Implementation of Task Recursion and Earliest Deadline First scheduling in Micrium-OS III

Priya Toshniwal and Abishek Sethupandi

***Abstract*— Existing Micrium /OS-III is a Real Time Operating System (RTOS) and it ha s a preemption-based scheduling kernel. Through this paper, we present an approach to change the task management structure of the OS and scheduling policy according to our requirement using splay tree and binomial heap data structures. In this report, we present the details of our implementation and the advantages/shortcomings of our implementation.**

I. INTRODUCTION

Micrium /OS-III is a scalable, ROMable, preemptive real-time kernel that manages an unlimited number of tasks. Micrium /OS-III is a third-generation kernel, offering all of the services expected from a modern real-time kernel including resource management, synchronization, inter-task communication, and more. The design process of a real-time application generally involves splitting the work to be completed into tasks, each responsible for a portion of the problem. Micrium /OS-III makes it easy for an application programmer to adopt this paradigm.

On a single CPU, only one task can execute at any given time. There are two types of tasks: run-to-completion and infinite loop. A run-to-completion task must delete itself by calling OSTaskDel. Tasks must be created in order for Micrium /OS-III to know about tasks. You create a task by simply calling OSTaskCreate. A task needs to be assigned a Task Control Block (TCB), a stack, a priority and a few other parameters which are initialized by OSTaskCreate. Next, a call is made to OSTaskCreateHook, where it is passed the pointer to the new TCB and this function allows you to extend the functionality of OSTaskCreate. The task is then placed in the ready-list and finally, if multitasking has started, Micrium /OS-III will invoke the scheduler.

With Micrium /OS-III, a low priority number indicates a high priority. In other words, a task at priority 1 is more important than a task at priority 10. A task control block (TCB) is a data structure used by kernels to maintain information about a task. Each task requires its own TCB and, for Micrium /OS-III, the user assigns the TCB in user memory space (RAM) (figure 1). Tasks are added to the ready list by a number of Micrium /OS-III services. The most obvious service is OSTaskCreate, which always creates a task in the ready-to-run state and adds the task to the ready list. OSTaskCreate calls OSRdyListInsertTail, which links the new TCB to the ready list by setting up four pointers and also incrementing the .Entriesfield of OSRdyList[prio]. The scheduler, also called the dispatcher, is a part of Micrium OS-III responsible for determining which task runs next. Micrium OS-III is a preemptive, priority-based kernel. The

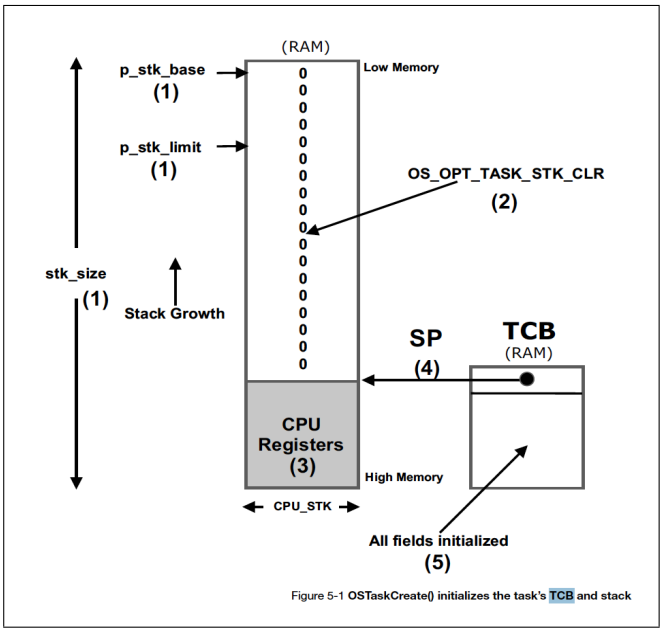

Fig. 1. TCB of every task holds all the details related to the task

word preemptive means that when an event occurs, and that event makes a more important task ready-to-run, then Micrium /OS-III will immediately give control of the CPU to that task. The scheduler looks into the bitmap OSPrioTB (figure 2) and finds out the first non-zero bit and moves to the corresponding Ready List (figure 3) to extract the tasks to be run at this priority.

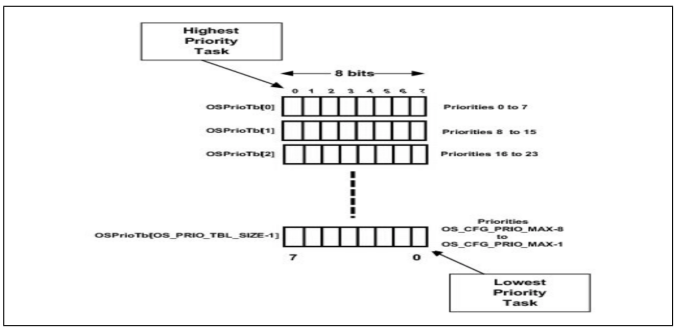

Fig. 2. Bitmap representation for insertion of the tasks according to their priorities

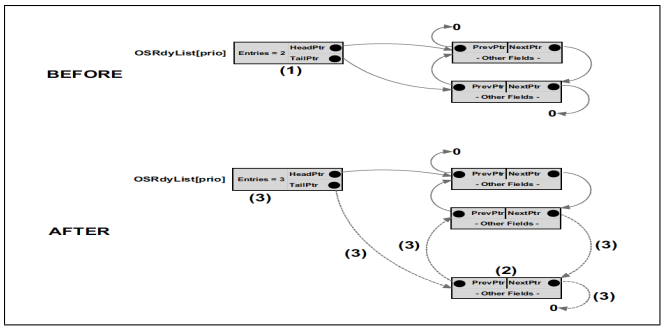

Fig. 3. Priority-based Ready List

II. IMPLEMENTATION

A. Data structure details

1) Splay Tree: The real power of the splay tree lies in its ability to hoist nodes up to the root when they are accessed, giving speedy access for nearby successive accesses. Since all operations also splay the tree on the node, the tree ends up roughly balancing itself, this results in a O(log n) amortized worst case time complexity for all operations.

In our implementation, we have every node represented by the release time value it holds. Every node contains an array that holds the TCB pointers of every task with the release time same as the value of the node. So, if there are more than one task with the same release time period, we store them together in the array in one single node. So we do not have to go and search for every node to check if their release time periods match the counter value. Additionally, if there are more than one task with the same minimum release time, they are all stored in the same node and extracted one by one from the array and put into the ready list. The task is to search the node with the minimum release time value, splay it to the root and extract the values of all the TCBs stored in this node. But for task recurssion and scheduling together, after we extract the minimum node from the splay tree, We insert these TCBs into the binomial heap and not the ready list directly, based on their deadline while we also update their values to new release time values and insert into the splay tree again and then delete the previous node.

Every time we perform the operation to extract the minimum node, time complexity is O(log n), where n is the depth of the node from the root. However, since we can store more than one task with the same release time in the same node itself, we do not have to traverse the tree again and again to find out the other tasks with the same release time. Hence, we save some traversing and searching operation time here.

However, scheduling overhead in the current implementation is that the tasks first need to be inserted into splay tree, then extracted from the splay tree and inserted back into the binomial heap and then again extracted back from the heap and inserted into the ready list and then the scheduling starts. All of these operations require sometime and may hamper the performance of the entire operation if the number of tasks to be scheduled are large. A minimum heap will perform better instead of the splay tree in our implementation of recursion.

2) Binomial Heap: In our implementation, we are using a binomial min heap for storing the TCBs based on their deadline value. All the operations of a binomial heap can be performed in O(log n) time - insertion, deletion, extract min, find min and merging. We also maintain a pointer to the min node in the tree. This makes finding the min value in the tree of O(1) complexity.

We insert the extracted TCBs from the splay tree into the heap and we find the min task(task with the earliest deadline) and we assign it a user-defined priority and insert it into the ready list for scheduling. If the four tasks with priorities of 0,1,2,3 are not available in the ready list, the scheduler executes the task we put into the ready list. Binomial heap works efficiently for EDF scheduling.

B. Algorithm for the approach

- Start with Main()
- Create a task called AppTaskStart using OSTaskCreate() and call OSStart()
- Inside AppTaskStart(), we create our five tasks using OSRecTaskCreate().
  - OSRecTaskCreate() is a replica of OSTaskCreate() with some modifications and more number of arguments.
  - Along with the 13 arguments of OSTaskCreate(), we also pass the release time of the task as well as its deadline.
- Every time we create a task inside AppTaskStart using OSRecTaskCreate(), it goes through the following steps:
  - Take the values passed as arguments to this function call and initialize the Task Control Block (TCB) with these values.
  - Compare the release time of the task and insert it into the splay tree using the insert() API call.
  - Check the current value of the counter and extract the the tasks with the minimum release time.
    - Currently, the frequency of the clock tick is 1000Hz. This implies that we get 1 tick interrupt after every 1ms.
    - After every 1000 ticks, interrupt occurs and the ISR (Timer0A) gets executed and the counter value gets increased by 1.
  - Extract the TCB of the task inside this node of the splay tree using GetMinRecTree() API function call and insert the TCB into the binomial heap based on its deadline time value.
  - Alongside this operation, update the TCB with the next release time value and insert it again back to the splay tree based on its next release time value (Notice here that the TCBs are preserved in our implementation and not deleted after the task is finished running).
- Delete OSRecTaskCreate() by calling OSTaskDel() function.

C. Splay tree API deatils

- DelRecTree() : Delete the required node from RecursionTree
- InsertRecTree(): Insert the required node to RecursionTree with updated TCB and release time elements
- GetMinRecTree(): Get the minimum release time key node from RecursionTree

D. Creating Hardware Timer

TIMER0A is configured as 32-bit periodic timer and its load value is configured with the value equal to system clock value in Hertz. The TIMER0A ISR is registered in appvect.c file as AppTaskLoader.c The ISR is called back for every 1 second, where we clear the interrupt flags. Steps to add a hardware time - In BSP.c

- Enable the peripheral for TIMER0A
- Initialize BSP Interrupt
- Disable the Timer0A
- Configuring and Loading value to TIMER0A
- Enable Timer and Master Interrupts
- Enable Timer

E. Counter overflow management

Counter is a static global variable which is used to manage the Time tick value from the Hardware timer (TIMER0A). Overflow of CPUINT32U variable is managed using a function CounterOverflow(). The function checks the window range for counter value, and if the counter's MSB bit is detected as 1, it goes into the window 1 category and we reset the counter to window 0 with the correct value (depending on the value of the counter when an overflow is detected, the reset can be to value 0 or greater than 0).

- Window 0 implies MSB = 0
- Window 1 implies MSB = 1

Counter Overflow is also taken care of for the release time and absolute deadline variables used in the tree.

III. Future work and improvements

Currently, we are figuring out some issues with scheduling. We are using binomial heap for scheduling because we want to extract the minimum deadline task to be put into the ready list, along with the 4 OS tasks. Also, in task recursion, we will keep a pointer to the min node so that the search operation time complexity becomes O(1).