Implementing priority inheritance semaphore on uC/OS real-time kernel

Jae-Ho Lee, Heung-Nam Kim

Internet Appliance Technology Department, ETRI, Korea
{bigleap,hnkim}@etri.re.kr

Abstract

In a preemptive priority based RTOS, priority inversion problem is among the major sources of deadline violations. Priority inheritance protocol is one of the approaches to reduce priority inversion. Unfortunately, RTOS like uC/OS can’t support priority inheritance protocol since it does not allow kernel to have multiple tasks at the same priority. Although it has different ways to avoid priority inversion such as priority ceiling protocol, developers still have some difficulties in programming real-time applications with it. In this paper, we redesign the uC/OS kernel to provide the ability to support round robin scheduling and implement priority inheritance semaphore on the modified kernel. As result, we port new kernel with priority inheritance semaphore to evaluation board, and evaluate the execution time of each of the kernel service as well as verify the operations of our implementation.

1. Introduction

In a priority based RTOS, ideally a high priority task must always be able to preempt a low priority task. But there are some resources that must be shared among tasks in real-time applications. In this situation, priority inversion problem occurs when a low priority task owns a common resource needed by a high priority task. The low priority task delays execution of the high priority task until it releases the resource. At this time, the condition is aggravated if the low priority task is preempted by intermediate priority tasks regardless of sharing the resources, more delaying access to the resource by the high priority task. A well-known solution is priority inheritance mutual exclusion semaphore (called mutex), which has ownership and priority.

uC/OS also provides mutex to reduce priority inversion problem. uC/OS kernel written by Jean Labrosse is highly portable, scalable, real-time, multitasking kernel[1][2]. In many fields, various products that are designed around uC/OS have already proved the stability of kernel[9]. But its mutex is implemented by making use of priority protection protocol that must reserve the ceiling priority[3]. This makes application programs complex and requires code analysis to assign the priority ceiling of each of the common resources.

In this paper, we introduce the concept of priority inversion, discuss how to avoid this problem in uC/OS, and draw some shortcomings from current method called priority protection. In order to get over the defects, we add several kernel structures, redesign scheduler to support round robin (RR) and implement priority inheritance mutex. We incorporate our implementation to uC/OS and port it to evaluation board including 16-bit processor. Finally, we measure the execution times of kernel services and verify the operations of mutex.

2. Backgrounds

2.1. Priority inversion

Figure 1 illustrates a priority inversion scenario with a shared resource and three tasks. In this scenario, Task(H) is preempted by Task(L) at point (6). Because Task(L) owns the resource, Task(H) has to wait until Task(L) releases shared resource. Moreover, the situation between (8) and (10) is getting worse when intermediate priority tasks like Task(M) preempt Task(L). The more longer the execution time of Task(M), the more further delayed the execution of Task(H). There are two approaches such as priority inheritance and priority protection(or priority ceiling) to avoid unbounded priority inversion. Although there are some commercial kernels that provide priority inheritance, uC/OS only support priority protection protocol since kernel does not support RR scheduling which is based on priority inheritance protocol.

2.2. Priority protection protocol in uC/OS

In uC/OS, each task is assigned a unique priority level between 0 and 63. Any tasks are not allowed to assign
same priority. This means that uC/OS does not support RR scheduling. Figure 2 presents the original kernel structure. All Task Control Blocks (TCBs) are placed in OSTCBPrioTbl[] and they are linked in a singly linked list. When a task is created, it must be assigned a unique TCB.

When a task is created, it must be assigned a unique TCB. 

**Figure 2. Task management in uC/OS**

For the above structure, uC/OS only provides priority protection protocol in place of priority inheritance protocol. The basic idea is that a priority above the high priority task must be reserved by the resource to allow a low priority task to be temporarily raised in priority. Each resource has a priority ceiling and the priority level is within the valid range of priorities. When a task owns a shared resource, it temporarily receives the mutex priority ceiling, if the ceiling is higher than its own priority. It recovers its original priority when it releases the mutex. The priority ceiling should have the value of the highest priority of all tasks that may lock the mutex.

But this protocol has some drawbacks. It is bothersome to assign the unique priority ceiling for each resource. The programmer is responsible for assigning the priority as well as need to analyze the total program statically. Unless the assigned priority is appropriate, priority inversions or even deadlocks may occur. Moreover, the protocol would be inefficient in average performance because of unnecessary context switches.

**2.4. Priority inheritance protocol**

Priority inheritance is no additional effort to assign the priority ceiling. In addition, many extra context switches are avoided since medium priority tasks are not preempted unnecessarily. This leads to more excellent average performance than that of priority protection protocol. Figure 3 shows how the priority inheritance reduces priority inversion time. Task(H) tries to access the shared resource that is locked by Task(L) at point (6), but it fails. Before Task(H) is preempted by Task(L), kernel raises the priority of Task(L) to the same level as Task(H) and switches back to Task(L) at point (6). Task(L) is done with the resource without any interruption, it releases the resource at point (8). Although Task(M) may occur any time between (6) and (8), Task(L) is not preempted because of the priority inheritance. After all the kernel returns the priority of Task(L) to its original value and gives the resource to Task(H) at point (8).

We obviously see that the priority inversion time is considerably reduced by priority inheritance protocol. In order to support this mechanism, in the first place, kernel must have the ability called RR scheduling.

**3. Implementation**

**3.1. Kernel modification for RR scheduling**

In order to support RR scheduling, we add several data structures to existing kernel and modify kernel code. Figure 4 presents the modified kernel structure for supporting RR scheduling. Data structure, **os_task_info**, maintains the lists of multiple tasks (or TCBs) at the same priority in OSTCBPrioTbl[]. **OSTCBHeader** points to the first TCB that is in each OSTCBPrioTbl[]. **OSTaskNum** is used to hold the count how many tasks are in each OSTCBPrioTbl[].

**Figure 3. Priority inheritance protocol**

**Figure 4. Kernel structure that supports RR scheduling**

New kernel saves a lot of memory since it initially reserves **OS_TCB_INFO** instead of **OS_TCB**. When existing kernel is initialized, it compulsory reserves 64
TCBs, but the implemented kernel allocates an OS_TCB only when a task is actually created.

```c
OSScheduler()
{
    SavedQuantum = CurrentTask's TimeQuantum;
    OSTCBHighRdy = "Get a new high priority task";
    if (SavedQuantum == 0)
        CurrentTask's TimeQuantum = "Default Value";
    if (OSTCBHighRdy == CurrentTask )
    {
        if (SavedQuantum == 0)
            if (OSTCBPrioTbl[].OSTaskNum >1)
                CPU context switch to CurrentTask's Neighbor;
        else  {
            if (SaveQuantum == 0) {
                CurrentTask's Quantum = 0;
                Put CurrentTask into Ready List;
                CPU context switch to OSTCBHighRdy;
            }
        }
    }
}
```

Figure 5. RR scheduler algorithm

In RR scheduler, each task is assigned a constant time called quantum which it is allowed to run. If the task is still running at the end of the quantum, the CPU is preempted by another tasks. If the task has blocked or finished before its quantum has elapsed, the context switch is done. As shown in Figure 6, the algorithm using our revised data structure supports RR scheduling.

### 3.2. Implementing priority inheritance mutex

We add several fields to existing Event Control Block(ECB) for implementing priority inheritance mutex. Figure 6 includes additional fields as the following.

```c
typedef struct os_event {
    int        OSEventCnt;
    void        *OSEventPtr;
    struct os_tcb  *OSSuspendlist;
    /* for Mutex Semaphore */
    struct os_tcb  *MutexOwner;
    uint        CurrentUse;
    uint               OriginPrio;
} OS_EVENT;
```

OSEventCnt contains the current value of the semaphore which indicates how many tasks can access the resource at one time or how many times an event has occurred.

OSEventPtr is a pointer to the messages mailbox or queue structures.

OSSuspendlist is used to singly link TCBs suspending on some events.

MutexOwner is pointer to a TCB which owns mutex.

CurrentUse is used to indicate if mutex is available

OriginPrio is used to hold the original priority of the task before the priority is changed by priority inheritance. When the task releases the mutex, this field is used to restore the original priority of the task.

We implement the following system calls for handling mutexes based on the above structure.

**OSMutexMake(OS_EVENT *event )**
This function creates and initializes a mutex, which is used to get exclusive access to a resource.

**OSMutexGet(OS_EVENT *event, uint timeout, uint *err)***
When a task requests mutex, if mutex is available, the task gets the mutex at once and kernel updates OriginPrio and MutexOwner in ECB. If mutex is not available and the task has a priority above the mutex owner, kernel raises the priority of a mutex owner to the priority of the requesting task. Optional field, timeout, prevents the task from waiting indefinitely for mutex to be gained.

**OSMutexRelease(OS_EVENT *event)**
A mutex is released only by task acquired the mutex by first calling OSMutexGet. If the priority of mutex owner has been raised when priority inversion occurs, the original priority of the task will be restored. At this time if one or more tasks are waiting for the mutex, the mutex is given to the highest priority task waiting on the mutex. If no task is waiting on the mutex, CurrentUse in ECB is set to 0 for indicating that mutex is free.

### 4. Experiments

#### 4.1. Porting kernel to CalmRISC16

We port the modified kernel to a CalmRISC16 evaluation board, including a 16-bit microprocessor made in Samsung Electronics. We use CalmShine16 as a software development tool, which includes C-Compiler, Assembler, Linker, and also debugging tool.

When developers write the application code to meet the deadline, they should refer to the execution time of each of the kernel services RTOS designers provide. So we measure and provide this information, as shown Table 1.

<table>
<thead>
<tr>
<th>Basic Services</th>
<th>CalmRISC16 (10Mhz)</th>
<th>Communication &amp; Coordination Services</th>
<th>Time (uSec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSCtxSw()</td>
<td>8-10u</td>
<td>OSSemGet()</td>
<td>30.5u</td>
</tr>
<tr>
<td>OSCreateTask()</td>
<td>224-248u</td>
<td>OSSemRelease()</td>
<td>30.5u</td>
</tr>
<tr>
<td>OSDelTask()</td>
<td>154u</td>
<td>OSMboxSend()</td>
<td>34.5u</td>
</tr>
<tr>
<td>OSTaskChangePrio()</td>
<td>(+) 301.4u</td>
<td>OSMboxReceive()</td>
<td>34.3u</td>
</tr>
<tr>
<td>OSTimeDelay()</td>
<td>(+) 11.3u</td>
<td>OSMutexGet()</td>
<td>31.9u</td>
</tr>
<tr>
<td>OSSchedule()</td>
<td>72-112u</td>
<td>OSMutexRelease()</td>
<td>40.5u</td>
</tr>
<tr>
<td>OSWaitEvent()</td>
<td>64-72u</td>
<td>OSMboxWait()</td>
<td>36.5u</td>
</tr>
<tr>
<td>ENTER/EXIT_CRITICAL()</td>
<td>5.6u</td>
<td>OSMboxWait()</td>
<td>38.5u</td>
</tr>
</tbody>
</table>

We evaluate the modified kernel using a 16-bit microprocessor made in Samsung Electronics. We use CalmShine16 as a software development tool, which includes C-Compiler, Assembler, Linker, and also debugging tool.

When developers write the application code to meet the deadline, they should refer to the execution time of each of the kernel services RTOS designers provide. So we measure and provide this information, as shown Table 1.
4.2. Testing experimental mutex

We consider only three tasks to simulate Figure 1 or Figure 3 in previous sections. We create three tasks(Task[H/M/L]) and a shared resource in main function. TaskH and TaskL share the resource but TaskM is irrelevant to the resource. Three tasks print various brackets such as (, ), [, ] respectively to indicate the state of processing their job while they are executing. Figure 7 and Figure 8 show the results for both the existing semaphore and the new implemented mutex.

When using a semaphore in application, there is long priority inversion time, as shown in Figure 7(L: Low priority task, M: Intermediate priority task, H: High priority task).

![Figure 7. Execution order for code using a semaphore](image)

Otherwise, when using a mutex, the priority inversion time is remarkably reduced by priority inheritance mutex, as shown in Figure 8. TaskL is never preempted by TaskM while it is holding mutex.

![Figure 8. Execution order for code using a mutex](image)

5. Related Work

Although priority inheritance promises a solution to priority inversion without code analysis, the lately paper written by Victor Yodaiken says that it is neither efficient nor reliable. According to his paper, the followings are incompatible with reliable real-time systems using priority inheritance.

- Nested critical region
- Mixed inheriting and non-inheriting operations
- Worst case performance & OS performance

In this paper, we just handle a simple situation like section 4. So we will more study priority inversion examples that may appear in above situations and improve our implementation to work correctly and effectively.

6. Conclusion

In RTOS, kernel should always execute the task with the highest priority. It is impossible for RTOS to achieve this characteristic when priority inversion occurs. This breaks the determinism of RTOS and makes it difficult to be met the deadline in real time software. In case of the uC/OS kernel, priority inversion is reduced by priority protection protocol. But it has several weaknesses. First, it requires static analysis of the system to find the priority ceiling of each shared resource. Second, it will pay the extra cost of changing a task’s priority twice regardless of whether there is contention for the lock or not, resulting in higher overhead and many unnecessary context switches. While there is another mechanism called priority inheritance protocol, uC/OS kernel does not support that because of the constraints of kernel structure.

In this paper, we redesign and modify the kernel structure to implement priority inheritance mutex on uC/OS. New kernel with priority inheritance mutex is ported to CalmRISC16 evaluation board, and the execution time of each of the kernel services is measured by logic analyzer. Priority inheritance mutex we concentrate on is verified through a simple example program. It makes developers easy to write real-time application codes since it does not require additional efforts such as code analysis or priority ceiling assignment in priority ceiling protocol. In addition it leads to excellent average performance since it does not require unnecessary context switching.

Priority inheritance protocol is simple and powerful. It, however, is still dangerous in some situation specified in academic papers. For future work, we plan to improve our implementation to work reliably in that situation.

References