Implementation of Task Recursion and Stack Resource Policy with 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. We also present the implementation of Stack Resource Policy protocol for resource sharing and synchronization. This report contains 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. 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.

A. Creation of Tasks

There are two types of tasks: run-to-completion and infinite loop.Tasks must be created in order for Micrium /OS-III to know about tasks. You create a task by simply calling OSTaskCreate. A task control block (figure 2) is a data structure used by kernels to maintain information about a task 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.

B. Existing Timer details

OSTmrTask is a task created by C/OS-III and its priority is configurable by the user. OSTmrTask is typically set to a medium priority. OSTmrTaskis a periodic task and uses the same interrupt source used to generate clock ticks. However, timers are generally updated at a slower rate and thus, the timer tick rate is divided down in software. If the tick rate is 1000 Hz and the desired timer rate is 10 Hz then the timer task will be signaled every 100th tick interrupt as shown in the figure 1.

C. Scheduling of tasks

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. 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

Fig. 1. Tick ISR and Timer Task relationship

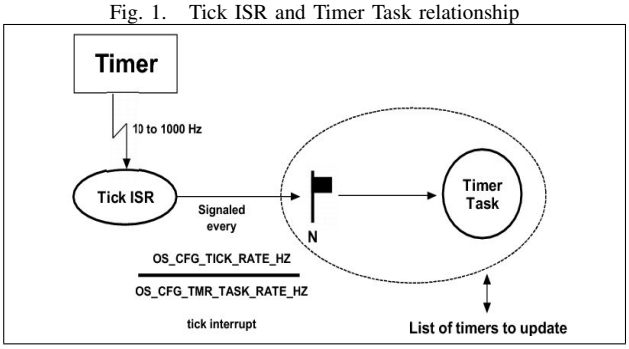
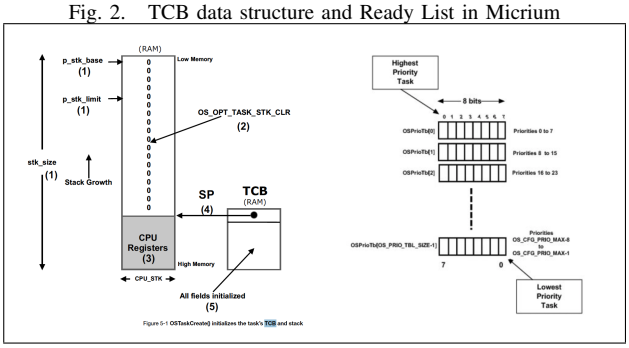


Fig. 2. TCB data structure and Ready List in Micrium



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 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.

D. Kernel objects for resource sharing

A shared resource is typically a variable (static or global), a data structure, table (in RAM), or registers in an I/O device. Although sharing data simplifies the exchange of information between tasks, it is important to ensure that each task has exclusive access to the data to avoid contention and

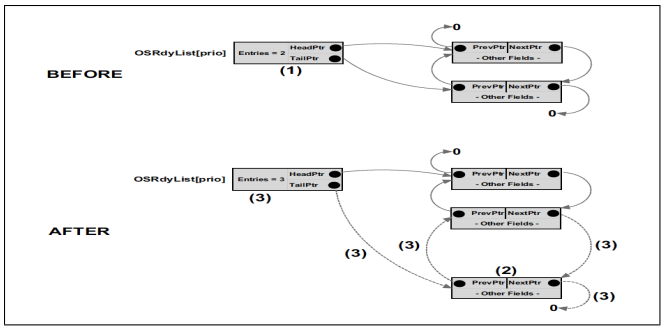


Fig. 3. Priority-based Ready List

data corruption. The most common methods of obtaining exclusive access to shared resources and to create critical sections are:disabling interrupts, disabling the scheduler, using semaphores or mutual exclusion semaphores (mutex).

1) Semaphores:

- A semaphore is a kernel object created for locking mechanism and code acquired that acquired key to this lock continues execution. Acquiring the key means that the executing task has permission to enter the section of otherwise locked code. Entering a section of locked code causes the task to wait until the key becomes available.

2) Mutex and Priority Inheritance Protocol:

- Micrium C/OS-III supports a special type of binary semaphore called a mutual exclusion semaphore (also known as a mutex) that eliminates unbounded priority inversions. An application may have an unlimited number of mutexes. Only tasks are allowed to use mutual exclusion semaphores (ISRs are not allowed). C/OS-III enables the user to nest ownership of mutexes. If a task owns a mutex, it can own the same mutex up to 250 times. The owner must release the mutex an equivalent number of times.
- OSMutexPend acquire a mutual exclusion semaphore. If a task calls this and the mutex is available, OSMutexPend gives the mutex to the caller and returns to its caller. However, if the mutex is already owned by another task, OSMutexPend places the calling task in the wait list for the mutex(2-3-4 TREE in our implementation).
- OSMutexPost is used to release the mutex. If the priority of the task that owns the mutex has been raised when a higher priority task attempted to acquire the mutex, the priority of the owning task will be set to the highest priority in the owning task's mutex group or its base priority, whichever is higher.
- If one or more tasks are waiting for the mutex, the mutex is given to the highest-priority task waiting on the mutex. The scheduler is then called to determine if the awakened task is now the highest-priority task ready-to-run, and if so, a context switch is performed to run the readied task. If no task is waiting for the mutex, the mutex value is simply set to available.
- This is the current implementation of Priority Inheritance Protocol in Micrium.

II. IMPLEMENTATION

A. Phase 1 Details

1) 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

All of our data structures maintain a pointer to the minimum node. This makes the time complexity of extracting minimum O(1).

2) 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, 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. This reduces searching time in cases of synchronous release times. 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.

3) 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. Phase 2 Details

C. AVL Tree

Most of the BST operations (search, max, min, insert, delete, etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Log (n)) after every insertion and deletion, then we can guarantee an upper bound of O(Log (n)) for all these operations. The height of an AVL tree is always O(Log (n)) where n is the number of nodes in the tree. We have used this tree to store Mutexes whenever a Mutex has been acquired. Every node in the tree is a structure that contains the Mutex pointer as well as resource ceiling of the Mutex, in addition to other details. Every time a node is inserted in the tree, the node with the minimum value (highest resource ceiling) becomes the system ceiling and every time a delete operation is performed, the system ceiling is re-calculated. AVL tree is more efficient for updating system cieling after every Mutex pend and post. This operation is more frequently performed, which is why AVL tree was chosen for this part.

This system ceiling is used to decide if the task that is ready to be released, should be inserted into the ready list(Binomial heap) or the blocked tree (2-3-4 tree), based o the tasks preemption threshold values that have been defined statically based on their deadline values (EDF)

D. 2-3-4 Tree

In place of this, we are currently using another instance of an AVL tree. This tree has been used to store the tasks that have been blocked from inserting into the binomial heap because of low preemption threshold. These tasks are inserted into the tree based on their preemption threshold. Multiple tasks with the same preemption threshold have been taken care of by using arrays to store the TCB pointers of the tasks. Every time a mutex has been unlocked, system ceiling is updated and based on this, all the blocked tasks with preemption threshold greater than the new system ceiling are released and inserted into the binomial heap. Using both of these data structures, it is made sure that EDF has been implemented correctly and SRP takes care of the resource contention amongst the tasks.

E. 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.

F. Implementation API details

1) Splay Tree APIs for storing Recursive tasks:

- DelRecTree() : Delete the required node from RecursionTree. **Time complexity = O(log n).**
- InsertRecTree(): Insert the required node to RecursionTree with updated TCB and release time elements. **Time complexity = O(log n)**.
- GetMinRecTree(): Get the minimum release time key node from RecursionTree. **Time complexity = O(1**) by maintaining pointer to the minimum.

2) Binomial Heap APIs for Ready List:

- DelRecTree() : Delete the required node from RecursionTree. **Time complexity = O(log n).**
- InsertRecTree(): Insert the required node to RecursionTree with updated TCB and release time elements. Time complexity = O(1).
- GetMinRecTree(): Get the minimum release time key node from RecursionTree. **Time complexity = O(1)** by maintaining pointer to the minimum.

3) AVL Tree APIs for determination of system cieling:

- DeleteMutex() : Delete the Mutex when it has been posted by a task. **Time complexity = O(log n)**.
- InsertMutex(): Insert an acquired Mutex with its resource cieling value. **Time complexity = O(log n).**
- MaxResCeil(): Get the minimum resouce cieling value (highest resource cieling) held by any Mutex at a particular instant of time to determine the system cieling. **Time complexity = O(1)** by maintaining pointer to the minimum.

4) 2-3-4 Tree APIs for storing Blocked tasks:

- Delblocktask() : Delete the task from the tree when its preemption threshold crosses the system cieling value. **Time complexity = O(log n)**.
- InsertBlkTask(): Insert a blocked task. **Time complexity = O(log n)**.
- MinTaskLevel(): Get the task with the highest preemption threshold (minimum value). **Time complexity = O(1)** by maintaining pointer to the minimum.

G. Test Cases and observations

H. Overhead value measurements and benchmarking

III. Key Features of the Implementation

- Our implementation takes care of synchronous release of tasks.
- Also, we have created our own Timer Task which updates the counter after every 1 sec instead of every 1 ms and is also re-configurable. This saves time consumed by context switching of existing ISR.
- By maintaining pointers to the minimum nodes in the data structures, we get a time complexity of O(1) in extraction of minimum value like minimum release time, earliest deadline, highest preemption threshold, etc.
- Runtime support for periodic and aperiodic tasks.
- Supports arbitrary period and arbitrary number of tasks.
- Support for arbitrary number of mutexes.

IV. Observations

We calculated few overhead values for benchmarking our implementation. The following fig 4, fig 5 and fig 6 show Overhead values for scheduling, releasing the tasks and Tick ISR interrupt latency.

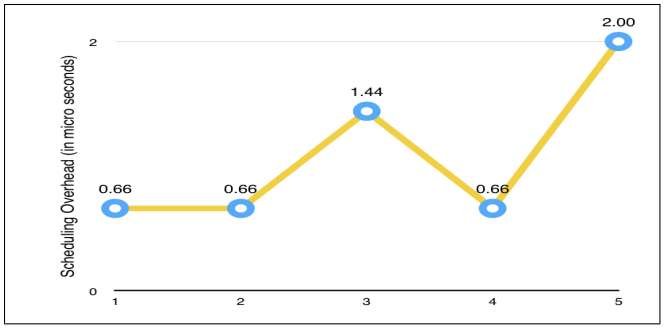


Fig. 4. scheduling overhead

Also, we compared number of context switches in execution of three tasks T(13,4,13), T2(10,2,10) and T3(8,1,8) where T1 and T2 are using resource R1 and T2 and T3 are using resource R2. The observations WERE TAKEN FOR BOTH PIP and SRP are shown in the image below.

V. Limitations

Managing Counter Overflow needs to be taken care of. The current implementation does not account for counter overflow management. Synchronous release of tasks is taken care of, by storing all the tasks' TCB pointers in an array, which is a static data structure. It can be made more dynamic by using data structures like Linked lists.

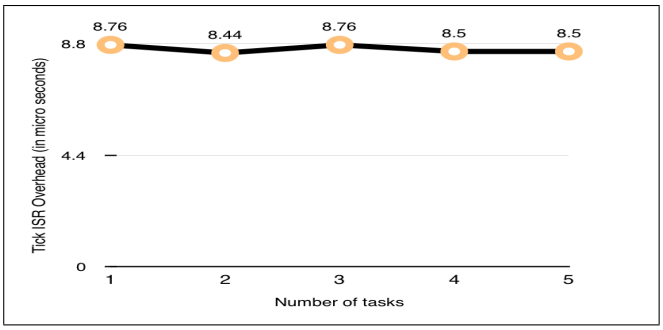


Fig. 5. Tick ISR overhead

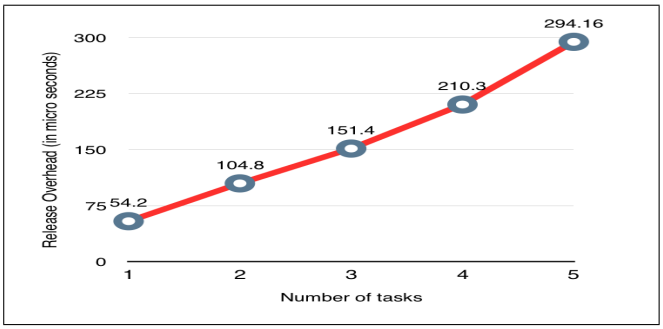


Fig. 6. release overhead

Number of context switches		
	PIP	SRP
TASK 1	4716	4708
TASK 2	2056	2056
TASK 3	1038	1030

Fig. 7. Context Switch - PIP and SRP