBCurPtr->Prio ,
                        OSTCBCurPtr->RelDeadline ,
                        p_mutex->PrioCeiling ,
                        p_mutex->RelDeadlineCeiling) >
    0) {
    *p_err = OS_ERR_PEND_ABOVE_MUTEX_CEILING;
    return;
}
```

Next, the system ceiling for the current task and mutex is calculated. This calculation is carried out by going through all active mutexes which are not owned by the current task, and update the ceiling appropriately.

```
p_mutex          = OSMutexHeadPtr;
prio_ceiling      = OS_CFG_PRIO_MAX;
rel_deadline_ceiling = DEF_INT_64U_MAX_VAL;
while (p_mutex != (OS_MUTEX *)0) {
    if (p_mutex != p_mutex &&
        p_mutex->OwnerNestingCtr != 0 &&
        p_mutex->OwnerTCBPtr != OSTCBCurPtr) {
        if (OSCompareCeilings(prio_ceiling ,
                                rel_deadline_ceiling ,
                                p_mutex->PrioCeiling ,
```

```

                                p_mutex->
                                RelDeadlineCeiling) <
                                0) {
        prio_ceiling = p_mutex->PrioCeiling;
        rel_deadline_ceiling = p_mutex->
                                RelDeadlineCeiling;
    }
}
p_mutex_next = p_mutex_next->NextPtr;
}

```

If the current task meets the requirements, access to mutex is granted to it. The task's priority is immediately raised to the ceiling of the mutex.

```

if (p_mutex->OwnerNestingCtr == 0 &&
      OSCompareCeilings(prio_ceiling ,
                        rel_deadline_ceiling ,
                        p_mutex->PrioCeiling ,
                        p_mutex->RelDeadlineCeiling) <
                        0) {
    p_mutex->OwnerTCBPtr      = OSTCBCurPtr;
    p_mutex->OwnerOriginalPrio = OSTCBCurPtr->Prio;
    OS_RdyListRemove(OSTCBCurPtr);
    OSTCBCurPtr->Prio        = p_mutex->PrioCeiling
        ;
    OSPrioCur                = OSTCBCurPtr->Prio;
    p_mutex->OwnerNestingCtr  = (OS_NESTING_CTR) 1;
    if (p_ts != (CPU_TS *)0) {
        *p_ts                = p_mutex->TS;
    }
    OS_PrioInsert(OSTCBCurPtr->Prio);
    OS_RdyListInsert(OSTCBCurPtr);
}

```

```

        CPU_CRITICAL_EXIT();

        *p_err = OS_ERR_NONE;

        return;
    }

```

If the current task is already the owner of the mutex, increase nexting level.

```

    if (OSTCBCurPtr == p_mutex->OwnerTCBPtr) {
        p_mutex->OwnerNestingCtr++;
        if (p_ts != (CPU_TS *)0) {
            *p_ts = p_mutex->TS;
        }
        CPU_CRITICAL_EXIT();
        *p_err = OS_ERR_MUTEX_OWNER;
        return;
    }

```

```

    if ((opt & OS_OPT_PEND_NON_BLOCKING) != (OS_OPT)0) {
        /* Caller wants to block if not available?
           */
        CPU_CRITICAL_EXIT();
        *p_err = OS_ERR_PEND_WOULD_BLOCK;
        /* No
           */
        return;
    } else {
        if (OSSchedLockNestingCtr > (OS_NESTING_CTR)0) {
            /* Can't pend when the scheduler is locked
               */
            CPU_CRITICAL_EXIT();
            *p_err = OS_ERR_SCHED_LOCKED;
            return;
        }
    }

```

```

    }
}

OS_CRITICAL_ENTER_CPU_CRITICAL_EXIT();

/* Lock the scheduler/re-enable
interrupts */
/* This code belongs to the basic Priority Inheritance
Protocol. As we are implementing Immediate
Priority Ceiling Protocol,
* it should not be relevant any more.
*/
// p_tcb = p_mutex->OwnerTCBPtr;
/* Point to the TCB of the
Mutex owner */
//

/* See if mutex owner has a lower priority than current
*/
// if (p_tcb != (OS_TCB*)0 && p_tcb->Prio > OSTCBCurPtr
->Prio) {
//     switch (p_tcb->TaskState) {
//         case OS_TASK_STATE_RDY:
//             OS_RdyListRemove(p_tcb);
//             /* Remove from ready list at current
priority */
//             p_tcb->Prio = OSTCBCurPtr->Prio;
//             /* Raise owner's priority
*/
//             OS_PrioInsert(p_tcb->Prio);
//             OS_RdyListInsert(p_tcb);
//             /* Insert in ready list at new

```

```

        priority                                     */
//
//          break;
//
//          case OS_TASK_STATE_DLY:
//          case OS_TASK_STATE_DLY_SUSPENDED:
//          case OS_TASK_STATE_SUSPENDED:
//          p_tcb->Prio = OSTCBCurPtr->Prio;
//          /* Only need to raise the owner's priority
//          */
//          break;
//
//          case OS_TASK_STATE_PEND:
//
//              /* Change the position of the
//              task in the wait list          */
//          case OS_TASK_STATE_PEND_TIMEOUT:
//          case OS_TASK_STATE_PEND_SUSPENDED:
//          case OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED:
//          OS_PendListChangePrio(p_tcb ,
//
//              OSTCBCurPtr->Prio
//          );
//          break;
//
//          default:
//          OS_CRITICAL_EXIT();
//          *p_err = OS_ERR_STATE_INVALID;
//          return;
//      }
//  }

OS_Pend(&pend_data ,

                                           /* Block task

```

```

    pending on Mutex                                     */
    (OS_PEND_OBJ *) ((void *)p_mutex),
    OS_TASK_PEND_ON_MUTEX,
    timeout);

OS_CRITICAL_EXIT_NO_SCHED();

OSSched();

                                                                    /*
    Find the next highest priority task ready to run
    */

CPU_CRITICAL_ENTER();
switch (OSTCBCurPtr->PendStatus) {
    case OS_STATUS_PEND_OK:
                                                                    /* We got the mutex
                                                                    */

        if (p_ts != (CPU_TS *)0) {
            *p_ts = OSTCBCurPtr->TS;
        }
        *p_err = OS_ERR_NONE;
        break;

    case OS_STATUS_PEND_ABORT:
                                                                    /* Indicate that we
                                                                    */
        aborted
        if (p_ts != (CPU_TS *)0) {
            *p_ts = OSTCBCurPtr->TS;
        }
        *p_err = OS_ERR_PEND_ABORT;
        break;

```

```

case OS_STATUS_PEND_TIMEOUT:

                                /* Indicate that we didn
                                't get mutex within timeout */
                                */
    if (p_ts != (CPU_TS *)0) {
        *p_ts = (CPU_TS )0;
    }
    *p_err = OS_ERR_TIMEOUT;
    break;

case OS_STATUS_PEND_DEL:

                                /* Indicate that
                                object pended on has been deleted */
                                */
    if (p_ts != (CPU_TS *)0) {
        *p_ts = OSTCBCurPtr->TS;
    }
    *p_err = OS_ERR_OBJ_DEL;
    break;

default:
    *p_err = OS_ERR_STATUS_INVALID;
    break;
}
CPU_CRITICAL_EXIT();

```

4.2.2 OSMutexPost()

Bibliography

- Witchel, E. (2009) *CS372 Operating Systems*. Retrieved from <http://www.cs.utexas.edu/users/witchel/372/lectures/01.OSHistory.pdf> on 2014/03/10.
- Labrosse, J. J. (2010) *μ C/OS-III: The Real-Time Kernel*. Retrieved from <http://micrium.com/books/ucosiii/ti-lm3s9b92/>.
- Liu, C. L. and Layland, J. W. (1973) *Scheduling Algorithm for multiprogramming in a Hard-Real-Time Environment*. Journal of ACM, 1973, vol 20, no 1, pp. 46-61.
- Lehoczky, J.; Sha, L. and Ding, Y. (1989) *The rate monotonic scheduling algorithm: exact characterization and average case behavior*. IEEE Real-Time Systems Symposium, pp. 166-171.
- Dijkstra, E. W. (1965) *Solution of a Problem in Concurrent Programming Control*. Communications of the ACM, Volume 8, Issue 9, p. 569.
- Reeves, G. E. (1997) *What really happened on Mars?*. http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html, accessed on 2014/03/17.
- Lui S.; Rajkumar, R.; Lehoczky, J.P. (1990) *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*. IEEE Transactions on Computer, Volume 39, Issue 9, pp. 1175-1185.