Time-Critical Computing Project Phase 2 Report

Raphael BOUILHOL
Nanyang Technological University
Polytech Sorbonne
rbouilhol@hotmail.fr

I. Introduction

In µOS-III, Semaphores are the kernel objects implemented to manage resource sharing, which is needed to prevent a task being preempted when it is accessing critical resources that the other task also need. Two types exist, Binary and Counting. When a task ”Pend” a Semaphore and access the resource, a +1 is added to the Semaphore counter and a -1 is added when the Semaphore is ”Posted”.

In this report, we are going to talk about a special type of Semaphore, which is ”Mutex”, Mutual Exclusion Semaphores. When a task Pend a Mutex, any higher priority task that want to Pend the Mutex will be Blocked and will have to wait for the Task owning the Mutex to Post it, freeing the Mutex.

With this being the only resource sharing protocol, we can easily find into a priority inversion problem, when a higher priority task is blocking another task that is holding a Mutex without using the resource protected by the Mutex. To solve this issue, µOS-III implemented the Priority Inversion Protocol (PIP).
In this Protocol, whenever a task blocks on a shared resource, the priority of the task holding the Mutex is raised to that of the blocked job. For this sole purpose, this protocol does the job, but it still allows the system to fall into a Deadlock situation, in which 2 tasks are waiting for the other to free a resource that the other needs.

For this reason, we chose to implement the Priority Ceiling Protocol (PCP) for resource sharing.
This protocol expand on PIP by adding a ceiling system. Each Mutex has a ”Resource Ceiling” which correspond the highest priority among all tasks that can use the shared resource (attributed statically), and the System as a whole has a ”System Ceiling”, which is the highest resource ceiling among all currently locked resources (defined dynamically).
When a task requests a shared resource, it is granted if the priority of the task is strictly higher than the system ceiling or if the task is already holding another resource.

Along with a more efficient resource ceiling protocol, we will also proceed to replace the data structure ”Pend List” which is implemented with a simple Linked List owned by every Mutexes, by a Red Black Tree.

II. Implementation

Firstly, we are going to describe how PCP is implemented.

We start with creating the ceiling value. For the ressource ceiling, we simply add a ”Ceiling” parameter in the *OS-MUTEX* data structure, that we are going to fill at the creation of the Mutex by passing the TCB of the task with the highest priority using the Mutex.
For the system ceiling, we chose to implement a Stack. When a task takes possession of a Mutex, we look at if it has a higher priority than the previous system ceiling and if it is the case, we push the value on the stack. The same of for ”Poping” the stack, when we free a mutex, we look at if the

priority of the task releasing the mutex is equal to that of the system ceiling, and we pop if it is the case.
With this implemented, all we have to do to grant a mutex is to see if the task has a higher priority than the system ceiling value, contained in the structure of the Mutex pointed to by the stack head, and if it is not the case we look at if the task is already a mutex. For this purpose, we also added a parameter called "*hasMutex*" to the structure of the TCB, that is going to increase/decrease when the task acquires/releases a Mutex.
And for the priority adjustment part, we simply keep what was already implemented in the OS.

That covers the work done to Pend a Task, now for the Post. We can separate two different cases : When tasks are waiting on the Mutex to be freed, and when it is not the case.
To simplify this treatment we have added a "*TCBWaiting*" parameter to the Mutex structure, that is going to increment each time the Mutex is responsible for the blocking of a task. The mutex responsible will also be pointed by a newly created parameter in the TCB, "*WaitingForMutex*".
If no tasks are waiting, we simply reset the values related to the owner, but if some are, we have to search the Red Black Tree for the highest priority task that is blocked by the said Mutex, then surrender the Mutex to this task and add it back on the Ready List.

Secondly, for the Red Black Tree, we have chosen to do only one that will cover all the blocked task. Their TCB are the elements comprising the tree, sorted by their Priority. As evoked previously, when looking for the task with the highest priority blocked by a specific mutex, we only have to look at the task sorted by the lowest priority value to the highest, and see which is gthe first that is blocked by the mutex using the WaitingForMutex pointer.
Because the Red Black Tree is a Binary Search Tree, its search complexity is O(log n), making it better than the original Linked List data structure, that has a search complexity of O(n).

III. Benchmarking

We now want to look at the effectiveness of our implementation. To do so, we look at the overhead needed to perform a Pend and a Post function, first by the original code of the OS, then on our version. The results will be displayed in term of Clock Cycles.

Action	PIP	PCP
Pend resulting in a granted resource	150	191
Pend resulting in a Block Task	323	266
Post with no task waiting	166	205
Post to a task waiting	-	6359

We can observe that the values are similar in every situation. For a successful Pend and post, very little is performed in both cases so having low numbers in normal and for the more complex operations, the results should come a lot from the improved search complexity of the implemented data structures, which will not show when only one task is blocked as we have done to perform the measurements.

Finally, as for the time between the Tick ISR and the release of the task onto the Ready List, it has not been affected by the modifications and stays at 130 clock cycles, instead of the 190 of the original OS code.

IV. Conclusion

To conclude, we have successfully implemented a better resource sharing protocol in BST and a better data structure to store the Blocked Task in the Red Black Tree, without increasing the Overhead on a small sample.